Hortonworks Data Platform

Kafka Guide

(June 28, 2016)

docs.cloudera.com

Hortonworks Data Platform: Kafka Guide

The Hortonworks Data Platform, powered by Apache Hadoop, is a massively scalable and 100% open source platform for storing, processing and analyzing large volumes of data. It is designed to deal with data from many sources and formats in a very quick, easy and cost-effective manner. The Hortonworks Data Platform consists of the essential set of Apache Hadoop projects including MapReduce, Hadoop Distributed File System (HDFS), HCatalog, Pig, Hive, HBase, ZooKeeper and Ambari. Hortonworks is the major contributor of code and patches to many of these projects. These projects have been integrated and tested as part of the Hortonworks Data Platform release process and installation and configuration tools have also been included.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. The Hortonworks Data Platform is Apache-licensed and completely open source. We sell only expert technical support, training and partner-enablement services. All of our technology is, and will remain, free and open source.

Please visit the Hortonworks Data Platform page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the Support or Training page. Feel free to contact us directly to discuss your specific needs.

Table of Contents

List of Tables

1. Introduction to Kafka

Apache Kafka is a fast, scalable, durable, fault-tolerant publish-subscribe messaging system. Common use cases include:

- Messaging
- Website activity tracking
- Metrics collection and monitoring
- Log aggregation
- Stream processing
- Event sourcing
- Commit logs

Kafka works with Apache Storm and Apache Spark for real-time analysis and rendering of streaming data. The combination of messaging and processing technologies enables stream processing at linear scale.

For example, Apache Storm ships with support for Kafka as a data source using Storm's core API or the higher-level, micro-batching Trident API. Storm's Kafka integration also includes support for writing data to Kafka, which enables complex data flows between components in a Hadoop-based architecture. For more information about Apache Storm, see the Storm User Guide.

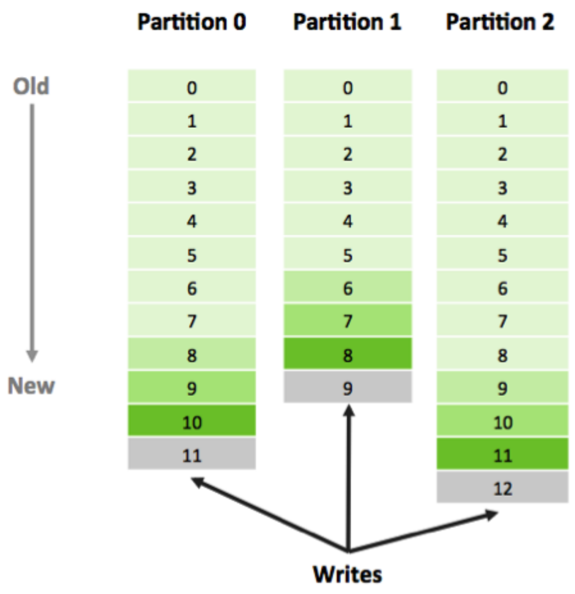
Kafka operates on streams of messages. Four main components move messages in and out of Kafka:

Table 1.1. Kafka Components

Kafka Component	Description
Topic	A user-defined category (or feed name) to which messages are published.
Producer	A process that publishes messages to one or more topics.
Consumer	A process that subscribes to one or more topics and processes the feeds of messages from those topics.
Broker	A Kafka server that manages the persistence and replication of message data (i.e., the commit log).

Topics

Topics consist of one or more partitions. Kafka appends new messages to a partition in an ordered, immutable sequence. Each message in a topic is assigned a unique, sequential ID called an **offset**.



Producers

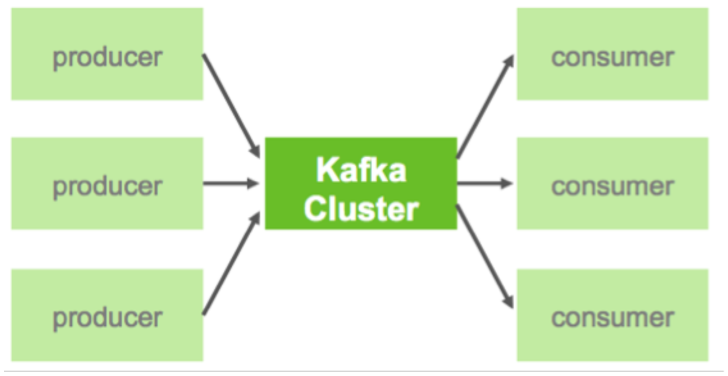
Kafka Producers publish messages to topics. The producer determines which message to assign to which partition within the topic. Assignment can be done in a round-robin fashion to balance load, or it can be based on a semantic partition function.

Consumers

Kafka Consumers keep track of which messages have already been consumed, or processed, by keeping track of an offset, a sequential id number that uniquely identifies a message within a partition. Because Kafka retains all messages on disk for a configurable amount of time, Consumers can rewind or skip to any point in a partition simply by supplying an offset value.

Brokers and Clusters

A Kafka Cluster consists of one or more Brokers (server processes). Producers send messages to the Kafka Cluster, which in turn serves them to Consumers.

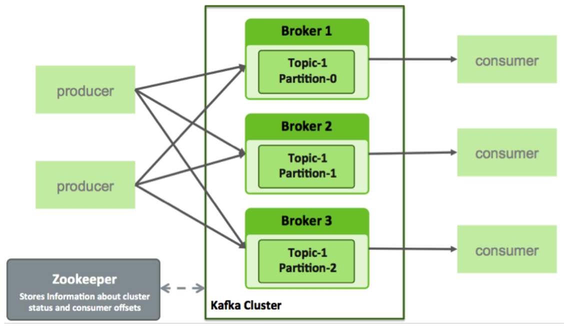


Performance

Partition support within topics provides parallelism within a topic. In addition, because writes to a partition are sequential, the number of hard disk seeks is minimized. This reduces latency and increases performance.

Kafka Brokers scale and perform well in part because Brokers are not responsible for keeping track of which messages have been consumed. The message Consumer is responsible for this. In traditional messaging systems such as JMS, the Broker bears this responsibility, which severely limits the system's ability to scale as the number of Consumers increase. Kafka's design eliminates the potential for back-pressure when consumers process messages at different rates.

For Kafka Consumers, keeping track of which messages have been consumed is simply a matter of keeping track of the offset – the sequential id that uniquely identifies a message within a partition. Because Kafka retains all messages on disk (for a configurable amount of time), Consumers can rewind or skip to any point in a partition simply by supplying an offset value.



Example

For an example that simulates the use of streaming geo-location information (using a previous version of Kafka), see Simulating and Transporting the Real-Time Event Stream with Apache Kafka.

2. Installing and Configuring Kafka

Prerequisite: ZooKeeper must be installed and running before using Kafka.

To install Kafka using Ambari, see Adding a Service to your Hadoop cluster in the Ambari User's Guide.

To configure Kafka for Kerberos security on an Ambari-managed cluster, see Configuring Kafka for Kerberos Over Ambari.

To install Kafka manually, see Installing and Configuring Kafka in the Non-Ambari Cluster Installation Guide.

To configure Ranger-based authorization for Kafka, see the Kafka section of the Ranger Ambari Installation Guide.

Note

HDP 2.4 supports JDK 1.7 and JDK 1.8 for Kafka.

2.1. Supported File Systems

The following underlying file systems are supported for use with Kafka:

- EXT4: supported and recommended
- EXT3: supported

Caution

Encrypted file systems such as SafenetFS are not supported for Kafka. Index file corruption can occur.

2.2. Saving Kafka Audits to HDFS

Audits to HDFS are activated automatically when Ranger is enabled for Kafka. You can also manually update the audit settings. For more information, see Manually Updating Ambari HDFS Audit Settings in the *HDP Security Guide*.

For secure clusters, perform the additional steps described in the note at the bottom of the Security Guide topic.

3. Developing Kafka Producers and Consumers

The examples in this chapter contain code for a basic Kafka producer and consumer, and similar examples for an SSL-enabled cluster.

For examples of Kafka producers and consumers that run on a Kerberos-enabled cluster, see Producing Events/Messages to Kafka on a Secured Cluster and Consuming Events/ Messages from Kafka on a Secured Cluster, in *Configuring Kafka for Kerberos over Ambari*.

Basic Producer Example

```
package com.hortonworks.example.kafka.producer;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

import java.util.Properties;
import java.util.Random;

public class BasicProducerExample {

   public static void main(String[] args){

       Properties props = new Properties();
       props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka.example.
com:6667");
       props.put(ProducerConfig.ACKS_CONFIG, "all");
       props.put(ProducerConfig.RETRIES_CONFIG, 0);
       props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringSerializer");
       props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringSerializer");

       Producer<String, String> producer = new KafkaProducer<String,
 String>(props);
       TestCallback callback = new TestCallback();
       Random rnd = new Random();
       for (long i = 0; i < 100 ; i++) {
           ProducerRecord<String, String> data = new ProducerRecord<String,
 String>(
                   "test-topic", "key-" + i, "message-"+i );
           producer.send(data, callback);
       }

       producer.close();
   }


   private static class TestCallback implements Callback {
       @Override
       public void onCompletion(RecordMetadata recordMetadata, Exception e) {
           if (e != null) {
               System.out.println("Error while producing message to topic :" +
 recordMetadata);
               e.printStackTrace();
           } else {
               String message = String.format("sent message to topic:%s
 partition:%s  offset:%s", recordMetadata.topic(), recordMetadata.partition(),
 recordMetadata.offset());
               System.out.println(message);
           }
       }
   }

}
```

To run the producer example, use the following command:

```
$ java com.hortonworks.example.kafka.producer.BasicProducerExample
```

Producer Example for an SSL-Enabled Cluster

The following example adds three important configuration settings for SSL encryption and three for SSL authentication. The two sets of configuration settings are prefaced by comments.

```
package com.hortonworks.example.kafka.producer;

import org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.config.SslConfigs;

import java.util.Properties;
import java.util.Random;

public class BasicProducerExample {

   public static void main(String[] args){

       Properties props = new Properties();
       props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka.example.
com:6667");

       //configure the following three settings for SSL Encryption
       props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
       props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG, "/var/private/ssl/
kafka.client.truststore.jks");
       props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,  "test1234");

       // configure the following three settings for SSL Authentication
       props.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG, "/var/private/ssl/
kafka.client.keystore.jks");
       props.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG, "test1234");
       props.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG, "test1234");

       props.put(ProducerConfig.ACKS_CONFIG, "all");
       props.put(ProducerConfig.RETRIES_CONFIG, 0);
       props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringSerializer");
       props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringSerializer");

       Producer<String, String> producer = new KafkaProducer<String,
 String>(props);
       TestCallback callback = new TestCallback();
       Random rnd = new Random();
       for (long i = 0; i < 100 ; i++) {
           ProducerRecord<String, String> data = new ProducerRecord<String,
 String>(
                   "test-topic", "key-" + i, "message-"+i );
           producer.send(data, callback);
       }

       producer.close();
   }


   private static class TestCallback implements Callback {
       @Override
       public void onCompletion(RecordMetadata recordMetadata, Exception e) {
           if (e != null) {
               System.out.println("Error while producing message to topic :" +
 recordMetadata);
               e.printStackTrace();
           } else {
               String message = String.format("sent message to topic:%s
 partition:%s  offset:%s", recordMetadata.topic(), recordMetadata.partition(),
 recordMetadata.offset());
               System.out.println(message);
           }
       }
   }

}
```

To run the producer example, use the following command:

```
$ java com.hortonworks.example.kafka.producer.BasicProducerExample
```

Basic Consumer Example

```
package com.hortonworks.example.kafka.consumer;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;

import java.util.Collection;
import java.util.Collections;
import java.util.Properties;

public class BasicConsumerExample {
   public static void main(String[] args) {

       Properties consumerConfig = new Properties();
       consumerConfig.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka.
example.com:6667");
       consumerConfig.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
       consumerConfig.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
 "earliest");
       consumerConfig.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
 "org.apache.kafka.common.serialization.StringDeserializer");
       consumerConfig.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "org.
apache.kafka.common.serialization.StringDeserializer");
       KafkaConsumer<byte[], byte[]> consumer = new
 KafkaConsumer<>(consumerConfig);
       TestConsumerRebalanceListener rebalanceListener = new
 TestConsumerRebalanceListener();
       consumer.subscribe(Collections.singletonList("test-topic"),
 rebalanceListener);

       while (true) {
           ConsumerRecords<byte[], byte[]> records = consumer.poll(1000);
           for (ConsumerRecord<byte[], byte[]> record : records) {
               System.out.printf("Received Message topic =%s, partition =%s,
 offset = %d, key = %s, value = %s\n", record.topic(), record.partition(),
 record.offset(), record.key(), record.value());
           }

           consumer.commitSync();
       }

   }

   private static class  TestConsumerRebalanceListener implements
 ConsumerRebalanceListener {
       @Override
       public void onPartitionsRevoked(Collection<TopicPartition> partitions)
 {
           System.out.println("Called onPartitionsRevoked with partitions:" +
 partitions);
       }

       @Override
       public void onPartitionsAssigned(Collection<TopicPartition> partitions)
 {
           System.out.println("Called onPartitionsAssigned with partitions:" +
 partitions);
       }
   }

}
```

To run the consumer example, use the following command:

```
# java com.hortonworks.example.kafka.consumer.BasicConsumerExample
```

Consumer Example for an SSL-Enabled Cluster

The following example adds three important configuration settings for SSL encryption and three for SSL authentication. The two sets of configuration settings are prefaced by comments.

```
package com.hortonworks.example.kafka.consumer;

import org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.config.SslConfigs;

import java.util.Collection;
import java.util.Collections;
import java.util.Properties;

public class BasicConsumerExample {

   public static void main(String[] args) {

       Properties props = new Properties();
       props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka.example.
com:6667");

       //configure the following three settings for SSL Encryption
       props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
       props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG, "/var/private/ssl/
kafka.client.truststore.jks");
       props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,  "test1234");

       //configure the following three settings for SSL Authentication
       props.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG, "/var/private/ssl/
kafka.client.keystore.jks");
       props.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG, "test1234");
       props.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG, "test1234");

       props.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
       props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
       props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringDeserializer");
       props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringDeserializer");
       KafkaConsumer<byte[], byte[]> consumer = new KafkaConsumer<>(props);
       TestConsumerRebalanceListener rebalanceListener = new
 TestConsumerRebalanceListener();
       consumer.subscribe(Collections.singletonList("test-topic"),
 rebalanceListener);

       while (true) {
           ConsumerRecords<byte[], byte[]> records = consumer.poll(1000);
           for (ConsumerRecord<byte[], byte[]> record : records) {
               System.out.printf("Received Message topic =%s, partition =%s,
 offset = %d, key = %s, value = %s\n", record.topic(), record.partition(),
 record.offset(), record.key(), record.value());
           }

           consumer.commitSync();
       }

   }
```

```
  private static class  TestConsumerRebalanceListener implements
ConsumerRebalanceListener {
      @Override
      public void onPartitionsRevoked(Collection<TopicPartition> partitions)
{
          System.out.println("Called onPartitionsRevoked with partitions:" +
partitions);
      }

      @Override
      public void onPartitionsAssigned(Collection<TopicPartition> partitions)
{
          System.out.println("Called onPartitionsAssigned with partitions:" +
partitions);
      }
  }

}
```

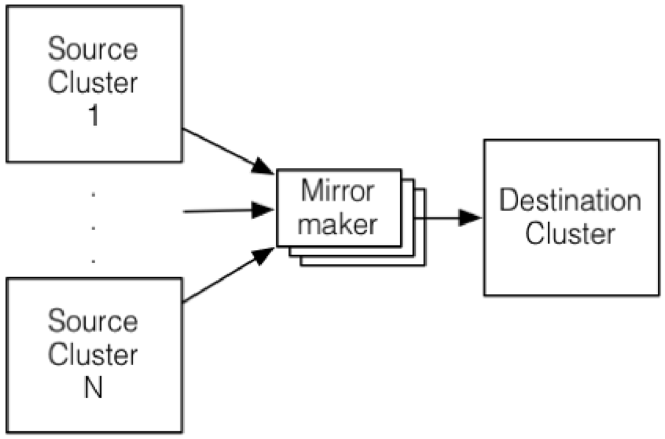
To run the consumer example, use the following command:

```
$ java com.hortonworks.example.kafka.producer.BasicProducerExample
```

4. Mirroring Data Between Clusters

The process of replicating data between Kafka clusters is called "mirroring", to differentiate cross-cluster replication from replication among nodes within a single cluster. A common use for mirroring is to maintain a separate copy of a Kafka cluster in another data center.

Kafka's MirrorMaker tool reads data from topics in one or more source Kafka clusters, and writes corresponding topics to a destination Kafka cluster (using the same topic names):



To mirror more than one source cluster, start at least one MirrorMaker instance for each source cluster.

You can also use multiple MirrorMaker processes to mirror topics within the same consumer group. This can increase throughput and enhance fault-tolerance: if one process dies, the others will take over the additional load.

The source and destination clusters are completely independent, so they can have different numbers of partitions and different offsets. The destination (mirror) cluster is not intended to be a mechanism for fault-tolerance, because the consumer position will be different. (The MirrorMaker process will, however, retain and use the message key for partitioning, preserving order on a per-key basis.) For fault tolerance we recommend using standard within-cluster replication.

4.1. Running MirrorMaker

Prerequisite: The source and destination clusters must be deployed and running.

To set up a mirror, run `kafka.tools.MirrorMaker`. The following table lists configuration options.

At a minimum, MirrorMaker requires one or more consumer configuration files, a producer configuration file, and either a whitelist or a blacklist of topics. In the consumer and producer configuration files, point the consumer to the ZooKeeper processon the source cluster, and point the producer to the ZooKeeper process on the destination (mirror) cluster, respectively.

Table 4.1. MirrorMaker Options

Parameter	Description	Examples
`--consumer.config`	Specifies a file that contains configuration settings for the source cluster. For more information about this file, see the "Consumer Configuration File" subsection.	`--consumer.config hdp1-consumer.properties`
`--producer.config`	Specifies the file that contains configuration settings for the target cluster. For more information about this file, see the "Producer Configuration File" subsection.	`--producer.config hdp1-producer.properties`
`--whitelist` `--blacklist`	(Optional) For a partial mirror, you can specify exactly one comma-separated list of topics to include (–whitelist) or exclude (–blacklist). In general, these options accept Java regex patterns. For caveats, see the note after this table.	`--whitelist my-topic`
`--num.streams`	Specifies the number of consumer stream threads to create.	`--num.streams 4`
`--num.producers`	Specifies the number of producer instances. Setting this to a value greater than one establishes a producer pool that can increase throughput.	`--num.producers 2`
`--queue.size`	Queue size: number of messages that are buffered, in terms of number of messages between the consumer and producer. Default = 10000.	`--queue.size 2000`
`--help`	List MirrorMaker command-line options.	

Note

- A comma (',') is interpreted as the regex-choice symbol ('|') for convenience.
- If you specify `--white-list=".*"`, MirrorMaker tries to fetch data from the system-level topic `__consumer-offsets` and produce that data to the target cluster. This can result in the following error:

  ```
  Producer cannot send requests to __consumer-offsets
  ```

  Workaround: Specify topic names, or to replicate all topics, specify `--blacklist="__consumer-offsets"`.

The following example replicates `topic1` and `topic2` from `sourceClusterConsumer` to `targetClusterProducer`:

```
/usr/hdp/current/kafka-broker/bin/kafka-run-class.sh kafka.tools.MirrorMaker
 --consumer.config sourceClusterConsumer.properties  --producer.config
 targetClusterProducer.properties --whitelist="topic1, topic"
```

Consumer Configuration File

The consumer configuration file must specify the ZooKeeper process in the source cluster.

Here is a sample consumer configuration file:

```
zk.connect=hdp1:2181/kafka
zk.connectiontimeout.ms=1000000
consumer.timeout.ms=-1
groupid=dp-MirrorMaker-test-datap1
shallow.iterator.enable=true
mirror.topics.whitelist=app_log
```

Producer Configuration File

The producer configuration should point to the target cluster's ZooKeeper process (or use the broker.list parameter to specify a list of brokers on the destination cluster).

Here is a sample producer configuration file:

```
zk.connect=hdp1:2181/kafka-test
producer.type=async
compression.codec=0
serializer.class=kafka.serializer.DefaultEncoder
max.message.size=10000000
queue.time=1000
queue.enqueueTimeout.ms=-1
```

4.2. Checking Mirroring Progress

You can use Kafka's Consumer Offset Checker command-line tool to assess how well your mirror is keeping up with the source cluster. The Consumer Offset Checker checks the number of messages read and written, and reports the lag for each consumer in a specified consumer group.

The following command runs the Consumer Offset Checker for group `KafkaMirror`, topic `test-topic`. The `--zkconnect` argument points to the ZooKeeper host and port on the source cluster.

```
/usr/hdp/current/kafka/bin/kafka-run-class.sh kafka.tools.
ConsumerOffsetChecker --group KafkaMirror --zkconnect source-cluster-
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
```

Table 4.2. Consumer Offset Checker Options

`--group`	(Required) Specifies the consumer group.
`--zkconnect`	Specifies the ZooKeeper connect string. The default is `localhost:2181`.
`--broker-info`	Lists broker information
`--help`	Lists offset checker options.
`--topic`	Specifies a comma-separated list of consumer topics. If you do not specify a topic, the offset checker will display information for all topics under the given consumer group.

4.3. Avoiding Data Loss

If for some reason the producer cannot deliver messages that have been consumed and committed by the consumer, it is possible for a MirrorMaker process to lose data.

To prevent data loss, use the following settings. (Note: these are the default settings.)

- For consumers:
  - `auto.commit.enabled=false`
- For producers:
  - `max.in.flight.requests.per.connection=1`
  - `retries=Int.MaxValue`
  - `acks=-1`
  - `block.on.buffer.full=true`
- Specify the `--abortOnSendFail` option to MirrorMaker

The following actions will be taken by MirrorMaker:

- MirrorMaker will send only one request to a broker at any given point.
- If any exception is caught in the MirrorMaker thread, MirrorMaker will try to commit the acked offsets and then exit immediately.
- On a `RetriableException` in the producer, the producer will retry indefinitely. If the retry does not work, MirrorMaker will eventually halt when the producer buffer is full.
- On a non-retriable exception, if `--abort.on.send.fail` is specified, MirrorMaker will stop.

  If `--abort.on.send.fail` is not specified, the producer callback mechanism will record the message that was not sent, and MirrorMaker will continue running. In this case, the message will not be replicated in the target cluster.

4.4. Running MirrorMaker on Kerberos-Enabled Clusters

To run MirrorMaker on a Kerberos/SASL-enabled cluster, configure producer and consumer properties as follows:

1. Choose or add a new principal for MirrorMaker. Do not use `kafka` or any other service accounts. The following example uses principal `mirrormaker`.
2. Create client-side Kerberos keytabs for your MirrorMaker principal. For example:

   ```
   sudo kadmin.local -q "ktadd -k /tmp/mirrormaker.keytab mirrormaker/
   HOSTNAME@EXAMPLE.COM"
   ```

3. Add a new Jaas configuration file to the node where you plan to run MirrorMaker:

   ```
   -Djava.security.auth.login.config=/usr/hdp/current/kafka-broker/config/
   kafka_mirrormaker_jaas.conf
   ```

4. Add the following settings to the KafkaClient section of the new Jaas configuration file. Make sure the principal has permissions on both the source cluster and the target cluster.

   ```
   KafkaClient {
        com.sun.security.auth.module.Krb5LoginModule required
        useKeyTab=true
        keyTab="/tmp/mirrormaker.keytab"
        storeKey=true
        useTicketCache=false
        serviceName="kafka"
        principal="mirrormaker/HOSTNAME@EXAMPLE.COM";
       };
   ```

5. Run the following ACL command on the source and destination Kafka clusters:

   ```
   bin/kafka-acls.sh --topic test-topic --add --allow-principal
    user:mirrormaker --operation ALL --config /usr/hdp/current/kafka-broker/
   config/server.properties
   ```

6. In your MirrorMaker `consumer.config` and `producer.config` files, specify `security.protocol=SASL_PLAINTEXT`.

7. Start MirrorMaker. Specify the `new.consumer` option in addition to your other options. For example:

   ```
   /usr/hdp/current/kafka-broker/bin/kafka-run-class.sh kafka.tools.MirrorMaker
    --consumer.config consumer.properties --producer.config target-cluster-
   producer.properties --whitelist my-topic --new.consumer
   ```