FASTPS: A Fast Publish-Subscribe service using RDMA

Arjun Balasubramanian Mohammed Danish Shaikh

University of Wisconsin - Madison

Abstract

Emerging applications such as Artificial Intelligence and Serverless Computing have been extremely popular over the past couple of years. These applications make use of several underlying sub-systems. The individual sub-systems need to be performant and scalable so as to meet requirements of these applications. Publish-Subscribe systems form an integral part of the pipeline that drives these emerging applications. Our analysis of workloads from these modern applications indicates that Publish-Subscribe systems do not offer the required performance and scalability desired by these applications.

We present FASTPS, a new ground-up system that provides better performance and scalability than existing Publish-Subscribe systems. FASTPS leverages the benefits of Remote Direct Memory Access (RDMA) networks to overcome the challenges imposed by emerging workloads. We evaluate a prototype implementation of FASTPS and show that FASTPS achieves good performance and overcomes the scalability bottlenecks suffered by current state-of-the-art Publish-Subscribe systems.

1 Introduction

Recent trends in computing point towards an increased adoption of artificial intelligence [56] and serverless computing [27]. Both of these domains rely on a bunch of sub-systems (hypervisors [14, 61], sandboxed execution environments [4–7], resource managers [2, 59, 60], schedulers [15, 29, 33, 40, 47, 53, 55], etc) to perform well and scale. One other such sub-system is the Publish-Subscribe system, where producers of data write to a service and clients can subscribe to receive the written data from the service. The Publish-Subscribe paradigm has been made exceedingly popular by a myriad of offerings by cloud providers [10, 12, 23, 35, 50].

While the emerging applications mentioned above heavily use Publish-Subscribe, it is unclear as to whether it meets the demands imposed by these new applications. These applications impose a bunch of new demands on Publish-Subscribe systems. As discussed in Section 2.2, we find that these applications require low latency and high throughput, support for a large number of subscribers/consumers of data, and support for *ephemeral* or short-lived data. Based on these new workload characteristics, we find that existing Publish-Subscribe systems do not scale well for these emerging applications (Section 2.3). For instance, we find that existing Publish-Subscribe systems cannot provide both high throughput and low latency simultaneously and inherently provide configurations to trade-off between the two. Similarly, existing Publish-Subscribe systems do not scale well for a large number of consumers of data. Our measurements indicate that the system gets bottlenecked by CPU when trying to serve a large number of consumers.

Remote Direct Memory Access (RDMA) has been successful in helping scale various applications including key-value stores [20, 30, 31, 42], graph processing systems [54, 63], deep learning [66], and distributed file systems [16, 24, 25, 65]. Due to the unique features offered by RDMA (Section 3), we believe that the problems imposed by emerging applications on Publish-Subscribe systems can be alleviated by deeply integrating RDMA into the design of Publish-Subscribe systems.

We present FASTPS, a new Publish-Subscribe system with RDMA at the heart of it. FASTPS uses RDMA as a fast transport conduit to meet the required throughput and latency goals. Additionally, FASTPS uses the right choice of RDMA primitives to overcome scalability issues both while producing and consuming data. In this paper, we outline a proposed design for FASTPS (Section 4) and describe details of the implementation we have so far in Section 5.

FASTPS uses a variety of techniques to overcome the bottlenecks in existing Publish-Subscribe systems. First, to overcome the CPU bottleneck imposed by multiple consumers, FASTPS uses one-sided remote reads that bypass the target CPU. This helps FASTPS scale to a large number of consumers. Second, for producing data, FASTPS leverages two-sided send/recv operations which incurs CPU involvement. The CPU involvement helps handle aspects such as replication for fault-tolerance as well as guaranteeing ordering semantics.

We build a prototype of FASTPS from scratch in C and evaluate its performance against Apache Kafka [35], an open-source Publish-Subscribe system (Section 6). The code is open-source and available at https://github.com/Arjunbala/RDMAPubSub. Our results show that FASTPS overcomes the challenges imposed by emerging application workloads and indicates that RDMA is a good fit for Publish-Subscribe systems. We outline the future scope of our work in Section 8.

2 Background and Motivation

We start by providing a primer on Publish-Subscribe (referred to as PUBSUB from here on) and Remote Direct Memory Access (RDMA). We then examine the requirements raised by emerging classes of applications such as Artificial Intelligence and Serverless Computing on PUBSUB systems. We then study the performance of Apache Kafka [35], an open-source and widely used PUBSUB system and show that workloads from emerging applications do not work well with existing PUBSUB architectures. This motivates a ground-up redesign of PUBSUB systems.

2.1 Background

2.1.1 Publish-Subscribe

A Publish-Subscribe paradigm is one where senders of messages, called *producers*, do not program the messages to be sent directly to specific receivers, called *consumers*, but instead categorize published messages into classes, called *topics* without knowledge of which consumers, if any, there may be. PUBSUB systems essentially act as the conduit between producers and consumers. They expose APIs for producers to push records to a specified topic and for consumers to consume records from a particular topic. We now describe the internal structure of PUBSUB systems. We choose Apache Kafka as a representative system to elucidate the design description that follows below.

Internally, a PUBSUB system represents each topic as an ordered append-only log of records. The log is replicated so as to provide fault-tolerance. Consensus algorithms [28, 36, 38, 46] are used to ensure that replicas are consistent with each other and to handle failures. Each topic is assigned a *broker*, which also happens to be the leader of the replicated log. Producers write records to the broker, while consumers also consume records from the broker instance. The PUBSUB system acknowledges writes to producers only once it has been durably committed at a majority of replicas. PUBSUB systems also generally expose a discovery service to allow clients (producers/consumers) to bootstrap the location of the broker. Additionally, for purposes of scalability, a topic may consists of multiple *partitions*, each having its own ordered log and broker. PUBSUB systems typically guarantee ordering within a partition but not across partitions.

2.1.2 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) provides applications with low latency access over a network. RDMA allows a node to perform one-sided read/write operations from/to memory on a remote node in addition to two-sided send/recv operations. Both user and kernel level applications can directly issue remote DMA requests (called *verbs*) on pre-registered memory regions (MRs). One-sided requests bypass CPU on the remote host, while two-sided requests require the CPU to handle them.

Software initiates RDMA requests by posting work queue entries (WQE) onto a pair of send/recv queues (a queue pair or "QP"), and polling for their completion from the completion queue (CQ). On completing a request, the RDMA NIC (RNIC) signals completion by posting a completion queue entry (CQE).

A send/recv operation requires both the sender and receiver to post requests to their respective send and receive queues that include the source and destination buffer addresses.

2.2 Requirements from emerging applications

The Publish-Subscribe paradigm is increasingly becoming a popular abstraction in a number of emerging applications. These applications make use of a myriad of PUBSUB systems [10, 12, 23, 35, 50].

Below, we list some of the emerging applications and how they use PUBSUB systems. We also highlight the requirements from PUBSUB raised by each use-case -

[A1] Serverless Computing. Function-as-a-service (FaaS) through offerings such as [11, 13, 22] is becoming a popular computation model among programmers. This is because with FaaS, programmers need to worry only about the programming logic and aspects such as scaling and resource management are handled by the serverless computing platform. Programmers frequently use PUBSUB systems to trigger the execution of dependent functions (also called *lambdas*) upon the execution of a lambda. For instance, one could have a workflow where one lambda handles the upload of an image to an S3 bucket. This lambda could in turn trigger the execution of another lambda that resizes the image that was uploaded in order to create a thumbnail. Similarly, frameworks such as PyWren [26] can use PUBSUB systems to trigger executions of successive stages of an execution DAG. We also believe that PUBSUB systems might be a viable way to exchange *ephemeral data* [34, 49] among these lambdas. These potential applications raise the following requirements from PUBSUB systems - (i) Lambdas are frequently used to serve user-facing applications and hence PUBSUB systems must be able to trigger execution of dependent lambdas or exchange ephemeral data at low latency. (ii) We also see that

there are scenarios where lambdas might serve background tasks such as image resize. Here, it is more important to be throughput oriented than latency sensitive and hence PUBSUB systems in this scenario must provide high throughput.

[A2] Scaling Artificial Intelligence. With deep-learning models becoming larger and computationally expensive, *distributed ML training* has become a popular technique to reduce the training time. A popular architecture for distributed training is the parameter server model [37] which consists of a *Parameter Server (PS)* that holds the learned model parameters and a bunch of *workers* that operate on portions of the data in parallel and periodically push local gradient updates back to the parameter server. The parameter server model frequently employs the *synchronous training* paradigm where the *PS* waits for each worker to complete one epoch of training and push its local gradients. Post this, each worker pulls the latest copy of the parameters before proceeding to the next training epoch. We can model this using the PUBSUB system paradigm where the *PS* produces a record containing the latest parameters and the workers consume this record to start the next epoch of training. This application raises the following requirements from PUBSUB systems - (i) There is usually a single *PS* task and multiple *worker* tasks. Hence, a PUBSUB system must be able to support a *fan-out* structure with a *single producer* and *multiple consumers*. (ii) In order to reduce the total training time, it is important that the records are delivered with *low latency*. Since a training epoch usually lasts for a significant amount of time, this application does not have a high throughput requirement from the PUBSUB system. Similar requirements exist for systems that support *reinforcement-learning* [43].

[A3] Change Data Capture (CDC). CDC is a design pattern that allows applications to observe the delta change in data from a data source. It is typically used to perform analytics over evolving data. As an example, CDC frameworks [41] use PUBSUB systems in order to stream updates from a MySQL database (by registering as a slave to listen to binary log events) to a stream processing engine [3, 9, 44, 58]. This application raises the following requirements from PUBSUB systems - (i) For analytics to be performed in real time, PUBSUB systems need to serve CDC records to stream processing engines at low latency. (ii) Database systems are typically throughput-oriented and hence will provide a large number of CDC records. This means that PUBSUB systems need to serve CDC records to stream processing engines at high throughput.

Based on the above, we summarize the requirements for PUBSUB systems raised by these new classes of applications as below -

[R1] Providing low latency and high throughput. Most applications require high throughput or low latency or even both of these simultaneously. Today, PUBSUB systems allow producers of data to control configurations to achieve these requirements. For instance, a Kafka producer exposes configurations such as *batch size and linger time*, where a larger batch size and longer linger time help in achieving higher throughput. However, the write throughput to a single Kafka partition is constrained by two factors - (i) Writes to disk. (ii) Establishing correct ordering for records within a partition. To overcome this bottleneck, application programmers can use multiple partitions. However, the drawback to this approach is that it is not possible to reason about the ordering of records across partitions. Hence, it is beneficial to application programmers if a PUBSUB system can provide both *higher read and write throughput* over a single partition.

[R2] Supporting heavy fan-out structure. Most applications have a single producer of data and multiple consumers of that data. Hence, it is important for any PUBSUB system to not become a performance bottleneck while serving multiple consumers. The general way this is achieved is by making different consumers read data records from different replicas, which effectively load balances consumers in order to prevent any single server from being bottlenecked on its CPU. However, when there is a *heavy fan-out structure*, CPU will easily become a bottleneck while serving consumers. One potential way to counteract this might be to increase the number of replicas, but this in turn would severely degrade the producer's write throughput. Hence, PUBSUB systems today have an inherent problem with this requirement.

[R3] Handling ephemeral data. Many modern applications do not have a hard requirement for data records to be persisted for extended periods of time. This is particularly true for *ephemeral or intermediate data* that is generated by serverless and big data frameworks. The useful lifetime for such data is usually just a few seconds. Such applications would be willing to *trade-off durability of data for better performance*. Hence, they can be sufficiently reliable with *in-memory replication*.

2.3 Apache Kafka Performance Analysis

We analyze the performance of Apache Kafka with respect to the aforementioned requirements. In this work, we primarily focus on requirement R1 and R2. To carry out this study, we setup a 10-node Apache Kafka cluster. In the Kafka configuration, we disable the batching of records, set the number of replicas to be 1, and write the ordered logs to tmpfs [8] so as to have in-memory logging of records. We produce a million records of 64 bytes each from a single producer and configure a variable number of consumers to each consume all of these records.

When a single producer and consumer are involved, we observe a producer throughput of *~15 Mbps* and a consumer throughput of *~20 Mbps*. Since these numbers are with no batching of records, it indicates that PUBSUB can in the best case offer micro-second level latency for both producing and consuming records. However, this comes at the cost of lower throughput. Higher throughput can be achieved by batching records, but this comes at the cost of inflated latency. Hence,

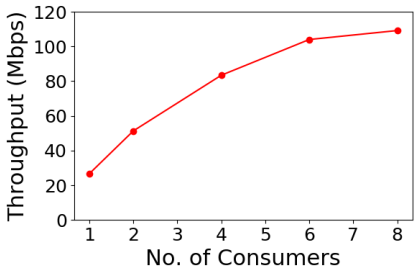

Figure 1: Variation of the cumulative throughput offered to Kafka consumers as the number of consumers increases

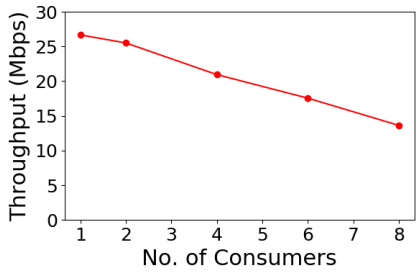

Figure 2: Variation of the throughput experienced by a single Kafka consumer as the number of competing consumers increase

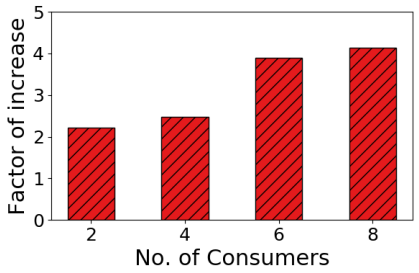

Figure 3: Factor of increase in CPU contention at the Kafka broker instance as the number of consumers increase

configuring Apache Kafka to simultaneously achieve low latency and high throughput is a challenge. This means that requirement R1 is hard to realize with Apache Kafka.

Next, we conduct a series of measurements to analyze requirement R2. Figure 1 shows the variation of the cumulative throughput across all consumers as we vary the number of consumers. We notice that the throughput initially increases at a near linear rate. However, we observe that throughput flattens as we approach 8 consumers. Further, from Figure 2, we observe that the throughput of a single consumer decreases as the number of competing consumers increase. Figure 3 shows that this problem exists due to increased CPU contention. These trends suggest that a *heavy fan-out* structure is a poor fit for existing PUBSUB systems and hence they fare poorly with respect to requirement R2.

3 RDMA as a fundamental building block

From Section 2.3, it is evident that existing PUBSUB systems do not meet the requirements imposed by emerging classes of applications. We believe that employing the use of RDMA networks can help remove the bottlenecks observed. We thus design FASTPS, a PUBSUB system that not only employs RDMA as a fast transport conduit, but also deeply integrates RDMA into its core design.

Below, we highlight some of the key features offered by RDMA and how we utilize them in the design of FASTPS -

[F1] RDMA one-sided reads and writes bypass the involvement of the remote CPU since an RDMA-capable NIC can directly issue DMA requests and read/write data from/to pinned memory regions. We thus leverage one-sided read operations to support *heavy fan-out structures*, ensuring that the server's CPU does not become a bottleneck even while it serves multiple consumers.

[F2] Inbound verbs, including recv and incoming one-sided read/write, incur lower overhead for the target, so a single node can handle many more inbound requests than it can initiate itself. This means that it is more efficient for consumers in FASTPS to poll for the availability of new records, rather than resort to notification-based mechanisms. We exploit this asymmetry to improve the scalability of the system.

[F3] For one-sided transfers, the receiver grants the sender access to a memory region through a shared, secret 32-bit "rkey". When the receiver RNIC processes an inbound one-sided request with a matching rkey, it issues DMAs directly to its local memory without notifying the CPU. FASTPS employs this feature to allow multiple consumers to consume data from a shared, remote memory region and thus avoids unnecessary copying of data.

4 System Design

This section presents the internal design of FASTPS. In Section 4.1, we discuss the architecture of FASTPS. We present the producer and consumer APIs in Sections 4.2.1 and 4.2.2 respectively. Finally, we describe the producer and consumer flows in Sections 4.3 and 4.4 respectively.

4.1 Architecture

FASTPS consists of *servers*, *producer library*, and *consumer library* components as described in the following subsections (refer Figure 4). Clients can utilize the producer and consumer library to produce and consume records respectively. Like Apache Kafka [35], we assume that a topic may consist of multiple partitions for purposes of scalability.

FASTPS consists of multiple servers that store the data produced by producer(s) in order to be consumed by consumer(s). Each server hold multiple individual buffers to serve producers and consumers interested in different topics. There are two types of buffers pertaining to a partition within a topic - a private *producer buffer* and a corresponding *consumer buffer*.

Producer buffer. This is a circular buffer that the *producer library* uses to perform RDMA writes consisting of the data to be written into FASTPS. This buffer is private to each producer in order to provide isolation from other producers.

Consumer buffer. This buffer keeps track of committed data after consensus. It also serves as a way for *consumer library* to

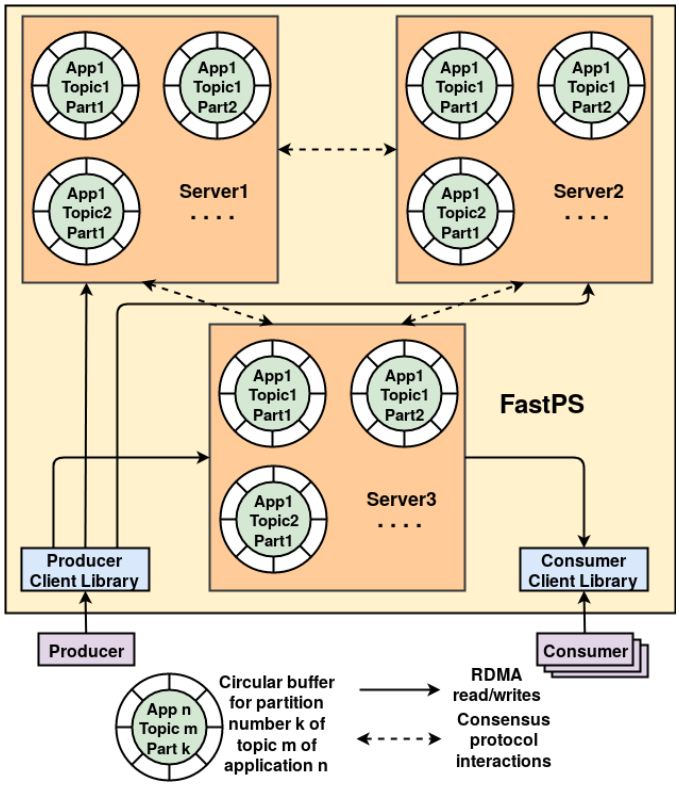

Figure 4: FastPS Architecture

know about the availability of committed data to be consumed by consumer(s).

An advantage of using RDMA is that both producer and consumer buffers can be spread across the entire cluster without incurring too much performance degradation. This makes aspects like load balancing easier to handle. This is similar in spirit to [45].

4.2 APIs

4.2.1 Producer library APIs

The producer library provides the following APIs to the producers:

InitProducer(application name, topic, number of partitions): When a producer wants to start producing data for a new *topic*, it has to call into the *InitProducer* API provided by the producer library and provide the *application name*, name of the *topic* and *number of partitions* to be created for the topic. The producer library thereafter informs a subset of *servers* to initialize their respective data structures. This subset can be decided so as to load balance the usage of servers in the cluster. The selected *servers* register a new, private memory region for the producer, initialize the data structures that will be used to hold the produced data, and finally return the starting address of the memory region that the producer library can perform RDMA writes into.

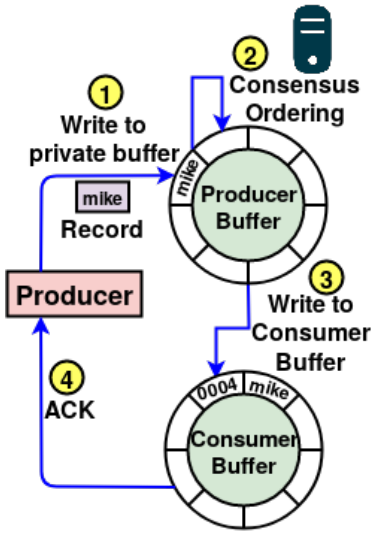

Figure 5: Producer Flow

Produce(topic, partition, data): When a producer wants to produce *data* for a given *partition* in a *topic*, it calls into the *Produce* API provided by the producer library. We outline the producer flow in Section 4.3.

4.2.2 Consumer library APIs

The consumer library provides the following APIs to the consumers:

InitConsumer(topic): When a consumer wants to starts consuming data from a given *topic*, it has to call into the *InitConsumer* API provided by the consumer library and provide the name of the *topic* it wants to subscribe to. The consumer library thereafter starts polling on *any server's consumer buffer* for the given *topic*.

Consume(): When a consumer wants to consume *data* produced in a given *partition* of a *topic*, it calls into the *Consume* API provided by the consumer library. We outline the consumer flow in Section 4.4.

4.3 Producer Flow

As depicted in Figure 5, the following sequence of events happen when a producer wants to produce a record:

1. The producer writes the record into a FastPS private producer buffer using *IBV_WR_RDMA_WRITE_WITH_IMM* RDMA verb. The *IBV_WR_RDMA_WRITE_WITH_IMM* verb provides the capability to write data to a remote memory region while also notifying the remote CPU about this operation with inlined *immediate data*. We write the producer record into the remote memory and specify the length of the producer record in the immediate data.

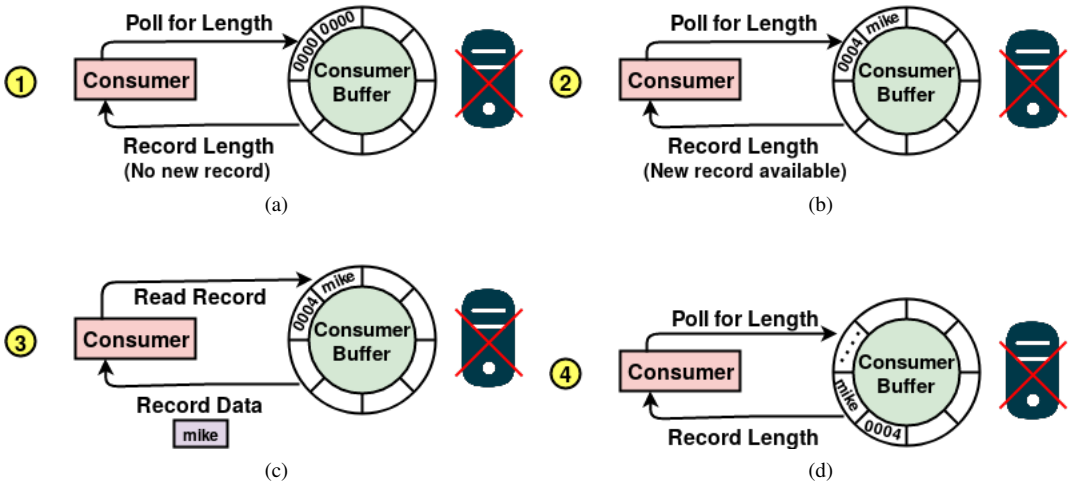

Figure 6: FASTPS *consumer flow (a) Polling for length (b) Reading length (c) Reading record (d) Polling for length*

2. The FASTPS server performs consensus and ordering of the written record. Note that this step requires involvement of CPU.

3. The FASTPS server writes the record into corresponding *consumer buffers*. For a local write, this would simply involve copying the data. Remote writes can be done using one-sided RDMA writes.

4. The FASTPS server sends an acknowledgement back to the producer.

4.4 Consumer Flow

As shown in Figure 6, the following sequence of events happen when a consumer wants to consumer a record:

1. The consumer polls at the current position of the consumer buffer in FASTPS server till it encounters a non-zero length using the *IBV_WR_RDMA_READ* RDMA verb.

2. When a record is available, the consumer reads the records length (lets say L) from FASTPS server and increments the remote pointer to point to the data. This can be done deterministically since a fixed number of bits are used to represent the length, which both the producer and consumer are aware of.

3. The consumer reads the L bits of record data from the FASTPS server.

4. The consumer increments the pointer pointing to the consumer buffer of FASTPS server by L bits and starts polling for the length of the next record.

5 Implementation

We implemented a simplified version of FASTPS wherein only a single topic with a single partition is supported. Hence, we did not implement replication, consensus, or failure handling. Our implementation uses the *rdma_cma* [51] which provides library functions for establishing communication over RDMA. It works in conjunction with the verbs API that is provided by the *libibverbs* [52] library. The *libibverbs* library provides the underlying interfaces needed to send and receive data. FASTPS has been implemented in *1090* lines of C code spread across *server*, *producer client* and *consumer client* components as described in the following subsections. Each component implements an asynchronous event driven communication interface on top of the *rdma_cma* library.

5.1 Server

As described in Section 4, a FASTPS server maintains private *producer buffer* and corresponding *consumer buffer* for a *topic*. Subsequently, producers and consumers interact with the FASTPS server using two-sided and one-sided RDMA operations in order to produce and consume data records respectively.

5.2 Producer Client

The producer client implements the APIs described in Section 4.2.1. On receiving an *Init* API call, the producer client registers a private *producer buffer* with a FASTPS server for the mentioned *topic*. Subsequently, on receiving a *Produce* API call, the producer client writes the record into the buffer registered with the FASTPS server. Two-sided send/recv RDMA operations are used by the producer client.

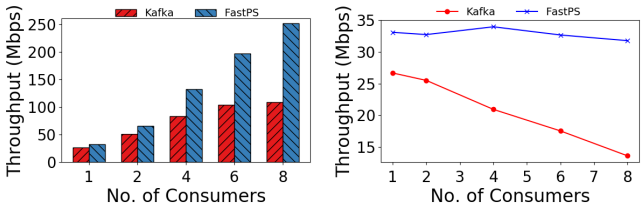

Figure 7: Macrobenchmarks comparing the performance of FASTPS *with Apache Kafka in terms of (a) Cumulative throughput across all consumers as the number of consumers vary (b) Throughput of a single consumer as the number of competing consumers vary.*

5.3 Consumer Client

The consumer client implements the APIs described in Section 4.2.2. On receiving an *Init* API call, the consumer client retrieves the *rkey* and *remote address* to the *consumer buffer* (refer Section 4) corresponding to the topic mentioned in the call. This design choice follows from feature *F3* mentioned in Section 3. Subsequently, on receiving a *Consume* API call, the consumer client starts polling at the *remote address* to detect when a data record is written. This design choice follows from the feature *F2* mentioned in Section 3. The consumer client uses one-sided RDMA read operations for reading data remotely from the FASTPS server.

6 Evaluation

We evaluate FASTPS on a 10-machine cluster deployed on CloudLab [19] and compare the performance of FASTPS against Apache Kafka. We also carry out micro-benchmarks to show the performance impact of varying record sizes.

6.1 Experimental Setup

Testbed. Our testbed has 10 machines, each having 16 cores and 64GB memory. The machines are connected through Mellanox SX6036G switches via a 40Gbps uplink. All of the machines use Ubuntu 16.04 and Mellanox OFED version 4.6.

Baseline Stack. We compare the performance of FASTPS against Apache Kafka. To enable a fair comparison with our initial implementation of FASTPS, we configure Kafka to disable the batching of records, disable log replication by setting the number of replicas to be 1, and record logs in tmpfs so as to store data in-memory.

Metrics. We measure *throughput* as the total amount of data produced/consumed divided by the total time taken. We measure *latency* as the total amount of time taken to produce/consume the record.

6.2 Macrobenchmarks

In this section, we compare the performance of FASTPS against Apache Kafka. For each system, we first produce 1 million records, where each record has a size of 64 bytes. We then configure a varying number of consumers to consume these records simultaneously.

Producer Throughput. From our measurements, we notice that FASTPS offers a producer throughput of ~16 Mbps, which is about 1 Mbps greater than the producer throughput offered by Apache Kafka. In the producer flow, both FASTPS and Apache Kafka incur CPU involvement and hence we do not see much of a performance difference. The slight increase in performance observed in FASTPS can be attributed to the fact that the TCP/IP stack in the kernel is bypassed.

Consumer Throughput. We configure a variable number of consumers to simultaneously consume the produced records. We then measure two quantities - (i) *Cumulative Throughput*, which is the sum of throughputs across all consumers (ii) *Single Consumer Throughput*, which is the throughput observed by a single chosen consumer. We confirm using the *top* command that the FASTPS server CPU is not involved in the consumer flow. Figure 7a compares FASTPS and Apache Kafka on cumulative throughput as the number of consumers vary. We observe that FASTPS offers near linear increase in cumulative throughput while it saturates for Apache Kafka. Figure 7b compares FASTPS and Apache Kafka on the throughput of a single consumer. We observe that in FASTPS, a consumer has nearly constant throughput irrespective of the number of competing consumers. However, in Apache Kafka, a single consumer's throughput nearly drops by half as we vary the number of consumers from 1 to 8. Both of these trends indicate that FASTPS offers better scalability for heavy fan-out structures with a large number of consumers.

6.3 Microbenchmarks

We perform a micro-benchmark to evaluate the performance of FASTPS as the record size changes. In this experiment, we consider a single producer producing 1 million records of each presented size and a single consumer consuming these records. Figure 8 shows the variation in producer throughput as record size changes. We see an increase in producer throughput as the record size increases and it caps off at ~2 Gbps as the record size becomes greater than 50KB. We did not investigate the reason for this, but we believe that it could be due to the fact that multiple PCIe transactions are required for transferring larger data over the PCIe channel between the RNIC and the memory subsystem. Figure 9 and Figure 10 capture the throughput and latency for consuming records when a single consumer is involved. We notice an increase in throughput which caps at ~8Gbps. We believe that throughput in this case gets limited by the PCIe channel, but we have not verified this. Latency starts increasing as the record size exceeds 50KB.

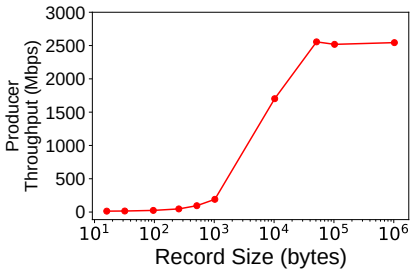

Figure 8: Variation of the producer throughput as the size of the produced record changes

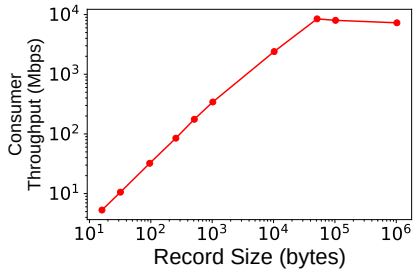

Figure 9: Variation of the consumer throughput as the size of the consumed record changes

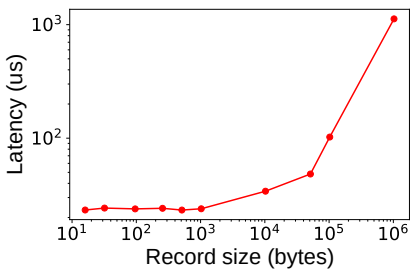

Figure 10: Variation of latency for consuming a record as the size of the consumer record changes

We believe that this may be due to multiple PCIe transactions being involved as in the producer case.

7 Related Work

FASTPS builds upon a rich line of work that investigates how to efficiently leverage the advantages offered by RDMA into system design to support various applications. However, to our knowledge, no prior work has looked at the requirements for PUBSUB systems imposed by emerging applications, benchmarked the performance of existing PUBSUB systems for these workloads, and deeply integrated RDMA into PUBSUB system design.

Key-Value Stores. FARM [20] shows how RDMA can be used to build a key-value store that offers high throughput and low latency. HERD [30] focuses its design on reducing network round trip time while using efficient RDMA primitives. Interestingly, the paper shows that single-RTT designs with server CPU involvement can outperform single-sided operations when they require multiple round-trips. We leverage similar observations in FASTPS to design the producer flow. Our design philosophy is very similar to PILAF [42], where *get* operations are served using one-sided RDMA operations, but *put* operations involve the CPU in order to synchronize memory access. Similarly, [31] studies the right choice of RDMA primitives for use in key-value stores.

Distributed File Systems. Several distributed file systems use RDMA as a drop-in replacement of traditional networking protocols [16, 24, 25, 65]. OCTOPUS [39] and ORION [67] leverage RDMA to build high-performance distributed file systems for NVMM-based storage.

Transactions and Database Management. There has been a wide variety of work that utilize RDMA features to speed up transaction processing [17, 18, 21, 32, 57, 68]. DRTM+H [64] performs a phase-by-phase analysis of optimistic concurrency control and identifies the right RDMA primitive for each phase. Active-Memory replication [69] is a technique that leverages RDMA to eliminate computational redundancy during replication in database systems.

Consensus using RDMA. FASTPS currently does not implement replication. However, to support replication, FASTPS would need to design an efficient replicated state machine mechanism to replicate log records. DARE [48] designs a replicated state-machine that efficiently utilizes RDMA primitives to build a strongly-consistent key-value store. APUS [62] shows how to scale Paxos using RDMA.

8 Future Work

Our current prototype assumes a single topic and single partition as a proof-of-concept. We wish to build a full-fledged Publish-Subscribe system that supports multiple topics and partitions, along with features like configurable replication, batching, and failure handling. This would require us to research on what RDMA primitives need to be used to efficiently implement consensus protocols used in systems like Kafka. Additionally, we would like to explore different data structures for the consumer buffer so as to possibly reduce the number of network round-trips to 1 for consuming records and also explore if any optimizations are feasible for handling small/large records. Finally, we wish to integrate FASTPS with systems for emerging applications such as Ray [43] and OpenWhisk [1], and evaluate the performance implications.

9 Conclusion

We show that emerging applications such as serverless computing and scaling artificial intelligence require PUBSUB systems to provide low latency and high throughput, support heavy fan-out structure, and handle ephemeral data. Further, we use Apache Kafka as a case study to delineate that current PUBSUB systems do not meet the requirements imposed by these emerging classes of applications. We propose FASTPS and implement a simple prototype to show that RDMA can be used as a fundamental building block to address the bottlenecks observed with current PUBSUB systems. The results from our simple prototype show promise and we believe that FASTPS has the potential to serve as an ideal PUBSUB system

for meeting the requirements of emerging applications.

10 Acknowledgements

We would like to thank Professor Michael Swift for giving us the opportunity to work on this project. His advice during office hours and classroom lectures helped us in successfully completing this project. We would also like to thank our advisor Aditya Akella for his valuable comments and insights.

References

[1] IBM Bluemix Openwhisk. https://www.ibm.com/cloud-computing/bluemix/openwhisk, 2017.

[2] Apache Hadoop Submarine. https://hadoop.apache.org/submarine/, 2019.

[3] Apache Spark Streaming. https://spark.apache.org/streaming/, 2019.

[4] Docker. https://www.docker.com/, 2019.

[5] Firecracker MicroVM. https://firecracker-microvm.github.io/, 2019.

[6] gVisor. https://github.com/google/gvisor, 2019.

[7] Kata Containers. https://katacontainers.io/, 2019.

[8] tmpfs - A virtual memory filesystem. http://man7.org/linux/man-pages/man5/tmpfs.5.html, 2019.

[9] Apache Flink. https://flink.apache.org/, 2019.

[10] AWS Kinesis. https://aws.amazon.com/kinesis/, 2019.

[11] AWS Lambda. https://aws.amazon.com/lambda/, 2019.

[12] AWS Simple Queuing Service. https://aws.amazon.com/sqs/, 2019.

[13] Azure Functions. https://azure.microsoft.com/en-us/services/functions/, 2019.

[14] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 164–177.

[15] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 285–300.

[16] Callaghan, B., Lingutla-Raj, T., Chiu, A., Staubach, P., and Asad, O. Nfs over rdma. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications* (New York, NY, USA, 2003), NICELI '03, ACM, pp. 196–208.

[17] Chen, H., Chen, R., Wei, X., Shi, J., Chen, Y., Wang, Z., Zang, B., and Guan, H. Fast in-memory transaction processing using rdma and htm. *ACM Trans. Comput. Syst. 35*, 1 (July 2017), 3:1–3:37.

[18] Chen, Y., Wei, X., Shi, J., Chen, R., and Chen, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 26:1–26:17.

[19] CloudLab. https://www.cloudlab.us/, 2019.

[20] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 401–414.

[21] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 54–70.

[22] Google Cloud Functions. https://cloud.google.com/functions/, 2019.

[23] Google Pub/Sub. https://cloud.google.com/pubsub/docs/, 2019.

[24] Guz, Z., Li, H. H., Shayesteh, A., and Balakrishnan, V. Performance characterization of nvme-over-fabrics storage disaggregation. *ACM Trans. Storage 14*, 4 (Dec. 2018), 31:1–31:18.

[25] Islam, N. S., Wasi-ur Rahman, M., Lu, X., and Panda, D. K. High performance design for hdfs with byte-addressability of nvm and rdma. In *Proceedings of the 2016 International Conference on Supercomputing*

(New York, NY, USA, 2016), ICS '16, ACM, pp. 8:1–8:14.

[26] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: Distributed computing for the 99 In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 445–451.

[27] JONAS, E., SCHLEIER-SMITH, J., SREEKANTI, V., TSAI, C.-C., KHANDELWAL, A., PU, Q., SHANKAR, V., CARREIRA, J., KRAUTH, K., YADWADKAR, N., GONZALEZ, J. E., POPA, R. A., STOICA, I., AND PATTERSON, D. A. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[28] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks* (Washington, DC, USA, 2011), DSN '11, IEEE Computer Society, pp. 245–256.

[29] KAFFES, K., YADWADKAR, N. J., AND KOZYRAKIS, C. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2019), SoCC '19, ACM, pp. 158–164.

[30] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.

[31] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 437–450.

[32] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 185–201.

[33] KARANASOS, K., RAO, S., CURINO, C., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G. M., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2015), USENIX ATC '15, USENIX Association, pp. 485–497.

[34] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 427–444.

[35] KREPS, J. Kafka : a distributed messaging system for log processing.

[36] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[37] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 583–598.

[38] LIN, W., YANG, M., ZHANG, L., AND ZHOU, L. Pacifica: Replication in log-based distributed storage systems. Tech. Rep. MSR-TR-2008-25, February 2008.

[39] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 773–785.

[40] MAHAJAN, K., BALASUBRAMANIAN, A., SINGHVI, A., VENKATARAMAN, S., AKELLA, A., PHANISHAYEE, A., AND CHAWLA, S. Themis: Fair and efficient gpu cluster scheduling, 2019.

[41] Maxwellś Daemon. https://maxwells-daemon.io/, 2019.

[42] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 103–114.

[43] MORITZ, P., NISHIHARA, R., WANG, S., TUMANOV, A., LIAW, R., LIANG, E., ELIBOL, M., YANG, Z., PAUL, W., JORDAN, M. I., AND STOICA, I. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 561–577.

[44] NOGHABI, S. A., PARAMASIVAM, K., PAN, Y., RAMESH, N., BRINGHURST, J., GUPTA, I., AND CAMPBELL, R. H. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow. 10*, 12 (Aug. 2017), 1634–1645.

[45] NOVAKOVIC, S., DAGLIS, A., BUGNION, E., FALSAFI, B., AND GROT, B. The case for rackout: Scalable data serving using rack-scale systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 182–195.

[46] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320.

[47] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 69–84.

[48] POKE, M., AND HOEFLER, T. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2015), HPDC '15, ACM, pp. 107–118.

[49] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 193–206.

[50] RabbitMQ. https://www.rabbitmq.com/, 2019.

[51] RDMA CMA library. https://linux.die.net/man/7/rdma_cm, 2019.

[52] RDMA Core Codebase. https://github.com/linux-rdma/rdma-core, 2019.

[53] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 351–364.

[54] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 317–332.

[55] SINGHVI, A., HOUCK, K., BALASUBRAMANIAN, A., SHAIKH, M. D., VENKATARAMAN, S., AND AKELLA, A. Archipelago: A scalable low-latency serverless platform, 2019.

[56] Stanford Artificial Intelligence Index. https://hai.stanford.edu/sites/g/files/sbiybj10986/f/ai_index_2019_report.pdf, 2019.

[57] TALEB, Y., STUTSMAN, R., ANTONIU, G., AND CORTES, T. Tailwind: Fast and atomic rdma-based replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 851–863.

[58] TOSHNIWAL, A., TANEJA, S., SHUKLA, A., RAMASAMY, K., PATEL, J. M., KULKARNI, S., JACKSON, J., GADE, K., FU, M., DONHAM, J., BHAGAT, N., MITTAL, S., AND RYABOY, D. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 147–156.

[59] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O'MALLEY, O., RADIA, S., REED, B., AND BALDESCHWIELER, E. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16.

[60] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 18:1–18:17.

[61] WALDSPURGER, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194.

[62] WANG, C., JIANG, J., CHEN, X., YI, N., AND CUI, H. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 94–107.

[63] WANG, S., LOU, C., CHEN, R., AND CHEN, H. Fast and concurrent rdf queries using rdma-assisted gpu graph exploration. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2018), USENIX ATC '18, USENIX Association, pp. 651–664.

[64] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 233–251.

[65] WU, J., WYCKOFF, P., AND PANDA, D. K. PVFS over infiniband: Design and performance evaluation. In *32nd International Conference on Parallel Processing (ICPP 2003), 6-9 October 2003, Kaohsiung, Taiwan* (2003), pp. 125–132.

[66] XUE, J., MIAO, Y., CHEN, C., WU, M., ZHANG, L., AND ZHOU, L. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, ACM, pp. 44:1–44:14.

[67] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 221–234.

[68] ZAMANIAN, E., BINNIG, C., HARRIS, T., AND KRASKA, T. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow. 10*, 6 (Feb. 2017), 685–696.

[69] ZAMANIAN, E., YU, X., STONEBRAKER, M., AND KRASKA, T. Rethinking database high availability with rdma networks. *Proc. VLDB Endow. 12*, 11 (July 2019), 1637–1650.