Lab 5.4: Kafka Metrics and Replication Verification

Welcome to the session 5 lab 4. The work for this lab is done in `~/kafka-training/lab5.4`. In this lab, you are going to set up Kafka metrics and replication verification.

Please refer to the Kafka course notes for any updates or changes to this lab.

Find the lastest version of this lab here.

Lab Adding Producer Metrics and Replication Verification

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

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

Create Producer Metrics Monitor

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

~/kafka-training/lab5.4/src/main/java/com/cloudurable/kafka/producer/MetricsProducerReporter.java

Kafka Producer: MetricsProducerReporter for reporting metrics is Runnable

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

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

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

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

    //Used to Filter just the metrics we want
    private final Set<String> metricsNameFilter = Sets.set(
            "record-queue-time-avg", "record-send-rate", "records-per-request-avg",
```

```
            "request-size-max", "network-io-rate", "record-queue-time-avg",
            "incoming-byte-rate", "batch-size-avg", "response-rate", "requests-in-
flight"
    );

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

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

~/kafka-training/lab5.4/src/main/java/com/cloudurable/kafka/producer/MetricsProducerReporter.java

Kafka Producer: MetricsProducerReporter calls producer.metrics()

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

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

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

  private void displayMetrics(Map<MetricName, ? extends Metric> metrics) {
    final Map<String, MetricPair> metricsDisplayMap = metrics.entrySet().stream()
            //Filter out metrics not in metricsNameFilter
            .filter(metricNameEntry ->
                    metricsNameFilter.contains(metricNameEntry.getKey().name()))
            //Filter out metrics not in metricsNameFilter
            .filter(metricNameEntry ->
                    !Double.isInfinite(metricNameEntry.getValue().value()) &&
                            !Double.isNaN(metricNameEntry.getValue().value()) &&
                            metricNameEntry.getValue().value() != 0
            )
            //Turn Map<MetricName,Metric> into TreeMap<String, MetricPair>
```

```
            .map(entry -> new MetricPair(entry.getKey(), entry.getValue()))
            .collect(Collectors.toMap(
                    MetricPair::toString, it -> it, (a, b) -> a, TreeMap::new
            ));


    //Output metrics
    final StringBuilder builder = new StringBuilder(255);
    builder.append("\n---------------------------------------\n");
    metricsDisplayMap.entrySet().forEach(entry -> {
        MetricPair metricPair = entry.getValue();
        String name = entry.getKey();
        builder.append(String.format(Locale.US, "%50s%25s\t\t%,-10.2f\t\t%s\n",
                name,
                metricPair.metricName.name(),
                metricPair.metric.value(),
                metricPair.metricName.description()
        ));
    });
    builder.append("\n---------------------------------------\n");
    logger.info(builder.toString());
  }
...
}
```

ACTION - Edit MetricsProducerReporter.java and follow the instructions.

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

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

~/kafka-training/lab5.4/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer adds MetricsProducerReporter to executorService

```
public class StockPriceKafkaProducer {
...

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

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

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

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

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

ACTION - Edit StockPriceKafkaProducer.java and follow the instructions.

Run it. Run Servers. Run Producer.

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

ACTION - Run ZooKeeper (if needed) and three brokers (like before).

ACTION - Run StockPriceKafkaProducer from the IDE.

Expected output from MetricsProducerReporter

```
--------------------------------------
    producer-metrics.batch-size-avg               batch-size-avg
9,030.78          The average number of bytes sent per partition per-request.
    producer-metrics.incoming-byte-rate           incoming-byte-rate
63.28             Bytes/second read off all sockets
    producer-metrics.network-io-rate              network-io-rate               1.32
The average number of network operations (reads or writes) on .
    producer-metrics.record-queue-time-avg        record-queue-time-avg
3,008.97          The average time in ms record batches spent in the..
    producer-metrics.record-send-rate             record-send-rate
485.76            The average number of records sent per second.
    producer-metrics.records-per-request-avg      records-per-request-avg
737.03            The average number of records per request.
    producer-metrics.request-size-max             request-size-max
46,064.00         The maximum size of any request sent in the window.
    producer-metrics.response-rate                response-rate                 0.66
Responses received sent per second.
    producer-node-metrics.incoming-byte-rate      incoming-byte-rate
127.41
    producer-node-metrics.request-size-max        request-size-max
46,673.00         The maximum size of any request sent in the window.
    producer-node-metrics.response-rate           response-rate                 1.32
The average number of responses received per second.
    producer-topic-metrics.record-send-rate       record-send-rate
```

```
960.73

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

Use replication verification to observe replicas getting behind

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

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

```
## In ~/kafka-training/lab5.4

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

cd ~/kafka-training

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

Run replica-verification to show max lag

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

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

ACTION - EDIT bin/replica-verification.sh and follow the instructions

ACTION - RUN bin/replica-verification.sh as described above

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

```
# In ~/kafka-training/lab5.4
$ cat bin/describe-topics.sh
#!/usr/bin/env bash

cd ~/kafka-training

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


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

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

ACTION - KILL one broker

ACTION - RUN bin/replica-verification.sh and bin/describe-topics.sh (notice leadership change and max lag increasing)

Change min.insync.replicas

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

ACTION - PERFORM the min.insync.replicas as decribed above

Expected output from changing min.insync.replicas

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

Once you are done, change it back

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

ACTION - SET min.insync.replicas back to 3 in all broker config.

Change min.insync.replicas at the topic level

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

bin/create-topic.sh after modification

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

cd ~/kafka-training

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

Run delete topic and then run create topic as follows.

running delete topic and then create topic.

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

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

ACTION - PERFORM the min.insync.replicas at topic level as decribed above

HINT - Edit create-topics.sh. Use delete-topics to delete and then recreate

Run it with the new topics.

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

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

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