KAFKA PRODUCER

Đơn vị: Công ty CP Giáo dục và Công nghệ QNET

QNET JOINT STOCK COMPANY

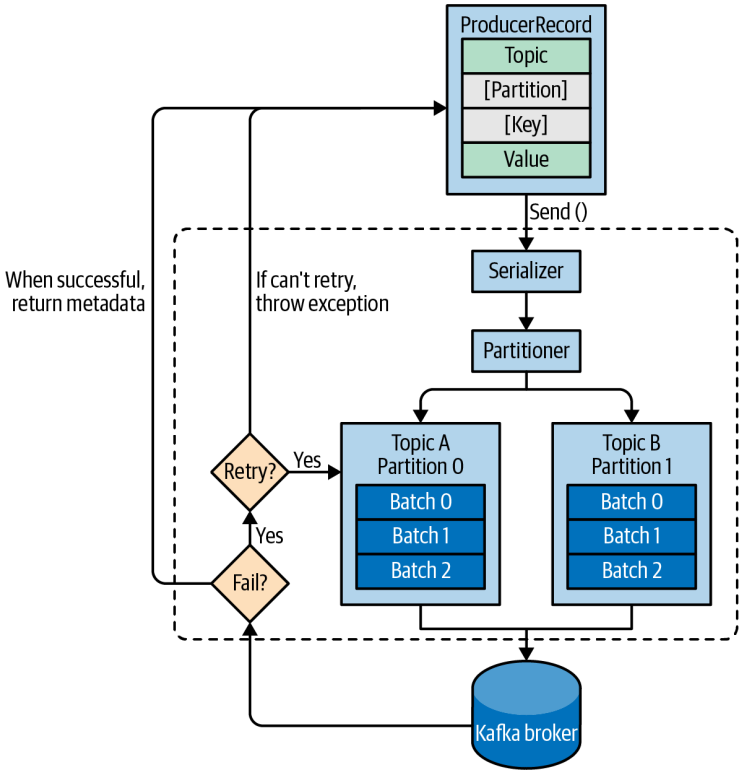
Address: 14th Floor, VTC Online Tower
18 Tam Trinh Street. Hoang Mai District
Hanoi, Vietnam

LEARNING OBJECTIVE

- Overview of Producer and Architecture
- Producer Configuration
- Send messages
- Serializers
- Partitions

Kafka Producer

Producer Overview



Kafka Producer

Constructing a Kafka Producer

- **Bootstrap.servers**: *List of* `host:port` *pairs of brokers that the producer will use to establish initial connection to the Kafka cluster.*

- **Key.serializer**: *Name of a class that will be used to serialize the keys of the records we will pro- duce to Kafka*
- **Value.serializer**: *Name of a class that will be used to serialize the values of the records we will pro- duce to Kafka.*
- **Value.serializer**: Name of a class that will be used to serialize the values of the records we will pro- duce to Kafka. The same way you set `key.serializer` to a name of a class that will serialize the message key object to a byte array, you set `value.serializer` to a class that will serialize the message value object.

Kafka Producer

Constructing a Kafka Producer

```
Properties kafkaProps = new Properties(); ❶
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer"); ❷
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

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

Kafka Producer

Primary methods of sending messages

- Fire-and-forget: We send a message to the server and don't really care if it arrives successfully or not.
- Synchronous send: Technically, Kafka producer is always asynchronous—we send a message and the `send()` method returns a `Future` object. However, we use `get()` to wait on the `Future` and see if the `send()` was successful or not before sending the next record.
- Asynchronous send: We call the `send()` method with a callback function, which gets triggered when it receives a response from the Kafka broker.

Kafka Producer

Fire and forget

Fire-and-forget: We send a message to the server and don't really care if it arrives successfully or not.

Usecase:

- Click impression data
- Monitoring events data
- Not fit for production application that requires avoiding data loss

Kafka Producer

Fire and forget: Sending a Message to Kafka

```
ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision
Products",
            "France");
    try {
        producer.send(record);
    } catch (Exception e) {
        e.printStackTrace();
    }
```

1. The producer accepts *ProducerRecord* objects
2. The producer use object *send() to* to send the ProducerRecord
3. While we ignore errors that may occur while sending messages to the Kafka broker or in the brokers themselves, we may still get an exeption if the producer encountered errors before sending the message to Kafka.

Kafka Producer

Synchronous Send

- Synchronous send:
  - Allow the producer to catch exceptions when Kafka responds to the producer request with an error, or when send retries were exhausted.
  - The main trade-off involved is performance. Depending on how busy the Kafka cluster is, broker can take anywhere from 2ms to a few seconds to respond to the producer requests. I
  - If you send messages synchronously, the sending thread will spend this time waiting and doing nothing else.
- Usecase: Usually not used in production (but are very common in code example)

Kafka Producer

Synchronous send

```
    ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision Products",
"France");
    try {
        producer.send(record).get();
    } catch (Exception e) {
        e.printStackTrace();
}
```

Kafka Producer

Asynchronous send

- Usecase:
  - Used when dealing with critical data
  - Customer orders
  - Banking Transactions

Kafka Producer

Sending a message Synchronously

```
 private class DemoProducerCallback implements Callback {
        @Override
        public void onCompletion(RecordMetadata recordMetadata, Exception e) {
            if (e != null) {
                e.printStackTrace();
            }
} }
    ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");
    producer.send(record, new DemoProducerCallback());
```

Kafka Producer

Common errors

Kafka has two type of errors:

- Retriable errors: are those that can be resolved by sending the message again (not leader for partition error…)

- Non-retriable errors: are those that can’t be resolved by retry-ing (message size too large)

Kafka Producer

Configuring Brokers

- client.id: `client.id` is a logical identifier for the client and the application it is used in. This can be any string and will be used by the brokers to identify messages sent from the client. It is used in logging and metrics and for quotas.

- acks: The `acks` parameter controls how many partition replicas must receive the record before the producer can consider the write successful.

- `acks=0` : The producer will not wait for a reply from the broker before assuming the mes- sage was sent successfully.
- `acks=1` : The producer will receive a success response from the broker the moment the leader replica receives the message.
- `acks=all` : The producer will receive a success response from the broker once all in sync replicas receive the message.

Kafka Producer

Configuring Brokers

max.block.ms

This parameter controls how long the producer may block when calling send() and when explicitly requesting metadata via partitionsFor(). Those methods may block when the producer's send buffer is full or when metadata is not available. When max.block.ms is reached, a timeout exception is thrown.

delivery.timeout.ms

This configuration will limit the amount of time spent from the point a record is ready for sending (send() returned successfully and the record is placed in a batch) until either the broker responds or the client gives up, including time spent on retries. As you can see in Figure 3-2, this time should be greater than linger.ms and request.timeout.ms. If you try to create a producer with an inconsistent timeout configuration, you will get an exception. Messages can be successfully sent much faster than delivery.timeout.ms, and typically will.

Kafka Producer

Configuring Brokers

request.timeout.ms

This parameter controls how long the producer will wait for a reply from the server when sending data. Note that this is the time spent waiting on each producer request before giving up; it does not include retries, time spent before sending, and so on. If the timeout is reached without reply, the producer will either retry sending or com- plete the callback with a TimeoutException.

retries and retry.backoff.ms

When the producer receives an error message from the server, the error could be transient (e.g., a lack of leader for a partition). In this case, the value of the retries parameter will control how many times the producer will retry sending the message before giving up and notifying the client of an issue. By default, the producer will wait 100 ms between retries, but you can contr

Kafka Producer

Configuring Brokers

linger.ms

linger.ms controls the amount of time to wait for additional messages before send- ing the current batch. KafkaProducer sends a batch of messages either when the cur- rent batch is full or when the linger.ms limit is reached. By default, the producer will send messages as soon as there is a sender thread available to send them, even if there's just one message in the batch. By setting linger.ms higher than 0, we instruct the producer to wait a few milliseconds to add additional messages to the batch before sending it to the brokers. This increases latency a little and significantly increases throughput—the overhead per message is much lower, and compression, if enabled, is much better.

buffer.memory

This config sets the amount of memory the producer will use to buffer messages wait- ing to be sent to brokers. If messages are sent by the application faster than they can be delivered to the server, the producer may run out of space, and additional send() calls will block for max.block.ms and wait for space to free up before throwing an exception. Note that unlike most producer exceptions, this timeout is thrown by send() and not by the resulting Future.

Kafka Producer

Configuring Brokers

compression.type

By default, messages are sent uncompressed. This parameter can be set to snappy, gzip, lz4, or zstd, in which case the corresponding compression algorithms will be used to compress the data before sending it to the brokers. Snappy compression was invented by Google to provide decent compression ratios with low CPU overhead and good performance, so it is recommended in cases where both performance and bandwidth are a concern. Gzip compression will typically use more CPU and time but results in better compression ratios, so it is recommended in cases where network bandwidth is more restricted. By enabling compression, you reduce network utiliza- tion and storage, which is often a bottleneck when sending messages to Kafka.

batch.size

When multiple records are sent to the same partition, the producer will batch them together. This parameter controls the amount of memory in bytes (not messages!) that will be used for each batch. When the batch is full, all the messages in the batch will be sent. However, this does not mean that the producer will wait for the batch to become full. The producer will send half-full batches and even batches with just a sin- gle message in them. Therefore, setting the batch size too large will not cause delays in sending messages; it will just use more memory for the batches

Kafka Producer

Configuring Brokers

max.request.size

This setting controls the size of a produce request sent by the producer. It caps both the size of the largest message that can be sent and the number of messages that the producer can send in one request. For example, with a default maximum request size of 1 MB, the largest message you can send is 1 MB, or the producer can batch 1,024 messages of size 1 KB each into one request. In addition, the broker has its own limit on the size of the largest message it will accept (message.max.bytes). It is usually a good idea to have these configurations match, so the producer will not attempt to send messages of a size that will be rejected by the broker.

receive.buffer.bytes and send.buffer.bytes

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to –1, the OS defaults will be used. It is a good idea to increase these when producers or consumers communicate with brokers in a different datacenter, because those network links typically have higher latency and lower bandwidth.

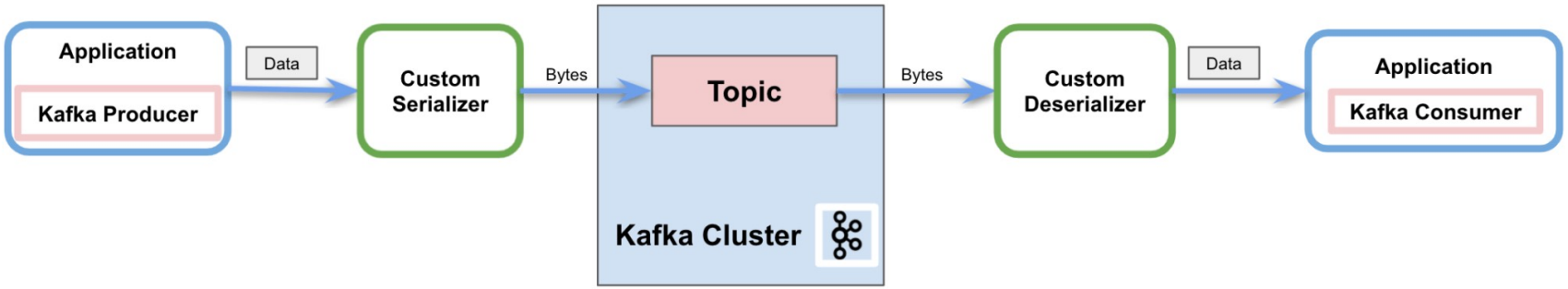
Kafka Producer

DEMO: Create a Kafka Producer

Kafka Producer

Serializers

- Serialization is the process of converting objects into bytes. Deserialization is the inverse process - converting a stream of bytes into an object. In a nutshell, it transforms the content into readable and interpretable information



Kafka Producer

Custom Secrializers

- StringSerializer
- ShortSerializer
- IntergerSerializer
- LongSerializer
- DoubleSerializer
- BytesSerializer

Kafka Producer

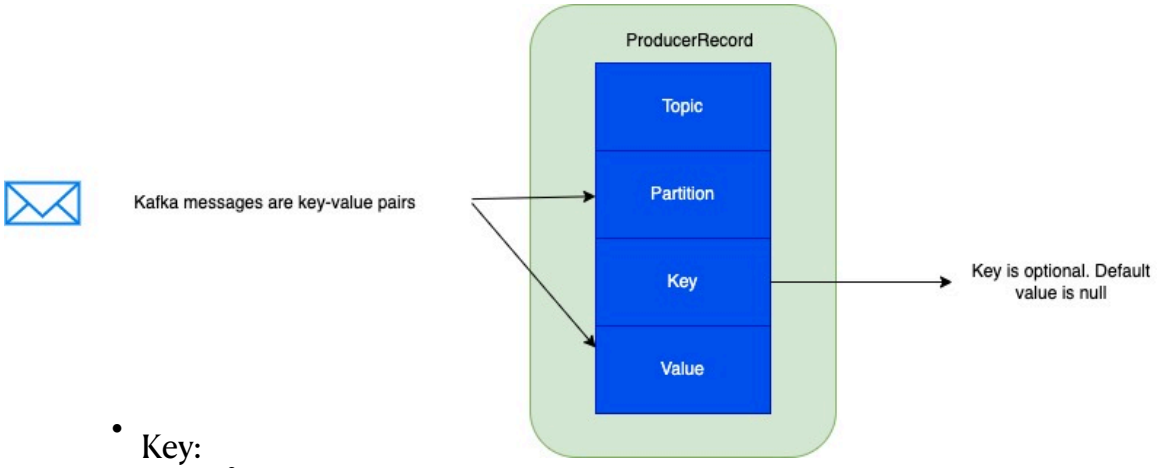
Demo: Creating a custom serializer

Kafka Producer

Serializers challenges and serializing using Apache Avro

Kafka Producer

Parititions



- Key:
  - Provides additional information
  - This information gets stored with the message
  - Which topic partition the message will be written in

Kafka Producer

Implementing a custom partitioning strategy

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;

public class BananaPartitioner implements Partitioner {

    public void configure(Map<String, ?> configs) {} ❶

    public int partition(String topic, Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes,
                         Cluster cluster) {
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if ((keyBytes == null) || (!(key instanceOf String))) ❷
            throw new InvalidRecordException("We expect all messages " +
                "to have customer name as key");

        if (((String) key).equals("Banana"))
            return numPartitions - 1; // Banana will always go to last partition

        // Other records will get hashed to the rest of the partitions
        return Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1);
    }

    public void close() {}
}
```

Kafka Producer

Headers

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

record.headers().add("privacy-level","YOLO".getBytes(StandardCharsets.UTF_8));
```

Kafka Producer

Demo: Setup custom partition

DISCUSSION

Quality Network for Education and Technology

XIN CHÂN THÀNH CẢM ƠN!