Cloudera Runtime 7.2.0

Developing Apache Kafka Applications

Date published: 2019-12-18
Date modified: 2020-06-16

https://docs.cloudera.com/

Legal Notice

Contents

Kafka producers

Learn more about Kafka producers and their most important configuration properties.

Kafka producers are the publishers responsible for writing records to topics. Typically, this means writing a program using the KafkaProducer API. To instantiate a producer:

```
KafkaProducer<String, String> producer = new
KafkaProducer<>(producerConfig);
```

Most of the important producer settings, and mentioned below, are in the configuration passed by this constructor.

Serialization of Keys and Values

For each producer, there are two serialization properties that must be set, key.serializer (for the key) and value.serial izer (for the value). You can write custom code for serialization or use one of the ones already provided by Kafka. Some of the more commonly used ones are:

- ByteArraySerializer: Binary data
- StringSerializer: String representations

Managing Record Throughput

There are several settings to control how many records a producer accumulates before actually sending the data to the cluster. This tuning is highly dependent on the data source. Some possibilities include:

- batch.size: Combine this fixed number of records before sending data to the cluster.
- linger.ms: Always wait at least this amount of time before sending data to the cluster; then send however many records has accumulated in that time.
- max.request.size: Put an absolute limit on data size sent. This technique prevents network congestion caused by a single transfer request containing a large amount of data relative to the network speed.
- compression.type: Enable compression of data being sent.
- retries: Enable the client for retries based on transient network errors. Used for reliability.

Acknowledgments

The full write path for records from a producer is to the leader partition and then to all of the follower replicas. The producer can control which point in the path triggers an acknowledgment. Depending on the acks setting, the producer may wait for the write to propagate all the way through the system or only wait for the earliest success point.

Valid acks values are:

- 0: Do not wait for any acknowledgment from the partition (fastest throughput).
- 1: Wait only for the leader partition response.
- all: Wait for follower partitions responses to meet minimum (slowest throughput).

Partitioning

In Kafka, the partitioner determines how records map to partitions. Use the mapping to ensure the order of records within a partition and manage the balance of messages across partitions. The default partitioner uses the entire key to determine which partition a message corresponds to. Records with the same key are always mapped to the same partition (assuming the number of partitions does not change for a topic). Consider writing a custom partitioner if you have information about how your records are distributed that can produce more efficient load balancing across partitions. A custom partitioner lets you take advantage of the other data in the record to control partitioning.

If a partitioner is not provided to the KafkaProducer, Kafka uses a default partitioner.

The ProducerRecord class is the actual object processed by the KafkaProducer. It takes the following parameters:

- Kafka Record: The key and value to be stored.
- Intended Destination: The destination topic and the specific partition (optional).

Kafka consumers

Learn more about Kafka consumers.

Kafka consumers are the subscribers responsible for reading records from one or more topics and one or more partitions of a topic. Consumers subscribing to a topic can happen manually or automatically; typically, this means writing a program using the KafkaConsumer API.

To instantiate a consumer:

```
KafkaConsumer<String, String> kafkaConsumer = new
KafkaConsumer<>(consumerConfig);
```

The KafkaConsumer class has two generic type parameters. Just as producers can send data (the values) with keys, the consumer can read data by keys. In this example both the keys and values are strings. If you define different types, you need to define a deserializer to accommodate the alternate types. For deserializers you need to implement the org.apache.kafka.common.serialization.Deserializer interface.

The most important configuration parameters that we need to specify are:

- bootstrap.servers: A list of brokers to initially connect to. List 2 to 3 brokers; you don't needed to list the full cluster.
- group.id: Every consumer belongs to a group. That way they'll share the partitions of a topic.

  Need to make this better by describing how 'groups' relate to partitions.
- key.deserializer/value.deserializer: Specify how to convert the Java representation to a sequence of bytes to send data through the Kafka protocol.

In addition to the configuration properties presented above, there are a number of other important configurations that any user of Kafka must know about. These are:

- heartbeat.interval.ms: The interval of the heartbeats. For example, if the heartbeat interval is set to 3 seconds, the consumer sends a short heartbeat message to the broker every 3 seconds to indicate that it is alive.
- session.timeout.ms: The consumer tells this timeout to the coordinator. This is used to control the heartbeats and remove the dead consumers. If it's set to 10 seconds, the consumer can miss sending 2 heartbeats, assuming the previous heartbeat setting. If we increase the timeout, the consumer has more room for delays but the broker notices lagging consumers later.
- max.poll.interval.ms: It is a very important detail: the consumers must maintain polling and should never do long-running processing. If a consumer is taking too much time between two polls, it will be detached from the consumer group. We can tune this configuration according to our needs. Note that if a consumer is stuck in processing, it will be noticed later if the value is increased.
- request.timeout.ms: Generally every request has a timeout. This is an upper bound that the client waits for the server's response. If this timeout elapses, then retries might happen if the number of retries are not exhausted.

Subscribing to a topic

Learn more about subscribing to a topic.

Subscribing to a topic using the subscribe() method call:

```
kafkaConsumer.subscribe(Collections.singletonList(topic), rebalanceListe
ner);
```

Here we specify a list of topics that we want to consume from and a 'rebalance listener.' Rebalancing is an important part of the consumer's life. Whenever the cluster or the consumers' state changes, a rebalance will be issued. This will ensure that all the partitions are assigned to a consumer.

After subscribing to a topic, the consumer polls to see if there are new records:

```
while (true) {
  data = kafkaConsumer.poll();
  // do something with 'data'
}
```

The poll returns multiple records that can be processed by the client. After processing the records the client commits offsets synchronously, thus waiting until processing completes before continuing to poll.

The last important point is to save the progress. This can be done by the commitSync() and commitAsync()methods respectively.

commitSync()

```
kafkaConsumer.subscribe(Collections.singletonList(topic), rebala
nceListener);

        while (true) {
            ConsumerRecords<String, String> records = kafkaCon
sumer.poll(1000);
            for (ConsumerRecord<String, String> record : recor
ds) {
                System.out.printf("Received Message with topic =
%s, partition =%s, offset = %d, key = %s, value = %s\n",
                        record.topic(), record.partition(), recor
d.offset(), record.key(), record.value());
            }
            // commit and wait until the offset is committed
            kafkaConsumer.commitSync();
        }
```

commitAsync()

```
kafkaConsumer.subscribe(Collections.singletonList(topic), rebala
nceListener);

        while (true) {
            ConsumerRecords<String, String> records = kafkaCon
sumer.poll(1000);
            for (ConsumerRecord<String, String> record : recor
ds) {
                System.out.printf("Received Message with topic =
%s, partition =%s, offset = %d, key = %s, value = %s\n",
                        record.topic(), record.partition(), recor
d.offset(), record.key(), record.value());
            }

            // Commit the offset and proceed with execution. The
 callback will be invoked when the offset
            // commit's result comes back from the broker.
            kafkaConsumer.commitAsync((offsets, exception) -> {
                if (exception != null) {
                    // handle the error that happened during off
set commit
                } else {
                    // do something on successful offset commit
if needed
                }
```

```
            });
        }
```

Auto commit is not recommended; manual commit is appropriate in the majority of use cases.

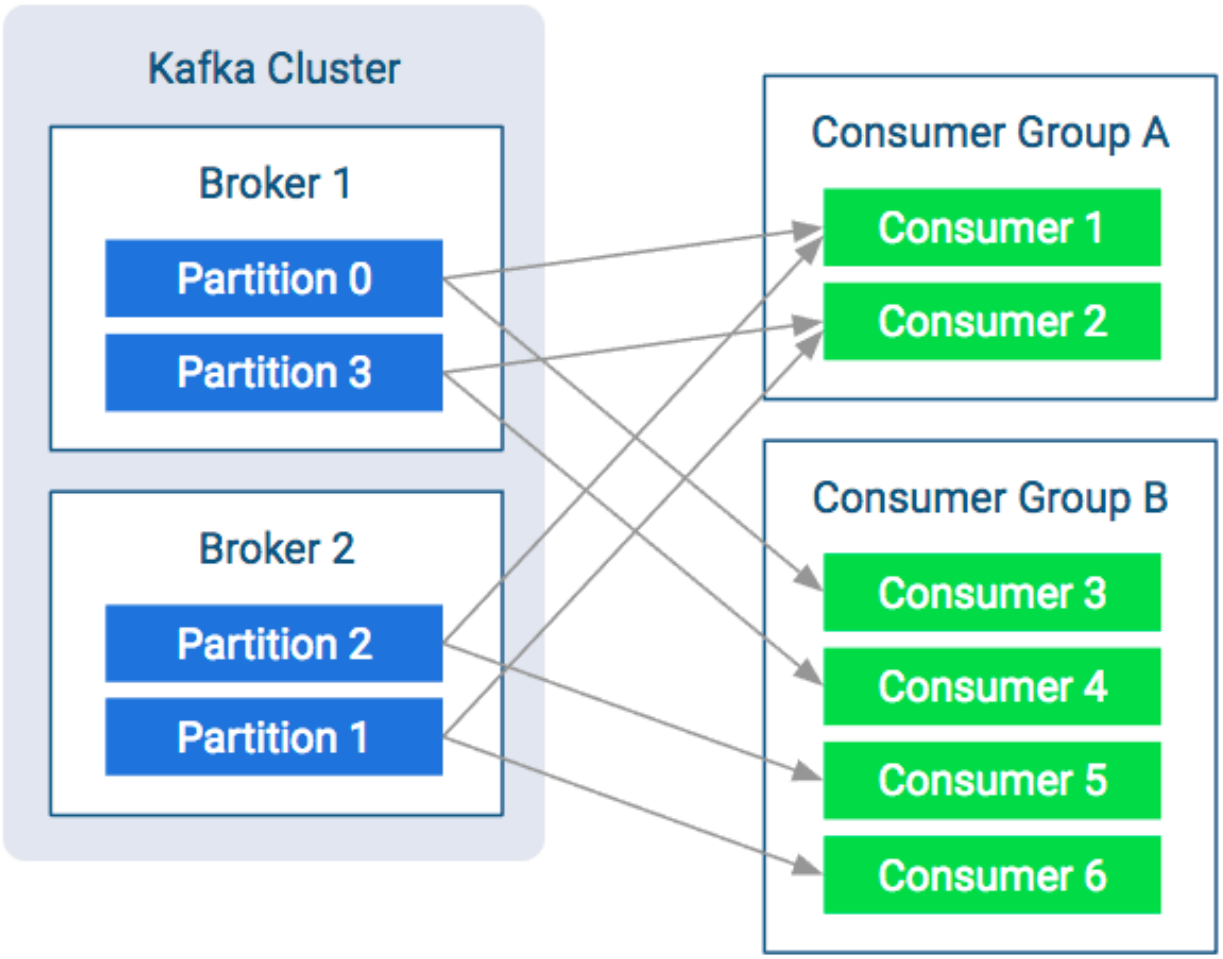
Groups and fetching

Learn more about how consumer groups fetch messages.

Kafka consumers are usually assigned to a group. This happens statically by setting the group.id configuration property in the consumer configuration. Consuming with groups will result in the consumers balancing the load in the group. That means each consumer will have their fair share of partitions. Also it can never be more consumers than partitions as that way there would be idling consumers.

As shown in the figure below, both consumer groups share the partitions and each partition multicasts messages to both consumer groups. The consumers pull messages from the broker instead of the broker periodically pushing what is available. This helps the consumer as it won't be overloaded and it can query the broker at its own speed. Furthermore, to avoid tight looping, it uses a so called "long-poll". The consumer sends a fetch request to poll for data and receives a reply only when enough data accumulates on the broker.

Figure 1: Consumer Groups and Fetching from Partitions



Protocol between consumer and broker

Get to know how the protocol works, what messages are going on the wire, and how that contributes to the overall behavior of the consumer.

When discussing the internals of the consumers, there are a couple of basic terms to know:

Heartbeat

When the consumer is alive and is part of the consumer group, it sends heartbeats. These are short periodic messages that tell the brokers that the consumer is alive and everything is fine.

Session

Often one missing heartbeat is not a big deal, but how do you know if a consumer is not sending heartbeats for long enough to indicate a problem? A session is such a time interval. If the consumer didn’t send any heartbeats for longer than the session, the broker can consider the consumer dead and remove it from the group.

Coordinator

The special broker which manages the group on the broker side is called the coordinator. The coordinator handles heartbeats and assigns the leader. Every group has a coordinator that organizes the startup of a consumer group and assist whenever a consumer leaves the group.

Leader

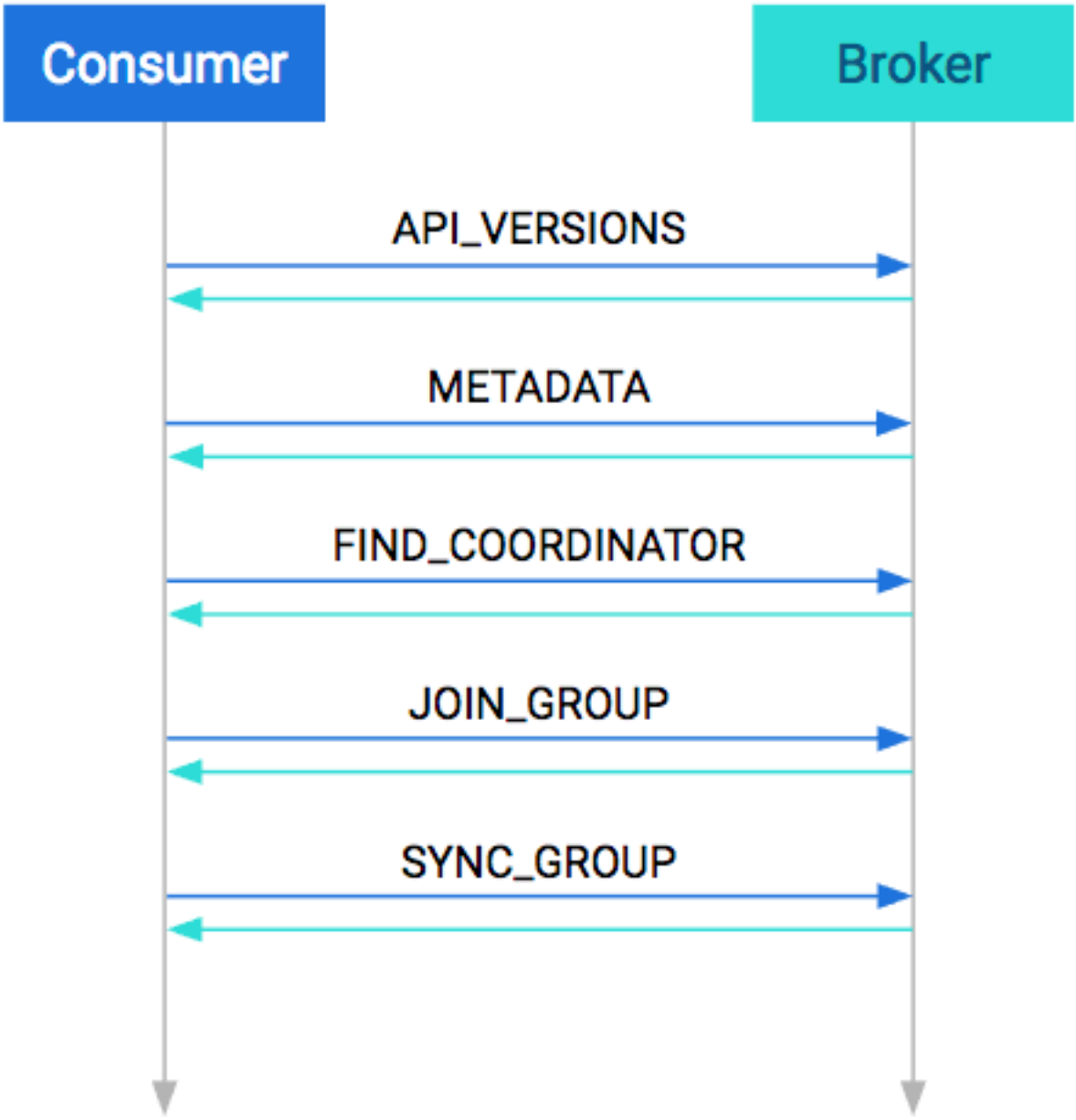
The leader consumer is elected by the coordinator. Its job is to assign partitions to every consumer in the group at startup or whenever a consumer leaves or joins the group. The leader holds the assignment strategy, it is decoupled from the broker. That means consumers can reconfigure the partition assignment strategy without restarting the brokers.

Startup Protocol

As mentioned before, the consumers are working usually in groups. So a major part of the startup process is spent with figuring out the consumer group.

At startup, the first step is to match protocol versions. It is possible that the broker and the consumer are of different versions (the broker is older and the consumer is newer, or vice versa). This matching is done by the API_VERS IONS request.

Figure 2: Startup Protocol



The next step is to collect cluster information, such as the addresses of all the brokers (prior to this point we used the bootstrap server as a reference), partition counts, and partition leaders. This is done in the METADATA request.

After acquiring the metadata, the consumer has the information needed to join the group. By this time on the broker side, a coordinator has been selected per consumer group. The consumers must find their coordinator with the FIND _COORDINATOR request.

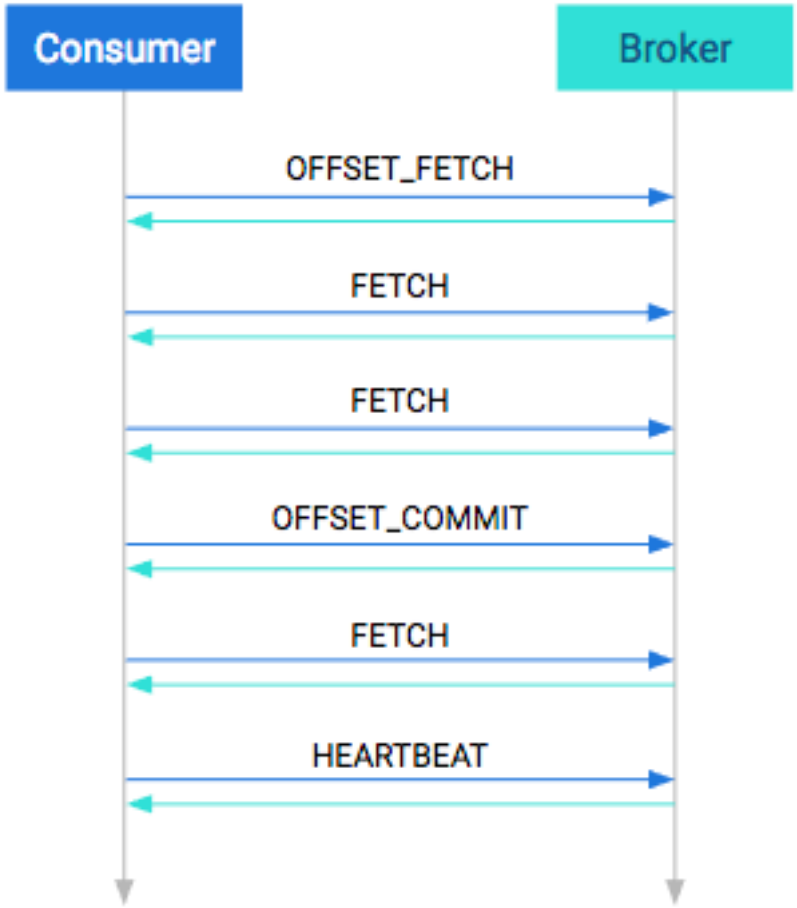
After finding the coordinator, the consumer(s) are ready to join the group. Every consumer in the group sends their own member-specific metadata to the coordinator in the JOIN_GROUP request. The coordinator waits until all the consumers have sent their request, then assigns a leader for the group. At the response plus the collected metadata are sent to the leader, so it knows about its group.

The remaining step is to assign partitions to consumers and propagate this state. Similar to the previous request, all consumers send a SYNC_GROUP request to the coordinator; the leader provides the assignments in this request. After it receives the sync request from each group member, the coordinator propagates this member state in the response. By the end of this step, the consumers are ready and can start consuming.

Consumption Protocol

When consuming, the first step is to query where should the consumer start. This is done in the OFFSET_FETCH request. This is not mandatory: the consumer can also provide the offset manually. After this, the consumer is free to pull data from the broker. Data consumption happens in the FETCH requests. These are the long-pull requests. They are answered only when the broker has enough data; the request can be outstanding for a longer period of time.

Figure 3: Consumption Protocol

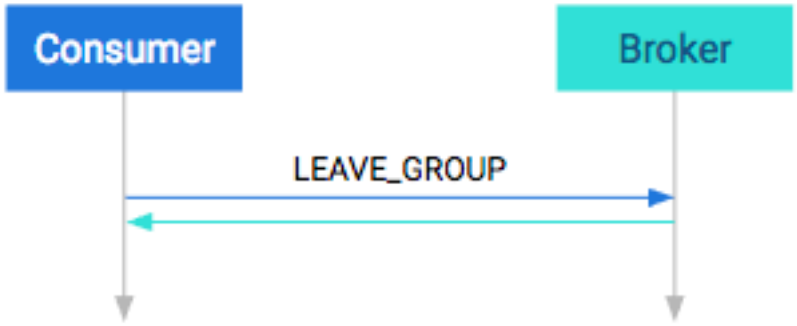


From time to time, the application has to either manually or automatically save the offsets in an OFFSET_COMMIT request and send heartbeats too in the HEARTBEAT requests. The first ensures that the position is saved while the latter ensures that the coordinator knows that the consumer is alive.

Shutdown Protocol

The last step when the consumption is done is to shut down the consumer gracefully. This is done in one single step, called the LEAVE_GROUP protocol.

Figure 4: Shutdown Protocol



Rebalancing partitions

Learn what rebalancing is, when it can occur, and how its propagated to the client.

There are multiple points in the protocol between consumers and brokers where failures can occur. There are points in the normal operation of the system where you need to change the consumer group assignments. For example, to consume a new partition or to respond to a consumer going offline. The process or responding to cluster information changing is called rebalance. It can occur in the following cases:

- A consumer leaves. It can be a software failure where the session times out or a connection stalls for too long, but it can also be a graceful shutdown.
- A consumer joins. It can be a new consumer but an old one that just recovered from a software failure (automatically or manually).
- Partition is adjusted. A partition can simply go offline because of a broker failure or a partition coming back online. Alternatively an administrator can add or remove partitions to/from the broker. In these cases the consumers must reassign who is consuming.
- The cluster is adjusted. When a broker goes offline, the partitions that are lead by this broker will be reassigned. In turn the consumers must rebalance so that they consume from the new leader. When a broker comes back, then eventually a preferred leader election happens which restores the original leadership. The consumers must follow this change as well.

On the consumer side, this rebalance is propagated to the client via the ConsumerRebalanceListener interface. It has two methods. The first, onPartitionsRevoked, will be invoked when any partition goes offline. This call happens before the changes would reflect in any of the consumers, so this is the chance to save offsets if manual offset commit is used. On the other hand onPartitionsAssigned is invoked after partition reassignment. This would allow for the programmer to detect which partitions are currently assigned to the current consumer. Complete examples can be found in the development section.

Retries

Learn more about retries and how they are constrained.

In Kafka retries typically happen only for certain kinds of errors. When a retriable error is returned, the clients are constrained by two facts: the timeout period and the backoff period.

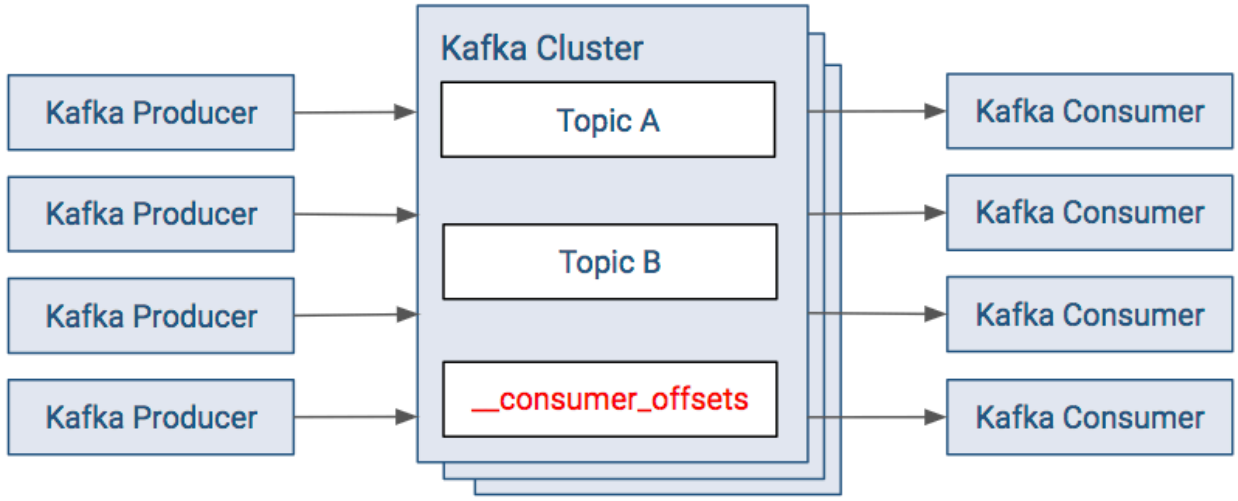
The timeout period tells how long the consumer can retry the operation. The backoff period how often the consumer should retry. There is no generic approach for "number of retries." Number of retries are usually controlled by timeout periods.

Kafka clients and ZooKeeper

Learn more about the differences between the old and new model for storing consumer offsets.

The default consumer model provides the metadata for offsets in the Kafka cluster. There is a topic named __consumer_offsets that the Kafka consumers write their offsets to.

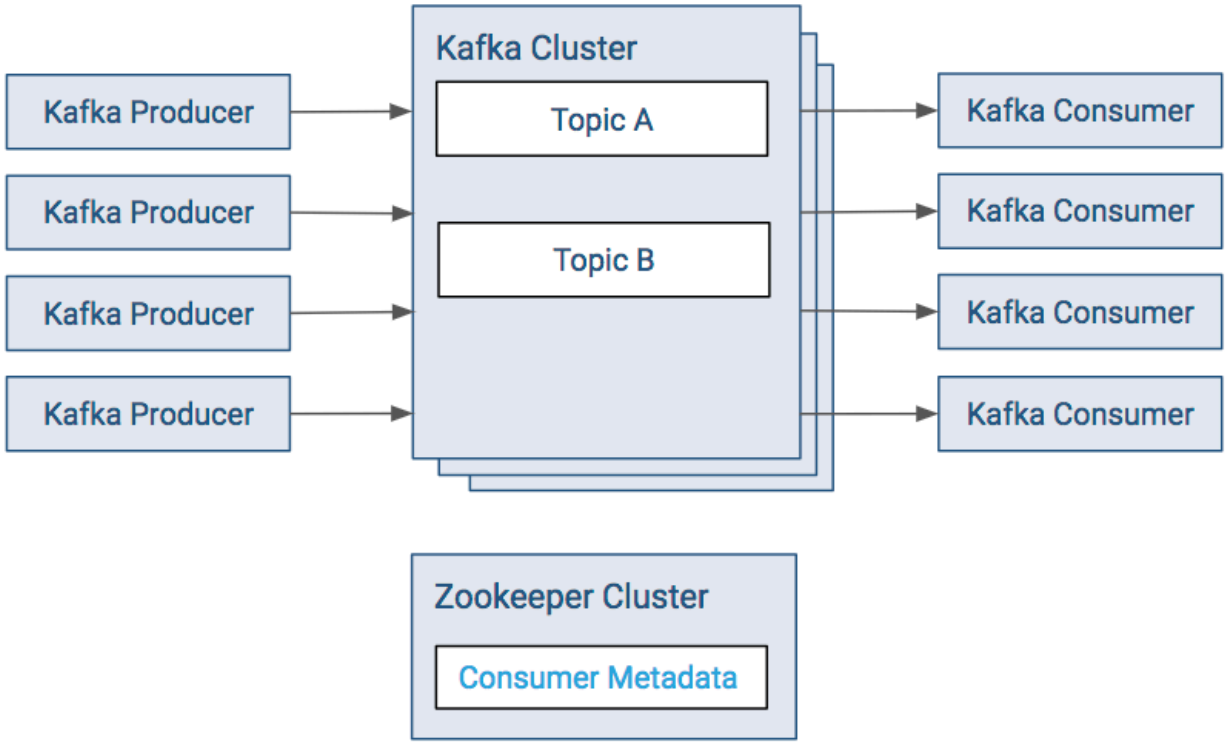
Figure 5: Kafka Consumer Dependencies



In releases before version 2.0 of CDK Powered by Apache Kafka, the same metadata was located in ZooKeeper. The new model removes the dependency and load from Zookeeper. In the old approach:

- The consumers save their offsets in a "consumer metadata" section of ZooKeeper.
- With most Kafka setups, there are often a large number of Kafka consumers. The resulting client load on ZooKeeper can be significant, therefore this solution is discouraged.

Figure 6: Kafka Consumer Dependencies (Old Approach)



Simple Client Examples

Example pom, producer and consumer.

You can produce messages to and consuming messages from a Kafka cluster using the command line. For most cases however, running Kafka producers and consumers using shell scripts and Kafka's command line scripts cannot be used in practice. In these cases, native Kafka client development is the generally accepted option.

pom.xml

An example pom.xml.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3
.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apach
e.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cloudera.kafkaexamples</groupId>
  <artifactId>kafka-examples</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>kafkadev</name>
  <url>http://maven.apache.org</url>
  <repositories>
    <repository>
      <id>cloudera</id>
      <url>https://repository.cloudera.com/artifactory/cloudera-repos/</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>[***KAFKA VERSION***]</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Note:

Replace [***KAFKA VERSION***] with the version of the kafka-clients artifact that you want to use. For available versions, see https://repository.cloudera.com/artifactory/cloudera-repos/org/apache/kafka/kafka-clients/.

SimpleConsumer.java

A simple working example of a producer program.

Note that this consumer is designed as an infinite loop. In normal operation of Kafka, all the producers could be idle while consumers are likely to be still running.

The example includes Java properties for setting up the client identified in the comments; the functional parts of the code are in bold. This code is compatible with versions as old as the 0.9.0-kafka-2.0.0 version of Kafka.

```
package com.cloudera.kafkaexamples;

import java.util.Arrays;
import java.util.Properties;
import org.apache.kafka.clients.consumer.ConsumerConfig;
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.serialization.StringDeserializer;

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

        // Set up client Java properties
        Properties props = new Properties();
        props.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                "host1:9092,host2:9092,host3:9092");
        // Just a user-defined string to identify the consumer group
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "test");
        // Enable auto offset commit
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
        props.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName());
        props.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName());

        try (KafkaConsumer<String, String> consumer = new
 KafkaConsumer<>(props)) {
            // List of topics to subscribe to
            consumer.subscribe(Arrays.asList("ufo_sightings"));
            while (true) {
                try {
                    ConsumerRecords<String, String> records =
 consumer.poll(100);
                    for (ConsumerRecord<String, String> record : records) {
                        System.out.printf("Offset = %d\n", record.offset());
                        System.out.printf("Key    = %s\n", record.key());
                        System.out.printf("Value  = %s\n", record.value());
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
```

```
    }
}
```

SimpleProducer.java

A simple working example of a producer program.

The example includes Java properties for setting up the client identified in the comments; the functional parts of the code are in bold. This code is compatible with versions as old as the 0.9.0-kafka-2.0.0 version of Kafka.

```
package com.cloudera.kafkaexamples;

import java.util.Date;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;
public class SimpleProducer {
    public static void main(String[] args) {
        // Generate total consecutive events starting with ufoId
        long total = Long.parseLong("10");
        long ufoId = Math.round(Math.random() * Integer.MAX_VALUE);
        // Set up client Java properties
        Properties props = new Properties();
        props.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                "host1:9092,host2:9092,host3:9092");
        props.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                StringSerializer.class.getName());
        props.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                StringSerializer.class.getName());
        props.setProperty(ProducerConfig.ACKS_CONFIG, "1");

        try (KafkaProducer<String, String> producer = new
 KafkaProducer<>(props)) {
            for (long i = 0; i < total; i++) {
                String key = Long.toString(ufoId++);
                long runtime = new Date().getTime();
                double latitude = (Math.random() * (2 * 85.05112878)) - 85.
05112878;
                double longitude = (Math.random() * 360.0) - 180.0;
                String msg = runtime + "," + latitude + "," + longitude;
                try {
                    ProducerRecord<String, String> data = new
                            ProducerRecord<String, String>("ufo_sightings",
 key, msg);
                    producer.send(data);
                    long wait = Math.round(Math.random() * 25);
                    Thread.sleep(wait);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Recommendations for using the producer and consumer APIs

A collection of recommendations regarding the use of the producer and consumer APIs.

After reviewing the basic examples of a producer and consumer, prototyping your own designs shouldn't be too difficult. However, your code will likely undergo several iterations that improve on scalability, debuggability, robustness, and maintainability.

This topic presents recommendations in the form of code snippets that illustrate some of the important ways to use the producer and consumer APIs.

In addition to the recommendations presented here, it is highly recommended that you also review the Javadoc for producers and consumers which have additional details about Kafka client programming.

These Javadoc pages are quite dense with information. They assume you have sufficient background in reliable computing, networking, multithreading, and distributed systems to use the APIs correctly. While the following recommendations point out many caveats in using the client APIs, the Javadoc (and ultimately the source code) provides a more detailed explanation.

Reuse your Producer/Consumer object

In these examples, the consumer constructor should be called once and the poll() method called within a loop. If this object is not reused, then a new connection to the broker is opened with each new KafkaConsumer object created.

Recommended

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

while (true) {
  ConsumerRecords<String, String> records = consumer.poll(100);
}
```

Not Recommended

```
while (true) {
  KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
  ConsumerRecords<String, String> records = consumer.poll(100);
}
```

Similarly, it is recommended that you use one KafkaConsumer and/or KafkaProducer object per thread. Creating more objects opens multiple ports per broker connection. Overusing ephemeral ports can cause performance issues.

In addition, Cloudera recommends to set and use a fixed client.id for producers and consumers when they are connecting to the brokers. If this is not done, Kafka will assign a new client id every time a new connection is established, which can severely increase resource utilization (memory) on the broker side.

Each KafkaConsumer object requires calling poll() frequently

As explained in the Apache Kafka documentation topic New Consumer Configs, any consumer connected to a partition will time out if poll() is not called within the period defined by max.poll.interval.ms.

In the example below, the call to myDataProcess.doStuff(records) can cause poll() to be called infrequently. This could be due to a combination of reasons:

- Being a blocking method call.
- Doing work on a remote machine.
- Having highly variable processing time.

- Saving to storage that has highly variable I/O throughput.

In such cases, consider having another thread or process doing the actual work and making the handoff as lightweight as possible.

Example: poll() gets KafkaException due to session timeout

```
while (true) {
  KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
  ConsumerRecords<String, String> records = consumer.poll(100);
  // the call below should return quickly in all cases
  myDataProcess.doStuff(records);
}
```

Catch all exceptions from poll()

From the poll() Javadoc page, you can see that the poll() method throws several exceptions. If the catch statements (bold in the example) are not complete, then any uncaught exception will end up in the finally statement calling Kafk aConsumer#close(). This will not be the desired behavior in many cases.

```
while (true) {
  try {
    ConsumerRecords<String, String> records = consumer.poll(100);
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    consumer.close();
  }
```

Callback#onCompletion() should always exit without errors

The interface org.apache.kafka.clients.producer.Callback (Javadoc) is used to define a class that can be used upon completion of a KafkaProducer#send() call. It allows for tracking, clean up, or other administrative code to be called. An example of unintended usage is to call KafkaProducer#send() within the Callback#onCompletion() method, essentially mimicking a retry. Because the onCompletion() method is always expected to return cleanly and the send () method makes no such guarantees, calling send() within the callback could end up hanging the code in case of network or broker issues.

Check your API usage against the latest API

The documentation for the latest upstream release of Apache Kafka indicates if there have been any changes to how the APIs are used (setup, read, write). Reviewing the latest information could help avoid upgrade-related changes to your producer or consumer.

Some examples from past versions include:

Old Class or Package	New Class or Package
kafka.producer.ProducerConfig	java.util.Properties
kafka.javaapi.*	kafka.api.*
kafka.producer.KeyedMessage	kafka.clients.producer.ProducerRecord

Hidden Dependency on Network Availability

Network dependency is one of the more subtle issues. Given the consumer dependencies on Ranger and Zookeeper, having a combination of frequent or prolonged DNS or network outages can also cause various session timeouts to occur. Such timeouts will force partition rebalancing on the brokers, which will worsen general Kafka reliability.

Should these issues be common in your network, you may need to have a less straightforward design that can handle such reliability issues outside of the Kafka client.

Related Information
KafkaConsumer Javadoc
KafkaProducer Javadoc

Kafka public APIs

Learn what is and what is not considered a Kafka public API by Cloudera.

What is a Public API

The following parts of Apache Kafka in CDP are considered as public APIs:

- Kafka wire protocol format: the format itself might change, but brokers will be able to use the old format as long as documentation and upgrade instructions are followed properly.
- Binary log format: the format itself might change, but brokers will be able to use the old format as long as documentation and upgrade instructions are followed properly.
- Interfaces and classes in the following packages:
  - org/apache/kafka/common/serialization
  - org/apache/kafka/common/errors
  - org/apache/kafka/clients/producer
  - org/apache/kafka/clients/consumer
- Command-line admin tools: arguments, except ZooKeeper related options, that are subject to change and/or removal.
- HttpMetricsReporter: existing fields will stay backward compatible, but new fields may be introduced. The only public API of HttpMetricsReporter is the /api/metrics REST endpoint. For a list of supported metrics, see Kafka Metrics.
- Properties, excluding their default values
- Config file content and format, and the effect of configuration attributes
- Endpoints

What is NOT a public API

There are structures that third parties might regard as an interface but Cloudera Kafka distributions do not consider them public APIs. In general, any API that is not listed as public in the *What is a Public API* section should be considered private, and client code should not rely on behavior/data content or format. Some examples are:

- Data structures in ZooKeeper: the content and format what Kafka stores in ZooKeeper are internal implementation details.
- Authorizer interface: the only supported authorizer in CDP is the Ranger one.
- AdminClient: it is a new and rapidly evolving part of Kafka, so Cloudera can’t provide the same guarantees as for other interfaces.
- Interfaces marked with the @Evolving or @Unstable annotations in the Kafka source code
- Index files generated by Kafka
- Application log file content and format (for example what Log4J/SLF4J/… produces)
- Any classes used for testing
- Relying on transitive dependencies: any dependency pulled in by Kafka
- Any other interfaces not listed above
- Anything that Cloudera does not support, even if it fits the definition of a public API

Related Information
Kafka Metrics

Kafka Streams

Learn more about Kafka Streams.

You can access the Apache Kafka website for information on how to use Kafka Streams.

- Read the Kafka Streams Introduction for an overview of the feature and an introductory video.
- Get familiar with Kafka Streams Core Concepts.
- Understand Kafka Streams Architecture.
- Access the Quick Start documentation to run a demonstration Kafka Streams Application.
- Use the Tutorial to write your first Kafka Streams Application.