Cloudera Runtime 7.3.2

Developing Apache Kafka Applications

Date published: 2019-08-22
Date modified: 2026-03-31



https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2026. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Kafka producers..... 4

Kafka consumers.....5
Subscribing to a topic.....7
Groups and fetching..... 9
Protocol between consumer and broker.....10
Rebalancing partitions..... 13
Retries.....13

Java client..... 14
Java Client examples..... 15
Simple Java consumer..... 15
Simple Java producer..... 16
Java client security examples..... 17

.NET client.....18
.Net client examples..... 18
Simple .NET consumer example..... 19
Simple .NET producer example..... 19
Performant .NET producer example.....20
Simple .Net consumer using Schema Registry..... 21
Simple .Net producer using Schema Registry.....22
.NET client security examples..... 24

Kafka Streams.....26

Kafka public APIs..... 26

Recommendations for using the producer and consumer APIs.....27

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 producer API available in your chosen client library. To instantiate a producer:

For Java

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

For C#

```
var producer = new ProducerBuilder<string, string>(config).Build();
```

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

Serialization of Keys and Values

For Java

For each producer, there are two serialization properties that must be set, key.serializer (for the key) and value.serializer (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

For C#

Serialization is passed to the producer builder object for key and value serialization:

```
var producer = new ProducerBuilder<string, string>(producerConfi
g)
                    .SetKeySerializer(Serializers.Utf8)
                    .SetValueSerializer(Serializers.ByteArray)
                    .Build())
```

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:

For Java

- 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.

For C#

- BatchNumMessages: Combine this fixed number of records before sending data to the cluster.
- LingerMS: Always wait at least this amount of time before sending data to the cluster; then send however many records has accumulated in that time.
- CompressionType: Enable compression of data being sent.
- MessageSendMaxRetries: Maximum number of retries for sending a failed message.

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.

For Java

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).

For C#

You can set what partitioner to use with the Partitioner property of ProducerConfig. By default the consistent_random partitioner is used. In C# you define the key and value types in the ProducerBuilder. When a new message is sent to a topic, a Message<Key,Value> object is processed with the key and value types specified in ProducerBuilder.

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 consumer API available in your chosen client library.

To instantiate a consumer:

For Java

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

For C#

```
var consumer = new ConsumerBuilder<string, string>(config).Build();
```

The consumer 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 an interface:

- For Java implement the org.apache.kafka.common.serialization.Deserializer interface
- For C# implement the Confluent.Kafka.IDeserializer interface

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

For Java

- 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.
- key.deserializer/value.deserializer: Specify how to convert the Java representation to a sequence of bytes to send data through the Kafka protocol.

For C#

- BootstrapServers: A list of brokers to initially connect to. List 2 to 3 brokers; you don't need to list the full cluster.
- GroupId: Every consumer belongs to a group. That way they'll share the partitions of a topic.
- key.deserializer/value.deserializer: Specify how to convert the Java representation to a sequence of bytes to send data through the Kafka protocol.
- Key and value deserializer: In C# the key and value deserializers are not configuration parameters, instead they are passed to the builder object. For example:

```
var consumer = new ConsumerBuilder<string, GenericRecord>(config)
                        .SetKeyDeserializer(new AvroDeserializer<strin
g>(schemaRegistry).AsSyncOverAsync())
                        .SetValueDeserializer(new AvroDeserializer<Gen
ericRecord>(schemaRegistry).AsSyncOverAsync())
                        .Build()
```

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:

For Java

- 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.

For C#

- HeartbeatIntervalMs: 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.
- SessionTimeoutMs: 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.
- MaxPollIntervalMs: 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.

Subscribing to a topic

Learn more about subscribing to a topic.

Subscribing to a topic

In order for the consumer to be able to consume messages, it first needs to subscribe to a topic. This can be done using a subscribe method:

For Java

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

For C#

```
consumer.Subscribe(new List<string>() { topic });
```

Here you specify a list of topics that you want to consume from. In Java a 'rebalance listener' is also specified. 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.

Polling

After subscribing to a topic, the consumer has to poll to see if there are new records.

For Java

The poll () method 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.

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

For C#

The Consume() method returns a single result which can either be a single Kafka message, or an end of partition event if the IsPartitionEOF property is true.

```
while (true)
```

```
{
    var result = consumer.Consume();
    // Handle result
}
```

Saving progress

The last important point is to save the progress.

In Java this can be done with by the commitSync() and commitAsync() methods respectively. Auto commit is not recommended; manual commit is appropriate in the majority of use cases.

In C# this can be done by calling the Close() method at the end of message processing which automatically commits offsets. Alternatively, this can also be done manually by calling the Commit() method:

For Java
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
```

```
                    }
                });
            }
```

For C#
Close()

```
consumer.Close();  // commit offset and unsubscribe
```

Commit()

```
consumer.Commit(); // commit offset
```

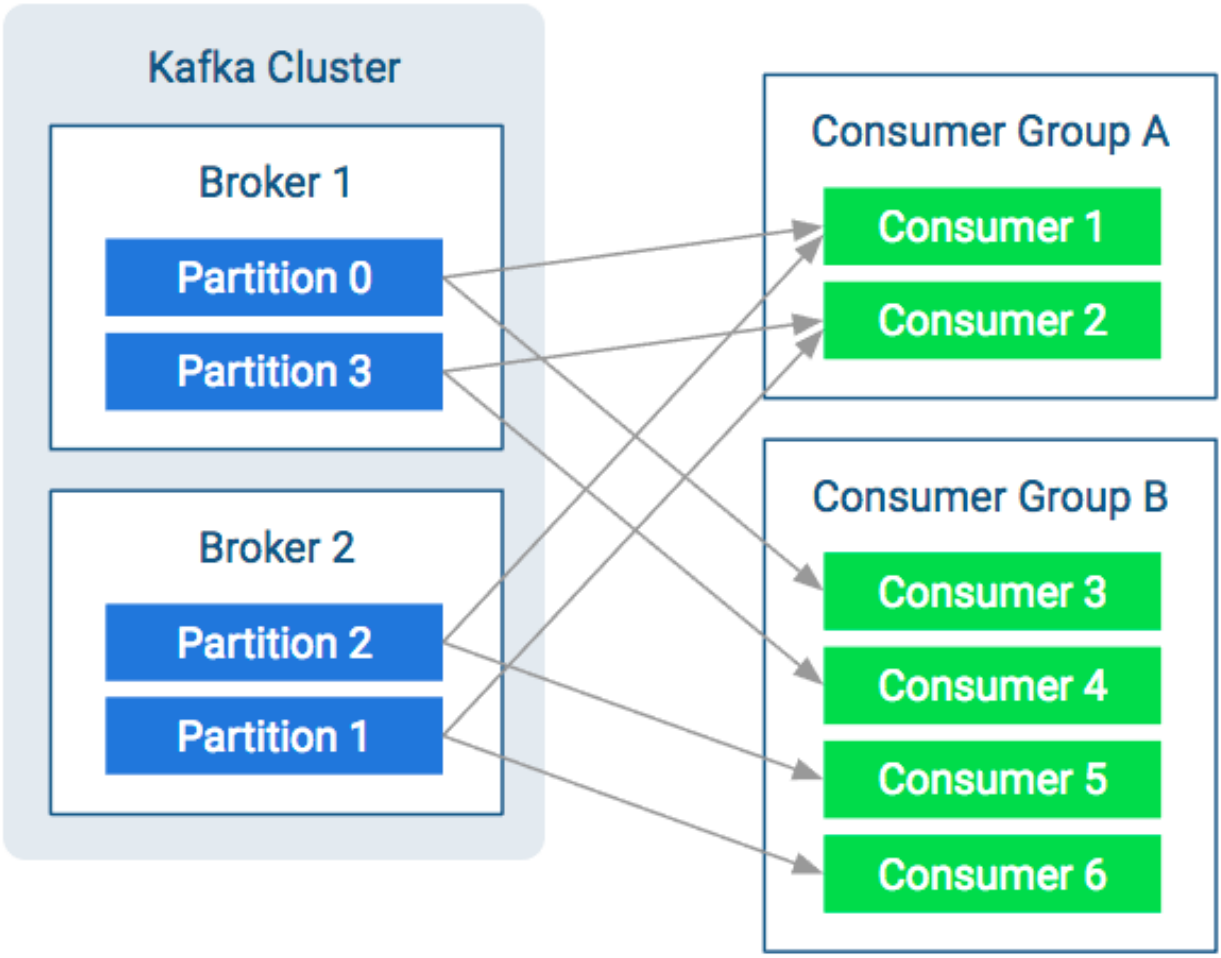
Groups and fetching

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

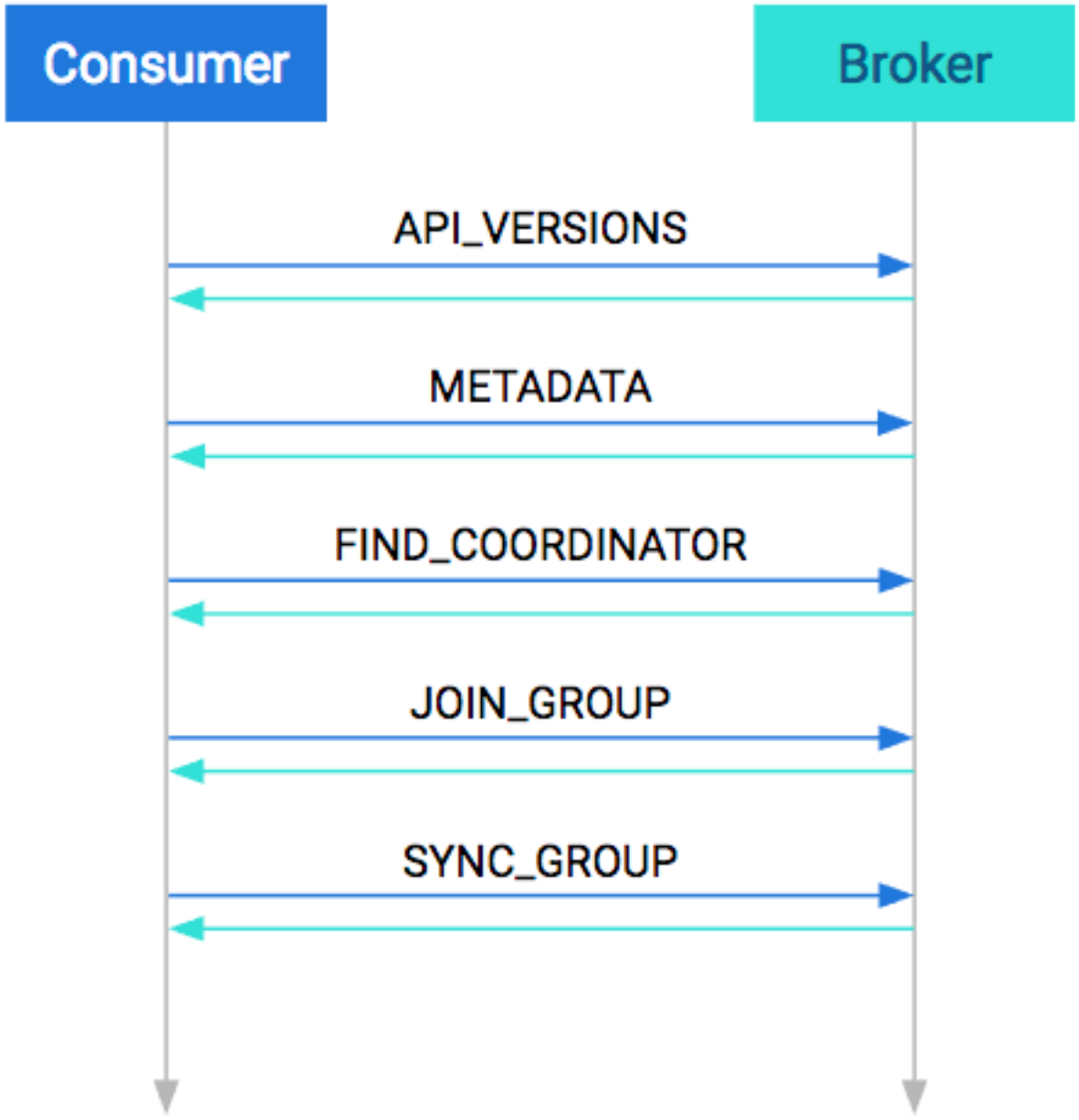
assignment strategy, it is decoupled from the broker. That means consumers can reconfigure the partition assignment strategy without restarting the brokers.

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.

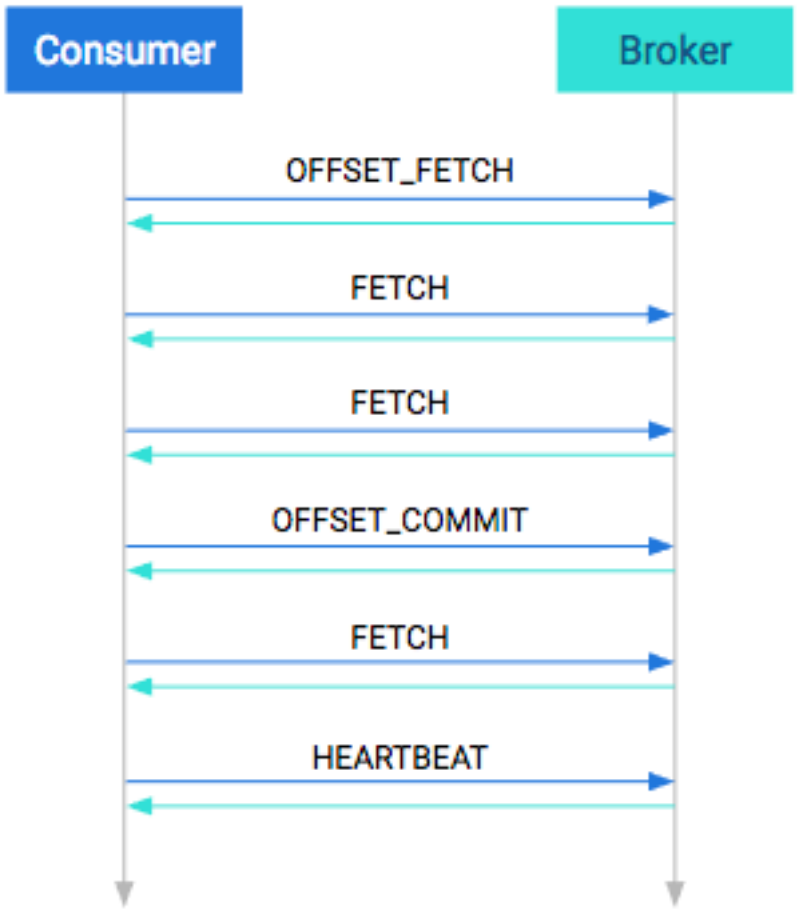
After finding the coordinator, the consumer(s) are ready to join the group. Every consumer in the group sends their own member-specific metadata to the coordinator in the JOIN_GROUP request. The coordinator waits until all the consumers have sent their request, then assigns a leader for the group. At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

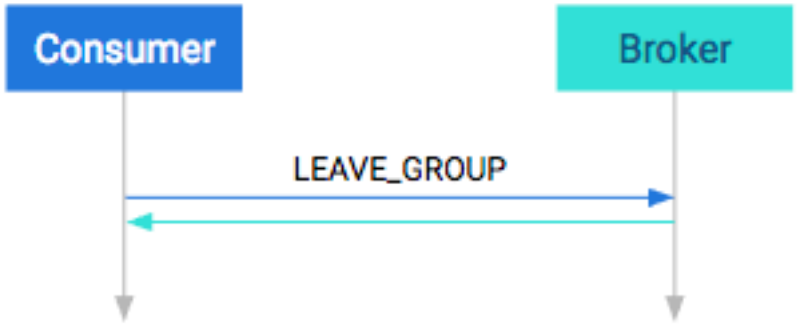


From time to time, the application has to either manually or automatically save the offsets in an OFFSET_COMMIT request and send heartbeats too in the HEARTBEAT requests. The first ensures that the position is saved while the latter ensures that the coordinator knows that the consumer is alive.

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.

Java client

Learn how the Java client libraries are distributed and how you can add the necessary artifacts to your projects.

Cloudera Runtime is shipped with the Java client libraries that are maintained as part of Apache Kafka. These are available in the Cloudera Maven Repository. Review the following POM example to understand how you can add the necessary artifacts to your project.

```
<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.apache
.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/.

Related Information

Cloudera Maven Repository

Java Client examples

You can produce messages to and consume 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. Review these code example to better understand how you can develop your own clients using the Java client library.

Simple Java consumer

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();
                }
            }
```

```
        }
    }
}
```

Simple Java producer

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();
                }
            }
        }
    }
}
```

Java client security examples

Review the Java client security examples to learn what configuration properties you have to set when connecting to secured or unsecured clusters.

The following code snippets give you a few simple examples on what configuration properties you need to set for your Kafka clients when connecting them to either secured or unsecured Kafka clusters. Use the following examples as a starting point and make changes as necessary. Depending on your environment, you may need to set other optional properties. See the upstream Apache Kafka documentation for a comprehensive list of available properties.

Unsecured

```
Properties producerConfig = new Properties();
producerConfig.put("bootstrap.servers", "***BROKER HOST***:***PORT***");
```

SSL/TLS

```
Properties producerConfig = new Properties();
producerConfig.put("bootstrap.servers", "***BROKER HOST***:***PORT***");
producerConfig.put(“security.protocol”, “SSL”);
producerConfig.put(“ssl.truststore.location”, “***PATH TO TRUSTSTORE***”);
// .jks format
producerConfig.put(“ssl.truststore.password”, “***TRUSTSTORE PASSWORD***”);
producerConfig.put(“ssl.keystore.location”, “***PATH TO KEYSTORE***”); // .
jks format
producerConfig.put(“ssl.keystore.password”, “***KEYSTORE PASSWORD***”);
```

PLAIN (LDAP, PAM, and others)

Important: Credentials are sent across the wire, Cloudera recommends that you always use channel encryption.

```
Properties producerConfig = new Properties();
producerConfig.put("bootstrap.servers", "***BROKER HOST***:***PORT***");
producerConfig.put(“security.protocol”, “SASL_SSL”);
producerConfig.put(“sasl.mechanism”, “PLAIN”);
producerConfig.put(“ssl.truststore.location”, “***PATH TO TRUSTSTORE***”);
// .jks format
producerConfig.put(“ssl.truststore.password”, “***TRUSTSTORE PASSWORD***”);
producerConfig.put(“sasl.jaas.config”, “org.apache.kafka.common.secu
rity.plain.PlainLoginModule required username="***USERNAME***" passwo
rd="***PASSWORD***"); // credentials sent across the wire, also enable SSL/
TLS encryption
```

Kerberos

```
Properties producerConfig = new Properties();
producerConfig.put("bootstrap.servers", "***BROKER HOST***:***PORT***");
producerConfig.put(“security.protocol”, “SASL_PLAINTEXT”);
producerConfig.put(“sasl.kerberos.service.name”, “***SERVICE NAME***”); // k
afka
producerConfig.put(“sasl.jaas.config”, “com.sun.security.auth.module.Krb5Lo
ginModule required useTicketCache=true;”); // use kerberos ticket cache
```

OAuth2/OAUTHBEARER

```
Properties producerConfig = new Properties();
producerConfig.put("bootstrap.servers", "***BROKER HOST***:***PORT***");
producerConfig.put("security.protocol", "SASL_PLAINTEXT");
producerConfig.put("sasl.mechanism", "OAUTHBEARER");
producerConfig.put("sasl.login.callback.handler.class", org.apache.kafka.com
mon.security.oauthbearer.secured.OAuthBearerLoginCallbackHandler.class.getNa
me());
producerConfig.put("sasl.jaas.config", "org.apache.kafka.common.security.oau
thbearer.OAuthBearerLoginModule required " +
                    "clientId=\"***CLIENT_ID***\" " +
                    "clientSecret=\"***CLIENT_SECRET***\" " +
                    "scope=\"***SCOPE***\";"
     );
producerConfig.put("sasl.oauthbearer.token.endpoint.url", "http://***OAUTH
 SERVER***/***TOKEN ENDPOINT***");
```

.NET client

Learn more about how you can download and install the Cloudera .NET client library for Apache Kafka.

Cloudera's .NET client library for Apache Kafka, called Cloudera.Kafka, is available through NuGet. This client library is not distributed or packaged with Cloudera Runtime.

Installation

The client library can be downloaded and installed through Visual Studio or a command line interface. Alternatively, you can also download it directly from nuget.org.

Visual Studio

In Visual Studio you can use the NuGet Package Manager UI to find and install the Cloudera.Kafka package for the appropriate project.

Command line interface

The Cloudera.Kafka package can be installed through a command line interface. You can use the Package Manager Console or the dotnet CLI.

Package Manager Console:

```
Install-Package Cloudera.Kafka -Version [***VERSION***]
```

dotnet CLI:

```
dotnet add package Cloudera.Kafka --version [***VERSION***]
```

Direct download

The Cloudera.Kafka package is available on the NuGet website at https://www.nuget.org/packages/Cloudera.Kafka/.

.Net client examples

You can produce messages to and consume 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, built-in Kafka client development is the generally accepted option. Review these code example to better understand how you can develop your own clients using the Cloudera .Net client library.

Simple .NET consumer example

Review this example code to learn how you can create a simple .NET consumer. This consumer is capable of consuming around 50000 messages every second in development environment.

```
var config = new ConsumerConfig
{
    BootstrapServers = "broker:9092",
    GroupId = "simple-dotnet-consumer",
    AutoOffsetReset = AutoOffsetReset.Earliest,
    EnablePartitionEof = true
};
using var consumer = new ConsumerBuilder<string, string>(config)
    .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}"))
    .Build();
consumer.Subscribe(new List<string>() {"topic"});
var start = DateTime.Now;
long messageCounter = 0;
try
{
    while (!(Console.KeyAvailable && Console.ReadKey().Key == ConsoleKey.Q))
    {
        var result = consumer.Consume(TimeSpan.FromMilliseconds(100));
        if (result == null) { continue; }
        if (result.IsPartitionEOF) { break; }

        ++messageCounter;
        if (messageCounter % 1024 == 0) { Console.WriteLine($"Received mess
age key: \"{result.Message.Key}\" value: {result.Message.Value}"); }
    }
} catch (OperationCanceledException) {}

consumer.Close();  // commit offset and unsubscribe
var elapsed = DateTime.Now - start;
Console.WriteLine("average throughput: {0:N3} msg/sec, {1} messages over {2
:N3} sec", messageCounter / elapsed.TotalSeconds, messageCounter, elapsed.To
talSeconds);
```

Simple .NET producer example

Review this example code to learn how you can create a simple .NET producer. This producer is capable of producing around 36000 messages every second in development environment.

```
internal class Program
{
    private static int _inFlight;
    private static long _delivered;

    private static void HandleDeliveryResult(int msgid, DeliveryResult<strin
g, string> deliveryResult)
    {
        if (msgid % 1024 == 0)  // writing to console on every message would
 be a bottleneck
        {
            Console.WriteLine($"Delivered '{deliveryResult.Value}' to '{deli
veryResult.TopicPartitionOffset}', in flight on delivery confirmation: {_inF
light}");
        }
        --_inFlight;
        ++_delivered;
    }
```

```
    private static void Main(string[] args)
    {
        using var producer = new ProducerBuilder<string, string>(new Produce
rConfig { BootstrapServers = "broker:9092" }).Build();
        int msgCounter = 0;
        var start = DateTime.Now;
        while (!(Console.KeyAvailable && Console.ReadKey().Key == ConsoleKe
y.Q || (DateTime.Now - start).TotalSeconds >= 60))
        {
            int msgid = ++msgCounter;
            try
            {
                producer.Produce("topic", new Message<string, string> { Key
 = msgid.ToString(), Value = $"{msgid}, in flight on send: {++_inFlight}" },
                    result => HandleDeliveryResult(msgid, result));
            }
            catch (ProduceException<string, string> e)
            {
                Console.WriteLine($"Produce failed: {e.Error.Reason}");
            }
        }
        producer.Flush();
        Console.WriteLine("average throughput: {0:N3} msg/sec", _delivered /
 (DateTime.Now - start).TotalSeconds);
    }
}
```

Performant .NET producer example

Review this example code to learn how you can create a more complex, high-performing .NET producer with async processing and TPL. This producer is capable of producing around 50000 messages every second in development environment.

```
internal class Program
{
    private static int _inFlight;
    private static long _delivered;

    private static void HandleDeliveryResult(int msgid, DeliveryResult<strin
g, string> deliveryResult)
    {
        if (msgid % 1024 == 0)  // writing to console on every message would
 be a bottleneck
        {
            Console.WriteLine($"Delivered '{deliveryResult.Value}' to '{deli
veryResult.TopicPartitionOffset}', in flight on delivery confirmation: {_inF
light}");
        }
        Interlocked.Decrement(ref _inFlight);
        Interlocked.Increment(ref _delivered);
    }

    private static async Task Main(string[] args)
    {
        const int inFlightRequests = 16384;
        const int bufferedMessages = inFlightRequests * 4;
        var config = new ProducerConfig { BootstrapServers = "broker:9092",
 LingerMs = 50, QueueBufferingMaxMessages = bufferedMessages };
        using var p = new ProducerBuilder<string, string>(config).Build();
        var actionBlock = new ActionBlock<int>(
            msgid => p.ProduceAsync("topic", new Message<string, string> {
Key=msgid.ToString(), Value = $"{msgid}, in flight on send: {Interlocked.Inc
rement(ref _inFlight)}" }
```

```
                .ContinueWith(async task => HandleDeliveryResult(msgid, aw
ait task)),
            new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism = inF
lightRequests, BoundedCapacity = inFlightRequests });

        int msgCounter = 0;
        bool accepted = true;
        var start = DateTime.Now;
        while (accepted && !(Console.KeyAvailable && Console.ReadKey().Key
 == ConsoleKey.Q || (DateTime.Now - start).TotalSeconds >= 60))
        {
            try
            {
                accepted = await actionBlock.SendAsync(msgCounter++);
            }
            catch (ProduceException<string, string> e)
            {
                Console.WriteLine($"Produce failed: {e.Error.Reason}");
            }
        }
        actionBlock.Complete();
        await actionBlock.Completion;
        Console.WriteLine("average throughput: {0:N3} msg/sec", _delivered /
 (DateTime.Now - start).TotalSeconds);
    }
}
```

Simple .Net consumer using Schema Registry

Review this example code to learn how you can create a simple .NET consumer which is using Schema Registry.

```
internal class Program
{
    private static void Main(string[] args)
    {
        var schemaRegistryConfig = new SchemaRegistryConfig {
            Url = "http://schemareg:7788/api/v1"
        };

        var consumerConfig = new ConsumerConfig {
            BootstrapServers = "broker:9092",
            GroupId = "simple-dotnet-consumer",
            AutoOffsetReset = AutoOffsetReset.Earliest,
            EnablePartitionEof = true
        };

        using (var schemaRegistry = new CachedSchemaRegistryClient(schemaR
egistryConfig))
        using (var consumer = new ConsumerBuilder<string, GenericRecord>(con
sumerConfig)
            .SetValueDeserializer(new AvroDeserializer<GenericRecord>(schem
aRegistry).AsSyncOverAsync())
            .Build())
        {
            consumer.Subscribe("topic");
            var start = DateTime.Now;
            long messageCounter = 0;
            try
            {
                while (!(Console.KeyAvailable && Console.ReadKey().Key == Co
nsoleKey.Q))
                {
```

```
                    var result = consumer.Consume(TimeSpan.FromMillisecon
ds(100));
                    if (result == null) { continue; }
                    if (result.IsPartitionEOF) { break; }

                    ++messageCounter;
                    if (messageCounter % 1024 == 0) {
                        Console.WriteLine(
                            $"Received message key: \"{result.Message.Key
}\" " +
                            $"value: < {result.Message.Value["message_id"]},
 in flight on send: {result.Message.Value["in_flight_on_send"]} >");
                    }
                }
            } catch (OperationCanceledException) {}

            consumer.Close();  // commit offset and unsubscribe

            var elapsed = DateTime.Now - start;
            Console.WriteLine("average throughput: {0:N3} msg/sec, {1} mess
ages over {2:N3} sec", messageCounter / elapsed.TotalSeconds, messageCounter
, elapsed.TotalSeconds);
        }
    }
}
```

Simple .Net producer using Schema Registry

Review this example code to learn how you can create a simple .NET producer which is using Schema Registry.

```
internal class Program
{
    private static int _inFlight;
    private static long _delivered;

    private static void HandleDeliveryResult(int msgid, DeliveryResult<stri
ng, GenericRecord> deliveryResult)
    {
        if (msgid % 1024 == 0)  // writing to console on every message would
 be a bottleneck
        {
            Console.WriteLine($"Delivered '{deliveryResult.Value}' to '{deli
veryResult.TopicPartitionOffset}', in flight on delivery confirmation: {_inF
light}");
        }
        --_inFlight;
        ++_delivered;
    }

    private static void Main(string[] args)
    {
        var schema = (Avro.RecordSchema) Avro.Schema.Parse(@"{
                ""type"": ""record"",
                ""name"": ""ExampleRecord"",
                ""namespace"": ""ExampleSchema"",
                ""fields"": [
                    {
                        ""name"": ""message_id"",
                        ""type"": ""int""
                    },
                    {
                        ""name"": ""in_flight_on_send"",
                        ""type"": ""int""
```

```
                    }
                ]
            }");

        var schemaRegistryConfig = new SchemaRegistryConfig { Url = "http://
schemareg:7788/api/v1" };
        var producerConfig = new ProducerConfig { BootstrapServers = "brok
er:9092" };

        using (var schemaRegistry = new CachedSchemaRegistryClient(schemaR
egistryConfig))
        using (var producer = new ProducerBuilder<string, GenericRecord>(pro
ducerConfig)
            .SetValueSerializer(new AvroSerializer<GenericRecord>(schemaReg
istry, new AvroSerializerConfig
            {
                ProtocolVersion = AvroSerializerConfig.ProtocolVersions.Sc
hemaMetadataIdAsVersionProtocol
            }))
            .Build())
        {
            int msgCounter = 0;
            var start = DateTime.Now;
            while (!(Console.KeyAvailable && Console.ReadKey().Key == Cons
oleKey.Q || (DateTime.Now - start).TotalSeconds >= 60))
            {
                int msgid = ++msgCounter;

                var record = new GenericRecord(schema);
                record.Add("message_id", msgid);
                record.Add("in_flight_on_send", ++_inFlight);

                try
                {
                    producer.ProduceAsync("topic", new Message<string, Gene
ricRecord>
                        {
                            Key = msgid.ToString(),
                            Value = record
                        }
                    ).ContinueWith(task => HandleDeliveryResult(msgid, tas
k.Result));
                }
                catch (ProduceException<string, string> e)
                {
                    Console.WriteLine($"Produce failed: {e.Error.Reason}");
                }
            }
            producer.Flush();
            Console.WriteLine("average throughput: {0:N3} msg/sec", _delive
red / (DateTime.Now - start).TotalSeconds);
        }
    }
}
```

Supported values for ProtocolVersion

The Schema Registry client consists of two major components: The REST client and the serializers (SerDes). You are supposed to define the record schemas, and to set the respective SerDes to Kafka client. When the first message with the given schema definition is produced, the REST client stores it on the Schema Registry server. The server might have multiple ways to reference these schemas from the client. Each message produced to Kafka is transformed to contain an ID to the schema stored on the Schema Registry server, so when the other side consumes these messages,

it is able to download these schemas by the IDs. The format of these IDs are denoted by the first byte of the message, which is called the Magic Byte or Protocol ID.

Note: Only Confluent protocol was supported until 1.8.2. Since 1.9.0, consumers can also handle non-confluent protocols.

ProtocolVersion	Integer value	Specification
AvroSerializerConfig.ProtocolVersions.ConfluentProtocol	0	https://registry-project.readthedocs.io/en/latest/serdes.html#confluent-protocol
AvroSerializerConfig.ProtocolVersions.SchemaMetadataIdAsVersionProtocol	1	https://registry-project.readthedocs.io/en/latest/serdes.html#schema-metadata-id-and-version-protocol
AvroSerializerConfig.ProtocolVersions.SchemaVersionIdAsLongProtocol	2	https://registry-project.readthedocs.io/en/latest/serdes.html#schema-version-id-as-long-protocol
AvroSerializerConfig.ProtocolVersions.SchemaVersionIdAsIntProtocol	3	https://registry-project.readthedocs.io/en/latest/serdes.html#schema-version-id-as-int-protocol

.NET client security examples

Review the .NET client security examples to learn what configuration properties you have to set when connecting to secured or unsecured clusters.

The following code snippets give you a few simple examples on what configuration properties you need to set for your Kafka clients when connecting them to either secured or unsecured Kafka clusters. Use the following examples as a starting point and make changes as necessary.

Unsecure

```
var producerConfig = new ProducerConfig
{
   BootstrapServers = "***BROKER HOST***:***PORT***"
};
```

SSL

Note: The Kafka .NET client does not use a truststore (.jks) like the Java clients. Instead, the Kafka .NET client requires configuring the location of the broker certificate authority (CA) certificate in .pem format by setting the SslCaLocation property.

```
var producerConfig = new ProducerConfig
{
   BootstrapServers = "***BROKER HOST***:***PORT***",
   SecurityProtocol = SecurityProtocol.Ssl,
   SslCaLocation = "***PATH TO BROKER CA CERTIFICATE***",
   SslKeystoreLocation = "***KEYSTORE LOCATION***", // client's keystore,
 pkcs12 format
   SslKeystorePassword = "***KEYSTORE PASSWORD***" // client's keystore p
assword
 };
```

PLAIN (LDAP, PAM, and others)

Note: Credentials are sent across the wire, Cloudera recommends that you always use channel encryption.

```
var producerConfig = new ProducerConfig
{
   BootstrapServers = "***BROKER HOST***:***PORT***",
   SecurityProtocol = SecurityProtocol.SaslSsl,
   SaslMechanism = SaslMechanism.Plain,
   SslCaLocation = "***PATH TO BROKER CA CERTIFICATE***",
   SaslUsername = "***USERNAME***",
   SaslPassword = "***PASSWORD***"
};
```

Kerberos

Important:

Windows does not support Kerberos principals or keytabs for authentication, it only supports Kerberos authentication through SSPI, The native windows authentication interface, which uses the currently logged on user as its credentials.

```
var producerConfig = new ProducerConfig
{
   BootstrapServers = "***BROKER HOST***:***PORT***",
   SecurityProtocol = SecurityProtocol.Sasl,
   SaslMechanism = SaslMechanism.Gssapi,
   SaslKerberosServiceName = "kafka"
};
```

Schema Registry

Review the .NET client security examples to learn what configuration properties you have to set when connecting to secured or unsecured clusters.

The following code snippets give you a few simple examples on what configuration properties you need to set for your Schema Registry clients when connecting them to either secured or unsecured Schema Registry. Use the following examples as a starting point and make changes as necessary.

Unsecure

```
var schemaRegistryConfig = new SchemaRegistryConfig {
    Url = "http://***SCHEMA REGISTRY HOST***:***PORT***/api/v1"
};
```

SSL/Mutual TLS

```
var schemaRegistryConfig = new SchemaRegistryConfig {
    Url = "https://***SCHEMA REGISTRY HOST***:***PORT***/api/v1",
    SslCaLocation = "***PATH TO SCHEMA REGISTRY CA CERTIFICATE**
*",
    SslKeystoreLocation = "***KEYSTORE LOCATION***",
    SslKeystorePassword = "***KEYSTORE PASSWORD***"
};
```

PLAIN/Knox authentication

```
var schemaRegistryConfig = new SchemaRegistryConfig {
    Url = "https://***SCHEMA REGISTRY HOST***:***PORT***/api/v1",
```

```
    SslCaLocation = "***PATH TO SCHEMA REGISTRY CA CERTIFICATE**
*",
    BasicAuthCredentialsSource = AuthCredentialsSource.UserInfo,
    BasicAuthUserInfo = "***USERNAME***:***PASSWORD***"
};
```

Note: Kerberos authentication is currently not supported by the .NET Schema Registry client.

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.

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 CDH 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 on page 26 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 Cloudera 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

Recommendations for using the producer and consumer APIs

Review the following collection of code snippets and recommendations regarding the use of the producer and consumer APIs to learn how you can develop better Kafka clients.

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 or any other API documentation available for the client library you are using. These will have additional details about Kafka client programming .

API documentation is known to be 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 API documentation (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 consumer object created.

For Java
Recommended

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

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

Not Recommended

```
while (true) {
```

```
  KafkaConsumer<String, String> consumer = new KafkaConsumer<>(pr
ops);
  ConsumerRecords<String, String> records = consumer.poll(100);
}
```

For C#
Recommended

```
var consumer = new ConsumerBuilder<string, string>(config).Build
();
while (true)
{
    var result = consumer.Consume();
}
```

Not recommended

```
while (true)
{
    var consumer = new ConsumerBuilder<string, string>(config).Bu
ild();
    var result = consumer.Consume();
}
```

Similarly, it is recommended that you use one consumer and/or producer 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 frequent polling

Any consumer connected to a partition will time out if no polling is done before the maximum poll interval is reached.

In the example below, the call to myDataProcess.doStuff(records) can cause infrequent polling. This could be due to a combination of reasons. 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.

For Java
Example: poll() gets KafkaException due to session timeout

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

For C#

Example: Consume() gets KafkaException due to session timeout

```
while (true) {
  var consumer = new ConsumerBuilder<string, string>(config).Buil
d();
  var result = consumer.Consume(100);
  // the call below should return quickly in all cases
  myDataProcess.DoStuff(records);
}
```

Catch all exceptions when polling

For Java

From the poll() Javadoc page, you can see that the poll() method throws several exceptions. If the catch statements are not complete, then any uncaught exception will end up in the finally statement calling KafkaConsumer#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();
  }
```

For C#

In the .NET client some overloads can only throw one, but another overload can throw multiple exceptions. Make sure that all exceptions are caught.

```
while (true)
{
  try
  {
    var result = consumer.Consume();
  }
  catch (Exception e)
  {
    Console.WriteLine($"Exception caught:\n {e.StackTrace}");
  }
  finally
  {
    consumer.Close();
  }
}
```

Callback#onCompletion() should always exit without errors

Note: This recommendation is for the Java client only.

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