RocketBufs: A Framework for Building Efficient, In-Memory, Message-Oriented Middleware

Huy Hoang, Benjamin Cassell, Tim Brecht, Samer Al-Kiswany
Cheriton School of Computer Science, University of Waterloo

ABSTRACT

As companies increasingly deploy message-oriented middleware (MOM) systems in mission-critical components of their infrastructures and services, the demand for improved performance and functionality has accelerated the rate at which new systems are being developed. Unfortunately, existing MOM systems are not designed to take advantages of techniques for high-performance data center communication (e.g., RDMA). In this paper, we describe the design and implementation of RocketBufs, a framework which provides infrastructure for building high-performance, in-memory Message-Oriented Middleware (MOM) applications. RocketBufs provides memory-based *buffer* abstractions and APIs, which are designed to work efficiently with different transport protocols. Applications implemented using RocketBufs manage buffer data using input (`rIn`) and output (`rOut`) classes, while the framework is responsible for transmitting, receiving and synchronizing buffer access.

We use our current implementation, that supports both TCP and RDMA, to demonstrate the utility and evaluate the performance of RocketBufs by using it to implement a publish/subscribe message queuing system called RBMQ and a live streaming video application. When comparing RBMQ against two widely-used, industry-grade MOM systems, namely RabbitMQ and Redis, our evaluations show that when using TCP, RBMQ achieves broker messaging throughput up to 1.9 times higher than RabbitMQ and roughly on par with that of Redis, when configured comparably. However, RBMQ subscribers require significantly less CPU resources than those using Redis, allowing those resources to be used for other purposes like processing application data. When configured to use RDMA, RBMQ provides throughput up to 3.7 times higher than RabbitMQ and up to 1.7 times higher than Redis. We also demonstrate the flexibility of RocketBufs by implementing a live streaming video service and show that it can increase the number of simultaneous viewers by up to 55%.

CCS CONCEPTS

• **Networks → Programming interfaces**; • **Computing methodologies** → *Distributed computing methodologies*; • **Information systems → Data management systems**.

DEBS '20, July 13–17, 2020, Virtual Event, QC, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8028-7/20/07…$15.00
https://doi.org/10.1145/3401025.3401744

KEYWORDS

message-oriented middleware, in-memory, event-based systems, message queuing, publish subscribe, live streaming video

ACM Reference Format:
Huy Hoang, Benjamin Cassell, Tim Brecht, Samer Al-Kiswany. 2020. RocketBufs: A Framework for Building Efficient, In-Memory, Message-Oriented Middleware. In *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20), July 13–17, 2020, Virtual Event, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3401025.3401744

1 INTRODUCTION

Message-Oriented Middleware (MOM) systems, also often referred to as publish/subscribe, message-queuing, or event-based systems, are a popular class of software designed to support loosely-coupled messaging in modern distributed applications. Examples of applications and services that utilize MOM systems include IBM's cloud functions [29], the Apache OpenWhisk serverless framework [10], the Hyperledger blockchain framework [7] and streaming media systems [60]. These applications all follow a produce-disseminate-consume (PDC) design pattern, where one or more producers (or publishers) send data as messages to an MOM substrate (often comprised of message brokers) for dissemination to a possibly large number of consumers (or subscribers). To scale to high loads some MOM systems support the distribution of messages to other brokers using topologies best suited to the application.

Many modern applications have high performance and scalability demands with regard to message delivery. Facebook's pub/sub system, for example, delivers over 35 Gigabytes per second within their event processing pipeline [63], while Twitch, a live streaming video service, handles hundreds of billions of minutes worth of video content a year [22]. Other classes of applications, such as online gaming [23] and stock trading [67], require messages to be delivered with extremely low latency. As a result, constructing high-performance and scalable MOM systems to deal with emerging workloads is of great interest in both industry and academia. Over the past five years alone, many new open-source MOM systems have been developed [9, 11, 20, 38, 61, 68], and cloud providers have continued to introduce new MOM services as part of their infrastructure [3, 4, 24, 25].

Often in MOM deployments, data is moved between hosts that reside within a data center [39, 62]. One approach to building high-performance MOM systems for such environments is to leverage data center networking capabilities, especially those that offer kernel-bypass features to enable high-throughput and/or low-latency communication such as Remote Direct Memory Access (RDMA) [44] or emerging technologies, accelerators and APIs such as DPDK [69], F-Stack [21], and Solarflare/TCPDirect [64]. Such technologies are becoming increasingly important to reduce message dissemination costs to the rapidly growing numbers of brokers and subscribers required by modern applications. Unfortunately, commonly-used MOM systems do not take advantage of these capabilities and instead use kernel-based TCP, which in many cases incurs protocol processing and copying overhead (even for communication within a data center [27]), limiting throughput and resulting in higher latency.

One reason current MOM systems avoid using emerging technologies is the complexity of supporting multiple communication substrates in a single MOM implementation. For instance, the native RDMA verb interface uses abstractions and APIs fundamentally different from the *socket* abstraction that are quite complicated to use [1]. As a result, significant engineering effort would be required for both new and existing MOM systems to support RDMA and TCP, since two separate data transfer implementations would be required. Similarly, emerging technologies often initially provide new abstractions and custom APIs that require additional modifications to the MOM system implementation (e.g., DPDK [69]).

We believe it is important for MOM systems to utilize RDMA and other existing and emerging technologies designed for high-throughput and/or low-latency data center communication while not being forced to implement code specific to each technology. To this end, we propose RocketBufs, a framework to facilitate the easy construction of high-performance in-memory Message-Oriented Middleware systems. RocketBufs provides a natural memory-based *buffer* abstraction. Application developers control the transmission and reception of data using input (`rIn`) and output (`rOut`) classes that are associated with buffers. RocketBufs implements and handles the transfers of buffered data between communicating nodes and provides mechanisms for flow control. This design allows flexible topologies to be constructed for scaling MOM systems. RocketBufs' APIs are agnostic to the underlying transport layer and developers can configure the framework to use different protocols or technologies without changing application code.

Figure 1 shows that applications can be built using RocketBufs which provides `rIn` and `rOut` objects. RocketBufs relies on its networking layer (RocketNet) for communication. In our prototype, RocketNet supports TCP and RDMA. In the future, support can be added to RocketNet for new data center networking technologies and all applications using RocketBufs can reap the benefits of those technologies by simply changing a configuration file.

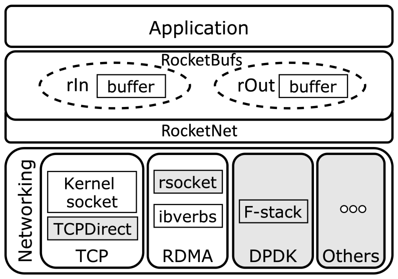


Figure 1: RocketBufs: shaded areas denote future work

Given the continuous development of new MOM systems, we believe that such a framework would greatly simplify their construction while allowing them to achieve high performance. As a proof of concept for this approach, we describe our prototype implementation that includes support for TCP and RDMA. By designing abstractions and APIs that work well with both RDMA and TCP, two protocols with vastly different programming interfaces, we believe that other transport layer APIs and technologies like QUIC [31], DPDK [69], F-Stack [21], and Solarflare/TCPDirect [64] could also be efficiently supported by the framework, providing benefits to all applications built using RocketBufs. This paper makes the following contributions:

- We describe RocketBufs, a framework that facilitates the construction of scalable, high-performance Message-Oriented Middleware systems.
- We describe a prototype implementation of RBMQ, a publish/subscribe messaging system built on top of RocketBufs. We evaluate the performance of RBMQ by comparing it against RabbitMQ and Redis, two widely-used, industry-grade MOM systems.
- We demonstrate the flexibility of RocketBufs by using it to build a second, quite different, application to support the replication and delivery of live streaming video. Our empirical evaluation shows that RocketBufs is able to support up to 27% and 55% higher simultaneous viewer throughput than Redis when using TCP and RDMA, respectively.

2 RELATED WORK

Networking Technologies and Libraries: The desire to provide efficient access to networking resources is long-standing concern and several technologies and libraries have been created for this reason. In order to provide the best performance possible these technologies are often accompanied with new abstractions and APIs. In order to provide some benefits to legacy applications new libraries are often created on top of the "native" APIs that attempt to mimic socket abstractions and APIs. Remote Direct Memory Access (RDMA) is one such example with the native verbs interface and the rsocket library that provides a socket abstraction on top of RDMA. Unfortunately, rsocket's protocol translation and copying overheads make it inefficient compared to native RDMA [37, 74] and as a result our implementation uses native RDMA APIs. Another library, libfabric [26] provides abstracted access to networking resources, enabling applications to discover which communication mechanisms they have access to (including RDMA), form connections between nodes, transfer data, and monitor transfer completions. The key differences between the systems described above and RocketBufs is that RocketBufs provides middleware with much higher level APIs and abstractions that are designed to naturally and explicitly support the building of a wide range of efficient message-oriented systems. Those systems would be better suited for use in RocketBufs networking layer (RocketNet) upon which RocketBufs is built.

Similar to RDMA, emerging networking technologies such as DPDK [69], F-Stack [21] and Solarflare/TCPDirect [64] bypass the kernel for higher performance and offer low level communication APIs, that are much lower-level than RocketBufs. Applications hoping to leverage these technologies need to manually create, maintain, and make use of networking mechanisms. We believe that any of these libraries could be used by RocketBufs' networking component (described in Section 3.5). One of the key goals of

RocketBufs is to provide higher level abstractions that allow MOM developers to concentrate on application logic while RocketBufs implements and handles data transfers.

RDMA-based Systems and Optimizations: It is well-known that RDMA is complicated to use [1, 35], and there has been a good deal of research into improving RDMA usability and performance. For example, Zhang et al. [76] propose a new high level abstraction for developing applications that bypass the kernel including RDMA-based applications, and Remote Regions [1] provides a file-based interface for interacting with remote memory. Likewise, LITE [71] provides a user space RDMA abstraction at the cost of giving up local kernel bypass, while RaMP [45] provides application-directed, RDMA-backed memory region sharing. Derecho [32] is a system designed primarily for building data center applications that require strongly consistent replicated objects. Because its protocols have been optimized for efficient use with TCP and RDMA they demonstrate better performance than other competing systems like APUS, LibPaxos, and ZooKeeper. Derecho does not offer the abstractions and APIs that RocketBufs does which are explicitly designed to simplify the implementation of efficient MOM systems. Multiple systems also explore RDMA within the context of distributed memory and distributed storage, including, Pilaf [46], FaRM [18, 19], HERD [34], Nessie [15], and many others [33, 36, 40, 48, 70, 72, 73]. RocketBufs is oriented around ease-of-use and efficiency for building Message-Oriented Middleware. Furthermore, RocketBufs is network protocol agnostic, and allows efficient data transfer regardless of the availability of RDMA.

Live Streaming Video: Several attempts have been made to improve the efficiency of live streaming video delivery, but mostly through means that are complimentary to our own. For example, Pires et al. [52] examine strategies for selecting efficient bit rate for transcoding to reduce resource consumption. Netflix, which does not serve live streaming video, has contributed to this space in a way, by exploring how to efficiently provide in-kernel TLS encryption to improve the scalability of web servers delivering video content [65, 66].

Our design of RocketBufs and use in a live-streaming video application is motivated by our previous work on RocketStreams [14], which uses TCP or RDMA to disseminate live streaming video. In this paper, we design more general abstractions for use in a wider range of applications (e.g., the scalable dissemination of data in MOM and live streaming video systems). RocketBufs, provides generic support for: producers, brokers, and consumers; the creation of general topologies; flow control; and synchronization control.

3 ROCKETBUFS

We have three main goals in mind when designing the framework. First, we want to provide natural abstractions and easy-to-use APIs which are suitable for developing a variety of MOM systems. A key property of MOM systems is the independence between communicating parties (i.e., publishers and subscribers) in the systems. Therefore with RocketBufs, publishers in the system should be able to continuously produce messages without being blocked while previously produced messages are being delivered. Analogously, subscribers should be able to subscribe to and continuously receive new data. Secondly, the framework's interface should be agnostic to the underlying transport protocol. The application should be able to switch transport protocols by simply changing the system's configuration without any modifications to the application code. Finally, the framework should also enable efficient MOM system implementations using a variety of networking technologies. As a proof of concept we focus on two very different APIs, sockets (for TCP) and by taking full advantage of native RDMA APIs.

To achieve these goals, RocketBufs uses an event-driven, memory-based interface that is strongly influenced by the requirement to work well with the RDMA-based transport layer, but also allows the TCP-based transport layer to be efficiently implemented. RocketBufs is a user-space C++ library with an object-oriented programming model. A typical RocketBufs application initializes framework objects and uses them to communicate with other processes using communication units called *buffers*. In designing the APIs, we recognize that data movement from publishers to brokers and from brokers to subscribers is analogous to a producer-consumer pattern (where brokers "consume" data from publishers while they also "produce" data to subscribers). RocketBufs provides two main classes: `rIn` and `rOut`. The `rOut` class is used by producers to create buffers and disseminate data, while the `rIn` class is used by consumers to subscribe to and receive data. Message brokers use both `rIn` and `rOut` objects for data ingestion and dissemination. The framework also provides support for buffer splicing (discussed in Section 3.4) to allow message brokers to efficiently forward data. Figure 2 shows an example of how these objects could be used in a MOM system. Using `rIn` and `rOut` objects, applications can implement flexible topologies and partitioning logic for scaling out the MOM system. We now describe the design and implementation of our framework. More details and discussion concerning design decisions can be found in Huy Hoang's thesis [28].

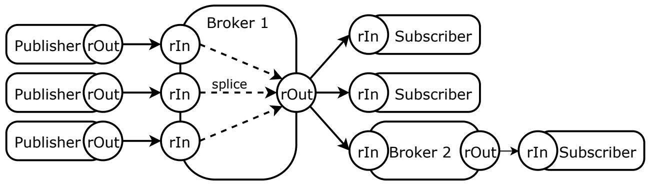


Figure 2: Use of `rIn` and `rOut` objects in MOM system.

3.1 Buffers in RocketBufs

A buffer is a region of memory used by RocketBufs for storing incoming and outgoing data. Buffers are encapsulated in `rIn`/`rOut` objects. Each buffer has a 32-bit identifier (defined using the `bid_t` data type) and is implemented as an in-memory circular buffer, as depicted in (Figure 3). Buffers have memory semantics and applications interact directly with buffers through byte ranges. While this abstraction is lower-level than a message-based abstraction, it gives the application complete control over how messages are formed in the buffer. Data in a buffer represents messages in a FIFO message queue. Circular buffers are well known, widely used, data structures that are particularly well suited to producer-consumer types of problems. As a result, they have been used in a wide variety of contexts including operating systems and communication systems. We also utilize them in RocketBufs because they provide a bounded space for communication between `rIn` and `rOut` objects (necessary for RDMA) and are useful for implementing buffer flow control (see Section 3.3).

When a buffer is created, its memory is allocated from framework-managed memory pools. This is done in order to reduce the consumption of RDMA-enabled NIC (called an RNIC) cache resources when RDMA is used [18]. Each buffer has a *free* region and a *busy* region. For an `rOut` buffer (which we refer to as an output buffer), the free region represents the space into which the application can produce data, while the busy region stores the data that has been produced and is waiting to be transferred. To produce messages, an application requests a free *segment* of the buffer, places the message in the segment, and signals the framework to deliver it. At that point the segment is marked busy and queued for delivery by the framework. Analogously, for an `rIn` buffer (input buffer) incoming data is received into the free region and data in the busy region is consumed and processed by the application.

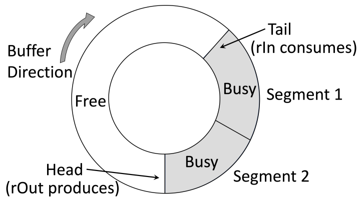


Figure 3: RocketBufs' circular buffer.

The boundaries of a segment are defined using an `iovec` structure which includes a pointer to the starting address of the segment and its size. Because a memory segment could begin near the end of the buffer and wrap around to the beginning, applications are required to work with memory segments that may not be contiguous. For this reason RocketBufs defines a structure `buf_seg` (Listing 1) to allow applications to directly access buffer memory. The `vecs` field is an array of `iovecs` that contains either one or two `iovecs`, indicated in the `vec_count` field.

Listing 1: Definition of `buf_seg`

```
struct buf_seg {
  struct iovec* vecs; // Array of iovecs.
  int vec_count; // Number of iovecs (at most 2).
};
```

3.2 `rIn` and `rOut` Classes

RocketBufs provides two main classes, `rIn` and `rOut`. Applications create and configure connections between `rIn` and `rOut` instances using APIs provided by the framework. One `rOut` object can be connected to one or more `rIn` object(s). However, one `rIn` object can only be connected to a single `rOut` object. This relationship is modeled after the publish/subscribe pattern, where a publisher broadcasts data to one or more subscriber(s). If a subscriber application needs to subscribe to data from multiple sources, it can do so by using multiple `rIn` instances. After initializing `rIn` and `rOut` instances, applications can use the methods exposed by these classes, shown in Listing 2, to manipulate buffers and manage data transfers. RocketBufs' key APIs are asynchronous and completion events related to buffers are handled by registering callback functions. The asynchronous API design serves two purposes. First, it allows for continuous data production and consumption by the application. Secondly, it allows for natural and efficient implementations for RDMA and TCP.

Listing 2: Key `rIn`/`rOut` methods

```
// rOut methods
void create_buffer(bid_t bid, size_t size);
buf_seg get_output_segment(bid_t bid, size_t size, bool blocking);
void deliver(buf_seg &segs, size_t size, void *ctx);
void set_data_out_cb(void (*cb)(void *));
void splice(rIn &in, bid_t buf_id);

// rIn methods
void subscribe(bid_t bid, size_t size);
void set_data_in_cb(void (*cb)(bid_t, buf_seg));
void data_consumed(bid_t bid, size_t size);
```

Before being able to send and receive messages, applications need to create and subscribe to buffers. An `rOut` object creates a buffer by calling `rOut::create_buffer` and providing the buffer identifier and the buffer size. An `rIn` object calls `rIn::subscribe` with a specified buffer identifier to register for data updates from that buffer. We refer to such registrations as *subscriptions*. Buffer identifiers are required to be unique per `rIn`/`rOut` object and the framework performs checks for identifier uniqueness when a buffer is created. The framework assumes that buffer identifiers are chosen and managed by the application. In a real deployment, these identifiers could be mapped to higher-level communication units (such as topics in a topic-based MOM system) using other software (e.g., a name service), if required. When `rIn::subscribe` is called, the framework implicitly allocates memory for the buffer and shares the local buffer's metadata with the connected `rOut` object. The metadata of a buffer includes the buffer's address and size. When RDMA is used, it also includes the remote key of the buffer's RDMA-registered memory region. For each subscription, the `rOut` object maintains the remote buffer's metadata, along with a queue data structure that tracks data pending delivery to that buffer. This allows the application to continue producing data into that buffer while previously-produced data is being transferred.

An application that is transmitting, adds data to an output buffer in two steps. First, the `rOut::get_output_segment` method is called. This requests an available memory segment from the specified output buffer. The application provides the buffer identifier and a size as arguments and this call returns a `buf_seg` structure which represents the next available (free) memory segment into which the application can place output data. If the call succeeds, the total size of the returned segment equals the requested amount and the application then assumes ownership of and control over the segment. When appropriate, the application informs the framework that the segments are ready-for-delivery by calling `rOut::deliver`, providing the `buf_seg` and a `size` as arguments. The `size` argument specifies the amount of data to be transferred, which may be less than or equal to the amount requested in `rOut::get_output_segment`. Upon calling `rOut::deliver`, the framework places the reference to the segment in the appropriate subscription queues and notifies the framework worker threads that there is data to transfer. The worker threads then transfer the data. This workflow is depicted in Figure 4.

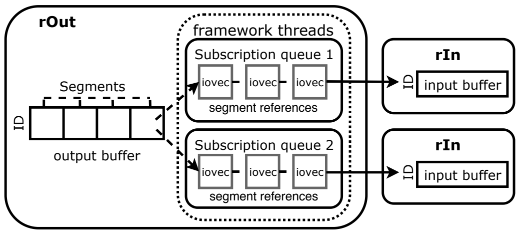


Figure 4: `rOut` buffer management and data transfers.

Once the segment has reached all subscribers, the framework notes that the corresponding buffer segment is free and will reuse the space to produce more data. The `rOut::deliver` method is asynchronous and may return before the segment is delivered to all subscribers. If an application needs to be notified upon completion of a transfer operation (e.g., for monitoring delivery), it can register a callback function using `rOut::set_data_out_cb`. RocketBufs does not provide any batching mechanisms by default. However, applications can easily implement batching by requesting large memory segments that can hold multiple messages and delaying the call to `rOut::deliver` until the point at which enough messages have been placed in the segment.

Applications using an `rIn` object can use `rIn::set_data_in_cb` to register a callback function that executes when new data arrives. When new data arrives, the framework executes this callback with two arguments: the identifier of the buffer to which new data is delivered, and a reference (`buf_seg`) to the data itself. The `rIn` class also provides an `rIn::get_buf_ref` which returns a reference to all the received data of a buffer. This allows the application to access the data in the input buffer without copying.

3.3 Buffer Flow Control

An important consideration for RocketBufs is how to make buffer space available for use and re-use for disseminated data. Typically in MOM systems, messages are delivered to subscribers using a FIFO message queue, and subscribers consume messages in the order they are sent. Therefore with RocketBufs, we allow the subscriber application to signal to the framework that it can reclaim the oldest segment of a specific buffer when it has consumed the data and no longer need that segment. The `rIn` class provides the `rIn::data_consumed` method (line 11 of Listing 2) for this purpose. When this method is called, RocketBufs implicitly synchronizes the buffer state (the free and busy portion) between the `rIn` and `rOut` objects. In the future, we plan to expand this functionality to allow applications to selectively discard messages (by specifying the `buf_seg`), signaling that that buffer space can be reused while keeping older messages for application access.

In real deployments, subscribers might process messages at a slower rate than they are produced, resulting in a large amount of data being queued at the message brokers. This scenario is commonly referred to as the creation of *back-pressure*. There are different ways that MOM systems handle back-pressure. RabbitMQ, for example, implements a credit-based flow control mechanism that slows down the rate of data production [55]. Redis, on the other hand, does not have such a mechanism and simply closes the connection to a slow subscriber if it cannot keep up [58]. Redis' approach avoids the overhead (e.g., message exchanges) required for flow control, making it more efficient when there is no back-pressure (as will be demonstrated in Section 4.2). However, it also results in possible data losses when back-pressure occurs.

With RocketBufs, the circular buffer design allows credit-based flow control to be naturally implemented at the buffer level. When a remote buffer does not have enough free space, the `rOut` object pauses the transfer of data to that buffer. If the discrepancy between the rate of data production and consumption continues, eventually, the output buffer will become too busy and a call to `rOut::get_output_segment` will fail (`vec_count` is set to `-1` in the returned value). In this case, the application using the `rOut` object is expected to suspend the production of data and retry later. Alternatively, applications can explicitly set the `blocking` flag when calling `rOut::get_output_segment`. This blocks the calling thread until the requested amount of memory is available.

RocketBufs offers the option of disabling flow control on a per-buffer basis. When flow control is disabled, the framework does not block data transfers and overwrites old segments in the circular buffer. While this option is not suitable for many MOM systems, it is useful for certain applications where data is usually produced and consumed at the same rate (e.g., live video streaming).

3.4 Buffer Splicing

RocketBufs' `rIn` and `rOut` classes allow for the construction of flexible MOM topologies. With RocketBufs, a message broker uses the `rIn` class to ingest data, either from publishers or other message brokers, and transfers ingested data to other nodes (which could be subscribers or other message brokers) using the `rOut` class. One way to implement a message broker is to copy ingest data from the `rIn` instance's buffers to the `rOut` instance's buffers and then signal the framework to transfer that data. However, this approach incurs overhead while copying data from input to output buffers.

To avoid this copying overhead, RocketBufs supports buffer splicing using the `rOut::splice` method. This method takes a buffer identifier and a reference to an `rIn` instance as arguments. When `rOut::splice` is used, the `rIn` instance shares the input buffer memory with the `rOut` instance. When data is received, the corresponding buffer segment is automatically added to the appropriate subscription queue managed by the `rOut` instance, allowing the data to be disseminated without additional copying. The `rOut::splice` method is useful when no modification of ingest data is required. This is especially efficient when both ingestion and dissemination can be done within a data center using RDMA.

3.5 The RocketNet Networking Layer

A key to RocketBufs' design is that it encapsulates the networking layer code (RocketNet) to easily support the addition of new methods and/or APIs for transmitting and receiving buffer data. RocketBufs uses dedicated worker threads to carry out networking operations and execute application callbacks. In this section, we describe how these threads perform TCP and RDMA-based networking, however RocketBufs' abstractions also provide for the future addition other types of transport protocols.

The TCP implementation uses an event-driven model. Worker threads manage non-blocking I/O on TCP connections to remote nodes and react to I/O events using `epoll`. In the current prototype, application data is sent from `rOut` to `rIn` objects following 64 bits of metadata: 32 bits for the buffer identifier and 32 bits for the data size. This implies that a single transfer operation is limited to 4 GB in size, however this is larger than the typical upper limit for RDMA operations [44] and is in line with many common messaging protocols [6, 12, 54].

RocketNet provides RDMA support to achieve better performance when RNICs are available. Our RDMA implementation uses Reliable Connections (RC) for communication, a choice strongly advocated for in recent research [48]. The framework's worker threads are used for posting work requests to RDMA queues and handling completion events. RocketNet maintains a completion queue for each RNIC and uses a thread to monitor each queue. When a completion event occurs, the blocked monitoring thread is unblocked which then passes the event to one of the worker threads, which in turn handles that event and executes any necessary callbacks. RDMA data transfers are performed using zero-copy `write-with-immediate` verbs directly from `rOut` output buffers into `rIn` input buffers, bypassing both nodes' operating systems and CPUs. An `rOut` object uses the metadata of the remote buffer to calculate the remote addresses for these operations. The framework stores the buffer identifier in the verbs' immediate data field. This is used to trigger completion events on the receiving end, as RDMA verbs are otherwise invisible to the CPU. To receive data, an `rIn` object posts work requests to the receive queue and waits on the completion queue for incoming data notifications. For small messages, RocketNet uses RDMA inlining to reduce latency [35, 43].

Now that we have implemented asynchronous communication using two drastically different transport protocols we do not expect it to be difficult to support other data center communication technologies (e.g., DPDK or TCPDirect). The underlying APIs have been defined and the necessary infrastructure is in place.

3.6 Configuration and Optimizations

We now discuss some configuration parameters that impact RocketBufs' messaging performance. These parameters are exposed by RocketBufs in several forms: as arguments when creating `rIn` and `rOut` objects, as part of a configuration file, or as arguments to various RocketBufs methods.

Buffer sizes control how much data can be placed in buffers for dissemination and consumption. The size of a buffer is set when it is created (either by using the `rOut::create_buffer` or the `rIn::subscribe` method). Typically, a larger buffer allows for more messages to be queued and delivered without blocking the application (while waiting for buffer space to be freed), resulting in higher message throughput. However, this comes at the cost of higher memory consumption. Therefore, buffer sizes should be carefully configured on systems with memory constraints. Additionally, for applications where subscribers require disseminated data to be maintained for a period of time (e.g., a streaming video application which needs to maintain video data for serving viewers), the buffer size should be set large enough to hold this data while allowing new data to arrive.

A key challenge when optimizing RocketBufs' performance is to fully utilize the CPUs of multi-core systems. Applications using TCP can set the `tcp_num_threads` parameter to control the number of threads that handle TCP sockets. Setting this to the number of CPU cores in the system allows CPU resources to be fully utilized. Additionally, the `tcp_thread_pinning` option signals the framework to set the affinity of each TCP worker thread to a specific CPU core. This ensures that these threads are load-balanced across CPU cores, as otherwise the Linux kernel tends to schedule more than one active thread on the same core [42].

When RDMA is used, RocketBufs maintains multiple threads to monitor the RDMA completion queues (one per RNIC). However, if those same threads were used to handle completion events, they could become a bottleneck when handling a high rate of messages (and therefore a high rate of completion events). As a result, in our implementation, the monitoring threads distribute the completion events among a configurable number of framework-managed worker threads, which then handle these events. The `rdma_num_threads` parameter controls the number of RDMA worker threads created by the framework per RNIC. This parameter should be tuned based on the number of RNICs and CPU cores being used on the host. For example, on a system with two RNICs and eight CPU cores, setting `rdma_num_threads` to four would allow all CPU cores to be utilized for event handling. When this parameter is set to zero, no worker thread is created and RDMA completion events are handled by the monitoring threads.

3.7 Fault Tolerance Semantics

Existing message oriented middleware systems provide a range of fault tolerance semantics. Some do not tolerate node failures [49], others use replication to tolerate a broker failure [54], while others copy messages to disk and can therefore tolerate broker and complete cluster failures [39]. Because fault tolerance techniques present a tradeoff between reliability and performance, some modern systems, such as ActiveMQ [9], even offer a range of configurable fault tolerance options. In order to support applications that may have a wide range of requirements for fault tolerance semantics, RocketBufs does not impose any fault tolerance protocols or semantics upon applications. While RocketBufs uses reliable message transport services (e.g., TCP and RDMA reliable connections) to transfer data, application-level code would be required to detect issues like node failures and/or network partitions. This allows applications to react to failures according to its desired semantics which may potentially vary with the cause of the failure.

4 RBMQ PUBLISH/SUBSCRIBE APPLICATION

To demonstrate RocketBufs' utility and measure its messaging performance, we use it to build an in-memory, topic-based publish/subscribe system which we call RBMQ (RocketBufs Message Queue).

4.1 RBMQ Design and Implementation

The components of RBMQ (i.e., the publishers, broker, and subscribers) are implemented following the design depicted in Figure 2. The publishers use the `rOut` class to send messages belonging to specified topics to the broker. The subscribers use the `rIn` class to subscribe to and receive messages from specific topics. Each topic is mapped to a separate buffer. In our prototype, the topic-to-buffer mapping is implemented using a hash table, although in a production setting, this mapping could be coordinated using a distributed key-value store, or other equivalent control schemes.

To send messages belonging to a topic, a publisher must first initialize the topic by sending a control message to the broker. This message contains the topic name and the corresponding buffer identifier. Upon this notification, the broker calls `rIn::subscribe` to ingest data from the buffer. The publisher then can send messages by following the data production steps outlined in Section 3.2. Note that, data placement into the output buffer can be done in any manner preferred by the publisher application, such as copying from other memory locations or reading the data from a device.

The RBMQ message broker is responsible for routing messages from publishers to subscribers. It also acts as the main contact point for the system, which listens to incoming connections from publishers and subscribers. For each publisher, the broker creates an `rIn` instance to ingest messages from that publisher. A single `rOut` instance is used to disseminate messages to the subscribers. Because ingested messages do not need to be modified, the RBMQ broker uses buffer splicing to efficiently forward messages from publishers to subscribers.

An RBMQ subscriber receives messages by connecting to the broker and subscribing to the corresponding buffers of certain topics using the `rIn` class. Inside the callback function registered using `rIn::set_data_in_cb`, the subscriber processes the messages in the FIFO message queue and informs the framework of the amount of data consumed. In our current prototype, for evaluation purpose, the subscriber does not perform any processing and simply notifies the framework that the message is consumed.

To illustrate the ease with which RocketBufs can be used to construct a broker, Listing 3 shows the pseudocode for key components of an RBMQ broker. A `Listener` instance is maintained to listen for and manage connection requests from publishers and subscribers. When connection requests arrive (line 14), the broker uses them to initialize the appropriate `rIn`/`rOut` objects. If a connection request arrives from a publisher (line 15), the broker creates an `rIn` instance and sets up callback functions to handle application data and control messages. Note that the `on_ctrl_msg` function is bound to the `rIn` instance (line 18), so that the broker can identify the source of the control messages when they arrive. If the connection request comes from a subscriber, it is added to the `rOut` instance so that the subscriber can receive published messages (line 24).

When a broker receives a topic-creation control message from a publisher (line 29), it calls `rIn::subscribe` to receive messages from the corresponding buffer. The `rOut::splice` method is also called to forward the messages from that buffer to the subscribers. Additionally, when a message arrives, the broker verifies that it has the correct buffer-to-topic mapping. This is done in the `publisher_data_cb` function, which is registered as the `rIn`'s callback function.

Listing 3 shows that the RocketBufs' code to implement a broker is relatively small and straightforward, especially compared to code that would use TCP and/or RDMA directly. The code for producers and consumers is even more concise and we believe RocketBufs provides an easy to use foundation for implementing MOM systems.

Listing 3: Pseudocode for key RBMQ broker components.

```
// Listen for connections (from publishers and subscribers)      1
Listener listener(broker_address, protocol);                     2
listener.set_conn_cb(on_conn);                                   3
rOut out; // Dissemination object                                4
                                                                 5
// Callback for data ingest (from publisher)                     6
void publisher_data_cb(bid_t bid, buf_seg data) {                7
  if (!verify_topic_mapping(bid, data))                          8
    throw exception("InvalidMapping");                           9
  // Splice prevents the need for other logic here              10
}                                                               11
                                                                12
// Callback for handling new connections                        13
void on_conn(Conn &conn) {                                      14
  if (from_publisher(conn)) {                                   15
    rIn *in = new rIn(conn);                                    16
    // Set callback to handle control messages                  17
    in.set_control_msg_cb(std::bind(on_ctrl_msg, in));          18
    // Set callback to handle application data                  19
    in.set_data_arrived_cb(publisher_data_cb);                  20
  }                                                             21
  // Add subscriber connections                                 22
  if (from_subscriber(conn))                                    23
    out.add_conn(conn);                                         24
}                                                               25
                                                                26
// Callback for handling control messages                       27
void on_ctrl_msg(rIn *in, iovec msg) {                          28
  if (is_topic_creation(msg)) {                                 29
    bid_t bid = bid_from_msg(msg);                              30
    in.subscribe(bid, buffer_size); // Subscribe to publisher   31
    out.splice(*in, bid); // Deliver data to subscribers        32
  }                                                             33
}                                                               34
```

4.2 RBMQ Evaluation Methodology

In our experiments, we use one host to run the message broker processes, and separate hosts to run the publisher and subscriber processes. Each subscriber subscribes to all topics and the message brokers disseminate messages to all subscribers. For RBMQ, buffer sizes and threading parameters are tuned for individual experiments. Generally, the buffer size is set to be at least 10 times as large as the size of a message (including message headers), which allows for continuous message production. We also set the `TCP_NODELAY` option for TCP sockets, where they are used (i.e., for RabbitMQ, Redis and the RBMQ configurations that use TCP). We measure the performance of different RBMQ transport protocol configurations: publisher-to-broker and broker-to-subscriber over TCP (denoted as *RBMQ-tcp-tcp*); publisher-to-broker over TCP and broker-to-subscriber over RDMA (*RBMQ-tcp-rdma*); and publisher-to-broker and broker-to-subscriber over RDMA (*RBMQ-rdma-rdma*).

For RabbitMQ, we use the *rabbitmq-c* client library [8] to implement the publishers and subscribers. Our Redis publishers and subscribers are implemented using the Redis-provided client library *hiredis* [56]. We run multiple Redis broker processes on the broker host in order to utilize all CPU cores, since each Redis broker process is single-threaded. For both Redis and RabbitMQ, we disable data persistence and event logging features to ensure that messages are handled in-memory only. We also tune both systems based on recommended best practices [53, 59] to optimize their messaging performance in our experiments.

The message broker processes run on a host containing a 2.6 GHz Intel Xeon E5-2660v3 CPU with 10 cores, 512 GB of RAM, and four 40 Gbps NICs for a total of 160 Gbps bidirectional bandwidth (we refer to this hardware as a "big host"). Subscribers and publishers run on separate hosts which contain a single 2.0 GHz Intel Xeon D-1540 CPU with eight cores, 64 GB of RAM, and a single 40 Gbps NIC (we refer to this hardware as a "regular host"). We benchmarked our NICs using iPerf [30] and found that the maximum throughput they can achieve is 38 Gbps. All nodes run Ubuntu 18.04.2 with a version 4.18.0 Linux kernel. To avoid network contention, each subscriber connects to a separate NIC on the host running the broker. Each experiment is run for 150 seconds, with data being collected during the 120 second steady-state period following a 30 second warmup period. The experimental results are reported with 95% confidence intervals, except for the latency experiments where the data points are reported as CDFs. Note that, the confidence intervals are typically small relative to the sizes of the data points, and therefore in most cases they are not visible on the graphs.

4.3 RBMQ Message Throughput

In this experiment, we measure the maximum message throughput that can be achieved by a message broker in terms of number of messages delivered per second (mps). To find the maximum throughput values, we configure the publishers to constantly send messages and increase the number of publishers until the throughput no longer increases. This corresponds to the point where, depending on the experiment, either the broker host's CPU utilization is nearly 100% or the NIC is saturated.

We run a series of experiments and record the results for varying message sizes and study the scalability of RocketBufs and RBMQ the number of subscribers. Figures 5a, 5c, and 5e show the broker message throughput (in mps) when disseminating to zero, two and four subscribers, and Figures 5b, 5d, and 5f show the application-level goodput (in Gbps) for the same experiments.

In the zero-subscriber benchmarks (Figure 5a and 5b), messages published to the broker are not forwarded to any subscriber and are discarded immediately. These benchmarks allow us to obtain insights about the ingest capacity of the broker and the cost of disseminating data to subscribers. In these results, among TCP-based systems, Redis performs better than RBMQ-tcp and RabbitMQ (ingest throughput is up to 23% and 36% higher, respectively). This can be explained by the fact that Redis does not implement a flow control scheme (as discussed in Section 3.3), allowing it to avoid flow control overhead. RBMQ-rdma-rdma achieves the highest ingest throughput, which is up to 1.7 times higher than Redis, due to the CPU-efficiency of RDMA. Additionally, the ingest throughput for RBMQ-rdma-rdma remains relatively consistent for all message sizes before the bandwidth-saturation point (up to 2 KB). The throughput when ingesting 2 KB messages is only 5% lower than when ingesting 8-byte messages. In contrast, before saturating the NIC bandwidth, the ingest throughput drops 27% for Redis, 27% for RBMQ-tcp and 76% for RabbitMQ.

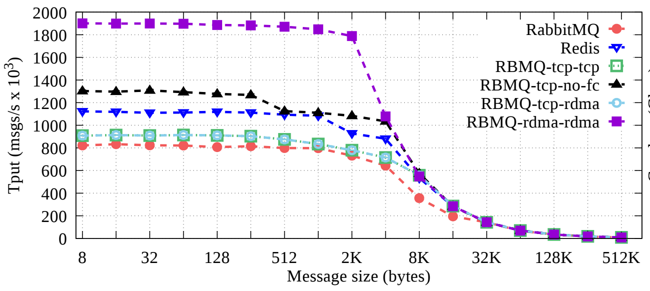
System	**Rabbit-MQ**	**Redis**	**RBMQ-tcp**	**RBMQ-rdma**	**RBMQ-no-fc**	**RBMQ-rdma-rdma**
utilization	94.0	55.3	63.0	52.3	29.2	9.6
vmlinux	23.6	74.2	85.2	84.9	78.2	73.7
mlx4	2.1	10.6	7.4	4.8	5.4	4.9
libpthread	0.7	1.0	2.9	4.3	6.7	9.9
application	71.5	8.7	3.3	4.0	6.2	6.5
others	2.2	5.5	1.3	2.1	3.4	5.0

Table 1: CPU utilization statistics from RBMQ broker. Load is 50,000 mps with 8 KB messages and four subscribers.

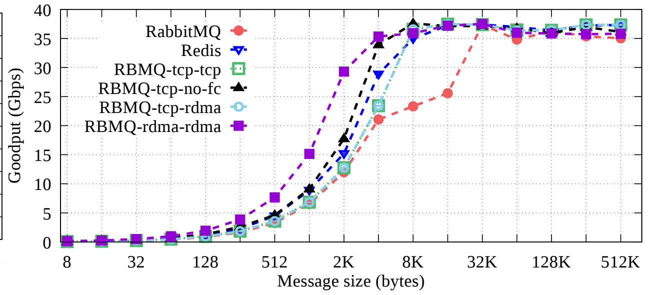
When subscribers are present, we make several observations from the experimental results. First, among TCP-based systems, Redis and RBMQ perform significantly better than RabbitMQ. In the one-subscriber case, RabbitMQ is the only system that is not able to saturate the NICs' bandwidth (due to CPU saturation). When disseminating messages to four subscribers, the message throughput of RBMQ-tcp and Redis is up to 1.9 times and 2.2 times higher than RabbitMQ, respectively. Secondly, Redis performs slightly better than RBMQ-tcp. In the four-subscriber case, Redis' throughput is up to 18% higher than RBMQ-tcp. Finally, leveraging RDMA with RBMQ yields substantial throughput improvements. Even when RDMA is only used for broker-to-subscriber communication (RBMQ-tcp-rdma), throughput with four subscribers is up to 1.3 times higher than Redis. When RDMA is used for all communication (RBMQ-rdma-rdma), message throughput with four subscribers is up to 2.0 times higher than RBMQ-tcp and 1.7 times higher than Redis. Additionally, RBMQ-rdma-rdma's throughput with four subscribers is higher than Redis' throughput with two subscribers, and RBMQ-rdma-rdma is also the only system that is able to fully saturate all 40 Gbps NICs.

We perform CPU profiling on the broker host using `perf` [41] to understand the differences in performance among the systems evaluated. Table 1 shows the analyzed profiling data, where each pub/sub system is subjected to a load of 50,000 messages per second with 8 KB messages and four subscribers. The "utilization" row shows the average system-wide CPU utilization of each system. The subsequent rows show the percentages of CPU time each system spends in various parts of the system, including the Linux kernel (vmlinux), the NIC driver (mlx4), the threading library (libpthread), the user-level application functions (application), and others.

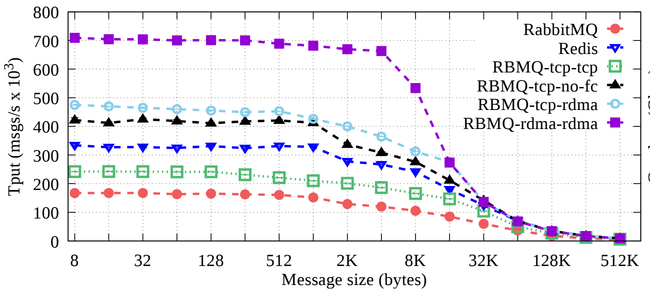
The profiling statistics clearly shows that systems requiring less CPU for the target load (50,000 mps) are able to obtain higher peak loads. For example, RBMQ-rdma-rdma requires the least amount of CPU (9.6%) while handling this load, due to the CPU-efficiency of RDMA, and as a result it achieves the highest maximum throughput with higher loads. RabbitMQ spends much more CPU time executing application-level code (71.5%) compared to Redis and RBMQ, which in all cases is less than 10%. This high application-level overhead reduces the CPU time available for transferring data, and is likely due to the overhead from the Erlang implementation (e.g., from just-in-time compilation and scheduling). In contrast, RBMQ and Redis spend the majority of their CPU time (more than 70%) in the Linux kernel, indicating that a larger portion of CPU time is used to transfer data, allowing these systems to scale to provide higher message throughput.



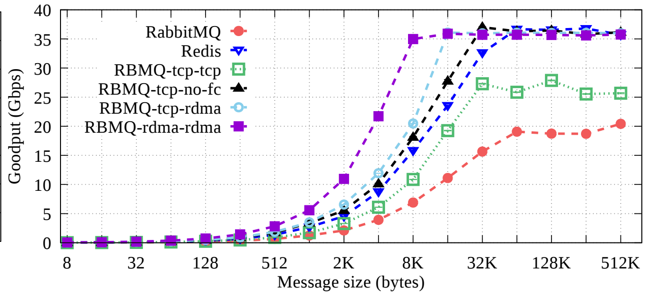
(a) RBMQ throughput with 0 subscribers



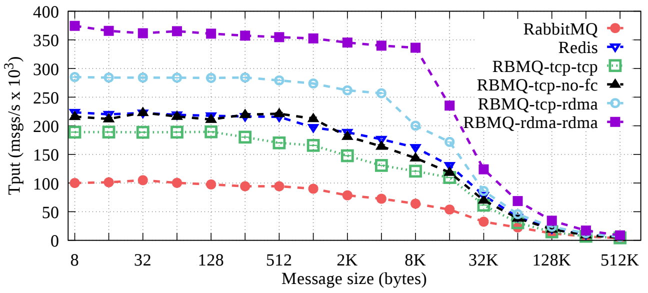
(b) RBMQ goodput with 0 subscribers



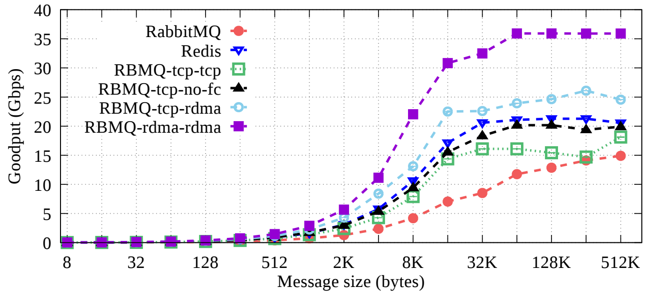
(c) RBMQ throughput with 2 subscribers



(d) RBMQ goodput with 2 subscribers



(e) RBMQ throughput with 4 subscribers



(f) RBMQ goodput with 4 subscribers

Figure 5: RBMQ message throughput (messages per second) and goodput (Gbps)

To explain RBMQ-tcp's higher CPU utilization compared to Redis' (63.0% versus 55.3%), we perform further profiling and find that about 10% of RBMQ-tcp's CPU time is spent on buffer synchronization and flow control. This overhead does not exist for Redis, since it does not implement a flow control scheme (which comes at the cost of lacking a mechanism to deal with back-pressure). When flow control is disabled, RBMQ-no-fc produces similar to significantly higher message throughput than Redis (Figures 5a, 5c, 5e). In Section 5, we show how our live streaming application uses RocketBufs' flexibility regarding flow control to reduce CPU utilization.

4.4 RBMQ Subscriber CPU Utilization

In most MOM deployments, subscribers are responsible for receiving and processing data, and in many cases data processing tasks are CPU-intensive [5, 7, 60]. Therefore, reducing the CPU overhead associated with receiving and managing data is critical as it allows the subscriber application to spend more CPU time processing data. To examine the reductions in subscriber CPU overhead provided by RocketBufs we now run a series of benchmarks where we measure and profile the CPU utilization on a subscriber host receiving large volumes of data. In these benchmarks, we use 10 publishers to send messages to the broker host, each sends messages at a rate of 5,000 mps for a total of 50,000 mps. We compare the Redis and RabbitMQ subscribers with three different configurations for the RBMQ subscriber: using `rIn` with TCP (*RBMQ-tcp*); using `rIn` with RDMA (*RBMQ-rdma*); and using `rIn` with TCP but with flow control disabled (*RBMQ-tcp-no-fc*).

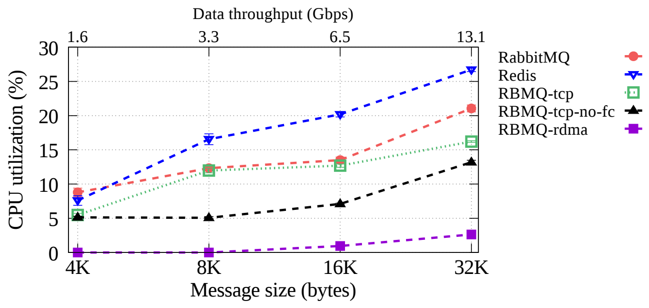


Figure 6: RBMQ subscriber CPU utilization (50,000 mps)

Figure 6 shows the average subscriber CPU utilization with 95% confidence intervals, obtained from `vmstat` samples collected every second. For TCP-based subscribers, the CPU is busy monitoring sockets and copying bytes from socket buffers to user-space. Therefore, the CPU utilization of these systems increases noticeably with larger messages. Overall, the Redis subscriber has the highest CPU utilization. Examining the source code of the hiredis client library reveals that it internally copies message data from an internal input buffer to another buffer before passing that buffer to the application, which explains Redis' higher CPU utilization compared to RabbitMQ and RBMQ-tcp. For RBMQ, a noticeable amount of CPU is spent on synchronizing buffers' state in RBMQ when flow control is enabled. When flow control is disabled, RBMQ's CPU utilization is further reduced (RBMQ-no-fc). Finally, the RBMQ subscriber using RDMA uses very little CPU resources regardless of the message size. This is because for RBMQ-rdma, the CPU is not used for data transfers and mainly used to handle RDMA completion events.

4.5 RBMQ Delivery Latencies

We also conduct experiments to measure delivery latency (how long it takes for messages to travel from publishers to subscribers). To avoid comparing timestamps generated on different machines, we measure and report the round-trip latencies of messages between two hosts communicating via a message broker (a similar approach to existing work [34]). The average latency of a MOM system might vary depending on the system's load, therefore, we compare the messaging latencies of the systems while handling 100,000 mps and 200,000 mps. We use a message size of 34 bytes (excluding the message header), which equals the average size of a Tweet [51].

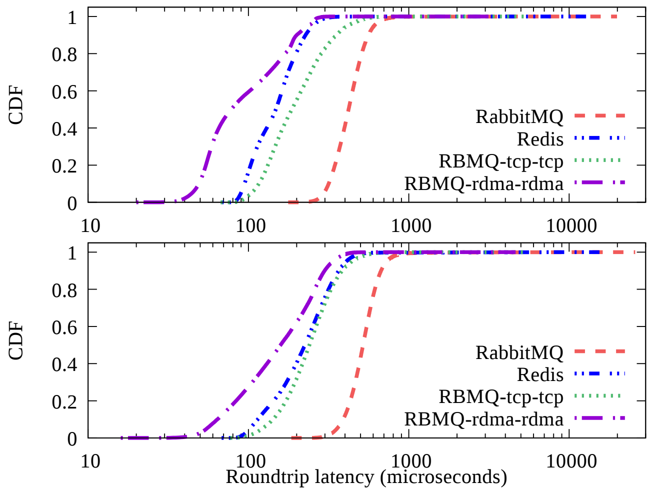


Figure 7: Round-trip latency: 100,000 mps (top) and 200,000 mps (bottom)

The CDFs of the round-trip latencies are shown in the two graphs in Figure 7. The x-axis represents log-scaled round-trip latency in microseconds. There are three important takeaways from the latency results. First, Redis and RBMQ are able to achieve lower latency than RabbitMQ. Secondly, RBMQ-rdma-rdma achieves significantly lower latency compared to TCP-based systems, since RDMA can avoid kernel overhead on data transfers. The median round-trip latency for RBMQ-rdma-rdma at 100,000 mps is 79 microseconds, which is 47% faster than Redis and 81% faster than RabbitMQ. At 200,000 mps, the median round-trip latency for RBMQ-rdma-rdma at 200,000 mps is 159 microseconds, 30% faster than Redis and 70% faster than RabbitMQ. Finally, we observe high tail latencies in all measured systems due to the queuing of messages. For example, RBMQ-rdma-rdma yields a maximum round-trip latency of 4,640 microseconds for the 200,000 mps workload, despite having low overall latencies (the 90^{th} percentile latency is 299 microseconds).

5 LIVE STREAMING VIDEO APPLICATION

To further demonstrate the utility of the RocketBufs framework, we present an application designed to manage the dissemination and delivery of live streaming video. Live streaming services like Twitch handle large amounts of video traffic and replicate them to many video servers within and across data centers in order to meet global demands [17]. We demonstrate how live video replication within a data center can be easily implemented using RocketBufs.

The design of the system is depicted in Figure 8. Emulated producers each generate a unique source of streaming video data. They send data in chunks (500 KB in our prototype) to a dissemination host over TLS-encrypted TCP connections (which emulates real services [75][16]). The dissemination host acts as a broker, which ingests video from stream sources and uses the `rOut` class to replicate this data to multiple web servers on separate hosts. When a video chunk arrives on the OpenSSL connection, the dissemination process requests an output buffer segment, places that chunk into the segment and signals the `rOut` instance to disseminate that chunk. The web server processes act as subscribers, which use the `rIn` class to subscribe to and receive all video streams. They also run a modified version of the *userver*, an open-source web server that has been shown to provide good performance [13, 50], to deliver video data to requesting viewers. Finally, viewers request desired video streams from the delivery servers, at the same rate that the streams are produced (2 Mbps in our prototype). These viewers are emulated using *httperf* [47], which makes requests to the userver over TLS-encrypted HTTP (HTTPS).

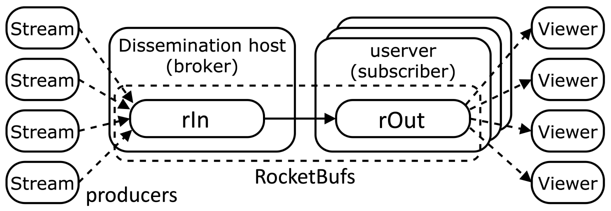


Figure 8: Using RocketBufs for live streaming video.

In our application, one buffer is used to disseminate each video stream, and is sized to hold five video chunks. Data is not retained in the delivery server's input buffer for long as it would become too stale to deliver to viewers. Therefore, we disable flow control and allow the framework to overwrite the oldest segment in the circular buffer when new data arrives. As shown in Section 4.4, this design avoids overhead due to buffer synchronization, however it makes it possible for data to be overwritten asynchronously while being delivered to viewers. To address this issue, we wrap the video chunks with consistency markers, which are checked by the server to see if data has been modified before or during the course of sending it to viewers. This may happen if the system is overloaded and the web server is unable to properly satisfy viewer requests. We consider such scenarios an error and terminate the viewing session. A real-world deployment would implement a load balancing mechanism to offload viewers to a less-loaded server.

We run a series of experiments to measure a delivery host's capacity to deliver live streaming video. These experiments are conducted for both TCP-based and RDMA-based dissemination with RocketBufs (denoted as *RB-tcp* and *RB-rdma*). The dissemination process resides on a big host (described in Section 4.2), and the web server-integrated subscribers reside on slim hosts. We use *tc* [2] to add bandwidth limits and network delays to simulate WAN effects for connections between viewers and delivery servers. We also modify httperf to include checks for data timeliness (timeouts) and validity. For comparison, we implement a version of our live streaming application using Redis [57] for video dissemination. We use Redis because it has previously been used for live video streaming [60] and it significantly outperforms RabbitMQ. In this case, RocketBufs does not use flow control (Redis does not support it) because ideally video is produced and consumed at the same rate.

We examine the performance of both systems as the load increases (by increasing the number of producers). For each experiment, we increase the number of emulated viewers until the delivery server's capacity is exceeded. This corresponds to the point at which the CPUs are saturated, video is not delivered to viewers in a timely manner and client requests time out We record the maximum number of viewers that can be supported which is reported using the total amount of viewer throughput. Benchmarks are repeated five times and we graph their means and 95% confidence intervals.

Figure 9 shows the results of these benchmarks (the 95% confidences intervals are not visible). As previously discussed RocketBufs incurs less CPU overhead than Redis when receiving data on a subscriber node. This difference grows as the amount of disseminated data (number of video streams) increases. For 20 Gbps of incoming data, while the Redis-based web server is only able to serve 13.5 Gbps of video to viewers, RB-tcp achieves over 17 Gbps, a relative increase of 27%. When RDMA is used (RB-rdma), the CPU overhead associated with receiving data is negligible, regardless of the dissemination throughput. As a result, RB-rdma's viewer throughput remains relatively consistent as the number of incoming streams increases and only drops from 21.7 Gbps (with 1,000 streams generating 2 Gbps of incoming data) to 21.0 Gbps (with 10,000 streams generating 20 Gbps of incoming data), an improvement of up to 55% versus Redis. Profiling the delivery servers reveals that Redis uses high amounts of CPU for `memcpy`. The design of RocketBufs, which is explicitly intended to work with data in-place, helps eliminate much of this overhead.

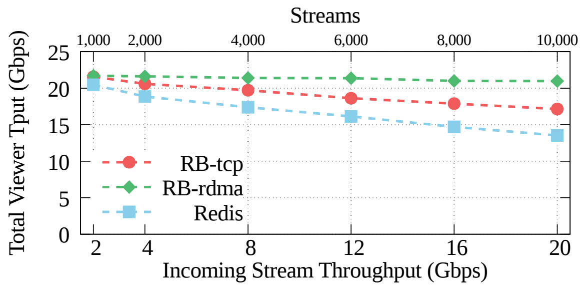


Figure 9: Max. error-free web server delivery throughput.

6 CONCLUSIONS

In this paper we present RocketBufs, a framework that facilitates the construction of high-performance MOM systems. RocketBufs provides abstractions and APIs for data dissemination between nodes that are easy to use and support different transport protocols. Because RocketBufs' APIs are agnostic to the underlying transport layer, developers can configure the framework to use different protocols or technologies without changing application code. This allows applications to reap the benefits of new technologies as support is added to RocketBufs.

Our evaluation of a publish/subscribe system (RBMQ) built using RocketBufs shows significant performance gains when compared to two widely-used industry grade MOM systems. When configured to use TCP, RBMQ achieves up to 1.9 times higher messaging throughput than RabbitMQ. RBMQ throughput is similar to that of Redis when configured comparably, however, RBMQ subscribers require significantly fewer CPU resources than Redis. When using RDMA, RBMQ throughput is up to 3.7 times higher than RabbitMQ and 1.7 times higher than Redis, while also reducing median delivery latency. In addition, on RBMQ subscriber hosts configured to use RDMA, data transfers occur with negligible CPU overhead. This allows CPU resources to be used for other purposes like application logic and data processing. Finally, we demonstrate the flexibility of our framework, by implementing a system to disseminate and deliver live-streaming video. We compare performance when using RocketBufs and Redis to disseminate video data and find that RocketBufs supports up to 27% and 55% higher simultaneous viewer throughput than Redis when using TCP and RDMA, respectively.

7 ACKNOWLEDGMENTS

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC); a David R. Cheriton Faculty Fellowship; a University of Waterloo President's Scholarship; and David R. Cheriton and Ontario Graduate Scholarships. We thank Zuhair AlSader for fruitful discussions and his feedback, and that of the anonymous reviewers, on earlier drafts of this paper.

REFERENCES

[1] Marcos K. Aguilera et al. 2018. Remote regions: A simple abstraction for remote memory. In *Proc. USENIX Annual Technical Conference (ATC)*. 775–787.
[2] W. Almesberger. 1999. Linux network traffic control – implementation overview.
[3] Amazon. [n.d.]. Amazon Kinesis Data Firehose. https://aws.amazon.com/kinesis/data-firehose/. Accessed October 8, 2019.
[4] Amazon. [n.d.]. Amazon Kinesis Data Streams. https://aws.amazon.com/kinesis/data-streams/. Accessed August 29, 2019.
[5] Amazon. [n.d.]. Amazon Serverless Data Processing. https://aws.amazon.com/lambda/data-processing/. Accessed June 23, 2019.
[6] AMQP. [n.d.]. OASIS AMQP 1.0 Specification. http://www.amqp.org/specification/1.0/amqp-org-download/. Accessed June 24, 2019.
[7] E. Androulaki et al. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *EuroSys*. Article 30, 15 pages.
[8] Alan Antonuk. [n.d.]. rabbitmq-c client library. https://github.com/alanxz/rabbitmq-c/. Accessed June 24, 2019.
[9] Apache. [n.d.]. ActiveMQ Artemis. https://activemq.apache.org/components/artemis/. Accessed October 8, 2019.
[10] Apache. [n.d.]. Apache OpenWhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org/. Accessed June 23, 2019.
[11] Apache. [n.d.]. Pulsar: distributed pub-sub messaging system. https://pulsar.apache.org/. Accessed June 23, 2019.
[12] Mike Belshe, Roberto Peon, and Martin Thomson. 2015. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540. https://doi.org/10.17487/RFC7540
[13] Tim Brecht, David Pariag, and Louay Gammo. 2004. accept()able strategies for improving web server performance. In *USENIX ATC*. 227–240.

[14] B. Cassell, H. Hoang, and T. Brecht. 2019. RocketStreams: A framework for the efficient dissemination of live streaming video. In *APSys*. 84–90.

[15] B. Cassell, T. Szepesi, B. Wong, T. Brecht, J. Ma, and X. Liu. 2017. Nessie: A decoupled, client-driven, key-value store using RDMA. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3537–3552.

[16] Colin Creitz. [n.d.]. Deadline Approaching: All Live Video Uploads Required to use RTMPS. https://developers.facebook.com/blog/post/v2/2019/04/16/live-video-uploads-rtmps/. Accessed September 11, 2019.

[17] Jie Deng, Gareth Tyson, Felix Cuadrado, and Steve Uhlig. 2017. Internet scale user-generated live video streaming: The Twitch case. In *Proc. Passive and Active Measurement Conference (PAM)*. Springer, 60–71.

[18] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2014. FaRM: Fast remote memory. In *NSDI*. 401–414.

[19] A. Dragojevic, D. Narayanan, E.B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *SOSP*. 54–70.

[20] Eclipse. [n.d.]. Vert.x. https://vertx.io/. Accessed October 8, 2019.

[21] F-Stack. [n.d.]. F-Stack. http://www.f-stack.org/. Accessed October 18, 2019.

[22] Evan Freitas. 2017. Presenting the Twitch 2016 year in review. https://blog.twitch.tv/presenting-the-twitch-2016-year-in-review-b2e0cdc72f18. Accessed April 12, 2019.

[23] J. Gascon-Samson, F. Garcia, B. Kemme, and J. Kienzle. 2015. Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud. In *IEEE International Conference on Distributed Computing Systems*. 486–496.

[24] Google. [n.d.]. Google Cloud Pub/Sub. https://cloud.google.com/pubsub/. Accessed June 23, 2019.

[25] Google. [n.d.]. Google Cloud Tasks. https://cloud.google.com/tasks/. Accessed October 8, 2019.

[26] Paul Grun et al. 2015. A brief introduction to the OpenFabrics interfaces — a new network API for maximizing high performance application efficiency. In *Proc. Symposium on High-Performance Interconnects (HOTI)*. 34–39.

[27] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *SIGCOMM*. 202–215.

[28] Hoang, Huy. 2019. *Building a Framework for High-performance In-memory Message-Oriented Middleware.* Master's thesis. University of Waterloo.

[29] IBM. [n.d.]. IBM Cloud Functions. https://www.ibm.com/cloud/functions. Accessed June 23, 2019.

[30] Iperf. [n.d.]. Iperf. https://iperf.fr/. Accessed September 11, 2019.

[31] J. Iyengar and M. Thomson. 2019. QUIC: A UDP-Based Multiplexed and Secure Transport. In *draft-ietf-quic-transport-19, Internet Engineering Task Force draft.*

[32] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. Van Renesse, S. Zink, and K.P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.* 36, 2, Article 4 (April 2019).

[33] A. Kalia, M. Kaminsky, and D.G. Andersen. 2016. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*. 185–201.

[34] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *SIGCOMM*. 295–306.

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *USENIX ATC*. 437–450.

[36] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be general and fast. In *NSDI*. 0–16.

[37] D. Kim et al. 2019. Freeflow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI*. 113–125.

[38] Richard Knop. [n.d.]. Machinery. https://github.com/RichardKnop/machinery. Accessed October 8, 2019.

[39] Joel Koshy. [n.d.]. Kafka Ecosystem at LinkedIn. https://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin. Accessed October 12, 2019.

[40] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. 2017. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *SOSP*. 137–152.

[41] Linux. [n.d.]. Perf. https://github.com/torvalds/linux/tree/master/tools/perf. Accessed Sept 12, 2019.

[42] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *EuroSys*. Article 1, 1:1–1:16 pages.

[43] P. MacArthur and R. D. Russell. 2012. A performance study to guide RDMA programming decisions. In *IEEE 14th International Conference on High Performance Computing and Communications*. 778–785.

[44] Mellanox. [n.d.]. RDMA Aware Networks Programming User Manual Rev 1.7. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf. Accessed April 12, 2019.

[45] B.N. Memon, X.C. Lin, A. Mufti, A.S. Wesley, T. Brecht, K. Salem, B. Wong, and B. Cassell. 2018. RaMP: A lightweight RDMA abstraction for loosely coupled applications. In *HotCloud*. 1–6.

[46] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX ATC*. 103–114.

[47] D. Mosberger and T. Jin. 1998. httperf – a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review* 26, 3 (1998), 31–37.

[48] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, L. Liss, M. Wei, D. Tsafrir, and M.K. Aguilera. 2019. Storm: A fast transactional dataplane for remote data structures. In *SYSTOR*. 97–108.

[49] NSQ. [n.d.]. NSQ: https://nsq.io/. Accessed Apr. 1, 2020.

[50] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, and Amol Shukla. 2007. Comparing the performance of web server architectures. In *EuroSys*. 231–243.

[51] Sarah Perez. 2018. Twitter's doubling of character count from 140 to 280 had little impact on length of tweets. https://techcrunch.com/2018/10/30/twitters-doubling-of-character-count-from-140-to-280-had-little-impact-on-length-of-tweets/. Techcrunch. Accessed June 25, 2019.

[52] Karine Pires and Gwendal Simon. 2014. DASH in Twitch: Adaptive bitrate streaming in live game streaming platforms. In *Proc. Workshop on Design, Quality and Deployment of Adaptive Video Streaming (VideoNext)*. ACM, 13–18.

[53] Pivotal. [n.d.]. Networking and RabbitMQ. https://www.rabbitmq.com/networking.html. Accessed June 6, 2019.

[54] RabbitMQ. [n.d.]. Advanced Message Queuing Protocol specification. https://rabbitmq.com/resources/specs/amqp0-9-1.pdf. Accessed June 25, 2019.

[55] RabbitMQ. [n.d.]. Flow Control. https://www.rabbitmq.com/flow-control.html. Accessed August 23, 2019.

[56] Redis. [n.d.]. hiredis. https://github.com/redis/hiredis. Accessed Apr 12, 2019.

[57] Redis. [n.d.]. Redis. https://redis.io. Accessed April 12, 2019.

[58] Redis. [n.d.]. Redis client handling. https://redis.io/topics/clients. Accessed Aug 23, 2019.

[59] Redis. [n.d.]. Redis Documentation. https://redis.io/documentation. Accessed Oct 6, 2019.

[60] L. Rodríguez-Gil, J. García-Zubia, P. Orduña, and D. López-de Ipiña. 2017. An open and scalable web-based interactive live-streaming architecture: The WILSP platform. *IEEE Access* 5 (2017), 9842–9856.

[61] S. Sanfilippo. [n.d.]. Disque. https://github.com/antirez/disque. Accessed Oct 8, 2019.

[62] V. Setty, R. Vitenberg, G. Kreitz, G. Urdaneta, and M. v. Steen. 2014. Cost-Effective Resource Allocation for Deploying Pub/Sub on Cloud. In *ICDCS*. 555–566.

[63] Yogeshwer Sharma et al. 2015. Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services. In *NSDI*. 351–366.

[64] Solarflare. [n.d.]. TCPDirect Delivers Lowest Possible Latency Between the Application and the Network. https://solarflare.com/wp-content/uploads/2019/02/SF-117079-AN-Solarflare-TCPDirect-White-Paper-Issue-5.pdf. Accessed October 18, 2019.

[65] Randall Stewart, John-Mark Gurney, and Scott Long. 2015. *Optimizing TLS for high-bandwidth applications in FreeBSD.* Technical Report. Netflix. 1–6 pages.

[66] Randall Stewart and Scott Long. 2016. *Improving high-bandwidth TLS in the FreeBSD kernel.* Technical Report. Netflix. 1–5 pages.

[67] H. Subramoni, G. Marsh, S. Narravula, Ping Lai, and D. K. Panda. 2008. Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand. In *Workshop on High Performance Computational Finance*. 1–8.

[68] Contributed Systems. [n.d.]. Faktory. http://contribsys.com/faktory/. Accessed October 8, 2019.

[69] The Linux Foundation projects. [n.d.]. DPDK Data Plan Development Kit. https://www.dpdk.org/. Accessed January 13, 2020.

[70] A. Trivedi, P. Stuedi, B. Metzler, C. Lutz, M. Schmatz, and T.R. Gross. 2015. RStore: A direct-access DRAM-based data store. In *ICDCS*. 674–685.

[71] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE: kernel RDMA support for data center applications. In *SOSP*. 306–324.

[72] Y. Wang et al. 2015. HydraDB: a resilient RDMA-driven key-value middleware for in-memory cluster computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[73] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled distributed transactions: Hybrid is better!. In *OSDI*. 233–251.

[74] B. Yi, J. Xia, L. Chen, and K. Chen. 2017. Towards Zero Copy Dataflows Using RDMA. In *SIGCOMM Posters and Demos*. 28–30.

[75] Youtube. [n.d.]. YouTube's road to HTTPS. https://youtube-eng.googleblog.com/2016/08/youtubes-road-to-https.html. Accessed July 26, 2019.

[76] I. Zhang, J. Liu, A. Austin, M.L. Roberts, and A. Badam. 2019. I'm Not Dead Yet!: The Role of the Operating System in a Kernel-Bypass Era. In *HotOS*. 73–80.