RocketStreams: A Framework for the Efficient Dissemination of Live Streaming Video

Benjamin Cassell
becassel@uwaterloo.ca
University of Waterloo

Huy Hoang
hdhoang@uwaterloo.ca
University of Waterloo

Tim Brecht
brecht@cs.uwaterloo.ca
University of Waterloo

ACM Reference Format:
Benjamin Cassell, Huy Hoang, and Tim Brecht. 2019. RocketStreams: A Framework for the Efficient Dissemination of Live Streaming Video. In *10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19), August 19–20, 2019, Hangzhou, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3343737.3343751

ABSTRACT

Live streaming video accounts for major portions of modern Internet traffic. Services like Twitch and YouTube Live rely on the high-speed distribution of live streaming video content to vast numbers of viewers. For popular content the data is disseminated (replicated) to multiple servers in data centres (or IXPs) for scalable, encrypted delivery to nearby viewers.

In this paper we sketch our design of RocketStreams, a framework designed to facilitate the high-performance dissemination of live streaming video content. RocketStreams removes the need for live streaming services to design complicated data management and networking solutions, replacing them with an easy-to-use API and backend that handles data movement on behalf of the applications. In addition to its support for TCP-based communication, RocketStreams supports CPU-efficient dissemination over RDMA, when available. We demonstrate the utility of RocketStreams for providing live streaming video dissemination by modifying a web server to make use of the framework. Preliminary results show that RocketStreams performs similarly to Redis on dissemination nodes. On delivery nodes, RocketStreams reduces CPU utilization by up to 54% compared to Redis, and therefore supports up to 27% higher simultaneous viewer throughput. When using RDMA, RocketStreams supports up to 73% higher ingest traffic on dissemination nodes compared with Redis, reduces delivery node CPU utilization by up to 95%, and supports up to 55% more simultaneous viewers.

APSys '19, August 19–20, 2019, Hangzhou, China. © 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19), August 19–20, 2019, Hangzhou, China*, https://doi.org/10.1145/3343737.3343751.

1 INTRODUCTION

Live streaming video services, such as Twitch [32] and YouTube Live [34], account for large amounts of Internet traffic [4]. Twitch's volume is of particular note, with Twitch being the fourth-largest consumer of peak Internet traffic in the United States [8, 31]. In 2016 alone, Twitch served over 292 billion minutes of video [9].

In order to meet consumer demand, live streaming video services ingest live video streams from producers into data centres (or equivalently IXPs). Requests from consumers are serviced by servers containing replicas of the live streaming video data. The scale of replication for a popular stream can be non-trivial: Previous research has shown that with only 30,000 viewers, a Twitch live stream could be dynamically served from up to 150 different servers [5]. Without this replication (which we refer to as dissemination), services would have difficulty achieving high scalability and low latency. Figure 1 shows a high-level overview of a live streaming video service in which producers (streamers) upload video to data centres. The streams are disseminated to servers within the same data centre and to nodes in another data centre. The streams are then delivered on demand from servers in either location to consumers (viewers).

Deploying software to accommodate this workflow is a challenging task. Services must either build their own software, which is difficult and time consuming, or use existing systems that are not necessarily optimized for live streaming workloads (for example, they often do not provide direct access to managed buffers, resulting in unnecessary copying). Both solutions require the software to be highly efficient, as live streaming video has strict real-time delivery constraints [5], and is disproportionately impacted by losses in quality of experience as users have much higher expectations for live streaming video than video-on-demand [6]. For example, a 1% increase in buffering ratio for a 90 minute soccer game translated to viewers watching 3 fewer minutes of the game (impacting viewer retention and revenue) [6].

We posit that services should not be forced to choose between suboptimal performance and a difficult implementation. As a solution, we present RocketStreams, an easy-to-use framework for enabling efficient and scalable live streaming video dissemination. The framework, which can be used

with new and existing software, provides TCP-based dissemination for replicating video within and across data centres. RocketStreams' API has been designed specifically to facilitate and accommodate live streaming video data, exposing a high-level, event-driven, buffer abstraction that eliminates the need for applications to worry about implementing efficient dissemination data management and networking code. The RocketStreams memory buffer abstraction and resulting APIs have been chosen and designed uniquely for their ease of integration with *both* RDMA and TCP. This abstraction ensures the CPU overhead incurred due to copying is avoided (which is sometimes required by applications using frameworks built around a sockets abstraction). In addition to providing good TCP performance relative to industry-grade solutions such as Redis [24], RocketStreams also provides RDMA-based dissemination which can dramatically reduce CPU utilization and further improve performance and scalability. The contributions of this paper are:

- A description of RocketStreams, a framework which provides applications with easy and efficient live streaming video dissemination, without the need to implement data management and networking code.
- An overview of RocketStreams' API, a description of our prototype implementation and an explanation of how, with little effort and relatively few lines of code, we use RocketStreams to disseminate video streams to an open source web server, the userver.
- Preliminary experiments indicate that RocketStreams provides similar dissemination node performance when compared to Redis, while on delivery nodes it reduces relative CPU utilization by up to 54%, leading to a 27% increase in simultaneous viewer throughput.
- When using RDMA, RocketStreams can ingest 18 Gbps while simultaneously delivering a total of 144 Gbps of traffic to delivery nodes. Comparatively, Redis is capped at 10.4 Gbps of ingest traffic and 83 Gbps traffic to delivery nodes. On delivery nodes, RDMA-enabled RocketStreams reduces CPU utilization by up to 95% versus Redis, allowing the userver to support up to 55% more viewers.

2 DESIGN

In this section we detail the RocketStreams live stream buffer management APIs, as well as the networking management component, RocketNet, that synchronizes data between buffers on different physical nodes. The intended use-case for RocketStreams is for live streaming video services where data moves as follows: data enters the system by being ingested from external producers by logical dissemination nodes (typically inside a data centre or IXP). The data is placed into circular buffers, using RocketStreams, and is disseminated by RocketNet (using framework-managed threads) from the dissemination nodes to logical delivery nodes. Delivery nodes access the data through RocketStreams, and transmit it to consumers on request. We refer to this design pattern as produce-ingest-disseminate-deliver-consume (PIDDC). Figure 1 depicts this design pattern. Our current live streaming video implementation supports streamers generating content as producers, and viewers requesting video data as consumers. In the future we plan to allow dissemination nodes to also produce data to other dissemination nodes, thus facilitating the geo-replication of data.

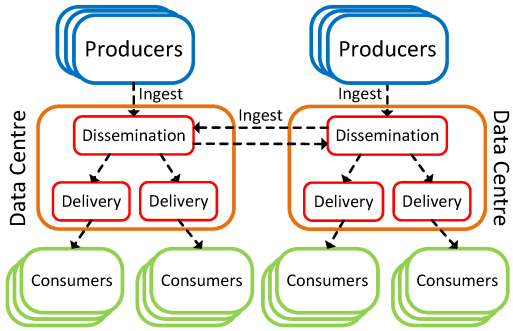

Figure 1: Live streaming video using PIDDC.

2.1 RocketStreams

RocketStreams is provided as a user-space library. Its APIs are event-driven and can be used to implement dissemination and delivery logic, such as the samples shown through pseudocode in Listing 1 and Listing 2. Applications initialize RocketStreams by providing it with a configuration file containing setup details (for example, networking addresses for nodes, the roles and IDs of nodes, and network protocol settings), seen on line 1 of Listing 1 and line 1 of Listing 2. Once initialized, RocketStreams manages live streaming video buffers on behalf of applications. Ingested data is placed into circular buffers on the dissemination node, one per live stream, in variable-sized chunks. Data is disseminated by RocketNet using framework-managed threads created during initialization. If dissemination and delivery nodes reside on the same server, network communication is avoided.

Listing 1: Sample dissemination node code.

```
rs::RsManager rs("config_file.cfg");                                    1
rs::RsId id = rs.create_stream("mystream");                             2
rs.set_outbound(delivery_node_endpoint, id);                            3
while (true) {                                                          4
  producer_await_stream(id); // Wait for ingest data.                   5
  rs::RsChunk chunk = rs.allocate_chunk(id, SIZE);                      6
  producer_receive_data(id, chunk); // Ingest data.                     7
  rs.disseminate_chunk(chunk, SIZE, nullptr);                           8
}                                                                       9
```

Applications call `rs_create_stream` on dissemination nodes, which allocates a dissemination buffer for a live stream and returns a unique ID (across all nodes) for that stream. This is depicted on line 2 of Listing 1. The application can choose which delivery nodes (of those specified in the configuration file) receive disseminated chunks from a live stream. This allows live streams to be associated with a flexible number of delivery nodes, per application requirements. This process can be initiated from dissemination nodes or delivery nodes, and in a more complex system would typically be done in consultation with or at the direction of a control plane performing service-wide load balancing. In our simple example, the dissemination node pushes to a single delivery node. This is set up on line 3 of Listing 1 using `rs_set_outbound`.

Listing 2: Sample delivery node code.

```
rs::RsManager rs("config_file.cfg");                          1
rs.inboud_callback(on_data_cb);                               2
webserver_run(); // Wait for incoming requests.               3
                                                              4
void on_data_cb(rs::RsId id, void* data, size_t size) {       5
  webserver_new_data(id, data, size);                         6
  size_t oldest = webserver_get_oldest(id);                   7
  rs.data_consumed(id, oldest); // Return oldest data.        8
}                                                             9
```

Applications perform dissemination using a two-phase workflow. In the first phase, memory is requested from a live stream's buffer using `rs_allocate_chunk`, as in line 6 of Listing 1. The application provides a requested size and receives a one-shot chunk of memory suitable for writing data to The chunk is owned by the application, and is returned to the framework during dissemination. Data entering the system through ingestion may be placed there directly by the application in whatever manner it chooses, as is done on line 7 of Listing 1. When the application is ready for the data placed in the chunk to be disseminated, it passes the chunk to `rs_disseminate_chunk`, providing the size of the data to disseminate and an optional application-specified callback context. This is done on line 8 of Listing 1. At this point, the application no longer owns the chunk (it is being managed by the framework and will be disseminated by RocketNet). `rs_disseminate_chunk` is non-blocking, queuing the chunk for dissemination by RocketNet. Once dissemination is complete, a callback is generated, which is provided with the application's context. The callback function to invoke can be registered by the application using a call to `rs_outbound_callback`. For simplicity, Listing 1 omits the callback function and registration.

On delivery nodes, applications register a callback function to be notified of incoming disseminated chunks using `rs_inbound_callback`. This callback is provided with the ID of the live stream receiving that data, as well as a reference to the data itself. Line 2 of Listing 2 shows this registration, with the callback function itself beginning on line 5. The application assumes ownership of data provided to this callback, and can use the associated buffers in whatever manner is needed, such as on line 6 of Listing 2, which indicates to the web server that the specified data is ready for delivery. This allows the video data to be sent to clients directly from the framework buffers, avoiding potential copying. When the oldest data in the circular buffer is no longer needed, the application returns that memory to the framework by calling `rs_data_consumed` with a size parameter to notify the framework of how much data is being returned. This is seen on line 8 of Listing 2.

2.2 RocketNet

RocketNet is responsible for sending and receiving data between buffers on different hosts. It uses framework-managed worker threads and an event-driven model to asynchronously disseminate data between buffers as seen in Figure 2. Where required, these threads initiate application-layer callbacks. RocketNet currently supports communication using TCP and RDMA.

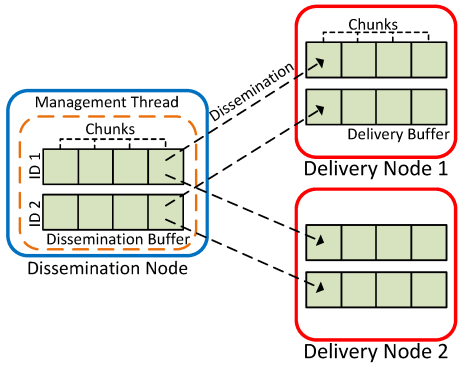

Figure 2: Buffer management with RocketStreams.

An important consideration for RocketNet is how to make buffer space available for use and re-use for disseminated data. For live streaming video, data is short-lived, and having it available for long periods is not useful (data past a certain age is considered stale and would not be suitable for live video delivery). Thus, RocketNet overwrites the oldest disseminated data in a buffer as new disseminated data arrives. This design avoids overheads due to synchronization, however it makes it possible for data to be overwritten asynchronously on delivery nodes as applications access it (particularly for RDMA). The framework wraps disseminated chunks with consistency markers, which may be checked by applications to see if data has been modified during the course of serving viewers. This behaviour occurs when the system has been underprovisioned or overloaded,

upon which viewers are notified of the error (described in our experiments in Section 3.2). In a real-world scenario, this would be an indicator to perform load balancing by moving either streams or viewers to other nodes.

RocketNet provides RDMA support to achieve a higher level of performance than is achievable with TCP when RDMA NICs are available. Worker threads perform zero-copy RDMA write-with-immediate verbs directly from dissemination buffers into delivery node buffers, bypassing both nodes' operating systems and CPUs. The immediate data is used to trigger the callbacks used by RocketStreams, notifying applications of incoming data (as otherwise, RDMA verbs are invisible to the CPU). By avoiding polling mechanisms, CPU resources are conserved for application-related purposes. To reduce RDMA resource contention on the NIC, the buffers used by RocketStreams and RocketNet are allocated from within larger framework-managed RDMA-registered regions of memory, instead of being RDMA-registered individually.

2.3 Implementation

Our RocketStreams implementation is coded in C++. We provide a native interface which can be used directly by C and C++ applications (and applications written in other languages with native support). In addition, we provide an interface for socket-based communication with a self-contained version of the framework that exposes delivery buffers through shared memory. This allows language-independent access to the delivery aspect of the framework. Currently RocketStreams supports unicast dissemination, but plans for the future include support for other strategies like tree-based dissemination, and multicast dissemination.

For our evaluation we have implemented a sample dissemination process that performs ingest, and integrated RocketStreams into an open source web server, the userver, which has been shown to perform well [2, 21, 28]. As a point of comparison, we also modified the userver to subscribe to and receive data from a Redis [24] broker using the `hiredis` client library [23]. The userver modifications amount to 248 and 229 lines of code for RocketStreams and Redis, respectively. Our evaluation in the Section 3.2 shows that the userver using RocketStreams is able to achieve similar to or better performance than when using Redis.

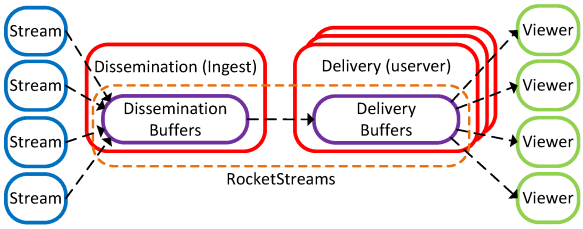

Figure 3: RocketStreams integration with the userver.

3 EVALUATION

In order to demonstrate the utility and efficiency of our framework, we perform a series of experiments comparing the ingest, dissemination and delivery performance of hosts that use RocketStreams and Redis [24]. Redis is an open source, in-memory data store that has performed well when used to disseminate data for live streaming [26].

In our experiments, a single host acts as a dissemination node that ingests live streaming video data from multiple simulated producers (running on a separate host). Data is ingested at a bitrate of 2 Mbps in chunks, each containing 2 seconds worth of video data. This is consistent with averages found in literature that examines Twitch-style workloads [22]. Ingested data is pushed to the delivery nodes, which consist of web servers processes described in Section 2.3. Figure 3 depicts this setup for RocketStreams, in addition to the viewers used to request data from the web server for our benchmarks in Section 3.2. For experiments with Redis, multiple instances of Redis brokers (one per CPU core) act as the dissemination node and corresponding web servers as delivery nodes.

In our dissemination node experiments, the producers use TLS encryption when communicating with the dissemination node. This is done to ensure that the CPU on the dissemination node incurs the overhead required for decryption. When conducting delivery node experiments we disable TLS on the producers to allow higher throughput through the dissemination host and since this does not impact the delivery nodes. For web server experiments TLS is enabled for all viewers.

Our Redis-based dissemination node uses Redis' publish/-subscribe feature to disseminate data to the delivery nodes. Connections between producers and the dissemination node are secured using Stunnel (per Redis' recommendation [25]). We use multiple Redis processes to fully utilize available CPUs (Redis processes are single-threaded).

For all experiments, our dissemination node runs on a host containing a 2.6 GHz Intel Xeon E5-2660v3 CPU with 10 cores and 512 GB of RAM and all delivery nodes run on separate hosts containing a single 2.0 GHz Intel Xeon D-1540 CPUs with 8 cores and 64 GB of RAM. All hosts use bidirectional Mellanox ConnectX-3 40 Gbps NICs, with both the switch and NICs configured to support bandwidths of up to 56 Gbps [16]. The dissemination host contains four NICs to permit high throughput dissemination to multiple delivery hosts which each contain one NIC. All hosts run Ubuntu 14.04.5 with Linux kernel 4.4.0. We evaluate RocketStreams using both its TCP (Rs-TCP) and RDMA (Rs-RDMA) modes and compare their performance against Redis. Each experiment consists of a 60 second warmup period, followed by a 120 second measurement period.

3.1 Microbenchmarks

We first perform a series of microbenchmarks to determine the maximum number of ingested streams (expressed in terms of throughput) that the system can handle as we vary the number of delivery hosts receiving disseminated data. We also observe the impact of handling this incoming data on the CPUs of the delivery hosts. For these microbenchmarks viewers are not issuing requests. This permits us to isolate the impact of disseminated data on the delivery hosts.

Dissemination host throughput: Figure 4 shows the maximum achievable ingest throughput of the dissemination node while disseminating to different numbers of delivery nodes. To find these throughput values, we increase the number of producers in the system until dissemination fails to keep pace with the rate at which the system ingests data. In these experiments, this corresponds to the point when the dissemination node's CPU is close to 100 % utilization (due to overhead from ingesting encrypted data and disseminating with Redis or RocketStreams).

In all cases, Rs-TCP achieves ingest throughput which is largely comparable to that of Redis (although slightly better). As the number of delivery nodes increases (and therefore the amount of CPU required for dissemination), both Rs-TCP and Redis are unable to keep pace with ingested data at roughly the same rate. Enabling RDMA for RocketStreams yields a significant boost in maximum ingest throughput versus Redis, which remains consistent even as the number of delivery nodes increases (since RDMA requires little CPU regardless of the amount of disseminated data, allowing it to be utilized to ingest data). With 8 delivery nodes, Rs-TCP achieves over 11 Gbps of ingest throughput, with Redis achieving 10.4 Gbps. Rs-RDMA achieves 18 Gbps of ingest throughput, while simultaneously supporting 144 Gbps of throughput to delivery nodes representing an increase of 73 % versus Redis.

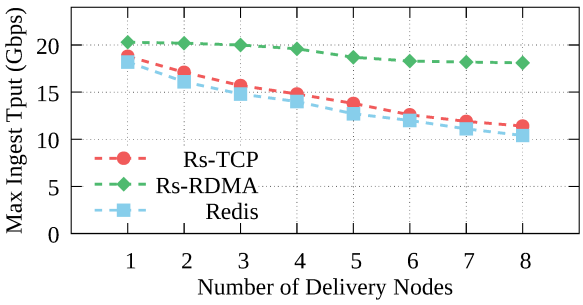

Figure 4: Maximum ingest throughput.

Delivery host CPU utilization: Figure 5 shows the average percentage of CPU utilization on a delivery server as the amount of disseminated data it receives increases. The microbenchmark results show that RocketStreams requires significantly less CPU on delivery hosts when compared with Redis. For example at 32 Gbps, CPU utilization of Redis is 50%, whereas for Rs-TCP it is 23%, a relative reduction of 54%. By profiling the Redis-based delivery server we found that 35% of the CPU time is spent in `memcpy`, used both by `hiredis` internally, and required by the server to get data out of `hiredis`. This is required because `hiredis` frees data immediately after a received data event is handled. This is not required when using RocketStreams because, by design, it provides direct access to receive buffers to avoid copying. For Rs-RDMA, CPU utilization is negligible regardless of the amount of data being received by the delivery node. At 32 Gbps CPU utilization is capped at 3% (a reduction of 95% versus Redis). In Section 3.2, we show how these CPU savings allow the system to support significantly more simultaneous viewers.

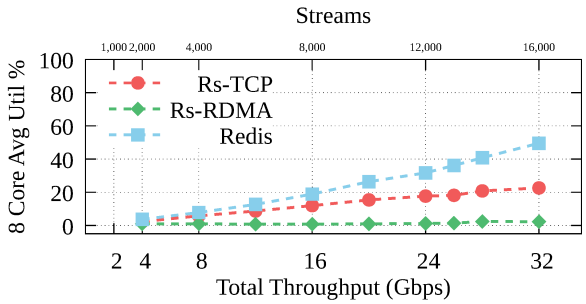

Figure 5: Delivery server average CPU utilization.

3.2 Live Streaming Video Benchmarks

We next run a set of benchmarks to determine how dissemination with Redis and RocketStreams impact the system's overall capacity to deliver live streaming video to consumers (viewers). For these experiments, we use `httperf` [18] to mimic large numbers of viewers using TLS connections generating load on the delivery nodes' web servers. Each viewer connection requests video content at the rate it is produced (2 Mbps). A sufficient number load generating hosts are used to ensure they and the network are not bottlenecks. Viewers also check received video data for timeliness and validity.

When the number of viewers exceeds the capacity of a delivery node (the web server is not able to meet the liveness constraints of viewer requests), viewers exhaust their playout buffers or request expired video segments. These occurrences are recorded as errors. For varying numbers of produced streams (measured in terms of their total throughput), we conduct a sequence of experiments to determine the maximum throughput a delivery node can support without any viewers reporting errors. In our experiments, we found that this threshold is reached when the delivery server's CPU is saturated servicing incoming dissemination data, and outgoing delivery requests.

Figure 6 shows the results of these experiments. As noted in Section 3.1, Rs-TCP uses less CPU than Redis for handling incoming disseminated data by avoiding copying, and this difference grows as the amount of disseminated data increases. As a result, for 20 Gbps of incoming disseminated data, while the web server using Redis is only able to serve 13.5 Gbps of video to viewers, the userver using Rs-TCP achieves over 17 Gbps, a relative increase of 27%. The userver using Rs-RDMA achieves delivery throughput of 21 Gbps (55% higher than Redis), regardless of dissemination throughput.

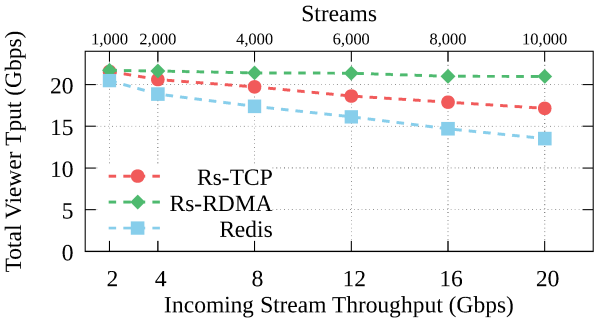

Figure 6: Maximum error-free viewer throughput.

4 RELATED WORK

The WILSP platform [26] provides an open-source framework for disseminating and delivering *interactive* video. As such, they have focused on low-latency and small numbers of users. Their delivery web servers are benchmarked serving a maximum of 50 users, representing about 230 Mbps of delivery throughput. In contrast, RocketStreams is designed for efficient and highly-scalable *live streaming* video.

The Remote Regions abstraction [1] provides a file-based interface meant to simplify the process of interacting with remote memory. Our framework's focus is on providing tools to allow for the easy dissemination of live streaming video. RocketStreams' interfaces are tailored to be convenient for this use-case, and it provides networking and data management on behalf of applications. Unlike Remote Regions, RocketStreams works in environments without access to RDMA.

libfabric [10] provides an efficient, platform-independent networking API intended to simplify application use of networking hardware. As such, it provides functionality at a lower level of abstraction than we do through RocketStreams. libfabric could be used to implement some portions of RocketStreams' networking component, RocketNet, however for the purposes of this paper we use some infrastructure from our previous work, Nessie [3, 29], as it should provide similar performance and we are familiar with the code base.

Some other frameworks have been created for data dissemination in other use cases. For example, work by Pallickara et al. [20] describe a framework for the secure delivery of pub/-sub messages, albeit with a focus on security. Li et al. [15], present a framework for providing Reliable Data Center Multicast (RDCM) using IP multicast, techniques for which could be used to improve dissemination to multiple delivery nodes.

Other research has examined high performance RDMA techniques under a variety of workloads [7, 11, 13, 14, 17, 19, 27, 30, 33]. This is important given that differences in RDMA-based designs can dramatically impact performance [12]. These systems, however, typically do not provide an API convenient for the type of flexible video dissemination that RocketStreams supports (e.g., their abstractions are not easily mapped to live streaming). In several cases, existing systems are not designed with reliable connections or large data in mind, and most do not seamlessly integrate with non-RDMA networking protocols. RocketStreams currently uses RDMA techniques from our work with Nessie [3, 29], but we could integrate techniques from other systems such as FaRM [7] and HERD [11] to enable a wider variety of options for more flexible dissemination.

5 CONCLUSIONS

In this paper we introduce RocketStreams, a framework for efficiently handling live streaming video dissemination. We highlight RocketStreams' easy-to-use design, and how it eliminates the need for applications to implement their own conceptually and technically difficult data management and networking code. By providing direct access to framework-managed buffers that eliminate the need to copy data RocketStreams provides performance similar to or better than an industry-grade solution, Redis. We modify a web server, the userver, to access disseminated live streaming video data through RocketStreams, and use load generators to evaluate its ingest and delivery capacities relative to those of Redis. Our benchmarks show that RocketStreams provides similar dissemination performance to Redis, and on delivery nodes it reduces CPU utilization by up to 54%, thereby increasing viewer throughput by up to 27%. RocketStreams also provides support for RDMA, which allows RocketStreams to service up to 73% more ingest traffic on dissemination nodes. Likewise, on delivery nodes, RDMA-enabled RocketStreams reduces CPU utilization by 95% compared with Redis, and increases simultaneous viewer throughput by 55%.

In the future we hope to support multicast (to reduce dissemination bandwidth when sending to multiple delivery nodes) and to extend RocketStreams and RocketNet to support features for other targeted workloads such as publish/subscribe and message queuing systems.

6 ACKNOWLEDGEMENTS

We acknowledge funding from a University of Waterloo President's Scholarship, a David R. Cheriton Graduate Scholarship, an Ontario Graduate Scholarship and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Aguilera, M. K., Amit, N., Calciu, I., Deguillard, X., Gandhi, J., Novakovic, S., Ramanathan, A., Subrahmanyam, P., Suresh, L., Tati, K., et al. Remote regions: A simple abstraction for remote memory. In *Proc. USENIX Annual Technical Conference (ATC)* (2018), USENIX, pp. 775–787.

[2] Brecht, T., Pariag, D., and Gammo, L. accept()able strategies for improving web server performance. In *Proc. USENIX Annual Technical Conference (ATC)* (2004), USENIX, pp. 227–240.

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

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

[5] Deng, J., Tyson, G., Cuadrado, F., and Uhlig, S. Internet scale user-generated live video streaming: The Twitch case. In *Proc. Passive and Active Measurement Conference (PAM)* (2017), Springer, pp. 60–71.

[6] Dobrian, F., Sekar, V., Awan, A., Stoica, I., Joseph, D., Ganjam, A., Zhan, J., and Zhang, H. Understanding the impact of video quality on user engagement. In *Proc. Conference on SIGCOMM* (2011), ACM SIGCOMM, pp. 362–373.

[7] Dragojevic, A., Narayanan, D., and Castro, M. FaRM: Fast remote memory. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)* (2014), USENIX, pp. 401–414.

[8] FitzGerald, D., and Wakabayashi, D. Apple quietly builds new networks. https://www.wsj.com/articles/apple-quietly-builds-new-networks-1391474149, 2014. Wall Street Journal. Accessed April 12, 2019.

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

[10] Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R. D., Pritchard, H., and Squyres, J. M. 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)* (2015), IEEE, pp. 34–39.

[11] Kalia, A., Kaminsky, M., and Andersen, D. G. Using RDMA efficiently for key-value services. In *Proc. Conference on SIGCOMM* (2014), ACM SIGCOMM, pp. 295–306.

[12] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance RDMA systems. In *Proc. USENIX Annual Technical Conference (ATC)* (2016), USENIX, pp. 437–450.

[13] Kalia, A., Kaminsky, M., and Andersen, D. G. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)* (2016), USENIX, pp. 185–201.

[14] Kalia, A., Kaminsky, M., and Andersen, D. G. Datacenter RPCs can be general and fast. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)* (2019), USENIX, pp. 1–16.

[15] Li, D., Xu, M., Zhao, M.-c., Guo, C., Zhang, Y., and Wu, M.-y. RDCM: Reliable data center multicast. In *Proc. International Conference on Computer Communications (INFOCOM)* (2011), IEEE, pp. 56–60.

[16] Mellanox. How to configure 56GbE link on Mellanox adapters and switches. https://community.mellanox.com/s/article/howto-configure-56gbe-link-on-mellanox-adapters-and-switches. Accessed April 12, 2019.

[17] Mitchell, C., Geng, Y., and Li, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX Annual Technical Conference (ATC)* (2013), USENIX, pp. 103–114.

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

[19] Novakovic, S., Shan, Y., Kolli, A., Cui, M., Zhang, Y., Eran, H., Pismenny, B., Liss, L., Wei, M., Tsafrir, D., et al. Storm: A fast transactional dataplane for remote data structures. In *Proc. International Conference on Systems and Storage (SYSTOR)* (2019), ACM, pp. 97–108.

[20] Pallickara, S., Pierce, M., Gadgil, H., Fox, G., Yan, Y., and Huang, Y. A framework for secure end-to-end delivery of messages in publish/-subscribe systems. In *Proc. International Conference on Grid Computing (GRID)* (2006), IEEE, pp. 215–222.

[21] Pariag, D., Brecht, T., Harji, A., Buhr, P., Shukla, A., and Cheriton, D. R. Comparing the performance of web server architectures. In *Proc. European Conference on Computer Systems (EuroSys)* (2007), ACM, pp. 231–243.

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

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

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

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

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

[27] Su, M., Zhang, M., Chen, K., Guo, Z., and Wu, Y. RFP: When RPC is faster than server-bypass with RDMA. In *Proc. European Conference on Computer Systems (EuroSys)* (2017), ACM, pp. 1–15.

[28] Summers, J., Brecht, T., Eager, D., Szepesi, T., Cassell, B., and Wong, B. Automated control of aggressive prefetching for HTTP streaming video servers. In *Proc. International Conference on Systems and Storage (SYSTOR)* (2014), ACM, pp. 5:1–5:11.

[29] Szepesi, T., Wong, B., Cassell, B., and Brecht, T. Designing a low-latency cuckoo hash table for write-intensive workloads using RDMA. In *Proc. International Workshop on Rack-scale Computing (WRSC)* (2014), pp. 1–6.

[30] Tsai, S.-Y., and Zhang, Y. LITE kernel RDMA support for datacenter applications. In *Proc. Symposium on Operating Systems Principles (SOSP)* (2017), ACM, pp. 306–324.

[31] Tsukayama, H. 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.

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

[33] Wei, X., Shi, J., Chen, Y., Chen, R., and Chen, H. Fast in-memory transaction processing using RDMA and HTM. In *Proc. Symposium on Operating Systems Principles (SOSP)* (2015), ACM, pp. 87–104.

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