Building a Framework for High-performance In-memory Message-Oriented Middleware

by

Huy Hoang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Message-Oriented Middleware (MOM) is a popular class of software used in many distributed applications, ranging from business systems and social networks to gaming and streaming media services. As workloads continue to grow both in terms of the number of users and the amount of content, modern MOM systems face increasing demands in terms of performance and scalability. Recent advances in networking such as Remote Direct Memory Access (RDMA) offer a more efficient data transfer mechanism compared to traditional kernel-level socket networking used by existing widely-used MOM systems. Unfortunately, RDMA's complex interface has made it difficult for MOM systems to utilize its capabilities.

In this thesis we introduce a framework called RocketBufs, which provides abstractions and interfaces for constructing high-performance MOM systems. Applications implemented using RocketBufs produce and consume data using regions of memory called *buffers* while the framework is responsible for transmitting, receiving and synchronizing buffer access. RocketBufs' buffer abstraction is designed to work efficiently with different transport protocols, allowing messages to be distributed using RDMA or TCP using the same APIs (i.e., by simply changing a configuration file).

We demonstrate the utility and evaluate the performance of RocketBufs by using it to implement a publish/subscribe system called RBMQ. We compare it against two widely-used, industry-grade MOM systems, namely RabbitMQ and Redis. Our evaluations show that when using TCP, RBMQ achieves up to 1.9 times higher messaging throughput than RabbitMQ, a message queuing system with an equivalent flow control scheme. When RDMA is used, RBMQ shows significant gains in messaging throughput (up to 3.7 times higher than RabbitMQ and up to 1.7 times higher than Redis), as well as reductions in median delivery latency (up to 81% lower than RabbitMQ and 47% lower than Redis). In addition, on RBMQ subscriber hosts configured to use RDMA, data transfers occur with negligible CPU overhead regardless of the amount of data being transferred. This allows CPU resources to be used for other purposes like processing data.

To further demonstrate the flexibility of RocketBufs, we use it to build a live streaming video application by integrating RocketBufs into a web server to receive disseminated video data. When compared with the same application built with Redis, the RocketBufs-based dissemination host achieves live streaming throughput up to 73% higher while disseminating data, and the RocketBufs-based web server shows a reduction of up to 95% in CPU utilization, allowing for up to 55% more concurrent viewers to be serviced.

Acknowledgements

I would first like to thank my supervisor, Professor Tim Brecht, for his guidance and support throughout my degree. His supervision and inspiration enabled me to overcome the challenges that I encountered during the course of this project. I would also like to thank my colleague Benjamin Cassell, who worked with me on this project, for his friendship and collaboration.

I would like to thank Professor Khuzaima Daudjee and Professor Samer Al-Kiswany, for reading and providing suggestions to improve this thesis.

I want to dedicate this thesis to my parents and sister, who have always supported me my entire life. I could not have made it without them.

Last but not least, I appreciate the financial support provided by the David R. Cheriton School of Computer Science, Professor Tim Brecht and the University of Waterloo.

Table of Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

1.1 Background, Motivation and Goals

In order to handle increasingly complex workloads and meet growing scalability requirements, many modern applications and services are designed as distributed, event-based systems, which consist of smaller components, each with a separate responsibility. These components work together by exchanging and reacting to data from other components. Message-Oriented Middleware (MOM) systems are a popular class of software used by many distributed applications to facilitate this type of data exchange in a loosely-coupled manner. Applications using MOM systems follow a produce-disseminate-consume (PDC) design pattern, where one or more producers send data as messages to an MOM substrate (often comprised of message brokers) for scalable dissemination to a possibly large number of consumers. Examples of applications and services that utilize MOM systems include IBM's cloud functions [49], the Apache OpenWhisk serverless framework [96], the Hyperledger blockchain framework [12], Facebook's event propagation system [16], and video streaming applications [89]. The emergence of new types of workloads and the rising popularity of MOM systems have led to the continued design and implementation of new systems. Over the past five years alone, many new open-source MOM systems have been developed [34, 100, 14, 90, 16, 58], and cloud providers have continued to introduce new MOM services as part of their infrastructure [45, 5, 6, 44].

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 [92], while Twitch, a live streaming video service, handles hundreds of billions of minutes worth of video content a year [40], and

this amount is expected to continue to grow. Other classes of applications, such as online gaming [41] and stock trading [98] require messages to be delivered with extremely low latency. As a result, constructing high-performance and scalable MOM systems is a problem that has received much attention in both industry and academia. In this thesis, we are interested in building in-memory MOM systems to support high-throughput and low-latency message delivery.

Often in MOM deployments, data is moved between hosts that reside within a data center [59, 91]. One approach to building high-performance MOM systems is to leverage data center networking capabilities, especially those that offer kernel-bypass features to enable high-throughput and/or low-latency communication [67, 37, 94]. For example, Remote Direct Memory Access (RDMA) is one example of a technology that provides such features. RDMA allows applications to bypass the operating system and offload the transport layer processing to the Network Interface Card (NIC), resulting in faster and more efficient data transfers. With the introduction of RDMA over Converged Ethernet (RoCE) NICs, data centers can support RDMA over classical Ethernet networks for little or no incremental cost over NICs that do not support RDMA [69]. Unfortunately, commonly-used MOM systems currently do not take advantage of this capability and instead typically use kernel-based TCP, which in many cases incurs protocol processing and copying overhead (even for communication within a data center [47]), limiting throughput and resulting in higher latency.

We identify two issues that we believe have prevented current MOM systems from adopting RDMA. First, it is well known that RDMA is difficult to use [1]. A simple application using RDMA to transfer data must go through the process of establishing device contexts, registering memory regions, exchanging keys and implementing code to monitor and handle events. Past studies have proposed both hardware primitives [2] and software abstractions [1] to address this difficulty and to simplify working with RDMA. However, in most cases these abstractions are designed for other types of applications (e.g., remote memory or key-value store systems), and are not suitable for building MOM systems. Secondly, the native RDMA *verb* interface has abstractions and APIs fundamentally different from the socket abstraction which is conventionally used with protocols like TCP. With RDMA, applications transfer data by posting work requests onto *queue pairs*, which require direct access to application-level memory, instead of calling `send/recv` on sockets which typically involves moving data between kernel-space and user-space buffers. Because of this difference, a lot of engineering effort would be required for both new and existing MOM systems to support RDMA, since two separate data transfer implementations would be required (since traditional protocols such as TCP are still necessary for non-RDMA NICs and for communication over a wide area network).

A solution that provides compatibility between RDMA and the socket abstraction is `rsocket` [62], an API wrapper on top of native RDMA. The `rsocket` APIs (e.g., `rsend`, `rrecv`) are intended to match the behavior of the corresponding socket function calls. The `rsocket` library internally allocates and registers buffers for RDMA "sockets", which are used to store incoming and outgoing data. It translates socket-like APIs to RDMA verbs by copying data between these internal buffers and application-defined memory. While this approach provides a familiar abstraction, it sacrifices performance due to copying overhead [57, 107]. This overhead is especially noticeable when transferring large amounts of data. For example, we have conducted some simple experiments and found that a host (with an eight-core 2.0 GHz Intel Xeon D-1540 CPU) using `rsocket` utilizes 18% of the CPU while receiving data at 32 Gbps. In contrast, as will be demonstrated later in Chapter 5, our RDMA implementation utilizes only 3% of the CPU while receiving the same amount of data.

We believe that MOM systems should be able to utilize RDMA (and other techniques designed for high-throughput and/or low-latency data center communication) for high-performance in-data center messaging, while not being forced to choose between suboptimal performance (i.e., `rsocket`) and a complex implementation (using the native RDMA APIs). To this end, we propose RocketBufs, a framework to facilitate the easy construction of high-performance in-memory Message-Oriented Middleware systems. The goals of this thesis are to:

- Provide a framework with natural abstractions and easy-to-use APIs, tailored for building a variety of MOM systems. As will be discussed in Chapter 2, a key property of MOM systems is the independence between data producers and consumers. RocketBufs' APIs therefore must satisfy this property. The framework should also provide support for the efficient implementation of flexible MOM topologies.

- Enable MOM applications built with RocketBufs to utilize different transport protocols (including TCP and RDMA) using the same abstraction. Changing the transport protocol should require only changes to the system's configuration and should not require any modifications to application code.

- Provide abstractions and APIs that enable efficient and scalable implementations for both RDMA and TCP-based networking.

Given the continuous development of new MOM systems, we believe that such a framework would greatly simplify the construction of new systems while allowing them to achieve high performance. Furthermore, 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 (e.g., QUIC [53], F-Stack [37], TCPDirect [94]) could also be efficiently supported by the framework, providing benefits to all systems built using RocketBufs.

1.2 Contributions

In this thesis, we make the following contributions:

- We propose and implement RocketBufs, a framework that facilitates the construction of high-performance 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`) objects that are associated with buffers. The framework transfers buffered data between communicating processes and provides mechanisms for flow control. RocketBufs' abstractions and APIs are designed to efficiently support different transport protocols, including RDMA and TCP.

- We describe a prototype implementation of RBMQ, a publish/subscribe messaging system built on top of RocketBufs. To demonstrate that such a system can be built with relatively little effort, we provide pseudocode for the key components of the message broker, publisher and subscriber.

- We evaluate the performance of RBMQ by comparing it against RabbitMQ and Redis, two widely-used, industry-grade MOM systems that support publish/subscribe messaging. Our evaluations show that, when disseminating to four subscribers, both Redis and RBMQ using TCP achieve significantly higher messaging throughput than RabbitMQ (which has a flow control scheme equivalent to RocketBufs). When RDMA is used, RBMQ shows substantial gains in performance and outperforms Redis in both messaging throughput and average delivery latency. Additionally, an RBMQ subscriber using RDMA incurs negligible CPU overhead while receiving data, allowing CPU resources to be used for other purposes like data processing.

- To further demonstrate the flexibility of RocketBufs, we use it to build a live streaming video application consisting of a dissemination host and multiple delivery hosts. The dissemination host acts as a message broker, which ingests video streams from video sources and disseminates them using the `rOut` class to web servers for delivery

to viewers. The web servers use the `rIn` class to subscribe to and receive disseminated video data, either using TCP or RDMA, and serve video content to viewers over HTTPS. We also build a version of this application that uses Redis to disseminate video streams. Our empirical evaluation shows that RocketBufs is able to support up to 55% more simultaneous viewers when compared with the Redis-based application.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 presents background information and related research. Chapter 3 describes the design, implementation and optimizations used in RocketBufs. In Chapter 4, we describe the implementation of RBMQ, a publish/-subscribe messaging system built using the RocketBufs framework. We then compare its performance against that of RabbitMQ and Redis. In Chapter 5, we describe how we use RocketBufs to disseminate live streaming video data between hosts to allow delivery to scale to a much larger number of viewers. We then compare the performance of our live streaming application against the same application built using Redis for data dissemination. Chapter 6 contains our conclusions and ideas for future work.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Message Oriented Middleware

To handle increasingly complex workloads and meet growing scalability requirements, many modern applications and services are designed as distributed, event-based systems, which consist of small and responsibility-separate components. These components work together by exchanging and reacting to data from one another. Often data produced by a component needs to be accessed by one or more other components. Message-Oriented Middleware (MOM) refers to software infrastructure supporting this type of data exchange in an independent and loosely-coupled manner. Applications that use MOM systems follow a produce-disseminate-consume (PDC) design pattern, depicted in Figure 2.1 (where the arrows represent the movements of data between nodes). In this design pattern, *publishers* (or *producers*) send data as *messages* to an MOM substrate instead of sending them directly to consumers. The MOM substrate, often composed of multiple message brokers to form a broker *overlay*, is responsible for routing messages to consumers (or *subscribers*). The routing of messages is based on *subscriptions*, registered by subscribers to express their interests in specific data (sometimes expressed as topics). This design pattern allows for the decoupling (or independence) of communicating parties (i.e., the publishers and subscribers). Specifically, Eugster et al. [36] list three independence properties of MOM systems:

- *Space independence*: The communicating parties do need to know the identity of one another. The publishers send data to the brokers and the subscribers obtain data

directly from the brokers. This mechanism allows the publishers and the subscribers to exchange data without having to hold any references to one another (only references to the brokers are required). Publishers and subscribers also do not need to know how many other publishers or subscribers are present in the system.

- *Time independence*: The interaction with the same data may take place on participating nodes at different points in time. In particular, a message published by a producer might get queued in the brokers and delivered to different subscribers at different times, depending on their processing rates and possibly other factors such as message priorities.

- *Synchronization independence*: Publishers are not necessarily blocked by slow consumers when producing data, since data can be queued and, in some cases, saved to secondary storage on the brokers. Similarly, subscribers can be notified that data is available while performing concurrent activities.

In Chapter 3, we discuss how these properties help inform our design of RocketBufs' APIs. The emergence of new types of workloads and the rising popularity of MOM systems have led to the continued development of new systems. We have conducted an informal search and found many popular open-source MOM systems (Github projects with a rating of at least 1000 stars, which indicates high popularity) being newly developed in the past five years alone [34, 100, 14, 90, 16, 58]. In the same period, cloud providers have continued to introduce new MOM services as part of their infrastructure [45, 5, 6, 44].

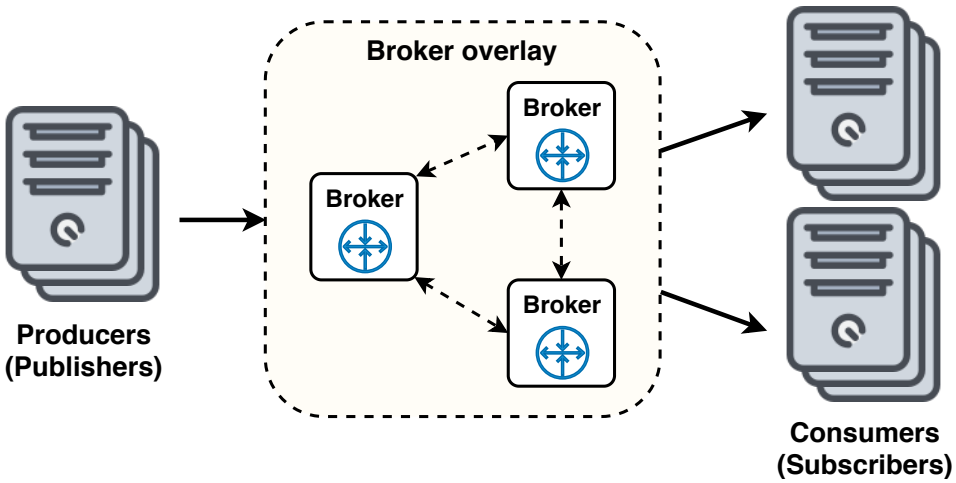

Figure 2.1: Message-Oriented Middleware.

The scale at which MOM systems operate has also become larger over time, both in terms of the number of users and the amount of data. Facebook, for example, reports in 2015 that their pub/sub system delivered over 35 Gigabytes of data per second [92]. Another report by LinkedIn shows that their MOM deployment handles over two Petabytes of data every week [59]. In order to scale to handle increasing workloads, MOM systems often contain multiple message brokers that form a broker cluster (or overlay). Data can be moved from one broker to another for scalable delivery to subscribers of that data. Broker overlays can have different topologies, depending on their specific workloads. In RocketBufs, we provide classes that allow flexible topologies to be easily constructed.

MOM systems typically operate using one of two main types of workflows: message-queuing or publish/subscribe (pub/sub). In the message-queuing workflow, processes *consume* messages from *queues*, and each message in a queue is delivered to a single consumer. This workflow is designed for scenarios such as a load balancer, where messages in the queue represent tasks to be carried out by the consumer processes. In the pub/sub workflow, messages produced by the publishers are delivered to all interested subscribers. Pub/sub systems can be further categorized into *topic-based* and *content-based*. In topic-based pub/sub systems, messages are associated with different *topics* (or *subjects*) and are delivered to subscribers of corresponding topics. On the other hand, content-based pub/sub systems deliver a message to a subscriber only if the message content matches the constraints defined with the subscription. For example, a subscriber can define pattern-matching rules with the subscription so that the brokers only deliver to the subscriber the messages that match those rules. An example of such a system is an intrusion detection system which detects and notifies control applications of abnormal traffic. RocketBufs' abstractions and APIs (described in Chapter 3) are designed to allow for the implementation of MOM systems using both the topic-based and the content-based pub/sub workflows, as well as the message-queuing workflow.

The delivery of messages from brokers to subscribers in MOM systems can be categorized as *push*-based and *pull*-based, depending on the party initiating the data transfer. A subscriber in a pull-based system sends *pull requests* to fetch messages from the brokers before consuming them. This mechanism allows the subscribers to control the rate at which messages are received, however this approach can lead to higher delivery latencies, since there is a delay between the pull request and the delivery of the message. The pull-based model is typically used by client-driven MOM systems such as Kafka [31] and Amazon SQS [9]. On the other hand, in pushed-based systems (e.g., RabbitMQ [80], Amazon Notification Service [8]) brokers proactively send published messages to subscribers as soon as possible. Our work with RocketBufs in this thesis uses a push-based model to allow us to focus on the framework's delivery latency. We discuss ideas for supporting pull-based

delivery in Chapter 6.

2.1.2 Kernel-based Networking

The primary function of MOM systems is to deliver data from producers to consumers. Implementing efficient data transfers is therefore a key to developing high performance MOM systems. In real deployments, MOM systems can be geo-distributed, however in many cases, data is also moved between hosts that reside within a data center [91]. For example, LinkedIn deploys several clusters of Kafka brokers in each of their data centers to handle data from other system components [59]. Another example is Bitnami, a VMWare service that offers cloud deployment automation. Bitnami deploys RabbitMQ clusters in their data center to allow their system to scale to handle large numbers of messages [43]. We believe that such systems would benefit from leveraging networking capabilities available in modern data centers, such as RDMA-enabled NICs or other specialized hardware [23, 93].

Most current and commonly-used MOM systems, including RabbitMQ [79], Redis [84] and Kafka [60], rely on TCP/IP for data transfers, even within a data center. Some systems, such as ActiveMQ [39] also provide support for messaging over UDP (although at the cost of sacrificing reliability). The conventional method of working with these protocols involves using the socket APIs, provided by most modern operating systems. These APIs are implemented in the system kernel's networking stack, and are responsible for packet processing (e.g., forming packets and ensuring reliability). To send data, application developers call `send` (or `write`) system calls, which copy data to kernel-space socket buffers, from which data is transferred over the network. Similarly, receiving data requires calling `recv` (or `read`) system calls, which eventually copy data from the kernel-space socket buffers into the application-defined user-space buffer.

Despite being a mature and familiar technology, kernel-based socket networking suffers from a number of inefficiencies. First, CPU resources are required for processing packets and copying data between user-space memory and kernel-space socket buffers. Secondly, context switching overhead from networking-related system calls results in extra latency. Guo et al. [47] show that, for example, kernel-based TCP uses 12% of a 32-core Intel Xeon CPU while receiving data at 40 Gbps, and latencies of networking-related system calls can be as high as tens of milliseconds. These limitations are undesirable in many modern applications, such as CPU-intensive data analysis [38] or latency-sensitive applications [110].

To address these limitations, recent work has focused on developing user-space networking stacks that bypass the kernel and in many cases, offload parts of the transport protocol to hardware. Some examples are U-Net [105], TCPDirect [94], and F-Stack [37]. In the

next section, we provide details of one such technology, Remote Direct Memory Access (RDMA), which is supported by the current RocketBufs prototype. We discuss other work related to user-space networking in Section 2.2

2.1.3 RDMA

Remote Direct Memory Access (RDMA) is a modern efficient networking alternative to traditional networking protocols like TCP or UDP. RDMA is enabled on specialized Network Interface Cards (referred to as RNICs) which carry out data transfer operations without involving the host's main CPU. RDMA also bypasses the kernel and provides applications with user-level access to the NIC's resources. This section summarizes the aspects of the protocol used in this thesis. Note that the research in this thesis uses RDMA over Converged Ethernet [68] and Mellanox's RDMA implementation [67].

RDMA works by interacting directly with user-space memory. To use RDMA, applications first allocate and register memory regions with the RNIC on both the sending and receiving host. These memory regions are then pinned (locked) into memory to prevent them from being paged out by the operating system. Data transfers can then be executed using the RDMA *verb* interface. The RNIC moves data from one host's registered memory regions to another's, bypassing both hosts' kernels and CPUs. This mechanism is depicted in Figure 2.2.

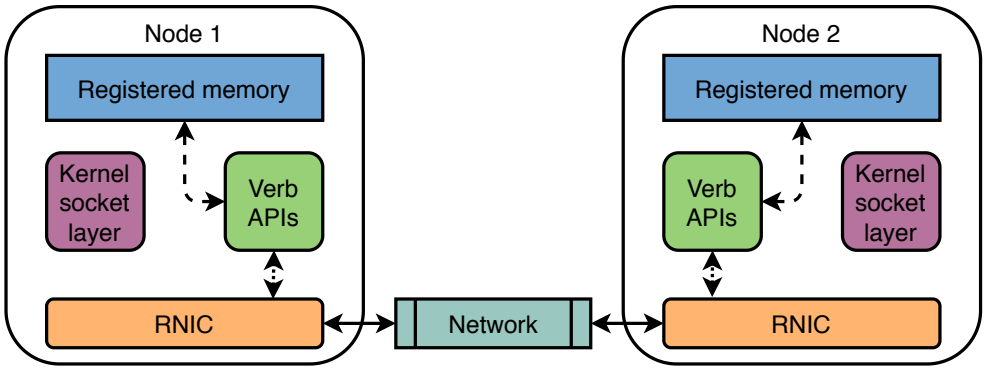

Figure 2.2: Overview of RDMA communication.

From the programmer's perspective, RDMA's verbs are executed by posting work requests onto a queue pair (QP). RDMA QPs are asynchronous in nature: work requests posted to a QP are queued for execution and when an operation completes, a completion event is posted to the QP's completion queue, which is a data structure that can be

monitored by the application. RDMA supports several types of transport mechanisms for QPs: reliable connection (RC), unreliable connection (UC) and unreliable datagram (UD). As suggested by their names, RC and UC are connection-oriented. The main difference between them is that with RC, the RNICs guarantee reliable data transfers using ACK and NACK packets [70], while UC does not provide this guarantee. On the other hand, the Unreliable Datagram transport type (UD) does not require connections to be established between hosts and also provides multicast functionality (similar to UDP and UDP-based multicast). However, the delivery of UD datagrams are not reliable and the datagram size is limited to the network's Maximum Transmission Unit (MTU). In this thesis, we use RC as the transport type for our RDMA implementation because of its reliability guarantees. This approach has also been advocated for in recent literature [72].

RDMA verbs can be categorized as *one-sided* and *two-sided*, based on which systems are involved in the operation. Two-sided verbs include `send` and `recv`, which require both communication endpoints to post work requests onto their corresponding queue pairs. The sender posts a `send` request indicating which data is to be transferred and the receiver must post a `recv` request to indicate where incoming data should be placed. One-sided verbs, on the other hand, only require one host to initiate the operation. For example, the `read` verb allows reading data from remote memory, and the `write` verb allows writing data to remote memory without generating completion events on the remote end. When remote completion events are desired (to notify the receiving node of incoming data), applications can make use of hybrid verbs such as `write-with-imm`. This verb sends an additional 32-bit piece of *immediate* data following the write operation, which can be consumed from the receiving node's queue pair. Additionally, RDMA also supports 64-bit *fetch-and-add* and *compare-and-swap* atomic operations.

The cost of RDMA-enabled NICs has become more competitive over the years. For example, a 40 Gbps Mellanox ConnectX-4 NIC [51] with RDMA support has an almost identical price to a non-RDMA 40 Gbps Intel Ethernet NIC [66]. With the introduction of RDMA over Converged Ethernet (RoCE) [68], RDMA can be deployed in data centers using a classical Ethernet network and switches. We expect that the decreasing costs and increasing accessibility of RDMA NICs will lead to their deployment in more data centers and their adoption in more types of applications, which is one of our motivations for developing RocketBufs.

2.1.4 Live Streaming Video

In Chapter 5, we show how RocketBufs can be used to implement an HTTP live streaming video application. Live streaming services such as Twitch [104] and YouTube Live [108]

are growing rapidly in popularity, with Twitch being the fourth largest consumer of peak Internet traffic in the United States in 2015 [103]. It is also estimated that live streaming video traffic will account for 17% of all Internet video traffic by 2022 [28]. This rapid growth requires streaming services to develop scalable infrastructure, capable of handling large amounts of live stream video data at a global scale. Efficiently disseminating live streaming video traffic is another motivation for our design and implementation of RocketBufs.

We now use Twitch as an example to describe how a live streaming service operates. Twitch streamers broadcast video content by sending video data (usually encrypted) to a transcoding endpoint using the *Real-Time Messaging Protocol* (RTMP) [95], which operates over TCP/IP. Video data is then transcoded into multiple *HTTP live streaming* (HLS) streams. Each stream uses a different video resolution resulting in streams with different bitrates. The most common video resolutions offered by Twitch are 360p, 480p, 720p and 1080p. To reduce video bandwidth and the cost of transcoding streams that are not popular, Twitch only selects certain streams to transcode to multiple resolutions, based on the popularity of the stream and Twitch's partnership with certain streamers. In order to meet global demands for content and to make content available on servers close to viewers, Twitch geo-replicates video data to multiple data centers containing clusters of servers around the world. From there video traffic is sent to nearby viewers. Work by Deng et al. in 2017 [30] identified at least 875 video servers distributed across 21 data centers in four continents. Their study also shows that Twitch dynamically replicates video streams across servers depending on the stream's popularity (number of current viewers). Streams with higher view counts are replicated to a larger number of servers. For example, a stream with approximately 20,000 simultaneous viewers was replicated to more than 50 servers in North America alone.

Viewers of HTTP streaming video services consume content by making HTTP requests to delivery servers for data and appending that data to the client device's playback buffer. Delivering video content can require significant amounts of CPU resources, which are required for ingesting data from video sources (either the transcoding service or other delivery servers) and sending them to requesting viewers over HTTP. Furthermore, in recent years, streaming services have adopted TLS encryption for HTTP video traffic [109], which also increases CPU resource consumption. This workload poses a challenge for building large-scale live streaming services to meet increasing demands, especially regarding latency requirements. Compared to video-on-demand (VOD) viewers, live streaming video viewers are more sensitive to latency. VOD playback devices can deal with changes in network latency by requesting large amounts of video data to pre-fill their playback buffers. This can not be done for live streaming, because the playback buffer can only be filled as fast as the content is generated. Viewers of live streaming content in general also have higher

expectations for service quality. Dobrian et al. [32] showed that, for example, a 1% increase in the buffering ratio (the percentage of time spent waiting for data to be available to play) for a 90-minute soccer game translated to viewers watching 3 fewer minutes of the game, impacting viewer retention and revenue. Our work with live streaming video leverages RocketBufs' networking capabilities to improve the CPU efficiency of video data dissemination. These savings in CPU utilization allow delivery servers to service more viewers in a timely manner.

2.2 Related Work

In this section, we look at past studies related to RocketBufs. We divide related research into several categories: networking systems and libraries, research on RDMA-based systems, research that focuses on improving the performance of MOM systems, and research on live streaming video.

2.2.1 Networking Frameworks

Our main goal with RocketBufs is to enable high-performance networking in MOM systems. One line of previous work has focused on developing techniques to improve the performance of traditional protocols like TCP by bypassing the kernel and moving the protocol processing into user space. This approach avoids the overhead incurred by network-related system calls and results in improved throughput and latencies. Examples of systems that use this approach include U-Net [105], F-Stack [37] and TCPDirect [94]. Unlike RocketBufs, these systems focus on the transport layer and are not tailored to a specific type of applications (i.e., MOM systems). Some of them (like U-Net [105]) require modifications to the Operating System or specialized hardware (TCPDirect [94]). Our work with RocketBufs focuses on designing abstractions and APIs for MOM systems that allow different transport protocols to be efficiently supported. In future work, RocketBufs could incorporate these other kernel-bypass techniques to further improve its messaging performance using protocols other than RDMA. We believe that our proof-of-concept RDMA implementation demonstrates that other kernel-bypass systems would also see significant benefits (e.g., due to reduced system call overhead and eliminating data copying).

Another goal of RocketBufs is to facilitate the efficient access to networking resources through an easy-to-use interface. There are several networking libraries that share this objective. For example, the `libfabric` library [46] provides abstractions for access to

networking hardware, enabling applications to discover and utilize available communication mechanisms (including RDMA). Their abstractions however operate at a lower level than RocketBufs. They do not provide MOM-tailored features that RocketBufs does, such as buffer management and flow control. Additionally, `libfabric` does not make as much effort to hide the complexity of RDMA programming (e.g., it still requires the application to perform memory registration for use with RDMA). Technically, `libfabric` could be used to implement parts of RocketBufs' networking components. However, due to our familiarity with the code base, we chose to implement the prototype for RocketBufs using our existing infrastructure which has been used to develop Nessie [22] - a high performance key value store.

Another library, ZeroMQ [111], provides a socket-based abstraction for building messaging systems. Each ZeroMQ socket has a *type* that indicates the messaging pattern it supports. For example, ZMQ_REQ and ZMQ_REP sockets are used to implement applications with RPC-style communication (using requests and replies), while ZMQ_PUB and ZMQ_SUB sockets are used to implement the pub/sub design pattern, in which data sent to a ZMQ_PUB socket is delivered to all connected ZMQ_SUB sockets. In a fashion similar to RocketBufs, ZeroMQ provides a messaging abstraction that allows for communication over different transport protocols (e.g., TCP and UDP). However ZeroMQ's abstraction is lower-level than RocketBufs', and is designed for simple direct messaging between nodes without the notion of brokers or other middleware. ZeroMQ also does not support RDMA, and adding RDMA support to ZeroMQ using a socket abstraction would also be difficult without sacrificing performance (as is the case with `rsocket`, explained in Section 2.2.2). RocketBufs is designed specifically for building new MOM systems (e.g., to allow consumers to subscribe to specific buffers and to support buffer splicing on message brokers). RocketBufs also provides a memory-based interface which is designed to avoid unnecessary data copies and to allow for efficient implementations of MOM systems, whether they choose to use RDMA or TCP. While ZeroMQ could be used to implement RocketBufs' TCP-based networking component, we chose to provide our own TCP implementation to avoid ZeroMQ protocol overhead [112] and to optimize TCP performance.

2.2.2 RDMA

As mentioned previously, despite showing significant benefits, directly using RDMA is complicated [1]. Some work has been conducted on building systems and designing abstractions to improve RDMA usability. For example, `rsocket` [62] provides an API wrapper on top of native RDMA functions to enable RDMA access using a socket abstraction. The `rsocket` APIs (e.g., `rsend`, `rrecv`) are intended to match the behavior of their corresponding socket

function calls. To translate these APIs to RDMA verbs, `rsocket` internally allocates and registers buffers for RDMA "sockets", which are used to store incoming and outgoing data (since RDMA requires access to pre-registered memory). Networking operations involve copying data between these internal buffers and application buffers. This copying overhead introduces inefficiencies during data transfers [57, 107]. We have conducted some simple experiments and found that a host using `rsocket` utilizes 18% of the CPU while receiving data at 32 Gbps, compared to only 3% when using RDMA. In RocketBufs, we provide a memory-based interface which is designed to avoid unnecessary data copying and to take full advantage of RDMA.

Recent research has explored ways to use RDMA in the context of remote memory and distributed key-value store systems, including FaRM [33], Remote Regions [1], LITE [102] and many others [55, 69, 106, 22, 72]. FaRM, for example, exposes the cluster's memory as a shared address space, and provides transactional `read`/`write` APIs to interact with objects within that address space. LITE is similar to FaRM with regards to the functionalities it provides, however LITE is implemented in the kernel and its APIs are exposed as system calls. This model enables LITE to hide RDMA's complex management from user applications, making it easier to access LITE's functionalities. For example, LITE allows different processes on a host to safely share RDMA resources (e.g., memory regions and CPU required for monitoring RNIC events). However, this comes at the cost of giving up RDMA's kernel bypass feature. Remote Regions expose the cluster's memory regions as files, which are represented as file descriptors. These file descriptors support `read`, `write` and `mmap` operations, providing RDMA access via familiar APIs. While these systems provide interfaces that simplify access to RDMA, they are not designed for the same use cases as RocketBufs (which is tailored to MOM systems). For example, they do not provide APIs for subscribing to and continuously receiving specific data, or for efficiently forwarding data on message brokers. Some of these systems also require operating system modifications (as is the case with FaRM).

In addition to having a complicated application interface, RDMA also requires careful tuning and optimizations in order to achieve high performance. Many studies have documented experiences with and guidelines for building efficient RDMA-based systems, some of which have been incorporated into RocketBufs' RDMA implementation. For example, the use of the one-sided `write-with-imm` verb has been advocated for by Novakovic et al. [72]. This verb allows the sender to communicate additional information (an extra 32-bit piece of immediate data following the write operation) while at the same time enabling notifications of incoming data on the receiver. RocketBufs leverages this verb to transfer application data by writing the message data to the receiver's memory location, and using the immediate data field to store the buffer's identifier. Another example of incorporating

ideas from other research into optimizing RDMA is using the advice from Kalia et al. [56], which describes the benefits from reducing the number of registered memory regions to avoid contention on the RNIC cache resources. RocketBufs minimizes these registrations by assigning memory space for buffers from pre-allocated memory pools.

Finally, by separating the application interface from the networking layer, RocketBufs allows for independent improvements of the networking layer which would benefit all applications built using our framework. This is especially important because RDMA-enabled hardware and best practices are continuing to evolve [72] and new kernel bypass systems and products are being developed.

2.2.3 Message-Oriented Middleware

RocketBufs' main goal is to enable the construction of high-performance MOM systems. Prior to our work, many studies have explored techniques for improving the performance of MOM systems.

Jokela et al. introduce LIPSIN [54], a multicast forwarding fabric for topic-based publish/subscribe systems. Instead of using a broker overlay to route messages, LIPSIN embeds the pub/sub view into the network layer (layer 3), making the network fabric pub/sub-aware. Additionally, instead of using endpoint IP addresses, LIPSIN routes packets by identifying links. Each unidirectional point-to-point link in the network is given a unique *Link ID* and a bitmap is used to determine which packets to forward over a link. When a message is published, LIPSIN uses the topic ID and the network graph to figure out which nodes are interested in the message and creates a conceptual forwarding tree. The publisher then encodes all Link IDs of the tree into a Bloom filter (referred to as *zFilter*) and places it into the packet header. To forward packets, each forwarding node (including the publisher) matches its outgoing links' IDs against the zFilter and sends the matched packets along the corresponding links. LIPSIN operates at a lower level than RocketBufs and does not support RDMA. However, it does provide interesting ideas for how MOM systems can be improved with a tailored network fabric. We hope to consider these ideas when making future improvements to RocketBufs' networking layer.

Much research effort has focused on designing MOM topologies to better utilize resources and improve the efficiency of message delivery [101, 27, 42, 73, 25, 26, 24, 18]. For example, Chockler et al. [26] introduce the concept of a *Topic-Connected Overlay* (TCO) for topic-based pub/sub systems. The idea of TCO is to group all nodes interested in a topic into a connected sub-overlay, so that messages published on a topic can be delivered to all interested nodes without being forwarded by non-interested nodes, thereby reducing

relaying overhead. Building upon this idea in 2016, Chen et al. [24] proposed several practical optimizations, allowing systems to form a TCO more quickly. Other work has also focused on improving content-based pub/sub systems. For instance in 2013, Barazzutti et al. introduced StreamHub [18], a content-based pub/sub system with a multi-tiered architecture. Instead of using message brokers, StreamHub consists of tiers (referred to as *operators*), each of which implements a subset of the content-based pub/sub service. When a message is published, each operator performs the appropriate message processing before passing the processed message to the next operator in the pipeline. This design focuses on improving resource utilization by avoiding repeated processing of messages by multiple message brokers. The aforementioned studies (some as recent as 2018) show that MOM systems are continuing to be studied and evolve. With RocketBufs, we focus on providing an efficient and easy-to-use interface, upon which a wide variety of MOM systems and flexible topologies can be implemented. In this thesis, we design and evaluate RocketBufs in the context of a topic-based pub/sub system. We plan to implement more targeted support for content-based pub/sub systems in future work.

Because of the growing popularity of MOM systems, there have been many open-source message-queuing systems developed to provide developers with ready-to-deploy building blocks [79, 39, 50, 15, 60]. In this thesis, we compare the performance of some example applications we build using RocketBufs against RabbitMQ [79] and Redis [84], two widely used systems that support in-memory publish/subscribe workflows. RabbitMQ is a popular, flexible message queuing system used by many Internet applications and companies, including Reddit [35], VMWare, AT&T and Mozilla [11]. RabbitMQ offers a variety of configurable options that allows trading off performance with reliability. We choose RabbitMQ as a point of comparison against the pub/sub system we build using RocketBufs (RBMQ) because of its popularity and flexibility. Redis, on the other hand, is primarily an in-memory key-value store system, but also provides support for publish/subscribe workflows. Redis' users include popular Internet services such as Twitter, Github and StackOverflow [88]. One key difference between Redis' and RabbitMQ's pub/sub implementations is how they deal with the discrepancy between the rate of data production and consumption. When data is produced faster than it is consumed by the subscribers, RabbitMQ uses a credit-based flow control mechanism (similar to RocketBufs), which slows down the production rate of the producers [82]. Redis, on the other hand, does not implement such a mechanism. Instead, if a subscriber can not keep up with the rate of data production, the Redis broker simply closes the connection to that subscriber [85]. This mechanism allows Redis to avoid the overhead that would be incurred to implement flow control, however it results in possible data losses (when a subscriber connection is dropped). In Chapter 4, we show how the difference in the support or lack of support for

flow control impacts the performance of these systems as well as our pub/sub system built using RocketBufs (RBMQ).

2.2.4 Live Streaming Video

A number of studies have looked at improving the performance and efficiency of live streaming video services. As discussed in Section 2.1.4, Twitch selects certain video streams for transcoding in order to reduce deployment costs. Pires et al. [77] examine strategies for selecting video streams to be bitrate-transcoded, in order to reduce video data bandwidth and minimize overall resource consumption. While the proposed strategies focus on bandwidth reduction rather than networking efficiency, they would also result in reduced CPU utilization on delivery servers thanks to reduced dissemination traffic. Similarly, work by He et al. [48] presents a technique for the efficient allocation of video transcoding servers in the cloud. Their work attempts to maximize viewer satisfaction while minimizing the costs associated with deploying transcoding servers.

Netflix, a video-on-demand streaming service, has introduced TLS encryption into the FreeBSD kernel [97] to help reduce the CPU utilization of TLS-encrypted HTTP connections (HTTPS). While this is done in the context of a video-on-demand service, similar benefits could also be obtained for live streaming. We believe that the approaches proposed by previous work are valuable and are complimentary to our goal of achieving better CPU efficiency. A live streaming video system built using RocketBufs could leverage these techniques to further reduce resource consumption and deployment costs.

In this thesis, we build and examine a live streaming video application. We also implement a similar system that uses Redis for video data dissemination to use as a point of comparison. We use Redis because we found (as described in Chapter 4) that it has good performance and provides significantly higher throughput than RabbitMQ. Additionally, Redis is used by the WILSP platform [89] to disseminate live streaming video data recorded by producer nodes to delivery hosts running web servers. WILSP is designed primarily for interactive video, and WILSP web servers have been shown to support up to 50 users, delivering a total video throughput of about 230 Mbps [89]. Our live streaming system is designed for larger scale both in terms of video throughput and number of users. For example, we conduct live streaming video experiments with a total viewer throughput of up to 22 Gbps.

Chapter 3

Design and Implementation

3.1 Overview

As discussed in Chapter 1, there are two main issues that have made it difficult for existing MOM systems to utilize RDMA for high-performance messaging: the complexity of the RDMA programming interface and the differences between the RDMA and the socket application programming interfaces. While the `rsocket` APIs provide a socket-like interface for RDMA networking, using them for MOM systems would result in reduced performance when compared with an application built to take full advantage of the native RDMA APIs. To address these issues, we have designed RocketBufs, a software framework for the construction of MOM systems with the following goals:

- The framework should provide natural abstractions and easy-to-use APIs, suited for developing a variety of MOM systems. As discussed in Chapter 2, the independence between data producers and consumers is a key property of the MOM systems' workflows, and this property therefore must be enabled by the framework. Specifically, publishers in the system should be able to continuously produce data to the message brokers, and under normal operating circumstances (i.e., without back-pressure) they should not be blocked while previously produced data is being delivered. Analogously, subscribers in the system should be able to subscribe to and continuously receive new data.
- The framework's abstractions and APIs should be agnostic to the transport protocol. Changing the transport protocol should require only changes to the system's configuration and should not require any modifications to application code.

- The framework should also enable efficient and scalable MOM system implementations using either TCP or RDMA networking.

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 buffer data, while `rIn` class is used by consumers to subscribe to and receive buffer data. In MOM systems, publishers (which predominantly send data) use `rOut` objects into which data is placed and sent to brokers, while subscribers (which predominantly receive data) use `rIn` objects to receive and process messages. Message brokers use both `rIn` and `rOut` objects for data ingestion and dissemination. The framework also provides support for buffer splicing (discussed in detail in Section 3.5) to allow message brokers to efficiently forward data. Figure 3.1 shows an example overview of how these objects could be used in an MOM system. Using `rIn` and `rOut` objects, applications can implement flexible topologies and partitioning logic for scaling out the MOM system.

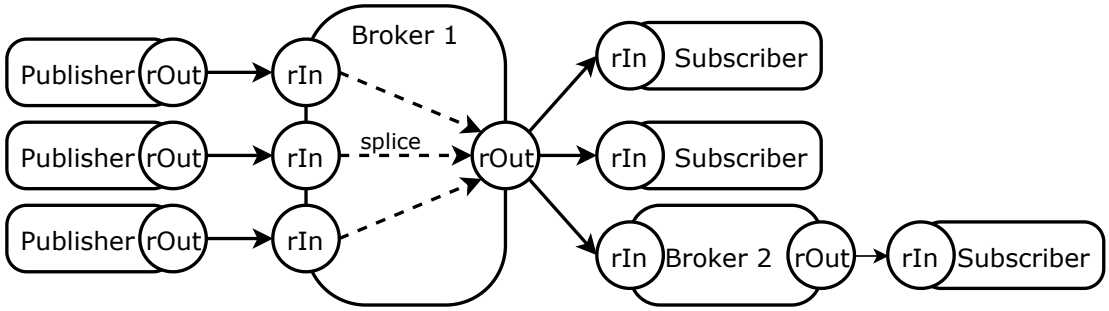

Figure 3.1: Use of `rIn` and `rOut` objects in an MOM system.

As discussed in Chapter 2, MOM systems typically operate using one of two main types of workflows: message-queuing or publish/subscribe. RocketBufs is designed to support both workflows. In this thesis, we focus on and implement RocketBufs within the context of a publish/subscribe system, and discuss work required in the future to support message-queuing in Chapter 6. Note that, some MOM systems such as Wormhole [92] distinguish

the roles of *producers* and *publishers*, referring to entities where data originates from and those that broadcast the data update events, respectively. For RocketBufs, we use the terms *producers* or *publishers* interchangeably to refer to entities that send data to the brokers, and *subscribers* or *consumers* to refer to nodes that receive data from the brokers (either within a message-queuing or publish/subscribe context).

3.2 Buffers

A buffer is a region of memory provided by RocketBufs for transmitting and receiving 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, which stores input or output data. 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. This design is also tailored to enable efficient data transfer with both RDMA (where direct access to memory is required) and socket-based communication (such as TCP). Data in a buffer represents messages in a FIFO message queue. The circular buffer provides a bounded space for communication between `rIn` and `rOut` objects, which is not only necessary for utilizing RDMA, but also useful for implementing flow control between buffers (discussed in Section 3.4).

Figure 3.2 depicts the structure of a buffer. 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 (RNIC) cache resources when RDMA is used [33]. 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 to 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 data an application requests a *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 received data in the busy region is consumed and processed by the application. We describe the details of the interface for producing and consuming data in Section 3.3.

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 of the buffer, applications are required to work with memory segments that may not be contiguous. For

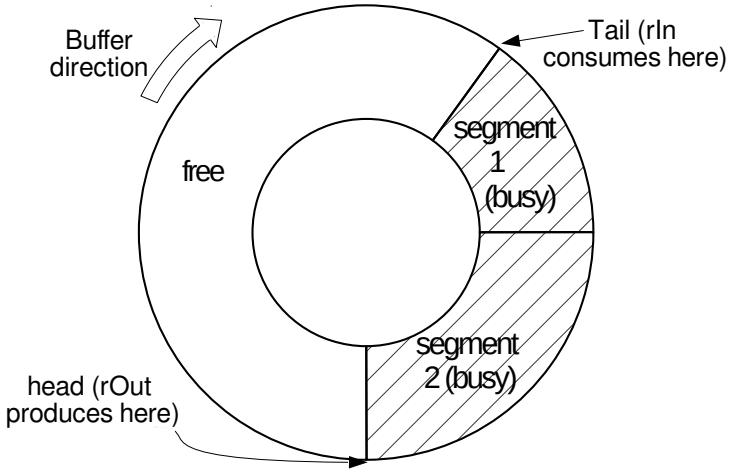

Figure 3.2: Circular buffer.

this reason RocketBufs defines a structure `buf_segment`, which utilizes either one `iovec` (if the entire segment is contiguous) or two `iovecs` (if the segment is split in the buffer due to wrap around). The definition of a `buf_segment` can be seen in Listing 3.1. The `vecs` field is an array of `iovecs` that contains either one or two `iovecs`, indicated in the `vec_count` field. Examples of these two types of segments are shown in Figure 3.3, where the shaded areas in each example represent the segment.

Listing 3.1: Definition of `buf_segment`

```
struct buf_segment {                                          1
  struct iovec* vecs; // array of iovecs                      2
  int vec_count; // number of iovecs (at most 2)              3
};                                                            4
```

3.3 `rIn` and `rOut`

RocketBufs provides two classes, `rIn` and `rOut`, for manipulating and accessing buffer data. 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 modelled after the publish/subscribe pattern, where a publisher broadcasts data to one or more

Figure 3.3: Segment representation using `iovec`.

subscriber(s). If a subscriber application needs to subscribe to data from multiple sources, it can do so by using multiple `rIn` instances. Connections between `rIn` and `rOut` objects can be established by using the `Listener` and `Connection` classes, which are also provided by the framework (we describe how connections are created, used and managed in more detail in Section 3.6). After initializing `rIn` and `rOut` instances, applications can use the methods exposed by these classes (shown in Listing 3.2) 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 both RDMA and TCP.

Listing 3.2: Key `rIn`/`rOut` methods

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

// rIn
void subscribe(bid_t bid, size_t size);
void set_data_arrived_cb(void (*cb)(bid_t, buf_segment));
void data_consumed(bid_t buf_id, size_t size);
...
```

Before being able to send and receive messages, applications need to create and sub-

scribe 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. RocketBufs 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, and 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 keeps track of pending data to be delivered to that buffer. This allows the application to continue producing data into the output 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_segment` 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_segment` and the `size` as arguments. Optionally, an application-defined context can also be provided, which is used with the callback function executed upon completion of the delivery. The `size` argument specifies the amount of data to be transferred, which could 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 3.4.

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

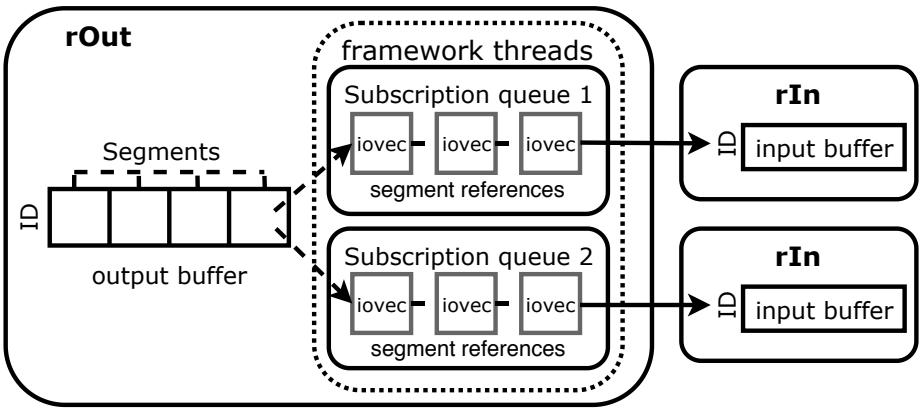

Figure 3.4: Overview of `rOut` buffer management and data transfer.

`rOut::set_data_delivered_cb`.

RocketBufs does not provide any batching mechanisms by default but rather allows applications to implement and control batching. An application can implement message 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_arrived_cb` to register a callback function that executes 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_segment`) to the data itself. The `rIn` class also provides an `rIn::get_buf_ref` method which returns a reference (`buf_segment`) to all the received data of a buffer. This allows the application to access the data in the buffer without copying.

By default, an `rOut` object disseminates data segments to all subscribed `rIn` objects. This behaviour is natural for implementing the publish/subscribe workflow, which we include in our implementation and evaluation for this thesis. In future work, to support the message-queuing workflow, the RocketBufs' API can be expanded to allow setting a delivery *policy* for buffers. Policies would indicate how memory segments are distributed among subscribed `rIn` objects, for example, in a round-robin, first-available, or priority-based fashion.

3.4 Buffer Flow Control

An important consideration for RocketBufs is how to make buffer space available for use and re-use for disseminated data. In MOM systems, messages are typically delivered to subscribers using a FIFO message queue, and subscribers consume messages in the order they are delivered. 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 (i.e., when it no longer needs that data and the buffer space can be reused by the framework). The `rIn` class provides the `rIn::data_consumed` method for this purpose. It takes the buffer identifier and the size of the consumed data as arguments. When this method is called, flow control messages are also implicitly sent by the `rIn` object to the corresponding `rOut` object to update the local buffer metadata (the free and busy portion). Typically, the `rIn::data_consumed` method is called when the application has finished processing a message in the FIFO message queue. However, in certain scenarios where the application requires access to old messages, calls to `rIn::data_consumed` could be delayed until access to the oldest message(s) is no longer needed. In future work, we plan to expand this functionality to enable applications to selectively discard messages in the message queue (by specifying the `buf_segment`), allowing the buffer space to be reused while still 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 in 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 [82]. Redis, on the other hand, does not have such a mechanism and simply closes the connection to a slow subscriber if it can not keep up [85]. While Redis' approach is simple and avoids flow control overhead, it results in possible data losses when back-pressure occurs. Note that, as will be seen later in our evaluation in Chapter 4, when there is no back-pressure, Redis' approach makes it more efficient because it does not need to implement the message exchange required for flow control.

With RocketBufs, we choose to implement flow control within the framework at the buffer level using a credit-based scheme (similar to RabbitMQ). In our scheme, when a remote buffer does not have enough free space, the `rOut` object pauses the transfer of data from the subscription queue to that buffer. Note that data can be continuously added to the output buffer even when data transfer is suspended (as long as the output buffer has enough free space). If the discrepancy between the rate of data production and consumption continues, eventually the output buffer will become full 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`, which will block the calling thread until the requested amount of memory is available and can be returned.

While flow control is required for most types of MOM systems, RocketBufs also provides an option for disabling flow control for certain types of applications (on a per-buffer basis). When flow control is disabled, the framework does not block data transfer and simply overwrites the old segments in the circular buffer. In applications where data most often needs to be produced and consumed at the same rate (such as live streaming video), this option can be useful because it can avoid the overhead required for buffer synchronization and flow control. In Chapter 4, we utilize this option to evaluate and understand the overhead of flow control, and to compare against Redis (which does not have a flow control mechanism). In Chapter 5, we describe how our live streaming application deals with data validity when flow control is disabled.

3.5 Buffer Splicing

RocketBufs's `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. It then uses the `rOut` class to transfer ingested data to other nodes (which could be subscribers or other message brokers). To enable the efficient forwarding of data, RocketBufs implements buffer splicing using an `rOut::splice` method (described later). To help motivate the need for and explain the design of `rOut::splice`, we first provide an example implementation of a message broker using `rIn` and `rOut` classes without using the support for buffer splicing (in Listing 3.3). When in the callback function for incoming data (`in_data_callback`), the broker application requests a segment from the output buffer. The `copy_data` function then copies data from the input segment to the output segment. After that, `rOut::deliver` is called to transfer the data to the subscribers. While this approach is straightforward, it incurs the overhead of copying data between the input and output buffers.

To avoid the copying overhead, the `rOut` class provides an `rOut::splice` method. This method takes a buffer identifier and a reference to an `rIn` object as arguments. An implementation of a message broker using `rOut::splice` can be seen in Listing 3.4, where message forwarding is initialized by iterating over the buffers of the `rIn` object (line 6) and calling `rOut::splice` (line 8). When `rOut::splice` is used, the `rIn` object shares the

Listing 3.3: Pseudocode for a copy-based broker implementation.

```
// established rIn and rOut object                                                    1
rOut out;                                                                             2
rIn in;                                                                               3
                                                                                      4
// register the data arrived callback                                                 5
in.set_data_arrived_cb(in_data_callback);                                             6
                                                                                      7
void in_data_callback(bid_t bid, buf_segment in_seg){                                 8
  size_t data_size = in_seg.vecs[0].iov_len + in_seg.vecs[1].iov_len;                 9
  // request an output buffer segment, block until one is available                  10
  buf_segment out_seg = out.get_output_segment(bid, data_size, BLOCK);                11
  // copy the data to the output segment                                             12
  copy_data(in_seg, out_seg);                                                         13
  // delivery the output segment (no delivery callback is required here)             14
  out.deliver(out_seg, data_size);                                                    15
}                                                                                     16
```

Listing 3.4: Pseudocode for a broker demonstrating the use of `splice`.

```
// established rIn and rOut object                                                    1
rOut out;                                                                             2
rIn in;                                                                               3
                                                                                      4
// iterate over rIn's buffers to initialize data forwarding                           5
for(bid_t buf_id: in.buffers()){                                                      6
  // call rOut::splice once for each buffer                                          7
  out.splice(in, buf_id);                                                             8
}                                                                                     9
```

input buffer memory with the `rOut` object. When data is received, the corresponding buffer segment is added to the appropriate subscription managed by the `rOut` object, 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. Note that, when a buffer is spliced, the framework executes the application-registered callback function (if one is registered) before queuing the data for dissemination. This allows the

broker application to optionally process ingested messages (e.g., to monitor messages or make messages persistent) in addition to forwarding messages.

3.6 RocketNet

RocketNet is RocketBufs's networking layer, which includes framework-managed worker threads responsible for carrying out networking operations (e.g., data transfers) and executing application callbacks. RocketNet threads are subcomponents of `rIn` and `rOut` objects, and are created by the framework when these objects are initialized. Specifically how these threads are configured and implemented depends on the object type (`rIn` or `rOut`) and the transport protocol being used. In this section we describe our TCP and RDMA implementations, however the RocketBufs abstraction allows adding other types of transport protocols to RocketNet in future work.

Our 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 using 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 [67], and is in line with many common messaging protocols [10, 19, 81]. If needed, this limit could be easily changed in the future.

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 [72]. 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 onto the receive queue and waits on the completion queue for notifications of incoming data. For small messages, RocketNet utilizes RDMA inlining for reduced latency [65, 56].

To connect `rIn` and `rOut` objects, RocketNet provides infrastructure to allow for the establishment of network connections (which could be either TCP or RDMA connections). In MOM systems, since a message broker is usually the main point of contact for communicating parties (as publishers and subscribers are not necessarily aware of one another), a broker typically plays the role of a *server* that passively listens for connections. Publishers and subscribers typically play the role of *clients*, which actively connect to the brokers in order to produce or consume data. RocketNet uses the `Connection` class to represent a network link between an `rIn` and an `rOut` object. The `Listener` class can be used by a message broker to listen for and handle incoming connection requests. When a new connection request arrives, the `Listener` object creates a `Connection` object and notifies the broker application of the new connection. From the client side (publisher or subscriber), a connection to a broker is established by constructing a `Connection` object and providing the broker's listening address and the desired protocol (RDMA or TCP) as parameters. Once created, `Connection` objects can be used to initialize `rIn` and `rOut` objects. Listing 3.5 shows some examples of how connections can be established to message brokers. Since an `rOut` object can have connections to multiple `rIn` objects, connections are added using the `rOut::add_conn` method. In Listing 3.5, we show a publisher connecting to multiple message brokers (line 5 and line 6). On the other hand, each `rIn` object has a single connection, and therefore the connection is provided in the constructor (line 9 and line 10 show an example of how a subscriber would connect to a broker). In Chapter 4, we provide a more detailed example of connection establishment and management in an MOM system.

Listing 3.5: Pseudocode for connecting to message brokers.

```
// this is example code used by a publisher to initialize rOut                1
rOut out;                                                                     2
Connection conn1(broker1_address, protocol);                                  3
Connection conn2(broker2_address, protocol);                                  4
out.add_conn(conn1);                                                          5
out.add_conn(conn2);                                                          6
                                                                              7
// this is example code used by a subscriber to initialize rIn               8
Connection conn(broker_address, protocol);                                    9
rIn in(conn);                                                                 10
```

3.7 Sending Control Messages

The `rIn` and `rOut` classes provide methods for sending messages unrelated to application data (referred to as *control* messages). These messages may include heartbeats, acknowledgements and status updates. Both `rIn` and `rOut` classes provide two methods: `send_control_msg`, used for sending control messages; and `set_control_msg_cb`, used for registering a call back function to handle incoming control messages. Framework-specific messages such as those used for flow-control and subscription messages are also communicated using this mechanism, however these are invisible to the application.

The implementation of sending and receiving control messages depends on the networking protocol being used. The main challenge here is to separate these messages from application data. The RDMA implementation addresses this by leveraging the RDMA `send` and `recv` verbs (application data is sent using the `write-with-imm` verb). Each communicating node posts a number of RDMA `recv` work requests onto their work queues. Whenever a control message is received, the receiving node gets a notification from the RDMA completion queue, executes the registered callback, and immediately posts another work request to receive future messages. For TCP, RocketNet simply uses a separate TCP connection between `rOut` and `rIn` objects to send and receive control messages.

3.8 Configurations and Optimizations

RocketBufs exposes a set of configuration parameters that can be used to configure the framework (e.g., TCP/RDMA) and for performance tuning. These parameters can be expressed in several forms: as arguments when creating `rIn` and `rOut` objects, as part of a configuration file, or as arguments to various RocketBufs methods. We now discuss several parameters that may impact messaging performance.

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 in memory 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 when deployed on multi-core systems. When using TCP, applications can set the `tcp_num_threads` parameter to control the number of threads that handle TCP socket communication. 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 [64].

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.

3.9 Chapter Summary

In this chapter we describe the design, implementation and some optimization strategies used in RocketBufs. We show examples of how RocketBufs' asynchronous API can be used to produce and consume data. We also explain how the framework manages buffers and implements a credit-based flow control scheme. Additionally, we describe RocketBufs' support for TCP and RDMA networking, and ways to configure and optimize RocketBufs for performance. In the following chapters, we describe the implementation and evaluation of two applications that we built using RocketBufs: a message-oriented publish/subscribe system and a live streaming video application.

Chapter 4

RBMQ: A Message-Oriented Publish/Subscribe System

4.1 Overview

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). In this chapter, we show how the components of the system, including the message publishers, brokers and subscribers can be implemented with relatively little effort. We also evaluate RBMQ's messaging performance under different transport protocol configurations and compare it to RabbitMQ and Redis, two popular, industry-grade message queuing systems.

An overview of RBMQ is depicted in Figure 4.1. In RBMQ, publishers use `rOut` objects to send messages belonging to specified topics to the message broker. Subscribers obtain messages belonging to a topic by connecting to the broker and using an `rIn` object to subscribe to that topic. Each topic is mapped to a separate buffer. In our prototype, the topic-to-buffer mapping is implemented using a hash table, although a more sophisticated deployment (for example, in a commercial setting) could choose to coordinate this mapping using a distributed key-value store, or other equivalent control schemes. The message broker uses multiple `rIn` objects to ingest messages from publishers and one `rOut` object to disseminate messages to subscribers. It also maintains a `Listener` instance, which listens for connection requests from publishers and subscribers and uses the connections to initialize `rIn`/`rOut` instances. RBMQ messages use a wire format with a 24-byte header which includes the topic name, similar to the frame structure of RabbitMQ messages [81].

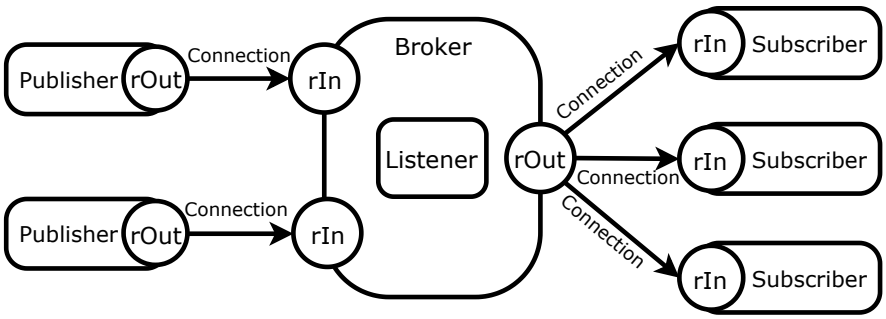

Figure 4.1: Overview of RBMQ implementation using RocketBufs.

4.2 Publisher Implementation

Listing 4.1 shows the pseudocode for key components of an RBMQ publisher. First an `rOut` instance is initialized and connected to the message broker using the `Connection` class, as described in Chapter 3. Before a publisher can start publishing messages belonging to a topic, it must first initialize the topic using the `create_topic` function (line 6). This function creates a local buffer and sends a control message to the broker. This control message contains the topic name and the corresponding buffer identifier. When the broker receives the control message, it calls `rIn::subscribe` to allow it to consume data from the that buffer. The buffer size is controlled by the application and should be tuned depending on the specific workload, as discussed in Chapter 3. The `publish` function (line 16) is used to send messages to the broker. This function calls `topic_to_buf` to obtain the buffer identifier for the topic, requests a memory segment for the output buffer, then places the message in the segment and asks the framework to deliver it (line 21). Placing the message in the 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.

4.3 Broker Implementation

The message broker is the centerpiece of RBMQ, because it is responsible for routing messages from publishers to subscribers. It also acts as the main contact point in the system. Listing 4.2 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 15), the broker uses them to initialize the appropriate `rIn`/`rOut` objects. If a connection request arrives from a publisher

Listing 4.1: Pseudocode for key components of an RBMQ publisher.

```
// connect rOut with the broker                                          1
rOut out;                                                                2
Connection conn(broker_address, protocol);                               3
out.add_conn(conn);                                                      4
                                                                         5
void create_topic (string &topic) {                                      6
  bid_t bid = topic_to_buf(topic);                                       7
  // create a buffer for the topic                                       8
  out.create_buffer(bid, buffer_size);                                   9
  // form the payload of a topic creation message                        10
  iovec topic_msg = create_topic_message(bid, topic);                    11
  // send the topic creation message to broker                           12
  out.send_control_msg(topic_msg);                                       13
}                                                                        14
                                                                         15
void publish (string &topic, size_t msg_size) {                          16
  bid_t bid = topic_to_buf(topic);                                       17
  // request an output buffer segment, block until a segment is returned 18
  buf_segment seg = out.get_output_segment(bid, msg_size, BLOCK);        19
  // copy the message to the memory segment                              20
  place_msg(seg, msg_size, topic);                                       21
  // signal the framework to transfer the message data                   22
  out.deliver(seg, msg_size, NO_CTX);                                    23
}                                                                        24
```

(line 16), 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 20), 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 26).

When the broker receives a topic-creation control message from a publisher (line 31), 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 by in the `publisher_data_cb` function, which is registered as the `rIn`'s callback function.

Listing 4.2: Pseudocode for key components of an RBMQ broker.

```
// listen for connections from publishers and subscribers
Listener listener(broker_address, protocol);
listener.set_conn_cb(on_conn);

// rOut object to disseminate to subscribers
rOut out;

// callback function to handle ingest data
void publisher_data_cb (bid_t bid, buf_segment data) {
  if (!verify_topic_mapping(bid, data))
    throw exception(“invalid mapping”);
  // because rOut::splice is used, nothing else needs to be done
}

void on_conn (Conn &conn) {
  if (from_publisher(conn)) {
    // create the rIn instance for this publisher connection
    rIn *in = new rIn(conn);
    // set callback function to handle control messages from the publisher
    in.set_control_msg_cb(std::bind(on_ctrl_msg, in));
    // set callback function to handle application data from the publisher
    in.set_data_arrived_cb(publisher_data_cb);
  }
  // add the subscriber connection to the rOut object
  if (from_subscriber(conn))
    out.add_conn(conn);
}

void on_ctrl_msg (rIn *in, iovec msg) {
  // check if the message is for creating a new topic
  if (is_topic_creation(msg)) {
    // retrieve the buffer id
    bid_t bid = bid_from_msg(msg);
    // subscribe to receive data from the publisher
    in.subscribe(bid, buffer_size);
    // deliver data to subscribers
    out.splice(*in, bid);
  }
}
```

4.4 Subscriber Implementation

Listing 4.3 shows the pseudocode for key components of an RBMQ subscriber. The `rIn` instance of the subscriber is initialized by establishing a connection to the broker. The `subscribe_topic` function is used to subscribe to a topic, which is done by obtaining the corresponding buffer identifier and calling `rIn::subscribe` with that buffer (line 7). The `data_cb` function is registered as the callback function to handle incoming data (line 11). When new data arrives, the subscriber processes the data (line 15) and informs the framework of the amount of data consumed. In our current prototype, we assume messages are consumed in the same order they are published. This assumption fits the FIFO message queue design described in Chapter 3. We plan to implement and evaluate more sophisticated consumption schemes in future work.

Listing 4.3: Pseudocode for key components of an RBMQ subscriber.

```
// connect to the broker                                                    1
Connection conn(broker_address, protocol);                                  2
rIn in(conn);                                                               3
                                                                            4
void subscribe_topic (string &topic) {                                      5
  bid_t buf_id = topic_to_buf(topic);                                       6
  in.subscribe(buf_id, buffer_size);                                        7
}                                                                           8
                                                                            9
// set callback function to handle incoming data                            10
in.set_data_arrived_cb(data_cb);                                            11
                                                                            12
void data_cb (bid_t bid, buf_segment data) {                                13
  // process the data and return the amount of processed data               14
  size_t consumed = process_data(data);                                     15
  // signal the framework to free the buffer space                          16
  in.data_consumed(data, consumed);                                         17
}                                                                           18
```

4.5 Evaluation

We conduct a series of experiments to evaluate the performance of RBMQ in terms of throughput, latency, and CPU utilization. We compare different configurations of RBMQ with two open-source MOM systems, namely RabbitMQ and Redis. These systems are used by many large-scale Internet applications and services, including Reddit [35], Twitter, Github and StackOverflow [88].

4.5.1 Methodology

In this section, we describe the general setup and hardware used in our experiments. The specific benchmarks and their results are described in the following sections. In our experiments, we use one host to run the message broker processes, and separate hosts to run the publisher and subscriber processes. Each publisher process contains several publisher threads, which publish messages belonging to different topics. Each subscriber process subscribes to all topics and the message brokers disseminate messages to all subscribers. To better understand the networking performance of the pub/sub systems, in our benchmarks, the subscribers do not perform any processing on disseminated data. For RBMQ, buffer sizes and threading parameters are tuned for individual experiments (as discussed in Chapter 3). 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 several different RBMQ transport protocol configurations: publisher-to-broker and broker-to-subscribers over TCP (denoted as *RB-tcp-tcp*); publisher-to-broker over TCP and broker-to-subscribers over RDMA (denoted as *RB-tcp-rdma*); and publisher-to-broker and broker-to-subscribers over RDMA (*RB-rdma-rdma*).

For RabbitMQ, we use the `rabbitmq-c` client library [13] to implement the publishers and subscribers. RabbitMQ offers an option where subscribers send an *acknowledgement* message to the broker upon receiving a message. This option allows the broker to inform the publisher about the completion of message delivery, however it incurs extra communication costs compared to Redis and RBMQ which do not implement this feature. We therefore disable it in all of our experiments for a fair comparison. Our Redis publishers and subscribers are implemented using the Redis-provided client library `hiredis` [83]. 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 [78, 86] 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 [52] 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.

When running experiments on a system using multiple CPU cores and NICs, we find that tuning the interrupt request queues (IRQ) of network devices is also important for performance. Our IRQ tuning includes: disabling the Linux `irqbalance` daemon (which is enabled by default in most current Linux distributions); and binding each device queue to a single CPU core, while distributing the number of device queues evenly among cores (note that a network interface can have multiple queues). This ensures that interrupts generated by a queue are handled by the same CPU core, while also allowing all cores to be used for interrupt handling.

4.5.2 Broker Message Throughput

In this experiment, we measure the maximum message throughput that can be achieved by a message broker host in terms of number of messages delivered per second (mps). We configure the publishers to constantly send messages to the broker by calling the `publish` function in a tight loop. To find the maximum throughput values, we increase the number of publisher threads (and therefore the number of topics) until the throughput stops increasing. This corresponds to the point at which, depending on the experiment, either the broker node's CPU is close to 100% utilization or the NIC is saturated. In our experiments, we find that using up to 20 publisher threads (running on one or more hosts

depending on the experiment) allows all target systems to reach these points. Additionally, for this experiment, we also benchmark RBMQ using TCP (for all communication) with flow control disabled (denoted as *RB-tcp-no-fc*). Note that, this configuration may not be useful for delivering messages in real-world pub/sub systems, because it could result in buffer data being overwritten by the framework. We include it as a theoretical point of comparison in order to understand the overhead of flow control.

We run a series of benchmarks and record the results for varying message sizes and numbers of subscribers. The message rate for each subscriber is measured by counting the number of received messages within a period of time. Figures 4.2, 4.4, 4.6, and 4.8 show the average message throughput (in messages per second) when disseminating to zero, one, two and four subscribers respectively. These graphs include 95% confidence intervals (obtained from throughput samples obtained during the experiments). Figures 4.3, 4.5, 4.7, and 4.9 show the application-level goodput (in Gbps) for the same benchmarks.

In the zero-subscriber benchmarks (Figure 4.2 and 4.3), 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 as well as the cost of disseminating data to subscribers. In these results, among TCP-based system, Redis performs better than RB-tcp-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 Chapter 3 and detailed later in our profiling results). This allows Redis to avoid flow control overhead and provide higher throughput. The throughput values for RB-tcp-tcp and RB-tcp-rdma are the same, because in both cases the broker host ingests messages using TCP and there is no subscriber. The RB-rdma-rdma configuration achieves the highest ingest throughput, which is up to 1.7 times higher than Redis, up to 2.0 times higher than RB-tcp-tcp and up to 2.3 times higher than RabbitMQ. It also saturates the NIC with the smallest message size (4 KB), compared to all other systems (8 KB for Redis, 8 KB for RB-tcp-tcp and 32 KB for RabbitMQ). Additionally, because RDMA does not involve the main CPU in data transfers, the ingest throughput for RB-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 RB-tcp-tcp and 76% for RabbitMQ.

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

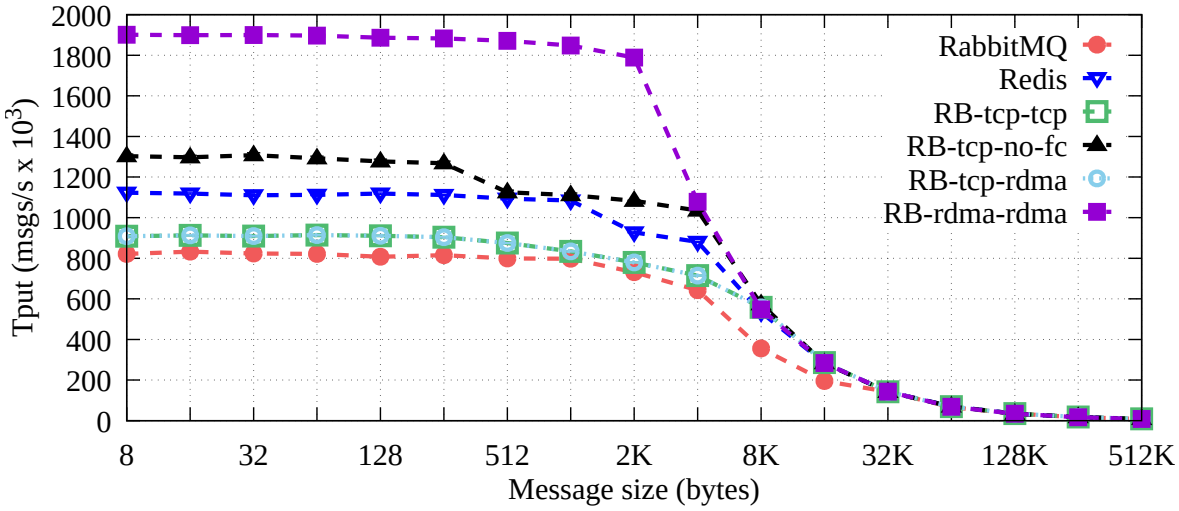

Figure 4.2: Throughput with zero subscribers.

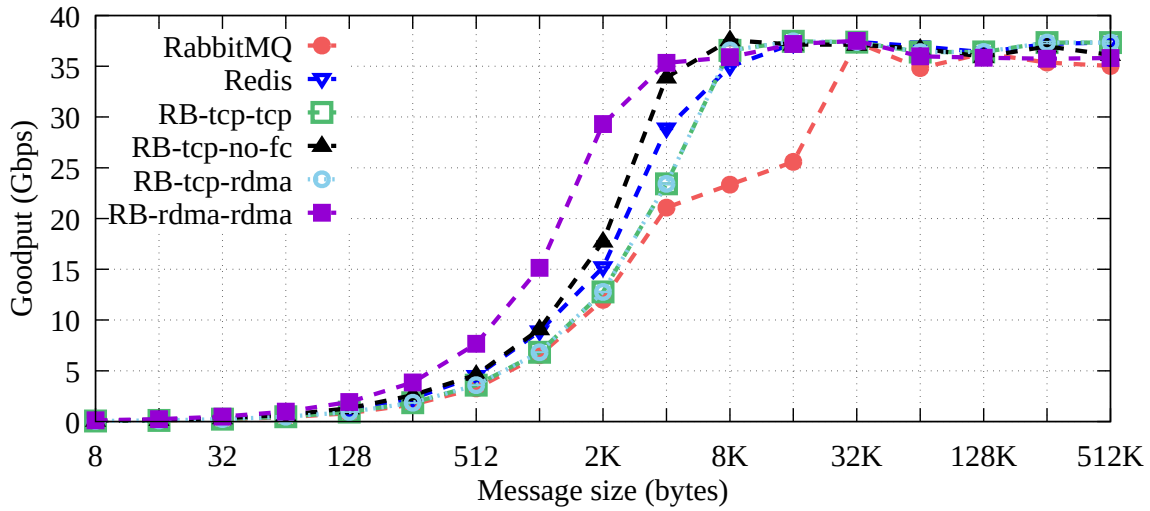

Figure 4.3: Goodput with zero subscribers.

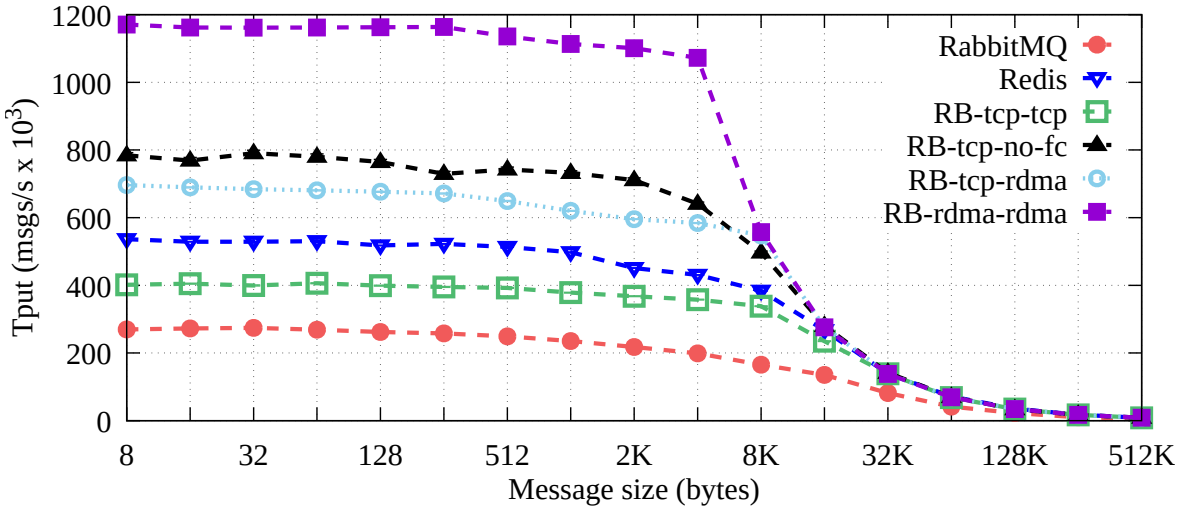

Figure 4.4: Throughput with 1 subscriber.

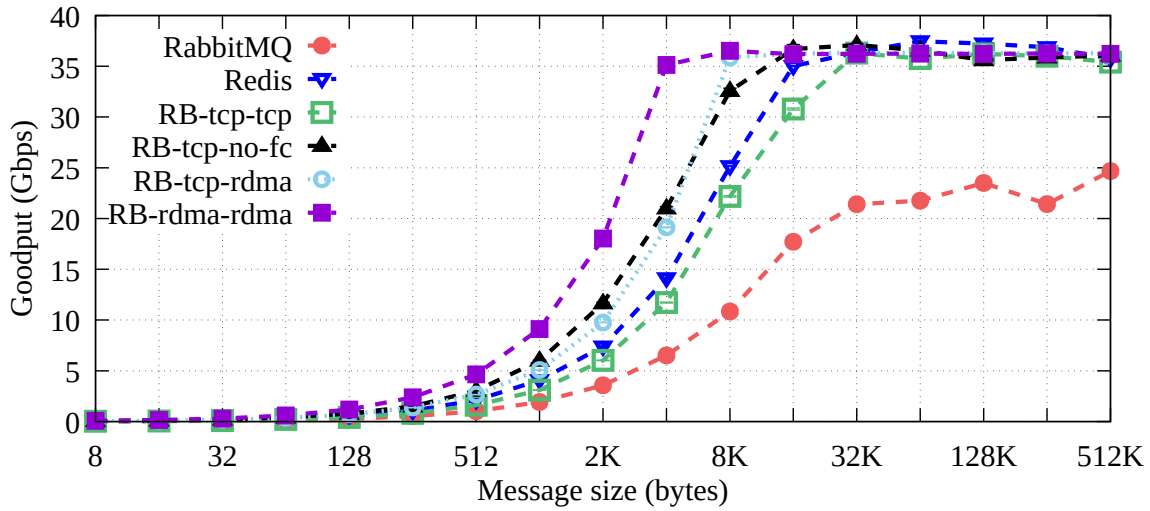

Figure 4.5: Goodput with 1 subscriber.

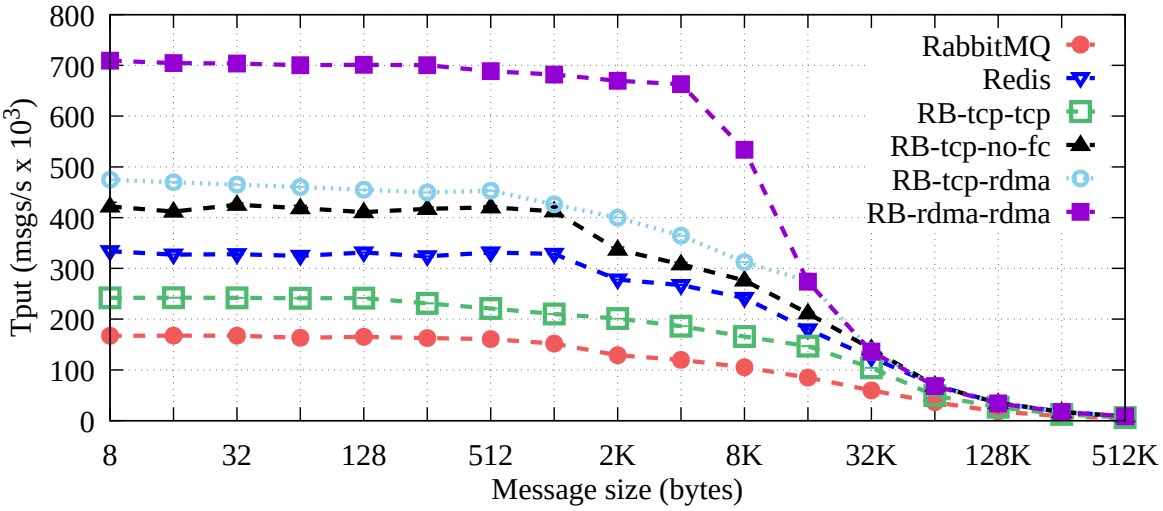

Figure 4.6: Throughput with 2 subscribers.

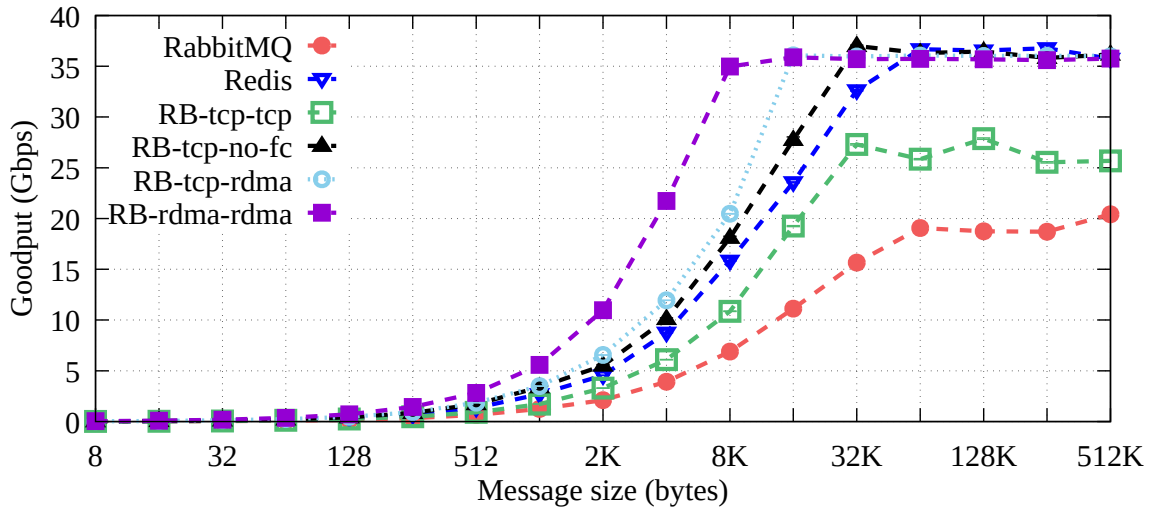

Figure 4.7: Goodput with 2 subscribers.

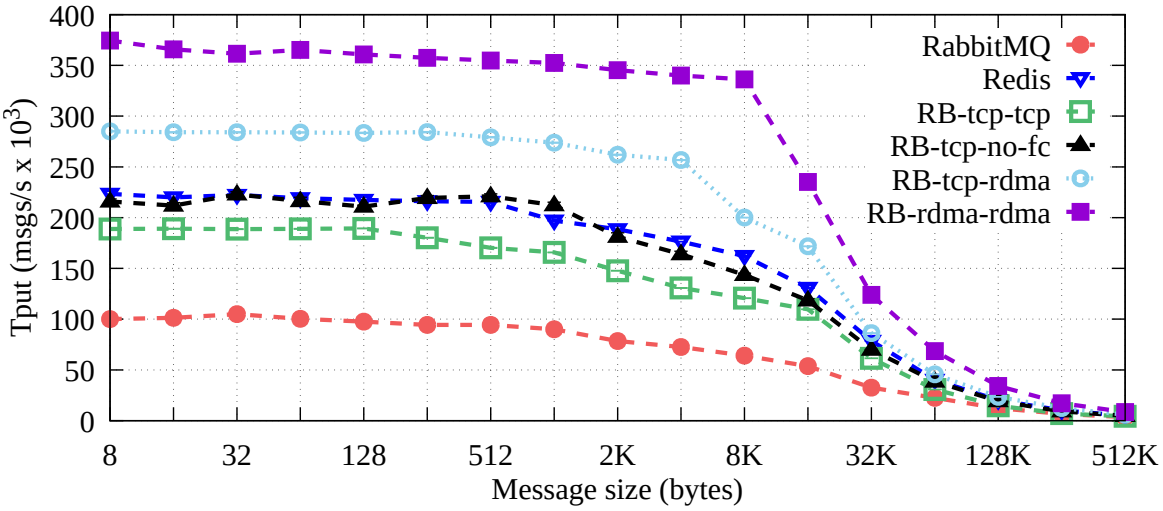

Figure 4.8: Throughput with 4 subscribers.

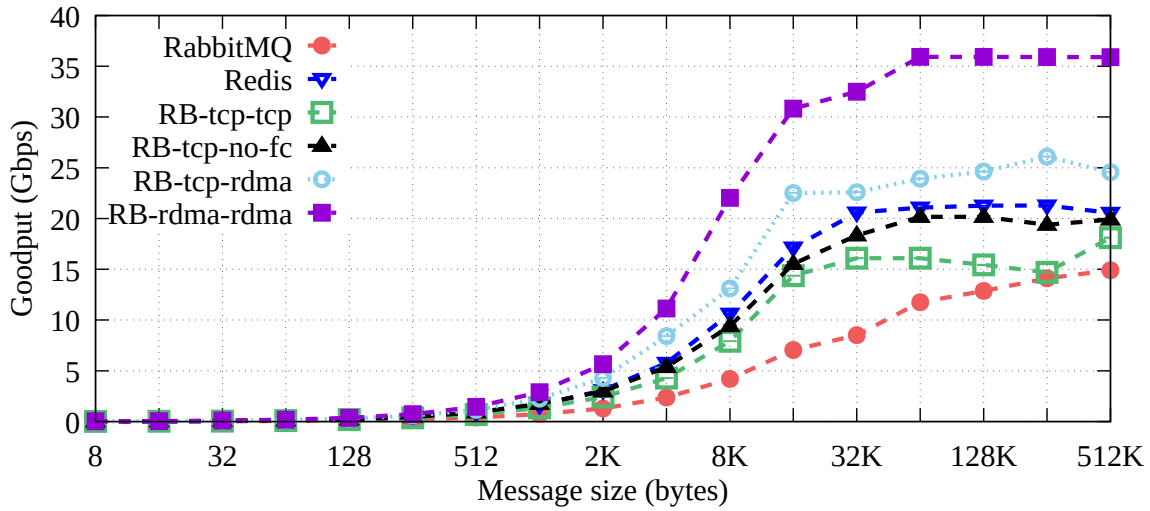

Figure 4.9: Goodput with 4 subscribers.

messages to four subscribers, the messaging throughput of RB-tcp-tcp and Redis is up to 1.9 times and 2.2 times higher than RabbitMQ, respectively. Secondly, Redis performs better than RB-tcp-tcp. With two subscribers, the Redis broker host is able to saturate the NICs' bandwidth (for message sizes of 64 KB and larger), while RB-tcp-tcp is not, due to CPU saturation. In the four-subscriber case, Redis's throughput is up to 18% higher than RB-tcp-tcp. Finally, leveraging RDMA with RBMQ yields substantial throughput improvements for all numbers of subscribers evaluated. Even when RDMA is only used for broker-to-subscriber communication (*RB-tcp-rdma*), throughput with four subscribers can be up to 2.8 times higher than RabbitMQ and up to 1.3 times higher than Redis. When RDMA is used for all communication (RB-rdma-rdma), messaging throughput with four subscribers is up to 3.7 times that of RabbitMQ, 2.0 times higher than RB-tcp-tcp and 1.7 times higher than Redis. The RB-rdma-rdma configuration is also the only system that is able to fully saturate all 40 Gbps NICs.

The main responsibility of a broker is to disseminate copies of published data to other nodes (e.g., subscribers or other brokers) to allow the data to be processed on several hosts. Therefore, we are interested in evaluating the RBMQ broker's ability to scale as the number of subscribers increases. Figure 4.10 shows the average messaging throughput of the systems when transferring 32-byte messages (which is close to the average size of a Tweet [76]) with varying numbers of subscribers. The average throughput values are normalized with the zero-subscribers case. We observe that, as the number of subscribers increases, RB-tcp-rdma shows the lowest rate of decline while RabbitMQ shows the highest rate of decline in terms of messaging throughput. With four subscribers, the normalized throughput of RB-tcp-rdma is 31%, whereas it is 12% for RabbitMQ. Redis and RB-tcp-tcp have relatively similar rates of decline. Their normalized throughput with four subscribers are 20% and 21%, respectively. The RB-rdma-rdma configuration scales better from zero subscribers to one subscriber when compared with Redis and RB-tcp-tcp: the normalized throughput with one subscriber is 61% of that with zero-subscribers, compared to 48% for Redis and 44% for RB-tcp-tcp. However, as the number of subscribers increases from one to four, RB-tcp-tcp and Redis show slightly better scalability. For example, Redis' normalized throughput with four subscriber drops by 58% compared to the one-subscriber case, while RB-rdma-rdma's throughput drops by 67%. With four subscribers, RB-rdma-rdma's normalized throughput is 20% of that with zero-subscribers, which is similar to Redis and RB-tcp-tcp. In future work, we plan to explore techniques to improve RocketBufs's ability to scale to a larger number of subscribers. However, it is important to note that, even with four subscribers, RB-rdma-rdma's throughput is still higher than Redis' throughput with two subscribers, demonstrating the benefits of leveraging RDMA with RocketBufs.

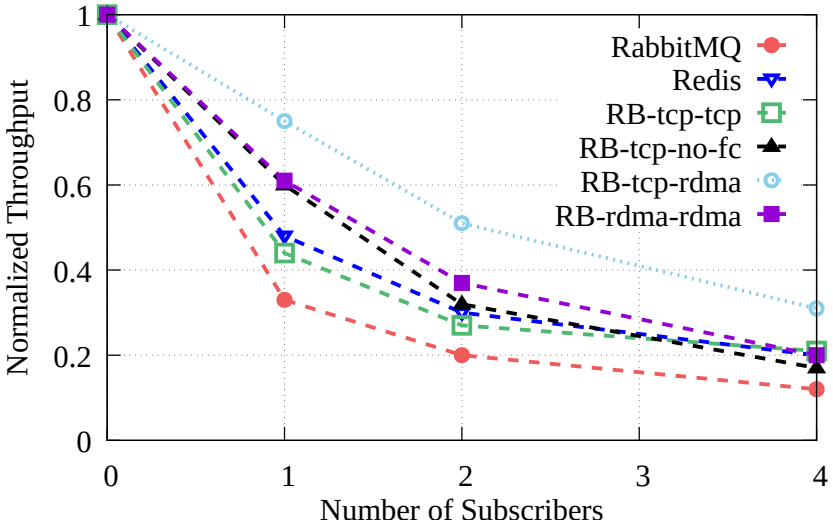

Figure 4.10: Normalized throughput with zero-subscribers (32-byte messages).

In order to understand the reasons for the differences in performance among the systems evaluated, we use `perf` [63] to profile the message broker host. Each pub/sub system is subjected to a load of 50,000 messages per second with 8 KB messages and four subscribers. This rate and message size are chosen to be close to the peak performance of RabbitMQ, which allows us to examine the systems operating under a relatively high load while handling the same amount of data. The profiling data is analyzed and summarized in Table 4.1. The rows in the table show the percentages of CPU time each system spends in various parts of the system, which includes the Linux kernel (vmlinux), the NIC driver (mlx4), the threading library (libpthread), and the user-level application functions (application). All remaining functions fall into the "others" category. The "utilization" row shows the average system-wide CPU utilization of each system obtained from `vmstat`.

Overall, RabbitMQ has the highest CPU utilization (94.0%) among all systems, followed by RB-tcp-tcp (63.0%), Redis (55.3%), RB-tcp-no-fc (52.3%), RB-tcp-rdma (29.2%), and RB-rdma-rdma (9.6%). These statistics show the amount of CPU required by the evaluated systems to service the same load. Clearly, systems requiring less CPU for this load are able to obtain higher peak loads. For example, RB-rdma-rdma requires the least amount of CPU (9.6%) while handling the load of 50,000 mps, due to the fact that RDMA does not require the main CPUs to be involved in data transfers. As a result, RB-rdma-rdma achieves the highest maximum throughput with higher loads. The profiling results also show that RabbitMQ spends much more CPU time executing application-level code

(71.5%) compared to Redis and all RBMQ configurations, 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). 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 messaging throughput.

To explain RB-tcp-tcp's higher CPU utilization compared to Redis' (63.0% versus 55.3%), we perform further profiling and find that roughly 10% of RB-tcp-tcp's CPU time is spent on functions that send and handle flow control messages (`send_control_msg` and `control_msg_cb`). This overhead does not exist for Redis, since it does not implement a flow control scheme. Redis' lower CPU utilization however comes at the cost of lacking a mechanism to deal with back-pressure (as discussed in Chapter 3). When flow control is disabled, RBMQ using TCP (RB-tcp-no-fc) produces similar to significantly higher messaging throughput than Redis (Figures 4.2, 4.4, 4.6, 4.8). In Chapter 5, we show how our live streaming application takes advantage of RocketBufs' flexibility regarding flow control to reduce CPU utilization.

System	**RabbitMQ**	**Redis**	**RB-tcp-tcp**	**RB-tcp-no-fc**	**RB-tcp-rdma**	**RB-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 4.1: CPU utilization statistics from the broker under a load of 50,000 mps with 8 KB messages and four subscribers. For example, the CPU utilization of RB-rdma-rdma is 9.6%, of which 73.7% is spent in the Linux kernel.

4.5.3 Subscriber CPU Utilization

In most MOM deployments, subscribers are responsible for processing received messages. For example, they may serve the messages through a web server to clients in video streaming applications [89], process the data in data analytics applications [7], or perform verifications (such as in a Hyperledger blockchain [12]). Minimizing the CPU overhead associated with receiving and managing messages is therefore critical.

We run a series of microbenchmarks to measure the CPU overhead on a subscriber host while receiving large volumes of data. To do this, we use 10 publishers to send messages to the broker host. Each publisher sends messages at a rate of 5,000 mps for a total of 50,000 mps. This rate is chosen to ensure that all compared systems can send and receive large messages (up to 32 KB in size) at the same rate. We compare the Redis and RabbitMQ subscribers with three different configurations for the RBMQ subscriber: using `rIn` with TCP (*RB-tcp*); using `rIn` with RDMA (*RB-rdma*); and using `rIn` with TCP but with flow control disabled (*RB-tcp-no-fc*).

We measure CPU utilization on a subscriber host receiving messages of various sizes. Figure 4.11 shows the average subscriber CPU utilization with 95% confidence intervals, obtained from `vmstat` samples collected every second. Two important trends can be observed here. First, the CPU utilization of TCP-based MOM systems increases noticeably with larger messages (due to the copying overhead associated with TCP). We can see that overall, the Redis subscriber has the highest CPU utilization. For 32 KB messages (which corresponds to a message throughput of 13.1 Gbps), Redis utilizes 26% of the CPU, whereas the utilization with RabbitMQ is 22% and utilization with RB-tcp-tcp is 17%. On the other hand, the RBMQ subscriber using RDMA yields negligible CPU overhead regardless of the message size.

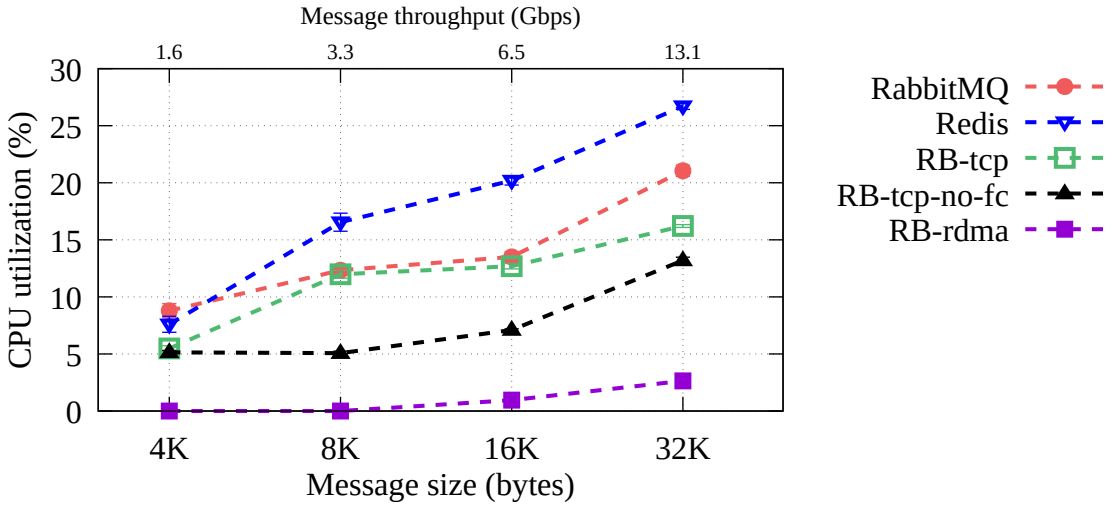

Figure 4.11: Subscriber CPU utilization (50,000 mps).

To understand where CPU resources are spent, we perform profiling on the subscriber host while receiving 32 KB messages (corresponding to a throughput of 13.1 Gbps). Ta-

ble 4.2 shows the three functions requiring the most CPU time in each system. We can see that, for TCP-based systems, the CPU is busy copying bytes from socket buffers to user-space (using `libc_recv` or `libc_read`), monitoring sockets for events (using `epoll_wait` or `poll`), and handling message data. Redis spends a significant portion of its CPU time copying data between user-space buffers (using `mem_cpy`). Examining the source code of the `hiredis` client library reveals that it internally copies message data to another buffer before passing that buffer to the application. This copying overhead explains Redis' higher CPU utilization compared to RabbitMQ and RB-tcp. We also see that, when flow control is enabled, a noticeable amount of CPU time is spent on sending flow control messages in RBMQ configurations (12.4% for RB-tcp and 10.2% for RB-rdma). However, their overall CPU utilization is still lower than Redis and RabbitMQ. When flow control is not required (as will be shown Chapter 5), disabling it further reduces CPU utilization (RB-tcp-no-fc).

Finally, for RB-rdma, most of the CPU time is spent handling RDMA completion events since the CPU is not involved in data transfers. Note that for RB-rdma, `epoll_wait` and `libc_read` are used for monitoring and reading completion event data from the RDMA completion channel's file descriptor, and not application data from sockets. As a result, the CPU utilization of RB-rdma remains negligible regardless of the message size.

RabbitMQ	**Redis**
libc_recv(50.0%)	libc_read(36.4%)
amqp_handle_input(15.5%)	epoll_ctl(18.4%)
poll(11.7%)	mem_cpy(17.1%)

RB-tcp	**RB-tcp-no-fc**	**RB-rdma**
libc_recv(77.9%)	libc_recv(88.0%)	epoll_wait(42.0%)
send_control_message(12.4%)	epoll_wait(8.8%)	libc_read(19.5%)
epoll_wait(6.5%)	rIn::do_cb(0.7%)	send_control_message(10.2%)

Table 4.2: Profiling statistics of a subscriber (50,000 mps, 32 KB messages).

4.5.4 Delivery Latencies

We also conduct experiments to measure delivery latency (i.e., how long it takes for a message to travel from the publisher to a subscriber). To avoid comparing timestamps generated on different machines, we use an ECHO-based method (also used in previous work [55]). With this method, a sending node reads the local clock, puts the timestamp in a message, and sends the message to the topic "ping" on the broker. An echo node

subscribes to "ping" and, upon receiving a message, relays the received message to topic "pong" on the same broker. The sending node subscribes to the topic "pong" and, upon receiving a "pong" message, calculates the difference between the current time and the time in the pong message. This calculated value represents the time required for a full round-trip, roughly twice the delivery latency. For simplicity and accuracy we report the actual full round-trip numbers. Publisher-to-subscriber numbers would be roughly half of these values. Note that we do not include the RBMQ configuration without flow control in these experiments, to ensure that the timestamps in the buffer are protected and not overwritten by the framework.

The average latency of an MOM system will vary depending on the load it encounters. We compare the latency of systems handling 100,000 mps and 200,000 mps (these values were chosen because a fair comparison requires all systems to have the same message rates and RabbitMQ is not able to support higher rates with 1 subscriber). Because the messages are essentially bi-directional we configure the sending node to send messages at half the target rate (since the echo node replies at the same rate) and record the round-trip latency of individual messages. To ensure that all systems can achieve the target message rates, we use 10 "ping" topics and 10 corresponding "pong" topics. The size of ping-pong messages is chosen to be 34 bytes (excluding the message header size), which is the average size of a Tweet [76].

The CDFs of the round-trip latencies are shown in Figure 4.12 and 4.13. The x-axis represents log-scaled round-trip latency in microseconds. There are three important takeaways from the latency results. First, Redis and both RBMQ configurations are able to achieve lower latency than RabbitMQ. For the 100,000 mps workload, the median round-trip latency is 422 microseconds for RabbitMQ, 187 microseconds for RB-tcp-tcp, and 149 microseconds for Redis. For the 200,000 mps workload, the median latencies are 519 microseconds, 241 microseconds and 225 microseconds for RabbitMQ, RB-tcp-tcp and Redis, respectively. Secondly, RB-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 RB-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 RB-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, RB-rdma-rdma yields a maximum round-trip latency of 4,640 microseconds for the 200,000 mps workload, despite having very low overall latencies (the 90^{th} percentile latency is 299 microseconds). Similarly, for the 200,000 mps workload Redis yields a maximum round-trip latency of 17,844 microseconds, while its 90^{th} percentile latency is 366 microseconds.

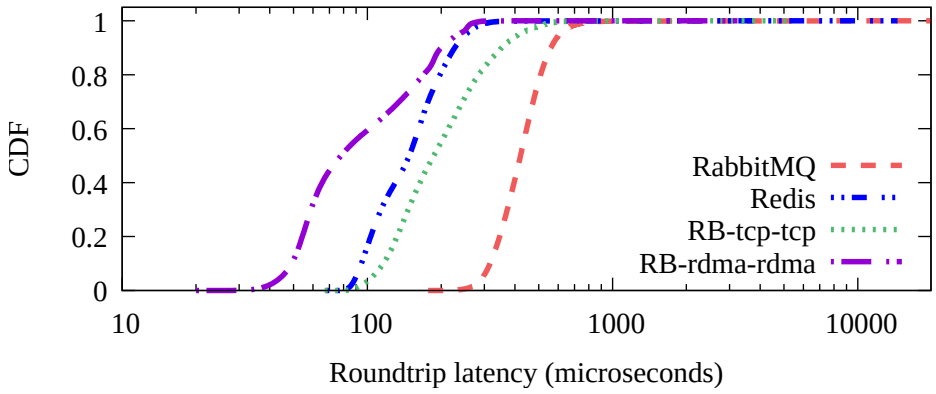

Figure 4.12: Full round-trip latency at 100,000 mps.

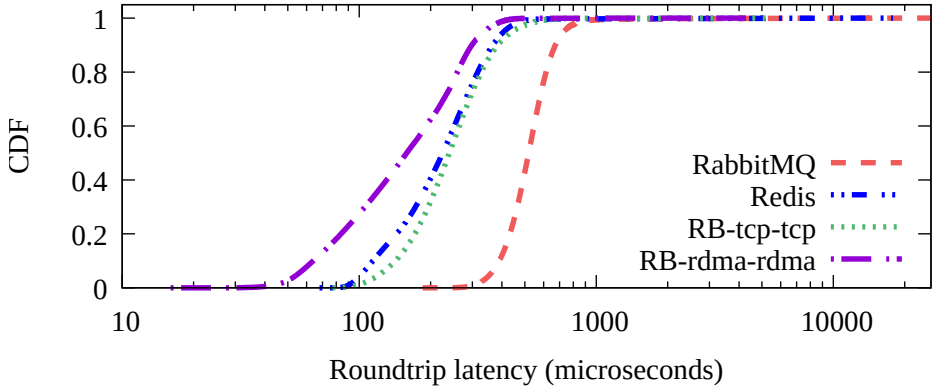

Figure 4.13: Full round-trip latency at 200,000 mps.

As described in Chapter 3, when using RDMA, RocketBufs detects incoming data by monitoring events on the RDMA completion channel's file descriptor using `epoll`. Since `epoll` puts the calling thread to sleep while awaiting events, it allows the system's CPU to idle when no event is being handled. However, because `epoll` is a system call, it incurs extra latency when events arrive due to context switching. With RocketBufs, we have experimented with a technique that does not use `epoll` and instead uses dedicated threads to actively poll the RDMA completion queues. This technique is used by several in-memory key-value store systems [61, 69, 22]. Using this approach allows RBMQ to further reduce delivery latencies. For example, the median latency at 200,000 mps using a polling approach is only 47 microseconds, which is 70% faster than RB-rdma-rdma and

80% faster than Redis. This low latency however comes at the cost of fully utilizing a CPU core per RNIC to poll the RDMA completion queues, even when no data is being transferred. For higher loads, more than one CPU core per RNIC may be required for polling RDMA completion queues. Thus, tuning for the best combination of throughput and latency would require determining the right number of CPUs devoted to polling for RDMA completion events and the number of CPUs to dedicate to the application. In future work, we plan to further explore the best approaches regarding these trade-offs, allowing applications to configure RocketBufs to suit their specific workloads.

4.6 Chapter Summary

In this chapter we describe how we use RocketBufs to implement RBMQ, a message-oriented publish/subscribe system. We compare RBMQ's performance in terms of messaging throughput, delivery latency and CPU utilization against RabbitMQ and Redis, two popular, industry-grade message queuing systems. Our evaluations show that RBMQ using TCP performs well when compared with these two systems. When RBMQ is configured to use RDMA, it achieves significant gains in performance, demonstrating RocketBufs' ability to leverage hardware features to construct high-performance MOM systems. In the next chapter, we further demonstrate the flexibility of the framework by using it to implement a live streaming video application.

Chapter 5

Live Streaming Video Application

In order to further demonstrate the general utility of the RocketBufs framework, in this chapter we present an application that we have designed and implemented to manage the dissemination and delivery of live streaming video. In Chapter 2, we discuss how live streaming services like Twitch handle large amounts of video traffic and disseminate them to many delivery servers within and across data centers in order to meet global demands. Deploying software to efficiently deal with such a workload is a challenging task. Streaming services must either implement custom software solutions, which is difficult and time-consuming, or make use of existing software systems which are not necessarily optimized for live streaming workload (as will be shown later in our evaluation). In this chapter, we show that by utilizing RocketBufs' networking capabilities, services can easily implement efficient video data dissemination within a data center. Here, we focus on video dissemination within a data center in order to highlight the benefits of leveraging RDMA, however RocketBufs can also be used for cross-data center dissemination using the TCP layer. We show that, within a data center, even when using TCP, our RocketBufs-based live streaming application uses fewer CPU resources when handling the same amount of video data compared to an equivalent solution using Redis. Using RDMA with RocketBufs further reduces CPU utilization, allowing the delivery web servers to support up to 55% more concurrent viewers than when using Redis.

5.1 Design

The general design of the system is depicted in Figure 5.1. Producers represent video stream sources which generate video data, each associated with a unique live stream.

They send this video data in chunks over TLS-encrypted TCP connections (which is done to emulate real services [29]) to a dissemination host. We emulate streaming sources using a purpose-written application. The dissemination host acts as a broker, which ingests video data from streamers and replicates the streams to multiple delivery web server processes (subscribers) on separate hosts. In our prototype, the dissemination and delivery processes run on separate hosts, but in a real deployment they might also reside on the same host, in which case network communication is avoided. The web server processes run a modified version of the `userver`, an open-source web server that has been shown to provide good performance [21, 75]. They act as subscribers which integrate with RocketBufs to subscribe to and receive video streams from the dissemination host. In a real-world deployment, dissemination corresponds to the inter-host replication required by entities like Twitch [104], and YouTube Live [108] to meet the scalability and latency demands of their services. Some delivery servers would be geo-distributed, but in many cases they also reside within the same data center or IXP when demand for popular content requires it [30]. Finally, viewers request video data they want to watch from the delivery servers. These viewers are emulated using `httperf` [71], and make requests to the `userver` over TLS-encrypted HTTP (HTTPS), as is the case with popular services such as YouTube Live [109].

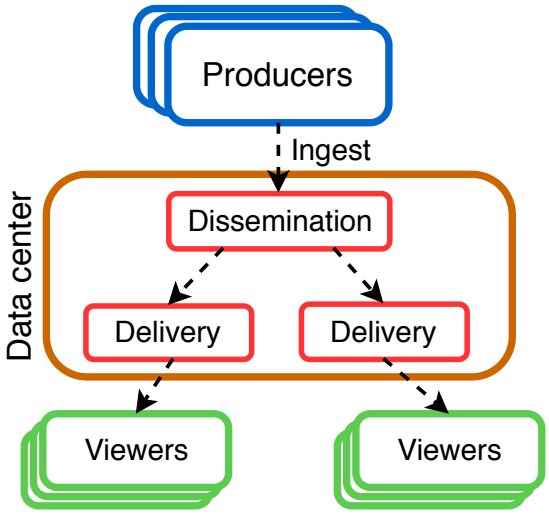

Figure 5.1: Design overview of a live streaming video application using RocketBufs.

Listing 5.1 shows the pseudocode for the dissemination process. This process uses one `rOut` instance to disseminate all live streams to the delivery servers. When a streaming session starts (i.e., a streamer connects to the dissemination host), the dissemination process creates a buffer and uses a thread to handle video data ingestion from the

stream's producer. When video data from the producer arrives, the dissemination process requests an output buffer segment using `rOut::get_output_segment` (line 9). It then calls `producer_place_data` (line 11), which in turn calls the `SSL_read` function of the OpenSSL library [74] which decrypts the ingested data and places it into the segment. After that, `rOut::deliver` is called to transfer this data to the delivery servers. Note that in this case, unlike RBMQ's implementation described in Chapter 4, we do not use `rOut::splice` because TLS decryption is required for ingested data and our current RocketBufs prototype does not provide support for encrypted communication. In future work, we plan to add support for encrypted communication to RocketBufs, allowing `rOut::splice` to be used to simplify the construction of the dissemination process.

Listing 5.1: Pseudocode for a dissemination process.

```
// initialized rOut object                                                          1
rOut out;                                                                           2
                                                                                    3
// the thread that handle the stream                                                4
while(stream_online(stream_id)){                                                    5
  // block and wait for notification of incoming data from producer                 6
  producer_await_notification(stream_id);                                           7
  // request a buffer segment to store the video chunk                              8
  buf_segment chunk = out.get_output_segment(stream_id, chunk_size, NON_BLOCK);     9
  // place the video chunk into the buffer segment                                  10
  producer_place_data(stream_id, chunk);                                            11
  // deliver the video chunk, no callback needed                                    12
  out.deliver(chunk, chunk_size, NO_CTX);                                           13
}                                                                                   14
```

Each delivery host contains one `rIn` instance to subscribe to and receive all live video streams. Data structures are also maintained to manage the disseminated video chunks in the input buffer. In our prototype, each buffer is sized to hold five video chunks. When a new viewer starts watching a video stream, the delivery server sends the video content to that user starting from the middle of the buffer, which allows the viewer's device to pre-fill its playback buffer. Given the nature of live streaming, 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. 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, 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. When the systems are not overloaded, the delivery servers send video data at the same rate at which it is generated and disseminated, and since only the oldest chunk can be overwritten with new data at any point in time, no viewer data is overwritten. However, if the system is overloaded (e.g., due to hardware under-provisioning), buffer data can be overwritten while being sent to viewers. When such a scenario is detected, we record it as an error and terminate the viewing session. A real-world deployment would implement a load balancing mechanism to direct the viewer to a more lightly loaded server.

As a point of comparison, we implement a version of our live streaming application using Redis [84] for data dissemination. We choose Redis because it is a high-performance in-memory data store used by many web services (e.g., Pinterest [20], Twitter, Github and StackOverflow [88]), and because it has also been used in a live streaming video platform in previous work [89]. We utilize Redis' publish/subscribe features to disseminate video from producers to delivery web servers. Our evaluation in Chapter 4 shows that Redis significantly outperforms RabbitMQ, in part due to its lack of a flow control scheme. In a live streaming workload, flow control is not necessary since ideally video data is produced and consumed at the same rate. This makes Redis a suitable system for live streaming video dissemination.

Our Redis-based live streaming video system has a similar design to the RocketBufs-based system. Producers send video data to the Redis dissemination host, which in turn disseminates this data to delivery hosts. The delivery hosts act as subscribers and use a modified version of the `userver` to deliver video to emulated viewers. Our Redis-based dissemination host runs multiple Redis broker processes and we load-balance streams between those processes to fully utilize available CPUs (since Redis processes are single-threaded). Connections between producers and the dissemination host are secured by using `stunnel`, per Redis' recommendation [87]. Recall that this is done to simulate real-world services [29] where streamers broadcast their live video content over TLS-encrypted connections. We integrate Redis with the `userver` using the Redis-provided client library `hiredis` [83] to subscribe to and receive video streams from the dissemination host. The `hiredis` library allows the application to handle disseminated data by registering callback functions, similar to RocketBufs. However, unlike RocketBufs, the memory used by `hiredis` to store data is deallocated after the callback function returns. Because the callback function is executed in a event loop, this memory can not be maintained for long since the callback function needs to return in order for new data to be received. Therefore, our `userver` implementation must copy this data to another managed buffer, which is necessary for

maintaining the data for delivery. We show in our evaluation how RocketBufs API design helps reduce CPU utilization on the delivery servers compared to `hiredis` by avoiding this type of data copy.

5.2 Evaluation

We run a series of experiments to measure each system's ability to disseminate and deliver live streaming video. In our setup, the dissemination process resides on a big host (described in Section 4.5), and the web server-integrated subscribers reside on regular hosts. We use a special host with an Intel Xeon E5-2697v3 CPU (with 14 real cores and hyper-threading disabled), 128 GB of RAM and two 40 Gbps NICs to run multiple producers that send video data to the dissemination host. Data is continually published and moved between nodes in 500 KB chunks, each representing 2 seconds worth of video data, and the buffer size is set to store 10 seconds worth of video data (5 chunks). In each experiment, we vary the number of producers to examine the performance of the systems as the load from producers increases. These experiments are repeated using RocketBufs with both TCP-based and RDMA-based dissemination (denoted as *RB-tcp* and *RB-rdma*), as well as the Redis-based system. Each experiment consists of a 60 second warmup period, followed by a 120 second measurement period. For each experiment, we report the mean of the results along with 95% confidence intervals. Note that in the graphs below, the confidence intervals are typically small relative to the sizes of the data points, and therefore in most cases they are not visible.

5.2.1 Microbenchmarks

We first perform a number of microbenchmarks to determine the maximum number of streams that the system can handle for a varying number of delivery servers that receive disseminated video data. We also observe the impact handling this data has on the CPUs of the delivery web servers. These microbenchmarks do not include emulated viewers, thus allowing us to isolate the impact of the dissemination process. Emulated viewers are included later in our delivery server benchmarks (Section 5.2.2).

Dissemination host throughput: Figure 5.2 shows the maximum achievable ingest throughput of the dissemination host while disseminating to varying numbers of delivery servers. The results and confidence intervals are obtained from five repetitions of the benchmarks. To determine the maximum throughput, we gradually increase the number

of producing streams until the dissemination host fails to keep pace with the rate at which data is produced. This corresponds to the point at which the dissemination host's CPU is close to 100% utilization, which is mainly due to overhead from decrypting ingest data and disseminating it to delivery servers. Figure 5.2 shows that as the number of delivery servers changes, RB-tcp achieves throughput values largely comparable to Redis. The throughput of both systems decreases with the increasing number of delivery servers at largely the same rate. Using RDMA yields a significant boost in maximum throughput, which remains relatively consistent even when disseminating to an increasing number of delivery servers, since RDMA incurs little CPU overhead regardless of the amount of data transferred. With 8 delivery servers, Redis and RB-tcp achieve 10.0 Gbps and 10.4 Gbps of video throughput respectively, while RB-rdma can handle up to 18.0 Gbps, representing a 73% improvement versus Redis. The throughput of RB-rdma is limited mostly by the overhead required to decrypt ingested data.

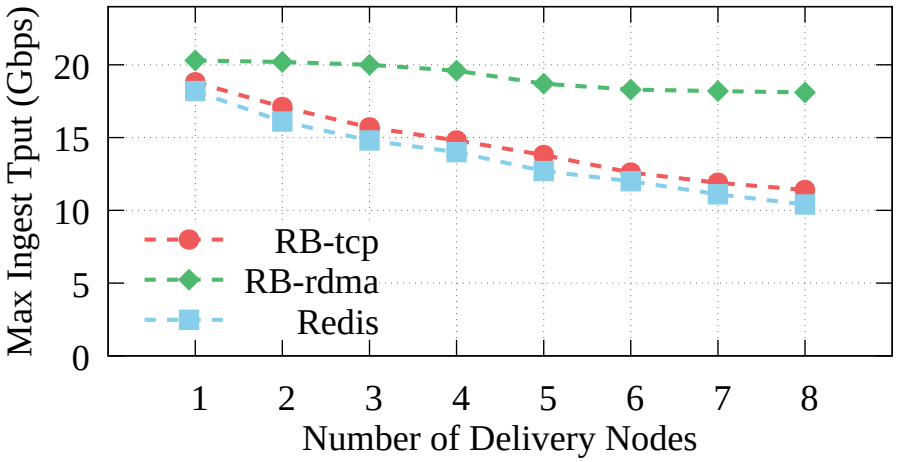

Figure 5.2: Maximum ingest throughput.

Delivery server CPU utilization: As discussed in Section 2.1.4, video delivery servers consume CPU to deliver video data to viewers. Therefore, we are interested in efficiently receiving video data from the dissemination host so that CPU resources can be conserved for delivery (where outgoing viewer data needs to be encrypted). Figure 5.3 shows the average CPU utilization with 95% confidence intervals (obtained from `vmstat` samples collected every second) on a delivery server as the amount of disseminated data increases. The results show that all RocketBufs-based systems require significantly less CPU on the delivery hosts when compared to Redis. For example, at 32 Gbps of incoming data, the

CPU utilization of Redis is approximately 50%, whereas it is 23% for RB-tcp and 3% for RB-rdma, amounting to relative reductions of 54% and 95% respectively. We also observe that while the CPU utilization of the TCP-based systems (RB-tcp and Redis) rises with increasing throughput, the utilization of *RB-rdma* remains negligible (3% or less) regardless of the amount of disseminated data.

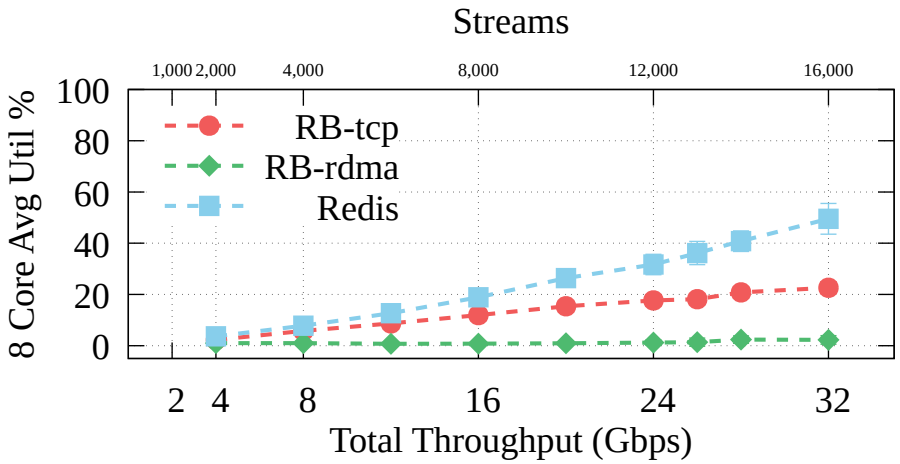

Figure 5.3: Delivery server CPU utilization as total stream throughput increases.

To understand where CPU resources are spent, we profile the delivery server running with these three configurations. Table 5.1 shows the three functions requiring the most CPU time for a delivery server receiving video data at 32 Gbps. We observe that the TCP-based systems (Redis and RB-tcp) spend a significant portion of time reading data from the socket buffer (`libc_read` and `libc_recv`). The Redis-based delivery server also spends about 34.7% of CPU time copying data between different memory locations (`mem_cpy`). This include `hiredis`' internal copying (as discussed in Chapter 4) and the copying of data to the web server managed buffer for delivery. RocketBufs' APIs do not require such copying, since they allow direct access to the input buffer, thus consuming fewer CPU resources. For RB-rdma, the main CPU is not involved in data transfers except for detecting and managing the RDMA completion events, which includes monitoring and reading data from the completion channel's file descriptor (`epoll_wait` and `libc_read`). As a result, RB-rdma incurs negligible CPU overhead, regardless of the amount of disseminated data. In the next section, we show how the CPU savings achieved with RocketBufs allow the delivery servers to support significantly more concurrent viewers.

	Redis	RB-tcp	RB-rdma
1st	mem_cpy (34.7%)	libc_recv (92.4%)	epoll_wait (51.6%)
2nd	libc_read (34.3%)	epoll_wait (5.8%)	libc_read (15.5%)
3rd	epoll_ctl (16.9%)	rIn::do_cb (0.5%)	pthread_spin_lock (1.8%)
total	85.9%	98.7%	68.9%

Table 5.1: Top three delivery server functions with most CPU time at 32 Gbps.

5.2.2 Live Streaming Video Delivery Benchmarks

We now run a set of benchmarks to measure a delivery host's capacity to deliver live streaming video using the `userver`. For these experiments, we use `httperf` [71] to simulate large numbers of viewers and to generate load on the web servers over secure (TLS) connections. Each simulated viewer connects to a server and requests video content from a specific stream at the same rate as it is produced (2 Mbps).

To simulate WAN effects for connections between viewers and delivery servers, we use `tc` [4] on the hosts running emulated viewers to add bandwidth limits and network delays to the in-lab local area network. This approach is similar to that used in existing work [99]. Each connection is subject to a simulated delay of 50 ms (for both ingress and egress traffic), which is approximately the coast-to-coast transmission delay in the United States [17]. We assign bandwidth limits to viewer connections following the distribution shown in Table 5.2. This distribution approximates the connection speeds of client computers in the United States to Akamai servers, according to the Akamai Connectivity Report [3]. We use a minimum bandwidth limit of 3 Mbps, which is higher than the streams' bitrate (2 Mbps) to ensure that all emulated clients can keep up with the video streams. Eight regular hosts are used to run emulated viewers to ensure that they do not become the bottlenecks. We also modify `httperf` to include checks for data timeliness (timeout) and validity.

Reported Rate	Reported Share	Used Rate	Used Share
Above 25 Mbps	21.0%	50.0 Mbps	25.0%
15 - 25 Mbps	31.0%	20.0 Mbps	25.0%
10 - 15 Mbps	15.0%	12.5 Mbps	12.5%
4 - 10 Mbps	23.0%	6.0 Mbps	25.0%
Below 4 Mbps	10.0%	3.0 Mbps	12.5%

Table 5.2: Akamai reported and emulated viewers access speeds.

We increase the number of emulated viewers until the delivery server's capacity is

exceeded, which corresponds to the point at which the CPUs are saturated. When this occurs, video data is not delivered to viewers in a timely manner and client requests time out. We record this as an error and terminate the viewer session. For varying numbers of produced streams (measured in terms of total throughput), we conduct a sequence of benchmarks to determine the maximum number of viewers that a delivery server can support without errors, which is reported using the total amount of viewer throughput. The benchmarks are repeated five times and we report their means along with 95% confidence intervals.

Figure 5.4 shows the results of these benchmarks. As discussed in Section 5.2.1, both RB-tcp and RB-rdma systems incur less CPU overhead when compared with the Redis-based system, allowing more viewers to be supported. This difference grows as the amount of disseminated data (number of video streams) increases. For 20 Gbps of incoming disseminated 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 of viewer throughput, a relative increase of 27%. When RDMA is used (RB-rdma), the CPU overhead associated with receiving video data is negligible, regardless of the dissemination throughput. As a result, RB-rdma's viewer throughput remains relatively consistent and only drops from 21.7 Gbps (with 2 Gbps of incoming disseminated data) to 21.0 Gbps (with 20 Gbps of incoming disseminated data). This amounts to an improvement of up to 55% when compared with Redis.

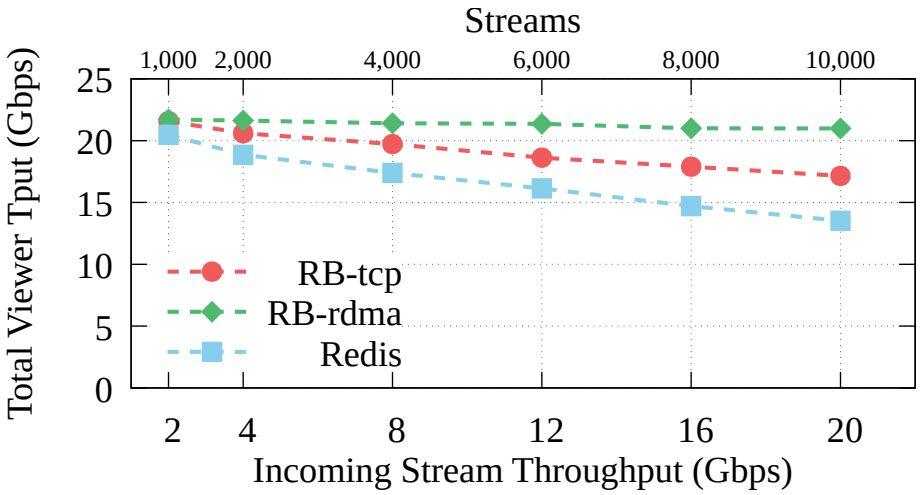

Figure 5.4: Maximum error-free web server delivery throughput.

5.3 Chapter Summary

In this chapter we further demonstrate the flexibility of RocketBufs by using it to implement a live streaming video application. We describe how the dissemination process integrates with and utilizes the framework's networking capabilities to manage live streaming video dissemination. We also modify a web server to obtain and deliver disseminated live streaming video to viewers. Our benchmarks show that when using TCP, RocketBufs provides similar to better performance than Redis, an equivalent industry-grade messaging system. However, when RDMA is utilized, RocketBufs is able to support up to 73% higher ingest traffic on the dissemination host and up to 55% more concurrent viewers on the delivery server. These performance improvements demonstrate RocketBufs' ability to make more efficient use of hardware to help scale live streaming video distribution and delivery in order to meet increasing demands.

Chapter 6

Conclusions and Future Work

6.1 Thesis Summary

Message-Oriented Middleware systems are a class of software that facilitates independent and loosely-coupled messaging in distributed and event-based systems. In recent years, as new types of applications emerge, many new MOM systems have been developed, both as open-source projects and as services offered by cloud providers. In many cases, applications using MOM systems are deployed in data centers and have strict performance requirements regarding throughput and latency. Modern kernel-bypass technologies such as Remote Direct Memory Access (RDMA) offer a solution for achieving high-throughput, low-latency and CPU-efficient data transfers. Unfortunately, existing commonly-used MOM systems do not utilize RDMA for high-performance data center communication. We identify two challenges with implementing RDMA support in MOM systems: the complexity of the RDMA programming interface and the difference between the RDMA abstraction and the socket abstraction. The `rsocket` APIs provide a socket-like interface for RDMA, however they are less efficient than native RDMA APIs due to protocol translation overhead.

These issues have motivated us to design and implement RocketBufs, a framework to facilitate the construction of future high-performance MOM systems. Our goals with RocketBufs in this thesis are: to provide natural abstractions and easy-to-use APIs suited for developing a variety of MOM systems; to enable access to RDMA and TCP (and potentially other transport layer technologies) without requiring code changes to the application; and to provide implementations for both RDMA and TCP-based networking that enable the construction of efficient and scalable MOM systems. To achieve these goals, we have designed memory-based buffer abstractions and APIs that are both suit-

able for building MOM systems and allow for efficient implementations with TCP and RDMA transport protocols. Our framework provides two main classes, `rIn` and `rOut`, for data dissemination between nodes, allowing the construction of flexible MOM topologies. Using RocketBufs' APIs, applications can produce and consume data continuously, while the framework manages buffers and provides mechanisms for flow control. RocketBufs also implements support for buffer splicing, which allows message brokers to forward data efficiently by avoiding unnecessary data copying.

We demonstrate the utility and evaluate the performance of RocketBufs using two applications while executing with RDMA and TCP protocols. The evaluation of our message-oriented publish/subscribe system (RBMQ) shows significant performance gains when compared with Redis and RabbitMQ, two production-quality MOM systems. In comparison with RabbitMQ (a system with a flow control scheme equivalent to RBMQ), both Redis and RBMQ using TCP show significantly higher messaging throughput. Using RDMA with RBMQ yields significant performance gains, allowing RBMQ to achieve up to 1.7 times higher messaging throughput and 47% lower median delivery latency versus Redis. On subscriber hosts, when using RDMA to receive data, RBMQ incurs negligible CPU overhead regardless of the amount of disseminated data tested. This allows CPU resources to be conserved for other purposes like processing data. These results demonstrate RocketBufs' ability to provide abstractions and APIs that are transport-layer agnostic, yet allow for the construction of high-performance MOM systems.

To further demonstrate the flexibility of RocketBufs, we use it to implement a live streaming video application consisting of a dissemination host and multiple delivery hosts. The dissemination host uses the `rOut` class to disseminate video data from video sources to web servers for delivery to viewers. The web servers use the `rIn` class to subscribe to and receive disseminated video data, either using TCP or RDMA, and service video content to viewers over HTTPS. We compare our application to a similar application implemented using Redis and find that a RocketBufs-based dissemination host can disseminate up to 73% higher live streaming throughput, while a RocketBufs-based delivery server is able to support up to 55% more concurrent viewers.

The results of implementing and evaluating RocketBufs suggest that our previously stated goals are met. The framework's `rIn`, `rOut` classes and their APIs allow applications to produce and consume data continuously, and enable the construction of flexible MOM topologies. In Chapter 4 and 5, we have demonstrated the flexibility of RocketBufs by using it to successfully build two applications: a publish/subscribe system and a live streaming video application. We believe that RocketBufs achieves our first goal of facilitating the easy construction of a variety of MOM systems. The second goal of enabling access to RDMA and TCP-based networking using the same interface is also met. Changing the

transport protocol in RocketBufs only requires changes to the system configuration and does not require modifications to application code. Finally, RocketBufs' memory-based abstractions and APIs allow for the implementation of both TCP and RDMA using their native APIs without any protocol translation overhead. The results of our evaluation show that, RocketBufs using TCP performs well when compared with industry-grade systems, while RocketBufs using RDMA achieves significant gains in performance, allowing applications to handle substantially higher loads. For example, when RDMA is used, RocketBufs achieves a higher messaging throughput with four subscribers than Redis' throughput with two subscribers. We therefore conclude that our third goal of enabling efficient and scalable implementations for RDMA and TCP-based networking is also achieved.

6.2 Future Work

In this section, we present ideas for future work to improve RocketBufs' performance and ability to support a wider variety of MOM systems.

6.2.1 Buffer Delivery Policies

While in this thesis we have focused on our implementation and evaluation of RocketBufs within the context of a publish/subscribe workflow (where subscribers receive all messages from a buffer), the framework is also designed to support a message-queuing workflow (where messages are selectively distributed among subscribers). In future work, this support could be realized by implementing delivery policies for buffers. These policies would be used to control how buffer segments are distributed among subscribers. A simple example of a delivery policy is round-robin, where messages (represented as buffer segments) are distributed evenly among subscribed `rIn` objects in a round-robin manner. In other systems, a first-available policy might be more suitable. In this case, data would be transferred from the `rOut` object to the first `rIn` object with enough free buffer space to receive the data. In future work, we plan to implement and evaluate the performance of different delivery policies when handling different types of MOM workloads.

6.2.2 Support for Data Persistence

In this thesis, we have focused on supporting in-memory MOM systems, where messages reside in memory only and thus have a limited lifetime. However in many MOM systems,

it is desirable to make messages persistent. For example, a system with fault-tolerance requirements needs to survive crashes and restarts without losing message data. Therefore, such systems require messages to be kept on persistent storage so that they can be recovered. In another class of MOM systems, often referred to as *log-processing* systems (e.g., Kafka [31], Amazon Kinesis [6]), messages published by producers are kept in log-like data structures on message brokers, which can be queried and processed by consumers. Because consumers can join the system at any point in time and request messages at arbitrary locations in the log, these messages also need to be kept on persistent storage on the brokers.

A possibility for future research is to expand RocketBufs' APIs to support data persistence. The new APIs should allow applications to specify that data needs to be kept on persistent storage and to provide access to persistent data. These APIs should also follow RocketBufs' key design principles to continue to enable easy and efficient access to different types of storage technologies (e.g., hard disk drive, solid-state drive, or persistent memory) using the same interface.

6.2.3 Support for Pull-based Delivery

As discussed in Chapter 2 and Section 6.2.2, in some MOM systems messages are delivered using a pull-based scheme. Examples of such systems are log-processing systems, where messages are kept in persistent storage on the brokers and requested by consumers at a later point in time. In future work, RocketBufs' APIs could be expanded to support pull-based delivery, which would allow subscribers to control the rate of receiving messages. However, as discussed in Chapter 2, pull-based delivery tends to produce higher latencies, and therefore its applicability to different workloads should be carefully examined.

6.2.4 Extending Networking Capabilities

Our current implementation of RocketBufs supports TCP and RDMA as transport protocols. An avenue for future research would be to extend the networking capabilities of the framework. For example, UDP and RDMA multicast could be implemented to reduce outgoing dissemination bandwidth, which would benefit hosts that disseminate data to a large number of brokers and/or subscribers. However, these multicast protocols have their own drawbacks (e.g., they are unreliable and have a limit on the size of a datagram).

Another extension to the framework's networking capabilities could be support for TLS-encrypted TCP communication. This feature would simplify the construction of systems

such as live streaming video (as presented in Chapter 5) where data needs to be transferred securely over a wide area network. Additionally, as discussed in Chapter 2, kernel-bypass techniques have been explored in previous work to improve the performance of traditional protocols like TCP/IP. We believe that RocketBufs could incorporate such techniques into the framework to further improve the performance of its TCP layer. Furthermore, we believe that these features should be implemented in a manner that is invisible to the application, enabling the application to take advantage of the benefits provided by simply changing the system configurations.

Finally, we plan to improve the current RocketBufs prototype to better handle cases where the limits of an RDMA device are exceeded. For example, when the creation of buffers results in a higher number of registered memory regions than supported by the RNIC, the framework currently treats this case as an error and notifies the application of the error. A future improvement to RocketBufs would be to implement a technique that dynamically un-registers/re-registers memory regions to support a higher number of buffers.

6.2.5 Support for Security and Fault Tolerance

Fault tolerance is a common requirement in real-world MOM deployments to deal with scenarios such as a system failure or power outage. In this thesis, we have not designed RocketBufs with fault tolerance in mind. We plan to build upon our current design to add support for this feature in future work. Additionally, the current RocketBufs prototype does not have built-in support for security, but rather leaves security to be implemented by the application. We plan to explore common security mechanisms used by production MOM systems and to consider adding support for such mechanisms to future versions of RocketBufs.

6.3 Concluding Remarks

We believe that, given the continuous development of new MOM systems, there is significant potential to improve their performance by utilizing modern data center networking technologies that offer kernel-bypass features (e.g., RDMA, TCPDirect). In this thesis, we have proposed a framework with intuitive abstractions and APIs to simplify the development of future MOM systems, allowing them to utilize such technologies for high-performance in-data center messaging. Our prototype implementation, which currently

supports TCP and RDMA, demonstrates the benefits of our framework. We envision that RocketBufs and the ongoing innovations in the transport layer will help pave way for the future development of better and more efficient MOM systems.

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: A simple abstraction for remote memory. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 775–787. USENIX, 2018.

[2] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 120–126, New York, NY, USA, 2019. ACM.

[3] Akamai. Q1 2017 state of the Internet connectivity report. https://content.akamai.com/gl-en-pg9135-q1-soti-connectivity.html, 2017. Accessed April 12, 2019.

[4] Werner Almesberger. Linux network traffic control – implementation overview. 1999.

[5] Amazon. Amazon Kinesis Data Firehose. https://aws.amazon.com/kinesis/data-firehose/. Accessed October 8, 2019.

[6] Amazon. Amazon Kinesis Data Streams. https://aws.amazon.com/kinesis/data-streams/. Accessed August 29, 2019.

[7] Amazon. Amazon Serverless Data Processing. https://aws.amazon.com/lambda/data-processing/. Accessed June 23, 2019.

[8] Amazon. Amazon Simple Notification Service. https://aws.amazon.com/sqs/. Accessed August 29, 2019.

[9] Amazon. Amazon Simple Queue Service, ReceiveMessage. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/API_ReceiveMessage.html. Accessed August 29, 2019.

[10] AMQP. OASIS AMQP 1.0 specification. http://www.amqp.org/specification/1.0/amqp-org-download/. Accessed June 24, 2019.

[11] AMQP. Products and success stories. https://www.amqp.org/about/examples. Accessed August 23, 2019.

[12] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM.

[13] Alan Antonuk. rabbitmq-c client library. https://github.com/alanxz/rabbitmq-c/. Accessed June 24, 2019.

[14] Apache. ActiveMQ Artemis. https://activemq.apache.org/components/artemis/. Accessed October 8, 2019.

[15] Apache. Apache RocketMQ. http://rocketmq.apache.org/. Accessed June 25, 2019.

[16] Apache. Pulsar: distributed pub-sub messaging system. https://pulsar.apache.org/. Accessed June 23, 2019.

[17] AT&T. AT&T Global IP Network. https://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html, 2019. Accessed April 12, 2019.

[18] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 63–74. ACM, 2013.

[19] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.

[20] Adam Bloom. Using Redis at Pinterest for billions of relationships. https://blog.pivotal.io/pivotal/case-studies/using-redis-at-pinterest-for-billions-of-relationships, 2013. Accessed April 12, 2019.

[21] Tim Brecht, David Pariag, and Louay Gammo. accept()able strategies for improving web server performance. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 227–240. USENIX, 2004.

[22] Benjamin Cassell, Tyler Szepesi, Bernard Wong, Tim Brecht, Jonathan Ma, and Xiaoyi Liu. Nessie: A decoupled, client-driven, key-value store using RDMA. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(12):3537–3552, 2017.

[23] Chelsio. Ultra-Low Latency with User-Mode Kernel-Bypass using WireDirect and TOE Solution Overview. https://www.chelsio.com/high-frequency-trading/. Accessed October 18, 2019.

[24] C. Chen, H. Jacobsen, and R. Vitenberg. Algorithms based on divide and conquer for topic-based publish/subscribe overlay design. *IEEE/ACM Transactions on Networking*, 24(1):422–436, Feb 2016.

[25] Chen Chen, Yoav Tock, and Sarunas Girdzijauskas. BeaConvey: Co-design of overlay and routing for topic-based publish/subscribe on small-world networks. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems (DEBS 2018)*, 2018.

[26] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Constructing scalable overlays for pub-sub with many topics. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07. ACM.

[27] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Spidercast: A scalable interest-aware overlay for topic-based pub/sub communication. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, pages 14–25. ACM, 2007.

[28] Cisco. Cisco visual networking index: Forecast and trends, 2017–2022 white paper, 2018.

[29] Colin Creitz. 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.

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

[31] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations. In *Proc. International Conference on Distributed and Event-based Systems (DEBS)*, pages 227–238. ACM, 2017.

[32] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *Proc. Conference on SIGCOMM*, pages 362–373. ACM SIGCOMM, 2011.

[33] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. FaRM: Fast remote memory. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414. USENIX, 2014.

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

[35] Jeremy Edberg. Reddit: Lessons learned from mistakes made scaling to 1 billion pageviews a month. http://highscalability.com/blog/2013/8/26/reddit-lessons-learned-from-mistakes-made-scaling-to-1-billi.html. Accessed August 12, 2019.

[36] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

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

[38] Z. Fadika and M. Govindaraju. Delma: Dynamically elastic map-reduce framework for CPU-intensive applications. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 454–463, May 2011.

[39] Apache Software Foundtaion. Apache ActiveMQ. https://activemq.apache.org. Accessed April 12, 2019.

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

[41] J. Gascon-Samson, F. Garcia, B. Kemme, and J. Kienzle. Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 486–496, June 2015.

[42] J. Gascon-Samson, J. Kienzle, and B. Kemme. Multipub: Latency and cost-aware global-scale cloud publish/subscribe. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2075–2082, June 2017.

[43] Raquel Campuzano Godoy. Bitnami RabbitMQ cluster: searching for the maximum performance. https://engineering.bitnami.com/articles/bitnami-rabbitmq-cluster-searching-for-the-maximum-performance.html. Accessed October 12, 2019.

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

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

[46] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D. Russell, Howard Pritchard, and Jeffrey M. Squyres. 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)*, pages 34–39. IEEE, 2015.

[47] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity Ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. ACM.

[48] Qiyun He, Jiangchuan Liu, Chonggang Wang, and Bo Li. Coping with heterogeneous video contributors and viewers in crowdsourced live streaming: A cloud-based approach. *IEEE Transactions on Multimedia*, 18(5):916–928, 2016.

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

[50] IBM. IBM message queue. http://ibm.com/products/mq/.

[51] Intel. Intel Ethernet converged network adapter XL710-QDA2. https://ark.intel.com/content/www/us/en/ark/products/83967/intel-ethernet-converged-network-adapter-xl710-qda2.html. Accessed November 4, 2019.

[52] Iperf. Iperf. https://iperf.fr/. Accessed September 11, 2019.

[53] M. Thomson J. Iyengar. QUIC: A UDP-based multiplexed and secure transport. In *draft-ietf-quic-transport-19, Internet Engineering Task Force draft*, 2019.

[54] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. LIPSIN: Line speed publish/subscribe inter-networking. *SIGCOMM Comput. Commun. Rev.*, 39(4):195–206, August 2009.

[55] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proc. Conference on SIGCOMM*, pages 295–306. ACM SIGCOMM, 2014.

[56] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 437–450. USENIX, 2016.

[57] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, pages 113–125, Berkeley, CA, USA, 2019. USENIX Association.

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

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

[60] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *Proc. International Workshop on Networking Meets Databases (NetDB)*, pages 1–7. ACM, 2011.

[61] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 137–152. ACM, 2017.

[62] LibRDMACM. rsocket. https://github.com/ofiwg/librdmacm/blob/master/docs/rsocket. Accessed September 12, 2019.

[63] Linux. Perf. https://github.com/torvalds/linux/tree/master/tools/perf. Accessed September 12, 2019.

[64] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 1:1–1:16, New York, NY, USA, 2016. ACM.

[65] P. MacArthur and R. D. Russell. A performance study to guide RDMA programming decisions. In *Proc. International Conference on High Performance Computing and Communications (HPCC) and International Conference on Embedded Software and Systems (ICESS)*, pages 778–785. IEEE, 2012.

[66] Mellanox. Mellanox MCX4131A-BCAT ConnectX-4 Network Interface Card. https://store.mellanox.com/products/mellanox-mcx4131a-bcat-connectx-4-lx-en-network-interface-card-40gbe-single-port-qsfp28-pcie3-0-x8-rohs-r6.html. Accessed November 4, 2019.

[67] Mellanox. 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.

[68] Mellanox. RoCEv2 considerations. https://community.mellanox.com/s/article/roce-v2-considerations. Accessed April 12, 2019.

[69] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 103–114. USENIX, 2013.

[70] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 313–326, New York, NY, USA, 2018. ACM.

[71] David Mosberger and Tai Jin. httperf — a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 26(3):31–37, 1998.

[72] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proc. International Conference on Systems and Storage (SYSTOR)*, pages 97–108. ACM, 2019.

[73] Melih Onus and Andréa W. Richa. Minimum maximum-degree publish-subscribe overlay network design. *IEEE/ACM Trans. Netw.*, 19(5):1331–1343, October 2011.

[74] OpenSSL. Openssl. https://www.openssl.org/. Accessed September 20, 2019.

[75] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R. Cheriton. Comparing the performance of web server architectures. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 231–243. ACM, 2007.

[76] Sarah Perez. 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/, 2018. Techcrunch. Accessed June 25, 2019.

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

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

[79] Pivotal. RabbitMQ. https://www.rabbitmq.com/. Accessed April 12, 2019.

[80] Pivotal. RabbitMQ consumers. https://www.rabbitmq.com/consumers.html/. Accessed August 29, 2019.

[81] RabbitMQ. Advanced message queuing protocol specification. https://rabbitmq.com/resources/specs/amqp0-9-1.pdf. Accessed June 25, 2019.

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

[83] Redis. hiredis. https://github.com/redis/hiredis. Accessed April 12, 2019.

[84] Redis. Redis. https://redis.io. Accessed April 12, 2019.

[85] Redis. Redis client handling. https://redis.io/topics/clients. Accessed August 23, 2019.

[86] Redis. Redis documentation. https://redis.io/documentation. Accessed October 6, 2019.

[87] Redis. Securing connections with SSL/TLS. https://docs.redislabs.com/latest/rc/securing-redis-cloud-connections. Accessed July 8, 2019.

[88] Redis. Who's using Redis. https://redis.io/topics/whos-using-redis. Accessed August 23, 2019.

[89] Luis Rodríguez-Gil, Javier García-Zubia, Pablo Orduña, and Diego López-de Ipiña. An open and scalable web-based interactive live-streaming architecture: The WILSP platform. *IEEE Access*, 5:9842–9856, 2017.

[90] Salvatore Sanfilippo. Disque. https://github.com/antirez/disque. Accessed October 8, 2019.

[91] V. Setty, R. Vitenberg, G. Kreitz, G. Urdaneta, and M. v. Steen. Cost-effective resource allocation for deploying pub/sub on cloud. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 555–566, June 2014.

[92] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 351–366, Oakland, CA, 2015. USENIX Association.

[93] Solarflare. Solarflare. https://solarflare.com/. Accessed October 18, 2019.

[94] Solarflare. 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.

[95] Douglas Soo. Twitch engineering: An introduction and overview. https://blog.twitch.tv/twitch-engineering-an-introduction-and-overview-a23917b71a25, 2015. Accessed April 12, 2019.

[96] Spache. Apache OpenWhisk: Open source serverless cloud platform. https://openwhisk.apache.org/. Accessed June 23, 2019.

[97] Randall Stewart, John-Mark Gurney, and Scott Long. Optimizing TLS for high-bandwidth applications in FreeBSD. Technical report, Netflix, 2015.

[98] H. Subramoni, G. Marsh, S. Narravula, Ping Lai, and D. K. Panda. Design and evaluation of benchmarks for financial applications using advanced message queuing protocol (AMQP) over infiniband. In *2008 Workshop on High Performance Computational Finance*, pages 1–8, Nov 2008.

[99] Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. Methodologies for generating HTTP streaming video workloads to evaluate web server performance. In *Proc. International Conference on Systems and Storage (SYSTOR)*, pages 2:1–2:12. ACM, 2012.

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

[101] Y. Teranishi, R. Banno, and T. Akiyama. Scalable and locality-aware distributed topic-based pub/sub messaging for IoT. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7, Dec 2015.

[102] Shin-Yeh Tsai and Yiying Zhang. LITE: kernel RDMA support for data center applications. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 306–324. ACM, 2017.

[103] Hayley Tsukayama. More than 21 million people watched gaming's biggest annual show on Twitch. https://www.washingtonpost.com/news/the-switch/wp/2015/06/29/more-than-21-million-people-watched-gamings-biggest-annual-show-on-twitch/, 2015. Washington Post. Accessed April 12, 2019.

[104] Twitch. Twitch. https://www.twitch.tv/. Accessed April 12, 2019.

[105] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. ACM.

[106] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 87–104. ACM, 2015.

[107] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. Towards zero copy dataflows using RDMA. In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, pages 28–30, New York, NY, USA, 2017. ACM.

[108] YouTube. YouTube Live. https://www.youtube.com/live. Accessed April 12, 2019.

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

[110] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 139–150, New York, NY, USA, 2012. ACM.

[111] ZeroMQ. ZeroMQ, an open-source universal messaging library. http://zeromq.org/. Accessed June 23, 2019.

[112] ZeroMQ. ZeroMQ message transport protocol. https://rfc.zeromq.org/spec:23/ZMTP/. Accessed October 13, 2019.