RDSQ: Reliable Queue Protocol over Shared Logs

Haolin Yu*
University of Waterloo
Waterloo, Ontario, Canada
h89yu@edu.uwaterloo.ca

ACM Reference Format:
Haolin Yu. 2018. RDSQ: Reliable Queue Protocol over Shared Logs. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3183713.3183718

1 INTRODUCTION

With the increasing popularity of message-oriented middleware as a solution to inter-component communication in distributed systems, great effort has been put into designing message queue protocols with strong consistency and persistence guarantees under failure scenarios such as process crashes and network partition. Although mature applications implementing reliable at-least-once semantics have been developed[4, 13], the behavior of these applications under fault proves the complexity and difficulty in designing error-free systems. For instance, RabbitMQ can lose a significant portion of acknowledged writes under network partition[7]. Meanwhile, many log services that provide verified safety guarantees have been developed and widely applied[1, 5, 6, 10, 11, 14]. The main contribution of this research is 1) to design and implement a provably correct shared message queue protocol that is highly consistent and durable and can tolerate client failures, while 2) maintaining competitive performance, and 3) to exploit the communication and replication protocols provided by shared log services. To the best of our knowledge, a reliable queue implementation over shared logs achieving all of these properties does not exist. In this work, we present and evaluate a log-based message queue protocol, Replicated Data Structure Queue (RDSQ).

2 BACKGROUND

A *log* is a data structure where new records (i.e., log entries) are appended to the end, and records are read from head to tail in order. Each entry is identified with a unique Log Sequential Number (*LSN*). Generally, logs support the following interface. Clients can *append* a record to the log and get back its LSN, *read* log records and their LSN, and *trim* a prefix of the log for space reclamation.

A *shared log* is a log service that can be accessed by multiple clients concurrently, where clients independently maintain their own read progress. Shared logs provide several valuable properties in distributed systems. Such *distributed shared logs* are usually designed to be highly scalable and may support hundreds of thousands of concurrent client operations per second[1]. Moreover, when used to achieve consensus over replicated data, distributed shared logs provide a simpler interface than complex algorithms like Paxos[9]. Finally, since shared logs keep a full history of modifications, application state is easily replicated by replaying the whole log (i.e., state machine replication[12]).

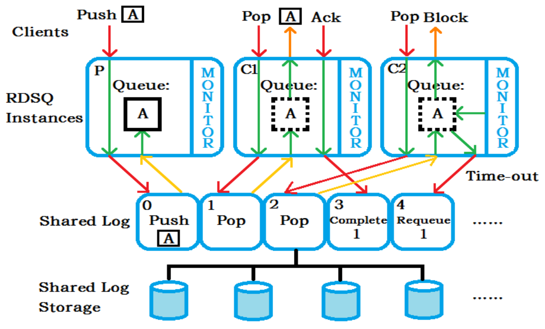

Figure 1: How RDSQ instances communicate

Some research has been done on building data structures[2] and databases[3] over shared logs, but we are not aware of any that addresses the unique challenges of a message queue. Unlike many data structures, message queues require additional coordination amongst readers. A *message queue* provides an asynchronous method to communicate between publishers and subscribers. Publishers *push* messages onto a queue without having to wait for an acknowledgment from subscribers, while subscribers *pop* existing messages from the queue or wait for new messages to arrive if the queue is empty. A message is said to be *delivered* once it is popped. With *at-most-once semantics*, a queued message is removed immediately when delivered. With *at-least-once semantics*, the subscriber can send an *acknowledgment* to indicate a successful delivery and permanently remove the message. Otherwise, a delivered message may be *requeue*d and delivered again.

3 DESIGN OF RDSQ

RDSQ is a protocol on top of the shared log abstraction providing a message queue interface while retaining the safety properties of the log on which it is built. Multiple instances coordinate through the underlying log by writing and reading operations which represent modifications to their in-memory state. Each RDSQ instance is a concurrent system that consists of client threads, a replay thread, and a monitor thread. Clients initiate API calls on client threads, appending records to the log when applicable; the replay thread reads and processes records from the log; the monitor thread monitors all on-going deliveries and looks for timed-out ones. Fig. 1 shows

*Work done during an internship at SAP Labs, Waterloo, Ontario, Canada.

a use case of RDSQ as a work queue. P pushes a job first. C1 and C2 concurrently pop, but only C1 obtains the job. C2 blocks until a new job comes in. Later, C1 acknowledges. Meanwhile, C2 declares a timed-out delivery, which is ignored. As illustrated, no additional coordination is needed outside of the reads and writes to the log.

RDSQ employs four types of records on the log. A *Push* record contains a message pushed to the queue. A *Pop* record indicates a pop request initiated by some instance. To achieve at-least-once semantics, in addition to these records addressing the messages, RDSQ also provides *Complete* and *Requeue* records containing a single LSN to indicate whether or not deliveries are successful. Typically, these receipts are managed as non-durable internal metadata. By recording them explicitly, RDSQ internal states are now recoverable from failure.

RDSQ presents familiar *Push*, *Pop*, and *Ack* API calls to clients. A *Push* or an *Ack* call need only append a single *Push* or *Complete* record to the log, while a *Pop* call works in a two-phase fashion to allow maximum concurrent pop requests. First, a *Pop* record is appended to the log, and its LSN is recorded in the metadata as identification of ownership. Then the client thread blocks and waits for the replay thread to process the *Pop* record and deliver a message. While clients submit *Complete* records, the monitor threads, running periodically in the background, append *Requeue* records to the log for timed-out deliveries. The behavior of the replay thread depends on the types of records it processes. These operations can be considered as transitions from the state machine replication point of view.

- *Pop*: If the in-memory queue is currently empty, queue up the record. Otherwise, pop a message from the queue, and if the LSN is recorded in the metadata of this instance, send the message to the client thread.
- *Push*: If no *Pop* records are queued up, push it to the in-memory queue. Otherwise, remove the first *Pop* record, and treat this *Push* record as a popped message.
- *Complete*: Permanently remove the corresponding message. If such a message is absent, ignore this record.
- *Requeue*: Return the corresponding message to the in-memory queue. If such a message is absent, ignore this record. This strategy ensures that only the first *Complete* or *Requeue* record referring to a delivery decides if it is successful, and all the following ones have absolutely no effect.

Except for *Pop* records, a replay thread is oblivious to record ownership. A client failure can cause loss of ownership and lead to implicit aborting of pop requests initiated by this instance. However, such popped messages will eventually time out and be requeued.

4 RESULTS AND CONTRIBUTIONS

The properties and performance of RDSQ can be customized by changing the underlying log. For instance, a RAFT log gives stronger guarantees while a Kafka log yields better performance. In this evaluation, RDSQ is implemented on top of the SAP Vora distributed log[5], a distributed log that guarantees linearizability, N-1 fault-tolerance with N replicas, and zero acknowledged data loss under network partition.

4.1 At-least-once delivery

RDSQ provides the safety property that every acknowledged write will be delivered at least once, even under circumstances of client failure, N-1 server failures, and network partition as guaranteed by the log. Intuitively, a pushed message will be removed permanently only if a *Complete* record appears before a *Requeue* record, which leaves an opportunity to regain any lost message. Any client can ask the system to reclaim the message at any time by simply appending a *Requeue* record. Since internal states of a client can be completely reconstructed by replaying the whole log, if the log guarantees not to lose records, the state of a timed-out message can always be reconstructed. Thus, RDSQ will not lose a message in any situation where the log guarantees not to lose the associated records.

To prove that no behavior violates the at-least-once delivery safety property, we created a formal specification that precisely describes this protocol in TLA+[8]. Model checking has been performed over the specification, of which the results demonstrated the correctness of the at-least-once delivery safety property.

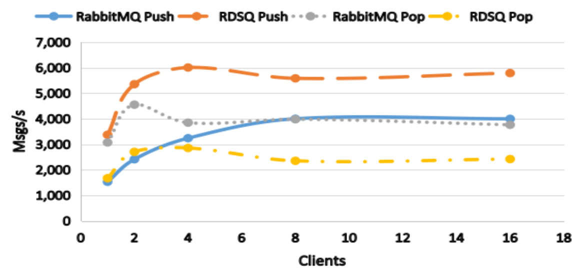

Figure 2: Performance

4.2 Performance

We evaluate the performance of RDSQ against RabbitMQ, a popular stand-alone message queue implementation. To perform this evaluation, we developed a workload generator that can perform tests over a variable number of messages and clients to ensure comparability between the two message queues. Fig. 2 shows the result of experiments where 10,000 messages are sent or read per client, each message is exactly 100 bytes, both message queues write and read one message at a time synchronously, and both guarantee the persistence of acknowledged writes with the at-least-once semantics. Publisher acknowledgment, subscriber acknowledgment, and message durability are turned on for RabbitMQ to achieve the same persistence of delivery guarantees as RDSQ.

The results show that RDSQ yields comparable performance while providing stronger safety properties. RDSQ achieves high write throughput by exploiting efficient underlying logs and preserving the simplicity of its write behavior. Read throughput is about half of the write throughput since a pop request writes twice to the log and takes time to process records in between.

5 CONCLUSION

In this work, we presented RDSQ, a message queue protocol where clients communicate through underlying shared logs to achieve strong safety guarantees. Performance testing demonstrates that RDSQ maintains comparable throughput to state-of-art message queues. By carefully designing the protocol and coordinating with the behavior of shared logs, RDSQ can guarantee zero message loss under client failures, server failures, and network partition while maintaining comparable performance and strong consistency.

REFERENCES

[1] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *NSDI*. 1–14.

[2] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 325–340.

[3] Philip A Bernstein, Colin W Reid, and Sudipto Das. 2011. Hyder-A Transactional Record Manager for Shared Flash. In *CIDR*, Vol. 11. 9–12.

[4] Nishant Garg. 2013. *Apache Kafka*. Packt Publishing Ltd.

[5] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards scalable real-time analytics: an architecture for scale-out of OLxP workloads. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1716–1727.

[6] Sijie Guo, Robin Dhamankar, and Leigh Stewart. 2017. DistributedLog: A High Performance Replicated Log Service. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE, 1183–1194.

[7] Kyle Kingsbury. 2014. Jepsen: RabbitMQ. (Jun 2014). Retrieved November 2, 2017 from https://aphyr.com/posts/315-jepsen-rabbitmq.

[8] Leslie Lamport. 2002. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc.

[9] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[10] Mark Marchukov. 2017. LogDevice: a distributed data store for logs. (Aug 2017). Retrieved November 2, 2017 from https://code.facebook.com/posts/357056558062811/logdevice-a-distributed-data-store-for-logs.

[11] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*. 305–319.

[12] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[13] Alvaro Videla and Jason JW Williams. 2012. *RabbitMQ in action: distributed messaging for everyone*. Manning.

[14] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. 2017. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *NSDI*. 35–49.