Distributed Message Queue

CPSC 416 Project 2 Final Report
Dec. 8th, 2018

Srinjoy Chakraborty **| a3x9a**
Justin Awrey **| i8b0b**
Wen Pan **| t6o0b**
Shuaiqi Ge **| t0r2b**

Abstract

In order to fully understand our proposed problem and solution, we must first understand the following:

- Corporations have pivoted to a microservice architecture which heavily relies on streaming data to and from logically separated units within a much larger overarching architecture.

- As time progresses, the raw size of our data collection continues to grow. Corporations such as Google handle billions of searches daily.

The above two facts do not complement each other - directly streaming high volumes of data between many different services is neither a flexible, nor scalable solution. For example, it is common for applications to want to produce or consume data at different rates, perhaps buffer data in the case of a failure, or batch work to be sent through the network at once (among other cases). We propose to streamline the distribution of data within a microservice architecture by introducing a distributed message queue, acting as an intermediary, to which individual services can both publish and consume messages.

In our solution, consumer services can read from the message queue at different rates and different indices. Similarly, producer services can produce to the message queue at different rates. Multiple message queues are supported - that is, messages are categorized according to a (topic, partition) tuple (where topic is a string and partition is an integer). When client services wish to consume or produce from a specific message queue, they must specify the desired (topic, partition) tuple. Message queues are distributed and failure tolerant. In all, our system allows for a greater level of flexibility and tolerance within the network - notable systems with similar goals that exist today are the MQ (Message Queue) family of platforms: ActiveMQ, ZeroMQ, and RabbitMQ, as well as Apache Kafka. Our implementation is heavily inspired by the latter.

Table of Contents

1.0 | Topology and Description of Roles

1.1 Topology Requirements

Below a is a diagram depicting the topology of a the system with two different message queues being aggregated on two different broker clusters. We enforce that each leader should have three followers, and each manager should have 3 backups. This set-up yields strong consistency in the presence of up to 3 node failures (failure and joining semantics discussed in a later section).

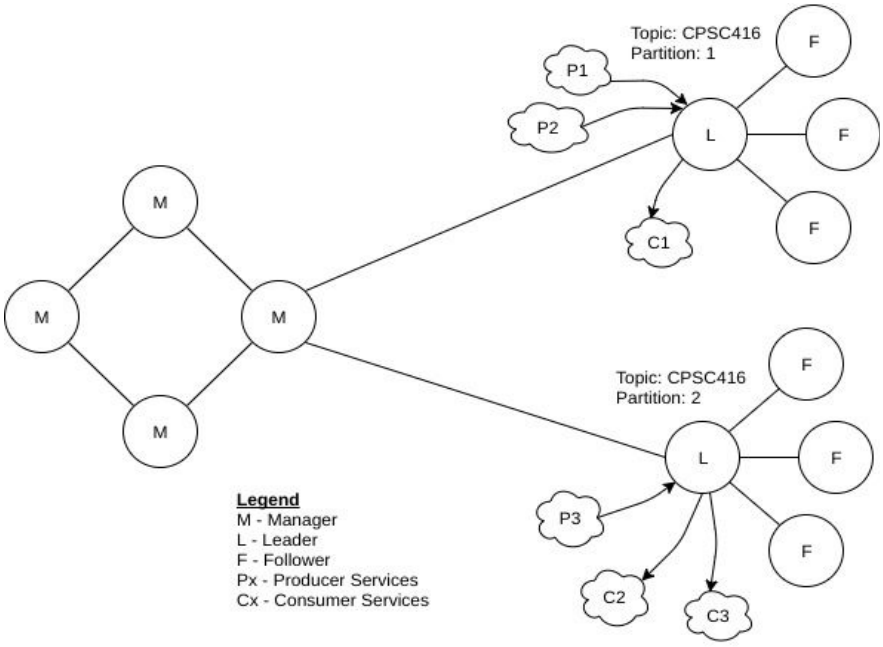


Fig. 1:
Normal operation of consumer and producer services
connected to different (topic, partition) message queues.

1.2 Node Roles

Our system has three types of nodes:

1.2.1 Managers

Managers are aware of the entire topology. They are responsible for coordinating client services with the correct leader nodes (which hold message data). The services that managers provide to producers/consumers are analogous to those that DNS provide to web clients. In addition, managers are involved in provisioning available nodes during the joining process.

1.2.2 Leaders

Leaders hold a copy of the message queue for a specific (topic, partition) tuple. Producers and consumers directly connect to Leaders in order to append to / consume from the message queue. Messages received by leaders from producers are time-stamped at sent time. Leaders will order messages coming from multiple producers according to their timestamps. In this way, our system is trusting of clients to provide accurate timestamps. Leaders flood their messages to multiple followers - leaders and followers always have consistent copies of a specific message queue.

1.2.3 Followers

Followers serve as message queue backups. There are multiple followers connected to one leader - leaders are responsible for coordinating the consistency of data between multiple followers and themselves. The leader and follower cluster is referred to as a “broker” or a “broker cluster”.

1.3 Normal Operation

1.3.1 Publish/Subscribe

The core functionality that our system provides is the ability for multiple services to append messages to a message queue, while other services simultaneously consume from the same queue. Messages are not removed from the message queue on consummation; a message is instead automatically deleted if it has been retained by a broker cluster for some long period (typically a couple of days). This allows messages to be consumed by multiple consumers at their own pace. The whole procedure works, at a high-level, in the following manner (major APIs used to achieve this functionality are described in a further section):

Producing a message

- Producers queries a manager to obtain the address of a leader node with their desired (topic, partition) tuple. If their desired (topic, partition) tuple does not already exist, managers will provision new nodes and supply the address of their leader.
- Producers connect directly to the provided leader.
- Producers send messages to the leader as they wish.
- Upon receiving a message from a producer, leaders disseminate the messages to their connected followers.

Consuming a message

- Consumers query a manager to obtain the address of a leader node with their desired (topic, partition) tuple.
- Consumers connect directly to the provided leader.
- Consumers read from the message queue at any index they wish.

- The consumer's index of the message is stored at both leader and follower nodes and will be consistent across the cluster to ensure consumer progress on consumption isn't lost during a leader node failure.

The major API's that implement this functionality can be found in *Appendix B*.

1.3.2 Fail-Stop Model

We employ a fail-stop model as opposed to a fail-recovery model. In the case of node failure, our system does attempt to restart the failed node. Managers instead provision a new, available node to take on the role of the failed node. Humans must manually restart the failed node if they wish for that particular node to rejoin the topology. Note that upon failure, data and system state still remains consistent.

1.3.3 Zero copy policy

In order to improve the performance of transferring messages, our implementation avoids explicitly caching messages in memory. Instead, we rely on the underlying file system page cache by calling OS APIs to implement Direct Memory Access (DMA). What this means is that instead of reading messages from disk, copying them to memory, and then sending them through the network, we effectively skip the intermediate copy into memory and send messages straight from disk to the network. As reading and writing messages to disk is an extremely common task, eliminating this single copy greatly improves the overall speed of the system.

2.0 | Key Failure Cases and Join Strategies

Whenever a new node join needs to occur due to a failure, it is the managers duty to choose a free node and assign to it a correct role. For this to happen, the manager needs to be aware of a list of free nodes which are ready to be provisioned. For this reason, when all nodes enter the topology for the first time, they broadcast to a manager their IPs, as well as set their status to be "ready for allocation". By doing this, managers gain a global knowledge of which nodes are and aren't available, and as such are prepared to coordinate the joining process when a node failure occurs. All transactions which involve the synchronization or data between multiple nodes use 3-phase commit to ensure consistency. A more detailed explanation of 3-phase commit is explained in section 3.2.2.

2.1 Manager Failure Cases

2.1.1 Primary Manager Fails

When the primary manager fails, the two other manager nodes in the manager cluster will detect this failure via 'fdlib'. On detection, the manager with the higher ID will be automatically elected as the new leader and send a notification to all the leaders to notify them of the change. If there are free

nodes available as per the process described above, a new manager will be provisioned as the new backup manager in the cluster.

2.1.2 Backup Manager Fails

If the primary manager notices a backup manager has failed via ‘fdlib’, it will provision a new backup manager from any free nodes available and send them a backup copy of the topic map on initialization. It will then send the new array of manager IP’s to all the leaders to let them know.

2.2 Leader/Follower Fail Cases

2.2.1 Leader Fails

When a client attempts to connect to its leader node and detects that the leader has failed, it will report this failure to a manager node (users will not manually have to do this, it is done behind the user-facing API). The manager will then promote one of the followers from that broker cluster to a leader, and provision a new follower from its list of available nodes. The new leader and followers then resync their message queue contents. While this entire process is happening, the entire broker cluster is blocking incoming messages from arriving in an effort to keep the broker cluster in sync. The manager will then provide the IP of the newly provisioned leader node to the client, who will then attempt to reconnect.

2.2.2 Follower Fails

When the leader detects that one if its followers has failed, it sends a request to a manager node to provision a new follower, as well as sends the manager the IP of the node that has failed. The manager can then amend its free nodes list, and provision a new follower, if nodes are available, and send the newly provisioned IP to the leader. The leader needs to resync its message queue contents to the new follower. While this entire process is happening, the entire broker cluster is blocking incoming messages from arriving in an effort to keep the newly provisioned follower and the old followers in sync.

3.0 | Distributed Principles

3.1 CAP

3.1.1 Consistency

We maintain consistency of data by using a three phase commit across all operations in the system (except basic information messages). The manager cluster will commit across it’s nodes on any topology change and the each leader will commit across their respective followers when processing messages. During provisioning a new leader by promoting an existing follower, the new leader will

query the other nodes in the cluster to synchronize state by aborting/committing any outstanding transactions (discussed in 3-phase commit section).

3.1.2 Availability

Although our three-phase commit is a synchronous process, we have implemented a timeout mechanism to prevent the process from blocking forever. During the joining process when a new topic is being provisioned as a new cluster the provider is also blocked and must wait. The manager will also block when pushing backups to its cluster and will block on requests such as ‘createTopic’ or ‘getLeader’.

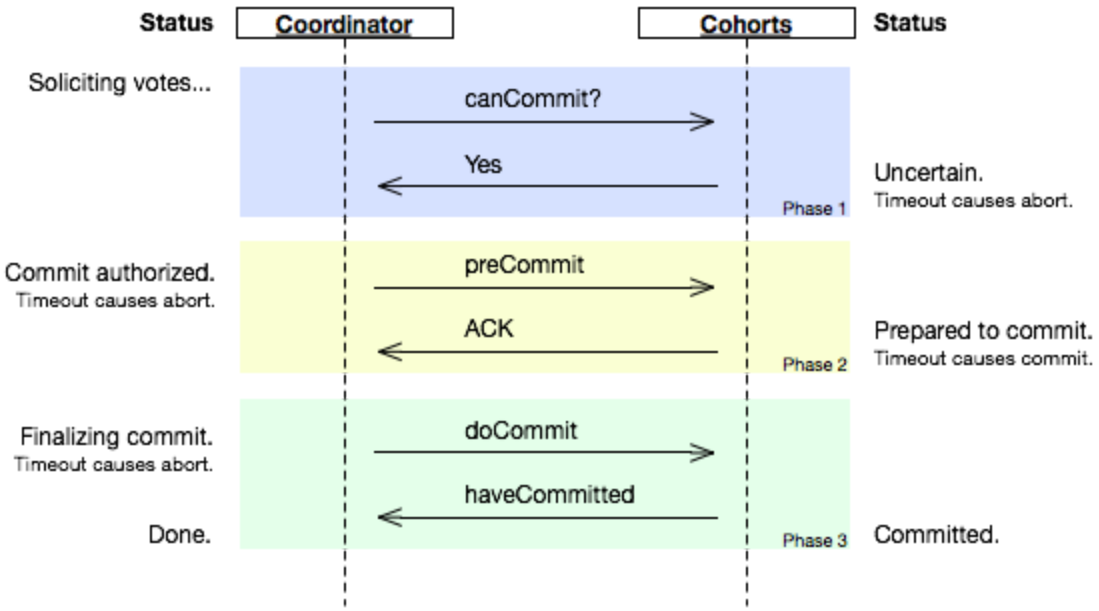
3.1.3 Partition Tolerance

The data can be partitioned into a new topic/partition combo and copied over entirely or partially by a provider. Our current implementation does not support sub-partitioning but this could be easily done using a 3-phase commit to synchronize across all messages from the leader. This would reduce availability during this period.

3.1.4 Our System

We chose to have strong consistency and partition tolerance and compromise some availability for providers and consumers.

3.2 3-phase commit



https://en.wikipedia.org/w/index.php?curid=16452453

3.2.1 2-phase vs 3-phase

When deciding how we wanted to handle synchronization, we chose to go with a three-phase as it allows for stronger consistency for our system. In a 2-phase commit, the coordinator could fail after initiating phase 1 which would result in some nodes that received the message and some that did not. This would mean any follower that is promoted would not be able to tell if has the correct latest index as it may have one less or one more than other followers. Using a 3-phase commit lets us avoid this scenario by having the promoted follower query the other nodes on what their last commit state was. Using this, and comparing the new leader's state, the whole cluster can synchronize by either all agreeing to commit or abort.

3.2.2 Commit Failure Scenarios

1. **During Phase 1**

During the first canCommit phase of our commit protocol, the leader of the cluster will initiate a commit proposal to all its peers/followers and blocks to wait for response. If any of the peers have failed to respond on either timeout or disapproval, the transaction will be aborted and the state will be notified in a cache that utilises the transaction hash as a primary key. If all peers have decided to proceed, set the leader to WAIT state, and followers to APPROVE state. Any failure during this phase will abort the transaction.

2. **During Phase 2**

In second phase of the protocol, the leader will send a prepare message its peer and set the its' state to PREPARE upon receiving agreement from all the peers. On the receiving end, the peers will set their state to PREPARE as well if they agree to proceed. Action to recover after failure during this phase depends on the state of peers. If connection times out before reaching to any of the peer, their states will remain unchanged (APPROVE). If any of peer has received the message from leader and set its state to PREPARE, commit the transaction during the recovery phase.

3. **During Phase 3**

After the leader collects all the responses from peers during phase 2, it'll send out a commit message to all the peers. The peers will set its state to COMMIT after the transaction have been executed. Finally, the leader will commit the transaction upon receiving ack from all peers. Any failure and termination during this phase will be recoverable through a recovery process.

4. **Recovery**

During phase 1 of the protocol, a recovery check will be performed asking all peers about its state of the transaction, if transaction have been executed before but aborted due to failure, at least one of the peer will have a state for the transaction. Recovery phase will initiate if one of the peer reports a PREPARE or COMMIT. The recovery process will involve commiting the transaction for

peers in PREPARE state. Peers with COMMIT state are assumed to already have the transaction committed before the failure has occurred.

3.3 DHT Routing

When a producer requests to create a new (topic, partition) queue, managers will pick up specific number of brokers by consistent hashing functions and return their IPs. By this way, the load can be divided evenly between all of the brokers. Consistent hashing operates independently of the number of brokers in a distributed hash table by assigning them a position on a ring circle, which makes our system more scalable. When a provider applies manager nodes to create a new topic and request a specific number of replica nodes, the system will call consistent hash function with specific key (e.g. nodeID) to get list of brokers' IP. The first one would be leader and the others will be regarded as followers.

4.0 | Specific Use Cases

The core motivation behind an intermediary distributed messaging queue is described in section 1. At a high-level, any infrastructure that benefits from having a intermediary data buffer between microservices can benefit from the use of our system. More specific use cases are as follows:

- Creating centralized data feeds for machine learning
- Facilitating ETL (Extract, Transform, Load) services
- In-order log aggregation
- Buffering data for consummation at a later time
- Use as a structured commit log

5.0 | Limitations & Future Improvements

5.1 Limitations

As messages are persisted on leaders (and replicated 3 times on followers!) our message queue implementation requires a large amount of data retention. Old messages expire after a period of days, however it may still be an issue to receive extremely heavy load over a short period of time. Other limitations include the fail-stop model, as dead nodes must manually be brought back up and re-added to the available nodes pool. It is also important to note that while the DNS-style coordination between managers and clients greatly simplifies the manager logic, it does impose the extra task of manually connecting to a leader node onto the client.

5.2 Future Improvements

Beyond amending the limitations as described above, the following improvements could be made in future revisions to the system.

5.2.1 Dynamically setting replication for a topic

Currently we are globally setting the replication of a particular topic/partition to be 3 follower for every leader. In the future we would like to set this dynamically to allow for higher or lower levels of replication.

5.2.2 Splitting a topic into multiple partitions

We would like to have a way to subdivide an existing partition into multiple other ones. This would mean we would need to create a protocol to inform all subscribers/providers of the change as well as coordinate with the manager to allocate a new cluster and change their map. This may introduce complications of consistency and availability when the topic is being divided and would also require a lot of free nodes and thus due to time limitations, we could not build it into this version.

6.0 | Work in progress sections

The following is a list of items which are still a work in progress at this time. All other implementation details as mentioned in this report are completed.

- Consistent join semantics when joining new nodes to broker clusters
- Producer and consumer client facing APIs
- Complete fdlib integration into the manager cluster

Appendix

A. ShiViz Diagram

https://drive.google.com/file/d/1x7O_Uc9VQcHVl5NJGYsPwVqTJ0NpS3if/view?usp=sharing

B. Major API's

The API's here are written as local function calls to simplify the interaction of rpc calls and the network but the inputs/outputs are still the same after a period of time.

B.i. Producer

The provider pushes data as a client to the network on a particular topic/partition. It needs to initially request the IP of the leader of particular existing topic via one of the manager nodes or if they would like to create a new topic/partition.

```
/*
* Dials the manager IP and requests to create a new topic with a given topic name and partition number.
*/
String leaderIP = createTopic(name, partition) @ config.managerIPs[0]

/*
* If a topic and partition exist, this will push a message to it. A leader IP is required prior to being
* able to push.
*/
bool confirm = pushMessage(topicName, partition, msg) @ LeaderIP

/*
* Dials the given manager IP and requests the leaderIP.
*/
String leaderIP = getLeader(topicName, partition) @ config.managerIPs[0]
```

B. ii. Consumer

The consumer can subscribe to a stream by continually polling a topic/partition with an index. If they receive a message, they will increment the index to get the next message.

```
/*
* Continually polls the topic/partition for messages.Increment the index after a message has been received.
*/
String msg = pollTopic(name, partition, index) @ LeaderIP

/*
* Returns the latest index the leader has for the topic/partition.
```

```
*/
int index = getLatestIndex(topicName, partition) @ LeaderIP

/*
* Gets the leader of the topic/partition from the manager on init.
*/
String leaderIP = getLeader(topicName, partition) @ config.managerIPs[0]
```

B. iii. Manager

The consumer can subscribe to a stream by continually polling a topic/partition with an index. If they receive a message, they will increment the index to get the next message.

```
/*
* When a broker inits they contact the manager to let the manager they are available for allocation.
*/
registerBroker(brokerID)

/*
*
*/
startLeader(name, partition, index) @ LeaderIP

/*
*
*/
pushBackup(topicMap) @ config.peerManagerIPs[]

/*
* Gets the leader of the topic/partition from its map and returns it.
*/
String leaderIP = getLeader(topicName, partition) @ requester's IP

/*
*
*/

String followerIP =followerFailed(topicName, partition) @ config.managerIPs[0]

/*
*
*/
String leaderIP =leaderFailed(topicName, partition) @ config.managerIPs[0]

/*
*
*/
String newManagerIP = managerFailed(managerId)
```

B.iv. Leader

B.v. Follower

C. Demo Script

Throughout the demo, all nodes will be brought up and down on azure VMs via command line programs.

- Demonstrate *normal operation* of your system (no failures/joins) with at least 3 nodes.
    - We will first bring up the minimum amount of nodes required for a failure tolerant system. That is 8 nodes (4 managers, and one broker cluster consisting of 1 leader and 3 followers). We will bring up 4 manager nodes, and the rest will be brought up in "standby mode", i.e. as ready to be provisioned nodes which will join the topology accordingly when our client programs request to create a (topic, partition) tuple.
    - We will then request to create a (topic, partition). This will automatically provision a new broker cluster of 4 nodes.
    - We will then produce messages to the (topic, partition) from two different client producers.
    - We will then read these messages from the (topic partition) from two different client producers.
- Demonstrate system can survive at least 3 node failures
    - We will first bring up more nodes into "standby" mode. These nodes will automatically undergo the join process when we cause our node failures.
    - We will bring down 3 nodes, via ctrl-c in their respective VMs. These could be any assortment of manager, leader, and follower nodes.
    - We will then show that we can still produce and consume to / from the same topic partition.
- Demonstrate system can *join and utilize* at least 3 new nodes
    - On the previous node failures, 3 more nodes will be provisioned by the managers and automatically join the topology.
    - We will show, through logging, that these nodes are being fed data and synchronized with their respective clusters.
    - At this point, we *may* take down all other backup nodes (not the newly provisioned 3!) and show the system functioning to prove the utilization of the newly provisioned nodes.