# Comparing Kafka and ActiveMQ for Purple Nectar

In this document I describe my findings regarding the performance, capabilities and usability of Kafka and ActiveMQ as a message bus for the Purple Nectar project. My findings on several aspects are briefly described and compared for both Kafka and ActiveMQ. The source code and installation instructions can be found at <https://github.com/PHameete/mqtesting/blob/master/> .

## Java Consumer / Producer

For both messaging systems I have developed a similar app that attempts to produce and consume 1 million small JSON messages as fast as possible. It maintains statistics (min, max, average, 99th percentile) on the latency of the consumed messages.

Development for both messaging systems was simple by using the provided APIs. In my opinion Kafka was slightly easier to use and configure. Moreover Kafka has a more convenient way to subscribe to multiple topics simultaneously.

With regards to the latency, Kafka performs much better in this case (loads of small messages). Kafka resulted in an average latency of 5.7ms compared to 3497.5 for ActiveMQ. Moreover, producing and consuming the total 1 million messages was much faster with Kafka. Many more thorough benchmarks confirm this (you can find plenty using Google, for example <https://softwaremill.com/mqperf/>).

**Kafka Streaming**

Kafka has released its streaming API (see <http://kafka.apache.org/0100/javadoc/index.html?org/apache/kafka/streams/KafkaStreams.html>) with version 0.10 which offers very low latency one-at-a-time processing of messages. This also makes it very easy to develop and run an extra process which can convert the messages in one topic to a different format, and then publish it to a next topic.

## NodeJS Consumer / Producer

For both messaging systems I have developed a very simple NodeJS producer and consumer app that interchange simple small text messages. For both messaging systems the NodeJS programs worked fine and interoperated perfectly with the Java producers and consumers.

**Kafka**

For Kafka I have made use of the **kafka-node** npm package (<https://www.npmjs.com/package/kafka-node>). The source code can be found at <https://github.com/SOHU-Co/kafka-node>.

**ActiveMQ**

For ActiveMQ I have used the **stompit** npm package (<https://www.npmjs.com/package/stompit>). The source code can be found at <https://github.com/gdaws/node-stomp>. STOMP is the simple text oriented messaging protocol which is an interface implemented by ActiveMQ. The **stompit** npm package uses this interface to ActiveMQ.

## REST Interface

Both Kafka and ActiveMQ are have a working REST interface to allow producing and consuming messages and creating of topics. For ActiveMQ this is a natively offered REST interface, see <http://activemq.apache.org/rest.html> . For Kafka this is an open souce REST proxy offered by Confluent, see <https://github.com/confluentinc/kafka-rest>.

## Exchanging Large Messages

In order to investigate the capability of handling large GeoJSON files and the point cloud I developed a Java consumer and producer for both messaging systems which exchanges a thousand 5MB GeoJSON messages.

**Kafka**

Kafka only supports messages of up to 1MB by default. By changing the configuration of the producer, consumer and broker it is possible to change this to an arbitrary amount as long as a message can fit into the available memory. See the Github readme at <https://github.com/PHameete/mqtesting/blob/master/README.md> for the exact parameters.

**ActiveMQ**

ActiveMQ supports large messages out of the box. When using compressions 5MB messages could be exchanged fine. However, without compression the ActiveMQ broker would crash.

**Performance comparison**

The performance for large messages is very similar between Kafka and ActiveMQ (about ten 5MB messages per second in my local setup). As stated in other tests Kafka excels at processing smaller messages (up to 10KB). When sending large messages the IO is probably the bottleneck instead.

**Alternatives for exchanging large messages**

People strongly advise to not exchange large messages using messaging services like ActiveMQ and Kafka. Instead it is suggested to place large files in a shared file system (such as FTP, or the HDFS) and send messages with references to the shared file system. For ActiveMQ this can be done by using BlobMessages. For Kafka such a solution needs to be implemented manually, but this should be simple.

## Time Ordering

Both Kafka and ActiveMQ guarantee time ordering for topics. This means that the messages sent by a producer are received by consumers in the order in which they were sent. For Kafka this only holds when a topic consists of one partition.

**Important**: For Kafka a consume can control where it starts consuming messages in a topic. This means it is possible to go back in time, or read every message since the beginning of a topic. For ActiveMQ this is not possible because the broker handles the position of each consumer.

## Acknowledgement of received messages

Both Kafka and ActiveMQ support acknowledgement of messages received by the broker. This ensures that it can be detected when sent messages do not arrive which is important to ensure that the state of the simulation is correct.

## Authorization per topic

Both Kafka and ActiveMQ support authorization per topic. For Kafka see <http://kafka.apache.org/documentation.html#security_authz> and for ActiveMQ see <http://activemq.apache.org/security.html>.