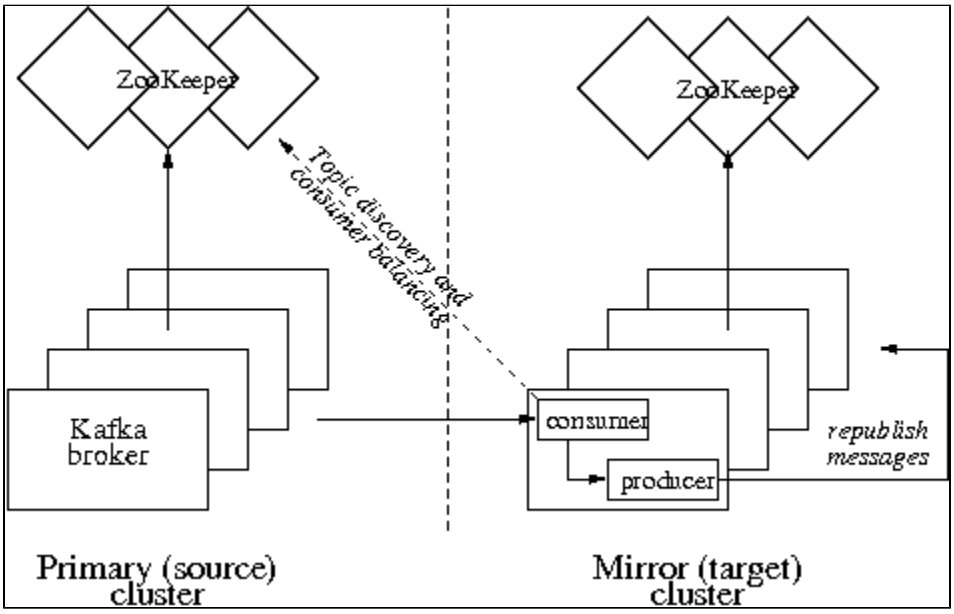
Kafka mirroring

Note: The embedded consumer approach is deprecated by the MirrorMaker approach in 0.7.1. See https://cwiki.apache.org/confluence/display/KAFKA/Kafka+mirroring+(MirrorMaker) for details.

Kafka mirroring

Kafka's mirroring feature makes it possible to maintain a replica of an existing Kafka cluster.



The Kafka mirror cluster uses an embedded Kafka consumer to consume messages from a source cluster, and re-publishes those messages to the local cluster using an embedded Kafka producer.

How to set up a mirror

Setting up a mirror cluster is easy - simply provide the embedded consumer's configuration and the embedded producer's configuration, in addition to the server configuration configuration when you start up the Kafka brokers in the mirror cluster. You need to point the consumer to the source cluster's ZooKeeper, and the producer to the mirror cluster's ZooKeeper. By default, the mirror cluster will mirror all topics present on the source cluster. You can instead configure a whitelist or blacklist as described in the next section.

The following demo uses the sample configurations provided for the Kafka quick-start, and the following configuration files: mirror-server.properties, mirror-consumer.properties, mirror-producer.properties, mirror-zookeeper.properties.

- Start the primary cluster:
  ```
  bin/zookeeper-server-start.sh config/zookeeper.properties
  bin/kafka-server-start.sh config/server.properties
  ```
- Start the mirror cluster:
  ```
  bin/zookeeper-server-start.sh config/mirror-zookeeper.properties
  JMX_PORT=8888 bin/kafka-server-start.sh config/mirror-server.properties config/mirror-consumer.properties
  config/mirror-producer.properties
  ```
- If you now send messages to topics on the source cluster, the mirror cluster will eventually mirror those topics.

Important configuration parameters for a mirror

Embedded producer timeout

In order to sustain a higher throughput, you would typically use an asynchronous embedded producer and it should be configured to be in blocking mode (i.e., *queue.enqueueTimeout.ms=-1*). This recommendation is to ensure that messages will not be lost. Otherwise, the default enqueue timeout of the asynchronous producer is zero which means if the producer's internal queue is full, then messages will be dropped due to QueueFullExceptions. A blocking producer however, will wait if the queue is full, and effectively throttle back the embedded consumer's consumption rate. You can enable trace logging in the producer to observe the remaining queue size over time. If the producer's queue is consistently full, it indicates that the mirror cluster is bottle-necked on re-publishing messages to the local (mirror) cluster and/or flushing messages to disk.

Embedded consumer whitelist or blacklist

If you do not wish to have full mirroring, you may use either the whitelist(*mirror.topics.whitelist*) configuration option to specify which topics to include, or the blacklist (*mirror.topics.blacklist*) configuration option to specify which topics to exclude. (It is invalid to specify both a whitelist and a blacklist.) These options accept a comma-separated list of topics.

Embedded consumer and source cluster socket buffer sizes

Mirroring is often used in cross-DC scenarios, and there are a few configuration options that you may want to tune to help deal with inter-DC communication latencies and performance bottlenecks on your specific hardware. In general, you should set a high value for the socket buffer size on the mirror cluster's consumer configuration (*socket.buffersize*) and the source cluster's broker configuration (*socket.send.buffer*). Also, the embedded consumer's fetch size (*fetch.size*) should be higher than the consumer's socket buffer size. Note that the socket buffer size configurations are a hint to the underlying platform's networking code. If you enable trace logging, you can check the actual receive buffer size and determine whether the setting in the OS networking layer also needs to be adjusted.

How to check whether a mirror is keeping up

The consumer offset checker tool is useful to gauge how well your mirror is keeping up with the source cluster. For example, the following was executed during the above demo:

```
bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --group KafkaMirror --zkconnect localhost:2181 --topic
test-topic
KafkaMirror,topic1,0-0 (Group,Topic,BrokerId-PartitionId)
            Owner = KafkaMirror_jkoshy-ld-1320972386342-beb4bfc9-0
  Consumer offset = 561154288
                  = 561,154,288 (0.52G)
        Log size = 2231392259
                  = 2,231,392,259 (2.08G)
     Consumer lag = 1670237971
                  = 1,670,237,971 (1.56G)
BROKER INFO
0 -> 127.0.0.1:9092
```

Note that the --zkconnect argument should point to the source cluster's ZooKeeper. Also, if the topic is not specified, then the tool prints information for all topics under the given consumer group.

Current limitations

There are few limitations with the current implementation of mirroring:

- The embedded producer uses the default (random) partitioner. So if you use a custom partitioner in your source cluster, the mirror cluster is not capable of using the same partitioning scheme.
- The embedded consumer instantiates the same number of message streams for each topic according to the *mirror.consumer.numthreads* configuration option. There is no per-topic override at the moment.
- The whitelist and blacklist configuration options do not accept regular expressions and only one of these options can be used in a given mirror setup.