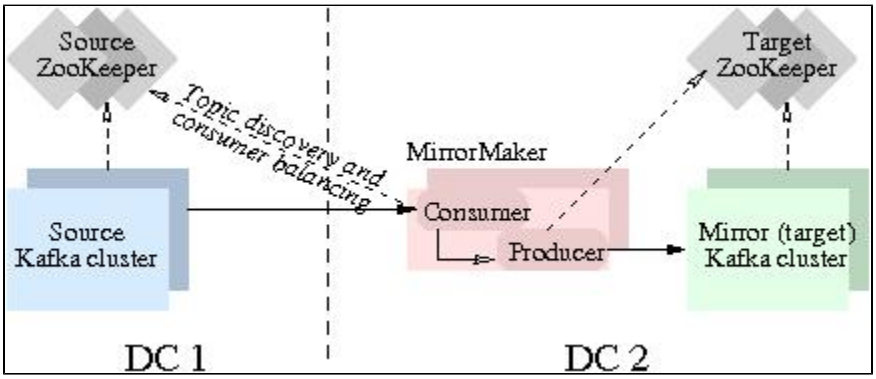
Kafka mirroring (MirrorMaker)

Kafka's mirroring feature makes it possible to maintain a replica of an existing Kafka cluster. The following diagram shows how to use the *MirrorMaker* tool to mirror a source Kafka cluster into a target (mirror) Kafka cluster. The tool uses a Kafka consumer to consume messages from the source cluster, and re-publishes those messages to the local (target) cluster using an embedded Kafka producer.

There is also documentation on the MirrorMaker at https://kafka.apache.org/documentation.html#basic_ops_mirror_maker.



How to set up a mirror

Setting up a mirror is easy - simply start up the mirror-maker processes after bringing up the target cluster. At minimum, the mirror maker takes one or more consumer configurations, a producer configuration and either a whitelist or a blacklist. You need to point the consumer to the source cluster's ZooKeeper, and the producer to the mirror cluster's ZooKeeper (or use the *broker.list* parameter).

```
bin/kafka-mirror-maker.sh --consumer.config sourceCluster1Consumer.config --consumer.config
sourceCluster2Consumer.config --num.streams 2 --producer.config targetClusterProducer.config --whitelist=".*"
```

Import configuration parameters for the mirror maker

Whitelist or blacklist

The mirror-maker accepts exactly one of whitelist or blacklist. These are standard Java regex patterns, although comma (',') is interpreted as the regex-choice symbol ('|') for convenience.

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-maker is bottle-necked on re-publishing messages to the local (mirror) cluster and/or flushing messages to disk.

Producer retries

If you use *broker.list* in the producer configuration and point it to a hardware load-balancer, then you can configure the number of retry attempts on producer failures. (The retry option only applies when you specify a load-balancer in the *broker.list* option since the broker is re-selected on retry only if you use a hardware load-balancer.)

Number of producers

You can use the *--num.producers* option to use a producer pool in the mirror maker to increase throughput. This helps because each producer's requests are effectively handled by a single thread on the receiving Kafka broker. i.e., even if you have multiple consumption streams (see next section), the throughput can be bottle-necked at handling stage of the mirror maker's producer requests.

Number of consumption streams

Use the *--num.streams* option to specify the number of mirror consumer threads to create. Note that if you start multiple mirror maker processes then you may want to look at the distribution of partitions on the source cluster. If the number of consumption streams is too high per mirror maker process, then some of the mirroring threads will be idle by virtue of the consumer rebalancing algorithm (if they do not end up owning any partitions for consumption).

Shallow iteration and producer compression (Kafka 0.7)

(N/A for Kafka 0.8, see JIRA issue KAFKA-732)

Our recommendation is to enable shallow iteration in the mirror maker's consumer. This means that the mirror maker's consumer will not attempt to decompress message-sets of compressed messages. It will simply re-publish these messages as is.

If you enable shallow iteration, you must disable compression in the mirror maker's producer, or message-sets can become double-compressed.

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-maker's consumer configuration (*socket.buffersize*) and the source cluster's broker configuration (socket.send.buffer). Also, the mirror-maker 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. Note that the *--zkconnect* argument should point to the source cluster's ZooKeeper (DC1 in this scenario). Also, if the topic is not specified, then the tool prints information for all topics under the given consumer group. For example:

```
bin/kafka-consumer-offset-checker.sh --group KafkaMirror --zkconnect dc1-zookeeper:2181 --topic test-topic
Group           Topic                          Pid Offset          logSize         Lag             Owner
KafkaMirror     test-topic                     0   5               5               0               none
KafkaMirror     test-topic                     1   3               4               1               none
KafkaMirror     test-topic                     2   6               9               3               none
```