Working with Kafka Producers

Kafka Producer Advanced

Working with producers in Java

Details and advanced topics

Objectives Create Producer

- Cover advanced topics regarding Java Kafka Consumers
- Custom Serializers
- Custom Partitioners
- Batching
- Compression
- Retries and Timeouts

Kafka Producer

- Kafka client that publishes records to Kafka cluster
- Thread safe
- Producer has pool of buffer that holds to-be-sent records
  - background I/O threads turning records into request bytes and transmit requests to Kafka
- Close producer so producer will not leak resources

Kafka Producer Send, Acks and Buffers

- ***send***() method is asynchronous
  - adds the record to output buffer and return right away
  - buffer used to batch records for efficiency IO and compression
- acks config controls Producer record durability. ”all" setting ensures full commit of record, and is most durable and least fast setting
- Producer can retry failed requests
- Producer has buffers of unsent records per topic partition (sized at ***batch.size***)

Kafka Producer: Buffering and batching

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

Producer Acks

- Producer Config property ***acks***
  - ***(default all)***
- Write Acknowledgment received count required from partition leader before write request deemed complete
- Controls ***Producer*** sent records durability
- Can be all (-1), none (0), or leader (1)

Acks 0 (NONE)

- acks=0
- Producer does not wait for any ack from broker at all
- Records added to the socket buffer are considered sent
- No guarantees of durability - maybe
- Record Offset returned is set to -1 (unknown)
- Record loss if leader is down
- Use Case: maybe log aggregation

Acks 1 (LEADER)

- acks=1
- Partition leader wrote record to its local log but responds without followers confirmed writes
- If leader fails right after sending ack, record could be lost
  - Followers might have not replicated the record
- Record loss is rare but possible
- Use Case: log aggregation

Acks -1 (ALL)

- acks=all or acks=-1
- Leader gets write confirmation from full set of ISRs before sending ack to producer
- Guarantees record not be lost as long as one ISR remains alive
- Strongest available guarantee
- Even stronger with broker setting ***min.insync.replicas*** (specifies the minimum number of ISRs that must acknowledge a write)
- Most Use Cases will use this and set a ***min.insync.replicas > 1***

KafkaProducer config Acks

StockPriceKafkaProducer.java

```
StockPriceKafkaProducer
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Stc
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27
28          //Set number of acknowledgments - acks - default is all
29          props.put(ProducerConfig.ACKS_CONFIG, "all");
30
31          return new KafkaProducer<>(props);
32      }
```

Producer Buffer Memory Size

- Producer config property: ***buffer.memory***
  - ***default 32MB***
- Total memory (bytes) producer can use to buffer records to be sent to broker
- Producer blocks up to ***max.block.ms*** if ***buffer.memory*** is exceeded
  - if it is sending faster than the broker can receive, exception is thrown

Batching by Size

- Producer config property: ***batch.size***
  - Default 16K
- Producer batch records
  - fewer requests for multiple records sent to same partition
  - Improved IO throughput and performance on both producer and server
- If record is larger than the batch size, it will not be batched
- Producer sends requests containing multiple batches
  - batch per partition
- Small batch size reduce throughput and performance. If batch size is too big, memory allocated for batch is wasted

Batching by Time and Size - 1

- ***Producer*** config property: ***linger.ms***
  - Default 0
- Producer groups together any records that arrive before they can be sent into a batch
  - good if records arrive faster than they can be sent out
- ***Producer*** can reduce requests count even under moderate load using ***linger.ms***

Batching by Time and Size - 2

- ***linger.ms*** adds delay to wait for more records to build up so larger batches are sent
  - ***good brokers throughput*** at ***cost of producer latency***
- If ***producer*** gets records who 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 to improve throughput of Brokers and reduce broker load (common improvement)

Compressing Batches

- ***Producer*** config property: ***compression.type***
  - Default 0
- Producer compresses request data
- By default producer does not compress
- Can be set to none, gzip, snappy, or lz4
- Compression is by batch
  - improves with larger batch sizes
- End to end compression possible if Broker config "compression.type" set to producer. Compressed data from producer sent to log and consumer by broker

Batching and Compression Example

```
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19          setupBatchingAndCompression(props);
```

```
57
58      private static void setupBatchingAndCompression(Properties props) {
59          //Wait up to 50 ms to batch to Kafka - linger.ms
60          props.put(ProducerConfig.LINGER_MS_CONFIG, 200);
61
62          //Holds up to 64K per partition default is 16K - batch.size
63          props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16_384 * 4);
64
65          //Holds up to 64 MB default is 32MB for all partition buffers
66          // - "buffer.memory"
67          props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33_554_432 * 2);
68
69          //Set compression type to snappy - compression.type
70          props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
71      }
```

Custom Serializers

- You don't have to use built in serializers
- You can write your own
- Just need to be able to convert to/fro a byte[]
- Serializers work for keys and values
- ***value.serializer*** and ***key.serializer***

Custom Serializers Config

```
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19
```

```
30
31      private static void setupBootstrapAndSerializers(Properties props) {
32          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
33                  StockAppConstants.BOOTSTRAP_SERVERS);
34          props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");
35          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
36                  StringSerializer.class.getName());
37
38          //Custom Serializer - config "value.serializer"
39          props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
40                  StockPriceSerializer.class.getName());
41      }
```

Custom Serializer

StockPriceSerializer.java

```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() {
    }
}
```

StockPrice

StockPrice.java

```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 String toJson() {
        return "{" +
                "\"dollars\": " + dollars +
                ", \"cents\": " + cents +
                ", \"name\": \"" + name + '\"' +
                '}';
    }

```

Broker Follower Write Timeout

- ***Producer*** config property: ***request.timeout.ms***
  - Default 30 seconds (30,000 ms)
- Maximum time broker waits for confirmation from followers to meet Producer acknowledgment requirements for ***ack=all***
- Measure of broker to broker latency of request
- 30 seconds is high, long process time is indicative of problems

Producer Request Timeout

- ***Producer*** config property: ***request.timeout.ms***
  - Default 30 seconds (30,000 ms)
- Maximum time producer waits for request to complete to broker
- Measure of producer to broker latency of request
- 30 seconds is very high, long request time is an indicator that brokers can't handle load

Producer Retries

- Producer config property: ***retries***
  - ***Default 0***
- Retry count if ***Producer*** does not get ack from Broker
  - only if record send fail deemed a transient error (API)
  - as if your producer code resent record on failed attempt
- timeouts are retried, ***retry.backoff.ms*** (default to 100 ms) to wait after failure before ***retry***

Retry, Timeout, Back-off Example

```
13
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19          setupBatchingAndCompression(props);
20          setupRetriesInFlightTimeout(props);
```

```
40
41      private static void setupRetriesInFlightTimeout(Properties props) {
42
43          //Only two in-flight messages per Kafka broker connection
44          // - max.in.flight.requests.per.connection (default 5)
45          props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
46                  1);
47
48          //Set the number of retries - retries
49          props.put(ProducerConfig.RETRIES_CONFIG, 2);
50
51          //Request timeout - request.timeout.ms
52          props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 15_000);
53
54          //Only retry after one second.
55          props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 1_000);
56      }
57
```

Producer Partitioning

- ***Producer*** config property: ***partitioner.class***
  - org.apache.kafka.clients.producer.internals.DefaultPartitioner
- Partitioner class implements Partitioner interface
- Default Partitioner partitions using hash of key if record has key
- Default Partitioner partitions uses round-robin if record has no key

Configuring Partitioner

StockPriceKafkaProducer.java

StockPriceKafkaProducer | createProducer()

```
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Sto
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27          //Install partitioner -- "partitioner.class"
28          props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
29                  StockPricePartitioner.class.getName());
30
31          props.put("importantStocks", "IBM,UBER");
```

StockPricePartitioner

StockPriceKafkaProducer.java × | StockPricePartitioner.java ×

StockPricePartitioner | StockPricePartitioner()

```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,
                         byte[] keyBytes, final Object value,
                         final byte[] valueBytes,
                         final Cluster cluster) {...}

    @Override
    public void close() {
    }

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

StockPricePartitioner partition()

StockPriceKafkaProducer.java × | StockPricePartitioner.java ×

StockPricePartitioner | partition()

```java
    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;
        }

    }
```

StockPricePartitioner

StockPriceKafkaProducer.java × | StockPricePartitioner.java ×

StockPricePartitioner | configure()

```
39
40      @Override
41      public void close() {
42      }
43
44      @Override
45      public void configure(Map<String, ?> configs) {
46          final String importantStocksStr = (String) configs.get("importantStocks");
47          Arrays.stream(importantStocksStr.split( regex: ","))
48                  .forEach(importantStocks::add);
49      }
50
```

Producer Interception

- ***Producer*** config property: ***interceptor.classes***
  - empty (you can pass an comma delimited list)
- interceptors implementing ProducerInterceptor interface
- intercept records producer sent to broker and after acks
- you could mutate records

KafkaProducer - Interceptor Config

StockPriceKafkaProducer.java

StockPriceKafkaProducer | getStockSenderList()

```
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Sto
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27          //Install interceptor list - config "interceptor.classes"
28          props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
29                  StockProducerInterceptor.class.getName());
30
31          //Set number of acknowledgments - acks - default is all
32          props.put(ProducerConfig.ACKS_CONFIG, "all");
33
34          return new KafkaProducer<>(props);
35      }
```

KafkaProducer ProducerInterceptor

StockProducerInterceptor.java

```
StockProducerInterceptor
10
11  public class StockProducerInterceptor implements ProducerInterceptor {
12
13      private final Logger logger = LoggerFactory
14              .getLogger(StockProducerInterceptor.class);
15      private int onSendCount;
16      private int onAckCount;
17
18      @Override
19      public ProducerRecord onSend(final ProducerRecord record) {...}
36
37      @Override
38      public void onAcknowledgement(final RecordMetadata metadata,
39                                    final Exception exception) {...}
54
55      @Override
56      public void close() {...}
59
60      @Override
61      public void configure(Map<String, ?> configs) {...}
64  }
```

ProducerInterceptor onSend

StockProducerInterceptor.java

StockProducerInterceptor

```
18      @Override
19      public ProducerRecord onSend(final ProducerRecord record) {
20          onSendCount++;
21          if (logger.isDebugEnabled()) {
22              logger.debug(String.format("onSend topic=%s key=%s value=%s %d \n",
23                      record.topic(), record.key(), record.value().toString(),
24                      record.partition()
25              ));
26          } else {
27              if (onSendCount % 100 == 0) {
28                  logger.info(String.format("onSend topic=%s key=%s value=%s %d \n",
29                          record.topic(), record.key(), record.value().toString(),
30                          record.partition()
31                  ));
32              }
33          }
34          return record;
35      }
```

Output

```
onSend topic=stock-prices2 key=UBER value=StockPrice{dollars=737,
                    cents=78, name='UBER'} null
```

ProducerInterceptor onAck

StockProducerInterceptor.java

StockProducerInterceptor | onAcknowledgement()

```
37      @Override
38      public void onAcknowledgement(final RecordMetadata metadata,
39                                    final Exception exception) {
40          onAckCount++;
41
42          if (logger.isDebugEnabled()) {
43              logger.debug(String.format("onAck topic=%s, part=%d, offset=%d\n",
44                      metadata.topic(), metadata.partition(), metadata.offset()
45              ));
46          } else {
47              if (onAckCount % 100 == 0) {
48                  logger.info(String.format("onAck topic=%s, part=%d, offset=%d\n",
49                          metadata.topic(), metadata.partition(), metadata.offset()
50                  ));
51              }
52          }
53      }
```

Output

```
onAck topic=stock-prices2, part=0, offset=18360
```

ProducerInterceptor the rest

StockProducerInterceptor.java

StockProducerInterceptor | configure()

```
48                  logger.info(String.format("onAck topic=%s,
49                          metadata.topic(), metadata.partitio
50                  ));
51              }
52          }
53      }
54
55      @Override
56      public void close() {
57      }
58
59      @Override
60      public void configure(Map<String, ?> configs) {
61      }
62  }
```

KafkaProducer send() Method

- Two forms of send with callback and with no callback both return Future
  - Asynchronously sends a record to a topic
  - Callback gets invoked when send has been acknowledged.
- send is asynchronous and return right away as soon as record has added to send buffer
- Sending many records at once without blocking for response from Kafka broker
- Result of send is a RecordMetadata
  - record partition, record offset, record timestamp
- Callbacks for records sent to same partition are executed in order

KafkaProducer send() 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 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)

Using send method

StockSender.java

StockSender | run()

```
51                  try {
52
53                      final Future<RecordMetadata> future = producer.send(record);
54
55                      if (sentCount % 100 == 0) {
56                          displayRecordMetaData(record, future);
```

StockSender.java

StockSender | displayRecordMetaData()

```
74      private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
75                                         final Future<RecordMetadata> future)
76                                 throws InterruptedException, ExecutionException {
77          final RecordMetadata recordMetadata = future.get();
78          logger.info(String.format("\n\t\t\tkey=%s, value=%s " +
79                          "\n\t\t\tsent to topic=%s part=%d off=%d at time=%s",
80                  record.key(),
81                  record.value().toJson(),
82                  recordMetadata.topic(),
83                  recordMetadata.partition(),
84                  recordMetadata.offset(),
85                  new Date(recordMetadata.timestamp())
86                  ));
87      }
```

KafkaProducer flush() method

- flush() method sends all buffered records now (even if *linger.ms* > 0)
  - blocks until requests complete
- Useful when consuming from some input system and pushing data into Kafka
- *flush*() ensures all previously sent messages have been sent
  - you could mark progress as such at completion of flush

KafkaProducer close()

- close() closes producer
  - frees resources (threads and buffers) associated with producer
- Two forms of method
  - both block until all previously sent requests complete or duration passed in as args is exceeded
  - close with no params equivalent to close(Long.MAX_VALUE, TimeUnit.MILLISECONDS).
  - If producer is unable to complete all requests before the timeout expires, all unsent requests fail, and this method fails

Orderly shutdown using close

StockPriceKafkaProducer.java

StockPriceKafkaProducer | main()

```
95      Runtime.getRuntime().addShutdownHook(new Thread(() -> {
96
97          executorService.shutdown();
98          try {
99              executorService.awaitTermination( timeout: 200, TimeUnit.MILLISECONDS);
100             logger.info("Shutting down executorService for workers nicely");
101         } catch (InterruptedException e) {
102             logger.warn("shutting down", e);
103         }
104
105         logger.info("Flushing producer");
106         producer.flush();
107         logger.info("Closing producer");
108         producer.close();
109
110         if (!executorService.isShutdown()) {
111             logger.info("Forcing shutdown of workers");
112             executorService.shutdownNow();
113         }
114     }));
```

Wait for clean close

StockPriceKafkaProducer.java

StockPriceKafkaProducer | main()

```
95      Runtime.getRuntime().addShutdownHook(new Thread(() -> {
96
97          executorService.shutdown();
98          try {
99              executorService.awaitTermination( timeout: 200, TimeUnit.MILLISECONDS);
100             logger.info("Flushing and closing producer");
101             producer.flush();
102             producer.close( timeout: 10_000, TimeUnit.MILLISECONDS);
103         } catch (InterruptedException e) {
104             logger.warn("shutting down", e);
105         }
106
107     }));
```

KafkaProducer partitionsFor() method

- partitionsFor(topic) returns meta data for partitions
- `public List<PartitionInfo> partitionsFor(String topic)`
- Get partition metadata for give topic
- Produce that do their own partitioning would use this
  - for custom partitioning
- `PartitionInfo(String topic, int partition, Node leader, Node[] replicas, Node[] inSyncReplicas)`
  - `Node(int id, String host, int port, optional String rack)`

KafkaProducer metrics() method

- metrics() method get map of metrics
- **public** Map<MetricName,? extends Metric> metrics()
- Get the full set of producer metrics

```
MetricName(
 String name,
 String group,
 String description,
 Map<String,String> tags
)
```

`public interface` **`Metric`**

A numerical metric tracked for monitoring purposes

Method Summary

Methods

Modifier and Type	Method and Description
MetricName	**metricName()** A name for this metric
double	**value()** The value of the metric

Metrics producer.metrics()

MetricsProducerReporter.java

MetricsProducerReporter

```
25
26          final Map<MetricName, ? extends Metric> metrics = producer.metrics();
27
28          metrics.forEach((metricName, metric) ->
29                  logger.info(
30                          String.format("\nMetric\t %s,\t %s,\t %s, " +
31                                  "\n\t\t%s\n",
32                                  metricName.group(),
33                                  metricName.name(),
34                                  metric.value(),
35                                  metricName.description())
36              ));
```

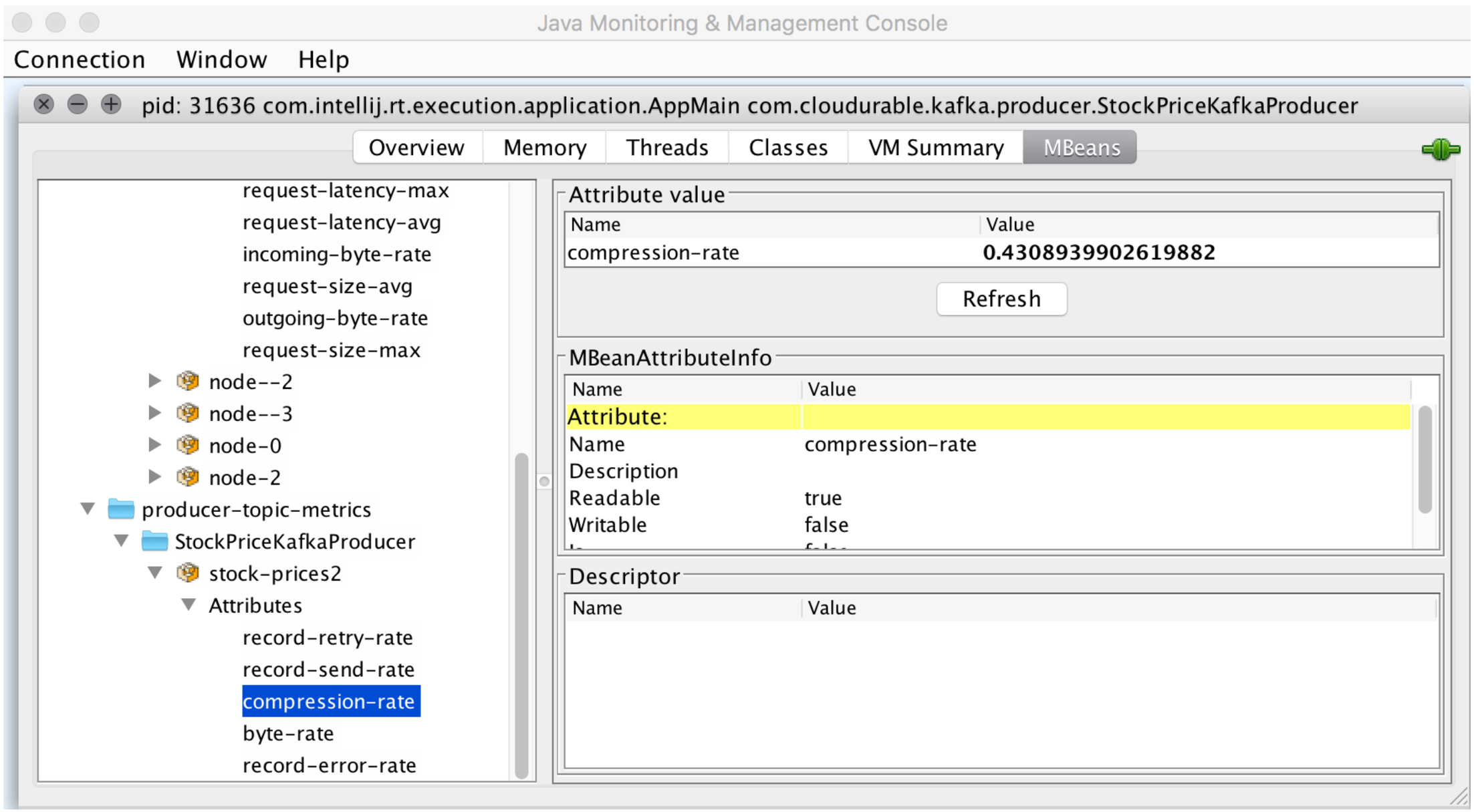
- Call producer.metrics()
- Prints out metrics to log

Metrics producer.metrics() output

```
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.
```

Metrics via JMX

```
~/kafka-training
$ jconsole
```



StockPrice Producer Java Example

Lab StockPrice Producer Java Example

StockPrice App to demo Advanced Producer

- ***StockPrice*** - holds a stock price has a name, dollar, and cents
- ***StockPriceKafkaProducer*** - Configures and creates ***KafkaProducer***<String, ***StockPrice***>, ***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
  - Shows using ***KafkaProducer*** from many threads

StockPrice domain object

StockPrice.java ×

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 String toJson() {
        return "{" +
                "\"dollars\": " + dollars +
                ", \"cents\": " + cents +
                ", \"name\": \"" + name + '\"' +
                '}';
    }

```

- has name
- dollars
- cents
- converts itself to JSON

StockPriceKafkaProducer

- Import classes and setup logger
- Create ***createProducer*** method to create ***KafkaProducer*** instance
- Create ***setupBootstrapAndSerializers*** to initialize bootstrap servers, client id, key serializer and custom serializer (***StockPriceSerializer***)
- Write ***main***() method - creates ***producer***, create ***StockSender*** list passing each instance a ***producer,*** creates a thread pool so every stock sender gets it own thread, runs each stockSender in its own thread

StockPriceKafkaProducer imports, createProducer

StockPriceKafkaProducer | main()

```
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.*;
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 final Logger logger =
            LoggerFactory.getLogger(StockPriceKafkaProducer.class);

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

Import classes and setup logger

createProducer used to create a KafkaProducer

createProducer() calls setupBoostrapAndSerializers()

Configure Producer Bootstrap and Serializer

StockPriceKafkaProducer

```
27    private static void setupBootstrapAndSerializers(Properties props) {
28        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
29                StockAppConstants.BOOTSTRAP_SERVERS);
30        props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");
31        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
32                StringSerializer.class.getName());
33
34        //Custom Serializer - config "value.serializer"
35        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
36                StockPriceSerializer.class.getName());
37    }
```

- Create ***setupBootstrapAndSerializers*** to initialize bootstrap servers, client id, key serializer and custom serializer (***StockPriceSerializer***)
- ***StockPriceSerializer*** will serialize ***StockPrice*** into bytes

StockPriceKafkaProducer.main()

StockPriceKafkaProducer.java

StockPriceKafkaProducer | main()

```
39      public static void main(String... args)
40                                  throws Exception {
41
42          //Create Kafka Producer
43          final Producer<String, StockPrice> producer = createProducer();
44
45          //Create StockSender list
46          final List<StockSender> stockSenders = getStockSenderList(producer);
47
48          //Create a thread pool so every stock sender gets it own.
49          final ExecutorService executorService =
50                  Executors.newFixedThreadPool(stockSenders.size());
51
52          //Run each stock sender in its own thread.
53          stockSenders.forEach(executorService::submit);
54
55      }
```

- ***main*** method - creates ***producer,***
- create ***StockSender*** list passing each instance a ***producer***
- creates a thread pool (executorService)
- every StockSender runs in its own thread

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";

}
```

- topic name for Producer example
- List of bootstrap servers

StockPriceKafkaProducer.getStockSenderList

StockPriceKafkaProducer | getStockSenderList()

```
57 @  private static List<StockSender> getStockSenderList(
58            final Producer<String, StockPrice> producer) {
59        return Lists.list(
60                new StockSender(StockAppConstants.TOPIC,
61                        new StockPrice( name: "IBM",  dollars: 100,  cents: 99),
62                        new StockPrice( name: "IBM",  dollars: 50,  cents: 10),
63                        producer,
64                         delayMinMs: 1,  delayMaxMs: 10
65                        ),
66                new StockSender(
67                        StockAppConstants.TOPIC,
68                        new StockPrice( name: "SUN",  dollars: 100,  cents: 99),
69                        new StockPrice( name: "SUN",  dollars: 50,  cents: 10),
70                        producer,
71                         delayMinMs: 1,  delayMaxMs: 10
72                ),
73                new StockSender(
74                        StockAppConstants.TOPIC,
75                        new StockPrice( name: "GOOG",  dollars: 500,  cents: 99),
76                        new StockPrice( name: "GOOG",  dollars: 400,  cents: 10),
77                        producer,
78                         delayMinMs: 1,  delayMaxMs: 10
79                ),
80                new StockSender(
81                        StockAppConstants.TOPIC,
82                        new StockPrice( name: "INEL",  dollars: 100,  cents: 99),
83                        new StockPrice( name: "INEL",  dollars: 50,  cents: 10),
84                        producer,
85                         delayMinMs: 1,  delayMaxMs: 10
```

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() {
    }
}
```

- Converts ***StockPrice*** into byte array

StockSender

- Generates random stock prices for a given ***StockPrice*** name,
- StockSender is Runnable
- 1 thread per StockSender
- Shows using ***KafkaProducer*** from many threads
- Delays random time between delayMin and delayMax,
  - then sends random StockPrice between ***stockPriceHigh*** and ***stockPriceLow***

StockSender imports, Runnable

StockSender

```
1   package com.cloudurable.kafka.producer;
2
3   import com.cloudurable.kafka.producer.model.StockPrice;
4   import org.apache.kafka.clients.producer.Producer;
5   import org.apache.kafka.clients.producer.ProducerRecord;
6   import org.apache.kafka.clients.producer.RecordMetadata;
7   import org.slf4j.Logger;
8   import org.slf4j.LoggerFactory;
9
10  import java.util.Date;
11  import java.util.Random;
12  import java.util.concurrent.ExecutionException;
13  import java.util.concurrent.Future;
14
15  public class StockSender implements Runnable{
```

- Imports Kafka *Producer, ProducerRecord, RecordMetadata, StockPrice*
- Implements Runnable, can be submitted to *ExecutionService*

StockSender constructor

StockSender

```
17      private final StockPrice stockPriceHigh;
18      private final StockPrice stockPriceLow;
19      private final Producer<String, StockPrice> producer;
20      private final int delayMinMs;
21      private final int delayMaxMs;
22      private final Logger logger = LoggerFactory.getLogger(StockSender.class);
23      private final String topic;
24
25      public StockSender(final String topic, final StockPrice stockPriceHigh,
26                         final StockPrice stockPriceLow,
27                         final Producer<String, StockPrice> producer,
28                         final int delayMinMs,
29                         final int delayMaxMs) {
30          this.stockPriceHigh = stockPriceHigh;
31          this.stockPriceLow = stockPriceLow;
32          this.producer = producer;
33          this.delayMinMs = delayMinMs;
34          this.delayMaxMs = delayMaxMs;
35          this.topic = topic;
36      }
```

- takes a topic, high & low stockPrice, producer, delay min & max

StockSender run()

StockSender | run()

```
38
39      public void run() {
40          final Random random = new Random(System.currentTimeMillis());
41          int sentCount = 0;
42
43          while (true) {
44              sentCount++;
45              final ProducerRecord <String, StockPrice> record =
46                                          createRandomRecord(random);
47              final int delay = randomIntBetween(random, delayMaxMs, delayMinMs);
48
49              try {
50                  final Future<RecordMetadata> future = producer.send(record);
51                  if (sentCount % 100 == 0) {displayRecordMetaData(record, future);}
52                  Thread.sleep(delay);
53              } catch (InterruptedException e) {
54                  if (Thread.interrupted()) {
55                      break;
56                  }
57              } catch (ExecutionException e) {
58                  logger.error("problem sending record to producer", e);
59              }
60          }
61      }
```

- In loop, creates random record, ***send*** record, waits random time

StockSender createRandomRecord

StockSender | createRandomRecord()

```
77
78      private final int randomIntBetween(final Random random,
79                                         final int max,
80                                         final int min) {
81          return random.nextInt( bound: max - min + 1) + min;
82      }
83
84      private ProducerRecord<String, StockPrice> createRandomRecord(
85                  final Random random) {
86
87          final int dollarAmount = randomIntBetween(random,
88                  stockPriceHigh.getDollars(), stockPriceLow.getDollars());
89
90          final int centAmount = randomIntBetween(random,
91                  stockPriceHigh.getCents(), stockPriceLow.getCents());
92
93          final StockPrice stockPrice = new StockPrice(
94                  stockPriceHigh.getName(), dollarAmount, centAmount);
95
96          return new ProducerRecord<>(topic, stockPrice.getName(),
97                  stockPrice);
98      }
```

- *createRandomRecord* uses *randomIntBetween*
- creates *StockPrice* and then wraps *StockPrice* in *ProducerRecord*

StockSender displayRecordMetaData

StockSender | displayRecordMetaData()

```
62
63  private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
64                                     final Future<RecordMetadata> future)
65                                 throws InterruptedException, ExecutionException {
66      final RecordMetadata recordMetadata = future.get();
67      logger.info(String.format("\n\t\t\tkey=%s, value=%s " +
68                      "\n\t\t\tsent to topic=%s part=%d off=%d at time=%s",
69              record.key(),
70              record.value().toJson(),
71              recordMetadata.topic(),
72              recordMetadata.partition(),
73              recordMetadata.offset(),
74              new Date(recordMetadata.timestamp())
75              ));
76  }
```

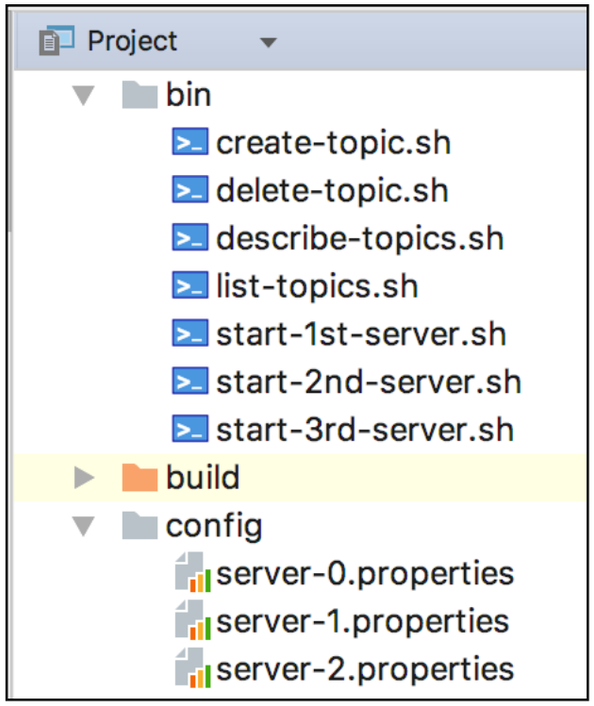
- Every 100 records displayRecordMetaData gets called
- Prints out record info, and recordMetadata info:
  - key, JSON value, topic, partition, offset, time
  - uses Future from call to ***producer.send***()

Run it

- Run ZooKeeper
- Run three Brokers
- run create-topic.sh
- Run ***StockPriceKafkaProducer***

Run scripts

- run ZooKeeper from ~/kafka-training
- use ***bin/create-topic.sh*** to create topic
- use ***bin/delete-topic.sh*** to delete topic
- use ***bin/start-1st-server.sh*** to run Kafka Broker 0
- use ***bin/start-2nd-server.sh*** to run Kafka Broker 1
- use ***bin/start-3rd-server.sh*** to run Kafka Broker 2



Config is under directory called config

server-0.properties is for Kafka Broker 0
server-1.properties is for Kafka Broker 1
server-2.properties is for Kafka Broker 2

Run All 3 Brokers

```
#!/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

```
#!/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

```
#!/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

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

server-1.properties

```
broker.id=1
port=9093
log.dirs=./logs/kafka-1
min.insync.replicas=3
compression.type=producer
```

server-2.properties

```
broker.id=2
port=9094
log.dirs=./logs/kafka-2
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
```

Run create-topic.sh script

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
```

Name of the topic is **stock-prices**

Three partitions

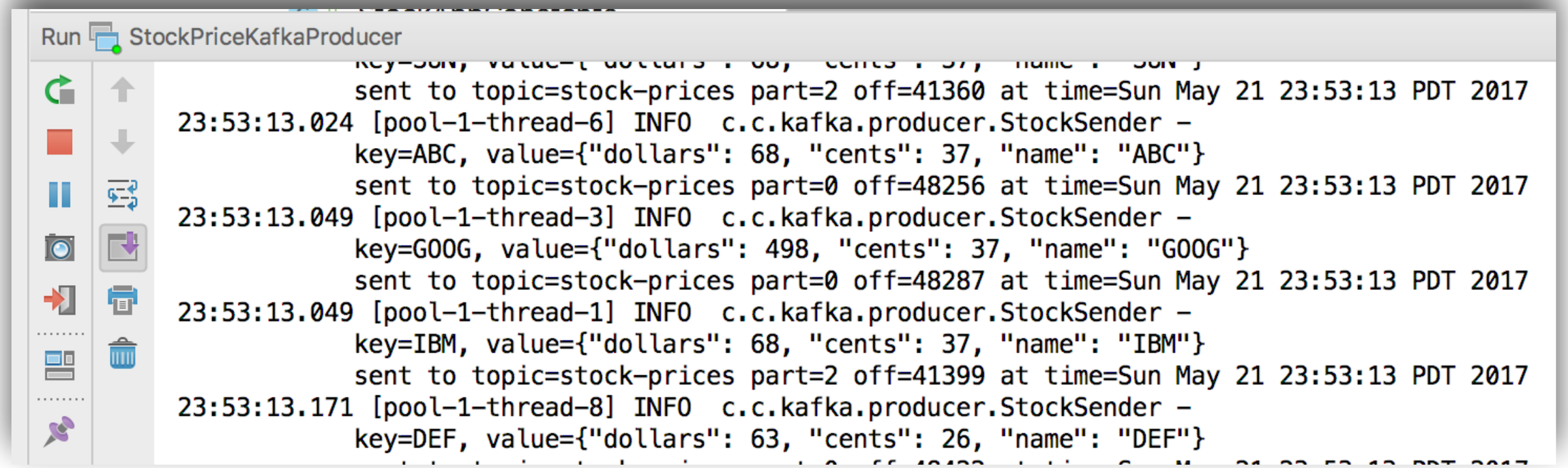
Replication factor of three

Terminal

Local | Local (1) | Local (2) | Local (3) | Local (4)

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

Run StockPriceKafkaProducer



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

Using flush and close

Lab Adding an orderly shutdown flush and close

Using flush and close

Shutdown Producer nicely

- Handle ctrl-C shutdown from Java
- Shutdown thread pool and wait
- Flush producer to send any outstanding batches if using batches (***producer.flush()***)
- Close Producer (***producer.close()***) and wait

Nice Shutdown producer.close()

StockPriceKafkaProducer.java

StockPriceKafkaProducer | main()

```
41  public static void main(String... args)
42          throws Exception {
43      //Create Kafka Producer
44      final Producer<String, StockPrice> producer = createProducer();
45      //Create StockSender list
46      final List<StockSender> stockSenders = getStockSenderList(producer);
47      //Create a thread pool so every stock sender gets it own.
48      final ExecutorService executorService =
49              Executors.newFixedThreadPool(stockSenders.size());
50      //Run each stock sender in its own thread.
51      stockSenders.forEach(executorService::submit);
52
53      //Register nice shutdown of thread pool, then flush and close producer.
54      Runtime.getRuntime().addShutdownHook(new Thread(() -> {
55          executorService.shutdown();
56          try {
57              executorService.awaitTermination( timeout: 200, TimeUnit.MILLISECONDS);
58              logger.info("Flushing and closing producer");
59              producer.flush();
60              producer.close( timeout: 10_000, TimeUnit.MILLISECONDS);
61          } catch (InterruptedException e) {
62              logger.warn("shutting down", e);
63          }
64      }));
65  }
```

Restart Producer then shut it down

- Add shutdown hook
- Start StockPriceKafkaProducer
- Now stop it (CTRL-C or hit stop button in IDE)

```
Run  StockPriceKafkaProducer
00:06:52.709 [pool-1-thread-4] INFO  c.c.kafka.producer.StockSender -
            key=INEL, value={"dollars": 73, "cents": 10, "name": "INEL"}
            sent to topic=stock-prices part=0 off=93522 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.710 [pool-1-thread-5] INFO  c.c.kafka.producer.StockSender -
            key=UBER, value={"dollars": 547, "cents": 90, "name": "UBER"}
            sent to topic=stock-prices part=2 off=80143 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.717 [pool-1-thread-2] INFO  c.c.kafka.producer.StockSender -
            key=SUN, value={"dollars": 73, "cents": 10, "name": "SUN"}
            sent to topic=stock-prices part=2 off=80146 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.718 [pool-1-thread-1] INFO  c.c.kafka.producer.StockSender -
            key=IBM, value={"dollars": 73, "cents": 10, "name": "IBM"}
            sent to topic=stock-prices part=2 off=80148 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.967 [Thread-1] INFO  c.c.k.p.StockPriceKafkaProducer - Flushing and closing producer
00:06:52.968 [Thread-1] INFO  o.a.k.clients.producer.KafkaProducer - Closing the Kafka producer

Process finished with exit code 130 (interrupted by signal 2: SIGINT)
```

Lab Configuring Producer Durability

Set default acks to **all**

```
19  public class StockPriceKafkaProducer {
20
21      private static void setupBootstrapAndSerializers(Properties props) {
22          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
23                  StockAppConstants.BOOTSTRAP_SERVERS);
24          props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProduc
25          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
26                  StringSerializer.class.getName());
27
28          //Custom Serializer - config "value.serializer"
29          props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
30                  StockPriceSerializer.class.getName());
31
32          //Set number of acknowledgments - acks - default is all
33          props.put(ProducerConfig.ACKS_CONFIG, "all");
34
35      }
```

- Set defaults acks to all (this is the default)
- This means that all ISRs in-sync replicas have to respond for producer write to go through

Note Kafka Broker 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
```

- At least this many in-sync replicas (ISRs) have to respond for producer to get ack
- NOTE: We have three brokers in this lab, all three have to be up

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

- If not already, startup ZooKeeper
- Startup three Kafka brokers
  - using scripts described earlier
- From the IDE run ***StockPriceKafkaProducer***
- 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***
  - Messages are rejected since there are fewer in-sync replicas than required.

What happens when we shut one down?

Terminal

Zookeeper | Broker 0 | Broker 1 | Broker 2

```
[2017-05-23 14:29:46,013] INFO [ExpirationReaper-0], Stopped  (kafka.server.DelayedOperationPurgatory$ExpiredOperation
[2017-05-23 14:29:46,013] INFO [ExpirationReaper-0], Shutdown completed (kafka.server.DelayedOperationPurgatory$Expire
[2017-05-23 14:29:46,013] INFO [ExpirationReaper-0], Shutting down (kafka.server.DelayedOperationPurgatory$ExpiredOper
[2017-05-23 14:29:46,179] INFO [ExpirationReaper-0], Stopped  (kafka.server.DelayedOperationPurgatory$ExpiredOperation
[2017-05-23 14:29:46,179] INFO [ExpirationReaper-0], Shutdown completed (kafka.server.DelayedOperationPurgatory$Expire
[2017-05-23 14:29:46,179] INFO [GroupCoordinator 0]: Shutdown complete. (kafka.coordinator.GroupCoordinator)
[2017-05-23 14:29:46,180] INFO Shutting down. (kafka.log.LogManager)
[2017-05-23 14:29:46,240] INFO Shutdown complete. (kafka.log.LogManager)
[2017-05-23 14:29:46,252] INFO Terminate ZkClient event thread. (org.I0Itec.zkclient.ZkEventThread)
[2017-05
[2017-05
[2017-05
~/kafka-
$
```

Run StockPriceKafkaProducer

```
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.kafka.common.errors.NotEnoughReplicasException: Messages are rejected sin
14:34:11.041 [pool-1-thread-2] ERROR c.c.kafka.producer.StockSender - problem sending record to
java.util.concurrent.ExecutionException: org.apache.kafka.common.errors.NotEnoughReplicasExcept
    at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.valueOrError(FutureReco
    at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadat
    at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadat
    at com.cloudurable.kafka.producer.StockSender.displayRecordMetaData(StockSender.java:66)
    at com.cloudurable.kafka.producer.StockSender.run(StockSender.java:51)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.kafka.common.errors.NotEnoughReplicasException: Messages are rejected sin
```

4: Run | 6: TODO | Terminal

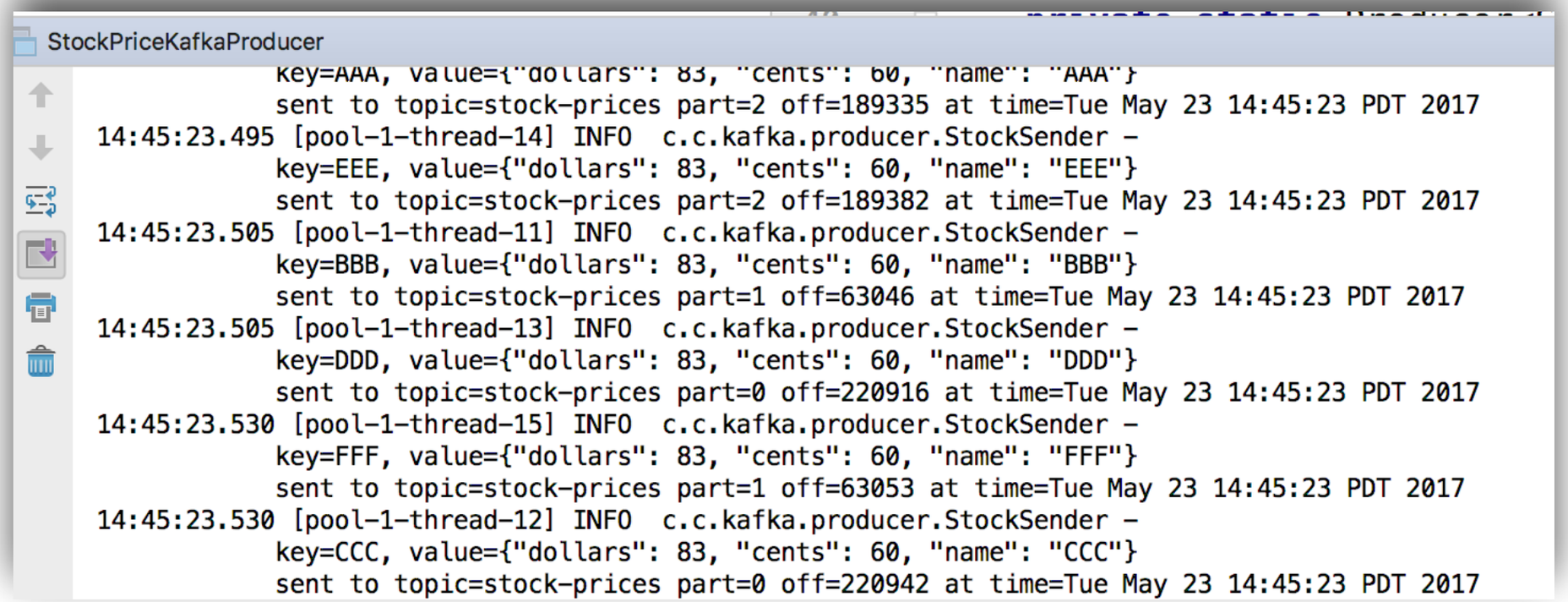
Why did the send fail?

- ProducerConfig.ACKS_CONFIG (acks config for producer) was set to *"all"*
  - Expects ***leader*** to ***only give successful ack after*** all followers ack the send
- Broker Config ***min.insync.replicas*** set to 3
  - At least three in-sync replicas must respond before send is considered successful

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

- If not already, startup ZooKeeper
- Ensure all three Kafka Brokers are running if not running
- Change ***StockPriceKafkaProducer*** acks config to 1
  props.put(ProducerConfig.ACKS_CONFIG, “1"); (leader sends ack after write to log)
- From the IDE run ***StockPriceKafkaProducer***
- From the terminal kill one of the Kafka Brokers
- ***StockPriceKafkaProducer*** runs normally

What happens when we shutdown acks "1" this time?



- Nothing happens!
- It continues to work because only the leader has to ack its write

Which type of application would you only want acks set to 1?

Why did the send not fail for acks 1?

- ProducerConfig.ACKS_CONFIG (acks config for producer) was set to *"1"*
  - Expects ***leader*** to ***only give successful ack after*** it writes to its log
  - Replicas still get replication but leader does not wait for replication
- Broker Config ***min.insync.replicas*** is still set to 3
  - ***This config only gets looked at if acks="all"***

Try describe topics before and after

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
```

- Try this last one again
- Stop a server while producer is running
- Run describe-topics.sh (shown above)
- Rerun server you stopped
- Run describe-topics.sh again

Describe-Topics After Each Stop/Start

Terminal

ZooKeeper | Kafka Broker 0 | Kafka Broker 1 | Kafka Broker 2 | Util

```
~/kafka-training/lab5/solution
$ bin/describe-topics.sh
Topic:stock-prices      PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: stock-prices     Partition: 0    Leader: 1       Replicas: 0,1,2 Isr: 1,2,0
        Topic: stock-prices     Partition: 1    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: stock-prices     Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 1,2,0
```

All 3 brokers running

```
~/kafka-training/lab5/solution
$ bin/describe-topics.sh
Topic:stock-prices      PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: stock-prices     Partition: 0    Leader: 1       Replicas: 0,1,2 Isr: 1,2
        Topic: stock-prices     Partition: 1    Leader: 1       Replicas: 1,2,0 Isr: 1,2
        Topic: stock-prices     Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 1,2
```

1 broker down.

Leader 1has 2 partitions

```
~/kafka-training/lab5/solution
$ bin/describe-topics.sh
Topic:stock-prices      PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: stock-prices     Partition: 0    Leader: 1       Replicas: 0,1,2 Isr: 1,2,0
        Topic: stock-prices     Partition: 1    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: stock-prices     Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 1,2,0
```

All 3 brokers running.
Look at Leader 1

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

All 3 brokers running
after a few minutes while

Review Question

```
~/kafka-training/lab5/solution
$ bin/describe-topics.sh
Topic:stock-prices      PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: stock-prices     Partition: 0    Leader: 2       Replicas: 0,1,2 Isr: 2
        Topic: stock-prices     Partition: 1    Leader: 2       Replicas: 1,2,0 Isr: 2
        Topic: stock-prices     Partition: 2    Leader: 2       Replicas: 2,0,1 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?

Retry with acks = 0

- Run the last example again (servers, and producer)
- Run all three brokers then take one away
- Then take another broker away
- 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***

Using Kafka built-in Producer Metrics

Adding Producer Metrics and Replication Verification

Metrics
Replication Verification

Objectives of lab

- Setup Kafka Producer Metrics
- Use replication verification command line tool
- Change min.insync.replicas for broker and observer metrics and replication verification
- Change min.insync.replicas for topic and observer metrics and replication verification

Create Producer Metrics Monitor

- Create a class called ***MetricsProducerReporter*** that is ***Runnable***
- Pass it a Kafka ***Producer***
- Call ***producer***.*metrics()* every 10 seconds in a while loop from run method, and print out the ***MetricName*** and ***Metric*** value
- Submit ***MetricsProducerReporter*** to the ***ExecutorService*** in the main method of ***StockPriceKafkaProducer***

Create MetricsProducerReporter (Runnable)

MetricsProducerReporter

```
1  package com.cloudurable.kafka.producer;
2
3  import com.cloudurable.kafka.producer.model.StockPrice;
4  import org.apache.kafka.clients.producer.Producer;
5  import org.apache.kafka.common.Metric;
6  import org.apache.kafka.common.MetricName;
7  import org.slf4j.Logger;
8  import org.slf4j.LoggerFactory;
9
10 import java.util.Map;
11
12 public class MetricsProducerReporter implements Runnable{
13     private final Producer<String, StockPrice> producer;
14     private final Logger logger =
15             LoggerFactory.getLogger(MetricsProducerReporter.class);
16
17     public MetricsProducerReporter(
18             final Producer<String, StockPrice> producer) {
19         this.producer = producer;
20     }
```

- Implements ***Runnable*** takes a ***producer***

Create MetricsProducerReporter (Runnable)

MetricsProducerReporter | run()

```
24  public void run() {
25      while (true) {
26          final Map<MetricName, ? extends Metric> metrics
27                  = producer.metrics();
28
29          metrics.forEach((metricName, metric) ->
30                  logger.info(String.format(
31                      "\nMetric\t %s,\t %s,\t %s,\n\t\t%s\n",
32                      metricName.group(), metricName.name(),
33                      metric.value(), metricName.description()))
34                  ));
35          try {
36              Thread.sleep( millis: 10_000);
37          } catch (InterruptedException e) {
38              logger.warn("metrics interrupted");
39              Thread.interrupted();
40              break;
41          }
```

- Call ***producer***.*metrics()* every 10 seconds in a while loop from run method, and print out ***MetricName*** and ***Metric*** value

Create MetricsProducerReporter (Runnable)

StockPriceKafkaProducer | main()

```
47  public static void main(String... args)
48          throws Exception {
49      //Create Kafka Producer
50      final Producer<String, StockPrice> producer = createProducer();
51      //Create StockSender list
52      final List<StockSender> stockSenders = getStockSenderList(producer);
53
54      //Create a thread pool so every stock sender gets it own.
55      // Increase by 1 to fit metrics.
56      final ExecutorService executorService =
57              Executors.newFixedThreadPool( nThreads: stockSenders.size() + 1);
58
59      //Run Metrics Producer Reporter which is runnable passing it the producer.
60      executorService.submit(new MetricsProducerReporter(producer));
61
62      //Run each stock sender in its own thread.
63      stockSenders.forEach(executorService::submit);
```

- Increase thread pool size by 1 to fit metrics reporting
- Submit instance of ***MetricsProducerReporter*** to the ***ExecutorService*** in the main method of ***StockPriceKafkaProducer (and pass new instance a producer)***

Run it. Run Servers. Run Producer.

- If not already, startup ZooKeeper
- Startup three Kafka brokers
  - using scripts described earlier
- From the IDE run ***StockPriceKafkaProducer (ensure acks are set to all first)***
- ***Observe metrics which print out every ten seconds***

Look at the output

15:40:43.858 [pool-1-thread-1] INFO c.c.k.p.MetricsProducerReporter -
Metric producer-node-metrics, outgoing-byte-rate, 1.8410309773473144,

15:40:43.858 [pool-1-thread-1] INFO c.c.k.p.MetricsProducerReporter -
Metric producer-topic-metrics, record-send-rate, 975.3229844767151,

15:40:43.858 [pool-1-thread-1] INFO c.c.k.p.MetricsProducerReporter -
Metric producer-node-metrics, request-rate, 0.040021611670301965,
The average number of requests sent per second.

15:40:43.858 [pool-1-thread-1] INFO c.c.k.p.MetricsProducerReporter -
Metric producer-node-metrics, incoming-byte-rate, 7.304382629577747,

Improve output with some Java love

MetricsProducerReporter.java

MetricsProducerReporter run()

```
11  import java.util.Locale;
12  import java.util.Map;
13  import java.util.Set;
14  import java.util.TreeMap;
15  import java.util.stream.Collectors;
16
17  public class MetricsProducerReporter implements Runnable {
18      private final Producer<String, StockPrice> producer;
19      private final Logger logger =
20              LoggerFactory.getLogger(MetricsProducerReporter.class);
21
22      //Used to Filter just the metrics we want
23      private final Set<String> metricsNameFilter = Sets.set(
24              "record-queue-time-avg", "record-send-rate", "records-per-request-avg",
25              "request-size-max", "network-io-rate", "record-queue-time-avg",
26              "incoming-byte-rate", "batch-size-avg", "response-rate", "requests-in-flight"
27      );
```

- Keep a set of only the metrics we want

Improve output: Filter metrics

MetricsProducerReporter

```
64      private void displayMetrics(Map<MetricName, ? extends Metric> metrics) {
65          final Map<String, MetricPair> metricsDisplayMap = metrics.entrySet().stream()
66                  //Filter out metrics not in metricsNameFilter
67                  .filter(metricNameEntry ->
68                          metricsNameFilter.contains(metricNameEntry.getKey().name()))
69                  //Filter out metrics not in metricsNameFilter
70                  .filter(metricNameEntry ->
71                          !Double.isInfinite(metricNameEntry.getValue().value()) &&
72                                  !Double.isNaN(metricNameEntry.getValue().value()) &&
73                                  metricNameEntry.getValue().value() != 0
74                  )
75                  //Turn Map<MetricName,Metric> into TreeMap<String, MetricPair>
76                  .map(entry -> new MetricPair(entry.getKey(), entry.getValue()))
77                  .collect(Collectors.toMap(
78                          MetricPair::toString, it -> it, (a, b) -> a, TreeMap::new
79                  ));
```

- Use Java 8 Stream to filter and sort metrics
- Get rid of ***metric values*** that are ***NaN, Infinite numbers*** and ***0s***
- Sort map my converting it to ***TreeMap***<String, ***MetricPair***>
- ***MetricPair*** is helper class that has a ***Metric*** and a ***MetricName***

Improve output: Pretty Print

MetricsProducerReporter | displayMetrics()

```
81
82          //Output metrics
83          final StringBuilder builder = new StringBuilder( capacity: 255);
84          builder.append("\n----------------------------------------\n");
85          metricsDisplayMap.entrySet().forEach(entry -> {
86              MetricPair metricPair = entry.getValue();
87              String name = entry.getKey();
88              builder.append(String.format(Locale.US, "%50s%25s\t\t%,-10.2f\t\t%s\n",
89                      name,
90                      metricPair.metricName.name(),
91                      metricPair.metric.value(),
92                      metricPair.metricName.description()
93              ));
94          });
95          builder.append("\n----------------------------------------\n");
96          logger.info(builder.toString());
97      }
98
```

- Give a nice format so we can read metrics easily
- Give some space and some easy indicators to find in log

Pretty Print Metrics Output

```
        batch-size-avg      146.42          The average number of bytes sent per partition per-reque
    incoming-byte-rate      6,234.05        Bytes/second read off all sockets
       network-io-rate      239.08          The average number of network operations (reads or write
  record-queue-time-avg     0.10            The average time in ms record batches spent in the recor
      record-send-rate      217.05          The average number of records sent per second.
records-per-request-avg     1.82            The average number of records per request.
      request-size-max      963.00          The maximum size of any request sent in the window.
    requests-in-flight      0.00            The current number of in-flight requests awaiting a resp
        response-rate       119.55          Responses received sent per second.
    incoming-byte-rate      4.13
      request-size-max      55.00           The maximum size of any request sent in the window.
         response-rate      0.06            The average number of responses received per second.
      record-send-rate      217.73
```

```
            sent to topic=stock-prices part=1 off=160219 at time=Tue May 23 16:42:40 PDT 2017
16:42:40.577 [pool-1-thread-1] INFO  c.c.k.p.MetricsProducerReporter -
----------------------------------------
                    producer-metrics.batch-size-avg          batch-size-avg     146.42          The average number of bytes
                producer-metrics.incoming-byte-rate      incoming-byte-rate     6,234.05        Bytes/second read off all so
                   producer-metrics.network-io-rate         network-io-rate     239.08          The average number of networ
             producer-metrics.record-queue-time-avg   record-queue-time-avg     0.10            The average time in ms recor
                  producer-metrics.record-send-rate        record-send-rate     217.05          The average number of record
           producer-metrics.records-per-request-avg  records-per-request-avg     1.82            The average number of record
                  producer-metrics.request-size-max        request-size-max     963.00          The maximum size of any requ
                producer-metrics.requests-in-flight      requests-in-flight     0.00            The current number of in-fli
                   producer-metrics.response-rate           response-rate     119.55          Responses received sent per
           producer-node-metrics.incoming-byte-rate      incoming-byte-rate     4.13
             producer-node-metrics.request-size-max        request-size-max     55.00           The maximum size of any requ
                producer-node-metrics.response-rate           response-rate     0.06            The average number of respor
            producer-topic-metrics.record-send-rate        record-send-rate     217.73

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

Validate Partition Replication

replica-verification.sh

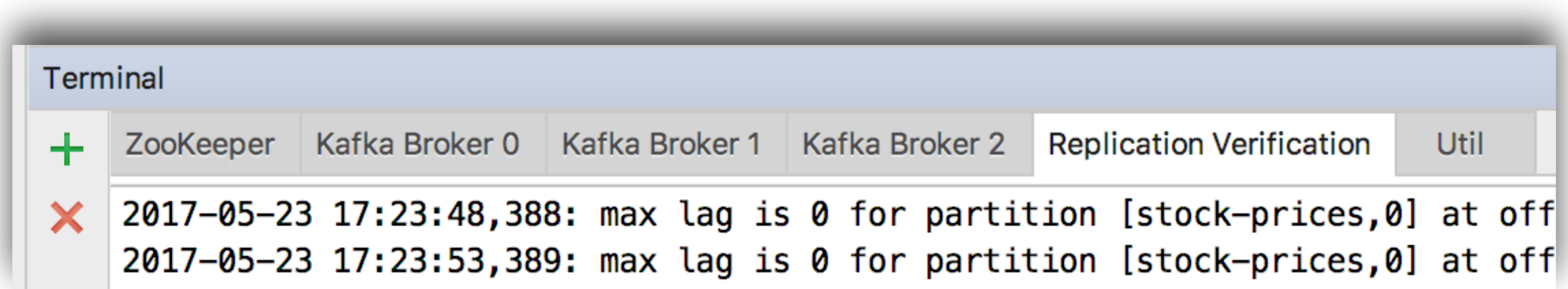
```
#!/usr/bin/env bash

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
```

```
$ bin/replica-verification.sh
2017-05-23 17:18:18,350: verification process is started.
2017-05-23 17:18:23,293: max lag is 0 for partition [stock-prices,0] at offset 562262 among 3 partitions
2017-05-23 17:18:28,294: max lag is 0 for partition [stock-prices,0] at offset 562262 among 3 partitions
2017-05-23 17:18:33,295: max lag is 0 for partition [stock-prices,0] at offset 562262 among 3 partitions
2017-05-23 17:18:38,296: max lag is 0 for partition [stock-prices,0] at offset 562262 among 3 partitions
```

Checks lag every 5 seconds for stock-price

Run it. Run Servers. Run Producer. Kill Brokers

- If not already, startup ZooKeeper and three Kafka brokers
- Run ***StockPriceKafkaProducer (ensure acks are set to all first)***
- ***Start*** and ***stop*** different Kafka Brokers while ***StockPriceKafkaProducer runs***, observe metrics, observe changes, Run replication verification in one terminal and check topics stats in another



Observe Partitions Getting Behind

Terminal

ZooKeeper | Kafka Broker 0 | Kafka Broker 1 | Kafka Broker 2 | Replication Verification | Util

```
2017-05-23 17:27:18,251: max lag is 3 for partition [stock-prices,2] at offset 534376
2017-05-23 17:27:23,775: max lag is 3 for partition [stock-prices,2] at offset 534376
2017-05-23 17:27:30,275: max lag is 3 for partition [stock-prices,2] at offset 534376
```

Terminal

ZooKeeper | Kafka Broker 0 | Kafka Broker 1 | Kafka Broker 2 | Replication Verification | Util

```
~/kafka-training/lab5/solution
$ bin/describe-topics.sh
Topic:stock-prices      PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: stock-prices     Partition: 0    Leader: 1       Replicas: 0,1,2 Isr: 2,1
        Topic: stock-prices     Partition: 1    Leader: 1       Replicas: 1,2,0 Isr: 2,1
        Topic: stock-prices     Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 2,1
```

```
        batch-size-avg     112.01          The average number of bytes sent per partition per-
    incoming-byte-rate     85,626.41       Bytes/second read off all sockets
       network-io-rate     3,139.14        The average number of network operations (reads or
 record-queue-time-avg     0.03            The average time in ms record batches spent in the
      record-send-rate     2,436.49        The average number of records sent per second.
records-per-request-avg    1.55            The average number of records per request.
      request-size-max     1,695.00        The maximum size of any request sent in the window.
    requests-in-flight     0.00            The current number of in-flight requests awaiting a
         response-rate     1,569.57        Responses received sent per second.
    incoming-byte-rate     49,889.92
      request-size-max     556.00          The maximum size of any request sent in the window.
         response-rate     687.12          The average number of responses received per second
      record-send-rate     2,436.32
```

Recover

Terminal

ZooKeeper | Kafka Broker 0 | Kafka Broker 1 | Kafka Broker 2 | Replication Verification | Util

```
org.apache.kafka.common.errors.NotEnoughReplicasException: Number of insync replicas
[2017-05-23 17:31:23,210] INFO Partition [stock-prices,0] on broker 1: Expanding ISR
[2017-05-23 17:31:23,216] INFO Partition [stock-prices,1] on broker 1: Expanding ISR
```

Output of 2nd Broker Recovering

Terminal

ZooKeeper | Kafka Broker 0 | Kafka Broker 1 | Kafka Broker 2 | Replication Verification | Util

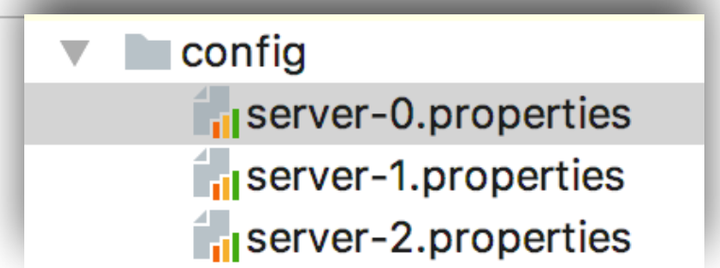
```
2017-05-23 17:31:22,996: max lag is 3 for partition [stock-prices,2] at offset 534376 among 3 partitions
2017-05-23 17:31:27,997: max lag is 1 for partition [stock-prices,0] at offset 629181 among 3 partitions
2017-05-23 17:31:32,998: max lag is 1 for partition [stock-prices,2] at offset 544108 among 3 partitions
2017-05-23 17:31:37,999: max lag is 2 for partition [stock-prices,2] at offset 549173 among 3 partitions
2017-05-23 17:31:43,000: max lag is 2 for partition [stock-prices,0] at offset 646785 among 3 partitions
2017-05-23 17:31:48,001: max lag is 1 for partition [stock-prices,2] at offset 559380 among 3 partitions
2017-05-23 17:31:53,002: max lag is 1 for partition [stock-prices,1] at offset 188445 among 3 partitions
2017-05-23 17:31:58,003: max lag is 1 for partition [stock-prices,2] at offset 569534 among 3 partitions
2017-05-23 17:32:03,004: max lag is 1 for partition [stock-prices,2] at offset 574566 among 3 partitions
2017-05-23 17:32:08,005: max lag is 1 for partition [stock-prices,0] at offset 676390 among 3 partitions
2017-05-23 17:32:13,006: max lag is 0 for partition [stock-prices,0] at offset 678579 among 3 partitions
```

Replication Verification

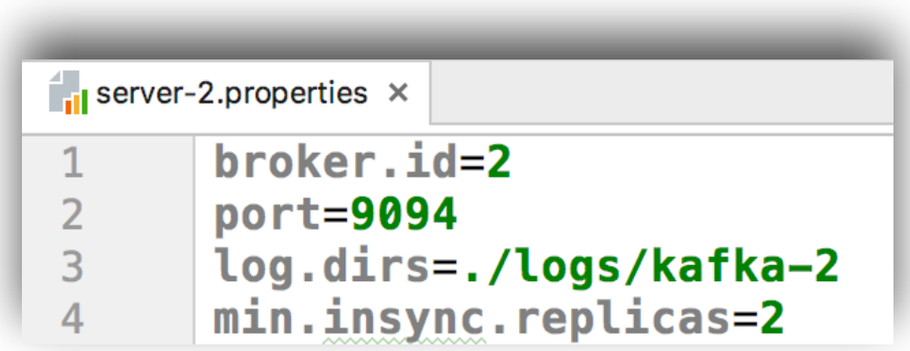
```
        Topic: stock-prices     Partition: 1    Leader: 1       Replicas: 1,2,0 Isr: 2,1,0
        Topic: stock-prices     Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 2,1,0
~/kafka-training/lab5/solution
$ bin/describe-topics.sh
Topic:stock-prices      PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: stock-prices     Partition: 0    Leader: 0       Replicas: 0,1,2 Isr: 2,1,0
        Topic: stock-prices     Partition: 1    Leader: 1       Replicas: 1,2,0 Isr: 2,1,0
        Topic: stock-prices     Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 2,1,0
```

Describe Topics

Run it. Run Servers. Run Producer. Kill Brokers



- Stop all Kafka Brokers (Kafka servers)
- Change ***min.insync.replicas*=3 *to* *min.insync.replicas*=2**
  - config files for broker are in lab directory under config
- Startup ZooKeeper if needed and three Kafka brokers
- Run ***StockPriceKafkaProducer (ensure acks are set to all first)***
- ***Start*** and ***stop*** different Kafka Brokers while ***StockPriceKafkaProducer runs,***
- Observe metrics, observe changes
- Run replication verification in one terminal and check topics stats in another with describe-topics.sh in another terminal



Expected outcome

- Producer will work even if one broker goes down.
- Producer will not work if two brokers go down because ***min.insync.replicas*=2,** two replicas have to be up besides ***leader***

Terminal

ZooKeeper | Kafka Broker 0 | Kafka Broker 1 | Kafka Broker 2 | Replication Verification | Util

```
2017-05-23 17:52:36,712: max lag is 92137 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:52:41,715: max lag is 97936 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:52:46,717: max lag is 103663 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:52:51,721: max lag is 109411 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:52:56,724: max lag is 115130 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:53:01,728: max lag is 120829 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:53:06,730: max lag is 126644 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:53:12,452: max lag is 132302 for partition [stock-prices,0] at offset 17947 among 3 partitions
```

Since Producer can run with 1 down broker, the replication lag can get really far behind.

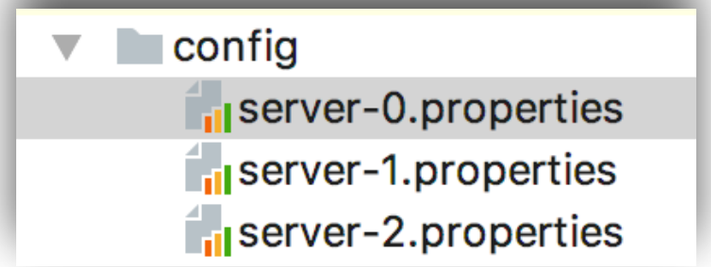
Terminal

ZooKeeper | Kafka Broker 0 | Kafka Broker 1 | Kafka Broker 2 | Replication Verification | Util

```
2017-05-23 17:54:31,583: max lag is 134183 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:54:36,586: max lag is 134183 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:54:43,828: max lag is 134183 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:54:48,831: max lag is 134183 for partition [stock-prices,0] at offset 17947 among 3 partitions
2017-05-23 17:54:53,832: max lag is 0 for partition [stock-prices,0] at offset 152130 among 3 partitions
2017-05-23 17:54:58,833: max lag is 0 for partition [stock-prices,0] at offset 152130 among 3 partitions
2017-05-23 17:55:03,834: max lag is 0 for partition [stock-prices,0] at offset 152130 among 3 partitions
```

When you startup "failed" broker, it catches up really fast.

Change min.insync.replicas back



- Shutdown all brokers
- Change back min.insync.replicas=3
  - Broker config for servers
  - Do this for all of the servers
- Start ZooKeeper if needed
- Start brokers back up

Modify bin/create-topic.sh

- Modify ***bin/create-topic.sh***
  - add --config min.insync.replicas=2
  - Add this as param to kafka-topics.sh
- Run delete-topic.sh
- Run create-topic.sh

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
```

delete-topic.sh

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

Recreate Topic

- Run delete-topic.sh
- Run create-topic.sh

Terminal

ZooKeeper | Kafka Broker 0 | Kafka Broker 1 | Kafka Broker 2 | Replication Verification | Util

```
~/kafka-training/lab5/solution
$ bin/delete-topic.sh
Topic stock-prices is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
~/kafka-training/lab5/solution
$ bin/create-topic.sh
Created topic "stock-prices".
```

Run it. Run Servers. Run Producer. Kill Brokers

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

Expected Results

- 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)

Terminal

ZooKeeper | Kafka Broker 0 | Kafka Broker 1 | Kafka Broker 2 | Replication Verification | Util

```
2017-05-23 18:18:15,249: max lag is 2 for partition [stock-prices,2] at offset 41599 among 3 partitions
2017-05-23 18:18:20,250: max lag is 2 for partition [stock-prices,2] at offset 46632 among 3 partitions
2017-05-23 18:18:25,251: max lag is 1 for partition [stock-prices,0] at offset 60164 among 3 partitions
2017-05-23 18:18:31,854: max lag is 552 for partition [stock-prices,0] at offset 65034 among 3 partitions
2017-05-23 18:18:37,368: max lag is 7702 for partition [stock-prices,0] at offset 65034 among 3 partitions
2017-05-23 18:18:43,548: max lag is 14718 for partition [stock-prices,0] at offset 65034 among 3 partitions
2017-05-23 18:18:49,381: max lag is 22185 for partition [stock-prices,0] at offset 65034 among 3 partitions
2017-05-23 18:18:54,384: max lag is 28808 for partition [stock-prices,0] at offset 65034 among 3 partitions
2017-05-23 18:18:59,387: max lag is 34474 for partition [stock-prices,0] at offset 65034 among 3 partitions
2017-05-23 18:19:09,293: max lag is 39960 for partition [stock-prices,0] at offset 65034 among 3 partitions
2017-05-23 18:19:14,295: max lag is 51912 for partition [stock-prices,0] at offset 65034 among 3 partitions
2017-05-23 18:19:19,296: max lag is 2 for partition [stock-prices,0] at offset 122690 among 3 partitions
```

Lab Batching Records

Objectives

- Disable batching and observer metrics
- Enable batching and observe metrics
- Increase batch size and linger and observe metrics
- Run consumer to see batch sizes change
- Enable compression, observe results

SimpleStockPriceConsumer

- We added a ***SimpleStockPriceConsumer*** to consume StockPrices and display batch lengths for ***poll***()
  - We won't cover in detail just quickly since this is a Producer lab not a Consumer lab. :)
- Run this 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

SimpleStockPriceConsumer

SimpleStockPriceConsumer.java

```
SimpleStockPriceConsumer
11
12  public class SimpleStockPriceConsumer {
13
14      private static Consumer<String, StockPrice> createConsumer() {
15          final Properties props = new Properties();
16          props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
17                  StockAppConstants.BOOTSTRAP_SERVERS);
18          props.put(ConsumerConfig.GROUP_ID_CONFIG,
19                  "KafkaExampleConsumer");
20          props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
21                  StringDeserializer.class.getName());
22          //Custom Deserializer
23          props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
24                  StockDeserializer.class.getName());
25          props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
26          // Create the consumer using props.
27          final Consumer<String, StockPrice> consumer =
28                  new KafkaConsumer<>(props);
29          // Subscribe to the topic.
30          consumer.subscribe(Collections.singletonList(
31                  StockAppConstants.TOPIC));
32          return consumer;
33      }
```

- Similar to other Consumer examples so far
- Subscribes to ***stock-prices*** topic
- Has custom serializer

SimpleStockPriceConsumer.runConsumer

SimpleStockPriceConsumer.java

SimpleStockPriceConsumer

```
35
36      static void runConsumer() throws InterruptedException {
37          final Consumer<String, StockPrice> consumer = createConsumer();
38          final Map<String, StockPrice> map = new HashMap<>();
39          try {
40              final int giveUp = 1000; int noRecordsCount = 0;
41              int readCount = 0;
42              while (true) {
43                  final ConsumerRecords<String, StockPrice> consumerRecords =
44                          consumer.poll( timeout: 1000);
45                  if (consumerRecords.count() == 0) {
46                      noRecordsCount++;
47                      if (noRecordsCount > giveUp) break;
48                      else continue;
49                  }
50                  readCount++;
51                  consumerRecords.forEach(record -> {
52                      map.put(record.key(), record.value());
53                  });
54                  if (readCount % 100 == 0) {
55                      displayRecordsStatsAndStocks(map, consumerRecords);
56                  }
57                  consumer.commitAsync();
58              }
59          }
```

- Drains topic; Creates map of current stocks; Calls ***displayRecordsStatsAndStocks()***

SimpleStockPriceConsumer.display...(

SimpleStockPriceConsumer.java

SimpleStockPriceConsumer | displayRecordsStatsAndStocks()

```
66  private static void displayRecordsStatsAndStocks(
67          final Map<String, StockPrice> stockPriceMap,
68          final ConsumerRecords<String, StockPrice> consumerRecords) {
69      System.out.printf("New ConsumerRecords par count %d count %d\n",
70              consumerRecords.partitions().size(),
71              consumerRecords.count());
72      stockPriceMap.forEach((s, stockPrice) ->
73              System.out.printf("ticker %s price %d.%d \n",
74                  stockPrice.getName(),
75                  stockPrice.getDollars(),
76                  stockPrice.getCents()));
77      System.out.println();
78  }
```

- Prints out size of each partition read and total record count
- Prints out each stock at its current price

StockDeserializer

SimpleStockPriceConsumer.java × | StockDeserializer.java ×

StockDeserializer

```
3   import ...
8
9   public class StockDeserializer implements Deserializer<StockPrice> {
10
11      @Override
12      public StockPrice deserialize(final String topic, final byte[] data) {
13          return new StockPrice(new String(data, StandardCharsets.UTF_8));
14      }
15
16      @Override
17      public void con
18      }
19
20      @Override
21      public void clo
22      }
23  }
```

SimpleStockPriceConsumer.java × | StockDeserializer.java × | StockPrice.java ×

StockPrice

```
1   package com.cloudurable.kafka.producer.model;
2
3   import io.advantageous.boon.json.JsonFactory;
4
5   public class StockPrice {
6
7       private final int dollars;
8       private final int cents;
9       private final String name;
10
11      public StockPrice(final String json) {
12          this(JsonFactory.fromJson(json, StockPrice.class));
13      }
14
```

Disable batching

```
StockPriceKafkaProducer
19  public class StockPriceKafkaProducer {
20
21      private static void setupBootstrapAndSerializers(Propert
22          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
23                  StockAppConstants.BOOTSTRAP_SERVERS);
24          props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPri
25          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
26                  StringSerializer.class.getName());
27
28          setupBatchingAndCompression(props);
29
```

```
StockPriceKafkaProducer
38
39      private static void setupBatchingAndCompression(
40              final Properties props) {
41          props.put(ProducerConfig.BATCH_SIZE_CONFIG,  0);
42      }
43
```

- Start by disabling batching
- This turns batching off
- Run this
- Check Consumer and stats

Metrics No Batch

```
        batch-size-avg      80.48        The average number of bytes sent per partition per-re
    incoming-byte-rate      28,726.21    Bytes/second read off all sockets
       network-io-rate      1,104.59     The average number of network operations (reads or wr
  record-queue-time-avg     268.69       The average time in ms record batches spent in the re
      record-send-rate      552.28       The average number of records sent per second.
records-per-request-avg     1.00         The average number of records per request.
      request-size-max      157.00       The maximum size of any request sent in the window.
    requests-in-flight      11.00        The current number of in-flight requests awaiting a r
        response-rate       552.20       Responses received sent per second.
    incoming-byte-rate      2.83
      request-size-max      55.00        The maximum size of any request sent in the window.
        response-rate       0.04         The average number of responses received per second.
      record-send-rate      553.19
```

```
New ConsumerRecords par count 3 count 6
ticker AAA price 59.55
ticker GOOG price 411.12
ticker ABC price 52.66
ticker CCC price 92.47
ticker BBB price 77.41
ticker EEE price 72.67
ticker DEF price 94.23
ticker DDD price 85.29
ticker FFF price 93.42
ticker SUN price 57.91
ticker INEL price 71.36
ticker IBM price 50.18
ticker XYZ price 56.82
ticker UBER price 263.72

New ConsumerRecords par count 3 count 11
ticker AAA price 91.22
```

```
New ConsumerRecords par count 1 count 1
ticker AAA price 83.13
ticker GOOG price 480.80
```

```
New ConsumerRecords par count 3 count 3
ticker AAA price 69.24
ticker GOOG price 442.19
```

Records per poll averages around 4
Batch Size is 80

Set batching to 16K

StockPriceKafkaProducer.java

StockPriceKafkaProducer | setupBatchingAndCompression()

```
38
39      private static void setupBatchingAndCompression(
40              final Properties props) {
41          props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384);
42      }
43
```

- Set the batch size to 16K
- Run this
- Check Consumer and stats

Set batching to 16K Results

16K Batch Size

batch-size-avg	136.02
incoming-byte-rate	33,028.83
network-io-rate	1,270.03
record-queue-time-avg	0.52
record-send-rate	1,072.34
records-per-request-avg	1.69
request-size-max	2,573.00
requests-in-flight	7.00
response-rate	634.94
incoming-byte-rate	2.52
request-size-max	55.00
response-rate	0.04
record-send-rate	1,073.69

No Batch

80.48	The average number of bytes sent per partition per-re
28,726.21	Bytes/second read off all sockets
1,104.59	The average number of network operations (reads or wr
268.69	The average time in ms record batches spent in the re
552.28	The average number of records sent per second.
1.00	The average number of records per request.
157.00	The maximum size of any request sent in the window.
11.00	The current number of in-flight requests awaiting a r
552.20	Responses received sent per second.
2.83	
55.00	The maximum size of any request sent in the window.
0.04	The average number of responses received per second.
553.19	

Consumer Records per poll averages around 7.5
Batch Size is now 136.02
59% more batching
Look how much the request queue time shrunk!

Look at the
record-send-rate
200% faster!

Set batching to 16K and linger to 10ms

StockPriceKafkaProducer.java

```
StockPriceKafkaProducer
39      private static void setupBatchingAndCompression(
40              final Properties props) {
41          props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
42          props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384);
43      }
44
```

- Set the batch size to 16K and linger to 10ms
- Run this
- Check Consumer and Stats

Results batch size 16K linger 10ms

	16K and 10ms linger	16K	No Batch	
batch-size-avg	796.60	136.02	80.48	The average number of bytes sent
incoming-byte-rate	3,584.17	33,028.83	28,726.21	Bytes/second read off all sockets
network-io-rate	137.21	1,270.03	1,104.59	The average number of network ope
record-queue-time-avg	10.55	0.52	268.69	The average time in ms record bat
record-send-rate	676.95	1,072.34	552.28	The average number of records sen
records-per-request-avg	9.89	1.69	1.00	The average number of records per
request-size-max	2,578.00	2,573.00	157.00	The maximum size of any request s
response-rate	68.60	7.00	11.00	The current number of in-flight r
incoming-byte-rate	3.24	634.94	552.20	Responses received sent per secon
request-size-max	37.00	2.52	2.83	
response-rate	0.02	55.00	55.00	The maximum size of any request s
record-send-rate	678.38	0.04	0.04	The average number of responses r
		1,073.69	553.19	

Consumer Records per poll averages around 17
Batch Size is now 796
585% more batching

Look at the **record-send-rate** went down but higher than start

Set batching to 64K and linger to 1 second

StockPriceKafkaProducer.java

```
StockPriceKafkaProducer
39      private static void setupBatchingAndCompression(
40              final Properties props) {
41          props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
42          props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384);
43      }
44
```

- Set the batch size to 64K and linger to 1 second
- Run this
- Check Consumer and Stats

Results batch size 64K linger 1s

	64K/1s	16K/10ms	16K	No Batch	
batch-size-avg	40,270.82	796.60	136.02	80.48	The average number of bytes sent
incoming-byte-rate	63.05	3,584.17	33,028.83	28,726.21	Bytes/second read off all sockets
network-io-rate	1.86	137.21	1,270.03	1,104.59	The average number of network ope
record-queue-time-avg	1,002.64	10.55	0.52	268.69	The average time in ms record bat
record-send-rate	392.69	676.95	1,072.34	552.28	The average number of records sen
records-per-request-avg	500.00	9.89	1.69	1.00	The average number of records per
request-size-max	56,597.00	2,578.00	2,573.00	157.00	The maximum size of any request s
response-rate	0.93	68.60	7.00	11.00	The current number of in-flight r
incoming-byte-rate	9.27	3.24	634.94	552.20	Responses received sent per secon
request-size-max	37.00	37.00	2.52	2.83	
response-rate	0.02	0.02	55.00	55.00	The maximum size of any request s
record-send-rate	402.99	678.38	0.04	0.04	The average number of responses r
			1,073.69	553.19	

Consumer Records per poll averages around 500
Batch Size is now 40K
Record Queue Time is very high :(

Look at the **network-io-rate**

Set batching to 64K and linger to 100 ms

StockPriceKafkaProducer.java

StockPriceKafkaProducer | setupBatchingAndCompression()

```
39      private static void setupBatchingAndCompression(
40              final Properties props) {
41          props.put(ProducerConfig.LINGER_MS_CONFIG, 100);
42          props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384 * 4);
43      }
44
```

- Set the batch size to 64K and linger to 100ms second
- Run this
- Check Consumer and Stats

Results batch size 64K linger 100ms

	64K/100ms	64K/1s	16K/10ms	16K
batch-size-avg	6,205.69	40,270.82	796.60	80.48
incoming-byte-rate	1,505.54	63.05	3,584.17	28,726.21
network-io-rate	57.11	1.86	137.21	1,104.59
record-queue-time-avg	102.98	1,002.64	10.55	268.69
record-send-rate	2,184.78	392.69	676.95	552.28
records-per-request-avg	77.05	500.00	9.89	1.00
request-size-max	11,780.00	56,597.00	2,578.00	157.00
response-rate	28.56	0.93	68.60	11.00
incoming-byte-rate	2.27	9.27	3.24	552.20
request-size-max	55.00	37.00	37.00	2.83
response-rate	0.07	0.02	0.02	55.00
record-send-rate	1,096.57	402.99	678.38	0.04
				553.19

64K batch size
100ms Linger
has the highest record-send-rate!

Turn on compression snappy, 50ms, 64K

StockPriceKafkaProducer.java

```
StockPriceKafkaProducer
39      private static void setupBatchingAndCompression(
40              final Properties props) {
41          props.put(ProducerConfig.LINGER_MS_CONFIG, 50);
42          props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384 * 4);
43          props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
44      }
```

- Enable compression
- Set the batch size to 64K and linger to 50ms second
- Run this
- Check Consumer and Stats

Results batch size Snappy linger 10ms

64K/100ms

batch-size-avg	6,205.69
incoming-byte-rate	1,505.54
network-io-rate	57.11
record-queue-time-avg	102.98
record-send-rate	2,184.78
records-per-request-avg	77.05
request-size-max	11,780.00
response-rate	28.56
incoming-byte-rate	2.27
request-size-max	55.00
response-rate	0.07
record-send-rate	1,096.57

Snappy 64K/50ms

batch-size-avg	1,520.76
incoming-byte-rate	2,846.67
network-io-rate	108.95
record-queue-time-avg	52.68
record-send-rate	2,261.90
records-per-request-avg	41.61
request-size-max	3,492.00
requests-in-flight	3.00
response-rate	54.44
incoming-byte-rate	3.49
request-size-max	55.00
response-rate	0.05
record-send-rate	2,270.03

Snappy 64K/50ms
has the highest record-send-rate and
1/2 the queue time!

Lab Adding Retries and Timeouts

Objectives

- Setup timeouts
- Setup retries
- Setup retry back off
- Setup inflight messages to 1 so retries don't store records out of order

Run it. Run Servers. Run Producer. Kill Brokers

- Startup ZooKeeper if needed and three Kafka brokers
- Modify ***StockPriceKafkaProducer*** to configure retry, timeouts, in-flight message count and retry back off
- Run ***StockPriceKafkaProducer***
- ***Start*** and ***stop any two*** different Kafka Brokers while ***StockPriceKafkaProducer runs,***
- Notice retry messages in log of ***StockPriceKafkaProducer***

Modify StockPriceKafkaProducer

StockPriceKafkaProducer

```
21      private static Producer<String, StockPrice>
22                                     createProducer() {
23          final Properties props = new Properties();
24          setupBootstrapAndSerializers(props);
25          setupBatchingAndCompression(props);
26
27          setupRetriesInFlightTimeout(props);
28
29          return new KafkaProducer<>(props);
```

StockPriceKafkaProducer | setupRetriesInFlightTimeout()

```
32      private static void setupRetriesInFlightTimeout(Properties props) {
33          //Only two in-flight messages per Kafka broker connection
34          // - max.in.flight.requests.per.connection (default 5)
35          props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
36                  1);
37          //Set the number of retries - retries
38          props.put(ProducerConfig.RETRIES_CONFIG, 3);
39
40          //Request timeout - request.timeout.ms
41          props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 15_000);
42
43          //Only retry after one second.
44          props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 1_000);
```

- to configure retry, timeouts, in-flight message count and retry back off

Expected output after 2 broker shutdown

```
- Got error produce response with correlation id 643 on topic-partition stock-prices-1, retrying (1 attempts left). Error: NOT_ENOUGH_REPLICAS
- Got error produce response with correlation id 643 on topic-partition stock-prices-2, retrying (2 attempts left). Error: NOT_ENOUGH_REPLICAS
- Got error produce response with correlation id 643 on topic-partition stock-prices-0, retrying (1 attempts left). Error: NOT_ENOUGH_REPLICAS
- Got error produce response with correlation id 645 on topic-partition stock-prices-1, retrying (0 attempts left). Error: NOT_ENOUGH_REPLICAS
- Got error produce response with correlation id 645 on topic-partition stock-prices-2, retrying (1 attempts left). Error: NOT_ENOUGH_REPLICAS
- Got error produce response with correlation id 645 on topic-partition stock-prices-0, retrying (0 attempts left). Error: NOT_ENOUGH_REPLICAS
```

Run all.
Kill any two servers.
Look for retry messages.

Restart them and see that it recovers

Also use replica verification to see when broker catches up

```
retrying (1 attempts left). Error: NOT_ENOUGH_REPLICAS
retrying (2 attempts left). Error: NOT_ENOUGH_REPLICAS
retrying (1 attempts left). Error: NOT_ENOUGH_REPLICAS
retrying (0 attempts left). Error: NOT_ENOUGH_REPLICAS
retrying (1 attempts left). Error: NOT_ENOUGH_REPLICAS
retrying (0 attempts left). Error: NOT_ENOUGH_REPLICAS
```

WARN Inflight Message Count

- ***MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION "max.in.flight.requests.per.connection"***
- max number of unacknowledged requests client sends on a single connection before blocking
- If >1 and
  - failed sends, then
  - Risk message re-ordering on partition during retry attempt
  - Depends on use but for StockPrices not good, you should pick retries > 1 or inflight > 1 but not both. Avoid duplicates. :)
  - June 2017 release might fix this with sequence from producer

Lab Write ProducerInterceptor

Objectives

- Setup an interceptor for request sends
- Create ProducerInterceptor
- Implement onSend
- Implement onAcknowledge

Producer Interception

- ***Producer*** config property: ***interceptor.classes***
  - empty (you can pass an comma delimited list)
- interceptors implementing ProducerInterceptor interface
- intercept records producer sent to broker and after acks
- you could mutate records

KafkaProducer - Interceptor Config

StockPriceKafkaProducer.java

StockPriceKafkaProducer | getStockSenderList()

```
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Sto
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27          //Install interceptor list - config "interceptor.classes"
28          props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
29                  StockProducerInterceptor.class.getName());
30
31          //Set number of acknowledgments - acks - default is all
32          props.put(ProducerConfig.ACKS_CONFIG, "all");
33
34          return new KafkaProducer<>(props);
35      }
```

KafkaProducer ProducerInterceptor

StockProducerInterceptor.java

```
StockProducerInterceptor
10
11  public class StockProducerInterceptor implements ProducerInterceptor {
12
13      private final Logger logger = LoggerFactory
14              .getLogger(StockProducerInterceptor.class);
15      private int onSendCount;
16      private int onAckCount;
17
18      @Override
19      public ProducerRecord onSend(final ProducerRecord record) {...}
36
37      @Override
38      public void onAcknowledgement(final RecordMetadata metadata,
39                                    final Exception exception) {...}
54
55      @Override
56      public void close() {...}
59
60      @Override
61      public void configure(Map<String, ?> configs) {...}
64  }
```

ProducerInterceptor onSend

StockProducerInterceptor.java

StockProducerInterceptor

```
18      @Override
19      public ProducerRecord onSend(final ProducerRecord record) {
20          onSendCount++;
21          if (logger.isDebugEnabled()) {
22              logger.debug(String.format("onSend topic=%s key=%s value=%s %d \n",
23                      record.topic(), record.key(), record.value().toString(),
24                      record.partition()
25              ));
26          } else {
27              if (onSendCount % 100 == 0) {
28                  logger.info(String.format("onSend topic=%s key=%s value=%s %d \n",
29                          record.topic(), record.key(), record.value().toString(),
30                          record.partition()
31                  ));
32              }
33          }
34          return record;
35      }
```

Output

```
onSend topic=stock-prices2 key=UBER value=StockPrice{dollars=737,
                    cents=78, name='UBER'} null
```

ProducerInterceptor onAck

StockProducerInterceptor.java

StockProducerInterceptor | onAcknowledgement()

```
37      @Override
38      public void onAcknowledgement(final RecordMetadata metadata,
39                                    final Exception exception) {
40          onAckCount++;
41
42          if (logger.isDebugEnabled()) {
43              logger.debug(String.format("onAck topic=%s, part=%d, offset=%d\n",
44                      metadata.topic(), metadata.partition(), metadata.offset()
45              ));
46          } else {
47              if (onAckCount % 100 == 0) {
48                  logger.info(String.format("onAck topic=%s, part=%d, offset=%d\n",
49                          metadata.topic(), metadata.partition(), metadata.offset()
50                  ));
51              }
52          }
53      }
```

Output

```
onAck topic=stock-prices2, part=0, offset=18360
```

ProducerInterceptor the rest

StockProducerInterceptor.java

StockProducerInterceptor | configure()

```
48                  logger.info(String.format("onAck topic=%s,
49                          metadata.topic(), metadata.partitio
50                  ));
51              }
52          }
53      }
54
55      @Override
56      public void close() {
57      }
58
59      @Override
60      public void configure(Map<String, ?> configs) {
61      }
62  }
```

Run it. Run Servers. Run Producer.

- Startup ZooKeeper if needed
- Start or restart Kafka brokers
- Run ***StockPriceKafkaProducer***
- Look for log message from ***ProducerInterceptor*** in output

ProducerInterceptor Output

Debug StockPriceKafkaProducer

Debugger Console

```
21:02:22.892 [kafka-producer-network-thread | StockPriceKafkaProducer] INFO  c.c.k.p.StockProducerInterceptor -
onAck topic=stock-prices, part=2, offset=-1
```

Debug StockPriceKafkaProducer

Debugger Console

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

21:03:26.909 [pool-1-thread-5] INFO  c.c.k.p.StockProducerInterceptor -
onSend topic=stock-prices key=INEL value=StockPrice{dollars=72, cents=13, name='INEL'} null
```

Lab Write Custom Partitioner

Objectives

- Create StockPricePartitioner
  - Implements interface ***Partitioner***
  - Implement ***partition***() method
  - Implement ***configure***() method with ***importantStocks*** property
- Configure new ***Partitioner*** in ***Producer config*** with property ProducerConfig.INTERCEPTOR_CLASSES_CONFIG
  - Pass config property ***importantStocks***

Producer Partitioning

- ***Producer*** config property: ***partitioner.class***
  - org.apache.kafka.clients.producer.internals.DefaultPartitioner
- Partitioner class implements Partitioner interface
- partition() method takes topic, key, value, and cluster
  - returns partition number for record

StockPricePartitioner

StockPriceKafkaProducer.java × | StockPricePartitioner.java ×

StockPricePartitioner | StockPricePartitioner()

```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,
                         byte[] keyBytes, final Object value,
                         final byte[] valueBytes,
                         final Cluster cluster) {...}

    @Override
    public void close() {
    }

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

StockPricePartitioner.configure()

StockPriceKafkaProducer.java × | StockPricePartitioner.java ×

StockPricePartitioner | configure()

```
39
40      @Override
41      public void close() {
42      }
43
44      @Override
45      public void configure(Map<String, ?> configs) {
46          final String importantStocksStr = (String) configs.get("importantStocks");
47          Arrays.stream(importantStocksStr.split( regex: ","))
48                  .forEach(importantStocks::add);
49      }
50
```

- Implement *configure*() method
  - with *importantStocks* property
  - importantStocks get added to importantStocks HashSet

StockPricePartitioner partition()

- IMPORTANT STOCK: If ***stockName*** is in ***importantStocks*** HashSet then put it in ***partitionNum*** = ***(partitionCount -1)*** (last partition)
- REGULAR STOCK: Otherwise if not in importantStocks then not important use 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)

StockPricePartitioner partition()

StockPriceKafkaProducer.java × | StockPricePartitioner.java ×

StockPricePartitioner | partition()

```java
    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

StockPriceKafkaProducer.java

StockPriceKafkaProducer | createProducer()

```
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Sto
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27          //Install partitioner -- "partitioner.class"
28          props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
29                  StockPricePartitioner.class.getName());
30
31          props.put("importantStocks", "IBM,UBER");
```

- Configure new ***Partitioner*** in ***Producer config*** with property ProducerConfig.INTERCEPTOR_CLASSES_CONFIG
  - Pass config property importantStocks
  - importantStock are the ones that go into priority queue

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