Project 3: Octopi

Andrew Chung
Jiunn Haur Lim

Nov 20, 2012

1 Overview

Octopi is a websockets-based messaging service that allows producers to publish messages under specified topics, and consumers to consume messages for requested topics. Topics are assigned or hashed to partitions, each containing multiple brokers. A single topic will be managed by all brokers in the same partition, and a single partition may manage multiple topics. A consumer may consume from any one of the brokers in the partition. In case of broker failures, the consumer will reconnect to one of the replicas in the same partition and resume from where it left off. Producers send messages to an elected leader in the corresponding partition, which streams it to the other replicas before acknowledging the produce request. [1]

2 Design

According to the authors of Kafka[**?**], it is increasingly likely that sequential reads/writes relying on the operating system's pagecache of disk files yield better performance than maintaining an in-memory cache of the data. This is due to several OS optimizations, one of which allows the process relying on pagecache to effectively use all of the available free memory. As such, we will follow their recommendations and create a pagecache-centric design where messages are written to a log file as soon as they are received. A disk flush will be called after some number of seconds/messages to persist the pagecache to disk.

The following narrative will help to explain the vocabulary used in the rest of this document:

> *Producers* send *messages* under a specific *topic* to *brokers* by making produce requests. Brokers save these messages to a *log*. *Consumers* subscribe to *topics* by connecting to the brokers, and receive messages from the brokers as they are produced.

[1] Several parts of the system design are inspired by Kafka, a distributed publish-subscribe messaging system written in Java and Scala.

2.1 Register

For the initial tiers of this project, there will be a centralized registration node that handles broker membership. It is responsible for remembering the identities of the brokers, the partitions they belong to, and the leader of each partition. It also plays a substantial role in failure and recovery: when the leader of a partition fails, the register is responsible for registering the new leader and assigning the rest of the brokers sharing the same partition as followers.

2.2 Brokers

Brokers are processes that receive messages from producers and relay them to subscribing consumers. Each broker instance can handle multiple topics, and messages for each topic are stored in a log file in the same order they are received from the producer. This log file is replicated across brokers in the same partition. The format of messages sent by producers is as follows:

```
version:    1 byte
checksum:   4 bytes   // crc32
request_id: 1 byte    // sequence number
payload:    n bytes
```

Each topic has a log file containing all of the messages received from producers. Each log file is a sequence of messages, and the on-disk format of the messages is as follows:

```
length:     4  bytes (value: 1 + 4 + 32 + n)
version:    1  byte
checksum:   4  bytes
request_id: 32 bytes  // sha256(host:port:seqnum)
payload:    n  bytes
```

Where necessary, the ID of a message is its offset in the log file. Every broker maintains two pointers for each log file: the tail of the file, and the highwater mark. The highwater mark is the offset of the last committed message for the topic (futher explained in the next section.)

2.3 Consumers

Consumers may be web browsers, desktop clients, or web applications. They subscribe to topics and receive messages from brokers. For any given topic, a consumer must contact the registration node to discover a broker to consume from. Octopi guarantees that these messages will be sent in the same order as they are received from the producers.

2.4 Producers

Producers are processes that send messages to brokers. For any given topic, the producer must contact the registration node to discover the responsible leader before sending produce requests to it. Producers should mostly be web applications.

3 Replication and Robustness

The register will use consistent hashing to determine the partition responsible for any given topic. Within each partition, mulitple brokers manage the assigned topics and synchronize the log files for each topic.

Using the primary-backup approach, each partition has n replicas and can handle $n-1$ failures. As mentioned in the previous section, each replica maintains a pointer to the tail of their log file and to the highwater mark. When a produce request is made, the following algorithm will be used to synchronize all replicas:

1. Leader receives produce request from producer
2. Leader forwards enclosed message to all followers
3. Follower appends the message to log file
4. Follower sends acknowledgement to leader
5. Leader waits for acknowledgements, and appends message to log file
6. Leader instructs followers to commit the message and advances highwater mark
7. Leader sends acknowledgement to producer
8. Follower receives commit and advances highwater mark

Consumers may only read messages up to the highwater mark.

3.1 Register Failure

The register is a single point of failure for Octopi. If it fails, data regarding broker membership are lost and cannot be recovered. A later version of the project will use consistent hashing and the Chord algorithm to manage partitions, but that may be beyond the scope of this proposal.

3.2 Consumer Failure

The broker does not store state about consumers, except for open connections and an read offset for each consumer. If a consumer fails, the connection is closed and the read offset is deleted. The recovered consumer must make another consume request with the ID of the last message received.

3.3 Broker Failure

In case of broker failure, consumers must send a consume request to another broker with the ID of the last message received. The new broker will use the ID as the offset into the log file and serve messages beginning from that offset.

During a produce request, the leader may fail during any one step of the algorithm outlined above. If the leader

1. fails before forwarding the new message, then no state has changed. The producer will time out and resend request to new leader.
2. fails while waiting for acknowledgements, then followers have appended the message but not advanced the highwater mark. When a new leader is elected, it should start from the last highwater mark.
3. fails after instructing followers to commit, then followers have committed the message, but the producer will attempt to resend the request to the new leader. The new leader will compare the `request_id` of the received message against that of the last committed message (for that topic) to check for duplicates.

For the last case, since the leader commits one message at a time for each topic, there should be at most one possible duplicate request.

If a follower fails, the recovering follower should consume messages from the leader after its highwater mark to catch up with the other replicas.

4 Tiers/Development Schedule

4.1 Libraries and Basic Functionality

1. Producer library
2. Consumer library
3. Broker library with logging and simple robustness
4. Registration server library

4.2 Replication, Failure Handling, and Robustness

1. Leader elections
2. Catching up for delayed brokers
3. Fail-over handling

4.3 Advanced Implementations[2]

1. Topic addition
2. Partition addition with new brokers
3. Rotation of log files when they get too large
4. Replacement of Registration Node with Distributed Hash Ring and Use of Chord Algorithm

5 Test Plan

5.1 Libraries and Basic Functionality

1. Registration test for one broker
2. Registration test for multiple brokers, one partition
3. Registration test for multiple brokers, multiple partitions
4. Broker assignment test for consumer with one broker, one partition
5. Broker assignment test for consumer with multiple brokers, one partition
6. Broker assignment test for consumer with multiple brokers, multiple partitions
7. Message and logging test with one producer, one consumer, one broker
8. Message test with multiple producers, multiple consumers, multiple brokers
9. Multiple consumer close connection and reconnect test
10. Simple failure case leaving no brokers
11. Simple robustness test with two brokers, one failure
12. Failure at different points in algorithms and handled in the ways described above

5.2 Replication, Failure Handling, and Robustness

1. Re-entering test of brokers
2. Logging consistency test among two brokers
3. Logging consistency test among multiple brokers

[2] If time allows

4. Logging catch-up test with two brokers, follower early failure and then re-enter
5. Logging catch-up test with hanging brokers
6. Leader election test with two brokers, leader failure
7. Leader election test with multiple brokers, leader failure
8. Leader election test and logging catch-up test with two brokers, leader failure, follower promotion, and leader catch-up after re-enter
9. Re-referral of consumer of assigned broker failure test
10. Re-referral of producer to new leader of leader broker failure test
11. Stress test of multiple producers, consumers, and brokers

5.3 Advanced Implementations[3]

1. Logging stress test with many messages to see if files are split correctly
2. Addition of topics test
3. Addition of partitions test
4. Chord algorithm accuracy test
5. Addition of broker into distributed hash ring

References

[3] If time allows