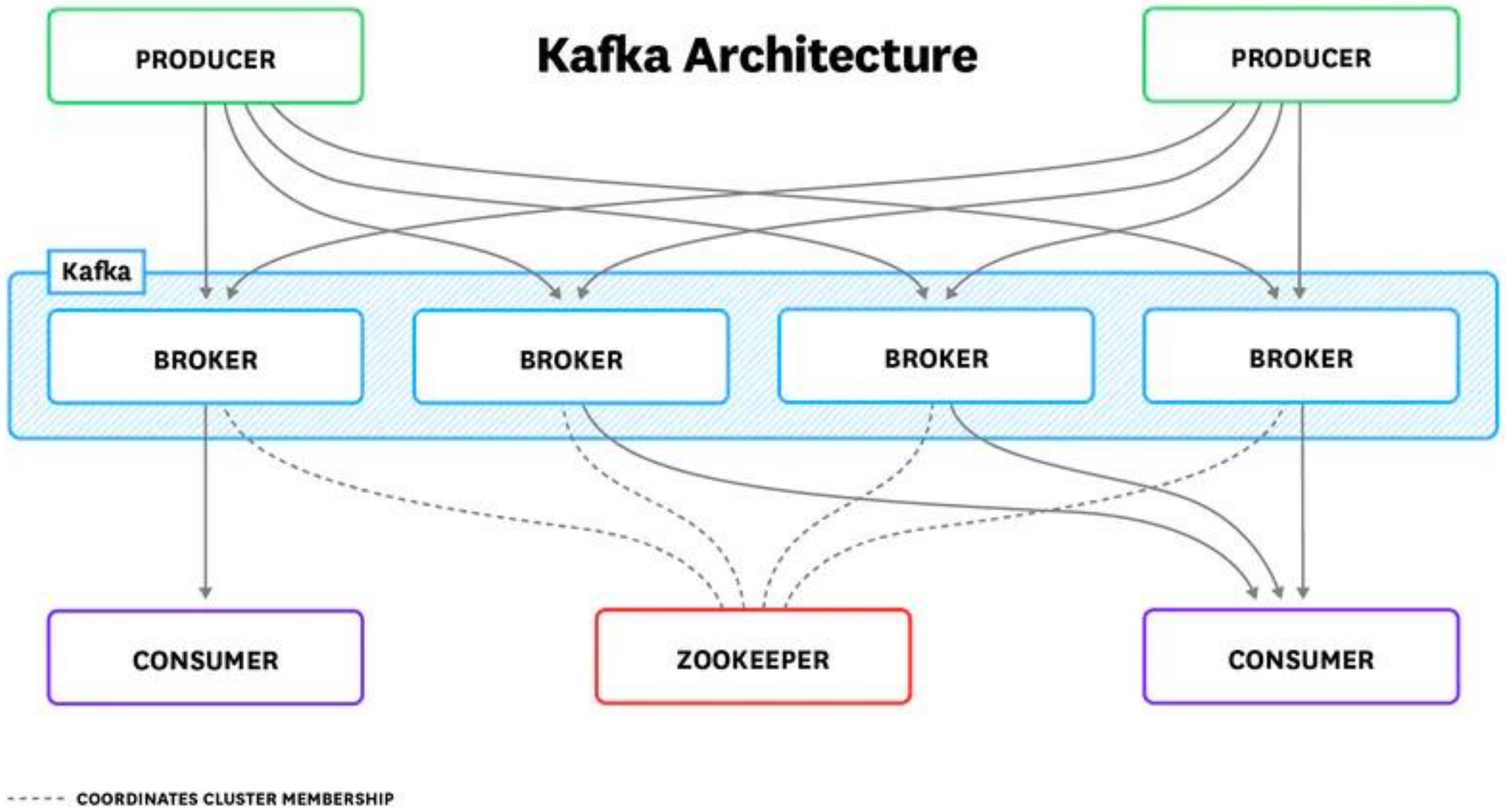
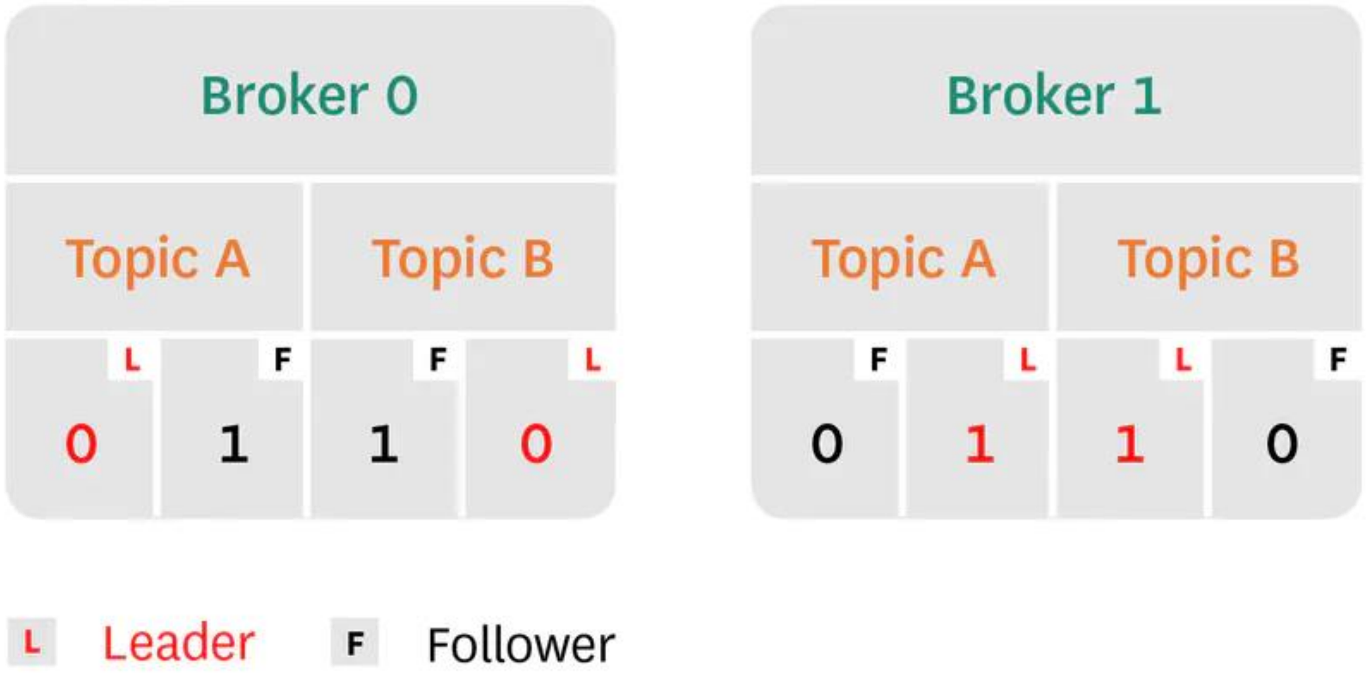
Developer Level Monitoring in Kafka

TriveraTech
TECHNOLOGY TRAINING


PRODUCER
Kafka Architecture
PRODUCER
Kafka
BROKER
BROKER
BROKER
BROKER
CONSUMER
ZOOKEEPER
CONSUMER
- - - - COORDINATES CLUSTER MEMBERSHIP


Broker 0
Topic A
Topic B
L 0
F 1
F 1
L 0
Broker 1
Topic A
Topic B
F 0
L 1
L 1
F 0
L Leader
F Follower

Kafka Developer Monitoring

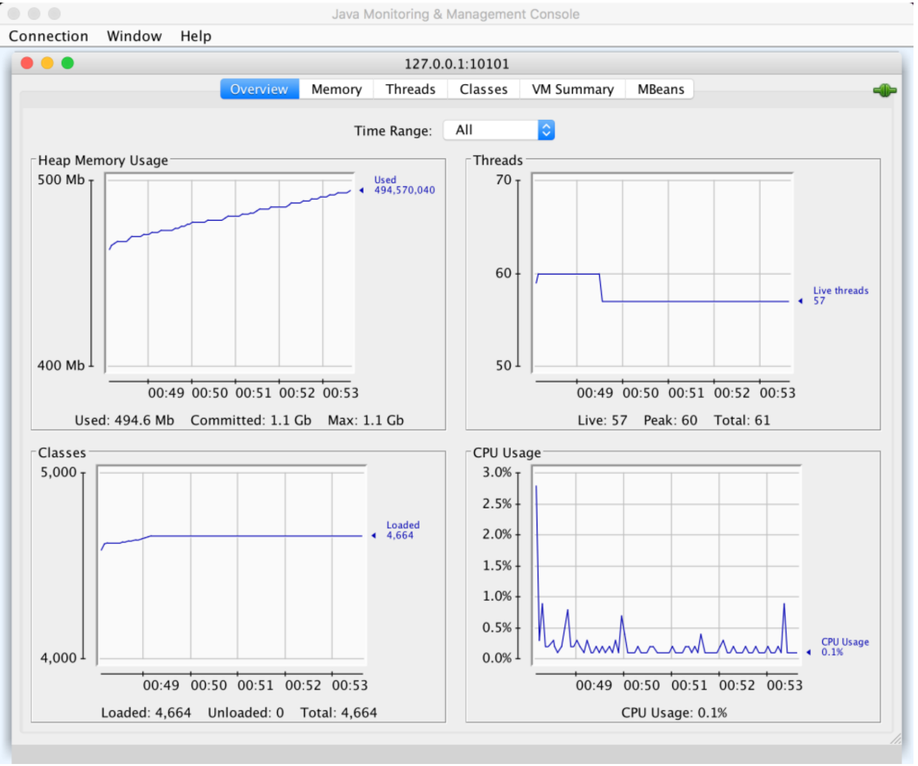
- Kafka uses Yammer metrics
- Gauges
- Counters
- Meters
- Historgrams
- Timers
- Health Checks

Do this…

- $ JMX_PORT=10101 start-1rst-server.sh

- $ jconsole 127.0.0.1:10101

Switch the Mbeans tab

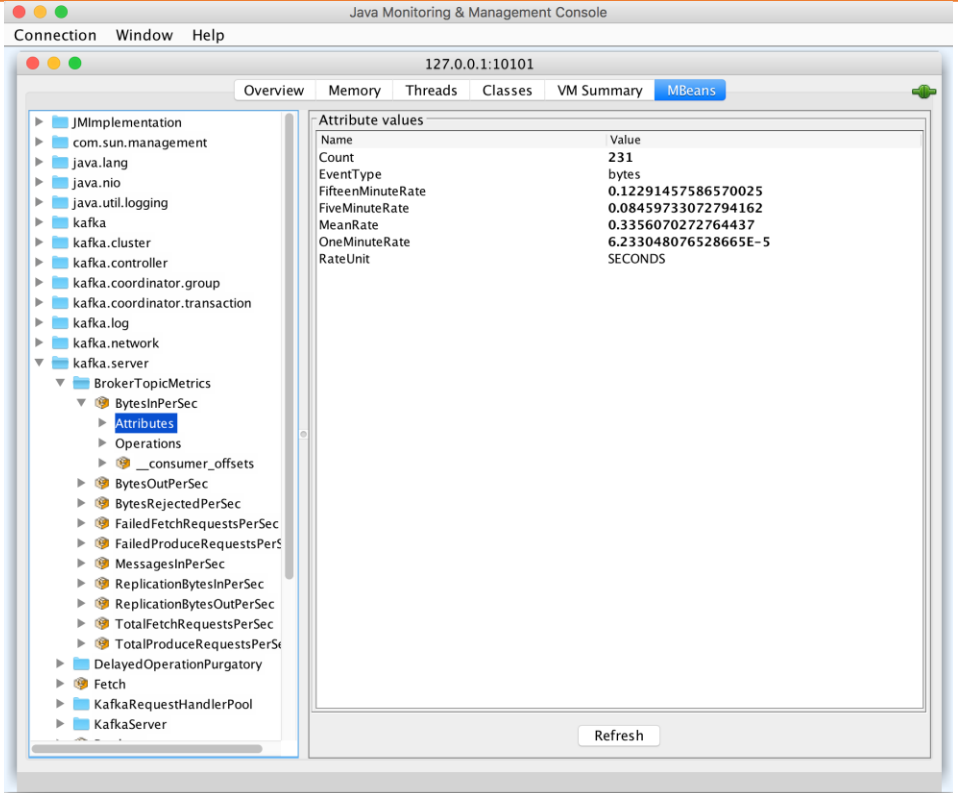


Used: 494.6 Mb Committed: 1.1 Gb Max: 1.1 Gb

Live: 57 Peak: 60 Total: 61

Loaded: 4,664 Unloaded: 0 Total: 4,664

CPU Usage: 0.1%


Java Monitoring & Management Console
Connection Window Help
127.0.0.1:10101
Overview Memory Threads Classes VM Summary MBeans
Refresh

```
JMImplementation
com.sun.management
java.lang
java.nio
java.util.logging
kafka
kafka.cluster
kafka.controller
kafka.coordinator.group
kafka.coordinator.transaction
kafka.log
kafka.network
kafka.server
  BrokerTopicMetrics
    BytesInPerSec
      Attributes
      Operations
      __consumer_offsets
    BytesOutPerSec
    BytesRejectedPerSec
    FailedFetchRequestsPerSec
    FailedProduceRequestsPerS
    MessagesInPerSec
    ReplicationBytesInPerSec
    ReplicationBytesOutPerSec
    TotalFetchRequestsPerSec
    TotalProduceRequestsPerSe
  DelayedOperationPurgatory
  Fetch
  KafkaRequestHandlerPool
  KafkaServer
```

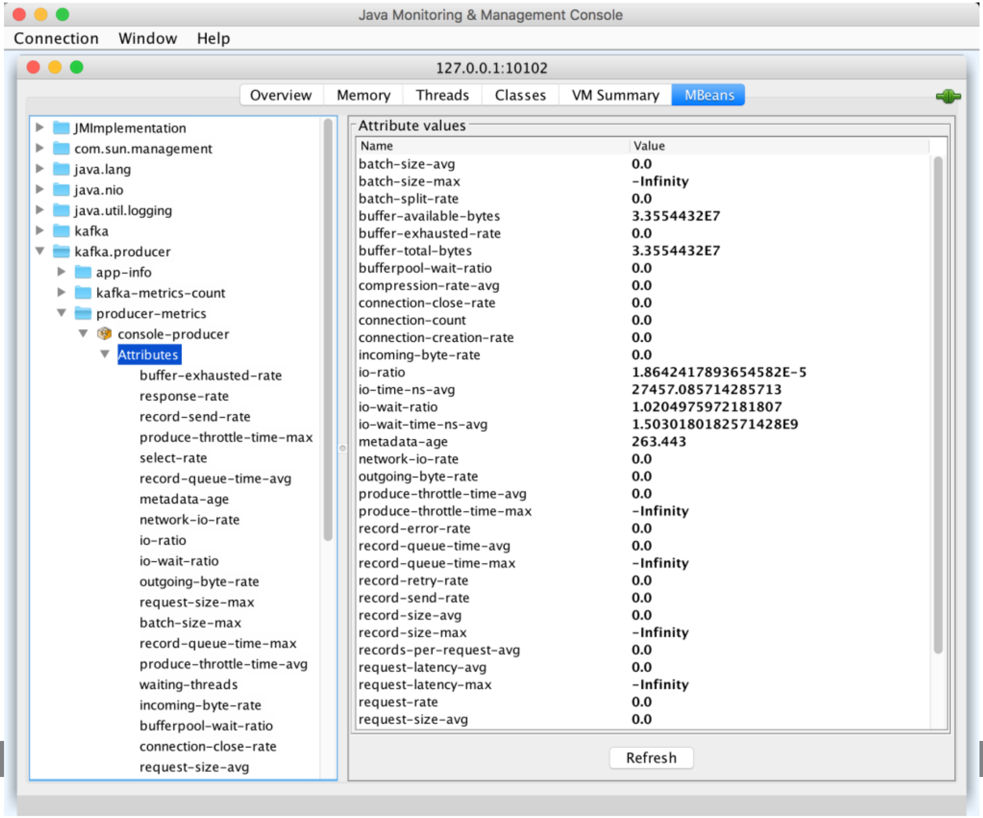
Attribute values

Name	Value
Count	231
EventType	bytes
FifteenMinuteRate	0.12291457586570025
FiveMinuteRate	0.08459733072794162
MeanRate	0.3356070272764437
OneMinuteRate	6.233048076528665E-5
RateUnit	SECONDS

Let’s Monitor Kafka (as a Developer)

```
$ JMX_PORT=10102 bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test_topic

$ jconsole 127.0.0.1:10102
```

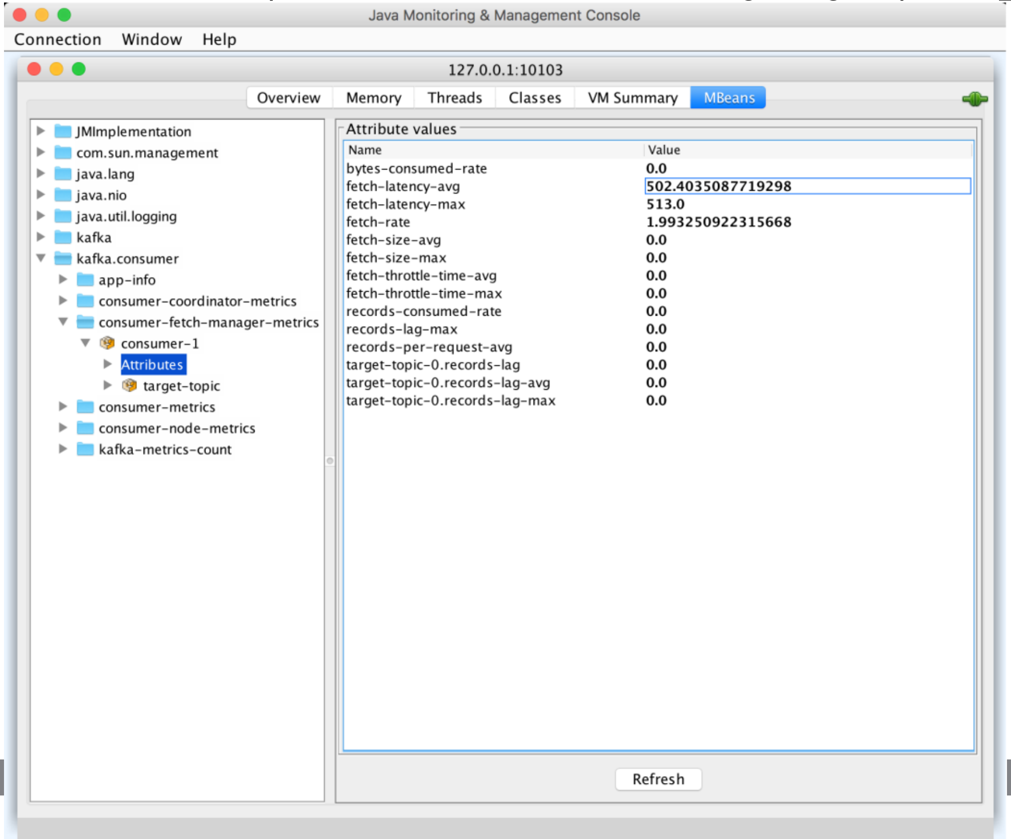


- JMImplementation
- com.sun.management
- java.lang
- java.nio
- java.util.logging
- kafka
- kafka.producer
  - app-info
  - kafka-metrics-count
  - producer-metrics
    - console-producer
      - Attributes
        - buffer-exhausted-rate
        - response-rate
        - record-send-rate
        - produce-throttle-time-max
        - select-rate
        - record-queue-time-avg
        - metadata-age
        - network-io-rate
        - io-ratio
        - io-wait-ratio
        - outgoing-byte-rate
        - request-size-max
        - batch-size-max
        - record-queue-time-max
        - produce-throttle-time-avg
        - waiting-threads
        - incoming-byte-rate
        - bufferpool-wait-ratio
        - connection-close-rate
        - request-size-avg

Name	Value
batch-size-avg	0.0
batch-size-max	-Infinity
batch-split-rate	0.0
buffer-available-bytes	3.3554432E7
buffer-exhausted-rate	0.0
buffer-total-bytes	3.3554432E7
bufferpool-wait-ratio	0.0
compression-rate-avg	0.0
connection-close-rate	0.0
connection-count	0.0
connection-creation-rate	0.0
incoming-byte-rate	0.0
io-ratio	1.8642417893654582E-5
io-time-ns-avg	27457.085714285713
io-wait-ratio	1.0204975972181807
io-wait-time-ns-avg	1.5030180182571428E9
metadata-age	263.443
network-io-rate	0.0
outgoing-byte-rate	0.0
produce-throttle-time-avg	0.0
produce-throttle-time-max	-Infinity
record-error-rate	0.0
record-queue-time-avg	0.0
record-queue-time-max	-Infinity
record-retry-rate	0.0
record-send-rate	0.0
record-size-avg	0.0
record-size-max	-Infinity
records-per-request-avg	0.0
request-latency-avg	0.0
request-latency-max	-Infinity
request-rate	0.0
request-size-avg	0.0

Monitor Consumer Stats

```
$ JMX_PORT=10103 kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic test_topic
$ jconsole 127.0.0.1:10103
```

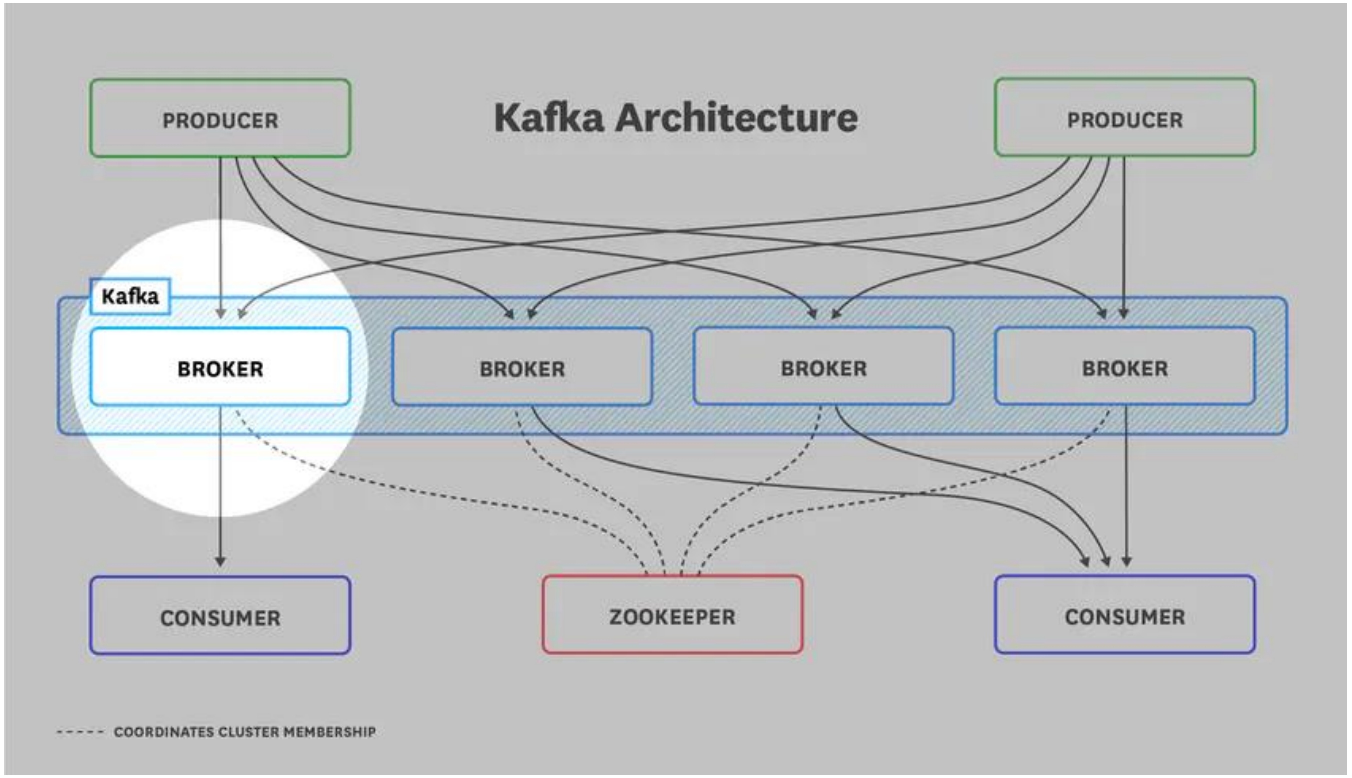


Name	Value
bytes-consumed-rate	0.0
fetch-latency-avg	502.4035087719298
fetch-latency-max	513.0
fetch-rate	1.993250922315668
fetch-size-avg	0.0
fetch-size-max	0.0
fetch-throttle-time-avg	0.0
fetch-throttle-time-max	0.0
records-consumed-rate	0.0
records-lag-max	0.0
records-per-request-avg	0.0
target-topic-0.records-lag	0.0
target-topic-0.records-lag-avg	0.0
target-topic-0.records-lag-max	0.0

Key Metrics to Monitor

- Broker Metrics
- Producer Metrics
- Consumer Metrics

Broker Metrics



Kafka Metrics

Name	MBean name	Description	Metric type
UnderReplicatedPartitions	kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions	Number of unreplicated partitions	Resource: Availability
IsrShrinksPerSec/IsrExpandsPerSec	kafka.server:type=ReplicaManager,name=IsrShrinksPerSec kafka.server:type=ReplicaManager,name=IsrExpandsPerSec	Rate at which the pool of in-sync replicas (ISRs) shrinks/expands	Resource: Availability
ActiveControllerCount	kafka.controller:type=KafkaController,name=ActiveControllerCount	Number of active controllers in cluster	Resource: Error
OfflinePartitionsCount	kafka.controller:type=KafkaController,name=OfflinePartitionsCount	Number of offline partitions	Resource: Availability
LeaderElectionRateAndTimeMs	kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs	Leader election rate and latency	Other
UncleanLeaderElectionsPerSec	kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec	Number of “unclean” elections per second	Resource: Error
TotalTimeMs	kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce|FetchConsumer|FetchFollower}	Total time (in ms) to serve the specified request (Produce/Fetch)	Work: Performance
PurgatorySize	kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation={Produce|Fetch}	Number of requests waiting in producer purgatory/Number of requests waiting in fetch purgatory	Other
BytesInPerSec/BytesOutPerSec	kafka.server:type=BrokerTopicMetrics,name={BytesInPerSec|BytesOutPerSec}	Aggregate incoming/outgoing byte rate	Work: Throughput
RequestsPerSecond	kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce|FetchConsumer|FetchFollower},version={0|1|2|3|…}		

UnderReplicated Partitions

- Should not exceed ZERO for extended periods of time

IsrShrinksPerSec

- No FLAPPING

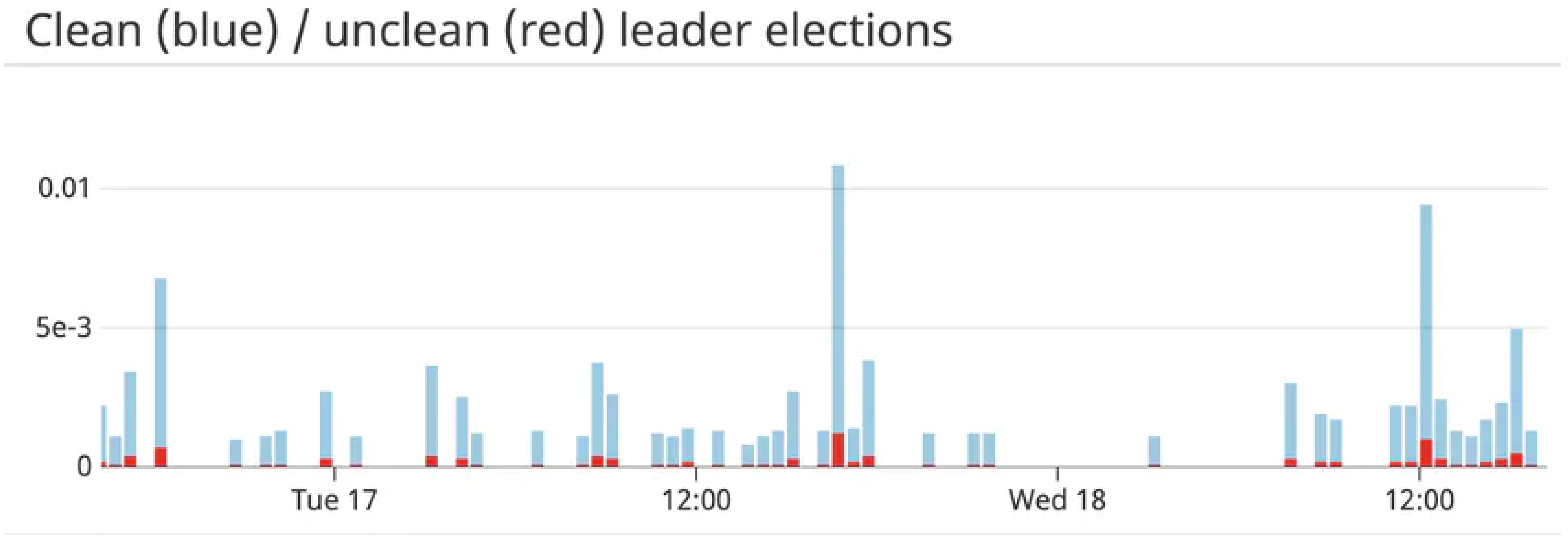
Active Controller Count

- Should always equal 1

Offline Partitions Count

- Partitions without an active leader will be shut down

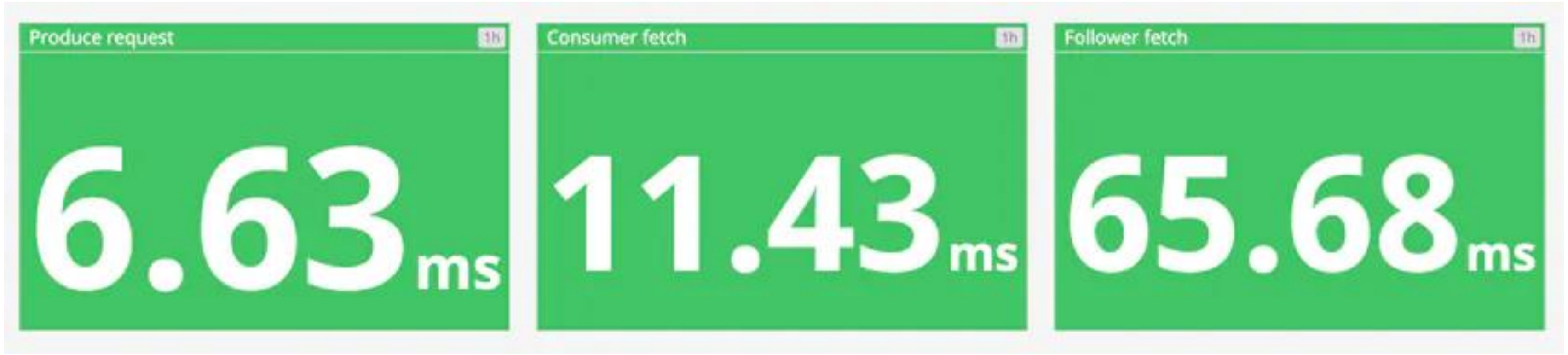
LeaderElectionRateAndTimeMs



UncleanLeaderElectionsPerSec

- This may indicate data loss

TotalTimeMs



PurgatorySize

- Keeping an eye on the size of purgatory is useful to determine the underlying causes of latency.

BytesInPerSec

- Consider end-end compression enable.

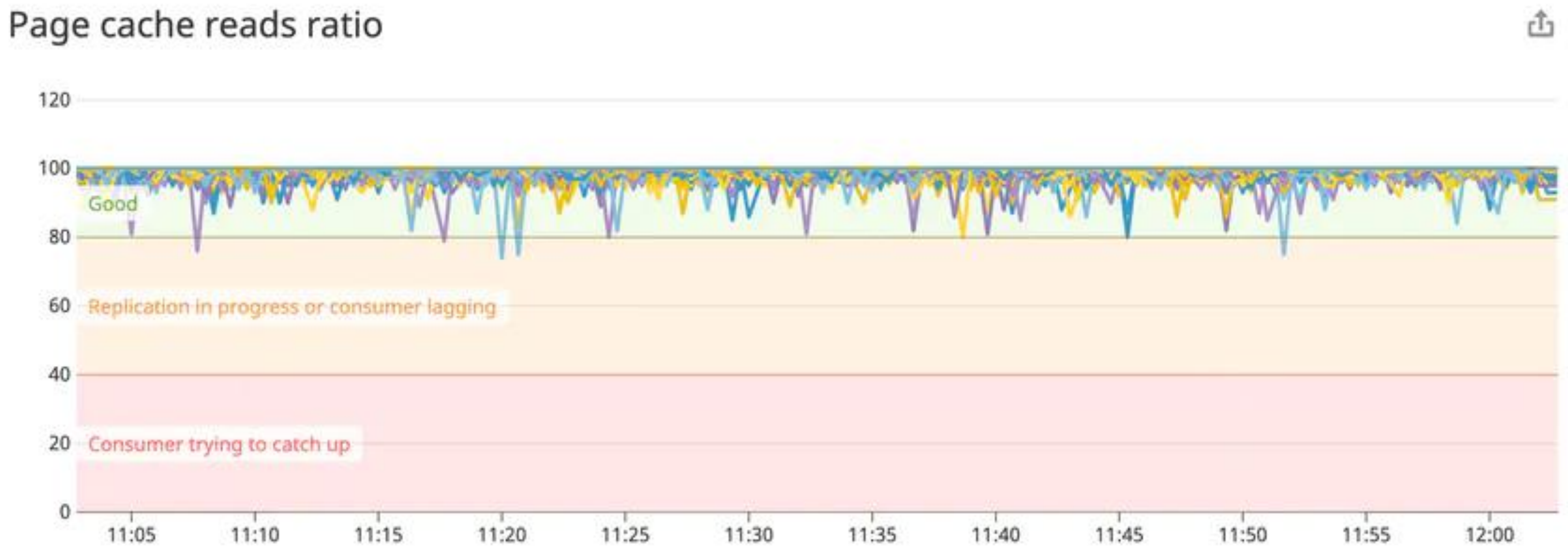
RequestsPerSec

- Use this to measure throughput

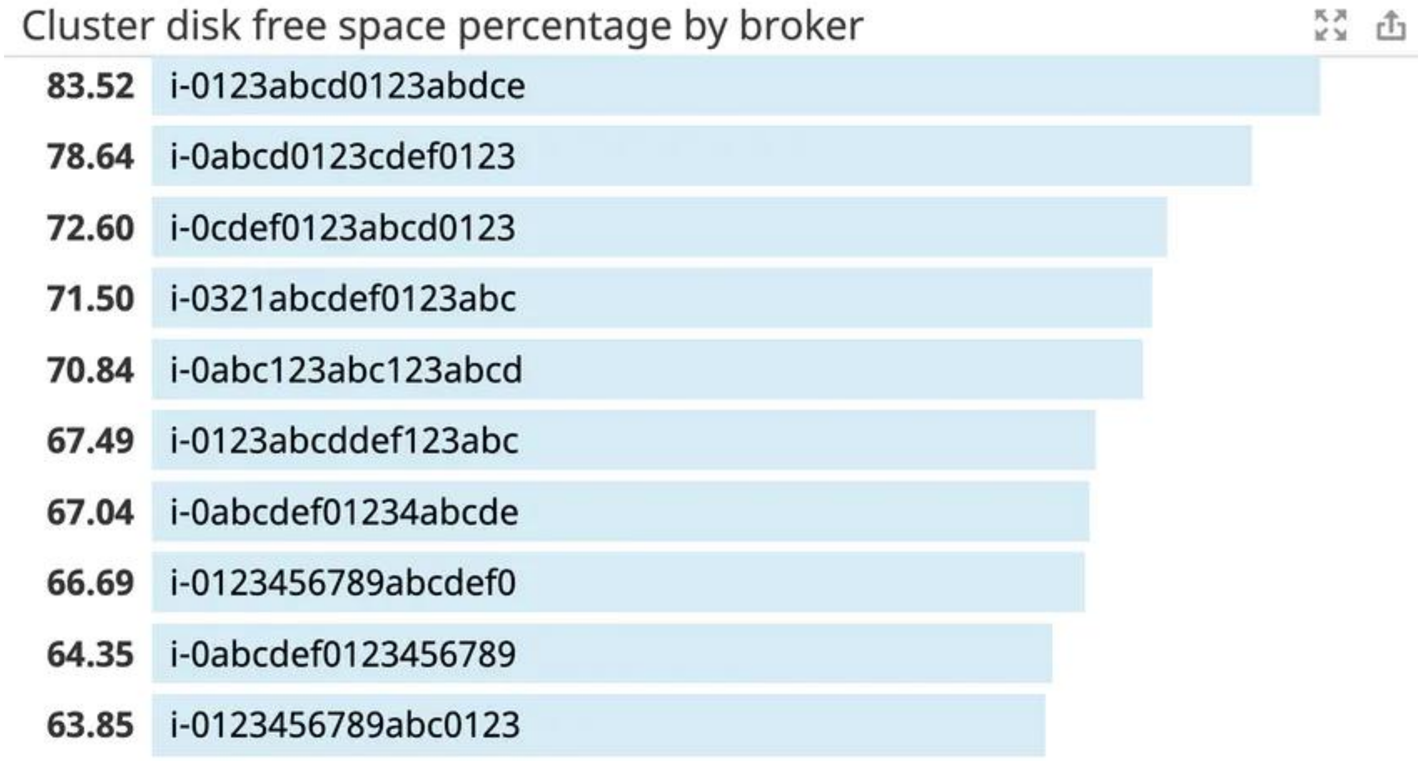
Metrics at the Host Level

Name	Description	Metric type
Page cache reads ratio	Ratio of reads from page cache vs reads from disk	Resource: Saturation
Disk usage	Disk space currently consumed vs. available	Resource: Utilization
CPU usage	CPU use	Resource: Utilization
Network bytes sent/received	Network traffic in/out	Resource: Utilization

Page Cache Read Ratio



Disk, CPU, Network Usage

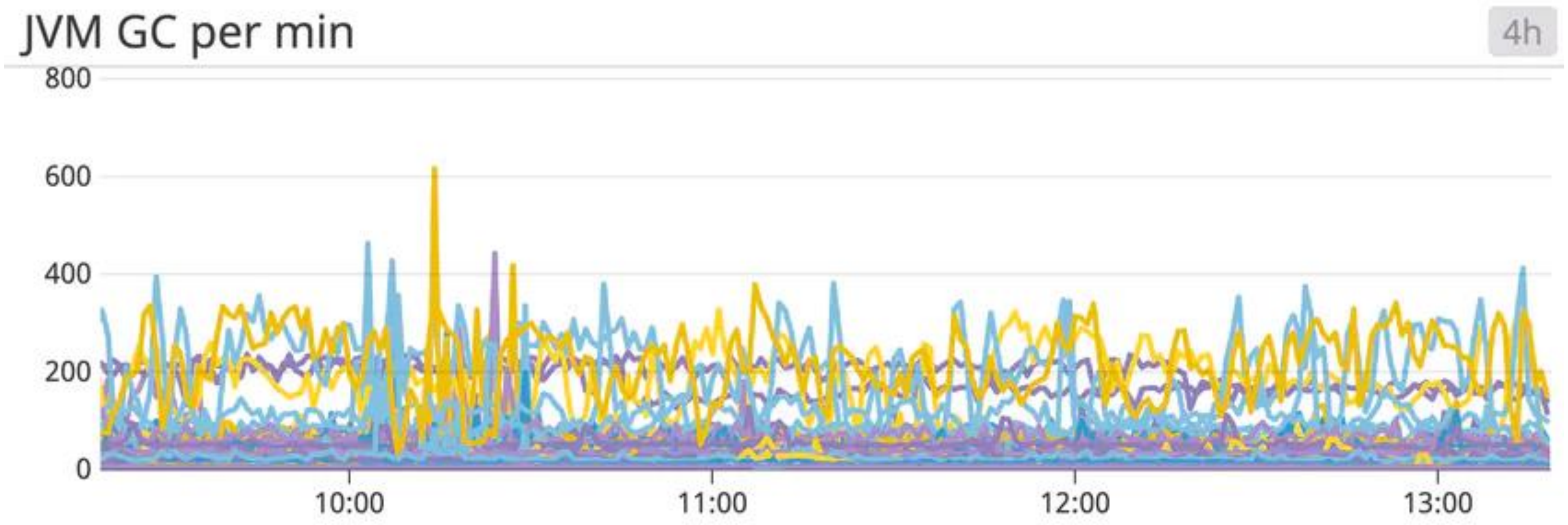


Cluster disk free space percentage by broker

83.52	i-0123abcd0123abdce
78.64	i-0abcd0123cdef0123
72.60	i-0cdef0123abcd0123
71.50	i-0321abcdef0123abc
70.84	i-0abc123abc123abcd
67.49	i-0123abcddef123abc
67.04	i-0abcdef01234abcde
66.69	i-0123456789abcdef0
64.35	i-0abcdef0123456789
63.85	i-0123456789abc0123

TriveraTech TECHNOLOGY TRAINING

JVM Metrics

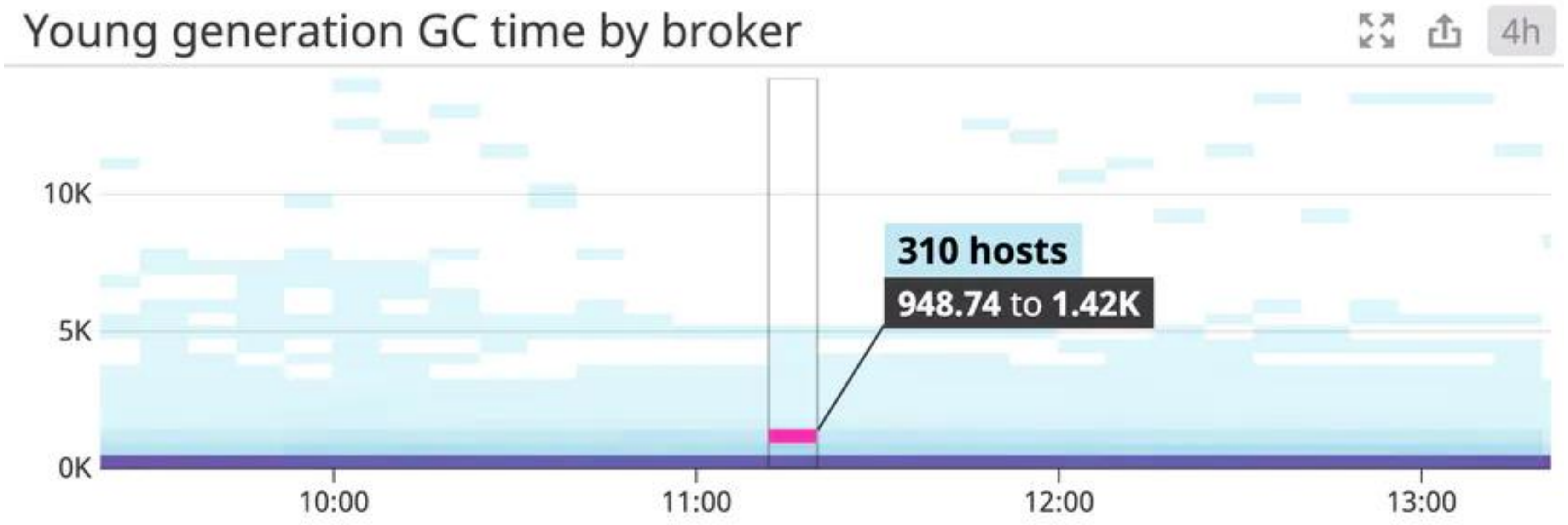


JVM Metrics

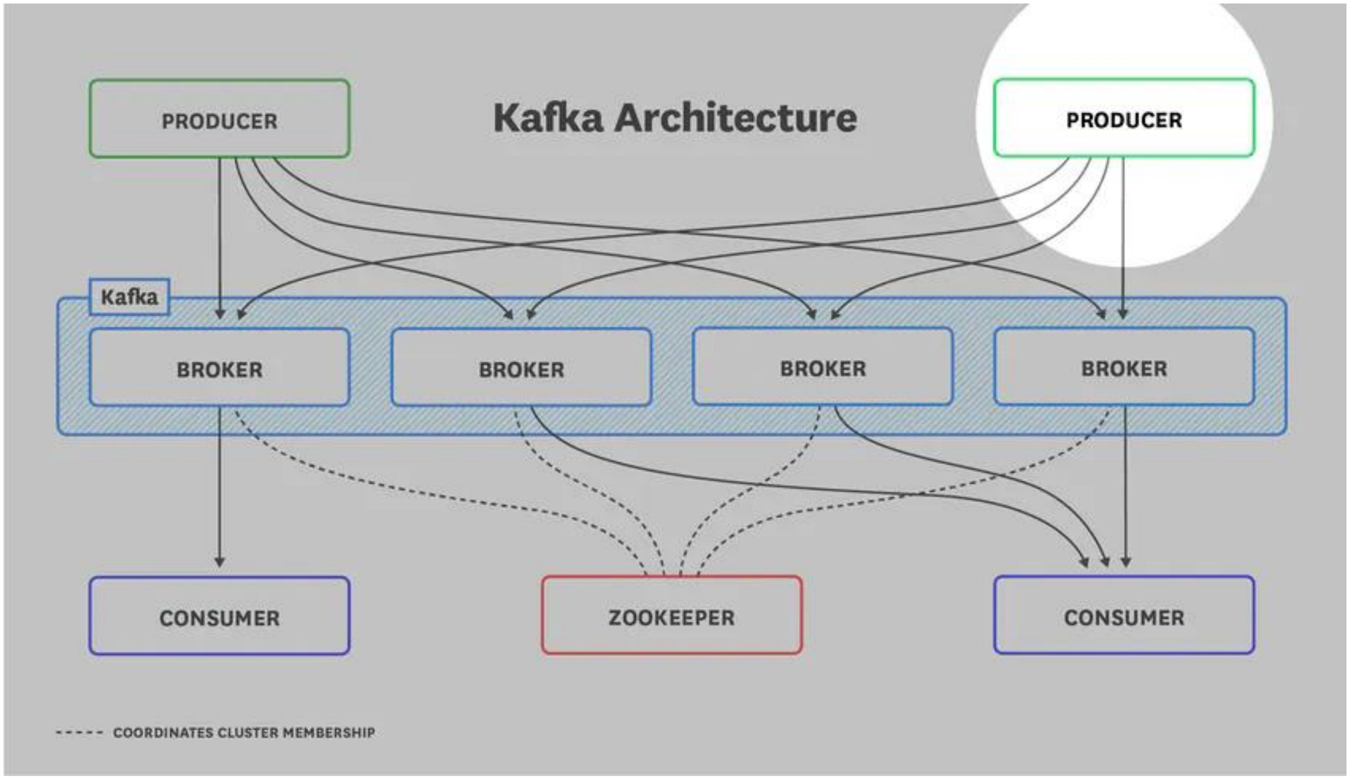
JMX attribute	MBean name	Description	Type
CollectionCount	java.lang:type=GarbageCollector,name=G1 (Young|Old) Generation	The total count of young or old garbage collection processes executed by the JVM	Other
CollectionTime	java.lang:type=GarbageCollector,name=G1 (Young|Old) Generation	The total amount of time (in milliseconds) the JVM has spent executing young or old garbage collection processes	Other

TriveraTech TECHNOLOGY TRAINING

Young/Old Generation Garbage Collection Time



Producer Metrics



Producer Metrics

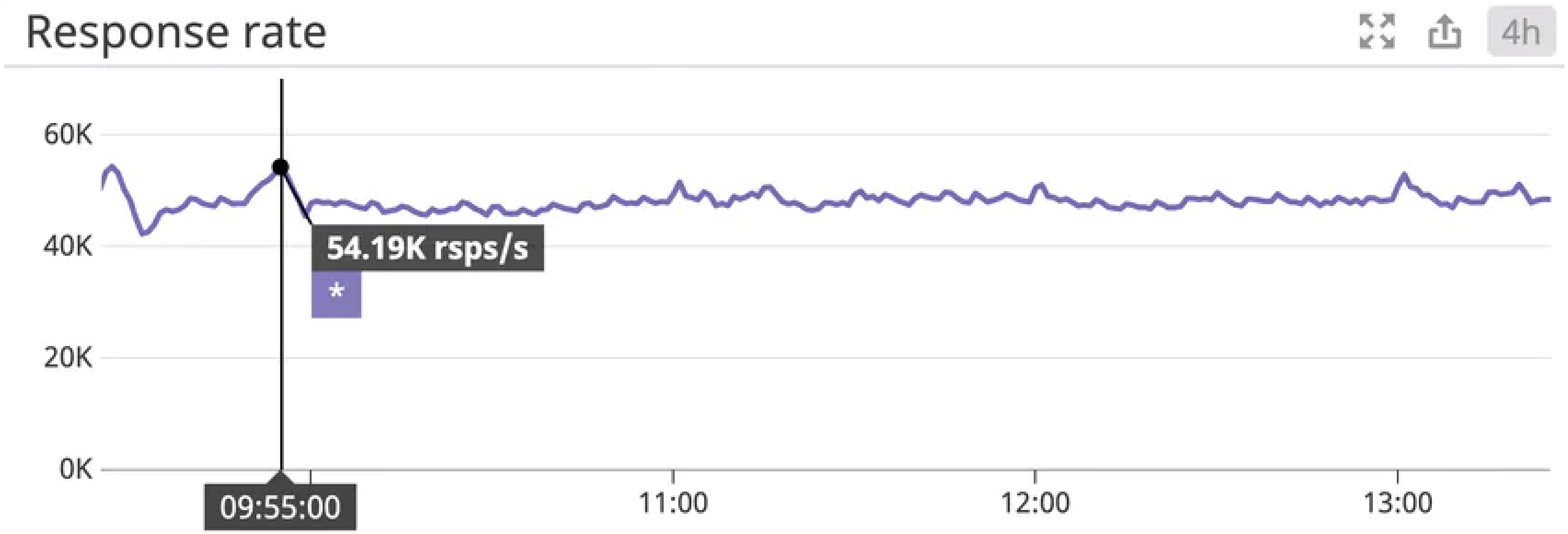
JMX attribute	MBean name	Description	Metric type
compression-rate-avg	kafka.producer:type=producer-metrics,client-id=([-.w]+)	Average compression rate of batches sent	Work: Other
response-rate	kafka.producer:type=producer-metrics,client-id=([-.w]+)	Average number of responses received per second	Work: Throughput
request-rate	kafka.producer:type=producer-metrics,client-id=([-.w]+)	Average number of requests sent per second	Work: Throughput
request-latency-avg	kafka.producer:type=producer-metrics,client-id=([-.w]+)	Average request latency (in ms)	Work: Throughput
outgoing-byte-rate	kafka.producer:type=producer-metrics,client-id=([-.w]+)	Average number of outgoing/incoming bytes per second	Work: Throughput
io-wait-time-ns-avg	kafka.producer:type=producer-metrics,client-id=([-.w]+)	Average length of time the I/O thread spent waiting for a socket (in ns)	Work: Throughput
batch-size-avg	kafka.producer:type=producer-metrics,client-id=([-.w]+)	The average number of bytes sent per partition per request	Work: Throughput

TriveraTech TECHNOLOGY TRAINING

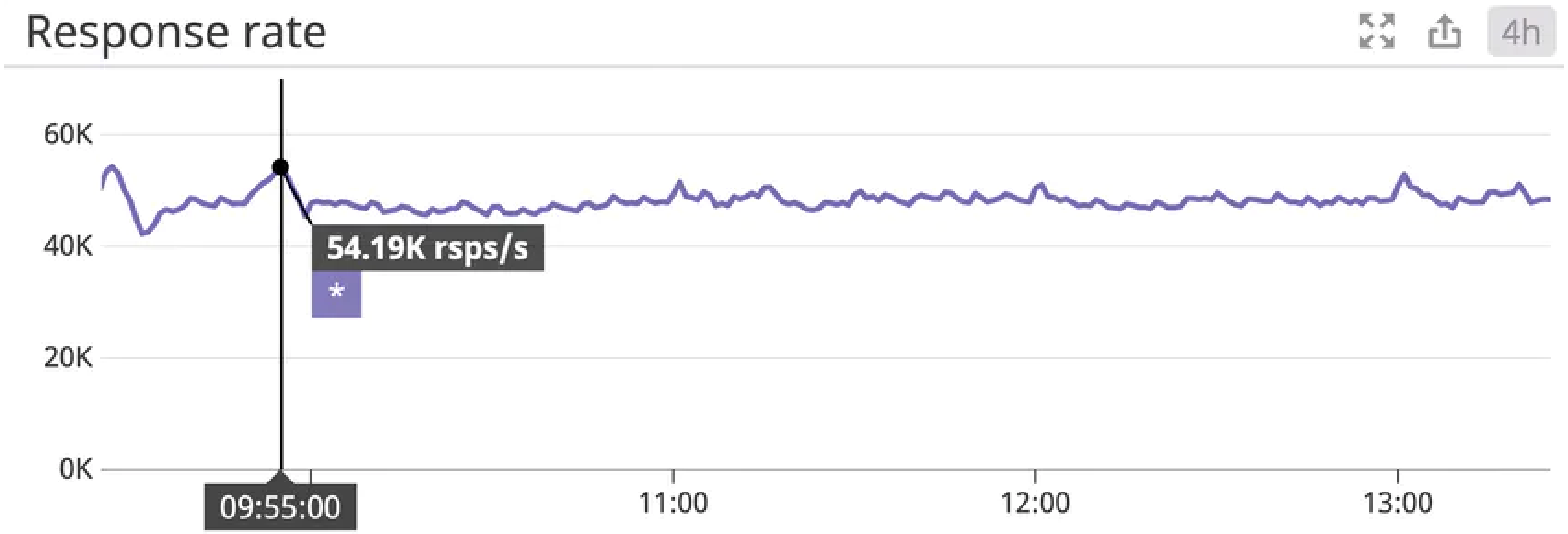
Compression Rate

The name says it all…

Response Rate



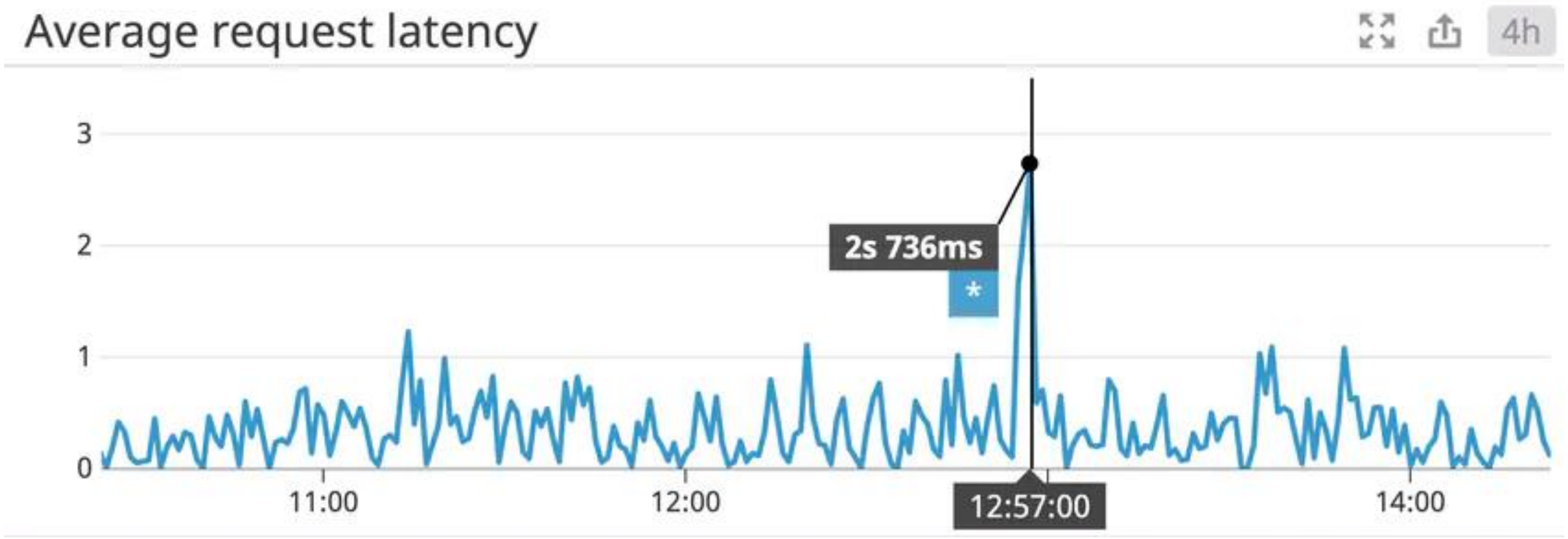
Response Rate



Request Rate

- Consider rate limiting if there is an issue here…

Request Latency Average



Outgoing byte rate

- This is directly related to network throughput

I/O wait time

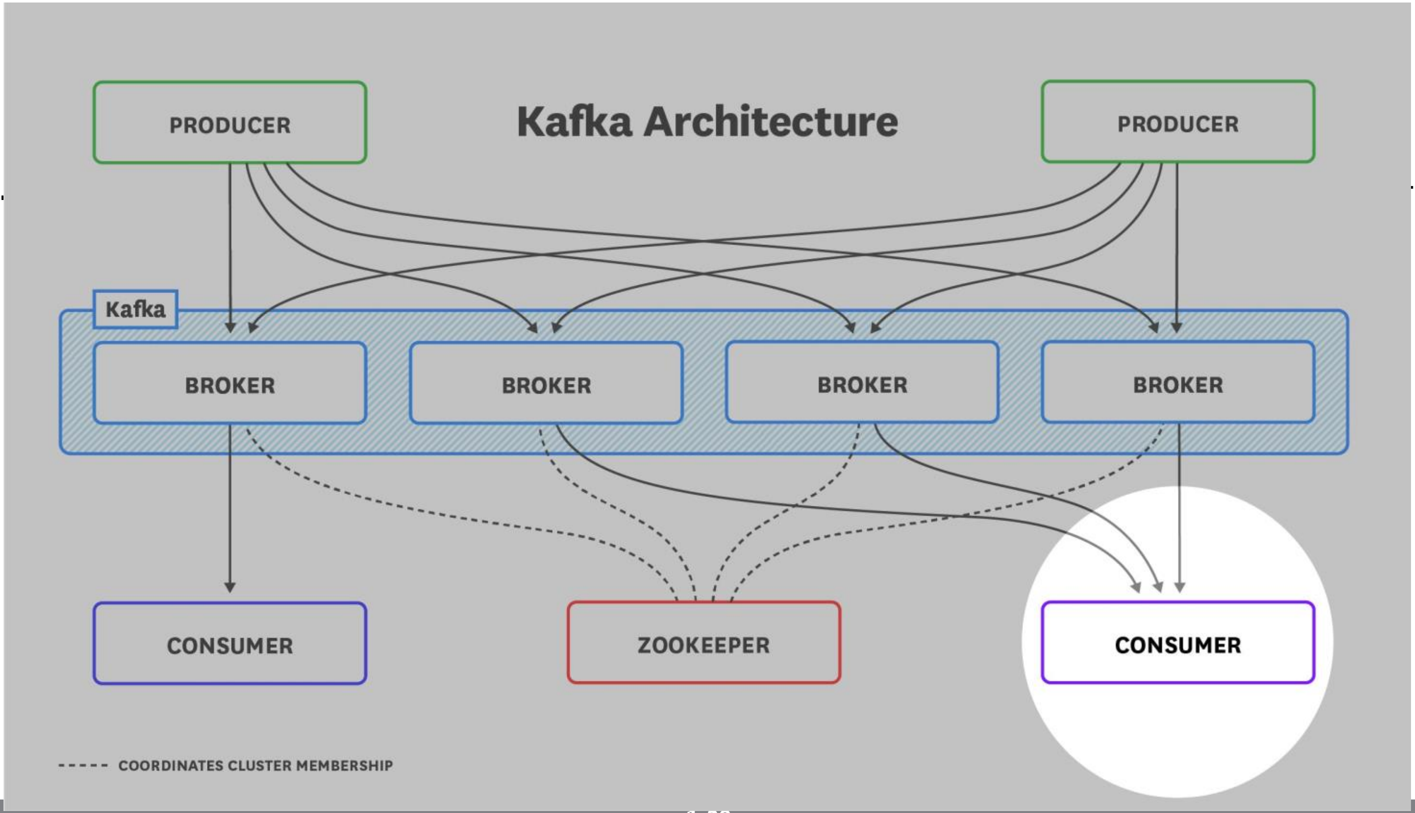
- High I/O – your producers can’t get what they need fast enough…

Batch Size

- KIP-91
- https://cwiki.apache.org/confluence/display/KAFKA/KIP-91+Provide+Intuitive+User+Timeouts+in+The+Producer

Consumer Metrics

- KIP-91
- https://cwiki. -The+Producer

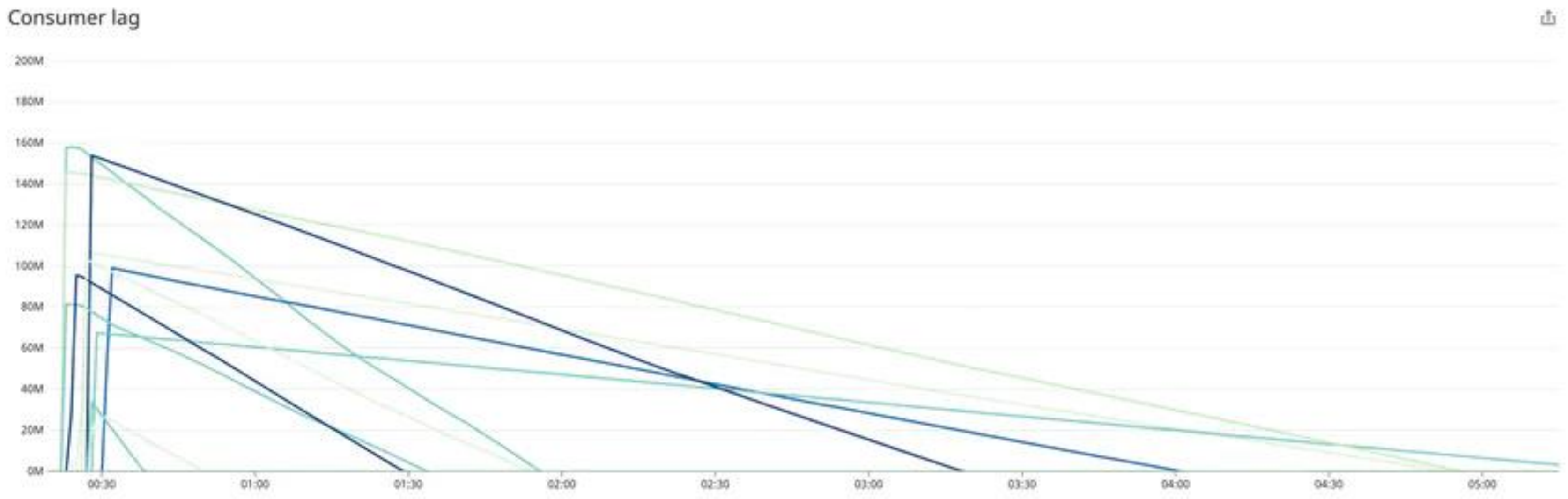


Consumer Metrics

JMX attribute	MBean name	Description	Metric type
records-lag	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+),topic=([-.w]+),partition=([-.w]+)	Number of messages consumer is behind producer on this partition	Work: Performance
records-lag-max	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+),topic=([-.w]+),partition=([-.w]+) kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+)	Maximum number of messages consumer is behind producer, either for a specific partition or across all partitions on this client	Work: Performance
bytes-consumed-rate	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+),topic=([-.w]+) kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+)	Average number of bytes consumed per second for a specific topic or across all topics.	Work: Throughput
records-consumed-rate	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+),topic=([-.w]+) kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+)	Average number of records consumed per second for a specific topic or across all topics	Work: Throughput
fetch-rate	kafka.consumer:type=consumer-fetch-manager-metrics,client_id=([-.w]+)	Number of fetch requests per second from the consumer	Work: Throughput

TriveraTech
TECHNOLOGY TRAINING

Record Lag


Consumer lag
200M
180M
160M
140M
120M
100M
80M
60M
40M
20M
0M
00:30
01:00
01:30
02:00
02:30
03:00
03:30
04:00
04:30
05:00

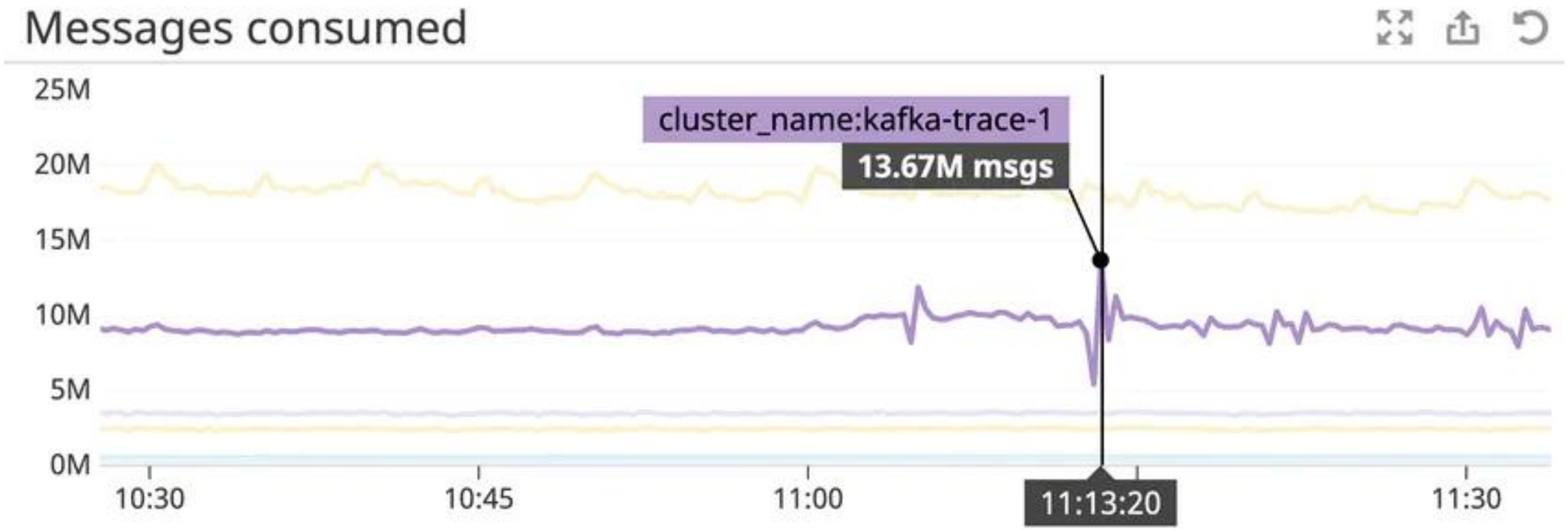
Bytes Consumed Rate

Network

Fetch Rate

Consumers are failing

Records Consumed Rate



Kafka Multi-Threading

TriveraTech TECHNOLOGY TRAINING

Multithreading

- Multithreading is “the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system.”

- In situations where the work can be divided into smaller units, which can be run in parallel, without negative effects on data consistency, multithreading can be used to improve application performance.

Thread per consumer model

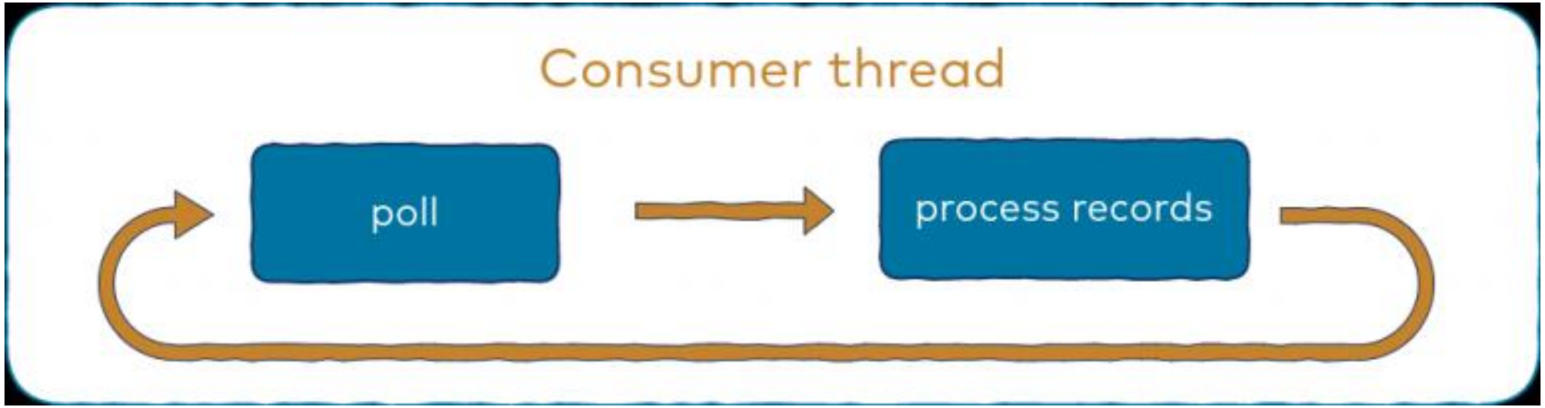
A typical single-threaded implementation is centered around a poll loop. Basically, it’s an infinite loop that repeats two actions:

1. Retrieving records using the poll() method

1. Processing fetched records

Thread per consumer model

```
while (true) {
        ConsumerRecords records = consumer.poll(Duration.ofMillis(10000));
        // Handle fetched records
}
```



Group rebalancing

Consumer group rebalancing is triggered when partitions need to be reassigned among consumers in the consumer group:

- A new consumer joins the group; an existing consumer leaves the group; an existing consumer changes subscription; or partitions are added to one of the subscribed topics.

Group rebalancing

It’s your responsibility to commit offsets before the join group request is sent. You can do this in two ways:

- Always call commitSync() after fetched records are processed and before the next poll method call
- Implement ConsumerRebalanceListener to get notified when partitions are about to be revoked, and commit corresponding offsets at that point

Motivation for a multi-threaded consumer architecture

- If you are familiar with basic Kafka concepts, you know that you can parallelize message consumption by simply adding more consumers in the same group.

- However, that approach is more suitable for horizontal scaling where you add new consumers by adding new application nodes (containers, VMs, and even bare metal instances).

The problem of slow processing

- The maximum delay allowed between poll method calls is defined by the max.poll.interval.ms config, which is five minutes by default.

- If a consumer fails to call the poll method within that interval, it is considered dead, and group rebalancing is triggered.

The problem of slow processing

When using the thread per consumer model, you can deal with this problem by tuning the following config values:

- Set max.poll.records to a smaller value
- Set max.poll.interval.ms to a higher value
- Perform a combination of both

Handling record processing exceptions

Record processing logic, including error handling, is application specific. In the case of processing errors, you can perform one of the following options:

- Stop processing and close the consumer (optionally, retry a few times beforehand)
- Send records to the dead letter queue and continue to the next record (optionally, retry a few times beforehand)
- Retry until the record is processed successfully (this might take forever)

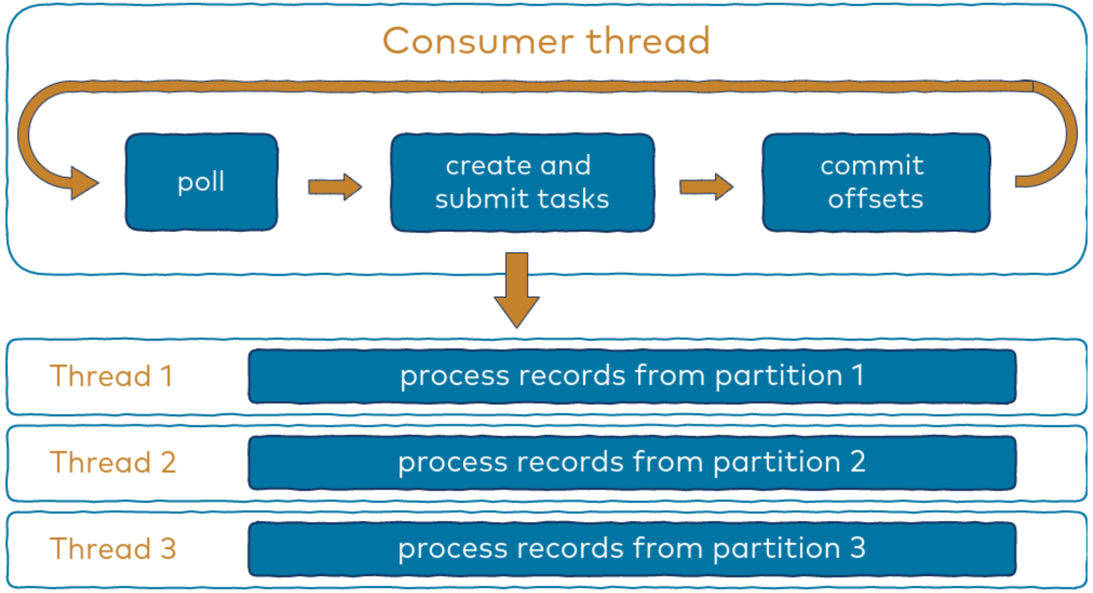
Multi-threaded Kafka consumer

There are many ways to design multi-threaded models for a Kafka consumer. A naive approach might be to process each message in a separate thread taken from a thread pool, while using automatic offset commits (default config). Unfortunately, this may cause some undesirable effects:

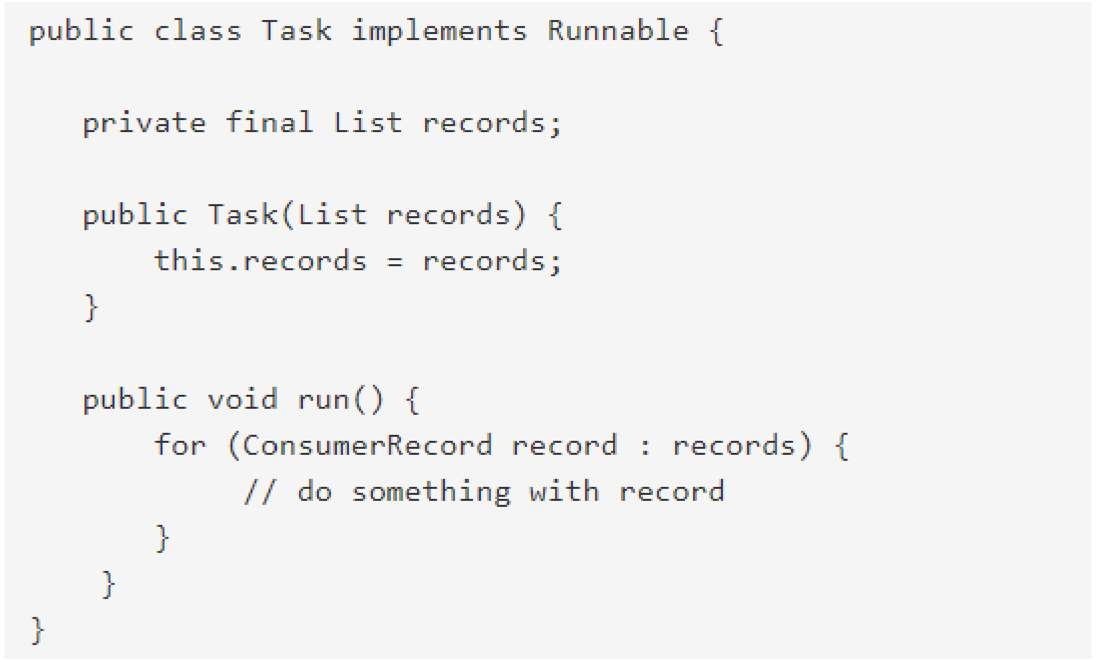
- Offset might be committed before a record is processed
- Message processing order can’t be guaranteed since messages from the same partition could be processed in parallel

Decoupling consumption and processing

- In a multi-threaded implementation, the main consumer thread delegates records processing to other threads.


Consumer thread
poll
create and submit tasks
commit offsets
Thread 1
process records from partition 1
Thread 2
process records from partition 2
Thread 3
process records from partition 3

- The task implementation is quite simple, and the following represents a basic version that will be improved upon and discussed later in this blog post:



```
public class Task implements Runnable {

   private final List records;

   public Task(List records) {
       this.records = records;
   }

   public void run() {
       for (ConsumerRecord record : records) {
            // do something with record
       }
    }
}
```

Decoupling consumption and processing

- In order to utilize the CPU efficiently, a thread pool with a fixed number of threads is used.
- That number is based on the number of available CPU cores as well as the nature of processing—is it CPU intensive or does it wait for I/O, etc.?
- For simplicity, assume eight core CPUs are used; therefore, a fixed thread pool with eight threads is configured:

```
private ExecutorService executor = Executors.newFixedThreadPool(8);
```

Ensuring processing order guarantees

- Since the main thread doesn’t wait for record processing threads to finish and continues with fetching, the probability of ending up with more than one task processing records from the same partition in parallel is high.
- This would obviously break processing order guarantees.
- To prevent it, pause the specific partition using the KafkaConsumer.pause() method after submitting the task holding records from it.

Committing offsets

- Using automatic offset commit, which happens during poll method calls, is no longer an option.
- Offsets have to be committed manually at appropriate times.
- The first step is to disable auto commit in the consumer configuration:

```
config.put(ConsumerConfig.ENABLE_AUTO_COMMIT_
CONFIG,  false);
```

TriveraTech TECHNOLOGY TRAINING

- The following describes how the main consumer thread handles the operations discussed above:

```
 public void run() {
        try {
            consumer.subscribe(Collections.singleton("topic-name"), this);
            while (!stopped.get()) {
                ConsumerRecords<String, String> records =
consumer.poll(Duration.of(100, ChronoUnit.MILLIS));
                handleFetchedRecords(records);
                checkActiveTasks();
                commitOffsets();
            }
        } catch (WakeupException we) {
            if (!stopped.get())
                throw we;
        } finally {
            consumer.close();
        }
    }
```

- It also pauses appropriate partitions and stores task references to a map instance named activeTasks, so their status can be checked on later.

```
private void handleFetchedRecords(ConsumerRecords<String, String>
records) {
    if (records.count() > 0) {
        records.partitions().forEach(partition -> {
            List<ConsumerRecord<String, String>> partitionRecords =
records.records(partition);
            Task task = new Task(partitionRecords);
            executor.submit(task);
            activeTasks.put(partition, task);
        });
        consumer.pause(records.partitions());
    }
}
```

- The next step of the poll loop iteration uses the checkActiveTasks() method, where it asks for the current progress of tasks and checks to see if they are finished:

```
private void checkActiveTasks() {
    List finishedTasksPartitions = new ArrayList<>();
    activeTasks.forEach((partition, task) -> {
        if (task.isFinished())
            finishedTasksPartitions.add(partition);
        long offset = task.getCurrentOffset();
        if (offset > 0)
            offsetsToCommit.put(partition, new OffsetAndMetadata(offset));
    });
    finishedTasksPartitions.forEach(partition ->
activeTasks.remove(partition));
    consumer.resume(finishedTasksPartitions);
}
```

Putting it together

```
private void commitOffsets() {
    try {
        long currentTimeMillis = System.currentTimeMillis();
        if (currentTimeMillis - lastCommitTime > 5000) {
            if(!offsetsToCommit.isEmpty()) {
                consumer.commitAsync(offsetsToCommit);
                offsetsToCommit.clear();
            }
            lastCommitTime = currentTimeMillis;
        }
    } catch (Exception e) {
        log.error("Failed to commit offsets!", e);
    }
}
```

Handling group rebalances

- Since the poll method is now called in parallel with processing, the consumer could rebalance and some partitions might be reassigned to another consumer while there are still tasks processing records from those partitions.

- As a result, some records could be processed by both consumers.

Handling group rebalances

If there are tasks currently processing records from partitions that are being revoked, there are two options to handle this situation:

1. Wait for the task to finish

1. Stop the task, waiting only for the record currently being processed to finish

Handling group rebalances

- The implementation of onPartitionsRevoked()method might look like this:

Refer to the file 1_1.txt

Handling group rebalances

- For a complete picture, revisit the Task class below to see how stopping the task works:

Refer to the file 1_2.txt

Handling group rebalances

The waitForCompletion() method waits for task completion by calling the get() method on the CompletableFuture instance. It can be completed in two ways:

- From the run() method after the processing loop has finished
- From the stop() method if it is called before the ExecutorService starts to process this task

TriveraTech TECHNOLOGY TRAINING

Conclusion

Implementing a multi-threaded consumer model offers significant advantages over the thread per consumer model for certain use cases. Although there are many ways to do this, the key considerations are always the same:

- Ensure that records from the same partitions are processed only by one thread at a time
- Commit offsets only after records are processed
- Handle group rebalancing properly

COMPLETE LAB