Chapter 3. Configuring a Producer and Consumer

In this chapter, we will cover the following topics:

- Configuring the basic settings for producer
- Configuring thread and performance for producer
- Configuring the basic settings for consumer
- Configuring the thread and performance for consumer
- Configuring the log settings for consumer
- Configuring the ZooKeeper settings for consumer
- Other configurations for consumer

Introduction

This chapter explains the configurations of Kafka consumer and producer and how to use them to our advantage. The default settings are good; but when you want to extract that extra mileage out of Kafka, these come in handy. Though these are explained with respect to command-line configurations, the same can be done in your favorite Kafka client library.

Linger.ms & batch.size

- **batch.size:** Maximum number of bytes that will be included in a batch. The default is 16KB.

- Increasing a batch size to something like 32KB or 64KB can help increasing the compression, throughput, and efficiency of requests
- Any message that is bigger than the batch size will not be batched
- A batch is allocated per partition, so make sure that you don't set it to a number that's too high, otherwise you'll run waste memory!
- (Note: You can monitor the average batch size metric using Kafka Producer Metrics)

Max.block.ms & buffer.memory

- If the producer produces faster than the broker can take, the records will be buffered in memory
- **buffer.memory=33554432 (32MB):** the size of the send buffer

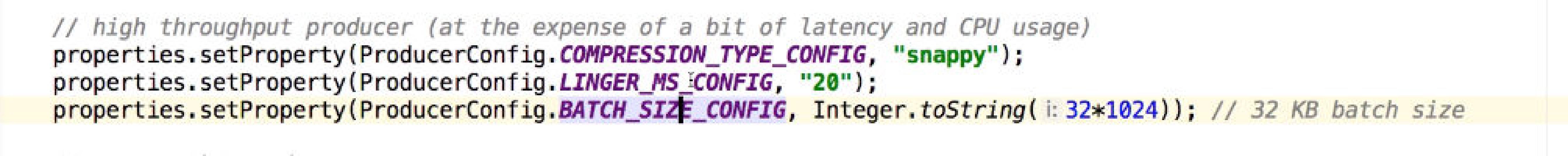
- That buffer will fill up over time and fill back down when the throughput to the broker increases

Max.block.ms & buffer.memory

- <u>If that buffer is full (all 32MB), then the .send() method will start to block (won't return right away)</u>

- **max.block.ms=60000**: the time the .send() will block until throwing an exception. Exceptions are basically thrown when
  - The producer has filled up its buffer
  - The broker is not accepting any new data
  - 60 seconds has elapsed.

- If you hit an exception hit that usually means your brokers are down or overloaded as they can't respond to requests

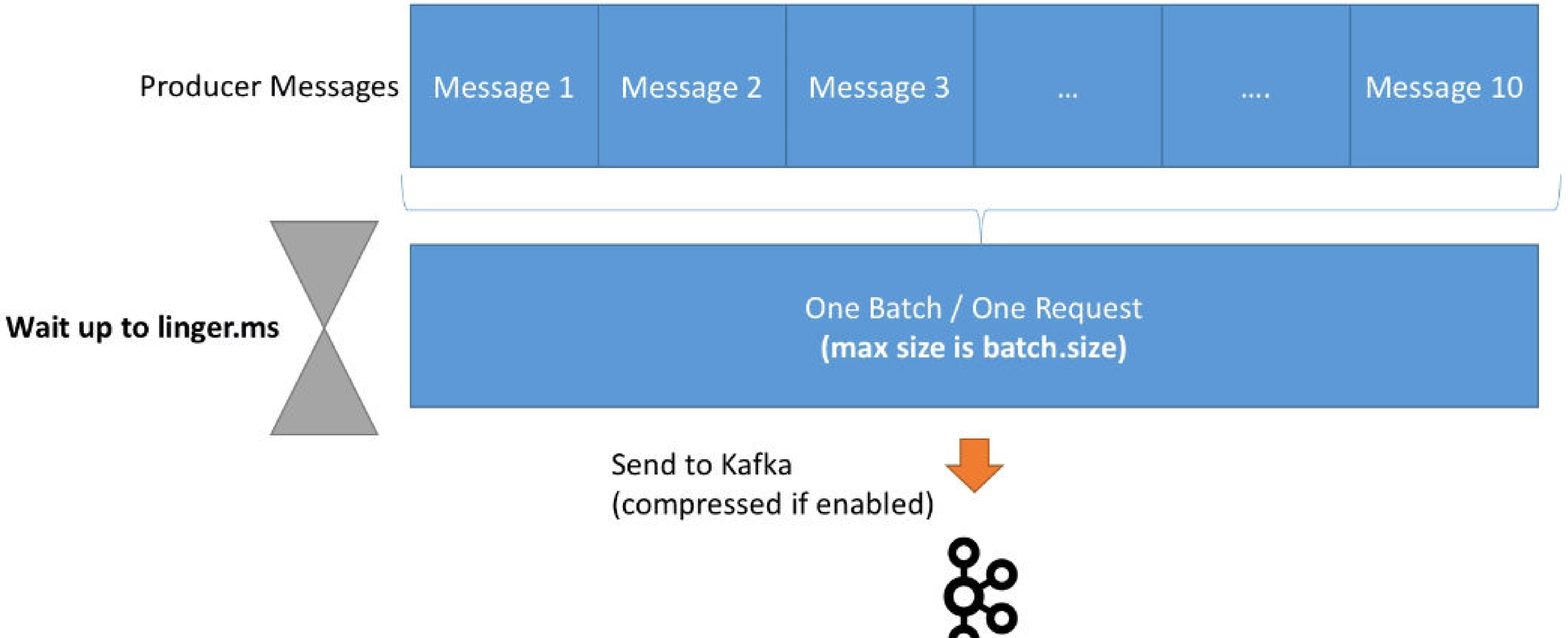


```java
// high throughput producer (at the expense of a bit of latency and CPU usage)
properties.setProperty(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
properties.setProperty(ProducerConfig.LINGER_MS_CONFIG, "20");
properties.setProperty(ProducerConfig.BATCH_SIZE_CONFIG, Integer.toString( i: 32*1024)); // 32 KB batch size
```

Linger.ms & batch.size

- **Linger.ms:** Number of milliseconds a producer is willing to wait before sending a batch out. (default 0)
- By introducing some lag (for example linger.ms=5), we increase the chances of messages being sent together in a batch
- So at the expense of introducing a small delay, we can increase throughput, compression and efficiency of our producer.
- If a batch is full (see batch.size) before the end of the linger.ms period, it will be sent to Kafka right away!

Linger.ms


Producer Messages
Message 1
Message 2
Message 3
...
....
Message 10
Wait up to linger.ms
One Batch / One Request
(max size is batch.size)
Send to Kafka
(compressed if enabled)

Producer Default Partitioner and how keys are hashed

- By default, your keys are hashed using the "murmur2" algorithm.
- It is most likely preferred to not override the behavior of the partitioner, but it is possible to do so (partitioner.class).
- The formula is:
  `targetPartition = Utils.abs(Utils.murmur2(record.key())) % numPartitions;`
- <u>This means that same key will go to the same partition (we already know this), and adding partitions to a topic will completely alter the formula</u>

Producer Retries

- In case of transient failures, developers are expected to handle exceptions, otherwise the data will be lost.
- Example of transient failure:
  - NotEnoughReplicasException
- There is a "retries" setting
  - defaults to 0 for Kafka <= 2.0
  - defaults to 2147483647 for Kafka >= 2.1
- The retry.backoff.ms setting is by default 100 ms

Getting ready

I believe, you have already installed Kafka. There is a starter file for producer in the `config` folder named `producer.properties`. Now, let's get cracking at it with your favorite editor.

How to do it...

Open your `producer.properties` file and perform the following steps to configure the basic settings for producer:

1. The first configuration that you need to change is `metadata.broker.list`:

   ```
   metadata.broker.list=192.168.0.2:9092,192.168.0.3:9092
   ```

2. Next, you need to set the `request.required.acks` value:

   ```
   request.required.acks=0
   ```

3. Set the `request.timeout.ms` value:

   ```
   request.timeout.ms=10000
   ```

How it works...

With these basic configuration parameters, your Kafka producer is ready to go. Based on your circumstances, you might have tweak these values for optimal performance.

- `metadata.broker.list`: This is the most important setting that is used to get the metadata such as topics, partition, and replicas. This information is used to set up the connection to produce the data. The format is `host1:port1`, `host2:port2`. You may not give all the Kafka brokers, but a subset of them or a VIP pointing to a subset of brokers.
- `request.required.acks`: Based on this setting, the producer determines when to consider the produced message as complete. You also need to set how many brokers would commit to their logs before you acknowledge the message. If the value is set to `0`, it means that the producer will send the message in the fire and forget mode. If the value is set to `1`, it means the producer will wait till the leader replica receives the message. If the value is set to `-1`, the producer will wait till all the in-sync replicas receive the message. This is definitely the most durable way to keep data, but this will also be the slowest way.
- `request.timeout.ms`: This sets the amount of time the broker will wait, trying to meet the `request.required.acks` requirement before sending back an error to the client.

These are the settings you need to configure if you want to get the best performance out of your Kafka producer.

Getting ready

You can start by editing the `producer.properties` file in the `config` folder of your Kafka installation. This is another key value pair file which you can open to edit in your favorite text editor.

How to do it...

Proceed with the following steps to configure the thread and performance for producer:

1. You can set the `producer.type` value:
```
producer.type=sync
```
2. Set the `serializer.class` value:
```
serializer.class=kafka.serializer.DefaultEncoder
```
3. Set the `key.serializer.class` value:
```
key.serializer.class=kafka.serializer.DefaultEncoder
```
4. Set the `partitioner.class` value:
```
partitioner.class=kafka.producer.DefaultPartitioner
```
5. Set the `compression.codec` value:
```
compression.codec=none
```
6. Set the `compressed.topics` value:
```
compressed.topics=mytesttopic1
```
7. Set the `message.send.max.retries` value:
```
message.send.max.retries=3
```
8. Set the `retry.backoff.ms` value:
```
retry.backoff.ms=100
```
9. Set the `topic.metadata.refresh.interval.ms` value:
```
topic.metadata.refresh.interval.ms=600000
```
10. Set the `queue.buffering.max.ms` value:
```
queue.buffering.max.ms=5000
```
11. Set the `queue.buffering.max.messages` value:
```
queue.buffering.max.messages=10000
```
12. Set the `queue.enqueue.timeout.ms` value:
```
queue.enqueue.timeout.ms=-1
```
13. Set the `batch.num.messages` value:
```
batch.num.messages=200
```
14. Set the `send.buffer.bytes` value:
```
send.buffer.bytes=102400
```
15. Set the `client.id` value:
```
client.id=my_client
```

How it works

- `producer.type`: This accepts the two values, `sync` and `async`. When in the `async` mode, the producer will send data to the broker via a background thread that allows for the batching up of requests. But this might lead to the loss of data if the client fails.
- `serializer.class`: This is used to declare the `serializer` class, which is used to serialize the message and store it in a proper format to be retrieved later. The default encoder takes a byte array and returns the same byte array.
- `key.serializer.class`: This same as the `serializer` class for keys. Its default is the same as the one for messages if nothing is given.
- `partitioner.class`: The default value for partitioning messages among subtopics is the hash value of the key.
- `compression.codec`: This defines the compression codec for all the generated data. The valid values are `none`, `gzip`, and `snappy`. In general, it is a good idea to send all the messages in compressed format.
- `compressed.topics`: This sets whether the compression is turned on for a particular topic.
- `message.send.max.retries`: This property sets the maximum number of retries to be performed before sending messages is considered a failure.
- `retry.backoff.ms`: Electing a leader takes time. The producer cannot refresh metadata during this time. An error in sending data will mean that it should refresh the metadata as well before retrying. This property specifies the time the producer waits before it tries again.
- `topic.metadata.refresh.interval.ms`: This specifies the refresh interval for the metadata from the brokers. If this value is set to `-1`, metadata will be refreshed only in the case of a failure. If this value is set to `0`, metadata will be refreshed with every sent message.
- `queue.buffering.max.ms`: This sets the maximum time to buffer data before it is sent across to the brokers in the `async` mode. This improves the throughput, but adds latency.
- `queue.buffering.max.messages`: This sets the maximum number of messages that are to be queued before they are sent across while using the `async` mode.
- `queue.enqueue.timeout.ms`: This sets the amount of time the producer will block before dropping messages while running in the `async` mode once the buffer is full. If this value is set to `0`, the events will be queued immediately. They will be dropped if the queue is full. If it is set to `-1`, the producer will block the event; it will never willingly drop a message.
- `batch.num.messages`: This specifies the number of messages to be sent in a batch while using the `async` mode. The producer will wait till it reaches the number of messages to be sent.
- `send.buffer.bytes`: This sets the socket buffer size.
- `client.id`: This sets the client ID that can be used to debug messages sent from the application.

See also

- Please refer to http://kafka.apache.org/documentation.html#producerconfigs for more details on producer configurations.

Producer Timeouts

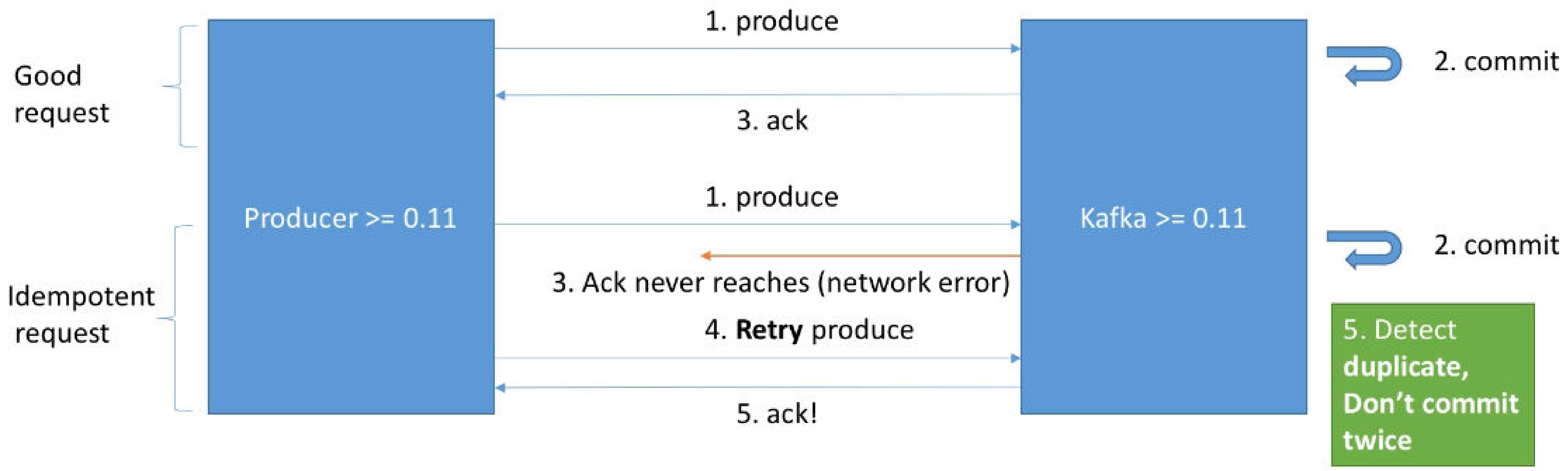
- If retries > 0, for example retries = 2147483647
- the producer won't try the request for ever, it's bounded by a timeout

- For this, you can set an intuitive Producer Timeout (KIP-91 – Kafka 2.1)
- delivery.timeout.ms = 120 000 ms == 2 minutes

- Records will be failed if they can't be acknowledged in delivery.timeout.ms

Idempotent Producer

- In Kafka >= 0.11, you can define a "idempotent producer" which won't introduce duplicates on network error



Idempotent Producer

- Idempotent producers are great to guarantee a stable and safe pipeline!

- They come with:
  - retries = Integer.MAX_VALUE (2^31–1 = 2147483647)
  - max.in.flight.requests=1 (Kafka == 0.11) or
  - max.in.flight.requests=5 (Kafka >= 1.0 – higher performance & keep ordering)
    See https://issues.apache.org/jira/browse/KAFKA-5494
  - acks=all

- These settings are applied automatically after your producer has started if you don't set them manually

- Just set:
  - **producerProps.put("enable.idempotence", true);**

Producer Retries: Warning

- In case of retries, there is a chance that messages will be sent out of order (if a batch has failed to be sent).
- If you rely on key-based ordering, that can be an issue.

- For this, you can set the setting while controls how many produce requests can be made in parallel: max.in.flight.requests.per.connection
  - Default: 5
  - Set it to 1 if you need to ensure ordering (may impact throughput)

Getting ready

You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it...

Proceed with the following steps to set the values of the other configurations for the consumer:

1. Set `offsets.storage`:

   ```
   offsets.storage=zookeeper
   ```

2. Set `offsets.channel.backoff.ms`:

   ```
   offsets.channel.backoff.ms=6000
   ```

3. Set `offsets.channel.socket.timeout.ms`:

   ```
   offsets.channel.socket.timeout.ms=6000
   ```

4. Set `offsets.commit.max.retries`:

   ```
   offsets.commit.max.retries=5
   ```

5. Set `dual.commit.enabled`:

   ```
   dual.commit.enabled=true
   ```

6. Set `client.id`:

   ```
   client.id=mycid
   ```

How it works...

In this section, we will discuss in detail about the properties set in the previous section:

- `offsets.storage`: This sets the location where the offsets need to be stored; it can be ZooKeeper or Kafka.
- `offsets.channel.backoff.ms`: This sets the time period between two attempts to reconnect offset channel or retrying the failed attempt to get offset fetch/commit requests.
- `offsets.channel.socket.timeout.ms`: This sets the socket timeout to read the responses for offset fetch or commit requests.
- `offsets.commit.max.retries`: This sets the number of retries for the consumer to save the offset during shut down. This does not apply to regular autocommits.
- `dual.commit.enabled`: If this value is `true` and your offset storage is set to Kafka, then the offset will be additionally committed to ZooKeeper.
- `client.id`: This sets the user-specified string for the application that can help you debug.

See also

- More information on the consumer configuration is available at http://kafka.apache.org/documentation.html#consumerconfigs.

Configuring the basic settings for consumer

Now that you have the producers all configured up, it is time to configure the consumer for your application.

Getting ready

This is another key value pair file. You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it...

Proceed with the following steps:

1. You can set the `group.id` value:

   ```
   group.id=mygid
   ```

2. Set the `zookeeper.connect` value:

   ```
   zookeeper.connect=192.168.0.2:2181
   ```

3. Set the `consumer.id` value:

   ```
   consumer.id=mycid
   ```

How it works...

In this section, we will discuss in detail about the properties set in the previous section:

- `group.id`: This is the string that identifies a group of consumers as a single group. By setting them to the same ID, you can mark them as a part of the same group.
- `zookeeper.connect`: This specifies the ZooKeeper connection string in the `host:port` format. You can add multiple ZooKeeper host names by keeping them comma-separated.
- `consumer.id`: This is used to uniquely identify the consumer. It will be autogenerated if it is not set.

Getting ready

You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it...

Proceed with the following steps to configure the log settings for the consumer:

1. Set `auto.commit.enable`:

   ```
   auto.commit.enable=true
   ```

2. Set `auto.commit.interval.ms`:

   ```
   auto.commit.interval.ms=60000
   ```

3. Set `rebalance.max.retries`:

   ```
   rebalance.max.retries=4
   ```

4. Set `rebalance.backoff.ms`:

   ```
   rebalance.backoff.ms=2000
   ```

5. Set `refresh.leader.backoff.ms`:

   ```
   refresh.leader.backoff.ms=200
   ```

6. Set `auto.offset.reset`:

   ```
   auto.offset.reset=largest
   ```

7. Set `partition.assignment.strategy`:

   ```
   partition.assignment.strategy=range
   ```

How it works...

In this section, we will discuss in detail about the properties set in the previous section:

- `auto.commit.enable`: If the value for this is set to `true`, the consumer will save the offset of the messages that will be used to recover in case the consumer fails.
- `auto.commit.interval.ms`: This is the interval in which it will commit the message offset to the zookeeper.
- `rebalance.max.retries`: The number of partitions is equally distributed among a group of consumers. If a new one joins, it will try to rebalance the allocation of partitions. If the group set changes while the rebalance is happening, it will fail to assign a partition to a new consumer. This setting specifies the number of retries the consumer will do before quitting.
- `rebalance.backoff.ms`: This specifies the time interval between two attempts by the consumer to rebalance
- `refresh.leader.backoff.ms`: This specifies the time the consumer will wait before it tries to find a new leader for the partition that has lost its leader.
- `auto.offset.reset`: This specifies what the consumer should do when there is no initial state saved in ZooKeeper or if an offset is out of range. It can take two values: `smallest` or `largest`. This will set the offset to the smallest or largest value, respectively.
- `partition.assignment.strategy`: This specifies the partition assignment strategy that can either be range or round robin.

While using consumers in production, you would want the best performance. These configurations are what makes you extract the last bit of performance from your servers.

Getting ready

You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it...

Proceed with the following steps to configure the thread and performance for the consumer:

1. Set `socket.timeout.ms`:

   ```
   socket.timeout.ms=30000
   ```

2. Set `socket.receive.buffer.bytes`:

   ```
   socket.receive.buffer.bytes=65536
   ```

3. Set `fetch.message.max.bytes`:

   ```
   fetch.message.max.bytes=1048576
   ```

4. Set `queued.max.message.chunks`:

   ```
   queued.max.message.chunks=2
   ```

5. Set `fetch.min.bytes`:

   ```
   fetch.min.bytes=1
   ```

6. Set `fetch.wait.max.ms`:

   ```
   fetch.wait.max.ms=100
   ```

7. Set `consumer.timeout.ms`:

   ```
   consumer.timeout.ms=-1
   ```

How it works...

In this section, we will discuss in detail about the properties set in the previous section:

- `socket.timeout.ms`: This sets the socket time value for the consumer.
- `socket.receive.buffer.bytes`: This sets the receive buffer size for network requests.
- `fetch.message.max.bytes`: This sets the number of bytes to fetch from each topic's partition in each request. This value must be at least as big as the maximum message size for the producer; else it may fail to fetch messages in case the producer sends a message greater than this value. This also defines the memory used by the consumer to keep the messages fetched in memory. So, you must choose this value carefully.
- `num.consumer.fetchers`: This sets the number of threads used to fetch data from Kafka.
- `queued.max.message.chunks`: This sets the maximum number of chunks that can be buffered for consumption. A chunk can have a maximum size of `fetch.message.max.bytes`.
- `fetch.min.bytes`: This sets the minimum number of bytes to be fetched from the server. It will wait till it has this much amount of data to be fetched before the request is serviced.
- `fetch.wait.max.ms`: This sets the maximum time a request waits if there is no sufficient data as specified by `fetch.min.bytes`.
- `consumer.timeout.ms`: This sets the timeout for a consumer thread to wait before throwing an exception if no message is available for consumption.

Configuring the ZooKeeper settings for consumer

ZooKeeper is used for cluster management and these are the settings to fine-tune it.

Getting ready

You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it...

Proceed with the following steps to configure the ZooKeeper settings for the consumer:

1. Set `zookeeper.session.timeout.ms`:

```
zookeeper.session.timeout.ms=6000
```

2. Set `zookeeper.connection.timeout.ms`:

```
zookeeper.connection.timeout.ms=6000
```

3. Set `zookeeper.sync.time.ms`:

```
zookeeper.sync.time.ms=2000
```

How it works...

In this section, we will discuss in detail about the properties set in the previous section:

- `zookeeper.session.timeout.ms`: This sets the time period before which if the consumer does not send a heartbeat message from the ZooKeeper, it will be considered dead and a consumer rebalance will happen.
- `zookeeper.connection.timeout.ms`: This sets the maximum time the client waits to establish a connection with ZooKeeper.
- `zookeeper.sync.time.ms`: This sets the time period the ZooKeeper follower can be behind the leader.