Kafka Replication

Kafka Replication High-level Design

The purpose of adding replication in Kafka is for stronger durability and higher availability. We want to guarantee that any successfully published message will not be lost and can be consumed, even when there are server failures. Such failures can be caused by machine error, program error, or more commonly, software upgrades. We have the following high-level goals:

1. Configurable durability guarantees: For example, an application with critical data can choose stronger durability, with increased write latency, and another application generating a large volume of soft-state data can choose weaker durability but better write response time.
2. Automated replica management: We want to simplify the assignment of replicas to broker servers and be able to grow the cluster incrementally.

There are mainly two problems that we need to solve here:

1. How to assign replicas of a partition to broker servers evenly?
2. For a given partition, how to propagate every message to all replicas?

Replica placements

Initial placement

(only create topic, make decision based on current live brokers (manual create topic command); rebalance command (recomputed assignment based on new cluster topology, every partition is moved) and make the move)
We first use an administrative api to create the initial set of brokers:

```
create cluster with brokers broker-0, broker-1, broker2
```

We then use another administrative api to create a new topic:

```
create topic topicX with 100 partitions
```

After that, the following information will be registered in zookeeper:

1. a list of brokers;
2. a list of topics and for each topic, a list of partitions.

For better load balancing, we want to over partition a topic. Typically, there will be many more partitions than servers. For each topic, we want to divide the partitions evenly among all the brokers. We sort the list of brokers and the list of partitions. If there are n brokers, we assign the ith partition to the (i mod n) th broker. The first replica of this partition will reside on this assigned broker and is referred to as the preferred replica of this partition. We want to place the other replicas in such a way that if a broker is down, its load is spread evenly to all surviving brokers, instead of to a single one. In order to achieve that, suppose there are m partitions assigned to a broker i. The jth replica of partition k will be assigned to broker (i + j + k) mod n. The following figure illustrates the replica assignments for partitions p0 to p14 on brokers broker-0 to broker-4. In this example, if broker-0 goes down, partitions p0, p1, and p2 can be served from all remaining 4 brokers. We store the information about the replica assignment for ach partition in Zookeeper.

Incrementally add brokers online

We'd like to be able to incrementally grow the set of brokers using an administrative command like the following.

```
alter cluster add brokers broker-3, broker-4
```

When a new broker is added, we will automatically move some partitions from existing brokers to the new one. Out goal is to minimize the amount of data movement while maintaining a balanced load on each broker. We use a standalone coordinator process to do the rebalance and the algorithm is given below.

Take brokers offline

We'd also like to support shrinking a cluster by taking existing set of brokers offline using an administrative command like the following.

```
alter cluster remove brokers broker-1
```

This will start the reassignment process for partitions currently hosted on broker-1. Once that is complete, the broker-1 will be taken offline. This command will also delete the state change path for broker-1

Data replication

We'd like to allow a client to choose either asynchronous or synchronous replication. In the former case, a message to be published is acknowledged as soon as it reaches 1 replica. In the latter case, we will make our best effort to make sure that a message is only acknowledged after it reaches multiple replicas. When a client tries to publish a message to a partition of a topic, we need to propagate the message to all replicas. We have to decide:

1. how to propagate a message;
2. how many replicas receive the message before we acknowledge to the client;
3. what happens when a replica goes down;
4. what happens when a failed replica comes back again.

We introduce existing replication strategies in Section 2.1. We then describe our synchronous and asynchronous replication in Section 2.2 and Section 2.3, respectively.

Related work

There are two common strategies for keeping replicas in sync, primary-backup replication and quorum-based replication. In both cases, one replica is designated as the leader and the rest of the replicas are called followers. All write requests go through the leader and the leader propagates the writes to the follower.
In primary-backup replication, the leader waits until the write completes on every replica in the group before acknowledging the client. If one of the replicas is down, the leader drops it from the current group and continues to write to the remaining replicas. A failed replica is allowed to rejoin the group if it comes back and catches up with the leader. With f replicas, primary-backup replication can tolerate f-1 failures.
In the quorum-based approach, the leader waits until a write completes on a majority of the replicas. The size of the replica group doesn't change even when some replicas are down. If there are 2f+1 replicas, quorum-based replication can tolerate f replica failures. If the leader fails, it needs at least f+1 replicas to elect a new leader.

There are tradeoffs between the 2 approaches:

1. The quorum-based approach has better write latency than the primary-backup one. A delay (e.g., long GC) in any replica increases the write latency in the latter, but not the former.
2. Given the same number of replicas, the primary-backup approach tolerates more concurrent failures.
3. A replication factor of 2 works well with the primary-backup approach. In quorum-based replication, both replicas have to be up for the system to be available.

We choose the primary-backup replication in Kafka since it tolerates more failures and works well with 2 replicas. A hiccup can happen when a replica is down or becomes slow. However, those are relatively rare events and the hiccup time can be reduced by tuning various timeout parameters.

Synchronous replication

Our synchronous replication follows the typical primary-backup approach. Each partition has n replicas and can tolerate n-1 replica failures. One of the replicas is elected as the leader and the rest of the replicas are followers. The leader maintains a set of in-sync replicas (ISR): the set of replicas that have fully caught up with the leader. For each partition, we store in Zookeeper the current leader and the current ISR.

Each replica stores messages in a local log and maintains a few important offset positions in the log (depicted in Figure 1). The log end offset (LEO) represents the tail of the log. The high watermark (HW) is the offset of the last committed message. Each log is periodically synced to disks. Data before the flushed offset is guaranteed to be persisted on disks. As we will see, the flush offset can be before or after HW.

Writes

To publish a message to a partition, the client first finds the leader of the partition from Zookeeper and sends the message to the leader. The leader writes the message to its local log. Each follower constantly pulls new messages from the leader using a single socket channel. That way, the follower receives all messages in the same order as written in the leader. The follower writes each received message to its own log and sends an acknowledgment back to the leader. Once the leader receives the acknowledgment from all replicas in ISR, the message is committed. The leader advances the HW and sends an acknowledgment to the client. For better performance, each follower sends an acknowledgment after the message is written to memory. So, for each committed message, we guarantee that the message is stored in multiple replicas in memory. However, there is no guarantee that any replica has persisted the commit message to disks though. Given that correlated failures are relatively rare, this approach gives us a good balance between response time and durability. In the future, we may consider adding options that provide even stronger guarantees. The leader also periodically broadcasts the HW to all followers. The broadcasting can be piggybacked on the return value of the fetch requests from the followers. From time to time, each replica checkpoints its HW to its disk.

Reads

For simplicity, reads are always served from the leader. Only messages up to the HW are exposed to the reader.

Failure scenarios

Follower failure

After a configured timeout period, the leader will drop the failed follower from its ISR and writes will continue on the remaining replicas in ISR. If the failed follower comes back, it first truncates its log to the last checkpointed HW. It then starts to catch up all messages after its HW from the leader. When the follower fully catches up, the leader will add it back to the current ISR.

Leader failure

There are 3 cases of leader failure which should be considered -

1. The leader crashes before writing the messages to its local log. In this case, the client will timeout and resend the message to the new leader.
2. The leader crashes after writing the messages to its local log, but before sending the response back to the client
    a. Atomicity has to be guaranteed: Either all the replicas wrote the messages or none of them
    b. The client will retry sending the message. In this scenario, the system should ideally ensure that the messages are not written twice. Maybe, one of the replicas had written the message to its local log, committed it, and it gets elected as the new leader.
3. The leader crashes after sending the response. In this case, a new leader will be elected and start receiving requests.

When this happens, we need to perform the following steps to elect a new leader.

1. Each surviving replica in ISR registers itself in Zookeeper.
2. The replica that registers first becomes the new leader. The new leader chooses its LEO as the new HW.
3. Each replica registers a listener in Zookeeper so that it will be informed of any leader change. Everytime a replica is notified about a new leader:
    a. If the replica is not the new leader (it must be a follower), it truncates its log to its HW and then starts to catch up from the new leader.
4. The leader waits until all surviving replicas in ISR have caught up or a configured time has passed. The leader writes the current ISR to Zookeeper and opens itself up for both reads and writes.

(Note, during the initial startup when ISR is empty, any replica can become the leader.)

Asynchronous replication

To support asynchronous replication, the leader can acknowledge the client as soon it finishes writing the message to its local log. The only caveat is that during the catchup phase, the follower may have to truncate the data before its HW. Since the replication is asynchronous, there is no guarantee that a commit message can survive any broker failure.

Open Issues

1. How can atomicity be guaranteed in the 2nd type of leader failure ?
2. How can we avoid the problem of multiple leaders for the same partition at the same time ?
3. If the brokers are in multiple racks, how to guarantee that at least one replica goes to a different rack?

Kafka Replication Detailed Design

The following are different proposals of the detailed implementation. In 0.8, the implementation is based on the v3 proposal.

V2 proposal

V3 proposal