TritonMQ: A Real-Time Fault-tolerant In-Memory Distributed Message Queue

Wenbin Zhu
University of California, San Diego
9500 Gilman Dr, La Jolla, CA
wez180@eng.ucsd.edu

Dangyi Liu
University of California, San Diego
9500 Gilman Dr, La Jolla, CA
d5liu@eng.ucsd.edu

Abstract

This paper describes the design and implementation of TritonMQ, distributed message queue that provides fault-tolerance and real-time message delivery. TritonMQ uses a Publish-Subscribe model where consumers can subscribe to topics and receive messages related to the subscribed topics. TritonMQ aims to achieve good throughput by allowing producers to send messages in batch and good latency by in-memory processing, while still keeping a fail level of fault tolerance. It is intended to be used in applications like group chatting, log processing and news feeding.

1. Introduction

With the spread of microservice architecture, middlewares such as message queues become an increasingly important component in distributed systems. While there are already plenty of various message queues, most of them are focused on performance rather than reliability. In this paper, we present the design and implementation of a distributed message queue called TritonMQ. It features fault-tolerance while ensuring the basic semantics of message queue. In particular, TritonMQ uses a publish/subscribe model, where messages are grouped into topics. We use the term "producer" referring to publisher, and "consumer" referring to subscriber. Each message is produced by one producer on one topic, but is delivered to as many consumers as they have subscribed to that topic.

In order to support fault tolerance, we employ a primary/backup model, where one message is replicated on multiple servers and then sent by the primary to the recipients. These servers are called "Brokers". A primary is elected using Zookeeper [1] during runtime thus no preliminary configuration is required and re-election happens automatically when the primary fails. As long as there's still one machine holding the message, the message will eventually be delivered to all subscribers.

TritonMQ is also horizontally scalable. We organize the broker server into groups and messages of a certain topic are sent to a designated broker group. By adding more groups of servers upon system start-up,it can achieve higher throughput. We'll show in the following section that if certain assumption holds, our system will scale linearly.

We designed our system to be useful for applications including:

1. *Group chatting*. All clients engaged are both producers and consumers. One chatting channel corresponds to one topic in our system. In this scenario, message payloads are fairly small but one message may need to be forwarded to hundreds or thousands of consumers.

2. *Log processing*. A large computer system could produce lots of logs everyday. They are generated from different modules of the system but we may want to process and store them in one place sorted by their time-stamps. In this scenario, TritonMQ receives logs from tens or hundreds of sources and sends them to one or more destinations. The requirement here is high availability since logs are produced every second and we don't want to lose any of them.

3. *News feeding*. A news agency can use TritonMQ

to publish news and forward it to different subscribers, e.g., email service and web service. In this scenario, the message could have a large payload, including pictures or even videos.

The rest of this paper is structured as followed. Section 2 talks some existing solution of fault-tolerant message queues. Section 3 describes the architecture and principle of our design, together with the consistency model. Section 4 presents the implementation details including APIs that most applications would use. Section 5 shows some benchmark we did on TritonMQ and the results. Section 6 discusses the limitation and the potential improvements to our design.

2. Related Work

There are a lot of commercial and open source implementations of message queue. But not many of them support scalability or fault tolerance.

RabbitMQ [4] is a high-performance message queue written in Erlang, which supports scalability through clustering deployment. RabbitMQ cluster can also be configured as mirrored mode [3], where messages are replicated on several nodes and one of these nodes serves as the master with the others as mirrors. When the master fails, the longest running mirror will be prompted as the new master. However, there are some limitations of RabbitMQ's replication which makes our design better. First, RabbitMQ doesn't ensure fault tolerance. Messages may be lost if it arrives the master but the master fails immediately. Our design guarantees that once a message is acknowledged by the master, the message will eventually be delivered as long as there's still one node in the broker group. Second, RabbitMQ requires explicit configuration for the mirrored nodes, while our design uses ZooKeeper's leader election algorithm to elect master automatically.

Kafka [2] is a powerful and general-purpose stream processing platform which can also serve as a message queue. Our design resembles it a lot, but still has some differences. The most notable difference may be that our design uses a push-based model for communication between brokers and consumers, while Kafka uses pull model. In pull-based models, a consumer itself decides when and where to pull the messages, making it possible to rewind the offset and read old messages again. However, since a broker has no idea what the consumer's offset is, it may not be able to do garbage collection promptly. Our design uses push model where the broker pushes messages actively to the consumers, which requires the broker to remember the offsets of all consumers. The advantage of push model over pull model is its real-time feature—a consumer could receive a message as soon as the message arrives at the broker, since the communication is initialized by the broker.

3. Architecture & Design Principle

Our system leverages a typical publish-subscribe model, where producers generate messages and consumers subscribe to some topics and receive messages related to these topics. Figure 1 shows the basic architecture of our system with three major components, namely Producer, Broker and Consumer. Producer and Consumer components are presented as a client library, which provides APIs for users to call while the Broker component is presented as a service that can be called by the producer and consumer library. Normally the Broker service involves several clusters which we call *broker groups*, and there are one primary server and several backup servers reside in each group.

We design our message queue to be an in-memory system due to its real-time advantage and easier implementation. Processing messages totally in memory helps reduce the latency between sending and receiving of a message. Some modern distributed queues make use of a pull model which allows consumers to poll messages at their own pace. However, in order to reduce the memory burden at the Broker servers and keep the real-time feature, we choose to use the push model, where Brokers deliver messages to consumers as soon as possible.

3.1. Producer Design

The end point users can call Producer APIs to publish messages related some topics. Each message corresponds to a specific topic which we refer to as a record. A Producer's record is first hashed by its topic to get the ID of the corresponding broker group which it would be sent to and publish it to the primary of the group. Each producer record is also assigned an unique ID used for de-duplication at the server to assure that duplicate records are correctly ignored in our

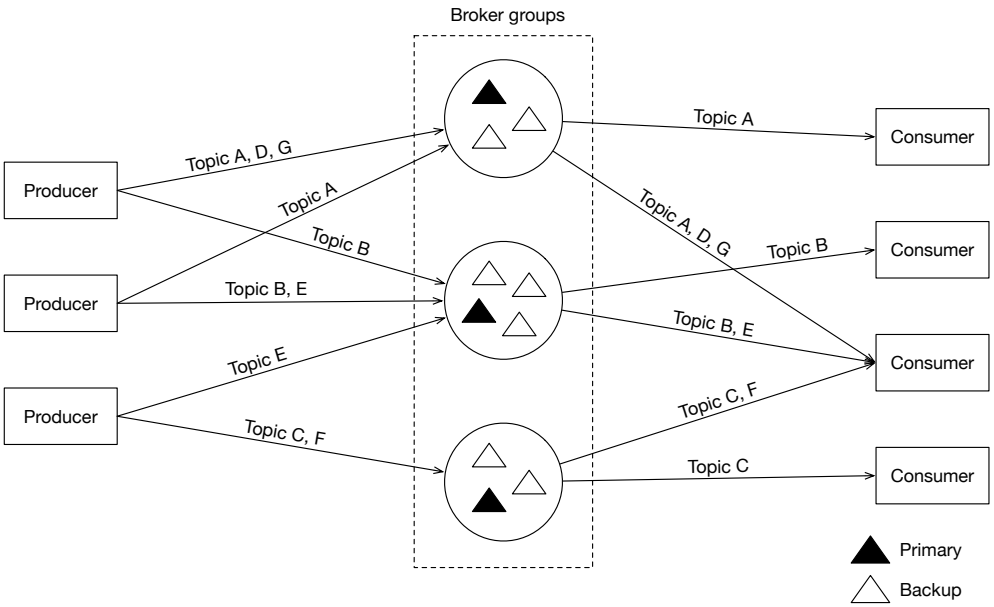

Figure 1. system Architecture

at-least-once RPC implementation since we allow producers to specify their desired maximum timeout and number of retry. We argue that the timeout and retry parameters are useful since different applications need to be customized to get different level of delivery assurance. The Producer can also specify how many records that can be asynchronously sent in batch to reduce network latency. We will discuss this feature shortly in more detail.

3.2. Consumer Design

End point users can also call the Consumer APIs to subscribe/unsubscribe to topics and receive records from the broker groups. Here we present a one-to-all semantics for the records received by the consumers: one message is pushed to all the consumers that subscribe to its corresponding topic to meet the semantic requirements of our target applications mentioned before, although some other applications like a distributed job scheduler requires that one message is delivered to only one of the consumers that subscribed. consumers can subscribe to topics at any time, but only the records of these topics that are produced after its subscription are guaranteed to be delivered.

3.3. Broker Design

The Broker service is organized into groups where each group has a primary server elected by ZooKeeper and zero or more backup servers.server accepts records published by producers, replicate to all the backups and push to the consumers the records which are successfully replicated to all the backups. The responsibility of backup servers while primary is working are mainly accepting replicated records, but any of the them may be promoted as a new primary in the case of primary failure.

Each record accepted by the primary would be assigned a logical time-stamp. As in most distributed systems, this time-stamp is required to be monotonically increasing within each group. Thus the time-stamps are totally ordered at each group. The Brokers also remembers which the largest time-stamped records have been successfully delivered to every consumer, which we refer to as *offset* like in Kafka. The primary guarantees that for each topic and each consumer, it sends a record with the next larger time-stamp than the last one. In other words, once a consumer gets a record of topic A with time-stamp t, it can never get a record of topic A that has a smaller time-stamp. A garbage collection thread runs on each broker server every 60 seconds to purge records that

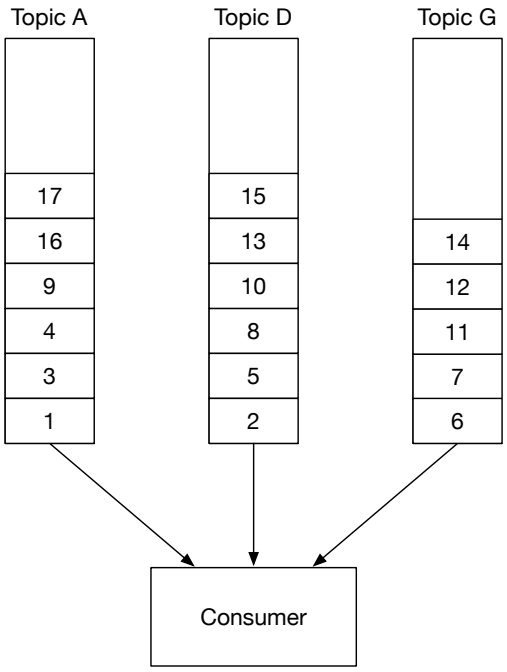

Figure 2. Record Storing Structure a Broker

have been delivered to all the currently subscribed consumers. Figure 2 shows the structure of how Brokers contains the records. Records are organized into queue-like data structures, each of which is related to a topic. The numbers represent the time-stamps assigned by the primary.

3.4. Role of ZooKeeper

Our system rely on the ZooKeeper service to support leader election with each broker group. The leader election service guarantees automatic re-election if the old primary fails and the new primary will be notified that it is elected. The producers can get a listener on the election result and update their own cache of the addresses of the primary brokers. Zookeeper is also leveraged for server and consumer status management in our system. Primary can be notified when a backup server is up or down, updating its cache of the backups list, and perform migration if needed. Consumers' subscription and unsubscription events are also sent to primaries through ZooKeeper so that the primaries can update the cache of consumer' offset. Another important usage of Zookeeper is to act as a central storage for preserving consumer offsets in order to save the bandwidth since otherwise the primaries need to send updated offsets to backups every time a record is delivered to a customer.

3.5. Consistency Model

Our system is able to provide a flexible consistency model. To be specific, the level of consistency our system provides is partially relied on the choice of producers. We provide each producer an import argument called *max-in-flight*. This parameter indicates how many records can be sent in a batch to the Brokers in order to improve throughput. Since we use asynchronous RPC, the order of records reaching the primary broker within a batch cannot be guaranteed, but the Producer library can guarantee to send one batch at a time after the previous batch receives all the replies. The Producer can wait to check the replies before sending another batch, or ignore the replies and just let the library send them one batch at a time. There are no ordering guarantee on the records of a topic concurrently sent by different producers.

The primary Broker assigns a monotonically increasing time-stamp to each record on the arrival. The delivery between brokers and consumers is independent of the producers. Suppose a primary server has received n topics. Let m_i be the number of consumers subscribed to the i-th topic ($1 \leq i \leq n$). In this case, there are $\sum m_i$ threads in the background sending records to consumers. For each consumer, records belongs to a topic can only be sent to it in the time-stamp order one at a time. Only when consumer acknowledges a record can the broker advance the offset of the topic of this record for this consumer. Since one consumer can subscribe to more than one topic, a consumer may get records from different topics in any order, but for each topic, the message received is strictly in time-stamp order. We use this method to get a good balance between latency and consistency.

Hence, combining the producer and broker behavior, the most strict consistency our system can provide is a *topic-wise sequential consistency*, if each producer publish records of only one topic and set the batch size to one . In this way, all the consumers that subscribe to a specific topic can observe that the order of records of this topic is exactly identical to the producer's perspective. As an example of the group chatting application, each topic represents for a group channel, a producer sends messages to a channel with the *max-in-flight* set

to one, and subscribed consumers can get messages from different groups in any order (although any message will not be delayed to much if the network is in good condition), but all the consumers in a channel can receive the messages from a producer in the same order as it sent.

3.6. Scalability

Our system support horizontal scaling by mapping different topics to different broker groups. This mapping is based on the hash code of the topic name. In order to ensure the uniformity between different groups, we have the following assumptions.

1. There are more topics than the number of groups.
2. All topics contains roughly the same number of messages.

As long as these assumptions hold, we can always increase the system's throughput by adding more broker groups, thus achieving horizontal scalability.

The number of groups should be decided before the system starts. As a result, there's no need to use a consistent hashing or migrate topics between broker groups. We call this *static scalability*. Section 6 discusses an improvement to our method.

3.7. Fault tolerance & Failure-Recovery

We made following assumptions to our fault-tolerance model, some of which will be discussed soon in the following paragraphs.

1. ZooKeeper cluster never fails.
2. Backups cannot be partitioned from the primary while still keeping connected with ZooKeeper.
3. All brokers can fail simultaneously, but primary cannot fail immediately after a broker is online, at least after the migration completes.

Our system provides fault tolerance with $n - 1$ fail-stop failures of brokers for a group of n servers under the above assumptions. In other words, as long as there is one server alive in a group, this group would be available and no records will be lost in this group. Here we assume However since each topic is mapped to one group, and we do not provide migration between broker groups, if one group has no server left standing, the topics that maps to this group will always be unavailable until some server in that group comes back.

3.7.1 Backup Failure

In the case of a backup failure, the primary will notice the event with the help of ZooKeeper and remove the backup from its cached backup list. It is possible that a backup fails after the primary sends out a replicate record to it and before receiving it or fails after receiving a replicated record from the primary and before it can reply to the primary. In either case, the primary cannot get acknowledges from all the backups after some timeout and will retry broadcasting records to backups. Since in our implementation, primary evicts a backup from its cached backup list only when being notified by the ZooKeeper, we assume that backups cannot be partitioned from the primary while still keeping connected with ZooKeeper. There are possibility that the producer may get a *fail* reply if it is not quick enough for the primary to be notified of the backup failure and remove it from its backup list.

When a backup in back online or a new broker is added to the group, the primary will also get the notification from the ZooKeeper. The primary fist add the backup to its cached backup list and now this backup is able to accept new replicate records. The primary then start a migration thread to migrate all the records to it. We assume that the primary cannot fail too soon after a broker respawns, at least after the migration completes to prevent the case that the not fully up-to-date backup is elected as the new primary.

3.7.2 Primary Failure

In the case of a primary failure, one of the remaining backups in the group will be re-elected as the new primary. The new primary needs to restore the largest time-stamp assigned by the previous primary by looking at its own queue and restore the offsets of consumers via ZooKeeper. Like in the backup failure case, the broker group may also become unavailable for a little amount of during the re-election delay.

4. Implementation

4.1. RPC

Throughout the system, we use RPCs for communications with Thrift protocol. All our RPCs are asynchronous, with some signal techniques to make it synchronous in some cases. We provide timeout and retry

```
class Producer<T> {
    Producer(Properties configs);
    Future<ProducerMetaRecord> publish(ProducerRecord<T> record);
    void close();
}

class Consumer {
    Consumer(Properties configs);
    void subscribe(String topic);
    void subscribe(String[] topics);
    void unSubscribe(String topic);
    void unSubscribe(String[] topics);
    void start();
    void stop();
    String[] subscription();
    Map<String, Queue<ConsumerRecord>> records();
}
```

Listing 1: Client APIs for Producer and Consumer

for each RPC client, making it more robust to network problems as well as node failures. Thus our RPC has an at-least-once semantics, leading to a problem of duplicated messages on RPC servers. For example, when producers send records to brokers and doesn't get the response in the user specified timeout period, the producer library will automatically resend the same record. However the previous record may have been correctly delivered to the broker and get processed. To solve this problem, the producer library will assign a unique identifier to each record. The broker remembers what records it has seen and accepts the record if the previous one actually fails or just reply *success* if the previous one already succeeded.

We leveraged *UUID* as the unique identifier for each record for two reasons. First it's easy to use with a simple API call in Java. Moreover it has a very strong guarantee of being unique with an extremely low probability of conflict that can be ignored. It is estimated that only when producing 1 billion *UUIDs* every second for the next 100 years can the probability of creating just one duplicate would be about 50%. So *UUID* is quite reliable in this sense.

4.2. Client APIs

We list some major producer and consumer APIs in Java that are provided for end point users in Listing 1.

4.2.1 Producer API

The Producer API is very simple to use. A user need to first create a producer instance with configurations including *retry*, *timeout* and *max-in-flight*. After calling *publish()*, a future instance is returned. The user can later check the future instance for the reply message. Although *publish()* is an asynchronous method, users can wait on the future instance to get reply in order to make it synchronous if need.

4.2.2 Consumer API

The Consumer has a richer API than the Producer. Once a consumer instance is established, the start() method should first be called for being able to be connected by broker servers. A consumer can subscribe or unsubscribe to one or more topics at time. Once it subscribes to some topics, it will start receiving the records of these topics in the background. In the meantime, the *records()* method can be called to explicitly retrieve these received records. For example, the consumer can spawn a thread for each topics it subscribed and get records from the queue in the order specified by the producer.

5. Test and Evaluation

We conducted various experiments to test the correctness of message delivery in cases of failure as well as the performance of the system including throughput and latency. The environment we used for testing is as follows: 100 Mbps WLAN, two MacBook Pro laptops with one running three broker groups with three replicas per group and another one for simulating both producer and consumers. The network in our test environment is not very stable with a average RTT of 50 ms and package loss of 8.1%.

5.1. Fault-tolerance test

We tested the correctness of message delivery in the cases that primary or backups fails following the assumptions listed in section 3.7. We use both manual and random ways to kill a broker server. We checked that the delivered message is still in order and no message is lost because of migration or period purge, although there is a little amount of time that the producers cannot get a "succ" reply from the broker for some records because the gaps between backup failure and primary getting notification as well as the gaps between the primary failure and re-election.

5.2. Throughput Evaluation

The throughput of our system could be measured in the following aspects. The basic setting used in all experiments is 9 brokers across 3 groups. Messages are distributed evenly in 10 topics, each with a size of 100 bytes.

Producer Throughput We measured the throughput of the producer when there's no consumer in the system. This would be the upper limit of our system since all messages will be merely stored on the brokers (primary as long as backups). The variables in this experiment are the number of messages sent at a time and the *max-in-flight* parameter. As is showed in Figure 3, increasing number of messages and *max-in-flight* will both increase the throughput, with a limitation of about 1,000 messages per second.

Consumer Throughput This experiment mimics the most simple scenario where there's one producer and one consumer in the system. The result shows that

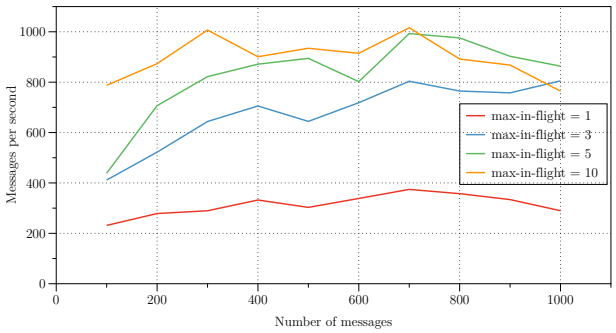

Figure 3. Producer Throughput

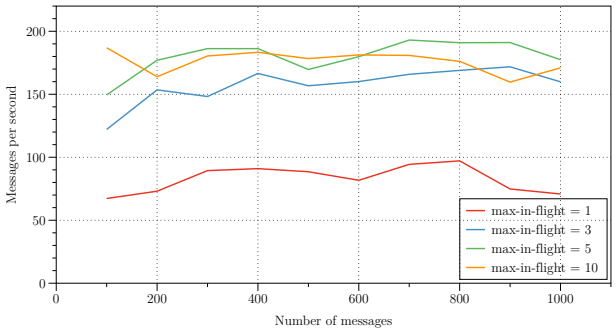

Figure 4. Consumer Throughput

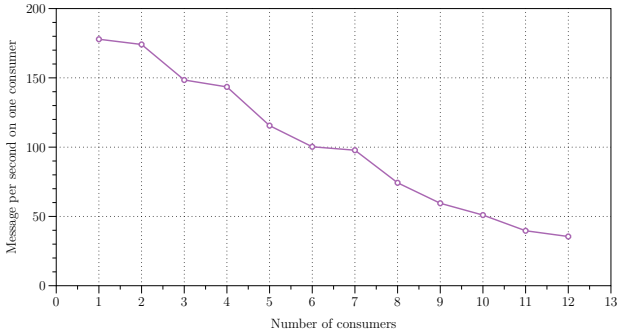

Figure 5. Throughput of Multiple Consumers

the limit of the throughput is about 200 messages per second, which is much lower than the previous one.

Throughput vs number of consumers Finally, we measured how throughput changes when the there are more than one consumer. Since we are using a publish-subscribe model here, the bottleneck of the system will most likely be the consumers, since every subscribed consumer will need to obtain a copy of all messages. The result is showed in Figure 5.

5.3. Latency Evaluation

The latency is defined as the time elapsed between the producer begins publishing the message and the consumer finishes receiving the message. Apparently, this metric is affected by many variables and our result shows the minimal latency of our system is about 100 ms, mostly due to the terrible network environment during the experiment.

6. Conclusion and Improvements

The paper descries the design, implementation and performance of TritonMQ, a real-time fault-tolerant distributed message queue that uses an in-memory push-based model and provides a flexible consistency model which is able to serve applications like group chatting, log processing and news feeding. We also found some limitations in our systems and we propose several improvements that could be applied to enhance its flexibility and reliability.

6.1. Dynamic Scalability

One of the limitations in our system is that the number of broker groups is fixed upon system startup. When adding backups to a broker group, although availability can be improved, it will actually decrease the performance since backups are dumb and all the primary does all the job of receiving and sending records. A solution is to let these backups share part of the primary's responsibility of pushing records to consumers. Thus, when more backups are added, the workload is better amortized to more backups and the better throughput can be achieved. This approach requires some extra synchronization among the replicas, but since storing the consumer offsets on ZooKeeper has decoupled some of the synchronization work, the complication is mostly on failure recovery.

6.2. Better Fault Tolerance

As Section 3.7 mentions, we assume several assumptions to achieve the level of fault tolerance we claims. Some of these assumption may be relaxed or thrown away with some improvements on our current implementation. For example, we can relax the first assumption to be allowing ZooKeeper to fail, but primary cannot fail until ZooKeeper service is available again. We may also get rid of the third assumption by making some modifications to ensure a not up-to-date broker can never be elected as the primary.

6.3. Better Load Balance

In real world, some topics can have far more messages than others do. In our system, topics are mapped to a fixed broker group. This may lead to unbalanced loads between broker groups if the number of messages of different topics are highly unbalanced. To solve this problem, we could apply a similar idea as the partitioning mechanism in Kafka. Records of a topic could be partitioned into different groups, and consumers can read Records from different broker groups, which can balance the loads across different broker groups.

References

[1] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9, 2010.

[2] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[3] RabbitMQ. Highly Available (Mirrored) Queues.

[4] A. Videla and J. J. Williams. *RabbitMQ in action*. Manning, 2012.