KAFKA TUTORIAL: CREATING ADVANCED KAFKA PRODUCERS IN JAVA

July 3, 2017

Kafka Tutorial 13: Creating Advanced Kafka Producers in Java

In this tutorial, you are going to create advanced *Kafka Producers*.

Before you start

The prerequisites to this tutorial are

- Kafka from the command line (http://cloudurable.com/blog/kafka-tutorial-kafka-from-command-line/index.html)
- Kafka clustering and failover basics (http://cloudurable.com/blog/kafka-tutorial-kafka-failover-kafka-cluster/index.html)
- and Creating a Kafka Producer in Java (http://cloudurable.com/blog/kafka-tutorial-kafka-producer/index.html).

This tutorial picks up right where Kafka Tutorial Part 11: Writing a Kafka Producer example in Java (http://cloudurable.com/blog/kafka-tutorial-kafka-producer/index.html) and Kafka Tutorial Part 12: Writing a Kafka Consumer example in Java (http://cloudurable.com/blog/kafka-tutorial-kafka-consumer/index.html) left off. In the last two tutorial, we created simple Java example that creates a Kafka producer and a consumer.

Kafka Tutorial 13: Creating Advanced Kafka Producers in Java Slides (http://cloudurable.com/ppt/kafka-java-producer-advanced-java-examples.pdf)

This tutorial covers advanced producer topics like custom *serializers*, *ProducerInterceptors*, custom *Partitioners*, timeout, record batching & linger, and compression.

This tutorial is under construction, but we have complete example code and slides explaining custom *Serializers*, *ProducerInterceptors*, custom *Partitioners*, timeout, record batching & linger, and compression.

Kafka Producers

A producer is a type of Kafka client that publishes records to Kafka cluster. The Kafka client API for Producers are thread safe. A Kafka *Producer* has a pool of buffer that holds to-be-sent records. The producer has background, I/O threads for turning records into request bytes and transmitting requests to Kafka cluster. The producer must be closed to not leak resources, i.e., connections, thread pools, buffers.

Kafka Producer Send, Acks and Buffers

The Kafka Producer has a *send()* method which is asynchronous. Calling the send method adds the record to the output buffer and return right away. The buffer is used to batch records for efficient IO and compression. The Kafka Producer configures acks to control record durability. The "all" acks setting ensures full commit of record to all replicas and is most durable and least fast setting. The Kafka Producer can automatically retry failed requests. The Producer has buffers of unsent records per topic partition (sized at *batch.size*).

Kafka Producer: Buffering and batching

The Kafka Producer buffers are available to send immediately. The buffers are sent as fast as broker can keep up (limited by in-flight *max.in.flight.requests.per.connection*). To reduce requests count and increase throughput, set linger.ms > 0. This setting forces the Producer to wait up to *linger.ms* before sending contents of buffer or until batch fills up whichever comes first. Under heavy load *linger.ms* not met as the buffer fills up before the linger.ms duration completes. Under lighter load, the producer can use to linger to increase broker IO throughput and increase compression. The *buffer.memory* controls total memory available to a producer for buffering. If records get sent faster than they can be transmitted to Kafka then and this buffer will get exceeded then additional *send* calls block up to *max.block.ms* after then Producer throws a *TimeoutException*.

Producer Acks

When using a producer, you can configure its acks (Acknowledgments) which default to "all". The acks config setting is the write-acknowledgment received count required from partition leader before the producer write request is deemed complete. This setting controls the producer's durability which can be very strong (all) or none. Durability is a tradeoff between throughput and consistency. The acks setting is set to "all" (-1), "none" (0), or "leader" (1).

Acks 0 (NONE)

The acks=0 is none meaning the Producer does not wait for any ack from Kafka broker at all. The records added to the socket buffer are considered sent. There are no guarantees of durability. The record offset returned from the send method is set to -1 (unknown). There could be record loss if the leader is down. There could be use cases that need to maximize throughput over durability, for example, log aggregation.

Acks 1 (LEADER)

The acks=1 is leader acknowledgment. The means that the Kafka broker acknowledges that the partition leader wrote the record to its local log but responds without the partition followers confirming the write. If leader fails right after sending ack, the record could be lost as the followers might not have replicated the record yet. Record loss is rare but possible, and you might only see this used if a rarely missed record is not statistically significant, log aggregation, a collection of data for machine learning or dashboards, etc.

Acks -1 (ALL)

The *acks=all* or *acks=-1* is all acknowledgment which means the leader gets write confirmation from the full set of ISRs before sending an ack back to the producer. This guarantees that a record is not lost as long as one ISR remains alive. This ack=all setting is the *strongest* available guarantee that Kafka provides for durability.
This setting is even stronger with broker setting *min.insync.replicas* which specifies the minimum number of ISRs that must acknowledge a write. Most use cases will use *acks=all* and set a *min.insync.replicas > 1*.

Setting acks config on Kafka Producer

```java
package com.cloudurable.kafka.producer;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;

public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice> createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        setupBatchingAndCompression(props);
        setupRetriesInFlightTimeout(props);

        //Set number of acknowledgments - acks - default is all
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        ...
        return new KafkaProducer<>(props);
    }
    ...
}
```

The “acks” (`ProducerConfig.ACKS_CONFIG`) config gets passed to the properties via the constructor of the `KafkaProducer` .

Kafka Producer: Producer Buffer Memory Size

You can also set the producer config property *buffer.memory* which default 32 MB of memory. This denotes the total memory (in bytes) that the producer can use to buffer records to be sent to the broker. The Producer blocks up to *max.block.ms* if *buffer.memory* is exceeded. If the Producer is sending records faster than the broker can receive records, an exception is thrown.

Kafka Producer: Batching by Size

The producer config property batch.size defaults to 16K bytes. This is used by the Producer to batch records. This setting enables fewer requests and allows multiple records to be sent to the same partition. Use this batch.size setting to improve IO throughput and performance on both producer and server (and consumer). Using a larger batch.size makes compression more efficient. If a record is larger than the batch size, it will not be batched. This setting allows the Producer to send requests containing multiple batches. The batches are per partition. The smaller the batch size the less the throughput and performance. If the batch size is too big and often gets sent before full, the memory allocated for the batch is wasted.

Kafka Producer: Batching by Time and Size

The producer config property *linger.ms* defaults to 0. You can set this so that the Producer will wait this long before sending if batch size not exceeded. This setting allows the Producer to group together any records that arrive before they can be sent into a batch. Setting this value to 5 ms is greater is good if records arrive faster than they can be sent out. The producer can reduce requests count even under moderate load using linger.ms.

The *linger.ms* setting adds a delay to wait for more records to build up, so larger batches get sent. Increase *linger.ms* to increase brokers throughput at the cost of producer latency. If the producer gets records whose size is *batch.size* or more for a broker's leader partitions, then it is sent right away. If Producers gets less than *batch.size* but *linger.ms* interval has passed, then records for that partition are sent. Increase *linger.ms* to improve the throughput of Brokers and reduce broker load (common improvement).

Kafka Producer: Compressing Batches and End to End compression

The producer config property *compression.type* defaults to none. Setting this allows the producer to compresses request data. By default, the producer does not compress request data. This setting is set to none, gzip, snappy, or lz4. The compression is by batch and improves with larger batch sizes. End to end compression is possible if the Kafka Broker config “compression.type” set to “producer”. The compressed data can be sent from a producer, then written to the topic log and forwarded to a consumer by broker using the same compressed format. End to end compression is efficient as compression only happens once and is reused by the broker and consumer. End to end compression takes the load off of the broker.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: Batching and Compression Example

```
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        setupBatchingAndCompression(props);
        ...
        return new KafkaProducer<>(props);
    }

    private static void setupBatchingAndCompression(
            final Properties props) {
        //Linger up to 100 ms before sending batch if size not met
        props.put(ProducerConfig.LINGER_MS_CONFIG, 100);

        //Batch up to 64K buffer sizes.
        props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384 * 4);

        //Use Snappy compression for batch compression.
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
    }
}
```

The above sets the Producer config ProducerConfig.LINGER_MS_CONFIG (“linger.ms”) config property to to 100 ms, ProducerConfig.BATCH_SIZE_CONFIG (“batch.size”) config property to 64K bytes, and the ProducerConfig.COMPRESSION_TYPE_CONFIG (“compression.type”) config property to snappy.

This means that after 100ms if 64K bytes of records (per partition) is not added to the output buffer then what is there will be compressed and sent to the Kafka broker.

Review: Buffering, batching and compression

- How can you increase throughput of Kafka?
- How can LINGER_MS_CONFIG to increase throughput?
- What is end to end batching and compression and how do you enable it?

- How can you increase throughput of Kafka? You can use batching to reduce the amount of IO.

- How can LINGER_MS_CONFIG to increase throughput? The linger allows batching to grow larger and takes some load off the brokers.
- What is end to end batching and compression and how do you enable it? End to end batching is possible if you use producer compression at the broker. The producer then becomes responsible for the compression taking the load of the broker. The records get written to the disk compressed and sent to the consumer compressed.

Custom Serializers

You don't have to use built-in Kafka serializers. You can write your own. You just need to be able to convert your custom keys and values using the serializer convert to and convert from byte arrays (`byte[]`). Serializers work for keys and values, and you set them up with the Kafka Producer properties `value.serializer` , and `key.serializer` .

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: Custom Serializers config

```java
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        ...
        return new KafkaProducer<>(props);
    }

    private static void setupBootstrapAndSerializers(Properties props) {
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                StockAppConstants.BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                StringSerializer.class.getName());


        //Custom Serializer - config "value.serializer"
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                StockPriceSerializer.class.getName());

    }

}
```

The above sets the `value.serializer` producer config property.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceSerializer.java

Kafka Producer: Custom Serializer

```
package com.cloudurable.kafka.producer;
import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.common.serialization.Serializer;
import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockPriceSerializer implements Serializer<StockPrice> {

    @Override
    public byte[] serialize(String topic, StockPrice data) {
        return data.toJson().getBytes(StandardCharsets.UTF_8);
    }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close() {
    }
}
```

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/model/StockPrice.java

Kafka Producer: Custom Domain Object for Stocks

```
package com.cloudurable.kafka.producer.model;

import io.advantageous.boon.json.JsonFactory;

public class StockPrice {

    private final int dollars;
    private final int cents;
    private final String name;
    ...
    public String toJson() {
        return "{" +
                "\"dollars\": " + dollars +
                ", \"cents\": " + cents +
                ", \"name\": \"" + name + '\"' +
                '}';
    }
}
```

The StockPrice domain object has a toJson method that is used to convert a stock price to a JSON string.

Broker Follower Write Timeout

The Producer config property *request.timeout.ms* default 30 seconds (30,000 ms). This config property sets the maximum time that the broker waits for confirmation from followers to meet Producer acknowledgment requirements for ack=all. It is a measure of broker to broker latency of the request. Note that 30 seconds is high, and a long process time is indicative of problems with the Kafka cluster or broker nodes.

Producer Retries

The Producer config property retries defaults to 0 and is the retry count if Producer does not get an ack from Kafka Broker. The Producer will only retry if record send fail is deemed a transient error (API). The Producer will act as if your producer code resent the record on a failed attempt. Note that timeouts are re-tried, but *retry.backoff.ms* (default to 100 ms) is used to wait after failure before retrying the request again. If you set retry > 0, then you should also set *max.in.flight.requests.per.connection to 1*, or there is the possibility that a re-tried message could be delivered out of order. You have to decide if out of order message delivery matters for your application.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: Custom Serializers config

```
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        setupRetriesInFlightTimeout(props);
        ...
        return new KafkaProducer<>(props);
    }

    private static void setupRetriesInFlightTimeout(Properties props) {
        //Only one in-flight messages per Kafka broker connection
        // - max.in.flight.requests.per.connection (default 5)
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
                1);
        //Set the number of retries - retries
        props.put(ProducerConfig.RETRIES_CONFIG, 3);

        //Request timeout - request.timeout.ms
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 15_000);

        //Only retry after one second.
        props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 1_000);
    }

}
```

The above code sets the retires to 3, and the maximum in-flight requests to 1 to avoid out of order retries. It sets the request timeout to 15 seconds and the retry back-off to 1 second.

Review: Kafka Retries

- How do you configure a custom producer serializer?
- If you start getting request timeouts and you increase the request.timeout.ms why might this not be a good idea?
- Why should you set max.in.flight.requests.per.connection if you use a retry count greater than 0?

- How do you configure a custom producer serializer? You just need to be able to convert your custom keys and values using the serializer convert to and convert from byte arrays (byte[]). Serializers work for keys and values, and you set them up with the Kafka Producer properties value.serializer, and key.serializer.
- If you start getting request timeouts and you increase the request.timeout.ms why might this not be a good idea? Getting timeouts is an indication that the broker is under too much load. Adding longer timeouts could mitigating things in the short term, but very likely it could put even more load on the brokers which could make things quite a bit worse.
- Why should you set max.in.flight.requests.per.connection if you use a retry count greater than 0?

Recall, if you set retry > 0, then you should also set max.in.flight.requests.per.connection to 1, or there is the possibility that a re-tried message could be delivered out of order. You have to decide if out of order message delivery matters for your application.

Producer Partitioning

The Producer config property partitioner.class sets the partitioner. By default partitioner.class is set to `org.apache.kafka.clients.producer.internals.DefaultPartitioner`. The default partitioner partitions using the hash of the record key if the record has a key. The default partitioner partitions using round-robin if the record has no key. A Partitioner class implements Partitioner interface. You can define a custom partitioner. Recall that a topic is divided into partitions and each partition has a Kafka Broker that is the partition leader. Only the partition leader accepts send requests from a Producer.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: Configure Producer Partitioning

```
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        ...
        props.put("importantStocks", "IBM,UBER");

        props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
                StockPricePartitioner.class.getName());

        return new KafkaProducer<>(props);
    }

}
```

The above sets the `partitioner.class` to StockPricePartitioner which is a custom partitioner that we defined to send `importantStocks` to a special partition.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPricePartitioner.java

Kafka Producer: Custom Producer Partitioner

```java
package com.cloudurable.kafka.producer;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;

import java.util.*;

public class StockPricePartitioner implements Partitioner{

    private final Set<String> importantStocks;
    public StockPricePartitioner() {
        importantStocks = new HashSet<>();
    }

    @Override
    public int partition(final String topic,
                         final Object objectKey,
                         final byte[] keyBytes,
                         final Object value,
                         final byte[] valueBytes,
                         final Cluster cluster) {

        final List<PartitionInfo> partitionInfoList =
                cluster.availablePartitionsForTopic(topic);
        final int partitionCount = partitionInfoList.size();
        final int importantPartition = partitionCount -1;
        final int normalPartitionCount = partitionCount -1;

        final String key = ((String) objectKey);

        if (importantStocks.contains(key)) {
            return importantPartition;
        } else {
            return Math.abs(key.hashCode()) % normalPartitionCount;
        }

    }

    @Override
    public void close() {
    }

    @Override
    public void configure(Map<String, ?> configs) {
        final String importantStocksStr = (String) configs.get("importantStocks");
        Arrays.stream(importantStocksStr.split(","))
                .forEach(importantStocks::add);
    }

}
```

The StockPricePartitioner parses `importantStocks` config setting and keeps a hash set of `importantStocks`. It checks the `importantStocks` hash set to determine if the stock price is important and if so sends the stock price to the `importantPartition` otherwise sends the stock price to one of the normal partitions (`abs(key.hashCode()) % normalPartitionCount`) based on the hash of the record key.

Producer Interceptor Config

The Producer config property interceptor.classes is empty but you can pass an comma delimited list of interceptors. The interceptors implement the ProducerInterceptor interface. A producer interceptor intercepts records a producer sends to broker and intercepts after acks are sent. You could mutate records with an interceptor.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: Configure Producer Partitioning

```
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        ...

        //Install interceptor list - config "interceptor.classes"
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
                        StockProducerInterceptor.class.getName());
        ...
        return new KafkaProducer<>(props);
    }

}
```

The above sets the `interceptor.classes` to StockProducerInterceptor which is an example interceptor that we defined.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockProducerInterceptor.java

Kafka Producer: Producer Interceptor

```
package com.cloudurable.kafka.producer;

import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Map;

public class StockProducerInterceptor implements ProducerInterceptor {

    private final Logger logger = LoggerFactory
            .getLogger(StockProducerInterceptor.class);
    private int onSendCount;
    private int onAckCount;

    @Override
    public ProducerRecord onSend(final ProducerRecord record) {
        onSendCount++;
        if (logger.isDebugEnabled()) {
            logger.debug(String.format("onSend topic=%s key=%s value=%s %d \n",
                    record.topic(), record.key(), record.value().toString(),
                    record.partition()
            ));
        } else {
            if (onSendCount % 100 == 0) {
                logger.info(String.format("onSend topic=%s key=%s value=%s %d \n",
                        record.topic(), record.key(), record.value().toString(),
                        record.partition()
                ));
            }
        }
        return record;
    }

    @Override
    public void onAcknowledgement(final RecordMetadata metadata,
                                  final Exception exception) {
        onAckCount++;
        if (logger.isDebugEnabled()) {
            logger.debug(String.format("onAck topic=%s, part=%d, offset=%d\n",
                    metadata.topic(), metadata.partition(), metadata.offset()
            ));
        } else {
            if (onAckCount % 100 == 0) {
                logger.info(String.format("onAck topic=%s, part=%d, offset=%d\n",
                        metadata.topic(), metadata.partition(), metadata.offset()
                ));
            }
        }
    }

    @Override
    public void close() {
    }

    @Override
    public void configure(Map<String, ?> configs) {
    }
}
```

The StockProducerInterceptor is a simple example that overrides onSend and onAcknowledgement to output some logging information.

KafkaProducer send() Method

There are two forms of send method. There is the send method with a callback and one without a callback. Both forms of the send method return a Future. They both asynchronously sends a record to a topic. The callback gets invoked when the broker has acknowledged the send. The send method is asynchronous and returns right away as soon as the record gets added to the send buffer. The send method allows sending many records at once without blocking for a response from Kafka broker. The result of send method is a RecordMetadata which contains a record’s partition, offset, and timestamp. Callbacks for records sent to the same partition are executed in the order sent.

KafkaProducer send() Exceptions

The Kafka Producer send method can emit the following exceptions:

- InterruptException - If the thread is interrupted while blocked (API).
- SerializationException - If key or value are not valid objects given configured serializers (API).

- TimeoutException - If the time taken for fetching metadata or allocating memory exceeds max.block.ms, or getting acks from Broker exceed timeout.ms, etc. (API)
- KafkaException - If Kafka error occurs not in public API. (API)

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

Kafka Producer: Producer StockSender

```
public class StockSender implements Runnable{
  ...


  try {
    final Future<RecordMetadata> future = producer.send(record);
    if (sentCount % 100 == 0) {displayRecordMetaData(record, future);}
  } catch (InterruptedException e) {
    if (Thread.interrupted()) {
        break;
    }
  } catch (ExecutionException e) {
    logger.error("problem sending record to producer", e);
  }


  ...
  ...
  private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
                                     final Future<RecordMetadata> future)
                              throws InterruptedException, ExecutionException {
    final RecordMetadata recordMetadata = future.get();
    logger.info(String.format("\n\t\t\tkey=%s, value=%s " +
                    "\n\t\t\tsent to topic=%s part=%d off=%d at time=%s",
            record.key(),
            record.value().toJson(),
            recordMetadata.topic(),
            recordMetadata.partition(),
            recordMetadata.offset(),
            new Date(recordMetadata.timestamp())
            ));
  }

}
```

The above shows using the producer send method to send a record.

KafkaProducer flush() method

The Producer flush() method sends all buffered records now (even if linger.ms > 0). It blocks until all requests sent are complete. The flush method is useful when consuming from some input system and pushing data into Kafka as flush() ensures all previously sent messages get sent. This method is useful for marking a completion point after a flush.

KafkaProducer close()

The close() closes producer and frees resources such as connections, threads, and buffers associated with the producer. There are two forms of the close method both block until all previously sent requests complete or a duration passed in as a parameter is exceeded. The close method with no parameters is equivalent to close(Long.MAX_VALUE, TimeUnit.MILLISECONDS). If a producer is unable to complete all requests before the timeout expires, all unsent requests fail, and this method fails.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: Close Producer

```java
package com.cloudurable.kafka.producer;
...
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;

public class StockPriceKafkaProducer {
   ...



    public static void main(String... args) throws Exception {
        //Create Kafka Producer
        final Producer<String, StockPrice> producer = createProducer();
        //Create StockSender list
        final List<StockSender> stockSenders = getStockSenderList(producer);

        //Create a thread pool so every stock sender gets it own.
        // Increase by 1 to fit metrics.
        final ExecutorService executorService = ...;

        //Run each stock sender in its own thread.
        stockSenders.forEach(executorService::submit);


        //Register nice shutdown of thread pool, then flush and close producer.
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            executorService.shutdown();
            try {
                executorService.awaitTermination(200, TimeUnit.MILLISECONDS);
                logger.info("Flushing and closing producer");
                producer.flush();
                producer.close(10_000, TimeUnit.MILLISECONDS);
            } catch (InterruptedException e) {
                logger.warn("shutting down", e);
            }
        }));
    }
}
```

Notice the nice orderly shutdown in the runtime addShutdownHook call.

KafkaProducer partitionsFor() method

The partitionsFor(topic) method returns meta data for partitions `public List<PartitionInfo> partitionsFor(String topic)`. This method is used to get partition metadata for a given topic. This method gets used by producers that do their own partitioning - used for custom partitioning. The PartitionInfo consist of a topic, partition, leader node (Node), replicas nodes (Node[]) and inSyncReplicas nodes. The node consists of id, host, port, and rack.

KafkaProducer metrics() method

The metrics() method is used to get a map of metrics: `public Map<MetricName,? extends Metric> metrics()`. This method is used to get a full set of producer metrics. The MetricName consists of name, group, description, and tags (Map). A metric consist of a Metric name and a value (double).

We created a MetricsProducerReporter which periodically prints out metrics for a Producer. It calls producer.metrics().

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/MetricsProducerReporter.java

Kafka Producer: Producer Metrics

```java
package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.producer.model.StockPrice;
import io.advantageous.boon.core.Sets;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.common.Metric;
import org.apache.kafka.common.MetricName;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Locale;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.stream.Collectors;

public class MetricsProducerReporter implements Runnable {
    private final Producer<String, StockPrice> producer;
    private final Logger logger =
            LoggerFactory.getLogger(MetricsProducerReporter.class);

    //Used to Filter just the metrics we want
    private final Set<String> metricsNameFilter = Sets.set(
            "record-queue-time-avg", "record-send-rate", "records-per-request-avg",
            "request-size-max", "network-io-rate", "record-queue-time-avg",
            "incoming-byte-rate", "batch-size-avg", "response-rate", "requests-in-flight"
    );

    public MetricsProducerReporter(
            final Producer<String, StockPrice> producer) {
        this.producer = producer;
    }

    @Override
    public void run() {
        while (true) {
            final Map<MetricName, ? extends Metric> metrics
                    = producer.metrics();

            displayMetrics(metrics);
            try {
                Thread.sleep(3_000);
            } catch (InterruptedException e) {
                logger.warn("metrics interrupted");
                Thread.interrupted();
                break;
            }
        }
    }


    static class MetricPair {
        private final MetricName metricName;
        private final Metric metric;
        MetricPair(MetricName metricName, Metric metric) {
            this.metricName = metricName;
            this.metric = metric;
        }
        public String toString() {
            return metricName.group() + "." + metricName.name();
        }
    }

    private void displayMetrics(Map<MetricName, ? extends Metric> metrics) {
        final Map<String, MetricPair> metricsDisplayMap = metrics.entrySet().stream()
                //Filter out metrics not in metricsNameFilter
                .filter(metricNameEntry ->
                        metricsNameFilter.contains(metricNameEntry.getKey().name()))
                //Filter out metrics not in metricsNameFilter
                .filter(metricNameEntry ->
                        !Double.isInfinite(metricNameEntry.getValue().value()) &&
                                !Double.isNaN(metricNameEntry.getValue().value()) &&
                                metricNameEntry.getValue().value() != 0
                )
                //Turn Map<MetricName,Metric> into TreeMap<String, MetricPair>
                .map(entry -> new MetricPair(entry.getKey(), entry.getValue()))
                .collect(Collectors.toMap(
                        MetricPair::toString, it -> it, (a, b) -> a, TreeMap::new
                ));
        //Output metrics
        final StringBuilder builder = new StringBuilder(255);
        builder.append("\n---------------------------------------\n");
        metricsDisplayMap.entrySet().forEach(entry -> {
            MetricPair metricPair = entry.getValue();
            String name = entry.getKey();
            builder.append(String.format(Locale.US, "%50s%25s\t\t%,-10.2f\t\t%s\n",
                    name,
                    metricPair.metricName.name(),
                    metricPair.metric.value(),
                    metricPair.metricName.description()
            ));
        });
        builder.append("\n---------------------------------------\n");
        logger.info(builder.toString());
    }


}
```

The MetricsProducerReporter would output the following metrics.

Output for MetricsProducerReporter

```
Metric   producer-metrics,       record-queue-time-max,  508.0,
                The maximum time in ms record batches spent in the record accumulator.

17:09:22.721 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric   producer-node-metrics,  request-rate,   0.025031289111389236,
                The average number of requests sent per second.

17:09:22.721 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric   producer-metrics,       records-per-request-avg,        205.55263157894737,
                The average number of records per request.

17:09:22.722 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric   producer-metrics,       record-size-avg,        71.02631578947368,
                The average record size

17:09:22.722 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric   producer-node-metrics,  request-size-max,       56.0,
                The maximum size of any request sent in the window.

17:09:22.723 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric   producer-metrics,       request-size-max,       12058.0,
                The maximum size of any request sent in the window.

17:09:22.723 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric   producer-metrics,       compression-rate-avg,   0.41441360272859273,
                The average compression rate of record batches.
```

Review: Metrics, orderly shutdown, producer partitioner

- How can you access Metrics for the Kafka Producer?
- How can you shut down a producer orderly?
- When might you use an producer interceptor?
- When might you use a producer partitioner?

- How can you access Metrics for the Kafka Producer? Call metrics() on the producer.

- How can you shut down a producer orderly? Call close.

- When might you use an producer interceptor? If you need to modify the record before sending it. (Add extra headers, etc.)
- When might you use a producer partitioner? If you want to implement some sort of custom partitioning like a priority queue.

Lab Creating an advanced Kafka Producer

Stock Price Producer

The Stock Price Producer example has the following classes:

- StockPrice - holds a stock price has a name, dollar, and cents
- StockPriceKafkaProducer - Configures and creates KafkaProducer, StockSender list, ThreadPool (ExecutorService), starts StockSender runnable into thread pool
- StockAppConstants - holds topic and broker list
- StockPriceSerializer - can serialize a StockPrice into byte[]
- StockSender - generates somewhat random stock prices for a given StockPrice name, Runnable, 1 thread per StockSender and shows using KafkaProducer from many threads

StockPrice

The StockPrice is a simple domain object that holds a stock price has a name, dollar, and cents. The StockPrice knows how to convert itself into a JSON string.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/model/StockPrice.java

Kafka Producer: StockPrice

```
package com.cloudurable.kafka.producer.model;

import io.advantageous.boon.json.JsonFactory;

public class StockPrice {

    private final int dollars;
    private final int cents;
    private final String name;

    public StockPrice(final String json) {
        this(JsonFactory.fromJson(json, StockPrice.class));
    }


    public StockPrice() {
        dollars = 0;
        cents = 0;
        name = "";
    }

    public StockPrice(final String name, final int dollars, final int cents) {
        this.dollars = dollars;
        this.cents = cents;
        this.name = name;
    }


    public StockPrice(final StockPrice stockPrice) {
        this.cents = stockPrice.cents;
        this.dollars = stockPrice.dollars;
        this.name = stockPrice.name;
    }


    public int getDollars() {
        return dollars;
    }
    public int getCents() {
        return cents;
    }


    public String getName() {
        return name;
    }


    @Override
    public String toString() {
        return "StockPrice{" +
                "dollars=" + dollars +
                ", cents=" + cents +
                ", name='" + name + '\'' +
                '}';
    }


    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        StockPrice that = (StockPrice) o;

        if (dollars != that.dollars) return false;
        if (cents != that.cents) return false;
        return name != null ? name.equals(that.name) : that.name == null;
    }

    @Override
    public int hashCode() {
        int result = dollars;
        result = 31 * result + cents;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        return result;
    }
    public String toJson() {
        return "{" +
                "\"dollars\": " + dollars +
                ", \"cents\": " + cents +
                ", \"name\": \"" + name + '\"' +
                '}';
    }
}
```

StockPrice is just a POJO.

StockPriceKafkaProducer

StockPriceKafkaProducer import classes and sets up a logger. It has a createProducer method to create a KafkaProducer instance. It has a setupBootstrapAndSerializers to initialize bootstrap servers, client id, key serializer and custom serializer (StockPriceSerializer). It has a main() method that creates the producer, creates a StockSender list passing each instance the producer, and it creates a thread pool, so every stock sender gets it own thread, and then it runs each stockSender in its own thread using the thread pool.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer imports, createProducer

```java
package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.StockAppConstants;
import com.cloudurable.kafka.producer.model.StockPrice;
import io.advantageous.boon.core.Lists;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.List;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        return new KafkaProducer<>(props);
    }
    ...
}
```

The above code imports Kafka classes and sets up the logger and calls createProducer to create a KafkaProducer. The createProducer() calls setupBoostrapAndSerializers().

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer imports, createProducer

```
public class StockPriceKafkaProducer {
  private static void setupBootstrapAndSerializers(Properties props) {
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            StockAppConstants.BOOTSTRAP_SERVERS);
    props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());


    //Custom Serializer - config "value.serializer"
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StockPriceSerializer.class.getName());

  }
}
```

The setupBootstrapAndSerializers method initializes bootstrap servers, client id, key serializer and custom serializer (StockPriceSerializer). The StockPriceSerializer will serialize StockPrice into bytes.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer.main - start thread pool

```java
public class StockPriceKafkaProducer {
  ...

  public static void main(String... args) throws Exception {
    //Create Kafka Producer
    final Producer<String, StockPrice> producer = createProducer();
    //Create StockSender list
    final List<StockSender> stockSenders = getStockSenderList(producer);

    //Create a thread pool so every stock sender gets it own.
    // Increase by 1 to fit metrics.
    final ExecutorService executorService =
            Executors.newFixedThreadPool(stockSenders.size() + 1);

    //Run Metrics Producer Reporter which is runnable passing it the producer.
    executorService.submit(new MetricsProducerReporter(producer));

    //Run each stock sender in its own thread.
    stockSenders.forEach(executorService::submit);

  }
  ...
}
```

The StockPriceKafkaProducer main method creates a Kafka producer, then creates StockSender list passing each instance the producer. It then creates a thread pool (executorService) and runs each StockSender, which is runnable, in its own thread from the thread pool.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer.getStockSenderList - create list of StockSenders

```
    private static List<StockSender> getStockSenderList(
            final Producer<String, StockPrice> producer) {
        return Lists.list(
                new StockSender(StockAppConstants.TOPIC,
                        new StockPrice("IBM", 100, 99),
                        new StockPrice("IBM", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("SUN", 100, 99),
                        new StockPrice("SUN", 50, 10),
                        producer,
                        1, 10
                ),
                ...,
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("FFF", 100, 99),
                        new StockPrice("FFF", 50, 10),
                        producer,
                        1, 10
                )
        );

    }
```

The getStockSenderList of StockPriceKafkaProducer just creates a list of StockSenders.

StockPriceSerializer

The StockPriceSerializer converts a StockPrice into a byte array.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceSerializer.java

Kafka Producer: StockPriceSerializer - convert StockPrice into a byte array

```
package com.cloudurable.kafka.producer;
import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.common.serialization.Serializer;
import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockPriceSerializer implements Serializer<StockPrice> {

    @Override
    public byte[] serialize(String topic, StockPrice data) {
        return data.toJson().getBytes(StandardCharsets.UTF_8);
    }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close() {
    }
}
```

Notice the StockPriceSerializer converts a StockPrice into a byte array by calling StockPrice.toJson.

StockAppConstants

The StockAppConstants defines a few constants, namely, topic name and a comma delimited list of bootstrap Kafka brokers.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/StockAppConstants.java

Kafka Producer: StockAppConstants defines constants

```
package com.cloudurable.kafka;

public class StockAppConstants {
    public final static String TOPIC = "stock-prices";
    public final static String BOOTSTRAP_SERVERS =
            "localhost:9092,localhost:9093,localhost:9094";

}
```

StockSender

The StockSender uses the Kafka Producer we created earlier. The StockSender generates random stock prices for a given StockPrice name. The StockSender is Runnable and runs in its own thread. There is one thread per StockSender. The StockSender is used to show using KafkaProducer from many threads. The StockSender Delays random time between delayMin and delayMax, then sends a random StockPrice between stockPriceHigh and stockPriceLow.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

Kafka Producer: StockSender imports, Runnable

```
package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Date;
import java.util.Random;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class StockSender implements Runnable{
  ...
}
```

The StockSender imports Kafka Producer, ProducerRecord, RecordMetadata, and StockPrice. It implements Runnable, and can be submitted to ExecutionService (thread pool).

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

Kafka Producer: StockSender fields

```
public class StockSender implements Runnable{

      private final StockPrice stockPriceHigh;
      private final StockPrice stockPriceLow;
      private final Producer<String, StockPrice> producer;
      private final int delayMinMs;
      private final int delayMaxMs;
      private final Logger logger = LoggerFactory.getLogger(StockSender.class);
      private final String topic;

      public StockSender(final String topic, final StockPrice stockPriceHigh,
                         final StockPrice stockPriceLow,
                         final Producer<String, StockPrice> producer,
                         final int delayMinMs,
                         final int delayMaxMs) {
          this.stockPriceHigh = stockPriceHigh;
          this.stockPriceLow = stockPriceLow;
          this.producer = producer;
          this.delayMinMs = delayMinMs;
          this.delayMaxMs = delayMaxMs;
          this.topic = topic;
      }
}
```

The StockSender takes a topic, high & low stockPrice, producer, and delay min & max.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

Kafka Producer: StockSender run method

```
public class StockSender implements Runnable{

  ...
  public void run() {
    final Random random = new Random(System.currentTimeMillis());
    int sentCount = 0;

    while (true) {
        sentCount++;
        final ProducerRecord <String, StockPrice> record =
                                    createRandomRecord(random);
        final int delay = randomIntBetween(random, delayMaxMs, delayMinMs);

        try {
            final Future<RecordMetadata> future = producer.send(record);
            if (sentCount % 100 == 0) {displayRecordMetaData(record, future);}
            Thread.sleep(delay);
        } catch (InterruptedException e) {
            if (Thread.interrupted()) {
                break;
            }
        } catch (ExecutionException e) {
            logger.error("problem sending record to producer", e);
        }
    }
  }
}
```

The StockSender run methods in a forever loop creates random record, sends the record, waits random time, and then repeats.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

Kafka Producer: StockSender createRandomRecord

```
public class StockSender implements Runnable{

  ...
  private final int randomIntBetween(final Random random,
                                     final int max,
                                     final int min) {
    return random.nextInt(max - min + 1) + min;
  }

  private ProducerRecord<String, StockPrice> createRandomRecord(
            final Random random) {

    final int dollarAmount = randomIntBetween(random,
            stockPriceHigh.getDollars(), stockPriceLow.getDollars());

    final int centAmount = randomIntBetween(random,
            stockPriceHigh.getCents(), stockPriceLow.getCents());

    final StockPrice stockPrice = new StockPrice(
            stockPriceHigh.getName(), dollarAmount, centAmount);

    return new ProducerRecord<>(topic, stockPrice.getName(),
            stockPrice);
  }
}
```

The StockSender createRandomRecord method uses randomIntBetween. The createRandomRecord creates StockPrice and then wraps StockPrice in ProducerRecord.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

Kafka Producer: StockSender displayRecordMetaData

```
public class StockSender implements Runnable{

  ...
  private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
                                     final Future<RecordMetadata> future)
                              throws InterruptedException, ExecutionException {
    final RecordMetadata recordMetadata = future.get();
    logger.info(String.format("\n\t\t\tkey=%s, value=%s " +
                    "\n\t\t\tsent to topic=%s part=%d off=%d at time=%s",
            record.key(),
            record.value().toJson(),
            recordMetadata.topic(),
            recordMetadata.partition(),
            recordMetadata.offset(),
            new Date(recordMetadata.timestamp())
            ));
  }
  ...
}
```

Every 100 records StockSender displayRecordMetaData method gets called, which prints out record info, and recordMetadata info: key, JSON value, topic, partition, offset, time. The displayRecordMetaData uses the Future from the call to producer.send().

Running the example

To run the example, you need to run ZooKeeper, then run the three Kafka Brokers. Once that is running, you will need to run create-topic.sh. And lastly run the StockPriceKafkaProducer from the IDE.

First run ZooKeeper.

Running ZooKeeper with run-zookeeper.sh (Run in a new terminal)

```
~/kafka-training

$ cat run-zookeeper.sh
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/zookeeper-server-start.sh \
   kafka/config/zookeeper.properties

$ ./run-zookeeper.sh
```

Now run the first Kafka Broker.

Running the 1st Kafka Broker (Run in a new terminal)

```
~/kafka-training/lab5

$ cat bin/start-1st-server.sh
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-0.properties"

$ bin/start-1st-server.sh
```

Now run the second Kafka Broker.

Running the 2nd Kafka Broker (Run in a new terminal)

```
~/kafka-training/lab5

$ cat bin/start-2nd-server.sh
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-1.properties"

$ bin/start-2nd-server.sh
```

Now run the third Kafka Broker.

Running the 3rd Kafka Broker (Run in a new terminal)

```
~/kafka-training/lab5

$ cat bin/start-3rd-server.sh
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-2.properties"

$ bin/start-3rd-server.sh
```

Once all brokers are running, run create-topic.sh as follows.

Running create topic

```
~/kafka-training/lab5

$ cat bin/create-topic.sh
#!/usr/bin/env bash

cd ~/kafka-training

kafka/bin/kafka-topics.sh \
    --create \
    --zookeeper localhost:2181 \
    --replication-factor 3 \
    --partitions 3 \
    --topic stock-prices \
    --config min.insync.replicas=2

$ bin/create-topic.sh
    Created topic "stock-prices".
```

The create-topics script creates a topic. The name of the topic is stock-prices. The topic has three partitions. The created topic has a replication factor of three.

For the config only the broker id and log directory changes.

config/server-0.properties

```
broker.id=0
port=9092
log.dirs=./logs/kafka-0
...
```

Run the StockPriceKafkaProducer from your IDE. You should see log messages from StockSender(s) with StockPrice name, JSON value, partition, offset, and time.

Lab Adding an orderly shutdown flush and close

Shutdown Producer Nicely

Let’s write some code to shut the Producer down nicely.

The shutdown code will happen if you are running the producer example from a terminal and type ctrl-C so shutdown from Java occurs. We will write some code to shutdown thread pool and wait. Then we will flush the producer to send any outstanding batches if using batches (producer.flush()). Lastly, we will close the producer using producer.close and wait five seconds for the producer to shutdown. (Note that closing the producer also flushes it.)

To this we will add shutdown hook to Java runtime. Then to test we will start the StockPriceKafkaProducer, and then you can stop it using CTRL-C or by pressing the stop button in your IDE.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer shutdown hook for clean shutdown

```
public class StockPriceKafkaProducer {

    ...

    private static final Logger logger = LoggerFactory.getLogger(StockPriceKafkaProducer.class);



    public static void main(String... args) throws Exception {
        //Create Kafka Producer
        final Producer<String, StockPrice> producer = createProducer();
        //Create StockSender list
        final List<StockSender> stockSenders = getStockSenderList(producer);

        //Create a thread pool so every stock sender gets it own.
        // Increase by 1 to fit metrics.
        final ExecutorService executorService =
                Executors.newFixedThreadPool(stockSenders.size() );

        //Run each stock sender in its own thread.
        stockSenders.forEach(executorService::submit);


        //Register nice shutdown of thread pool, then flush and close producer.
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            executorService.shutdown();
            try {
                executorService.awaitTermination(200, TimeUnit.MILLISECONDS);
                logger.info("Flushing and closing producer");
                producer.flush();
                producer.close(10_000, TimeUnit.MILLISECONDS);
            } catch (InterruptedException e) {
                logger.warn("shutting down", e);
            }
        }));
    }

...
}
```

Notice we add a shutdown hook using Runtime.getRuntime().addShutdownHook and this shutdown hook that shuts down the thread pool, then calls flush on the producer and then closes the producer whilst waiting 10 seconds for the close to happen.

Lab Configuring Producer Durability

In this lab we will configure producer durability.

Set default acks to all

Change StockPriceKafkaProducer and set Producer config acks to all (this is the default). This means that all ISRs in-sync replicas have to respond for producer write to go through.

```
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                createProducer() {
        final Properties props = new Properties();
        ...

        //Set number of acknowledgments - acks - default is all
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        return new KafkaProducer<>(props);
    }
}
```

Notice we set the Producer config property “acks” (ProducerConfig.ACKS_CONFIG) to all.

Kafka Broker Config for Min.Insync.Replicas

Next let’s set the the min.insync.replicas to three. This will force at least three in-sync replicas (ISRs) have to respond for our producer to get an ack from the Kafka Broker. NOTE: We have three brokers in this lab, thus all three have to be up for the Producer to work.

Ensure min.insync.replicas is set to three in all of the broker config files (server-0.properties, server-1.properties and server-2.properties).

Run this lab. Run it. Run Servers. Run Producer. Kill 1 Broker.

If not already, startup ZooKeeper. Now startup three Kafka brokers (or ensure they are running) using scripts described earlier. From the IDE run StockPriceKafkaProducer class. From the terminal kill one of the Kafka Brokers. Now look at the logs for the StockPriceKafkaProducer, you should see Caused by: org.apache.kafka.common.errors.NotEnoughReplicasException. Note that the Messages are rejected since there are fewer in-sync replicas than required. Repeat this with only 2 min.insync.replicas set. (Change config and restart brokers and restart producer). Observe the behavior of using 2 for min.insync.replicas vs. three.

Why did the send fail?

The producer config ProducerConfig.ACKS_CONFIG (acks config for producer) was set to “all”. This settings expects leader to only give successful ack after all followers ack the send was written to their log. The Broker config min.insync.replicas set to 3. At least three in-sync replicas must respond before send is considered successful. Since we took one broker out and only had three to begin with, it forces the send to fail since the send can not get three acks from ISRs.

Modify durability to leader only

Change StockPriceKafkaProducer acks config to 1 `props.put(ProducerConfig.ACKS_CONFIG, “1"` , i.e., leader sends ack after write to log. From the IDE run StockPriceKafkaProducer again. From the terminal kill one of the Kafka Brokers. Notice that the StockPriceKafkaProducer now runs normally.

Why did the send not fail for acks 1?

Setting the Producer config ProducerConfig.ACKS_CONFIG (acks config for producer) to “1”. This setting expects leader to only give successful ack after it writes to its log. Replicas still get replication but leader does not wait for replication to send ack. Broker Config min.insync.replicas is still set to 3, but this config only gets looked at if acks=“all”.

Running describe topics before and after stoping broker

Try the last steps again. Stop a server while producer is running. Then run describe-topics.sh. Then Rerun server you stopped and run describe-topics.sh again. Observe the changes to ISRs and partition to broker ownership.

bin/describe-topics.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-topics.sh --describe \
    --topic stock-prices \
    --zookeeper localhost:2181
```

The script bin/describe-topics.sh calls kafka-topics.sh to describe the topic layout with regards to brokers and partitions.

Retry with acks = 0

Run the last example again (servers, and producer), but this time set acks to 0. Run all three brokers then take one away. Then take another broker away. Try Run describe-topics. Take all of the brokers down and continue to run the producer. What do you think happens? When you are done, change acks back to acks=all.

Lab Review

Look at the following listing.

Describe topic listing

```
$ bin/describe-topics.sh
Topic:stock-prices      PartitionCount:3        ReplicationFactor:3     Configs:min.insync.replicas=2
        Topic: stock-prices     Partition: 0    Leader: 2       Replicas: 1,2,0 Isr: 2
        Topic: stock-prices     Partition: 1    Leader: 2       Replicas: 2,0,1 Isr: 2
        Topic: stock-prices     Partition: 2    Leader: 2       Replicas: 0,1,2 Isr: 2
```

- How would you describe the above?
- How many servers are likely running out of the three?
- Would the producer still run with acks=all? Why or Why not?
- Would the producer still run with acks=1? Why or Why not?
- Would the producer still run with acks=0? Why or Why not?
- Which broker is the leader of partition 1?

- How would you describe the above? Two servers are down. Broker 0 and Broker 1.

- How many servers are likely running out of the three? Just the third broker.
- Would the producer still run with acks=all? Why or Why not? No. Only one server is running.
- Would the producer still run with acks=1? Why or Why not? No. Yes. The 3rd server owns the partitions.
- Would the producer still run with acks=0? Why or Why not? Yes.
- Which broker is the leader of partition 1? The third server one with broker id 2.

Lab Adding Producer Metrics and Replication Verification

The objectives of this lab is to setup Kafka producer metrics and use the replication verification command line tool.

To do this you will change the *min.insync.replicas* for broker and observer metrics and replication verification and then change *min.insync.replicas* for topic and observer metrics and replication verification.

Create Producer Metrics Monitor

As part of this lab we will create a class called `MetricsProducerReporter` that is `Runnable`. The `MetricsProducerReporter` gets passed a Kafka Producer, and the `MetricsProducerReporter` calls the `producer.metrics()` method every 10 seconds in a while loop from run method, and prints out the MetricName and Metric value. The `StockPriceKafkaProducer` main method submits `MetricsProducerReporter` to the `ExecutorService` (thread pool).

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/MetricsProducerReporter.java

Kafka Producer: MetricsProducerReporter for reporting metrics is Runnable

```java
package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.producer.model.StockPrice;
import io.advantageous.boon.core.Sets;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.common.Metric;
import org.apache.kafka.common.MetricName;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Locale;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.stream.Collectors;

public class MetricsProducerReporter implements Runnable {
    private final Producer<String, StockPrice> producer;
    private final Logger logger =
            LoggerFactory.getLogger(MetricsProducerReporter.class);

    //Used to Filter just the metrics we want
    private final Set<String> metricsNameFilter = Sets.set(
            "record-queue-time-avg", "record-send-rate", "records-per-request-avg",
            "request-size-max", "network-io-rate", "record-queue-time-avg",
            "incoming-byte-rate", "batch-size-avg", "response-rate", "requests-in-flight"
    );

    public MetricsProducerReporter(
            final Producer<String, StockPrice> producer) {
        this.producer = producer;
    }
}
```

Note that the `MetricsProducerReporter` is `Runnable` and it will get submitted to the `ExecutorService` (thread pool). Note that the `MetricsProducerReporter` gets passed a Kafka Producer, which it will call to report metrics.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/MetricsProducerReporter.java

Kafka Producer: MetricsProducerReporter calls producer.metrics()

```
package com.cloudurable.kafka.producer;
...

public class MetricsProducerReporter implements Runnable {
  ...
@Override
public void run() {
    while (true) {
        final Map<MetricName, ? extends Metric> metrics
                = producer.metrics();

        displayMetrics(metrics);
        try {
            Thread.sleep(3_000);
        } catch (InterruptedException e) {
            logger.warn("metrics interrupted");
            Thread.interrupted();
            break;
        }
    }
}

private void displayMetrics(Map<MetricName, ? extends Metric> metrics) {
    final Map<String, MetricPair> metricsDisplayMap = metrics.entrySet().stream()
            //Filter out metrics not in metricsNameFilter
            .filter(metricNameEntry ->
                    metricsNameFilter.contains(metricNameEntry.getKey().name()))
            //Filter out metrics not in metricsNameFilter
            .filter(metricNameEntry ->
                    !Double.isInfinite(metricNameEntry.getValue().value()) &&
                            !Double.isNaN(metricNameEntry.getValue().value()) &&
                            metricNameEntry.getValue().value() != 0
            )
            //Turn Map<MetricName,Metric> into TreeMap<String, MetricPair>
            .map(entry -> new MetricPair(entry.getKey(), entry.getValue()))
            .collect(Collectors.toMap(
                    MetricPair::toString, it -> it, (a, b) -> a, TreeMap::new
            ));
    //Output metrics
    final StringBuilder builder = new StringBuilder(255);
    builder.append("\n---------------------------------------\n");
    metricsDisplayMap.entrySet().forEach(entry -> {
        MetricPair metricPair = entry.getValue();
        String name = entry.getKey();
        builder.append(String.format(Locale.US, "%50s%25s\t\t%,-10.2f\t\t%s\n",
                name,
                metricPair.metricName.name(),
                metricPair.metric.value(),
                metricPair.metricName.description()
        ));
    });
    builder.append("\n---------------------------------------\n");
    logger.info(builder.toString());
}
...
}
```

The run method calls `producer.metrics()` every 10 seconds in a while loop, and print out `MetricName` and `Metric` value. It only prints out the names that are in metricsNameFilter.

Notice we are using Java 8 Stream to filter and sort metrics. We get rid of metric values that are NaN, infinite numbers and 0s. Then we sort map by converting it to TreeMap. The MetricPair is helper class that has a Metric and a MetricName. Then we give it a nice format so we can read metrics easily and use so some space and some easy indicators to find the metrics in the log.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer adds MetricsProducerReporter to executorService

```java
public class StockPriceKafkaProducer {
...

  public static void main(String... args) throws Exception {
      //Create Kafka Producer
      final Producer<String, StockPrice> producer = createProducer();
      //Create StockSender list
      final List<StockSender> stockSenders = getStockSenderList(producer);

      //Create a thread pool so every stock sender gets it own.
      // Increase by 1 to fit metrics.
      final ExecutorService executorService =
              Executors.newFixedThreadPool(stockSenders.size() + 1);

      //Run Metrics Producer Reporter which is runnable passing it the producer.
      executorService.submit(new MetricsProducerReporter(producer));

      //Run each stock sender in its own thread.
      stockSenders.forEach(executorService::submit);
      ...
  }
...
}
```

The main method of StockPriceKafkaProducer increases thread pool size by 1 to fit metrics reporting. Then submits a new instance of MetricsProducerReporter to the ExecutorService. The new MetricsProducerReporter is passed the producer from createProducer.

Run it. Run Servers. Run Producer.

If not already, startup ZooKeeper as before and then startup the three Kafka brokers using scripts described earlier. Then from the IDE run StockPriceKafkaProducer (ensure acks are set to all first). Observe metrics which print out every ten seconds.

Expected output from MetricsProducerReporter

```
---------------------------------------
    producer-metrics.batch-size-avg               batch-size-avg                         9,030.78              The average numbe
r of bytes sent per partition per-request.
    producer-metrics.incoming-byte-rate           incoming-byte-rate                 63.28                     Bytes/second read
off all sockets
    producer-metrics.network-io-rate              network-io-rate                      1.32                    The average numbe
r of network operations (reads or writes) on .
    producer-metrics.record-queue-time-avg        record-queue-time-avg            3,008.97            The average time in ms re
cord batches spent in the..
    producer-metrics.record-send-rate             record-send-rate                     485.76                  The average numbe
r of records sent per second.

    producer-metrics.records-per-request-avg      records-per-request-avg              737.03                  The average numbe
r of records per request.
    producer-metrics.request-size-max             request-size-max                     46,064.00               The maximum size 
of any request sent in the window.
    producer-metrics.response-rate                response-rate                          0.66                  Responses receive
d sent per second.
    producer-node-metrics.incoming-byte-rate      incoming-byte-rate                 127.41
    producer-node-metrics.request-size-max        request-size-max                     46,673.00               The maximum size 
of any request sent in the window.
    producer-node-metrics.response-rate           response-rate                          1.32                  The average numbe
r of responses received per second.
    producer-topic-metrics.record-send-rate       record-send-rate                     960.73

---------------------------------------
```

Use replication verification to observe replicas getting behind

Next we will use replica-verification.sh to show max lag between replicas increase after we stop one of the brokers.

Running replica-verification.sh while killing one broker while running StockPriceKafkaProducer

```
## In ~/kafka-training/lab5

$ cat bin/replica-verification.sh
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-replica-verification.sh  \
    --report-interval-ms 5000 \
    --topic-white-list  "stock-prices.*" \
    --broker-list localhost:9092,localhost:9093,localhost:9094

## Run replica-verification to show max lag
$ bin/replica-verification.sh
2017-07-05 10:50:15,319: verification process is started.
2017-07-05 10:50:21,062: max lag is 0 for partition [stock-prices,0] at offset 29559 among 3 partitions
2017-07-05 10:50:27,479: max lag is 0 for partition [stock-prices,0] at offset 29559 among 3 partitions
2017-07-05 10:50:32,974: max lag is 0 for partition [stock-prices,0] at offset 29559 among 3 partitions
2017-07-05 10:50:39,672: max lag is 1327 for partition [stock-prices,0] at offset 29559 among 3 partitions
2017-07-05 10:50:45,520: max lag is 7358 for partition [stock-prices,0] at offset 29559 among 3 partitions
2017-07-05 10:50:50,871: max lag is 13724 for partition [stock-prices,0] at offset 29559 among 3 partitions
2017-07-05 10:50:57,449: max lag is 19122 for partition [stock-prices,0] at offset 29559 among 3 partitions
```

While running `StockPriceKafkaProducer` from the command line, we kill one of the Kafka brokers and watch the max lag increase.

Running replica-verification.sh after killing one broker while running StockPriceKafkaProducer

```
# In ~/kafka-training/lab5
$ cat bin/describe-topics.sh
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-topics.sh --describe \
    --topic stock-prices \
    --zookeeper localhost:2181


$ bin/describe-topics.sh
Topic:stock-prices      PartitionCount:3        ReplicationFactor:3     Configs:min.insync.replicas=2
 Topic: stock-prices    Partition: 0    Leader: 1       Replicas: 1,2,0 Isr: 2,1
 Topic: stock-prices    Partition: 1    Leader: 2       Replicas: 2,0,1 Isr: 2,1
 Topic: stock-prices    Partition: 2    Leader: 1       Replicas: 0,1,2 Isr: 2,1
```

Notice that one of the servers are down and one brokers owns two partitions.

Change min.insync.replicas

Now stop all Kafka Brokers (Kafka servers). Then change `min.insync.replicas=3` to `min.insync.replicas=2`. Make this change in all of the configuration files for the brokers under the config directory of the lab (config/server-0.properties, config/server-1.properties, and config/server-2.properties). The config files for the brokers are in lab directory under config. After you are done, restart Zookeeper if needed and then restart the servers. Try starting and stopping different Kafka Brokers while StockPriceKafkaProducer is running. Be sure to observe metrics, and observe any changes. Also run replication verification utility in one terminal while checking topics stats in another with describe-topics.sh in another terminal.

Expected output from changing min.insync.replicas

The Producer will work even if one broker goes down. The Producer will not work if two brokers go down because min.insync.replicas=2, thus two replicas have to be up besides leader. Since the Producer can run with 1 down broker, notice that the replication lag can get really far behind.

Once you are done, change it back

Shutdown all brokers, change the all the broker config back to min.insync.replicas=3 (broker config for servers). Restart the brokers.

Change min.insync.replicas at the topic level

Modify bin/create-topic.sh and add add –config min.insync.replicas=2 Add this as param to kafka-topics.sh. Now run bin/delete-topic.sh and then run bin/create-topic.sh.

bin/create-topic.sh after modification

```
$ cat bin/create-topic.sh
#!/usr/bin/env bash

cd ~/kafka-training

kafka/bin/kafka-topics.sh \
    --create \
    --zookeeper localhost:2181 \
    --replication-factor 3 \
    --partitions 3 \
    --topic stock-prices \
    --config min.insync.replicas=2
```

Run delete topic and then run create topic as follows.

running delete topic and then create topic.

```
$ bin/delete-topic.sh
Topic stock-prices is marked for deletion.

$ bin/create-topic.sh
Created topic "stock-prices".
```

Run it with the new topics.

Now stop all Kafka Brokers (Kafka servers) and startup ZooKeeper if needed and the three Kafka brokers, Run StockPriceKafkaProducer (ensure acks are set to all first). Start and stop different Kafka Brokers while StockPriceKafkaProducer runs as before. Observe metrics, and observe any changes, Also Run replication verification in one terminal and check topics stats in another with describe-topics.sh in another terminal.

Expected results of changing topic to use min.insync.replicas

The min.insync.replicas on the Topic config overrides the min.insync.replicas on the Broker config. In this setup, you can survive a single node failure but not two (output below is recovery).

Lab Batching Records

Objectives is to understand Kafka Batching. You will disable batching and observer metrics, then you will reenable batching and observe metrics. Next you increase batch size and linger and observe metrics. In this lab, we will run a consumer to see batch sizes change from a consumer perspective as we change batching on the producer side. Lastly you will enable compression, and then observe results.

SimpleStockPriceConsumer

We added a SimpleStockPriceConsumer to consume StockPrices and display batch lengths for poll(). We won't cover the consumer in detail just quickly, since this is a Producer lab not a Consumer lab. You will run this consumer while you are running the StockPriceKafkaProducer. While you are running SimpleStockPriceConsumer with various batch and linger config, observe output of Producer metrics and StockPriceKafkaProducer output.

…

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Producer: SimpleStockPriceConsumer to consumer records

```
package com.cloudurable.kafka.consumer;
import com.cloudurable.kafka.StockAppConstants;
import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.StringDeserializer;
...

public class SimpleStockPriceConsumer {

    private static Consumer<String, StockPrice> createConsumer() {
        final Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                StockAppConstants.BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
                "KafkaExampleConsumer");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName());
        //Custom Deserializer
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StockDeserializer.class.getName());
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
        // Create the consumer using props.
        final Consumer<String, StockPrice> consumer =
                new KafkaConsumer<>(props);
        // Subscribe to the topic.
        consumer.subscribe(Collections.singletonList(
                StockAppConstants.TOPIC));
        return consumer;
    }
    ...
}
```

The SimpleStockPriceConsumer is similar to other Consumer examples we have covered so far. SimpleStockPriceConsumer subscribes to stock-prices topic and uses a custom serializer (StockDeserializer).

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Producer: SimpleStockPriceConsumer runConsumer

```java
package com.cloudurable.kafka.consumer;
...

public class SimpleStockPriceConsumer {
    ...

    static void runConsumer() throws InterruptedException {
        final Consumer<String, StockPrice> consumer = createConsumer();
        final Map<String, StockPrice> map = new HashMap<>();
        try {
            final int giveUp = 1000; int noRecordsCount = 0;
            int readCount = 0;
            while (true) {
                final ConsumerRecords<String, StockPrice> consumerRecords =
                        consumer.poll(1000);
                if (consumerRecords.count() == 0) {
                    noRecordsCount++;
                    if (noRecordsCount > giveUp) break;
                    else continue;
                }
                readCount++;
                consumerRecords.forEach(record -> {
                    map.put(record.key(), record.value());
                });
                if (readCount % 100 == 0) {
                    displayRecordsStatsAndStocks(map, consumerRecords);
                }
                consumer.commitAsync();
            }
        }
        finally {
            consumer.close();
        }
        System.out.println("DONE");
    }
    ...
    public static void main(String... args) throws Exception {
      runConsumer();
    }
    ...
}
```

The run method drains Kafka topic. It creates map of current stocks prices, and Calls displayRecordsStatsAndStocks().

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Producer: SimpleStockPriceConsumer displayRecordsStatsAndStocks

```
package com.cloudurable.kafka.consumer;
...

public class SimpleStockPriceConsumer {
    ...
    private static void displayRecordsStatsAndStocks(
        final Map<String, StockPrice> stockPriceMap,
        final ConsumerRecords<String, StockPrice> consumerRecords) {
    System.out.printf("New ConsumerRecords par count %d count %d\n",
            consumerRecords.partitions().size(),
            consumerRecords.count());
    stockPriceMap.forEach((s, stockPrice) ->
            System.out.printf("ticker %s price %d.%d \n",
                stockPrice.getName(),
                stockPrice.getDollars(),
                stockPrice.getCents()));
    System.out.println();
  }

  ...
}
```

The displayRecordsStatsAndStocks method prints out size of each partition read and total record count. Then it prints out each stock at its current price.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/consumer/StockDeserializer.java

Kafka Producer: StockDeserializer

```
package com.cloudurable.kafka.consumer;

import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.common.serialization.Deserializer;

import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockDeserializer implements Deserializer<StockPrice> {

    @Override
    public StockPrice deserialize(final String topic, final byte[] data) {
        return new StockPrice(new String(data, StandardCharsets.UTF_8));
    }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close() {
    }
}
```

The StockDeserializer is used to deserialize StockPrice objects from the Kafka topic.

Disable batching for the Producer

Let's start by disabling batching in the StockPriceKafkaProducer. Setting `props.put(ProducerConfig.BATCH_SIZE_CONFIG, 0)` turns batching off. After you do this rerun StockPriceKafkaProducer and check Consumer stats and Producer stats.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer disable batching

```
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        setupBatchingAndCompression(props);

        return new KafkaProducer<>(props);
    }
    ...
    private static void setupBatchingAndCompression(
            final Properties props) {
        props.put(ProducerConfig.BATCH_SIZE_CONFIG,  0);
    }
    ...
}
```

Set batching to 16K and retest

Now let’s enable batching in the StockPriceKafkaProducer by setting batch size to 16K. Setting `props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16_384)` turns batching on and allows us to batch 16K of stock price records per partition. After you do this rerun StockPriceKafkaProducer and check Consumer stats and Producer stats.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer set batch size to 16K

```
public class StockPriceKafkaProducer {
    ...
    private static void setupBatchingAndCompression(
            final Properties props) {
        props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384);
    }
    ...
}
```

Results Set batching to 16K

We saw the consumer records per poll averages around 7.5 and saw the batch size increase to 136.02 - 59% more batching. Look how much the request queue time shrunk! The record-send-rate is 200% faster! You can see record-send-rate in the metrics of the producer.

Set batching to 16K and linger to 10ms

Now let's enable linger in the StockPriceKafkaProducer by setting the linger 10 ms. Setting `props.put(ProducerConfig.LINGER_MS_CONFIG, 10)` turns linger on and allows us to batch for 10 ms or 16K bytes of a stock price records per partition whichever comes first. After you do this rerun StockPriceKafkaProducer and check Consumer stats and Producer stats.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer set linger to 10 ms

```
public class StockPriceKafkaProducer {
    ...
    private static void setupBatchingAndCompression(
            final Properties props) {

              //Linger up to 10 ms before sending batch if size not met
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384);
    }
    ...
}
```

Results Set batching to 16K and linger to 10 ms

We saw the consumer records per poll averages around 17 and saw the batch size increase to 796 - 585% more batching. The record-send-rate went down, but higher than without batching.

Try different sizes and times

Try 16K, 32K and 64K batch sizes and then try 10 ms, 100 ms, and 1 second linger. Which is the best for which type of use case?

Set compression to snappy, then batching to 64K and linger to 50ms

Now let’s enable compression in the StockPriceKafkaProducer by setting the compression to linger. Setting `props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy)` turns compression. After you do this rerun StockPriceKafkaProducer and check Consumer stats and Producer stats as before.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer enable compression

```
public class StockPriceKafkaProducer {
    ...
    private static void setupBatchingAndCompression(
            final Properties props) {

              //Linger up to 50 ms before sending batch if size not met
              props.put(ProducerConfig.LINGER_MS_CONFIG, 50);

              //Batch up to 64K buffer sizes.
              props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384 * 4);

              //Use Snappy compression for batch compression.
              props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
    }
    ...
}
```

Results for turning on compression

The Snappy compression 64K/50ms should have the highest record-send-rate and $^1/_2$ the queue time.

Lab Adding Retries and Timeouts

In this lab we setup timeouts, setup retries, setup retry back off and change inflight messages to 1 so retries don’t store records out of order.

Change it and then run it

As before startup ZooKeeper if needed and three Kafka brokers. Then we will modify StockPriceKafkaProducer to configure retry, timeouts, in-flight message count and retry back off. We will run the StockPriceKafkaProducer. While we start and stop any two different Kafka Brokers while StockPriceKafkaProducer runs. Please notice retry messages in log of StockPriceKafkaProducer.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer disable batching

```
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        setupBatchingAndCompression(props);
        setupRetriesInFlightTimeout(props);

        return new KafkaProducer<>(props);
    }
    ...
    private static void setupRetriesInFlightTimeout(Properties props) {
        //Only two in-flight messages per Kafka broker connection
        // - max.in.flight.requests.per.connection (default 5)
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
                1);
        //Set the number of retries - retries
        props.put(ProducerConfig.RETRIES_CONFIG, 3);

        //Request timeout - request.timeout.ms
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 15_000);

        //Only retry after one second.
        props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 1_000);
    }
    ...
}
```

The above configures Producer retry, timeouts, in-flight message count and retry back off.

Expected output after 2 broker shutdown

Run all brokers and then kill any two servers. Look for retry messages in the Producer log. Restart brokers and see the Producer recover. Also use replica verification to see when the broker catches up.

WARN Inflight Message Count

The MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION is the max number of unacknowledged requests that a client can send on a single connection before blocking If >1 and failed sends, then there is a risk of message re-ordering on partition during retry attempt (they could be written out of order of the send). If this is bad, depends on use but for our StockPrices this is not good, you should pick retries > 1 or inflight > 1 but not both. Avoid duplicates. The June 2017 release might fix this with sequence from producer.

Lab Write ProducerInterceptor

Let’s setup an interceptor for request sends. To do this we will create a ProducerInterceptor and implement the onSend method and the onAcknowledge method.

Producer Interception

We will configure our Producer config and set the config property: interceptor.classes to our ProducerInterceptor which we will define shortly. The ProducerInterceptor will print out debug information when we send a message and when the broker acknowledges a message. The interceptors we pass must implement ProducerInterceptor interface so we will define a StockProducerInterceptor that implements ProducerInterceptor. The StockProducerInterceptor will intercept records that the producer sends to broker and after intercept acks from the broker.

Let’s define the StockProducerInterceptor as follows.

KafkaProducer ProducerInterceptor

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockProducerInterceptor.java

Kafka Producer: StockProducerInterceptor

```
package com.cloudurable.kafka.producer;

import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Map;

public class StockProducerInterceptor implements ProducerInterceptor {
   ...
}
```

Notice that the StockProducerInterceptor implements ProducerInterceptor.

ProducerInterceptor onSend

The onSend method gets called before the record is sent to the broker.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockProducerInterceptor.java

Kafka Producer: StockProducerInterceptor onSend

```
package com.cloudurable.kafka.producer;
...

public class StockProducerInterceptor implements ProducerInterceptor {

   private final Logger logger = LoggerFactory
           .getLogger(StockProducerInterceptor.class);
   private int onSendCount;
   private int onAckCount;


   @Override
   public ProducerRecord onSend(final ProducerRecord record) {
       onSendCount++;
       if (logger.isDebugEnabled()) {
           logger.debug(String.format("onSend topic=%s key=%s value=%s %d \n",
                   record.topic(), record.key(), record.value().toString(),
                   record.partition()
           ));
       } else {
           if (onSendCount % 100 == 0) {
               logger.info(String.format("onSend topic=%s key=%s value=%s %d \n",
                       record.topic(), record.key(), record.value().toString(),
                       record.partition()
               ));
           }
       }
       return record;
   }
}
```

The StockProducerInterceptor overrides the onSend method and increments onSendCount. Every 100 onSendCount, we print out record data.

ProducerInterceptor onAck

The onAck method gets called after the broker acknowledges the record.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockProducerInterceptor.java

Kafka Producer: StockProducerInterceptor onAck

```
package com.cloudurable.kafka.producer;
...

public class StockProducerInterceptor implements ProducerInterceptor {

   private final Logger logger = LoggerFactory
           .getLogger(StockProducerInterceptor.class);
   ...
   private int onAckCount;


   @Override
   public void onAcknowledgement(final RecordMetadata metadata,
                                 final Exception exception) {
       onAckCount++;

       if (logger.isDebugEnabled()) {
           logger.debug(String.format("onAck topic=%s, part=%d, offset=%d\n",
                   metadata.topic(), metadata.partition(), metadata.offset()
           ));
       } else {
           if (onAckCount % 100 == 0) {
               logger.info(String.format("onAck topic=%s, part=%d, offset=%d\n",
                       metadata.topic(), metadata.partition(), metadata.offset()
               ));
           }
       }
   }

}
```

The StockProducerInterceptor overrides the onAck method and increments onAckCount. Every 100 onAckCount, we print out record data.

ProducerInterceptor the rest

There are other methods to override.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockProducerInterceptor.java

Kafka Producer: StockProducerInterceptor the rest

```
package com.cloudurable.kafka.producer;

import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Map;

public class StockProducerInterceptor implements ProducerInterceptor {

    private final Logger logger = LoggerFactory
            .getLogger(StockProducerInterceptor.class);
    private int onSendCount;
    private int onAckCount;


    @Override
    public ProducerRecord onSend(final ProducerRecord record) {
        onSendCount++;
        if (logger.isDebugEnabled()) {
            logger.debug(String.format("onSend topic=%s key=%s value=%s %d \n",
                    record.topic(), record.key(), record.value().toString(),
                    record.partition()
            ));
        } else {
            if (onSendCount % 100 == 0) {
                logger.info(String.format("onSend topic=%s key=%s value=%s %d \n",
                        record.topic(), record.key(), record.value().toString(),
                        record.partition()
                ));
            }
        }
        return record;
    }

    @Override
    public void onAcknowledgement(final RecordMetadata metadata,
                                  final Exception exception) {
```

```
        onAckCount++;

        if (logger.isDebugEnabled()) {
            logger.debug(String.format("onAck topic=%s, part=%d, offset=%d\n",
                    metadata.topic(), metadata.partition(), metadata.offset()
            ));
        } else {
            if (onAckCount % 100 == 0) {
                logger.info(String.format("onAck topic=%s, part=%d, offset=%d\n",
                        metadata.topic(), metadata.partition(), metadata.offset()
                ));
            }
        }
    }

    @Override
    public void close() {
    }

    @Override
    public void configure(Map<String, ?> configs) {
    }
}
```

We have to override close and configure.

Next we need to configure the StockProducerInterceptor in the StockPriceKafkaProducer producer config.

KafkaProducer - Interceptor Config

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer

```
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        setupBatchingAndCompression(props);
        setupRetriesInFlightTimeout(props);

        //Install interceptor list - config "interceptor.classes"
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
                StockProducerInterceptor.class.getName());

        return new KafkaProducer<>(props);
    }
```

The above sets the `StockProducerInterceptor.class.getName()` in the config property `ProducerConfig.INTERCEPTOR_CLASSES_CONFIG` .

Run it. Run Servers. Run Producer. Results.

Next we startup ZooKeeper if needed, and start or restart Kafka brokers as before. Then run the StockPriceKafkaProducer and look for log message from ProducerInterceptor in output.

Results ProducerInterceptor Output

You should see oAck and onSend messages in the log from the interceptor.

Lab Write Custom Partitioner

Next let's create a StockPricePartitioner. The StockPricePartitioner will implement a priority queue. It will treat certain stocks as important and send those stocks to the last partition. The StockPricePartitioner implements the Kafka interface Partitioner. The Partitioner interface is used to pick which partition a record lands. We will need to implement the partition() method to choose the partition. And we will need to implement the configure() method so we can read the importantStocks config property to setup importantStocks set which we use to determine if a stock is important and needs to be sent to the important partition. To do this we need to configure new Partitioner in Producer config with property ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, and pass config property importantStocks.

Producer Partitioning

To set a custom partitioner set the Producer config property `partitioner.class`. The default `partitioner.class` is `org.apache.kafka.clients.producer.internals.DefaultPartitioner`. All Partitioner class implements the Kafka `Partitioner` interface and have to override the partition() method which takes topic, key, value, and cluster and then returns partition number for record.

StockPricePartitioner configure

StockPricePartitioner implements the configure() method with importantStocks config property. The importantStocks gets parsed and added to importantStocks HashSet which is used to filter the stocks.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer configure partitioner

```
package com.cloudurable.kafka.producer;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;

import java.util.*;

public class StockPricePartitioner implements Partitioner{

    private final Set<String> importantStocks;
    public StockPricePartitioner() {
        importantStocks = new HashSet<>();
    }
    ...
    @Override
    public void configure(Map<String, ?> configs) {
        final String importantStocksStr = (String) configs.get("importantStocks");
        Arrays.stream(importantStocksStr.split(","))
                .forEach(importantStocks::add);
    }

}
```

StockPricePartitioner partition()

IMPORTANT STOCK: If stockName is in the importantStocks HashSet then put it in partitionNum = (partitionCount -1) (last partition). REGULAR STOCK: Otherwise if not in importantStocks set then not important use the the absolute value of the hash of the stockName modulus partitionCount -1 as the partition to send the record `partitionNum = abs(stockName.hashCode()) % (partitionCount - 1)`.

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPricePartitioner.java

Kafka Producer: StockPricePartitioner partition

```
package com.cloudurable.kafka.producer;

public class StockPricePartitioner implements Partitioner{

    private final Set<String> importantStocks;
    public StockPricePartitioner() {
        importantStocks = new HashSet<>();
    }

    @Override
    public int partition(final String topic,
                         final Object objectKey,
                         final byte[] keyBytes,
                         final Object value,
                         final byte[] valueBytes,
                         final Cluster cluster) {

        final List<PartitionInfo> partitionInfoList =
                cluster.availablePartitionsForTopic(topic);
        final int partitionCount = partitionInfoList.size();
        final int importantPartition = partitionCount -1;
        final int normalPartitionCount = partitionCount -1;

        final String key = ((String) objectKey);

        if (importantStocks.contains(key)) {
            return importantPartition;
        } else {
            return Math.abs(key.hashCode()) % normalPartitionCount;
        }

    }
    ...
}
```

Producer Config: Configuring Partitioner

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPricePartitioner.java

Kafka Producer: StockPricePartitioner configure()

```
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        setupBatchingAndCompression(props);
        setupRetriesInFlightTimeout(props);

        //Set number of acknowledgments - acks - default is all
        props.put(ProducerConfig.ACKS_CONFIG, "all");


        //Install interceptor list - config "interceptor.classes"
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
                StockProducerInterceptor.class.getName());

        props.put("importantStocks", "IBM,UBER");

        props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
                StockPricePartitioner.class.getName());

        return new KafkaProducer<>(props);
    }
```

Configure the new Partitioner in Producer config with property ProducerConfig.INTERCEPTOR_CLASSES_CONFIG. Pass config property to importantStocks. The importantStock are the ones that go into priority queue. Run it as before. The important stocks are IBM and UBER in this example and are the only ones that will go into the last partition.

Review of lab work

You implemented custom ProducerSerializer. You tested failover configuring broker/topic min.insync.replicas, and acks. You implemented batching and compression and used metrics to see how it was or was not working. You implemented retires and timeouts, and tested that it worked. You setup max inflight messages and retry back off. You implemented a ProducerInterceptor. You implemented a custom partitioner to implement a priority queue for important stocks.

Slides

Please enjoy these slides which is the outline for this tutorial, and the complete code listing from all of the labs for this tutorial, which is below.

Cassandra / Kafka Support in EC2/AWS. Kafka Training, Kafka Consulting, K
Tutorial

CLOUDURABLE TM

Working with Kafka Producers

Kafka Producer Advanced

Working with producers in Java

Details and advanced topics

1 of 152

Kafka Tutorial: Advanced Producers (//www.slideshare.net/JeanPaulAzar1/kafka-tutorial-advanced-producers) from **Jean-Paul Azar (https://www.slideshare.net/JeanPaulAzar1)**

Full source code for the labs

Gradle build - Build file for Advanced Kafka Producers tutorial

~/kafka-training/lab5/solution/build.gradle

```
group 'cloudurable-kafka'
version '1.0-SNAPSHOT'

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.apache.kafka:kafka-clients:0.10.2.0'
    compile 'ch.qos.logback:logback-classic:1.2.2'
    compile 'io.advantageous.boon:boon-json:0.6.6'
    testCompile 'junit:junit:4.12'
}
```

SimpleStockPriceConsumer - Consumer shows batches sizes seen as batch size params vary on producer

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java

```
package com.cloudurable.kafka.consumer;
import com.cloudurable.kafka.StockAppConstants;
import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class SimpleStockPriceConsumer {

    private static Consumer<String, StockPrice> createConsumer() {
        final Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                StockAppConstants.BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
                "KafkaExampleConsumer");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName());
        //Custom Deserializer
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StockDeserializer.class.getName());
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
        // Create the consumer using props.
        final Consumer<String, StockPrice> consumer =
                new KafkaConsumer<>(props);
        // Subscribe to the topic.
        consumer.subscribe(Collections.singletonList(
                StockAppConstants.TOPIC));
        return consumer;
    }


    static void runConsumer() throws InterruptedException {
        final Consumer<String, StockPrice> consumer = createConsumer();
        final Map<String, StockPrice> map = new HashMap<>();
        try {
            final int giveUp = 1000; int noRecordsCount = 0;
            int readCount = 0;
            while (true) {
                final ConsumerRecords<String, StockPrice> consumerRecords =
                        consumer.poll(1000);
                if (consumerRecords.count() == 0) {
                    noRecordsCount++;
                    if (noRecordsCount > giveUp) break;
                    else continue;
                }
                readCount++;
                consumerRecords.forEach(record -> {
                    map.put(record.key(), record.value());
                });
                if (readCount % 100 == 0) {
                    displayRecordsStatsAndStocks(map, consumerRecords);
                }
                consumer.commitAsync();
            }
        }
        finally {
            consumer.close();
        }
        System.out.println("DONE");
    }

    private static void displayRecordsStatsAndStocks(
            final Map<String, StockPrice> stockPriceMap,
            final ConsumerRecords<String, StockPrice> consumerRecords) {
        System.out.printf("New ConsumerRecords par count %d count %d\n",
                consumerRecords.partitions().size(),
                consumerRecords.count());
        stockPriceMap.forEach((s, stockPrice) ->
                System.out.printf("ticker %s price %d.%d \n",
                    stockPrice.getName(),
                    stockPrice.getDollars(),
                    stockPrice.getCents()));
        System.out.println();
    }


    public static void main(String... args) throws Exception {
        runConsumer();
    }


}
```

StockDeserializer - Java example for Kafka Deserializer

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/consumer/StockDeserializer.java

```
package com.cloudurable.kafka.consumer;

import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.common.serialization.Deserializer;

import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockDeserializer implements Deserializer<StockPrice> {

    @Override
    public StockPrice deserialize(final String topic, final byte[] data) {
        return new StockPrice(new String(data, StandardCharsets.UTF_8));
    }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close() {
    }
}
```

StockPrice - Java example showing custom Kafka serialization

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/model/StockPrice.java

```
package com.cloudurable.kafka.producer.model;

import io.advantageous.boon.json.JsonFactory;

public class StockPrice {

    private final int dollars;
    private final int cents;
    private final String name;

    public StockPrice(final String json) {
        this(JsonFactory.fromJson(json, StockPrice.class));
    }


    public StockPrice() {
        dollars = 0;
        cents = 0;
        name ="";
    }

    public StockPrice(final String name, final int dollars, final int cents) {
        this.dollars = dollars;
        this.cents = cents;
        this.name = name;
    }



    public StockPrice(final StockPrice stockPrice) {
        this.cents = stockPrice.cents;
        this.dollars = stockPrice.dollars;
        this.name = stockPrice.name;
    }


    public int getDollars() {
        return dollars;
    }
    public int getCents() {
        return cents;
    }


    public String getName() {
        return name;
    }


    @Override
    public String toString() {
        return "StockPrice{" +
                "dollars=" + dollars +
                ", cents=" + cents +
                ", name='" + name + '\'' +
                '}';
    }


    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        StockPrice that = (StockPrice) o;

        if (dollars != that.dollars) return false;
        if (cents != that.cents) return false;
        return name != null ? name.equals(that.name) : that.name == null;
    }

    @Override
    public int hashCode() {
        int result = dollars;
        result = 31 * result + cents;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        return result;
    }
    public String toJson() {
        return "{" +
                "\"dollars\": " + dollars +
                ", \"cents\": " + cents +
                ", \"name\": \"" + name + '\"' +
                '}';
    }
}
```

MetricsProducerReporter - Java example showing using Kafka Producer Metrics

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/MetricsProducerReporter.java

```
package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.producer.model.StockPrice;
import io.advantageous.boon.core.Sets;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.common.Metric;
import org.apache.kafka.common.MetricName;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Locale;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.stream.Collectors;

public class MetricsProducerReporter implements Runnable {
    private final Producer<String, StockPrice> producer;
    private final Logger logger =
            LoggerFactory.getLogger(MetricsProducerReporter.class);

    //Used to Filter just the metrics we want
    private final Set<String> metricsNameFilter = Sets.set(
            "record-queue-time-avg", "record-send-rate", "records-per-request-avg",
            "request-size-max", "network-io-rate", "record-queue-time-avg",
            "incoming-byte-rate", "batch-size-avg", "response-rate", "requests-in-flight"
    );

    public MetricsProducerReporter(
            final Producer<String, StockPrice> producer) {
        this.producer = producer;
    }

    @Override
    public void run() {
        while (true) {
            final Map<MetricName, ? extends Metric> metrics
                    = producer.metrics();

            displayMetrics(metrics);
```

```java
            try {
                Thread.sleep(3_000);
            } catch (InterruptedException e) {
                logger.warn("metrics interrupted");
                Thread.interrupted();
                break;
            }
        }
    }


    static class MetricPair {
        private final MetricName metricName;
        private final Metric metric;
        MetricPair(MetricName metricName, Metric metric) {
            this.metricName = metricName;
            this.metric = metric;
        }
        public String toString() {
            return metricName.group() + "." + metricName.name();
        }
    }

    private void displayMetrics(Map<MetricName, ? extends Metric> metrics) {
        final Map<String, MetricPair> metricsDisplayMap = metrics.entrySet().stream()
                //Filter out metrics not in metricsNameFilter
                .filter(metricNameEntry ->
                        metricsNameFilter.contains(metricNameEntry.getKey().name()))
                //Filter out metrics not in metricsNameFilter
                .filter(metricNameEntry ->
                        !Double.isInfinite(metricNameEntry.getValue().value()) &&
                                !Double.isNaN(metricNameEntry.getValue().value()) &&
                                metricNameEntry.getValue().value() != 0
                )
                //Turn Map<MetricName,Metric> into TreeMap<String, MetricPair>
                .map(entry -> new MetricPair(entry.getKey(), entry.getValue()))
                .collect(Collectors.toMap(
                        MetricPair::toString, it -> it, (a, b) -> a, TreeMap::new
                ));
        //Output metrics
        final StringBuilder builder = new StringBuilder(255);
        builder.append("\n---------------------------------------\n");
        metricsDisplayMap.entrySet().forEach(entry -> {
            MetricPair metricPair = entry.getValue();
            String name = entry.getKey();
            builder.append(String.format(Locale.US, "%50s%25s\t\t%,-10.2f\t\t%s\n",
                    name,
                    metricPair.metricName.name(),
                    metricPair.metric.value(),
                    metricPair.metricName.description()
            ));
        });
        builder.append("\n---------------------------------------\n");
        logger.info(builder.toString());
    }


}
```

StockPriceKafkaProducer - Java Example that shows using Kafka producer multi-threaded

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

```
package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.StockAppConstants;
import com.cloudurable.kafka.producer.model.StockPrice;
import io.advantageous.boon.core.Lists;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.List;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        setupBatchingAndCompression(props);
        setupRetriesInFlightTimeout(props);

        //Install interceptor list - config "interceptor.classes"
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
                StockProducerInterceptor.class.getName());

        props.put("importantStocks", "IBM,UBER");

        return new KafkaProducer<>(props);
    }

    private static void setupRetriesInFlightTimeout(Properties props) {
        //Only two in-flight messages per Kafka broker connection
        // - max.in.flight.requests.per.connection (default 5)
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
                1);
        //Set the number of retries - retries
        props.put(ProducerConfig.RETRIES_CONFIG, 3);

        //Request timeout - request.timeout.ms
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 15_000);

        //Only retry after one second.
        props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 1_000);
    }



    private static void setupBootstrapAndSerializers(Properties props) {
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                StockAppConstants.BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                StringSerializer.class.getName());


        //Custom Serializer - config "value.serializer"
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                StockPriceSerializer.class.getName());

        //Set number of acknowledgments - acks - default is all
        props.put(ProducerConfig.ACKS_CONFIG, "all");

    }

    private static void setupBatchingAndCompression(
            final Properties props) {
        props.put(ProducerConfig.LINGER_MS_CONFIG, 100);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384 * 4);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
    }


    private static final Logger logger =
            LoggerFactory.getLogger(StockPriceKafkaProducer.class);
    public static void main(String... args)
            throws Exception {
        //Create Kafka Producer
        final Producer<String, StockPrice> producer = createProducer();
        //Create StockSender list
        final List<StockSender> stockSenders = getStockSenderList(producer);

        //Create a thread pool so every stock sender gets it own.
        // Increase by 1 to fit metrics.
        final ExecutorService executorService =
                Executors.newFixedThreadPool(stockSenders.size() + 1);

        //Run Metrics Producer Reporter which is runnable passing it the producer.
        executorService.submit(new MetricsProducerReporter(producer));

        //Run each stock sender in its own thread.
        stockSenders.forEach(executorService::submit);


        //Register nice shutdown of thread pool, then flush and close producer.
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            executorService.shutdown();
            try {
                executorService.awaitTermination(200, TimeUnit.MILLISECONDS);
                logger.info("Flushing and closing producer");
                producer.flush();
                producer.close(10_000, TimeUnit.MILLISECONDS);
            } catch (InterruptedException e) {
                logger.warn("shutting down", e);
            }
        }));
    }

    private static List<StockSender> getStockSenderList(
            final Producer<String, StockPrice> producer) {
        return Lists.list(
                new StockSender(StockAppConstants.TOPIC,
                        new StockPrice("IBM", 100, 99),
                        new StockPrice("IBM", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("SUN", 100, 99),
                        new StockPrice("SUN", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("GOOG", 500, 99),
                        new StockPrice("GOOG", 400, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("INEL", 100, 99),
                        new StockPrice("INEL", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("UBER", 1000, 99),
                        new StockPrice("UBER", 50, 0),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("ABC", 100, 99),
                        new StockPrice("ABC", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("XYZ", 100, 99),
                        new StockPrice("XYZ", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("DEF", 100, 99),
                        new StockPrice("DEF", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("DEF", 100, 99),
                        new StockPrice("DEF", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("AAA", 100, 99),
                        new StockPrice("AAA", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("BBB", 100, 99),
                        new StockPrice("BBB", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("CCC", 100, 99),
                        new StockPrice("CCC", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("DDD", 100, 99),
                        new StockPrice("DDD", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("EEE", 100, 99),
                        new StockPrice("EEE", 50, 10),
                        producer,
                        1, 10
                ),
                new StockSender(
                        StockAppConstants.TOPIC,
                        new StockPrice("FFF", 100, 99),
                        new StockPrice("FFF", 50, 10),
                        producer,
                        1, 10
                )
        );

    }

}
```

StockPricePartitioner - Java example showing implementation of custom Kafka Partitioner

/Users/jean-paul/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPricePartitioner.java

```java
package com.cloudurable.kafka.producer;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;

import java.util.*;

public class StockPricePartitioner implements Partitioner{

    private final Set<String> importantStocks;
    public StockPricePartitioner() {
        importantStocks = new HashSet<>();
    }

    @Override
    public int partition(final String topic,
                         final Object objectKey,
                         final byte[] keyBytes,
                         final Object value,
                         final byte[] valueBytes,
                         final Cluster cluster) {

        final List<PartitionInfo> partitionInfoList =
                cluster.availablePartitionsForTopic(topic);
        final int partitionCount = partitionInfoList.size();
        final int importantPartition = partitionCount -1;
        final int normalPartitionCount = partitionCount -1;

        final String key = ((String) objectKey);

        if (importantStocks.contains(key)) {
            return importantPartition;
        } else {
            return Math.abs(key.hashCode()) % normalPartitionCount;
        }

    }

    @Override
    public void close() {
    }

    @Override
    public void configure(Map<String, ?> configs) {
        final String importantStocksStr = (String) configs.get("importantStocks");
        Arrays.stream(importantStocksStr.split(","))
                .forEach(importantStocks::add);
    }

}
```

StockPriceSerializer - Java example showing implementation of custom Kafka Producer Serializer

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceSerializer.java

```
package com.cloudurable.kafka.producer;
import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.common.serialization.Serializer;
import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockPriceSerializer implements Serializer<StockPrice> {

    @Override
    public byte[] serialize(String topic, StockPrice data) {
        return data.toJson().getBytes(StandardCharsets.UTF_8);
    }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close() {
    }
}
```

StockProducerInterceptor - Java example showing implementation of custom Kafka ProducerInterceptor

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockProducerInterceptor.java

```
package com.cloudurable.kafka.producer;

import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Map;

public class StockProducerInterceptor implements ProducerInterceptor {

    private final Logger logger = LoggerFactory
            .getLogger(StockProducerInterceptor.class);
    private int onSendCount;
    private int onAckCount;

    @Override
    public ProducerRecord onSend(final ProducerRecord record) {
        onSendCount++;
        if (logger.isDebugEnabled()) {
            logger.debug(String.format("onSend topic=%s key=%s value=%s %d \n",
                    record.topic(), record.key(), record.value().toString(),
                    record.partition()
            ));
        } else {
            if (onSendCount % 100 == 0) {
                logger.info(String.format("onSend topic=%s key=%s value=%s %d \n",
                        record.topic(), record.key(), record.value().toString(),
                        record.partition()
                ));
            }
        }
        return record;
    }

    @Override
    public void onAcknowledgement(final RecordMetadata metadata,
                                  final Exception exception) {
        onAckCount++;
        if (logger.isDebugEnabled()) {
            logger.debug(String.format("onAck topic=%s, part=%d, offset=%d\n",
                    metadata.topic(), metadata.partition(), metadata.offset()
            ));
        } else {
            if (onAckCount % 100 == 0) {
                logger.info(String.format("onAck topic=%s, part=%d, offset=%d\n",
                        metadata.topic(), metadata.partition(), metadata.offset()
                ));
            }
        }
    }

    @Override
    public void close() {
    }

    @Override
    public void configure(Map<String, ?> configs) {
    }
}
```

StockSender- Java Example that shows using Kafka producer from many threads (multi-threaded producer)

/Users/jean-paul/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

```java
package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Date;
import java.util.Random;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class StockSender implements Runnable{

    private final StockPrice stockPriceHigh;
    private final StockPrice stockPriceLow;
    private final Producer<String, StockPrice> producer;
    private final int delayMinMs;
    private final int delayMaxMs;
    private final Logger logger = LoggerFactory.getLogger(StockSender.class);
    private final String topic;

    public StockSender(final String topic, final StockPrice stockPriceHigh,
                       final StockPrice stockPriceLow,
                       final Producer<String, StockPrice> producer,
                       final int delayMinMs,
                       final int delayMaxMs) {
        this.stockPriceHigh = stockPriceHigh;
        this.stockPriceLow = stockPriceLow;
        this.producer = producer;
        this.delayMinMs = delayMinMs;
        this.delayMaxMs = delayMaxMs;
        this.topic = topic;
    }


    public void run() {
        final Random random = new Random(System.currentTimeMillis());
        int sentCount = 0;

        while (true) {
            sentCount++;
            final ProducerRecord <String, StockPrice> record =
                                        createRandomRecord(random);
            final int delay = randomIntBetween(random, delayMaxMs, delayMinMs);

            try {
                final Future<RecordMetadata> future = producer.send(record);
                if (sentCount % 100 == 0) {displayRecordMetaData(record, future);}
                Thread.sleep(delay);
            } catch (InterruptedException e) {
                if (Thread.interrupted()) {
                    break;
                }
            } catch (ExecutionException e) {
                logger.error("problem sending record to producer", e);
            }
        }
    }

    private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
                                       final Future<RecordMetadata> future)
                                throws InterruptedException, ExecutionException {
        final RecordMetadata recordMetadata = future.get();
        logger.info(String.format("\n\t\t\tkey=%s, value=%s " +
                        "\n\t\t\tsent to topic=%s part=%d off=%d at time=%s",
                record.key(),
                record.value().toJson(),
                recordMetadata.topic(),
                recordMetadata.partition(),
                recordMetadata.offset(),
                new Date(recordMetadata.timestamp())
                ));
    }

    private final int randomIntBetween(final Random random,
                                       final int max,
                                       final int min) {
        return random.nextInt(max - min + 1) + min;
    }

    private ProducerRecord<String, StockPrice> createRandomRecord(
                final Random random) {

        final int dollarAmount = randomIntBetween(random,
                stockPriceHigh.getDollars(), stockPriceLow.getDollars());

        final int centAmount = randomIntBetween(random,
                stockPriceHigh.getCents(), stockPriceLow.getCents());

        final StockPrice stockPrice = new StockPrice(
                stockPriceHigh.getName(), dollarAmount, centAmount);

        return new ProducerRecord<>(topic, stockPrice.getName(),
                stockPrice);
    }
}
```

StockAppConstants - Constants

~/kafka-training/lab5/solution/src/main/java/com/cloudurable/kafka/StockAppConstants.java

```
package com.cloudurable.kafka;

public class StockAppConstants {
    public final static String TOPIC = "stock-prices";
    public final static String BOOTSTRAP_SERVERS =
            "localhost:9092,localhost:9093,localhost:9094";

}
```

logback.xml - example of setting up logging for Kafka

~/kafka-training/lab5/solution/src/main/resources/logback.xml

```
<configuration>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <!-- encoders are assigned the type
             ch.qos.logback.classic.encoder.PatternLayoutEncoder by default -->
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %n%msg%n</pattern>
        </encoder>
    </appender>


    <logger name="com.cloudurable.kafka" level="INFO" />

    <logger name="org.apache.kafka" level="INFO"/>
    <logger name="org.apache.kafka.common.metrics" level="INFO"/>

    <root level="debug">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

create-topic.sh - example of creating a topic with kafka-topics.sh passing min.insync.replicas=2

~/kafka-training/lab5/solution/bin/create-topic.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

kafka/bin/kafka-topics.sh \
    --create \
    --zookeeper localhost:2181 \
    --replication-factor 3 \
    --partitions 3 \
    --topic stock-prices \
    --config min.insync.replicas=2

    #--config unclean.leader.election.enable=true \
    #--config min.insync.replicas=2 \
    #--config compression.type=producer \
    #--config cleanup.policy=compact \
    #--config retention.ms=60000
```

delete-topic.sh - example of deleting a Kafka topic

~/kafka-training/lab5/solution/bin/delete-topic.sh

```
#!/usr/bin/env bash
cd ~/kafka-training
kafka/bin/kafka-topics.sh \
    --delete \
    --zookeeper localhost:2181 \
    --topic stock-prices
```

describe-topics.sh - example of using kafka-topics.sh to describe topics partitions/leader

~/kafka-training/lab5/solution/bin/describe-topics.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-topics.sh --describe \
    --topic stock-prices \
    --zookeeper localhost:2181
```

list-topics.sh - example of using kafka-topics.sh to show topics

~/kafka-training/lab5/solution/bin/list-topics.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-topics.sh --list \
    --zookeeper localhost:2181
```

replica-verification.sh - example of kafka-replica-verification.sh to show how far replicas are behind

~/kafka-training/lab5/solution/bin/replica-verification.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-replica-verification.sh  \
    --report-interval-ms 5000 \
    --topic-white-list  "stock-prices.*" \
    --broker-list localhost:9092,localhost:9093,localhost:9094
```

start-1st-server.sh - example using kafka-server-start.sh passing custom properties

~/kafka-training/lab5/solution/bin/start-1st-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-0.properties"
```

start-2nd-server.sh

~/kafka-training/lab5/solution/bin/start-2nd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-1.properties"
```

start-3rd-server.sh

~/kafka-training/lab5/solution/bin/start-3rd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-2.properties"
```

server-0.properties - example properties file for Kafka with min.insync.replicas=3 set

~/kafka-training/lab5/solution/config/server-0.properties

```
broker.id=0
port=9092
log.dirs=./logs/kafka-0
## Require three replicas to respond
## before acknowledging send from producer.
min.insync.replicas=3



compression.type=producer
auto.create.topics.enable=false
message.max.bytes=65536
replica.lag.time.max.ms=5000
delete.topic.enable=true
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
num.partitions=1
num.recovery.threads.per.data.dir=1
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connect=localhost:2181
zookeeper.connection.timeout.ms=6000
```

server-1.properties

~/kafka-training/lab5/solution/config/server-1.properties

```
broker.id=1
port=9092
log.dirs=./logs/kafka-1
## Require three replicas to respond
## before acknowledging send from producer.
min.insync.replicas=3



compression.type=producer
auto.create.topics.enable=false
message.max.bytes=65536
replica.lag.time.max.ms=5000
delete.topic.enable=true
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
num.partitions=1
num.recovery.threads.per.data.dir=1
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connect=localhost:2181
zookeeper.connection.timeout.ms=6000
```

server-2.properties

~/kafka-training/lab5/solution/config/server-2.properties

```
broker.id=2
port=9092
log.dirs=./logs/kafka-2
## Require three replicas to respond
## before acknowledging send from producer.
min.insync.replicas=3



compression.type=producer
auto.create.topics.enable=false
message.max.bytes=65536
replica.lag.time.max.ms=5000
delete.topic.enable=true
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
num.partitions=1
num.recovery.threads.per.data.dir=1
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connect=localhost:2181
zookeeper.connection.timeout.ms=6000
```

More to come.

Kafka Tutorial

This comprehensive *Kafka tutorial* covers Kafka architecture and design. The *Kafka tutorial* has example Java Kafka producers and Kafka consumers. The *Kafka tutorial* also covers Avro and Schema Registry.

Complete Kafka Tutorial: Architecture, Design, DevOps and Java Examples. (http://cloudurable.com/blog/kafka-tutorial-kafka-producer-advanced-java-examples/index.html)

- Kafka Tutorial Part 1: What is Kafka? (http://cloudurable.com/blog/what-is-kafka/index.html)
- Kafka Tutorial Part 2: Kafka Architecture (http://cloudurable.com/blog/kafka-architecture/index.html)
- Kafka Tutorial Part 3: Kafka Topic Architecture (http://cloudurable.com/blog/kafka-architecture-topics/index.html)
- Kafka Tutorial Part 4: Kafka Consumer Architecture (http://cloudurable.com/blog/kafka-architecture-consumers/index.html)
- Kafka Tutorial Part 5: Kafka Producer Architecture (http://cloudurable.com/blog/kafka-architecture-producers/index.html)
- Kafka Tutorial Part 6: Using Kafka from the command line (http://cloudurable.com/blog/kafka-tutorial-kafka-from-command-line/index.html)
- Kafka Tutorial Part 7: Kafka Broker Failover and Consumer Failover (http://cloudurable.com/blog/kafka-tutorial-kafka-failover-kafka-cluster/index.html)
- Kafka Tutorial Part 8: Kafka Ecosystem (http://cloudurable.com/blog/kafka-ecosystem/index.html)
- Kafka Tutorial Part 9: Kafka Low-Level Design (http://cloudurable.com/blog/kafka-architecture-low-level/index.html)
- Kafka Tutorial Part 10: Kafka Log Compaction Architecture (http://cloudurable.com/blog/kafka-architecture-log-compaction/index.html)
- Kafka Tutorial Part 11: Writing a Kafka Producer example in Java (http://cloudurable.com/blog/kafka-tutorial-kafka-producer/index.html)
- Kafka Tutorial Part 12: Writing a Kafka Consumer example in Java (http://cloudurable.com/blog/kafka-tutorial-kafka-consumer/index.html)
- Kafka Tutorial Part 13: Writing Advanced Kafka Producer Java examples (http://cloudurable.com/blog/kafka-tutorial-kafka-producer-advanced-java-examples/index.html)
- Kafka Tutorial 14: Writing Advanced Kafka Consumer Java examples
- Kafka Tutorial Part 15: Kafka and Avro (http://cloudurable.com/blog/avro/index.html)
- Kafka Tutorial Part 16: Kafka and Schema Registry (http://cloudurable.com/blog/kafka-avro-schema-registry/index.html)
- Kafka Tutorial (http://cloudurable.com/ppt/kafka-tutorial-cloudruable-v2.pdf) ____

About Cloudurable

We hope you enjoyed this article. Please provide feedback (http://cloudurable.com/contact/index.html). Cloudurable provides Kafka training (http://cloudurable.com/kafka-training/index.html), Kafka consulting (http://cloudurable.com/kafka-aws-consulting/index.html), Kafka support (http://cloudurable.com/subscription_support/index.html) and helps setting up Kafka clusters in AWS (http://cloudurable.com/services/index.html).

Check out our new GoLang course. We provide onsite Go Lang training which is instructor led (http://cloudurable.com/golang-onsite-instructor-led-training/index.html).

Share Tweet

Like 49 Share

SEARCH

Search

SHARE

Tweet

Share

facebook

49

Like

Share

FOLLOW

Follow @cloudurable

Follow

64

facebook

CATEGORIES

amazon-ebs (1) (http://cloudurable.com/categories/amazon-ebs/index.html)

amazon-ec2 (1) (http://cloudurable.com/categories/amazon-ec2/index.html)

amazon-vpc (1) (http://cloudurable.com/categories/amazon-vpc/index.html)

ansible (4) (http://cloudurable.com/categories/ansible/index.html)

avro (2) (http://cloudurable.com/categories/avro/index.html)

aws (4) (http://cloudurable.com/categories/aws/index.html) aws-cassandra (6) (http://cloudurable.com/categories/aws-cassandra/index.html)

aws-command-line (1) (http://cloudurable.com/categories/aws-command-line/index.html)

cassandra (12) (http://cloudurable.com/categories/cassandra/index.html)

cassandra-aws (3) (http://cloudurable.com/categories/cassandra-aws/index.html)

cassandra-cluster (1) (http://cloudurable.com/categories/cassandra-cluster/index.html)

cassandra-database (2) (http://cloudurable.com/categories/cassandra-database/index.html)

cassandra-training (5) (http://cloudurable.com/categories/cassandra-training/index.html)

cassandra-tutorial (5) (http://cloudurable.com/categories/cassandra-tutorial/index.html)

cloud (4) (http://cloudurable.com/categories/cloud/index.html)

cloudformation (1) (http://cloudurable.com/categories/cloudformation/index.html)

cloudurable (15) (http://cloudurable.com/categories/cloudurable/index.html)

cluster (1) (http://cloudurable.com/categories/cluster/index.html)

devops (16) (http://cloudurable.com/categories/devops/index.html)

ebs (3) (http://cloudurable.com/categories/ebs/index.html)

ec2 (1) (http://cloudurable.com/categories/ec2/index.html)

kafka (13) (http://cloudurable.com/categories/kafka/index.html)

kafka-advanced-consumers (1) (http://cloudurable.com/categories/kafka-advanced-consumers/index.html) kafka-architecture (13) (http://cloudurable.com/categories/kafka-architecture/index.html)

kafka-avro-serialization (1) (http://cloudurable.com/categories/kafka-avro-serialization/index.html)

kafka-consulting (2) (http://cloudurable.com/categories/kafka-consulting/index.html)

kafka-consumer (1) (http://cloudurable.com/categories/kafka-consumer/index.html)

kafka-consumers (1) (http://cloudurable.com/categories/kafka-consumers/index.html)

kafka-ecosystem (1) (http://cloudurable.com/categories/kafka-ecosystem/index.html)

kafka-schema-registry (1) (http://cloudurable.com/categories/kafka-schema-registry/index.html)

kafka-training (23) (http://cloudurable.com/categories/kafka-training/index.html)

kafka-tutorial (20) (http://cloudurable.com/categories/kafka-tutorial/index.html)

kaka-replication (1) (http://cloudurable.com/categories/kaka-replication/index.html)

kinesis (1) (http://cloudurable.com/categories/kinesis/index.html)

kinesis-consulting (1) (http://cloudurable.com/categories/kinesis-consulting/index.html)

linux (1) (http://cloudurable.com/categories/linux/index.html)

metricsd (1) (http://cloudurable.com/categories/metricsd/index.html)

microservices (3) (http://cloudurable.com/categories/microservices/index.html)

nodetool (1) (http://cloudurable.com/categories/nodetool/index.html)

schema-registry (1) (http://cloudurable.com/categories/schema-registry/index.html) smack (3) (http://cloudurable.com/categories/smack/index.html)

spark (3) (http://cloudurable.com/categories/spark/index.html)

spark--cassandra (1) (http://cloudurable.com/categories/spark--cassandra/index.html)

spark--kafka (1) (http://cloudurable.com/categories/spark--kafka/index.html)

spark-training (3) (http://cloudurable.com/categories/spark-training/index.html)

spark-tutorial (3) (http://cloudurable.com/categories/spark-tutorial/index.html)

ssh (1) (http://cloudurable.com/categories/ssh/index.html)

ssh-config (1) (http://cloudurable.com/categories/ssh-config/index.html)

ssl (1) (http://cloudurable.com/categories/ssl/index.html)

systemd (1) (http://cloudurable.com/categories/systemd/index.html)

tls (1) (http://cloudurable.com/categories/tls/index.html)

vagrant (5) (http://cloudurable.com/categories/vagrant/index.html)

TAGS

AKKA (HTTP://CLOUDURABLE.COM/TAGS/AKKA/INDEX.HTML)

AKKA-CONSULTING (HTTP://CLOUDURABLE.COM/TAGS/AKKA-CONSULTING/INDEX.HTML)

AMAZON-EBS (HTTP://CLOUDURABLE.COM/TAGS/AMAZON-EBS/INDEX.HTML)

AMAZON-EC2 (HTTP://CLOUDURABLE.COM/TAGS/AMAZON-EC2/INDEX.HTML)

AMI (HTTP://CLOUDURABLE.COM/TAGS/AMI/INDEX.HTML) ANSIBLE (HTTP://CLOUDURABLE.COM/TAGS/ANSIBLE/INDEX.HTML)

AVRO (HTTP://CLOUDURABLE.COM/TAGS/AVRO/INDEX.HTML)

AVRO-KAFKA (HTTP://CLOUDURABLE.COM/TAGS/AVRO-KAFKA/INDEX.HTML)

AWS (HTTP://CLOUDURABLE.COM/TAGS/AWS/INDEX.HTML)

AWS-CASSANDRA (HTTP://CLOUDURABLE.COM/TAGS/AWS-CASSANDRA/INDEX.HTML)

AWS-COMMAND-LINE (HTTP://CLOUDURABLE.COM/TAGS/AWS-COMMAND-LINE/INDEX.HTML)

CASSANDRA (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA/INDEX.HTML)

CASSANDRA-ARCHITECTURE (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-ARCHITECTURE/INDEX.HTML)

CASSANDRA-AWS (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-AWS/INDEX.HTML)

CASSANDRA-CLOUD (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-CLOUD/INDEX.HTML)

CASSANDRA-CLUSTER (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-CLUSTER/INDEX.HTML)

CASSANDRA-DATABASE (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-DATABASE/INDEX.HTML)

CASSANDRA-DBA (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-DBA/INDEX.HTML)

CASSANDRA-DEVOPS (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-DEVOPS/INDEX.HTML)

CASSANDRA-OS-SYSTEM-MEMORY (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-OS-SYSTEM-MEMORY/INDEX.HTML)

CASSANDRA-TRAINING (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-TRAINING/INDEX.HTML)

CASSANDRA-TUTORIAL (HTTP://CLOUDURABLE.COM/TAGS/CASSANDRA-TUTORIAL/INDEX.HTML)

CLOUD (HTTP://CLOUDURABLE.COM/TAGS/CLOUD/INDEX.HTML)

CLOUDFORMATION (HTTP://CLOUDURABLE.COM/TAGS/CLOUDFORMATION/INDEX.HTML)

CLOUDFORMATION-TUTORIAL (HTTP://CLOUDURABLE.COM/TAGS/CLOUDFORMATION-TUTORIAL/INDEX.HTML)

CLOUDURABLE (HTTP://CLOUDURABLE.COM/TAGS/CLOUDURABLE/INDEX.HTML)

CLUSTER (HTTP://CLOUDURABLE.COM/TAGS/CLUSTER/INDEX.HTML)

COMPUTE (HTTP://CLOUDURABLE.COM/TAGS/COMPUTE/INDEX.HTML)

CONSUMERS (HTTP://CLOUDURABLE.COM/TAGS/CONSUMERS/INDEX.HTML)

DBA (HTTP://CLOUDURABLE.COM/TAGS/DBA/INDEX.HTML) DEVOPS (HTTP://CLOUDURABLE.COM/TAGS/DEVOPS/INDEX.HTML)

EBS (HTTP://CLOUDURABLE.COM/TAGS/EBS/INDEX.HTML) EC2 (HTTP://CLOUDURABLE.COM/TAGS/EC2/INDEX.HTML)

EC2-INSTANCE-STORE (HTTP://CLOUDURABLE.COM/TAGS/EC2-INSTANCE-STORE/INDEX.HTML)

ECU (HTTP://CLOUDURABLE.COM/TAGS/ECU/INDEX.HTML) FAILOVER (HTTP://CLOUDURABLE.COM/TAGS/FAILOVER/INDEX.HTML)

IMAGES (HTTP://CLOUDURABLE.COM/TAGS/IMAGES/INDEX.HTML)

INSTANCES (HTTP://CLOUDURABLE.COM/TAGS/INSTANCES/INDEX.HTML) JMS (HTTP://CLOUDURABLE.COM/TAGS/JMS/INDEX.HTML)

KAFKA (HTTP://CLOUDURABLE.COM/TAGS/KAFKA/INDEX.HTML)

KAFKA-ADVANCED-CONSUMERS (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-ADVANCED-CONSUMERS/INDEX.HTML)

KAFKA-ADVANCED-PRODUCERS (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-ADVANCED-PRODUCERS/INDEX.HTML)

KAFKA-ARCHITECTURE (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-ARCHITECTURE/INDEX.HTML)

KAFKA-AVRO (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-AVRO/INDEX.HTML)

KAFKA-CONNECT (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-CONNECT/INDEX.HTML)

KAFKA-CONSULTING (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-CONSULTING/INDEX.HTML)

KAFKA-CONSUMERS (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-CONSUMERS/INDEX.HTML)

KAFKA-CONSUMERS-ADVANCED (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-CONSUMERS-ADVANCED/INDEX.HTML)

KAFKA-ECOSYSTEM (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-ECOSYSTEM/INDEX.HTML)

KAFKA-LOG-COMPACTION (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-LOG-COMPACTION/INDEX.HTML)

KAFKA-REST-PROXY (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-REST-PROXY/INDEX.HTML)

KAFKA-STREAMS (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-STREAMS/INDEX.HTML)

KAFKA-TRAINING (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-TRAINING/INDEX.HTML)

KAFKA-TUTORIAL (HTTP://CLOUDURABLE.COM/TAGS/KAFKA-TUTORIAL/INDEX.HTML)

KINESIS (HTTP://CLOUDURABLE.COM/TAGS/KINESIS/INDEX.HTML)

KINESIS-CONSULTING (HTTP://CLOUDURABLE.COM/TAGS/KINESIS-CONSULTING/INDEX.HTML)

LINUX (HTTP://CLOUDURABLE.COM/TAGS/LINUX/INDEX.HTML) METRICSD (HTTP://CLOUDURABLE.COM/TAGS/METRICSD/INDEX.HTML)

MICROSERVICES (HTTP://CLOUDURABLE.COM/TAGS/MICROSERVICES/INDEX.HTML)

MICROSERVICES-ARCHITECTURE (HTTP://CLOUDURABLE.COM/TAGS/MICROSERVICES-ARCHITECTURE/INDEX.HTML)

NAT (HTTP://CLOUDURABLE.COM/TAGS/NAT/INDEX.HTML) NODETOOL (HTTP://CLOUDURABLE.COM/TAGS/NODETOOL/INDEX.HTML)

NUMA (HTTP://CLOUDURABLE.COM/TAGS/NUMA/INDEX.HTML)

PRODUCERS (HTTP://CLOUDURABLE.COM/TAGS/PRODUCERS/INDEX.HTML)

QBIT (HTTP://CLOUDURABLE.COM/TAGS/QBIT/INDEX.HTML) RAM (HTTP://CLOUDURABLE.COM/TAGS/RAM/INDEX.HTML)

REAKT (HTTP://CLOUDURABLE.COM/TAGS/REAKT/INDEX.HTML)

REPLICATION (HTTP://CLOUDURABLE.COM/TAGS/REPLICATION/INDEX.HTML)

SCHEMA-REGISTRY (HTTP://CLOUDURABLE.COM/TAGS/SCHEMA-REGISTRY/INDEX.HTML)

SMACK (HTTP://CLOUDURABLE.COM/TAGS/SMACK/INDEX.HTML) **SPARK (HTTP://CLOUDURABLE.COM/TAGS/SPARK/INDEX.HTML)**

SPARK--CASSANDRA (HTTP://CLOUDURABLE.COM/TAGS/SPARK--CASSANDRA/INDEX.HTML)

SPARK-TRAINING (HTTP://CLOUDURABLE.COM/TAGS/SPARK-TRAINING/INDEX.HTML)

SPARK-TUTORIAL (HTTP://CLOUDURABLE.COM/TAGS/SPARK-TUTORIAL/INDEX.HTML)

SSH (HTTP://CLOUDURABLE.COM/TAGS/SSH/INDEX.HTML) **SSH-CONFIG (HTTP://CLOUDURABLE.COM/TAGS/SSH-CONFIG/INDEX.HTML)**

SSL (HTTP://CLOUDURABLE.COM/TAGS/SSL/INDEX.HTML) **SYSTEMD (HTTP://CLOUDURABLE.COM/TAGS/SYSTEMD/INDEX.HTML)**

TLS (HTTP://CLOUDURABLE.COM/TAGS/TLS/INDEX.HTML) **VAGRANT (HTTP://CLOUDURABLE.COM/TAGS/VAGRANT/INDEX.HTML)**

VCPU (HTTP://CLOUDURABLE.COM/TAGS/VCPU/INDEX.HTML) **VPC (HTTP://CLOUDURABLE.COM/TAGS/VPC/INDEX.HTML)**

WHAT-IS-KAFKA (HTTP://CLOUDURABLE.COM/TAGS/WHAT-IS-KAFKA/INDEX.HTML)

Apache Spark Training (http://cloudurable.com/spark-training/index.html)
Kafka Tutorial (http://cloudurable.com/blog/kafka-tutorial/index.html)
Akka Consulting (http://cloudurable.com/akka-consulting/index.html)
Cassandra Training (http://cloudurable.com/cassandra-course/index.html)
AWS Cassandra Database Support (http://cloudurable.com/subscription_support_benefits_cassandra/index.html)
Kafka Support Pricing (http://cloudurable.com/subscription_support/index.html?q=kafka)
Cassandra Database Support Pricing (http://cloudurable.com/subscription_support/index.html?q=cassandra)
Non-stop Cassandra (http://cloudurable.com/cloudurable-cassandra-watchdog/index.html?q=cassandra)
Watchdog (http://cloudurable.com/cloudurable-cassandra-watchdog/index.html?q=watchdog)
Advantages of using Cloudurable™ (http://cloudurable.com/advantages/index.html)
Cassandra Consulting (http://cloudurable.com/service-quick-start-mentoring-cassandra-or-kafka-aws-ec2/index.html)

Cloudurable™| Guide to AWS Cassandra Deploy (http://cloudurable.com/ppt/amazon-cassandra.pdf)
Cloudurable™| AWS Cassandra Guidelines and Notes (http://cloudurable.com/ppt/amazon-cassandra-notes.pdf)
Free guide to deploying Cassandra on AWS (http://cloudurable.com/cassandra-aws-consulting/index.html)
Kafka Training (http://cloudurable.com/kafka-training/index.html)
Kafka Consulting (http://cloudurable.com/kafka-aws-consulting/index.html)
DynamoDB Training (http://cloudurable.com/dynamodb-training/index.html)
DynamoDB Consulting (http://cloudurable.com/dynamodb-consulting/index.html)
Kinesis Training (http://cloudurable.com/kinesis-training/index.html)
Kinesis Consulting (http://cloudurable.com/kinesis-consulting/index.html)
Kafka Tutorial PDF (http://cloudurable.com/blog/kafka-tutorial-v1/index.html)
Redis Consulting (http://cloudurable.com/redis-consulting/index.html)
Redis Training (http://cloudurable.com/redis-onsite-instructor-led-training/index.html)
ElasticSearch / ELK Consulting (http://cloudurable.com/elk-consulting/index.html)
ElasticSearch Training (http://cloudurable.com/elasticsearch-onsite-instructor-led-training/index.html)
InfluxDB/TICK Training (http://cloudurable.com/influxdb-onsite-instructor-led-training/index.html) TICK Consulting (http://cloudurable.com/tick-consulting/index.html)

ABOUT US

Cloudurable™: Leader in AWS cloud computing for Kafka™, Cassandra™ Database, Apache Spark, AWS CloudFormation™ DevOps. We do **Cassandra training**, **Apache Spark**, **Kafka training**, **Kafka consulting** and **cassandra consulting** with a focus on AWS and data engineering. (FAQ (http://cloudurable.com/faq/index.html))

FOLLOW CLOUDURABLE™

facebook page (https://www.facebook.com/cloudurable)
google plus (https://plus.google.com/116648719730180908239)
twitter (https://twitter.com/cloudurable)
linkedin (https://www.linkedin.com/company/17964258/)

Why Cloudurable™?
Advantage of using Cloudurable™ (http://cloudurable.com/advantages/index.html)
About Cloudurable™?
About Cloudurable™ (http://cloudurable.com/faq/index.html)

What are the benefits of using subscription support?

Benefits of Subscription Cassandra Support (http://cloudurable.com/subscription_support_benefits_cassandra/index.html)

RECENT POSTS

KAFKA CONSUMER: ADVANCED CONSUMERS (HTTP://CLOUDURABLE.COM/BLOG/KAFKA-ADVANCED-CONSUMER-1/INDEX.HTML)

KAFKA TUTORIAL (HTTP://CLOUDURABLE.COM/BLOG/KAFKA-TUTORIAL/INDEX.HTML)

m/blog/kafka-

KAFKA TUTORIAL: CREATING ADVANCED KAFKA PRODUCERS IN JAVA (HTTP://CLOUDURABLE.COM/BLOG/KAFKA-TUTORIAL-KAFKA-PRODUCER-ADVANCED-JAVA-EXAMPLES/INDEX.HTML)

m/blog/kafka-

KAFKA CONSULTING (HTTP://CLOUDURABLE.COM/BLOG/KAFKA-CONSULTING/INDEX.HTML)

m/blog/kafka-

CONTACT

java-
http://cloudurable.com/blog/kafka-
examples/index.html)
consulting/index.html)

Cloudurable Tech
101 California Street
San Francisco
CA 94111
USA
America
(415) 758-1113 (tel:14157581113)

GO TO CONTACT PAGE (/CONTACT/INDEX.HTML)