Saturday, July 11, 2020 11:19 PM

ZOOKEEPER

>bin/zookeeper-server-start.sh config/zookeeper.properties
start the Kafka server:
> bin/kafka-server-start.sh config/server.properties

Create a topic

bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic test

Send some messages

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
This is a message
This is another message

Setting up a multi-broker cluster

First we make a config file for each of the brokers
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties

Now edit these new files and set the following properties:
config/server-1.properties:
broker.id=1
listeners=PLAINTEXT://:9093
log.dirs=/tmp/kafka-logs-1

config/server-2.properties:
broker.id=2
listeners=PLAINTEXT://:9094
log.dirs=/tmp/kafka-logs-2

We already have Zookeeper and our single node started, so we just need to start the two new nodes:
> bin/kafka-server-start.sh config/server-1.properties &
> bin/kafka-server-start.sh config/server-2.properties &

create a new topic with a replication factor of three:
> bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 3 --partitions 1 --topic my-replicated-topic
O/P of cmd:
> bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-replicated-topic
Topic:my-replicated-topic PartitionCount:1 ReplicationFactor:3 Configs:
Topic: my-replicated-topic Partition: 0 Leader: 1 Replicas: 1,2,0 Isr: 1,2,0

The first line gives a summary of all the partitions, each additional line gives information about one partition. Since we have only one partition for this topic there is only one line.

- "leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.
- "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
- "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

Kafka producer/consumer

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-clients</artifactId>
   <version>2.5.0</version>
</dependency>
```

Streams API

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-streams</artifactId>
   <version>2.5.0</version>
</dependency>
```

Kafka Producer

A Kafka client that publishes records to the Kafka cluster.

The producer is *thread safe* and sharing a single producer instance across threads will generally be faster than having multiple instances.
Here is a simple example of using the producer to send records with strings containing sequential numbers as the key/value pairs.

```
 Properties props = new Properties();
 props.put("bootstrap.servers", "localhost:9092");
 props.put("acks", "all");
 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
Producer<String, String> producer = new KafkaProducer<>(props);
 for (int i = 0; i < 100; i++)z
    producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i), Integer.toString(i)));
producer.close();
```

Now we can start the console producer in a separate terminal to write some input data to this topic:

```
1 > bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic streams-plaintext-input
```

and inspect the output of the WordCount demo application by reading from its output topic with the console consumer in a separate terminal:

```
1 > bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
2     --topic streams-wordcount-output \
3     --from-beginning \
4     --formatter kafka.tools.DefaultMessageFormatter \
5     --property print.key=true \
6     --property print.value=true \
7     --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
8
      --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

From <https://kafka.apache.org/25/documentation/streams/quickstart>

The producer consists of a pool of buffer space that holds records that haven't yet been transmitted to the server as well as a background I/O thread that is responsible for turning these records into requests and transmitting them to the cluster. Failure to close the producer after use will leak these resources.
The send() method is asynchronous. When called it adds the record to a buffer of pending record sends and immediately returns. This allows the producer to batch together individual records for efficiency.
The acks config controls the criteria under which requests are considered complete. The "all" setting we have specified will result in blocking on the full commit of the record, the slowest but most durable setting.
If the request fails, the producer can automatically retry, though since we have specified retries as 0 it won't. Enabling retries also opens up the possibility of duplicates (see the documentation on message delivery semantics for details).
The producer maintains buffers of unsent records for each partition. These buffers are of a size specified by the batch.size config. Making this larger can result in more batching, but requires more memory (since we will generally have one of these buffers for each active partition).

By default a buffer is available to send immediately even if there is additional unused space in the buffer. However if you want to reduce the number of requests you can set linger.ms to something greater than 0. This will instruct the producer to wait up to that number of milliseconds before sending a request in hope that more records will arrive to fill up the same batch. This is analogous to Nagle's algorithm in TCP. For example, in the code snippet above, likely all 100 records would be sent in a single request since we set our linger time to 1 millisecond. However this setting would add 1 millisecond of latency to our request waiting for more records to arrive if we didn't fill up the buffer. Note that records that arrive close together in time will generally batch together even with linger.ms=0 so under heavy load batching will occur regardless of the linger configuration; however setting this to something larger than 0 can lead to fewer, more efficient requests when not under maximal load at the cost of a small amount of latency.
The buffer.memory controls the total amount of memory available to the producer for buffering. If records are sent faster than they can be transmitted to the server then this buffer space will be exhausted. When the buffer space is exhausted additional send calls will block. The threshold for time to block is determined by max.block.ms after which it throws a TimeoutException.
The key.serializer and value.serializer instruct how to turn the key and value objects the user provides with their ProducerRecord into bytes. You can use the included ByteArraySerializer or StringSerializer for simple string or byte types.

The KafkaProducer supports two additional modes: the idempotent producer and the transactional producer. The idempotent producer strengthens Kafka's delivery semantics from at least once to exactly once delivery. In particular producer retries will no longer introduce duplicates. The transactional producer allows an application to send messages to multiple partitions (and topics!) atomically.

To take advantage of the idempotent producer, it is imperative to avoid application level re-sends since these cannot be de-duplicated. As such, if an application enables idempotence, it is recommended to leave the retries config unset, as it will be defaulted to Integer.MAX_VALUE. Additionally, if a send(ProducerRecord) returns an error even with infinite retries (for instance if the message expires in the buffer before being sent), then it is recommended to shut down the producer and check the contents of the last produced message to ensure that it is not duplicated. Finally, the producer can only guarantee idempotence for messages sent within a single session.
To use the transactional producer and the attendant APIs, you must set the transactional.id configuration property. If the transactional.id is set, idempotence is automatically enabled along with the producer configs which idempotence depends on. Further, topics which are included in transactions should be configured for durability. In particular, the replication.factor should be at least 3, and the min.insync.replicas for these topics should be set to 2. Finally, in order for transactional guarantees to be realized from end-to-end, the consumers must be configured to read only committed messages as well.
The purpose of the transactional.id is to enable transaction recovery across multiple sessions of a single producer instance. It would typically be

derived from the shard identifier in a partitioned, stateful, application. As such, it should be unique to each producer instance running within a partitioned application.

All the new transactional APIs are blocking and will throw exceptions on failure. The example below illustrates how the new APIs are meant to be used. It is similar to the example above, except that all 100 messages are part of a single transaction.

```
Properties props = new Properties();
 props.put("bootstrap.servers", "localhost:9092");
 props.put("transactional.id", "my-transactional-id");
 Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new StringSerializer());
producer.initTransactions();
try {
    producer.beginTransaction();
    for (int i = 0; i < 100; i++)
        producer.send(new ProducerRecord<>("my-topic", Integer.toString(i), Integer.toString(i)));
    producer.commitTransaction();
 } catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
    // We can't recover from these exceptions, so our only option is to close the producer and exit.
    producer.close();
 } catch (KafkaException e) {
    // For all other exceptions, just abort the transaction and try again.
    producer.abortTransaction();
 }
 producer.close();
```

By calling producer.abortTransaction() upon receiving a KafkaException we can ensure that any successful writes are marked as aborted, hence keeping the transactional guarantees.

From <https://kafka.apache.org/25/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html>

Kafka Consumer

OFFSET STORAGE - KAFKA
Offsets in Kafka are stored as messages in a separate topic named **'__consumer_offsets'** . Each consumer commits a message into the topic at periodic intervals. The message contains the metadata related to the current offset, the consumer group , partition number, topic associated with the offset and other useful information.
READING OFFSETS FROM KAFKA
Since __consumer_offsets is a just like any other topic, its possible to consume the message off. Before we do that we need make this topic visible to the consumers since this is an internal KAFKA topic and is not visible to the consumers by default. In order to make the topic visible, execute the following command.

```
bash$echo "exclude.internal.topics=false" > /tmp/consumer.config

bash$./kafka-console-consumer.sh --consumer.config /tmp/consumer.config --formatter "kafka.coordinator.GroupMetadataManager
\$OffsetsMessageFormatter" --zookeeper : --topic __consumer_offsets
```

From <https://elang2.github.io/myblog/posts/2017-09-20-Kafak-And-Zookeeper-Offsets.html>

From <https://elang2.github.io/myblog/posts/2017-09-20-Kafak-And-Zookeeper-Offsets.html>

```
Properties props = new Properties();
 props.setProperty("bootstrap.servers", "localhost:9092");
 props.setProperty("group.id", "test");
 props.setProperty("enable.auto.commit", "true");
 props.setProperty("auto.commit.interval.ms", "1000");
 props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
 props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
 consumer.subscribe(Arrays.asList("foo", "bar"));
 while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());
 }
```

Manual Offset Control
Instead of relying on the consumer to periodically commit consumed offsets, users can also control when records should be considered as consumed and hence

commit their offsets. This is useful when the consumption of the messages is coupled with some processing logic and hence a message should not be considered as consumed until it is completed processing.

```
    Properties props = new Properties();
    props.setProperty("bootstrap.servers", "localhost:9092");
    props.setProperty("group.id", "test");
    props.setProperty("enable.auto.commit", "false");
    props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Arrays.asList("foo", "bar"));
    final int minBatchSize = 200;
    List<ConsumerRecord<String, String>> buffer = new ArrayList<>();
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
        for (ConsumerRecord<String, String> record : records) {
            buffer.add(record);
        }
        if (buffer.size() >= minBatchSize) {
            insertIntoDb(buffer);
            consumer.commitSync();
            buffer.clear();
        }
    }
```

From <https://kafka.apache.org/25/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html>

Creating your own stratergy in Kafka

https://medium.com/streamthoughts/understanding-kafka-partition-assignment-strategies-and-how-to-write-your-own-custom-assignor-ebeda1fc06f3

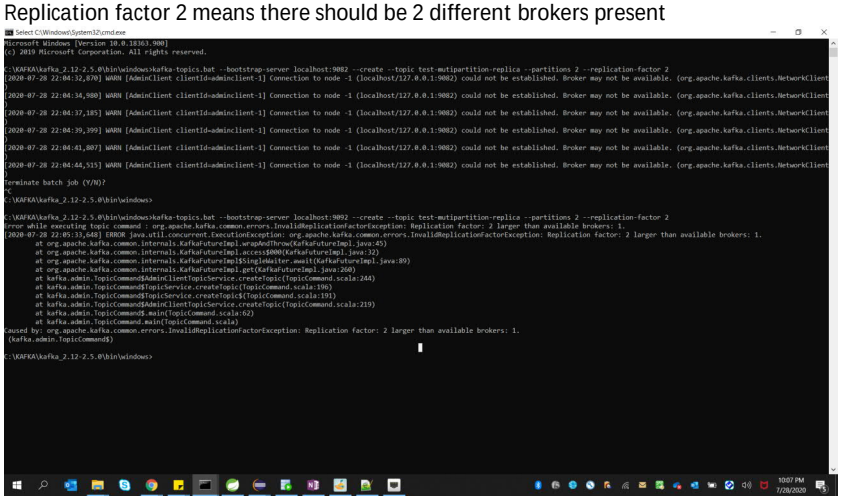
KAKFA REBALANCE PROTOCOL
https://medium.com/streamthoughts/apache-kafka-rebalance-protocol-or-the-magic-behind-your-streams-applications-e94baf68e4f2

KAFKA Details Advanced brief: -- Its interesting it has details of implementation of customizing
http://cloudurable.com/ppt/6-kafka-consumers-advanced.pdf

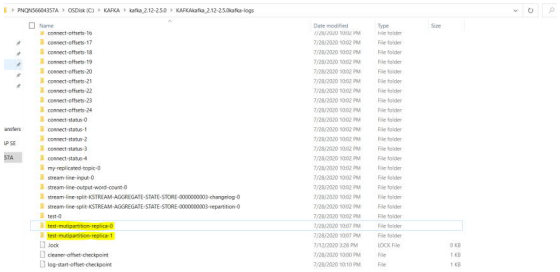
What happens when we add a topic ?

```
bin/kafka-topics.sh --bootstrap-server broker_host:port --create --topic my_topic_name \
      --partitions 20 --replication-factor 3 --config x=y
```

The replication factor controls how many servers will replicate each message that is written. If you have a replication factor of 3 then up to 2 servers can fail before you will lose access to your data. We recommend you use a replication factor of 2 or 3 so that you can transparently bounce machines without interrupting data consumption.
Replication factor 2 means there should be 2 different brokers present




```
C:\KAFKA\kafka_2.12-2.5.0\bin\windows>kafka-topics.bat --bootstrap-server localhost:9092 --create --topic test-mutipartition-replica --partitions 2 --replication-factor 1
Created topic test-mutipartition-replica.
```

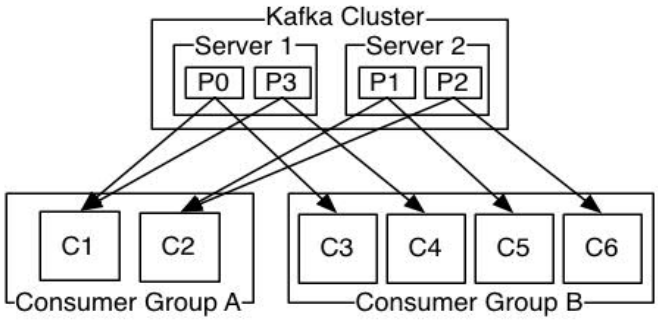


The partition count controls how many logs the topic will be sharded into. There are several impacts of the partition count. First each partition must fit entirely on a single server. So if you have 20 partitions the full data set (and read and write load) will be handled by no more than 20 servers (not counting replicas). Finally the partition count impacts the maximum parallelism of your consumers.

If you create a topic with no of partition =2 then under kafkalogs folder there will be 2 folder created namely <topicname>-partionNumber

Each sharded partition log is placed into its own folder under the Kafka log directory. The name of such folders consists of the topic name, appended by a dash (-) and the partition id. Since a typical folder name can not be over 255 characters long, there will be a limitation on the length of topic names. We assume the number of partitions will not ever be above 100,000. Therefore, topic names cannot be longer than 249 characters. This leaves just enough room in the folder name for a dash and a potentially 5 digit long partition id.

From <https://kafka.apache.org/documentation/#design_replicatedlog>



Consumers label themselves with a *consumer group* name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines. i.e If a record is pushed to a topic then P0
>