KIP-628: ConsumerPerformance's multi-thread implementation

This page is meant as a template for writing a KIP. To create a KIP choose Tools->Copy on this page and modify with your content and replace the heading with the next KIP number and a description of your issue. Replace anything in italics with your own description.

Status

Current state: *Under Discussion*

Discussion thread: *here [Change the link from the KIP proposal email archive to your own email thread]*

JIRA: *here*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

The implementation of multiple threads for ConsumerPerformance, which was written for old consumer, has been removed completely in https://github.com/apache/kafka/pull/5230/. Making option [threads] work for the new consumer so that we can test the consumer performance while setting threads to different numbers.

Public Interfaces

kafka.tools.ConsumerPerformance is used in kafka-consumer-perf-test.sh. This KIP is to make option [threads] work.

Proposed Changes

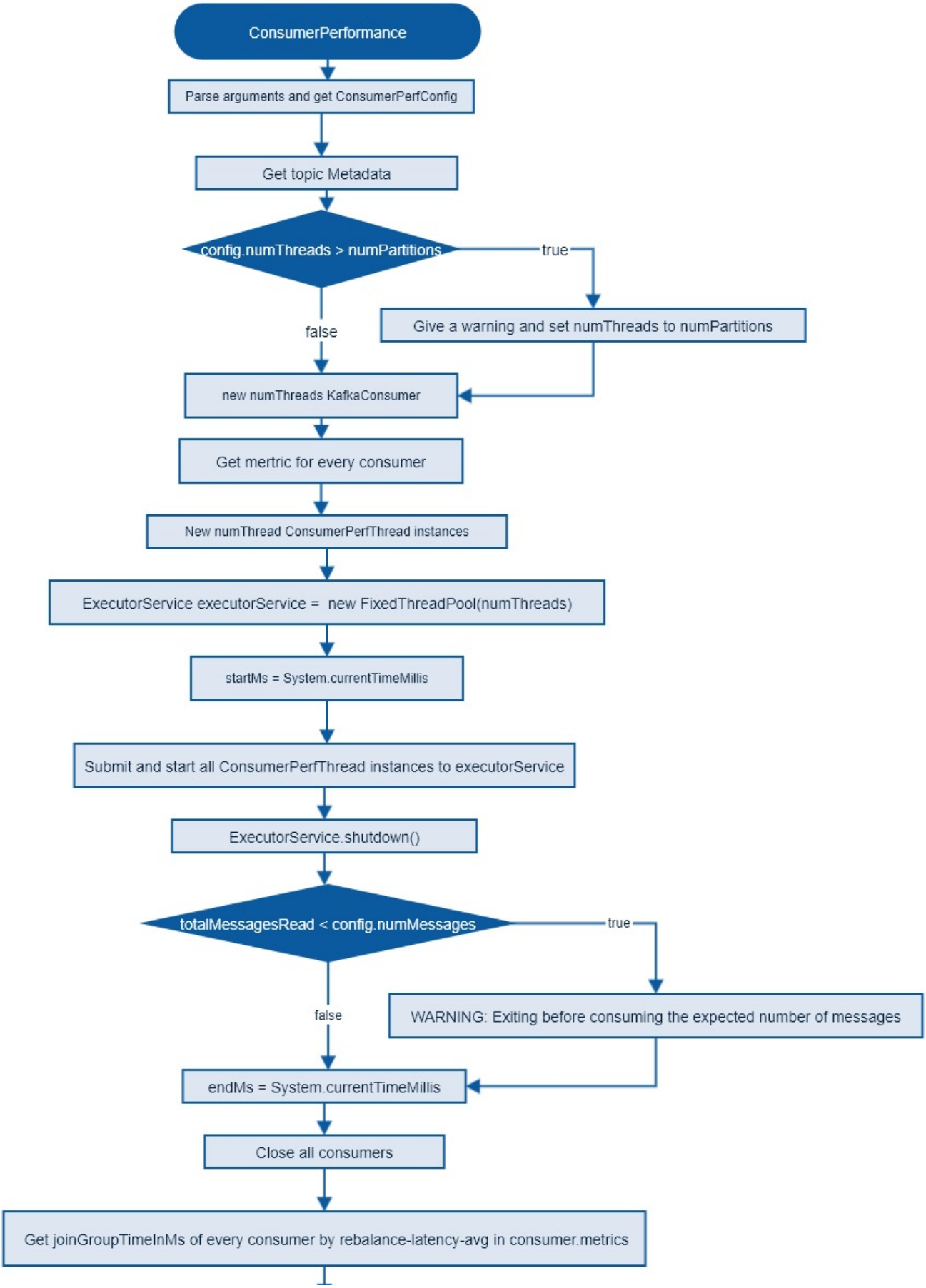
We define a new class ConsumerPerfThread in ConsumerPerformance to implement multi-thread, which is not a public interface.

Attribute	Description
threadId : Int	The thread Id.
config : ConsumerPerfConfig	The configuration of ComsumerPerformance
totalMessagesRead : LongAdder	The total number of messages which have been read
totalBytesRead : LongAdder	The total byte of messages which have been read
consumer : KafkaConsumer[Array[Byte], Array[Byte]]	The consumer to fetch messages
topics: List[String]	Topic to test
metric : mutable.Map[MetricName, _ <: Metric]	The metric for consumer, which used to get rebalanceTime
testStartTime: Long	The time when the test starts

Below are the flowcharts of ConsumerPerformance and *ConsumerPerfThread. The general process is:*

1. *Get ConsumerPerfConfig, like the topic to test, message number to consume, threads number, broker list and etc.*
2. *Get how many partitions the topic has. If threads number is bigger than partitions number, setting threads number to partitions number and giving a warning.*
3. *New Consumers with the same config. The number of Consumers is the same as the threads number.*
4. *New ConsumerPerfThread for every Consumer. They share a variable totalMessageRead. If the totalMesssageRead is bigger than the message number to consume, all threads will end.*
5. *New a thread pool, and submit ConsumerPerfThread to it.*
6. *Wait for the threads to finish and shutdown thread pool.*
7. *Compute and print stats.*

Chart1: ConsumerPerformance


ConsumerPerformance
Parse arguments and get ConsumerPerfConfig
Get topic Metadata
config.numThreads > numPartitions
true
Give a warning and set numThreads to numPartitions
false
new numThreads KafkaConsumer
Get mertric for every consumer
New numThread ConsumerPerfThread instances
ExecutorService executorService = new FixedThreadPool(numThreads)
startMs = System.currentTimeMillis
Submit and start all ConsumerPerfThread instances to executorService
ExecutorService.shutdown()
totalMessagesRead < config.numMessages
true
WARNING: Exiting before consuming the expected number of messages
false
endMs = System.currentTimeMillis
Close all consumers
Get joinGroupTimeInMs of every consumer by rebalance-latency-avg in consumer.metrics

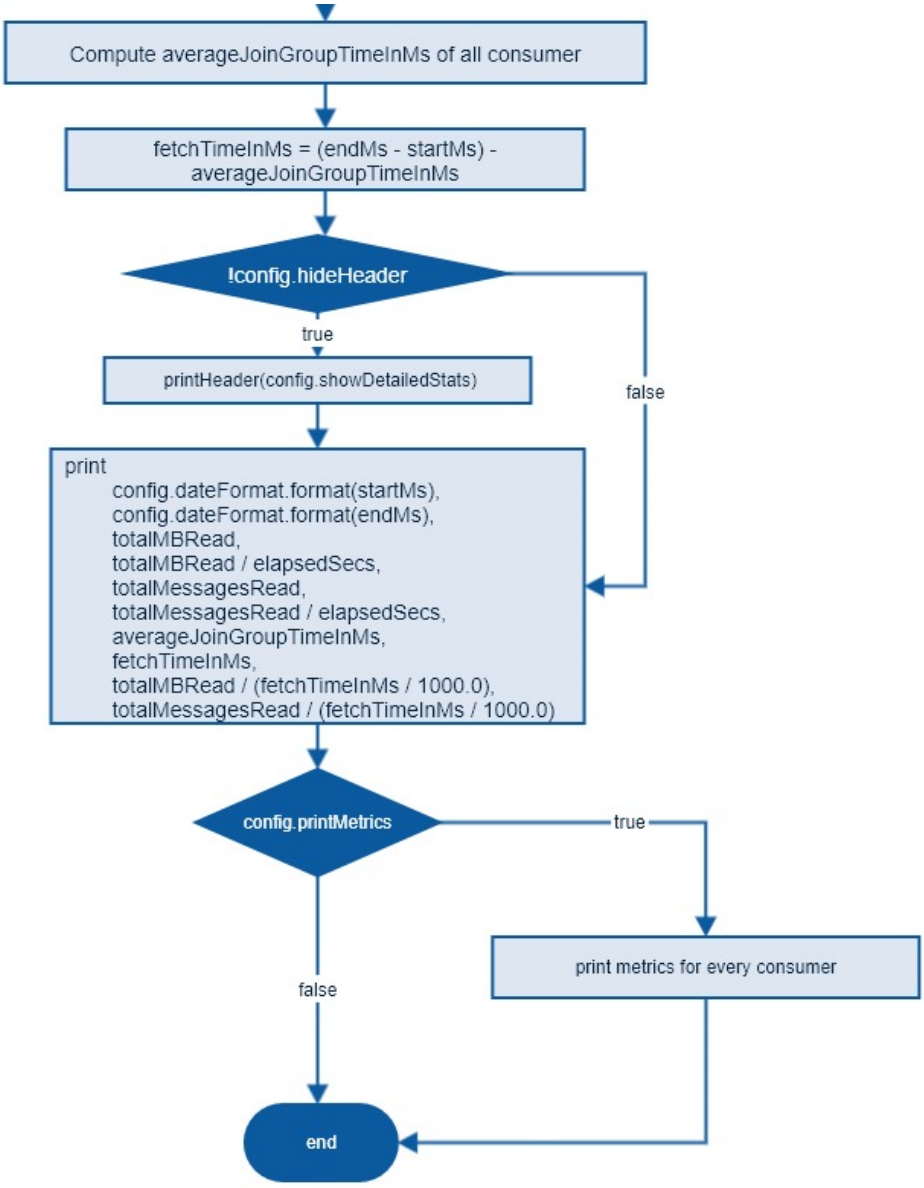
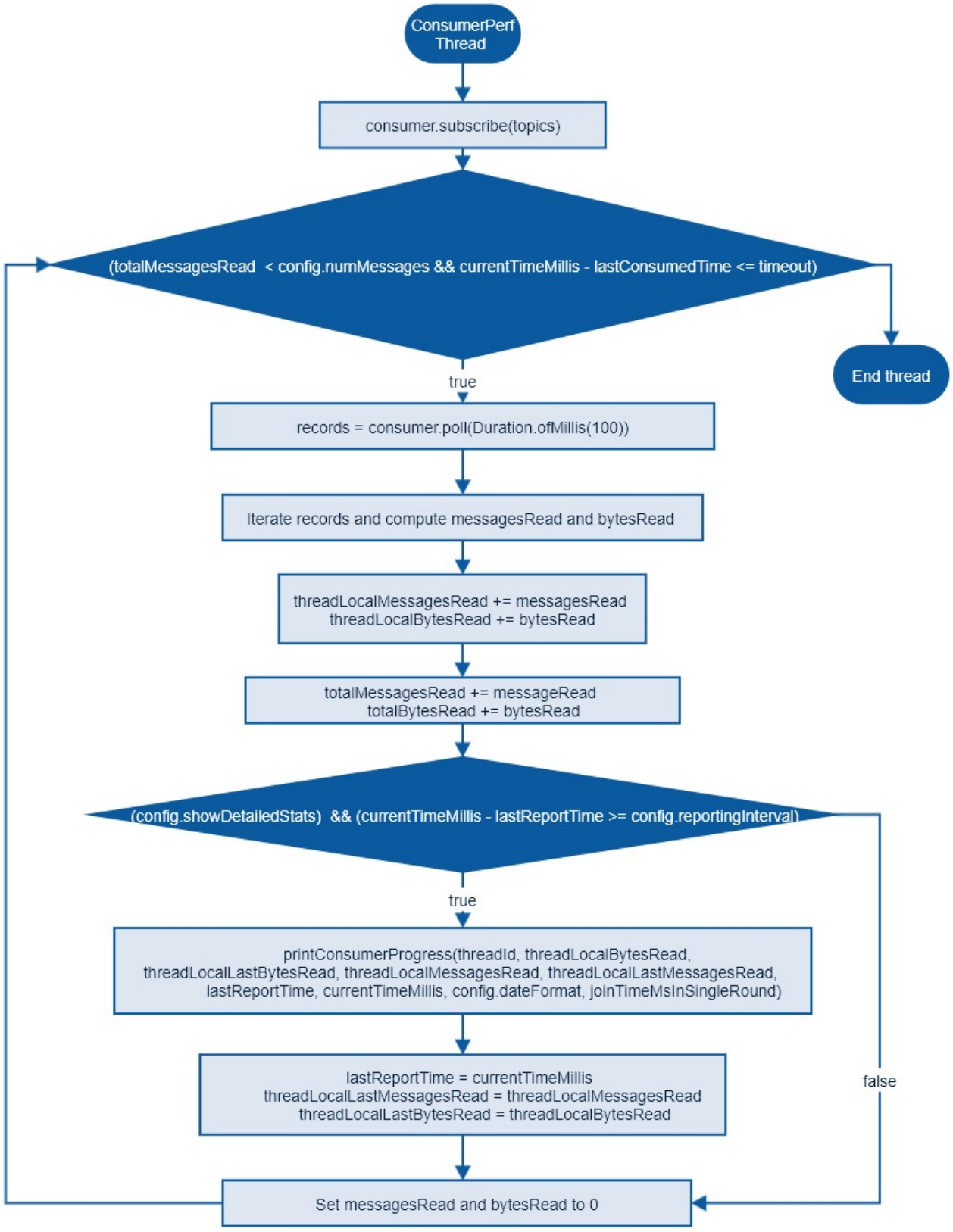


Chart2: ConsumerPerfThread



Below is a general implement for it.

ConsumerPerformance.scala

```
package kafka.tools

import java.text.SimpleDateFormat
import java.time.Duration
import java.util
import java.util.concurrent.{ExecutorService, Executors}
import java.util.concurrent.atomic.LongAdder
import java.util.{Properties, Random}

import com.typesafe.scalalogging.LazyLogging
import kafka.utils.{CommandLineUtils, ToolsUtils}
import org.apache.kafka.clients.admin.{Admin, TopicDescription}
import org.apache.kafka.clients.consumer.{ConsumerConfig, KafkaConsumer}
import org.apache.kafka.common.serialization.ByteArrayDeserializer
import org.apache.kafka.common.utils.Utils
import org.apache.kafka.common.{KafkaFuture, Metric, MetricName}

import scala.jdk.CollectionConverters._
import scala.collection.mutable

object ConsumerPerformance extends LazyLogging {

  def main(args: Array[String]): Unit = {
    val totalMessagesRead : LongAdder = new LongAdder()
    val totalBytesRead : LongAdder = new LongAdder()
    val config = new ConsumerPerfConfig(args)
    logger.info("Starting consumer...")
    var joinGroupTimeInMs:Double = 0
    if (!config.hideHeader)
      printHeader(config.showDetailedStats)

    var startMs, endMs = 0L
    var numThreads = config.numThreads
    val cluster_props = new Properties
    cluster_props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, config.props.getProperty(ConsumerConfig.
BOOTSTRAP_SERVERS_CONFIG))
    val topicService = Admin.create(cluster_props)
    val topicsInfo: util.Map[String, KafkaFuture[TopicDescription]] = topicService.describeTopics(Seq(config.
topic).asJavaCollection).values()
    val numPartitions = topicsInfo.get(config.topic).get().partitions().size()
    if (numThreads > numPartitions) {
      println(s"Warning: numThreads $numThreads is bigger than numPartition $numPartitions, set numThread to
numPartition")
      numThreads = numPartitions
    }
    val threadPool:ExecutorService=Executors.newFixedThreadPool(numThreads)
    val consumers = new Array[KafkaConsumer[Array[Byte], Array[Byte]]](numThreads)
    val metrics = new Array[mutable.Map[MetricName, _ <: Metric]](numThreads)
    startMs = System.currentTimeMillis
    for(i <- 0 until numThreads) {
      consumers(i) = new KafkaConsumer[Array[Byte], Array[Byte]](config.props)
      metrics(i) = consumers(i).metrics.asScala
      threadPool.execute(new ConsumerPerfThread(i, consumers(i), List(config.topic), config.
recordFetchTimeoutMs, config, totalMessagesRead, totalBytesRead, metrics(i)))
    }
    threadPool.shutdown()
    import java.util.concurrent.TimeUnit
    threadPool.awaitTermination(10, TimeUnit.HOURS)
    endMs = System.currentTimeMillis
    if (totalMessagesRead.sum() < config.numMessages)
      println(s"WARNING: Exiting before consuming the expected number of messages: timeout (${config.
recordFetchTimeoutMs} ms) exceeded. " +
        "You can use the --timeout option to increase the timeout.")

    for(i <- 0 until numThreads) {
      metrics(i).find {
        case (name, _) => name.name() == "rebalance-latency-avg"
      }.map {
        case (_, metric) => joinGroupTimeInMs += metric.metricValue.asInstanceOf[Double]
      }.getOrElse(0)
          }
```

```
    joinGroupTimeInMs = joinGroupTimeInMs / numThreads

    val elapsedSecs = (endMs - startMs) / 1000.0
    val fetchTimeInMs = (endMs - startMs) - joinGroupTimeInMs
    if (!config.showDetailedStats) {
      val totalMBRead = (totalBytesRead.sum() * 1.0) / (1024 * 1024)
      println("%s, %s, %.4f, %.4f, %d, %.4f, %.4f, %.4f, %.4f, %.4f".format(
        config.dateFormat.format(startMs),
        config.dateFormat.format(endMs),
        totalMBRead,
        totalMBRead / elapsedSecs,
        totalMessagesRead.sum(),
        totalMessagesRead.sum() / elapsedSecs,
        joinGroupTimeInMs,
        fetchTimeInMs,
        totalMBRead / (fetchTimeInMs / 1000.0),
        totalMessagesRead.sum() / (fetchTimeInMs / 1000.0)
      ))
    }

    if (config.printMetrics ) {
      for (i <- 0 until numThreads) {
        println(s"Metrics for thread $i")
        if (metrics(i) != null)
        ToolsUtils.printMetrics(metrics(i))
      }
    }
  }

  private[tools] def printHeader(showDetailedStats: Boolean): Unit = {
    val newFieldsInHeader = ", rebalance.time.ms, fetch.time.ms, fetch.MB.sec, fetch.nMsg.sec"
    if (!showDetailedStats)
      println("start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg, nMsg.sec" +
newFieldsInHeader)
    else
      println("time, threadId, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg, nMsg.sec" +
newFieldsInHeader)
  }

  private def printBasicProgress(id: Int,
                                 bytesRead: Long,
                                 lastBytesRead: Long,
                                 messagesRead: Long,
                                 lastMessagesRead: Long,
                                 startMs: Long,
                                 endMs: Long,
                                 dateFormat: SimpleDateFormat): Unit = {
    val elapsedMs: Double = (endMs - startMs).toDouble
    val totalMbRead = (bytesRead * 1.0) / (1024 * 1024)
    val intervalMbRead = ((bytesRead - lastBytesRead) * 1.0) / (1024 * 1024)
    val intervalMbPerSec = 1000.0 * intervalMbRead / elapsedMs
    val intervalMessagesPerSec = ((messagesRead - lastMessagesRead) / elapsedMs) * 1000.0
    print("%s, %d, %.4f, %.4f, %d, %.4f".format(dateFormat.format(endMs), id, totalMbRead,
      intervalMbPerSec, messagesRead, intervalMessagesPerSec))
  }

  private def printExtendedProgress(bytesRead: Long,
                                    lastBytesRead: Long,
                                    messagesRead: Long,
                                    lastMessagesRead: Long,
                                    startMs: Long,
                                    endMs: Long,
                                    periodicJoinTimeInMs: Double): Unit = {
    val fetchTimeMs = endMs - startMs - periodicJoinTimeInMs
    val intervalMbRead = ((bytesRead - lastBytesRead) * 1.0) / (1024 * 1024)
    val intervalMessagesRead = messagesRead - lastMessagesRead
    val (intervalMbPerSec, intervalMessagesPerSec) = if (fetchTimeMs <= 0)
      (0.0, 0.0)
    else
      (1000.0 * intervalMbRead / fetchTimeMs, 1000.0 * intervalMessagesRead / fetchTimeMs)
    print(", %.4f, %.4f, %.4f, %.4f".format(periodicJoinTimeInMs, fetchTimeMs, intervalMbPerSec,
```

```
intervalMessagesPerSec))
  }

  def printConsumerProgress(id: Int,
                            bytesRead: Long,
                            lastBytesRead: Long,
                            messagesRead: Long,
                            lastMessagesRead: Long,
                            startMs: Long,
                            endMs: Long,
                            dateFormat: SimpleDateFormat,
                            periodicJoinTimeInMs: Double): Unit = {
    val elapsedMs: Double = (endMs - startMs).toDouble
    val totalMbRead = (bytesRead * 1.0) / (1024 * 1024)
    val intervalMbRead = ((bytesRead - lastBytesRead) * 1.0) / (1024 * 1024)
    val intervalMbPerSec = 1000.0 * intervalMbRead / elapsedMs
    val intervalMessagesRead = messagesRead - lastMessagesRead
    val intervalMessagesPerSec = ((messagesRead - lastMessagesRead) / elapsedMs) * 1000.0
    val fetchTimeMs = endMs - startMs - periodicJoinTimeInMs

    val (fetchIntervalMbPerSec, fetchIntervalMessagesPerSec) = if (fetchTimeMs <= 0)
      (0.0, 0.0)
    else
      (1000.0 * intervalMbRead / fetchTimeMs, 1000.0 * intervalMessagesRead / fetchTimeMs)
    println("%s, %d, %.4f, %.4f, %d, %.4f, %.4f, %.4f, %.4f, %.4f".format(dateFormat.format(endMs), id,
totalMbRead,
      intervalMbPerSec, messagesRead, intervalMessagesPerSec, periodicJoinTimeInMs, fetchTimeMs,
fetchIntervalMbPerSec, fetchIntervalMessagesPerSec))

  }

  class ConsumerPerfConfig(args: Array[String]) extends PerfConfig(args) {
    val brokerListOpt = parser.accepts("broker-list", "DEPRECATED, use --bootstrap-server instead; ignored if --
bootstrap-server is specified.  The broker list string in the form HOST1:PORT1,HOST2:PORT2.")
      .withRequiredArg
      .describedAs("broker-list")
      .ofType(classOf[String])
    val bootstrapServerOpt = parser.accepts("bootstrap-server", "REQUIRED unless --broker-list(deprecated) is
specified. The server(s) to connect to.")
      .requiredUnless("broker-list")
      .withRequiredArg
      .describedAs("server to connect to")
      .ofType(classOf[String])
    val topicOpt = parser.accepts("topic", "REQUIRED: The topic to consume from.")
      .withRequiredArg
      .describedAs("topic")
      .ofType(classOf[String])
    val groupIdOpt = parser.accepts("group", "The group id to consume on.")
      .withRequiredArg
      .describedAs("gid")
      .defaultsTo("perf-consumer-" + new Random().nextInt(100000))
      .ofType(classOf[String])
    val fetchSizeOpt = parser.accepts("fetch-size", "The amount of data to fetch in a single request.")
      .withRequiredArg
      .describedAs("size")
      .ofType(classOf[java.lang.Integer])
      .defaultsTo(1024 * 1024)
    val resetBeginningOffsetOpt = parser.accepts("from-latest", "If the consumer does not already have an
established " +
      "offset to consume from, start with the latest message present in the log rather than the earliest
message.")
    val socketBufferSizeOpt = parser.accepts("socket-buffer-size", "The size of the tcp RECV size.")
      .withRequiredArg
      .describedAs("size")
      .ofType(classOf[java.lang.Integer])
      .defaultsTo(2 * 1024 * 1024)
    val numThreadsOpt = parser.accepts("threads", "Number of processing threads.")
      .withRequiredArg
      .describedAs("count")
      .ofType(classOf[java.lang.Integer])
      .defaultsTo(10)
```

```
    val numFetchersOpt = parser.accepts("num-fetch-threads", "Number of fetcher threads.")
      .withRequiredArg
      .describedAs("count")
      .ofType(classOf[java.lang.Integer])
      .defaultsTo(1)
    val consumerConfigOpt = parser.accepts("consumer.config", "Consumer config properties file.")
      .withRequiredArg
      .describedAs("config file")
      .ofType(classOf[String])
    val printMetricsOpt = parser.accepts("print-metrics", "Print out the metrics.")
    val showDetailedStatsOpt = parser.accepts("show-detailed-stats", "If set, stats are reported for each
reporting " +
      "interval as configured by reporting-interval")
    val recordFetchTimeoutOpt = parser.accepts("timeout", "The maximum allowed time in milliseconds between
returned records.")
      .withOptionalArg()
      .describedAs("milliseconds")
      .ofType(classOf[Long])
      .defaultsTo(10000)

    options = parser.parse(args: _*)
    CommandLineUtils.printHelpAndExitIfNeeded(this, "This tool helps in performance test for the full zookeeper
consumer")

    CommandLineUtils.checkRequiredArgs(parser, options, topicOpt, numMessagesOpt)

    val printMetrics = options.has(printMetricsOpt)

    val props = if (options.has(consumerConfigOpt))
      Utils.loadProps(options.valueOf(consumerConfigOpt))
    else
      new Properties

    import org.apache.kafka.clients.consumer.ConsumerConfig

    val brokerHostsAndPorts = options.valueOf(if (options.has(bootstrapServerOpt)) bootstrapServerOpt else
brokerListOpt)
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerHostsAndPorts)
    props.put(ConsumerConfig.GROUP_ID_CONFIG, options.valueOf(groupIdOpt))
    props.put(ConsumerConfig.RECEIVE_BUFFER_CONFIG, options.valueOf(socketBufferSizeOpt).toString)
    props.put(ConsumerConfig.MAX_PARTITION_FETCH_BYTES_CONFIG, options.valueOf(fetchSizeOpt).toString)
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, if (options.has(resetBeginningOffsetOpt)) "latest" else
"earliest")
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, classOf[ByteArrayDeserializer])
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, classOf[ByteArrayDeserializer])
    props.put(ConsumerConfig.CHECK_CRCS_CONFIG, "false")

    val numThreads = options.valueOf(numThreadsOpt).intValue
    val topic = options.valueOf(topicOpt)
    val numMessages = options.valueOf(numMessagesOpt).longValue
    val reportingInterval = options.valueOf(reportingIntervalOpt).intValue
    if (reportingInterval <= 0)
      throw new IllegalArgumentException("Reporting interval must be greater than 0.")
    val showDetailedStats = options.has(showDetailedStatsOpt)
    val dateFormat = new SimpleDateFormat(options.valueOf(dateFormatOpt))
    val hideHeader = options.has(hideHeaderOpt)
    val recordFetchTimeoutMs = options.valueOf(recordFetchTimeoutOpt).longValue()
  }

  class ConsumerPerfThread(threadId: Int, consumer : KafkaConsumer[Array[Byte], Array[Byte]], topics: List
[String], timeout: Long, config : ConsumerPerfConfig, totalMessagesRead : LongAdder,
                           totalBytesRead : LongAdder, metrics : mutable.Map[MetricName, _ <: Metric]) extends
Runnable {

    override def run(): Unit = {
      var messagesRead = 0L
      var threadLocalMessagesRead = 0L
      var threadLocalLastMessagesRead = 0L
      var bytesRead = 0L
      var threadLocalBytesRead = 0L
      var threadLocalLastBytesRead = 0L
```

```
      var joinTimeMsInSingleRound: Double = 0
      consumer.subscribe(topics.asJava)
      // Now start the benchmark
      var currentTimeMillis = System.currentTimeMillis
      var lastReportTime: Long = currentTimeMillis
      var lastConsumedTime = currentTimeMillis

      while (totalMessagesRead.sum() < config.numMessages && currentTimeMillis - lastConsumedTime <= config.
recordFetchTimeoutMs) {
        val records = consumer.poll(Duration.ofMillis(100)).asScala
        currentTimeMillis = System.currentTimeMillis
        if (records.nonEmpty)
          lastConsumedTime = currentTimeMillis
        for (record <- records) {
          messagesRead += 1
          if (record.key != null)
            bytesRead += record.key.size
          if (record.value != null)
            bytesRead += record.value.size

          if (currentTimeMillis - lastReportTime >= config.reportingInterval) {
            if (config.showDetailedStats) {
              val currentThreadLocalBytesRead = threadLocalBytesRead + bytesRead
              val currentThreadLocalMessagesRead = threadLocalMessagesRead + messagesRead

              metrics.find {
                case (name, _) => name.name() == "last-rebalance-seconds-ago"
              }.map {
                case (_, metric) => joinTimeMsInSingleRound = metric.metricValue.asInstanceOf[Double]
              }.getOrElse(0)

             printConsumerProgress(threadId, currentThreadLocalBytesRead, threadLocalLastBytesRead,
currentThreadLocalMessagesRead, threadLocalLastMessagesRead,
              lastReportTime, currentTimeMillis, config.dateFormat, joinTimeMsInSingleRound)
              lastReportTime = currentTimeMillis
              threadLocalLastBytesRead = currentThreadLocalBytesRead
              threadLocalLastMessagesRead = currentThreadLocalMessagesRead
            }
          }
        }
        totalMessagesRead.add(messagesRead)
        totalBytesRead.add(bytesRead)
        threadLocalMessagesRead += messagesRead
        threadLocalBytesRead += bytesRead
        bytesRead = 0
        messagesRead = 0
      }
      consumer.close()

    }
  }
}
```

Compatibility, Deprecation, and Migration Plan

Once this is implemented, we can remove the warning message about deprecating option [threads]

Rejected Alternatives

Just as the multi-thread implement for the old Consumer, the option [numMessages] is the number of messages every thread to consume not the total message to consume for all threads. This way can avoid using global variable totalMesssageRead as a condition to end thread.