FASTPS: A Fast Publish-Subscribe service using RDMA

Arjun Balasubramanian Mohammed Danish Shaikh

University of Wisconsin - Madison

Abstract

We present FASTPS, a fast publish-subscribe service that utilizes RDMA both for fast transport and for efficient CPU offload. We first present requirements from a new class of applications that have very different requirements from the use-case for which Queue Management Systems were designed for. These requirements motivate a ground-up redesign of such systems with RDMA at the heart of it.

For the course project, we will build a basic version of FASTPS (using in-memory replication) and evaluate its performance against Apache Kafka. We may not handle all failure scenarios and all possible client configurations and will only support the ones .

1 Introduction

The publish-subscribe paradigm is increasingly becoming a popular abstraction in a number of emerging applications. These applications make use of *Queue management systems (QMS)* such as Kafka [11], RabbitMQ [16], AWS Kinesis [2], AWS SQS [4], and Google Pub/Sub [7]. Some of these applications are -

- **Serverless Computing.** Function-as-a-service (FaaS) through offerings such as [3,5,6] is becoming a popular computation model among programmers. This is because with FaaS, programmers need to worry only about the programming logic and aspects such as scaling and resource management are handled by the serverless computing platform. Programmers frequently use QMS systems to trigger the execution of dependent functions (also called *lambdas*) upon the execution of a lambda. For instance, one could have a workflow where one lambda handles the upload of an image to an S3 bucket. This lambda could in turn trigger the execution of another lambda that resizes the image that was uploaded in order to create a thumbnail. Similarly, frameworks such as PyWren [8] can use QMS systems to trigger executions of successive stages of an execution DAG. We also believe that QMS systems might be a viable way to exchange *ephemeral data* [10, 15] among these lambdas. These potential applications raise the following requirements from QMS systems - (i) Lambdas are frequently used to serve user-facing applications and hence QMS systems must be able to trigger execution of dependent lambdas or exchange ephemeral data at low latency. (ii) We also see that there are scenarios where lambdas might serve background tasks such as image resize. Here, it is more important to be throughput oriented than latency sensitive and hence QMS systems in this scenario must provide high throughput.

- **Scaling Artificial Intelligence.** With deep-learning models becoming larger and computationally expensive, *distributed ML training* has become a popular technique to reduce the training time. A popular architecture for distributed training is the parameter server model [12] which consists of a *Parameter Server (PS)* that holds the learned model parameters and a bunch of *workers* that operate on portions of the data in parallel and periodically push local gradient updates back to the parameter server. The parameter server model frequently employs the *synchronous training* paradigm where the *PS* waits for each worker to complete one epoch of training and push its local gradients. Post this, each worker pulls the latest copy of the parameters before proceeding to the next training epoch. We can model this using the QMS system paradigm where the *PS* produces a record containing the latest parameters and the workers consume this record to start the next epoch of training. This application raises the following requirements from QMS systems - (i) There is usually a single *PS* task and multiple *worker* tasks. Hence, a QMS system must be able to support a *fan-out* structure with a *single producer* and *multiple consumers*. (ii) In order to reduce the total training time, it is important that the records are delivered with *low latency*. Since a training epoch usually lasts for a significant amount of time, this application does not have a high throughput requirement from the QMS system. Similar requirements exist for systems that support *reinforcement-learning* [14].

- **Change Data Capture (CDC).** CDC is a design pattern that allows applications to observe the delta change in data from a data source. It is typically used to perform analytics over evolving data. As an example, CDC frameworks such as [13] use QMS systems in order to stream updates from a MySQL database (by registering as a slave to listen to binary log events) to a stream processing engine such as [1]. This application raises the following requirements from QMS systems - (i) For analytics to be performed in real time, QMS systems need to serve CDC records to stream processing engines at low latency. (ii) Database systems are typically throughput-oriented and hence will provide a large number of CDC records. This means that QMS systems need to serve CDC records to stream processing engines at high throughput.

2 Goals

Below, we summarize the requirements for QMS systems raised by these new classes of applications -

- **Providing low latency and high throughput.** As observed in Section 1, applications have different requirements. Most applications require high throughput or low latency or even both of these simultaneously. Today, QMS systems allow producers of data to control configurations to achieve these requirements. For instance, a Kafka producer exposes configurations such as *batch size and linger time*, where a larger batch size and longer linger time help in achieving higher throughput. However, the write throughput to a single Kafka partition is constrained by two factors - (i) Writes to disk. (ii) Establishing correct ordering for records within a partition. To overcome this bottleneck, application programmers can use multiple partitions. However, the drawback to this approach is that it is not possible to reason about the ordering of records across partitions. Hence, it is beneficial to application programmers if a QMS system can provide *higher read and write throughput* to a single partition.

- **Supporting heavy fan-out structure.** As observed in Section 1, most applications have a single producer of data and multiple consumers of that data. Hence, it is important for any QMS system to not become a performance bottleneck while serving multiple consumers. The general way this is achieved is by making different consumers read data records from different replicas, which effectively load balances consumers in order to prevent any single server from being bottlenecked on its CPU. However, when there is a *heavy fan-out structure*, CPU will easily become a bottleneck while serving consumers. One potential way to counteract this might to increase the number of replicas, but this in turn would severely degrade the producer's write throughput. Hence, QMS systems today have an inherent problem with this requirement.

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

With the above requirements, it is clear that the architecture of QMS systems needs to be re-visited. This work is an effort in this step and looks at a ground up re-design of QMS systems.

Given the above requirements for QMS systems, we believe that they can greatly benefit by using *Remote Dynamic Memory Access (RDMA)* networks. Unlike previous work that simply uses RDMA as a fast transport conduit, FastPS deeply integrates RDMA into its core design. We discuss an initial sketch of such a design in Section 3.3. Some of the important RDMA features we leverage are -

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

- RDMA inbound verbs incur lower overheads for the target server's CPU and hence a server can handle an order of magnitude more requests than it can initiate. Traditionally, QMS systems have servers notify consumers about the availability of a new data record. With RDMA capable networks, we can adopt a model where consumers can poll the server for the availability of new records without incurring overheads. The polling model helps provide low latency delivery of records to consumers.

- RDMA favors short data transfers for two reasons - (i) Sending smaller packets involves a smaller number of PCIe transactions [9] (ii) Modern RDMA hardware can inline small messages along with WQE headers. Both of these promote low latency.

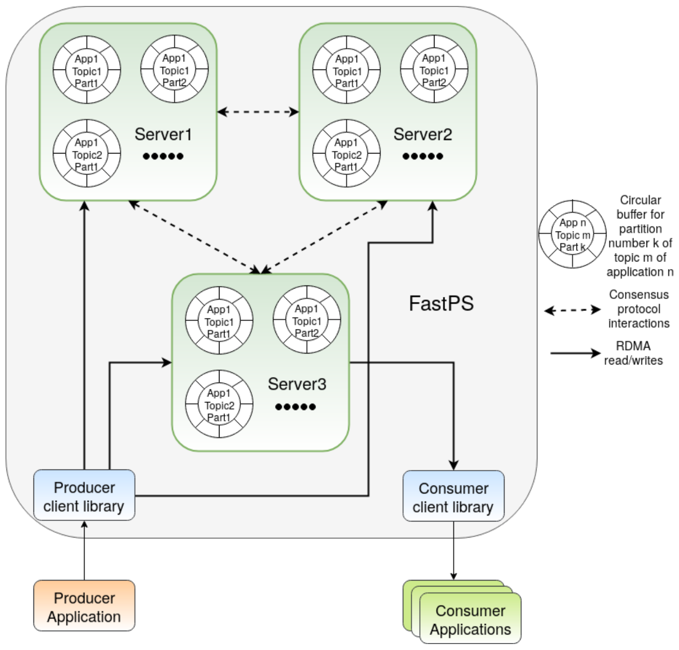

Figure 1: FastPS Architecture

3 Design

This sections presents the internal design of FASTPS. In Section 3.1, we discuss the architecture of FASTPS. We present the APIs we plan to support in Sections 3.1.2 and 3.1.3. Finally, we describe the producer and consumer flows in Sections 3.2 and 3.3 respectively.

3.1 Architecture

The FASTPS QMS system consists of *servers*, a *producer library*, and a *consumer library* components as described in the following subsections (refer Figure 1).

Like [11], we assume that producers write to a topic which may consist of multiple partitions.

3.1.1 Server

FASTPS consists of multiple servers that store the data produced by producer(s) in order to be consumed by consumer(s). Each server maintains two buffers, namely an *application buffer* and a *log buffer* for each *partition* belonging to a *topic* of an *application*.

Application buffer. This is a circular buffer that the *producer library* uses to perform RDMA writes consisting of the data to be written into the FASTPS.

Log buffer. This buffer keeps track of committed data after consensus. It also serves as a way for *consumer library* to know about the availability of committed data to be consumed by consumer(s).

3.1.2 Producer library APIs

The producer library provides the following APIs to the producers:

InitProducer(application name, topic, number of partitions): When a producer wants to start producing data for a new *topic*, it has to call into the *InitProducer* API provided by the producer library and provide the *application name*, name of the *topic* and *number of partitions* to be created for the topic. The producer library thereafter informs the *servers* to initialize their respective data structures. The *servers* initialize the data structures and return the memory region that the producer library can perform RDMA writes into.

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

3.1.3 Consumer library APIs

The consumer library provides the following APIs to the consumers:

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

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

3.2 Producer Flow

1. The producer calls *Produce* API specifying the *data* to be produced.
2. On receiving the call, the producer library writes the data using RDMA into application buffers of each server. For this, we use a one-sided RDMA write.
3. The producer library informs the leader that it has written new data to be produced into the application buffers. This can also be done using a one-sided RDMA write into the leader's application buffer.
4. The leader executes a consensus protocol to commit the data on all servers. The leader executes consensus by first asking all servers if they have the data written by the client library. If yes, the leader asks each server to commit the data by appending a pointer to the application data in it's log buffer. This step requires 2 sided RPCs and hence server CPU involvement.

5. Once the consensus has succeeded, the leader returns *success* to the producer library.

3.3 Consumer Flow

1. The consumer calls *Consume* API specifying that it wants to subscribe to data produced by producers at partition *p* of topic *t* (for an application *a* specified during the *InitConsume* API call).
2. On receiving the call, the consumer library starts polling the *log buffer* for the partition *p* belonging to topic *t* of the given application *a* on *any server*, waiting for some data to be committed. Polling can be done using one-sided read operations and hence does not bottleneck the server CPU.
3. When the consumer library notices a new commit on the aforementioned *log buffer*, it retrieves the required data from the corresponding *application buffer* and returns it to the consumer. This requires 2 one-sided RDMA reads - one to read the pointer to the application data from the log buffer and one to read the actual application data from the application buffer.

4 Discussion

Section 3.3 only discusses the normal case operation of FastPS. In this section, we address some concerns with regards to using FastPS practically.

4.1 Failure Handling

We describe below how we could possibly handle some failure scenarios.
What happens in case a leader fails? The notion of leader is basically just used for the consensus protocol. If a leader fails, the producer library just contacts another server in FastPS. Depending on whether a consensus was ongoing when the leader failed or not, the new leader might have to use a *unique* log index number for starting a new consensus run. Furthermore, if a consensus protocol was ongoing when the leader failed, the corresponding log indices will have to be filled with a *no-op* at some point.
Let's say that the producer library writes data into all servers in FastPS *and informs the leader. Furthermore, let's assume that the leader starts sending a commit message to all servers, but fails before it can send the message to all servers. How is this inconsistency resolved?* In such a case, a failure is returned to the producer library, which thereby communicates the failure to the producer. The producer would then have to retry publishing the given *data*. The log indices corresponding to the failed consensus would have to be filled with a *no-op* at some point. Alternatively, we can let the new leader commit the pending data to all servers. Note that in this case the data will get committed twice,thereby providing an *atleast once* semantics to the consumers (the same can also be done for the aforementioned scenario).

4.2 Log Compaction

As discussed in section 3.3, each server in FastPS maintains a log buffer unique to a *partition* of a *topic* belonging to an *application*. The number of log buffers thus maintained could be very large, and hence would need to be managed properly in order to ensure that FastPS doesn't run out of memory space. One way to do this could be to associate a TTL (*time to live*) with each *topic* during initialization. Subsequently, a periodic purge of expired entries in the *log buffer* can be performed. Note that associating a *TTL* automatically ensures the management of *application buffer* since it is circular.

4.3 Persistence

In order to keep the scope of the project constrained to start with, we're only storing the data generated by publishers in memory. However, practically we would want to persist this data. Persisting data would have an impact on the latency offered by FastPS since the servers would respond success to the producer library only after either the data or a log has been persisted. In such a case, our disks might end up being the bottleneck. One possible way of dealing with this issue could be to use NVMMs to persist data.

4.4 Impact of batching on throughput/latency

As discussed in Section 3.3, on receiving a request from the producer library to commit, the leader starts the consensus protocol and commits the data. Another interesting strategy here would be to batch the requests and do consensus on batches of request, the batch size could be a client configurable parameter. Then, it would be interesting to evaluate the throughput/latency tradeoff in doing so.

References

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

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

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

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

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

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

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

[8] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 445–451.

[9] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 437–450.

[10] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 427–444.

[11] Kreps, J. Kafka : a distributed messaging system for log processing.

[12] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 583–598.

[13] Maxwells Daemon. https://maxwells-daemon.io/, 2019.

[14] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 561–577.

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

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