Operations

Here is some information on actually running Kafka as a production system. This is meant as a page for people to record their operational and monitoring practices to help people gather knowledge about successfully running Kafka in production. Feel free to add a section for your configuration if you have anything you want to share. There is nothing magically about most of these configurations, you may be able to improve on them, but they may serve as a helpful starting place.

LinkedIn

Hardware

We are using dual quad-core Intel Xeon machines with 24GB of memory. In general this should not matter too much, we only see pretty low CPU usage at peak even with GZIP compression enabled and a number of clients that don't batch requests. The memory is probably more than is needed for caching the active segments of the log.

The disk throughput *is* important. We have 8x7200 rpm SATA drives in a RAID 10 array. In general this is the performance bottleneck, and more disks is more better. Depending on how you configure flush behavior you may or may not benefit from more expensive disks (if you flush often then higher RPM SAS drives may be better).

OS Settings

We use Linux. Ext4 is the filesystem and we run using software RAID 10. We haven't benchmarked filesystems so other filesystems may be superior.

We have added two tuning changes: (1) we upped the number of file descriptors since we have lots of topics and lots of connections, and (2) we upped the max socket buffer size to enable high-performance data transfer between data centers (described here).

Java

```
$ java -version
java version "1.6.0_21"
Java(TM) SE Runtime Environment (build 1.6.0_21-b06)
Java HotSpot(TM) 64-Bit Server VM (build 17.0-b16, mixed mode)
```

Here are our command line options:

```
java -server -Xms3072m -Xmx3072m -XX:NewSize=256m -XX:MaxNewSize=256m -XX:+UseConcMarkSweepGC -XX:
CMSInitiatingOccupancyFraction=70
     -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -Xloggc:logs/gc.log -Djava.awt.
headless=true
     -Dcom.sun.management.jmxremote -classpath <long list of jars>


In 0.8,the GC setting is changed slightly to:
-Xms3g -Xmx3g -XX:NewSize=256m -XX:MaxNewSize=256m -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:
CMSInitiatingOccupancyFraction=30
-XX:+UseCMSInitiatingOccupancyOnly -XX:+CMSConcurrentMTEnabled -XX:+CMSScavengeBeforeRemark -XX:+PrintGCDetails
-XX:+PrintGCDateStamps
-XX:+PrintTenuringDistribution -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -
Xloggc:logs/gc.log
```

Kafka

We are running Kafka 0.7 right now but may move to trunk as we fix bugs.

The most important *server* configurations for performance are those that control the disk flush rate. The more often data is flushed to disk, the more "seek-bound" Kafka will be and the lower the throughput. However we don't hand out data until is is sync'd to disk, so delaying flush also adds some latency. The flush behavior can be set by either giving a timeout (flush at most every 30 seconds, say) or a number of messages (flush every 1000 messages). This can be overridden at the topic level, if desired. The setting always applies to each partition.

The most important *client* configurations for performance are (1) compression, (2) sync vs async production, (3) batch size for async production, (4) fetch size.

Here is our server configuration:

```
kafka.log.default.flush.interval.ms=10000
kafka.log.file.size=1073741824
kafka.log.default.flush.scheduler.interval.ms=2000
kafka.log.flush.interval=3000
kafka.socket.send.buffer=2097152
kafka.socket.receive.buffer=2097152
kafka.monitoring.period.secs=30
kafka.num.threads=8
kafka.log.cleanup.interval.mins=30
kafka.log.retention.hours=168
kafka.zookeeper.sessiontimeoutms=6000
kafka.zookeeper.connection.timeout=2000
kafka.num.partitions=1
```

Client configuration varies a fair amount.

Monitoring

Our monitoring is done though a centralized monitoring system custom to LinkedIn, but it keys off the JMX stats exposed from Kafka. To see what is available the easiest thing is just to start a Kafka broker and/or client and fire up JConsole and take a look.

Server Stats

- bean name: kafka:type=kafka.SocketServerStats

```
  def getProduceRequestsPerSecond: Double
  def getFetchRequestsPerSecond: Double
  def getAvgProduceRequestMs: Double
  def getMaxProduceRequestMs: Double
  def getAvgFetchRequestMs: Double
  def getMaxFetchRequestMs: Double
  def getBytesReadPerSecond: Double
  def getBytesWrittenPerSecond: Double
  def getNumFetchRequests: Long
  def getNumProduceRequests: Long
  def getTotalBytesRead: Long
  def getTotalBytesWritten: Long
  def getTotalFetchRequestMs: Long
  def getTotalProduceRequestMs: Long
```

- bean name: kafka:type=kafka.BrokerAllTopicStat kafka:type=kafka.BrokerAllTopicStat.[topic]

```
  def getMessagesIn: Long
  def getBytesIn: Long
  def getBytesOut: Long
  def getFailedProduceRequest: Long
  def getFailedFetchRequest: Long
```

- bean name: kafka:type=kafka.LogFlushStats

```
def getFlushesPerSecond: Double
def getAvgFlushMs: Double
def getTotalFlushMs: Long
def getMaxFlushMs: Double
def getNumFlushes: Long
```

Producer stats

- bean name: kafka:type=kafka.KafkaProducerStats

```
def getProduceRequestsPerSecond: Double
def getAvgProduceRequestMs: Double
def getMaxProduceRequestMs: Double
def getNumProduceRequests: Long
```

- bean name: kafka.producer.Producer:type=AsyncProducerStats

```
def getAsyncProducerEvents: Int
def getAsyncProducerDroppedEvents: Int
```

Consumer stats

- bean name: kafka:type=kafka.ConsumerStats

```
def getPartOwnerStats: String
def getConsumerGroup: String
def getOffsetLag(topic: String, brokerId: Int, partitionId: Int): Long
def getConsumedOffset(topic: String, brokerId: Int, partitionId: Int): Long
def getLatestOffset(topic: String, brokerId: Int, partitionId: Int): Long
```

- bean name: kafka:type=kafka.ConsumerAllTopicStat kafka:type=kafka.ConsumerTopicStat.[topic]

```
def getMessagesPerTopic: Long
def getBytesPerTopic: Long
```

- bean name: kafka:type=kafka.SimpleConsumerStats

```
def getFetchRequestsPerSecond: Double
def getAvgFetchRequestMs: Double
def getMaxFetchRequestMs: Double
def getNumFetchRequests: Long
def getConsumerThroughput: Double
```

Audit

The final alerting we do is on the correctness of the data delivery. We audit that every message that is sent is consumed by all consumers and measure the lag for this to occur. For important topics we alert if a certain completeness is not achieved in a certain time period. The details of this are discussed in KAFKA-260.

Zookeeper

Zookeeper is essential for the correct operation of Kafka. There are a number of things that must be done to keep Zookeeper running happily as we have learned the hard way, hopefully Dave and Neha will add this since I don't know what we did.

Stable version

At LinkedIn, we are running Zookeeper 3.3.*. Version 3.3.3 has known serious issues regarding ephemeral node deletion and session expirations. After running into those issues in production, we upgraded to 3.3.4 and have been running that smoothly for 1/2 year now.

Operationalizing Zookeeper

Operationally, we do the following for a healthy Zookeeper installation -

1. Redundancy in the physical/hardware/network layout: try not to put them all in the same rack, decent (but don't go nuts) hardware, try to keep redundant power and network paths, etc
2. I/O segregation: if you do a lot of write type traffic you'll almost definitely want the transaction logs on a different disk group than application logs and snapshots (the write to the Zookeeper service has a synchronous write to disk, which can be slow).
3. Application segregation: Unless you really understand the application patterns of other apps that you want to install on the same box, it can be a good idea to run Zookeeper in isolation (though this can be a balancing act with the capabilities of the hardware).
4. Use care with virtualization: It can work, depending on your cluster layout and read/write patterns and SLAs, but the tiny overheads introduced by the virtualization layer can add up and throw off Zookeeper, as it can be very time sensitive
5. Zookeeper configuration and monitoring: It's java, make sure you give it 'enough' heap space (We usually run them with 3-5G, but that's mostly due to the data set size we have here). Unfortunately we don't have a good formula for it. As far as monitoring, both JMZ and the 4 letter commands are very useful, they do overlap in some cases (and in those cases we prefer the 4 letter commands, they seem more predictable, or at the very least, they work better with the LI monitoring infrastructure)
6. Don't overbuild the cluster: large clusters, especially in a write heavy usage pattern, means a lot of intracluster communication (quorums on the writes and subsequent cluster member updates), but don't underbuild it (and risk swamping the cluster).
7. Try to run on a 3-5 node cluster: Zookeeper writes use quorums and inherently that means having an odd number of machines in a cluster. Remember that a 5 node cluster will cause writes to slow down compared to a 3 node cluster, but will allow more fault tolerance.

Overall, we try to keep the Zookeeper system as small as will handle the load (plus standard growth capacity planning) and as simple as possible. We try not to do anything fancy with the configuration or application layout as compared to the official release as well as keep it as self contained as possible. For these reasons, we tend to skip the OS packaged versions, since it has a tendency to try to put things in the OS standard hierarchy, which can be 'messy', for want of a better way to word it.