Kafka Producer API

Kafka Producer API is primarily made available in Java.

Detailed Java documentation is available at following URL.
https://kafka.apache.org/21/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html

Kafka client API is a Java API. However, these API are also available in other languages. Some of the resources are listed below.

1. Kafka Client for .NET
   - https://github.com/confluentinc/confluent-kafka-dotnet
2. Kafka Client for Go
   - https://github.com/confluentinc/confluent-kafka-go
3. Kafka Client for Python
   - https://github.com/confluentinc/confluent-kafka-python

There are many more language bindings that are implemented by the open source community, and we can find details in Kafka documentation. Except for Java API, all the above language APIs are based on C/C++ Kafka client library named **librdkafka**. We can find documentation and an exhaustive list of supported languages at following URL.

https://github.com/edenhill/librdkafka

Kafka producer API is used to create a message record and send it to Kafka broker.

Problem-Hello Producer:

We want to create a simplest possible producer code that sends million messages to a Kafka topic.

Solution-Hello Producer:

Sending a message record to Kafka is a four-step process as shown in the below Let's try to understand them in detail.

Step-1: add the kafka clients dependencies in the pom.xml file as shown below:

```xml
<properties>
    <kafkaVersion>3.4.0</kafkaVersion>
    <lombokVersion>1.18.26</lombokVersion>
</properties>

<dependencies>
    <!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients -->
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>${kafkaVersion}</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>${lombokVersion}</version>
        <scope>provided</scope>
    </dependency>
```


Step-2: Create a AppConfig class as shown below:

```java
public class AppConfig {

    public final static String APPLICATION_ID = "POS-MACHINE";
    public final static String BOOTSTRAP_SERVERS = "localhost:9092,localhost:9093,localhost:9094";
    public final static String TOPIC_NAME = "stock-ticks";
    public static final int NUM_EVENTS = 1000000;
}
```

Step-3: (Line 6 to 10) Create a Java Properties object and put the necessary configurations. Kafka producer API is highly configurable, and we customize the behaviour by setting different producer configurations.

```java
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.IntegerSerializer;
import org.apache.kafka.common.serialization.StringSerializer;

public class PosMachine {

    public static void main(String[] args) {
        System.out.println("Creating Kafka Producer Configurations");
        System.out.println("======================================");

        /**
         * Step-1: Create Producer Configurations
         */
        Properties props = new Properties();
        props.put(ProducerConfig.CLIENT_ID_CONFIG, AppConfig.APPLICATION_ID);
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, AppConfig.BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        /**
         * Step-2: Create the Kafka Producer
         */
        KafkaProducer<Integer, String> posMachine = new KafkaProducer<Integer, String>(props);

        /**
         * Step-3: Create Producer Record and Send to the Kafka Brokers
         */
        for(int i=1; i<=AppConfig.NUM_EVENTS;i++) {
            String message = "Invoice Copy-"+i;
            ProducerRecord<Integer, String> invoice = new ProducerRecord<Integer, String>(AppConfig.TOPIC_NAME, i, message);
            posMachine.send(invoice);
        }

        /**
         * Step-4: Close the Producer Session
         */
        posMachine.close();
    }

}
```

In the above example, we are setting four basic configuration properties.

1. ***CLIENT_ID_CONFIG:*** which is a simple string pass to the Kafka server. Purpose of the client id is to track the source of the message.

2. ***BOOTSTRAP_SERVER_CONFIG:*** this is a comma separated list of host, port pairs. The producer will use this information for establishing the initial connection to the Kafka cluster. If we are running on a single node kafka cluster, we can supply an individual host, port information. The bootstrap

configurations used only for the initial connections. Once connected the kafka producer will automatically query for the metadata and discovers the full list of kafka brokers in the cluster.

That means we do not need to supply a complete list of kafka brokers as a boot strap configuration. However it is recommended to provide 2 to 3 broker address of a multi node cluster. Doing so will help the producer to check for the 2nd or 3rd broker in case the first broker in the list is down.

3. KEY_SERIALIZER_CLASS_CONFIG

4. VALUE_SERIALIZER_CLASS_CONFIG

A Kafka message should be structured as a key/value pair. That means each message that we want to send to the Kafka server should have a key and a value. We can have a null key, but the message is still structured as a key/value pair. Kafka messages are sent over the network. So, the key and the value must be serialized into bytes before they are streamed over the network. Kafka producer API comes with a bunch of ready to use serializer classes. The above example is setting an IntegerSerializer for the key and a StringSerializer for the message value.

Step-4:

Create an instance of KafkaProducer<K, V> class by passing the properties object that we created in step-3. The type parameter K is the type of the key, and the V is the type of value. Once created, we can use this producer instance to send messages to the Kafka cluster.

```
Properties props = new Properties();
props.put(ProducerConfig.CLIENT_ID_CONFIG,AppConfigs.applicationID);
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,AppConfigs.bootstrapServers);
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

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

Step-5:

The next step is, we are calling the send() API on the producer instance. The send method takes a ProducerRecord object and sends it to the Kafka cluster. Many things happen inside the producer object, but at the simplest level, the producer finally transmits the ProducerRecord to the Kafka broker. The ProducerRecord constructor takes three arguments. These arguments are the topic name, the message key, and the message itself.

```
for(int i = 0; i< AppConfigs.numEvents; i++){
    producer.send(new ProducerRecord<>(AppConfigs.topicName,i, value: "Simple Message-"+i));
}
```

Step-6:

The last and final step is to close the producer instance. The producer functionality is involved, and it does a lot of things internally. We will cover producer internals as we progress with this chapter. However, it is essential to understand that the producer consists of some buffer space and background I/O thread. If we do not close the producer after sending all the required messages, we will leak the resources created by the producer.

Kafaka Producer API Internal Working

We Create an KafkaProducer<K, V> API by passing some essential configurations.

```
Properties props = new Properties();
props.put(ProducerConfig.CLIENT_ID_CONFIG, AppConfigs.applicationID);
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, AppConfigs.bootstrapServers);
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

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

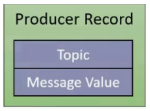
And start sending the messages using the send method. There is only one restriction that we must package the message in a Producer Record Object.

```
producer.send(new ProducerRecord<>(AppConfigs.topicName, i, value: "Simple Message-" + i));
```

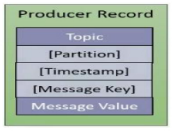
Now lets try to understand the detailed things happening behind the send method.

We package the message content in the producer record object with at least two mandatory arguments:

Kafka Topic name and message value. Kafka topic name is the destination address of the message. The message value is the main content of the message.



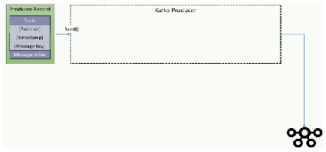
Other than these 2 mandatory values we can also supply the following optional items.



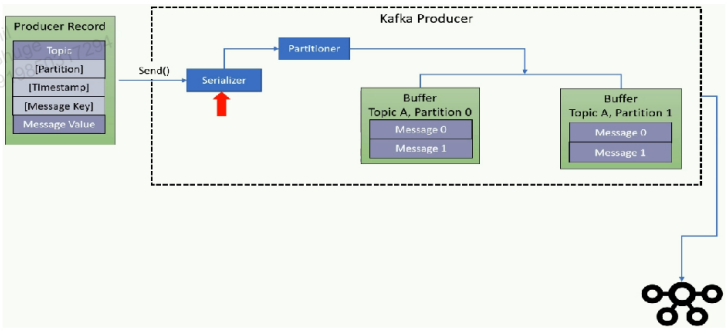
Here the message key is one of the most important argument and it is used for many purposes such as partitioning, grouping and joins.

The producer record wraps our message content, with all necessary information such as topic name, message key, and time stamp. Once created we can handover the Producer Record to the Kafak Producer using the send() method.

Apache Kafka



When Kafka producer supposed to transmit the producer record to the kafka broker over the network, it doesn't immediately transfer the record. Every Record goes through Serialization, Partitioner and then it is stored in the buffer.



Producer Serializer

The serialization is required to send the data over the network. Without serializing the data we cant transmit to our remote location.however Kafka doesn't know how to serializer our key and value. that's why we are specifying the key serializer and value serializer class we supplied as part of the producer configuration before we are calling the send method.

```
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
```

In the above example, we have used the IntegerSerializer and StringSerializer. However these Serializers are elementary Serializers and these do not cover the most of the use cases.

In Realtime scenario, events are represented by complex java objects and these objects must be serialized before the producer can transmit them to the broker. The string serializer may not be do a good help for those scenarios hence kafka is giving an option to use a generic serialization library like AVRO or thrift. Alternatively we can also create our own custom Serializer.

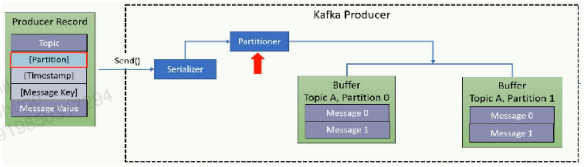
Apache Kafka

Producer Partitioner

Every Producer Record includes a mandatory topic name as the destination address of the data. However the kafka topics are partitioned and hence the producer should also decide on which partition the message should sent.

There are two approaches to specify the target partition number for the messages.

1. Set partition number argument in the producer record



2. Supply a partition class to determine the partition number at run time.

```
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, MyPartitioner.class.getName());
```

The best practice to supply the partition class to implement the desired partitioning strategy to assign the partition number at run time to each partition. We can specify the custom Partitioner object using the properties object.

Creating custom Partitioner often is a not needed, because the kafka producer comes with the default Partitioner which is the most commonly used Partitioner.

The default Partitioner takes one of the below two approaches, to determine the destination topic.

1. Hash key Partitioning:

This is based on the message key. When the message key exists, the default Partitioner will uses the hashing algorithm on the key to determine the partition number for the messages. It is as simple as hashing the key to convert it into a numeric value. Then use that hash number to deduce the partition number

The hashing ensures that all the messages with the same key go to the same partition .however this algorithm also takes the total number of partitions as one of the input. So if we increase the number of

partitions in the topic, the default Partitioner is start giving a different output. That means if the partition is based on the key, then we should create a topic with the enough partitions and never increase it at the later stage.

We can easily over provision the number of partitions in the topic. Meaning if we need 100 partitions we can easily over provision it by 25% and create 125 partitions.

I. e. **100 partitions+25% of partitions= 125 Partitions.**

If we do increase the number of partitions later, we may have to re-distribute the existing messages.

2. **Round robin Partitioning**

When the message key is null, the default Partitioner will uses the round robin partitioning algorithm to achieve an equal distribution among the available partitions.

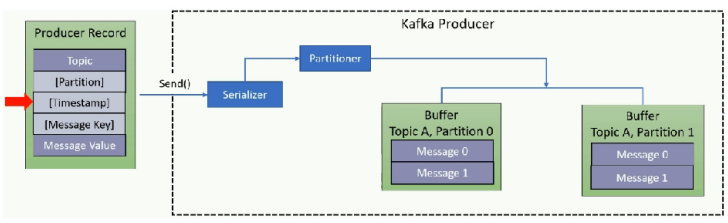
Here the math principle followed by the Round Robin Algorithm is:
totalNoOfMessages/totalnoOfAvailablePartitions;

Example: 100/10-->each partion will get exactly 10 messages this is called as equal distribution.

That means if the first message goes to partition-1, then the second one goes to partition-2 like this Partitioner repeats in the loop. The default Partitioner is the most commonly used Partitioner in most of the use cases. However kafka allows us to use the partitioning strategy by implementing a custom Partitioner class. But we may not even need the custom partitioning class in most of the cases

Message Timestamp:

The producer records takes an optional time stamp field. the message time stamp is optional. However for a realtime streaming applications, the time stamp is the most critical value. For that reason every ***kafka message is automatically attached with a time stamp even if we do not explicitly specify it***.



Kafka allows us to implement the one of the following two time message stamping mechanism:

1. Create Time

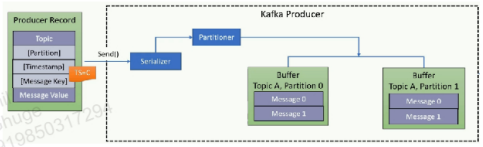
2. Log Append Time

1. Create time:

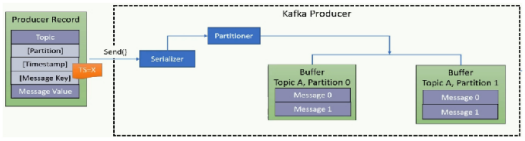
The **CreateTime** is the time when the message was produced. The **LogAppendTime** is the time when the message was received at the Kafka broker. However, we cannot use both. Our application must decide between these two time stamping methods while creating the topic.

Once determined, we must set the ***message.timestamp.type*** topic configuration to 0 for using CreateTime, or we can set it to 1 for using ***LogAppendTime.*** The default value is zero (CreateTime).

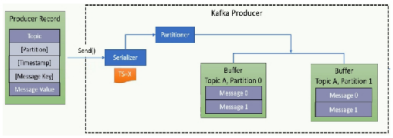
The producer API automatically sets the current producer time to the ProducerRecord#timestamp field.

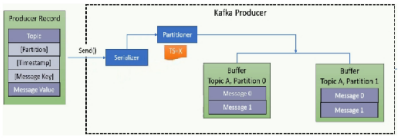


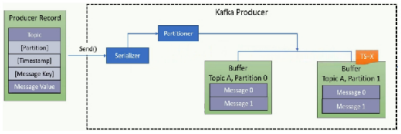
However,we can override the auto time stamping by explicitly specifying this argument.

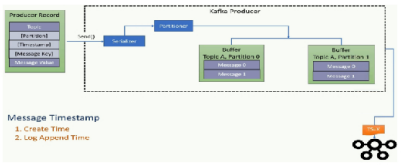


So, the message is transmitted with a producer time either automatically set by the producer, or explicitly set by the programmer.



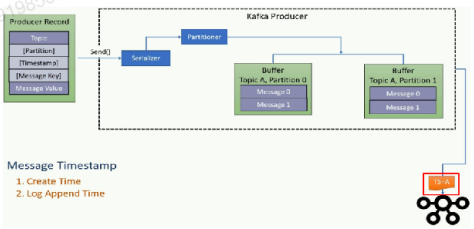






LogAppendTime:

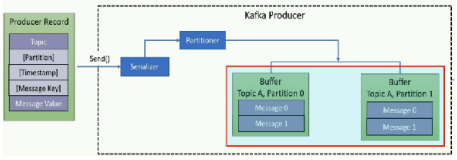
When using LogAppendTime configuration, the broker will override the producer timestamp with its current local time before appending the message to the log. In this case, the producer time is overwritten by the broker time. However, the message will always have a timestamp, either a producer time or the broker time.



Note: It is recommended to use the default message time stamp as Create Stamp. Because the producer API automatically assigns a time stamp

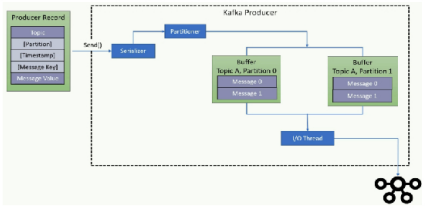
Producer Message Buffer

Once serialized and assigned a target partition number, the message goes to sit in the buffer waiting to be transmitted. The producer object consists of a partition-wise buffer space that holds the records that haven't yet been sent to the server.



The producer also runs a background I/O thread that is responsible for turning these records into requests and transferring them to the cluster.

Apache Kafka



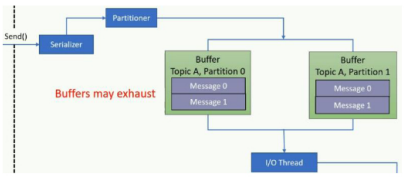
The buffering of the message is designed to offer the following advantages.

1. Asynchronous send API
2. Network Roundtrip Optimization

Buffering arrangement makes the send method asynchronous. That means The send method will add the messages to the buffer and return without blocking. Those records are then transmitted by the background I/O thread. This arrangement is quite convincing as our send() method is not delayed for the network operation.

Buffering also allows the background I/O thread to combine multiple messages from the same buffer and transmit them together as a single packet to achieve better throughput.

However, here is the critical consideration. if the records are posted faster than they can be transmitted to the server, then this buffer space will be exhausted, and our next send method will block for few milliseconds until the buffer is freed by the I/O thread.

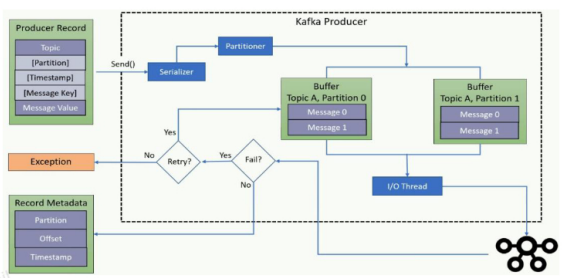


If the I/O thread takes too long to release the buffer, then our send method throws a Timeout Exception. When we are seeing such timeout exceptions, we may want to increase the producer memory. The default producer memory size is 32 MB. However we can expand the total memory allocated for the buffer by setting ***buffer.memory*** producer configuration.

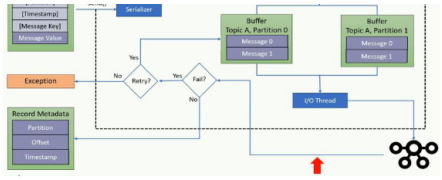
Producer I/O Threads and Retries

The producer background I/O thread is responsible for transmitting the serialized messages that are waiting in the topic partition buffer.

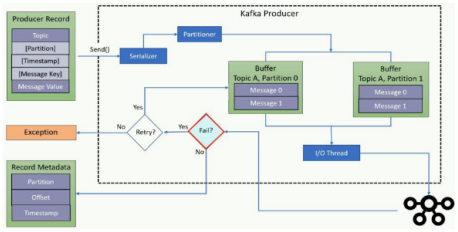
Apache Kafka



✓ When the broker receives the message, it sends back an acknowledgment. If the message is successfully written to Kafka, the broker will return a success acknowledgment.

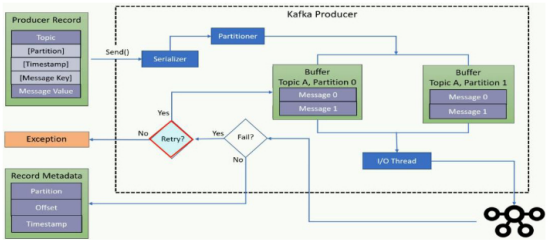


If the broker failed to write the message, it would return an error. When the background I/O thread receives an error or does not receive any acknowledgment,



it may retry sending the message a few more times before giving up and throwing back an error.

Apache Kafka



we can control the number of retries by setting the retries producer configuration. When all the retries are failed, the I/O thread will return the error back to the send() method.