Kafka Clients (At-Most-Once, At-Least-Once, Exactly-Once, and Avro Client)

Detailed tutorials and code snippets for setting up specific messaging scenarios in Apache Kafka.

by

Ajmal Karuthakantakath

·

Jan. 15, 16 · Big Data Zone

Liked (18)

Comment (5)

Saved

Tweet

29.85k Views

Kafka

Apache Kafka is a high performing message middleware that allows the implementation of real-time, batch, and stream type of message processing. Apache Kafka is used by many corporations for their web scale needs.

This article explains how to create Kafka clients using the **0.9.0** version of the Kafka API. We'll also show various ways Kafka clients can be created for *at-most-once*, *at-least-once*, and *exactly-once* message processing needs. This article also shows how to use the Avro client.

First, setup Apache Kafka middleware on your local machine. Instructions for setting up Kafka are available at the Kafka web site at http://kafka.apache.org/documentation.html#quickstart.

The article assumes that a locally installed single node Kafka instance is running on your local machine. Zookeeper is running at default port **2181**, and the Kafka node is running at default port **9092.** Once Kafka is up and running, then open up a command prompt and create a topic named **normal-topic** with two partitions. Following is the command to create the topic, execute the command from the Kafka installation folder.

```
bin/kafka-topics --zookeeper localhost:2181 --create --topic normal-topic
--partitions 2 --replication-factor 1
```

To check the status of the created topic, execute the following command from the Kafka installation folder:

```
bin/kafka-topics --list --topic normal-topic --zookeeper localhost:2181
```

If the topic needs to be altered to increase the partition, execute the following command from the Kafka installation folder:

```
bin/kafka-topics.sh --alter --topic normal-topic --zookeeper localhost:2181
--partitions 2
```

Producer

The code below shows how to implement a Kafka producer client to send test messages and adjust the for loop to control the number of messages that needs to be send.

```
public class ProducerExample {
    public static void main(String[] str) throws InterruptedException,
IOException {
            System.out.println("Starting ProducerExample ...");
            sendMessages();
    }
    private static void sendMessages() throws InterruptedException,
IOException {
            Producer<String, String> producer = createProducer();
            sendMessages(producer);
            // Allow the producer to complete sending of the messages before
program exit.
            Thread.sleep(20);
    }
    private static Producer<String, String> createProducer() {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("acks", "all");
        props.put("retries", 0);
        // Controls how much bytes sender would wait to batch up before
publishing to Kafka.
        props.put("batch.size", 10);
        props.put("linger.ms", 1);
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        return new KafkaProducer(props);
    }
    private static void sendMessages(Producer<String, String> producer) {
        String topic = "normal-topic";
        int partition = 0;
        long record = 1;
        for (int i = 1; i <= 10; i++) {
            producer.send(
                new ProducerRecord<String, String>(topic, partition,
Long.toString(record),Long.toString(record++)));
        }
    }
}
```

Various Ways a Consumer Can Register With Kafka

Before explaining how to register *at-most-once, a-least-once,* or *exactly-once* consumers, let us look at the two ways a consumer can register with a Kafka broker.

1. Registration using the ***subscribe*** method call. When a consumer registers with Kafka with a *subscribe* method call, Kafka rebalances the available consumers when a topic or partition gets added/deleted, or when a consumer gets added or deleted. This registration further offers two variants: **(a)** Along with the registration call, consumer can also provide a listener as a second parameter to the *subscribe* method call. When a consumer is registered this way,

Kafka notifies the listener whenever a rebalance occurs. The listener could provide the consumer an opportunity to manually manage the **offset,** which is very useful for an *exactly-once* type consumer. **(b)** Subscription of a consumer that does not provide the second optional listener parameter.

2. Registration of consumer to Kafka with an ***assign*** method call. When a consumer is registered with an ***assign*** method call, Kafka does not offer an automatic re-balance of the consumers.

Either of the above registration options **(1 or 2)** can be used by *at-most-once, a-least-once* or *exactly-once* consumers.

In the consumer examples explained below, the *at-most-once* and *at-least-once* consumer registers to Kafka by using option **(1, b).** The *exactly-once* consumer shows two examples, the first example registers to Kafka by using option **(1, a)**, and the second example registers to Kafka by using option **(2)**.

At-most-once Kafka Consumer (Zero or More Deliveries)

At-most-once consumer is the default behavior of a KAFKA consumer.

To configure this type of consumer:

1. Set '**enable.auto.commit'** to **true.**
2. Set **'auto.commit.interval.ms'** to a lower timeframe.
3. And do not make call to **consumer.commitSync();** from the consumer. With this configuration of consumer, Kafka would auto commit offset at the specified interval.

When a consumer is configured this way, there is a chance that consumer could exhibit *at-most-once* or *at-least-once* behavior. Since *at-most-once* is the lower messaging guarantee, let us declare this consumer as *at-most-once*. Below are the explanations of such consumer behaviors.

An *at-most-once* scenario happens when the commit interval has occurred, and that in turn triggers Kafka to automatically commit the last used offset. Meanwhile, let us say the consumer did not get a chance to complete the processing of the messages and consumer has crashed. Now when consumer restarts, it starts to receive messages from the last committed offset, in essence consumer could lose a few messages in between.

At-least-once scenario happens when consumer processes a message and commits the message into its persistent store and consumer crashes at that point. Meanwhile, let us say Kafka could not get a chance to commit the offset to the broker since commit interval has **not** passed. Now when the consumer restarts, it gets delivered with a few older messages from the last committed offset.

Code for this configuration is shown below:

```
public class AtMostOnceConsumer {
        public static void main(String[] str) throws InterruptedException {
            System.out.println("Starting  AtMostOnceConsumer ...");
            execute();
        }
        private static void execute() throws InterruptedException {
                KafkaConsumer<String, String> consumer = createConsumer();
                // Subscribe to all partition in that topic. 'assign' could
be used here
                // instead of 'subscribe' to subscribe to specific partition.
                consumer.subscribe(Arrays.asList("normal-topic"));
                processRecords(consumer);
        }
        private static KafkaConsumer<String, String> createConsumer() {
                Properties props = new Properties();
                props.put("bootstrap.servers", "localhost:9092");
                String consumeGroup = "cg1";
                props.put("group.id", consumeGroup);
                // Set this property, if auto commit should happen.
                props.put("enable.auto.commit", "true");
                // Auto commit interval, kafka would commit offset at this
interval.
                props.put("auto.commit.interval.ms", "101");
                // This is how to control number of records being read in
each poll
                props.put("max.partition.fetch.bytes", "135");
                // Set this if you want to always read from beginning.
                // props.put("auto.offset.reset", "earliest");
                props.put("heartbeat.interval.ms", "3000");
                props.put("session.timeout.ms", "6001");
                props.put("key.deserializer",

"org.apache.kafka.common.serialization.StringDeserializer");
                props.put("value.deserializer",

"org.apache.kafka.common.serialization.StringDeserializer");
                return new KafkaConsumer<String, String>(props);
        }
        private static void processRecords(KafkaConsumer<String, String>
consumer)  {
                while (true) {
                        ConsumerRecords<String, String> records =
consumer.poll(100);
                        long lastOffset = 0;
                        for (ConsumerRecord<String, String> record : records)
{
                                System.out.printf("\n\roffset = %d, key = %s,
value = %s", record.offset(),
record.key(), record.value());
                                lastOffset = record.offset();
                         }
                System.out.println("lastOffset read: " + lastOffset);
                process();
                }
        }
        private static void process() throws InterruptedException {
                // create some delay to simulate processing of the message.
                Thread.sleep(20);
        }
}
```

At-least-once Kafka Consumer (One or More Message Deliveries, Duplicate Possible)

To configure this type of consumer:

1. Set '**enable.auto.commit'** to **false or**
2. Set '**enable.auto.commit'** to **true** with **'auto.commit.interval.ms'** to a higher number.
3. Consumer should now then take control of the message offset commits to Kafka by making the following call **consumer.commitSync()**;

The consumer should make this commit call after it has processed the entire messages from the last *poll*. For this type of consumer, try to implement **'idempotent'** behavior within consumer to avoid reprocessing of the duplicate messages. Duplicate message delivery could happen in the following scenario. Consumer has processed the messages and committed the messages to its local store, but consumer crashes and did not get a chance to commit offset to Kafka before it has crashed. When consumer restarts, Kafka would deliver messages from the last offset.

Code for this configuration is shown below:

```
public class AtLeastOnceConsumer {
    public static void main(String[] str) throws InterruptedException {
            System.out.println("Starting
AutoOffsetGuranteedAtLeastOnceConsumer ...");
            execute();
     }
    private static void execute() throws InterruptedException {
            KafkaConsumer<String, String> consumer = createConsumer();
            // Subscribe to all partition in that topic. 'assign' could be
used here
            // instead of 'subscribe' to subscribe to specific partition.
            consumer.subscribe(Arrays.asList("normal-topic"));
            processRecords(consumer);
     }
     private static KafkaConsumer<String, String> createConsumer() {
            Properties props = new Properties();
            props.put("bootstrap.servers", "localhost:9092");
            String consumeGroup = "cg1";
            props.put("group.id", consumeGroup);
            // Set this property, if auto commit should happen.
            props.put("enable.auto.commit", "true");
            // Make Auto commit interval to a big number so that auto commit
does not happen,
            // we are going to control the offset commit via
consumer.commitSync(); after processing             // message.
            props.put("auto.commit.interval.ms", "999999999999");
            // This is how to control number of messages being read in each
poll
            props.put("max.partition.fetch.bytes", "135");
            props.put("heartbeat.interval.ms", "3000");
            props.put("session.timeout.ms", "6001");
            props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer","org.apache.kafka.common.serialization.StringD
eserializer");
            return new KafkaConsumer<String, String>(props);
    }
     private static void processRecords(KafkaConsumer<String, String>
consumer) throws {
            while (true) {
                    ConsumerRecords<String, String> records =
consumer.poll(100);
                    long lastOffset = 0;
                    for (ConsumerRecord<String, String> record : records) {
                        System.out.printf("\n\roffset = %d, key = %s, value =
%s", record.offset(),                                         record.key(),
record.value());
                        lastOffset = record.offset();
                    }
                    System.out.println("lastOffset read: " + lastOffset);
                    process();
                    // Below call is important to control the offset commit.
Do this call after you
                    // finish processing the business process.
                    consumer.commitSync();
            }
    }
    private static void process() throws InterruptedException {
        // create some delay to simulate processing of the record.
        Thread.sleep(20);
    }
}
```

Exactly-once Kafka Dynamic Consumer via Subscribe (One and Only One Message Delivery)

This example below demonstrates an *exactly-once* scenario. In this example, a consumer registers with Kafka via a ***'subscribe'*(1,a)** registration method call.

Let us now look at how to achieve an *exactly-once* scenario, in this case offset should be manually managed. Following are the steps to setup *exactly-once* scenario:

1. Set **enable.auto.commit = false.**
2. **Do not** make call to **consumer.commitSync();** after processing message.
3. Register consumer to a topic by making a '**subscribe**' call. *Subscribe* call behavior is explained earlier in the article.
4. Implement a **ConsumerRebalanceListener** and within the listener perform **consumer.seek(topicPartition,offset);** to start reading from a specific offset of that topic/partition.
5. While processing the messages, get hold of the offset of each message. Store the processed message's offset in an atomic way along with the processed message using atomic-transaction. When data is stored in relational database atomicity is easier to implement. For non-relational data-store such as HDFS store or No-SQL store one way to achieve atomicity is as follows: Store the offset along with the message.
6. Implement idempotent as a safety net.

Code for this configuration is shown below:

```
public class ExactlyOnceDynamicConsumer {
       private static OffsetManager offsetManager = new
OffsetManager("storage2");
        public static void main(String[] str) throws InterruptedException {
                System.out.println("Starting
ExactlyOnceDynamicConsumer ...");
                readMessages();
        }
        private static void readMessages() throws InterruptedException {
                KafkaConsumer<String, String> consumer = createConsumer();
                // Manually controlling offset but register consumer to
topics to get dynamically
                //  assigned partitions. Inside MyConsumerRebalancerListener
use
                // consumer.seek(topicPartition,offset) to control offset
which messages to be read.
                consumer.subscribe(Arrays.asList("normal-topic"),
                                new MyConsumerRebalancerListener(consumer));
                processRecords(consumer);
        }
        private static KafkaConsumer<String, String> createConsumer() {
                Properties props = new Properties();
                props.put("bootstrap.servers", "localhost:9092");
                String consumeGroup = "cg3";
                props.put("group.id", consumeGroup);
                // Below is a key setting to turn off the auto commit.
                props.put("enable.auto.commit", "false");
                props.put("heartbeat.interval.ms", "2000");
                props.put("session.timeout.ms", "6001");
                // Control maximum data on each poll, make sure this value is
bigger than the maximum                   // single message size
                props.put("max.partition.fetch.bytes", "140");
                props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
                props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
                return new KafkaConsumer<String, String>(props);
        }
        private static void processRecords(KafkaConsumer<String, String>
consumer) {
            while (true) {
                    ConsumerRecords<String, String> records =
consumer.poll(100);
                    for (ConsumerRecord<String, String> record : records) {
                            System.out.printf("offset = %d, key = %s, value =
%s\n", record.offset(),                                    record.key(),
record.value());
                            // Save processed offset in external storage.

offsetManager.saveOffsetInExternalStore(record.topic(), record.partition(),
record.offset());
                    }
               }
        }
}
public class MyConsumerRebalancerListener implements
org.apache.kafka.clients.consumer.ConsumerRebalanceListener {
        private OffsetManager offsetManager = new OffsetManager("storage2");
        private Consumer<String, String> consumer;
        public MyConsumerRebalancerListener(Consumer<String, String>
consumer) {
                this.consumer = consumer;
        }
        public void onPartitionsRevoked(Collection<TopicPartition>
partitions) {
                for (TopicPartition partition : partitions) {

offsetManager.saveOffsetInExternalStore(partition.topic(),
partition.partition(),                    consumer.position(partition));
                }
        }
        public void onPartitionsAssigned(Collection<TopicPartition>
partitions) {
```

```
                for (TopicPartition partition : partitions) {
                        consumer.seek(partition,
offsetManager.readOffsetFromExternalStore(partition.topic(),
partition.partition()));
                }
        }
}
/**
* The partition offset are stored in an external storage. In this case in a
local file system where
* program runs.
*/
public class OffsetManager {
        private String storagePrefix;
        public OffsetManager(String storagePrefix) {
                this.storagePrefix = storagePrefix;
        }
    /**
        * Overwrite the offset for the topic in an external storage.
        *
        * @param topic - Topic name.
        * @param partition - Partition of the topic.
        * @param offset - offset to be stored.
        */
        void saveOffsetInExternalStore(String topic, int partition, long
offset) {
            try {
                FileWriter writer = new FileWriter(storageName(topic,
partition), false);
                BufferedWriter bufferedWriter = new BufferedWriter(writer);
                bufferedWriter.write(offset + "");
                bufferedWriter.flush();
                bufferedWriter.close();
            } catch (Exception e) {
                    e.printStackTrace();
                    throw new RuntimeException(e);
            }
        }
        /**
            * @return he last offset + 1 for the provided topic and
partition.
        */
        long readOffsetFromExternalStore(String topic, int partition) {
                try {
                        Stream<String> stream =
Files.lines(Paths.get(storageName(topic, partition)));
                        return
Long.parseLong(stream.collect(Collectors.toList()).get(0)) + 1;
                } catch (Exception e) {
                    e.printStackTrace();
                }
                return 0;
        }
        private String storageName(String topic, int partition) {
            return storagePrefix + "-" + topic + "-" + partition;
        }
}
```

Exactly-once Kafka Static Consumer via Assign (One and Only One Message Delivery)

This example below demonstrates an *exactly-once* scenario. In this example, the consumer registers with Kafka via a ***‘assign*** **(2)** registration method call.

Let us now look at how to achieve exactly once scenario, in this case offset should be manually managed.

1. Set **enable.auto.commit = false**
2. **Don't** make call to **consumer.commitSync();** after processing message.
3. Register consumer to specific partition using '***assign***' call.
4. On start up of the consumer seek to specific message offset by calling consumer.seek(topicPartition,offset);
5. While processing the messages, get hold of the offset of each message. Store the processed message's offset in an atomic way along with the processed message using atomic-transaction. When data is stored in relational database atomicity is easier to implement. For non-relational data-store such as HDFS store or No-SQL store one way to achieve atomicity is as follows: Store the offset along with the message.
6. Implement idempotent as a safety net.

Code for this configuration is shown below:

```
public class ExactlyOnceStaticConsumer {
        private static OffsetManager offsetManager = new
OffsetManager("storage1");
        public static void main(String[] str) throws InterruptedException,
IOException {
                System.out.println("Starting ExactlyOnceStaticConsumer ...");
                readMessages();
        }
        private static void readMessages() throws InterruptedException,
IOException {
                KafkaConsumer<String, String> consumer = createConsumer();
                String topic = "normal-topic";
                int partition = 1;
                TopicPartition topicPartition =
                                registerConsumerToSpecificPartition(consumer,
topic, partition);
                // Read the offset for the topic and partition from external
storage.
                long offset =
offsetManager.readOffsetFromExternalStore(topic, partition);
                // Use seek and go to exact offset for that topic and
partition.
                consumer.seek(topicPartition, offset);
                processRecords(consumer);
        }
        private static KafkaConsumer<String, String> createConsumer() {
                Properties props = new Properties();
                props.put("bootstrap.servers", "localhost:9092");
                String consumeGroup = "cg2";
                props.put("group.id", consumeGroup);
                // Below is a key setting to turn off the auto commit.
                props.put("enable.auto.commit", "false");
                props.put("heartbeat.interval.ms", "2000");
                props.put("session.timeout.ms", "6001");
                // control maximum data on each poll, make sure this value is
bigger than the maximum                 // single message size
                props.put("max.partition.fetch.bytes", "140");
                props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
                props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
                return new KafkaConsumer<String, String>(props);
        }
        /**
            * Manually listens for specific topic partition. But, if you are
looking for example of how to                * dynamically listens to
partition and want to manually control offset then see
            * ExactlyOnceDynamicConsumer.java
            */
         private static TopicPartition registerConsumerToSpecificPartition(
                    KafkaConsumer<String, String> consumer, String topic, int
partition) {
                    TopicPartition topicPartition = new TopicPartition(topic,
partition);
                    List<TopicPartition> partitions =
Arrays.asList(topicPartition);
                    consumer.assign(partitions);
                    return topicPartition;
          }
            /**
                * Process data and store offset in external store. Best
practice is to do these operations
                * atomically.
                */
            private static void processRecords(KafkaConsumer<String, String>
consumer) throws {
                    while (true) {
                           ConsumerRecords<String, String> records =
consumer.poll(100);
                            for (ConsumerRecord<String, String> record :
records) {
                                    System.out.printf("offset = %d, key = %s,
value = %s\n", record.offset(),
record.key(), record.value());

offsetManager.saveOffsetInExternalStore(record.topic(), record.partition(),
record.offset());
                            }
                    }
            }
}
```

Avro Producer and Consumer

Avro is an open source binary message exchange protocol. Avro helps to send optimized messages across the wire hence reducing the network overhead. Avro can enforce schema for messages that can be defined using JSON. Avro can generate binding objects in various programming languages from these schemas. Message payloads are automatically bound, and these generate objects on the consumer side. Avro is natively supported and highly recommended to use along with Kafka. Here is a simple Avro consumer and Producer. Supporting classes for marshalling and unmarshalling messages are available in GitHub.

```
public class AvroConsumerExample {
        public static void main(String[] str) throws InterruptedException {
                System.out.println("Starting
AutoOffsetAvroConsumerExample ...");
                readMessages();
        }
        private static void readMessages() throws InterruptedException {
                KafkaConsumer<String, byte[]> consumer = createConsumer();
                // Assign to specific topic and partition.
                consumer.assign(Arrays.asList(new TopicPartition("avro-
topic", 0)));
                processRecords(consumer);
          }
          private static void processRecords(KafkaConsumer<String, byte[]>
consumer) throws {
                while (true) {
                        ConsumerRecords<String, byte[]> records =
consumer.poll(100);
                        long lastOffset = 0;
                        for (ConsumerRecord<String, byte[]> record : records)
{
                                GenericRecord genericRecord
=   AvroSupport.byteArrayToData(AvroSupport.getSchema(),
record.value());
                                String firstName =
AvroSupport.getValue(genericRecord,
"firstName", String.class);
                                System.out.printf("\n\roffset = %d, key = %s,
value = %s", record.offset(),
record.key(), firstName);
                                lastOffset = record.offset();
                        }
                    System.out.println("lastOffset read: " + lastOffset);
                    consumer.commitSync();
                }
            }
            private static KafkaConsumer<String, byte[]> createConsumer() {
                        Properties props = new Properties();
                        props.put("bootstrap.servers", "localhost:9092");
                        String consumeGroup = "cg1";
                        props.put("group.id", consumeGroup);
                        props.put("enable.auto.commit", "true");
                        props.put("auto.offset.reset", "earliest");
                        props.put("auto.commit.interval.ms", "100");
                        props.put("heartbeat.interval.ms", "3000");
                        props.put("session.timeout.ms", "30000");
                        props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
                        props.put("value.deserializer",
"org.apache.kafka.common.serialization.ByteArrayDeserializer");
                    return new KafkaConsumer<String, byte[]>(props);
            }
}
public class AvroProducerExample {
        public static void main(String[] str) throws InterruptedException,
IOException {
                System.out.println("Starting ProducerAvroExample ...");
                sendMessages();
        }
        private static void sendMessages() throws InterruptedException,
IOException {
                Producer<String, byte[]> producer = createProducer();
                sendRecords(producer);
        }
        private static Producer<String, byte[]> createProducer() {
                    Properties props = new Properties();
                    props.put("bootstrap.servers", "localhost:9092");
                    props.put("acks", "all");
                    props.put("retries", 0);
                    props.put("batch.size", 16384);
                    props.put("linger.ms", 1);
                    props.put("buffer.memory", 33554432);
                    props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
                    props.put("value.serializer",
"org.apache.kafka.common.serialization.ByteArraySerializer");
                return new KafkaProducer(props);
        }
        private static void sendRecords(Producer<String, byte[]> producer)
throws IOException, {
                String topic = "avro-topic";
                int partition = 0;
                while (true) {
                        for (int i = 1; i < 100; i++)
                            producer.send(new ProducerRecord<String,
byte[]>(topic, partition,
Integer.toString(0), record(i + "")));
                }
         }
         private static byte[] record(String name) throws IOException {
                    GenericRecord record = new
GenericData.Record(AvroSupport.getSchema());
                    record.put("firstName", name);
                    return
AvroSupport.dataToByteArray(AvroSupport.getSchema(), record);
          }
}
```

Source Code

The complete source code can be found here at GitHub: https://github.com/ajmalbabu/kafka-clients