

Apache Kafka: Next Generation Distributed Messaging System

KHIN ME ME THEIN

Ph.D Scholar, UCSY, Myanmar, E-mail: khinmemethein@gmail.com.

Abstract: Apache Kafka is publish-subscribe messaging implemented as a distributed commit log, suitable for both offline and online message consumption. It is a messaging system initially developed at LinkedIn for collecting and delivering high volumes of event and log data with low latency. Message publishing is a mechanism for connecting various applications with the help of messages that are routed between them, for example, by a message broker such as Kafka. It acts as a kind of write-ahead log that records messages to a persistent store and allows subscribers to read and apply these changes to their own stores in a system appropriate time-frame. Common subscribers include live services that do message aggregation or other processing streams, as well as Hadoop and data warehousing pipelines which load virtually all feeds for batch-oriented processing.

Keywords: Publish-Subscribe Messaging, Distributed, Aggregation, Batch-Oriented.

I. INTRODUCTION

In today's world, real-time information is continuously getting generated by applications (business, social, or any other type), and this information needs easy ways to be reliably and quickly routed to multiple types of receivers. Most of the time, applications that are producing information and applications that are consuming this information are well apart and inaccessible to each other. This, at times, leads to redevelopment of information producers or consumers to provide an integration point between them. Therefore, a mechanism is required for seamless integration of information of producers and consumers to avoid any kind of rewriting of an application at either end [4]. In the present big data era, the very first challenge is to collect the data as it is a huge amount of data and the second challenge is to analyze it. This analysis typically includes following type of data and much more:

- User behavior data
- Application performance tracing
- Activity data in the form of logs
- Event messages

Kafka is a solution to the real-time problems of any software solution, that is, to deal with real-time volumes of information and route it to multiple consumers quickly. Kafka provides seamless integration between information of producers and consumers without blocking the producers of the information and without letting producers know who the final consumers are. Kafka [2] is distributed and scalable, and offers high throughput. On the other hand, Kafka provides an API similar to a messaging system and allows applications to consume log events in real time. Kafka has been open sourced and used successfully in production at LinkedIn for more than 6 months. It greatly simplifies the infrastructure, since LinkedIn can exploit a single piece of

software for both online and offline consumption of log data of all types. The rest of this paper is organized as follows. We show the related work of Kafka in Section 2. In Section 3, we describe the architecture of Kafka and its design and about the zookeeper which needs to run Kafka. We describe our deployment of Kafka at console in Section 4. We discuss future work and conclude in Section 5.

II. RELATED WORK

In a modern data architecture that is built on YARN - enabled (Apache Hadoop NextGen MapReduce) Apache Hadoop [1], Kafka works in combination with Apache Storm, Apache Hbase and Apache Spark for real-time analysis and rendering of streaming data. Kafka can message geospatial data from fleet of long-haul trucks or sensor data from heating and cooling equipment in office buildings. Whatever the industry or use case, Kafka brokers massive message streams for low-latency analysis in Enterprise Apache Hadoop. Kafka is fully supported and included in HDP (HORTONWORKS DATA PLATFORM) today. Some of the companies that are using Apache Kafka in their respective use cases are as follow [4]:

LinkedIn (www.linkedin.com): Apache Kafka is used at LinkedIn for the streaming of activity data and operational metrics. This data powers various products such as LinkedIn news feed and LinkedIn Today in addition to offline analytics systems such as Hadoop.

DataSift (www.datasift.com/): At DataSift, Kafka is used as a collector for monitoring events and as a tracker of users' consumption of data streams in real time.

Twitter (www.twitter.com/): Twitter uses Kafka as a part of its Storm— a stream-processing infrastructure.

Foursquare (www.foursquare.com/): Kafka powers online-to-online and online-to-offline messaging at Foursquare. It is used to integrate foursquare monitoring and production systems with Foursquare, Hadoop-based offline infrastructures.

Square (www.squareup.com/): Square uses Kafka as a bus to move all system events through Square's various datacenters. This includes metrics, logs, custom events, and so on. On the consumer side, it outputs into Splunk, Graphite, or Esper-like real-time alerting.

III. KAFKA ARCHITECTURE AND DESIGN

Kafka is a distributed, partitioned, replicated commit log service. Kafka [3] maintains feeds of messages in categories called topics. We'll call processes that publish messages to a Kafka topic are producers. And we'll call processes that subscribe to topics and process the feed of published messages are consumers. Kafka is run as a cluster comprised of one or more servers each of which is called a broker. At a high level, producers send messages over the network to the Kafka cluster which in turn serves them up to consumers like this in Fig.1.

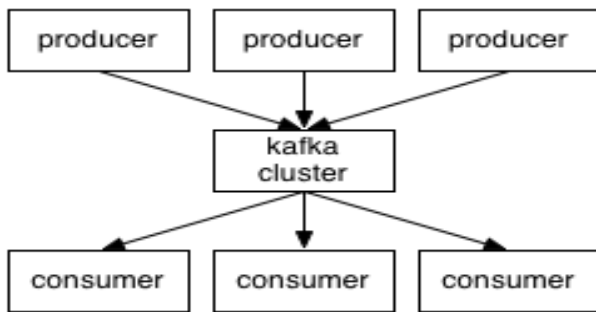


Fig.1. Basic architecture of Kafka.

Producers publish messages to Kafka topics, and consumers subscribe to these topics and consume the messages. A server in a Kafka cluster is called a broker. For each topic, the Kafka cluster maintains a partition for scaling, parallelism and fault-tolerance. Each partition is an ordered, immutable sequence of messages that is continually appended to a commit log. The messages in the partitions are each assigned a sequential id number called the offset. Anatomy of a topic is described in Fig.2.

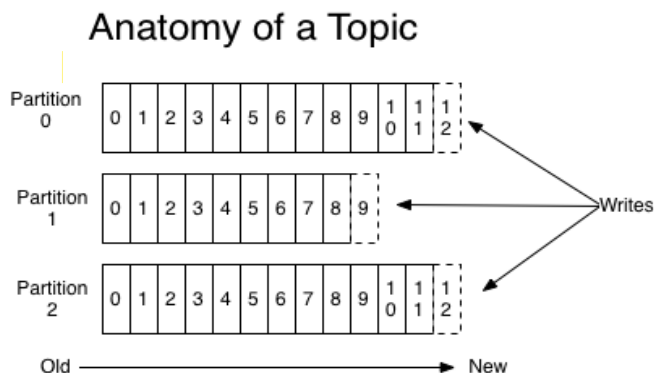


Fig.2. Anatomy of a topic.

The offset is controlled by the consumer. The typical consumer will process the next message in the list, although it can consume messages in any order, as the Kafka cluster retains all published messages for a configurable period of time. This makes consumers very cheap, as they can come and go without much impact on the cluster, and allows for offline consumers like Hadoop clusters. Producers are able to choose which topic, and which partition within the topic, to publish the message to. Consumers assign themselves a consumer group name, and each message is delivered to one consumer within each subscribing consumer group. If all the consumers have different consumer groups, then messages are broadcasted to each consumer. Kafka can be used like a traditional message broker. It has high throughput, built-in partitioning, replication, and fault-tolerance, which makes it a good solution for large scale message processing applications. Kafka can also be used for high volume website activity tracking. Site activity can be published, and can be processed real-time, or loaded into Hadoop or offline data warehousing systems. Kafka can also be used as a log aggregation solution. Instead of working with log files, logs can be treated a stream of messages [5].

Kafka is also an open source, distributed publish-subscribe messaging system, mainly designed with the following characteristics:

Persistent Messaging: To derive the real value from big data, any kind of information loss cannot be afforded. Apache Kafka is designed with $O(1)$ disk structures that provide constant-time performance even with very large volumes of stored messages, which is in order of TB.

High Throughput: Keeping big data in mind, Kafka is designed to work on commodity hardware and to support millions of messages per second.

Distributed: Apache Kafka explicitly supports messages partitioning over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics.

Multiple Client Support: Apache Kafka system supports easy integration of clients from different platforms such as Java, .NET, PHP, Ruby, and Python.

Real Time: Messages produced by the producer threads should be immediately visible to consumer threads; this feature is critical to event-based systems such as Complex Event Processing (CEP) systems.

Kafka provides a real-time publish-subscribe solution, which overcomes the challenges of real-time data usage for consumption, for data volumes that may grow in order of magnitude, larger than the real data. Kafka also supports parallel data loading in the Hadoop systems [4]. The overall architecture of Kafka is shown in Fig.3. Since **Kafka is distributed in nature**, a Kafka cluster **typically consists of multiple brokers**. To balance load, a topic is divided into multiple partitions and each broker stores one or more of

Apache Kafka: Next Generation Distributed Messaging System

these partitions. Multiple producers and consumers can publish and retrieve messages at the same time [2]. In section 3.1, we describe about the producers and about the consumers is in section 3.2. And we describe about the Zookeeper in section 3.3.

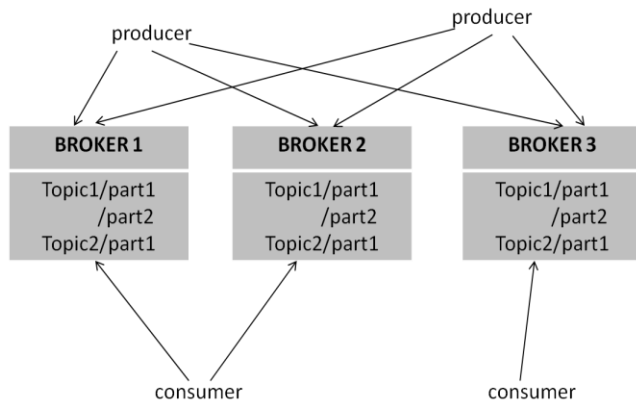


Fig.3. Kafka Architecture.

A. Producers

Producers publish data to the topics of their choice. The producer is responsible for choosing which message to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (say based on some key in the message) [3].

B. Consumers

Messaging traditionally has two models: queuing and publish-subscribe. In a queue, a pool of consumers may read from a server and each message goes to one of them; in publish-subscribe the message is broadcast to all consumers. Kafka offers a single consumer abstraction that generalizes both of these – the consumer group [3]. Consumers label themselves with a consumer group name, and each message published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines. If all the consumer instances have the same consumer group, then this works just like a traditional queue balancing load over the consumers. If all the consumer instances have different consumer groups, then this works like publish-subscribe and all messages ARE broadcast to all consumers. By having a notion of parallelism—the partition—within the topics, Kafka is able to provide both ordering guarantees and load balancing over a pool of consumer processes.

This is achieved by assigning the partitions in the topic to the consumers in the consumer group so that each partition is consumed by exactly one consumer in the group. By doing this we ensure that the consumer is the only reader of that partition and consumes the data in order. Since there are many partitions this still balances the load over many consumer instances. Note however that there cannot be more consumer instances than partitions. Fig.4 shows the example of consumer groups. Kafka only provides a total

order over messages within a partition, not between different partitions in a topic. Per-partition ordering combined with the ability to partition data by key is sufficient for most applications. However, if you require a total order over messages this can be achieved with a topic that has only one partition, though this will mean only one consumer process.

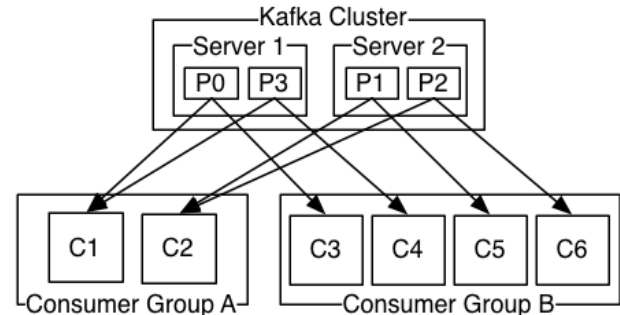


Fig.4. A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups. Consumer group A has two consumer instances and group B has four.

C. Zookeeper

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skip on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed [6]. ZooKeeper is also a high-performance coordination service for distributed applications. It exposes common services - such as naming, configuration management, synchronization, and group services - in a simple interface so you don't have to write them from scratch. You can use it off-the-shelf to implement consensus, group management, leader election, and presence protocols. And you can build on it for your own, specific needs [7]. Kafka provides the default and simple Zookeeper configuration file used for launching a single local Zookeeper instance.

IV. TESTING KAFKA IN CONSOLE

In this section, we'll describe how we test Kafka from command prompt in Ubuntu. First we need to download the Kafka that is suitable with Ubuntu. We use Kafka 0.8.1.1.tgz for this testing.

Step 1: Start the Server

Kafka uses ZooKeeper. It serves as the coordination interface between the Kafka broker and consumers. So, firstly we need to start a ZooKeeper server if you don't already have one. You can use the convenience script

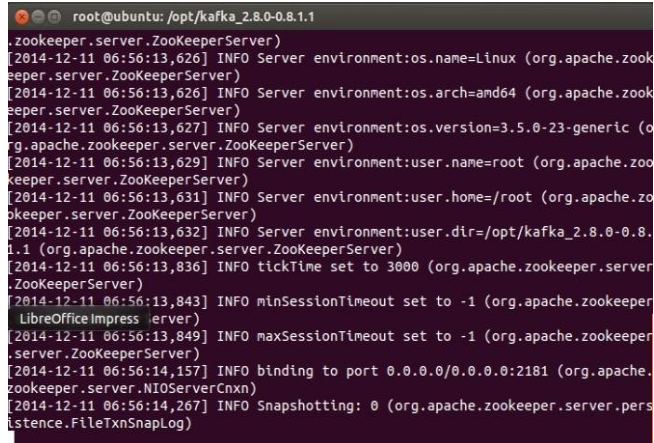
packaged with kafka to get a ZooKeeper instance. By default, the Zookeeper server will listen on *:2181/tcp. First start the Zookeeper using the following command.

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
#bin/zookeeper-server-start.sh config/zookeeper.properties
```

You should get an output as shown in the following screenshot 1.

Screen Shot 1:



```
root@ubuntu:/opt/kafka_2.8.0-0.8.1.1
./zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:13,626] INFO Server environment:os.name=Linux (org.apache.zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:13,626] INFO Server environment:os.arch=amd64 (org.apache.zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:13,627] INFO Server environment:os.version=3.5.0-23-generic (org.apache.zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:13,629] INFO Server environment:user.name=root (org.apache.zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:13,631] INFO Server environment:user.homedir=/root (org.apache.zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:13,632] INFO Server environment:user.dir=/opt/kafka_2.8.0-0.8.1.1 (org.apache.zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:13,836] INFO tickTime set to 3000 (org.apache.zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:13,843] INFO minSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:13,849] INFO maxSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2014-12-11 06:56:14,157] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.zookeeper.server.NIOServerCnxn)
[2014-12-11 06:56:14,267] INFO Snapshotting: 0 (org.apache.zookeeper.server.persistence.FileTxnSnapLog)
```

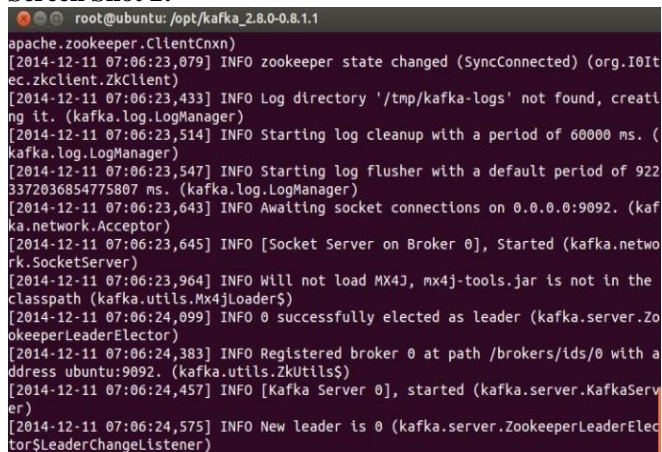
Now start the Kafka server using the following command:

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
#bin/kafka-server-start.sh config/server.properties
```

You should get an output as shown in the following screenshot 2.

Screen Shot 2:



```
root@ubuntu:/opt/kafka_2.8.0-0.8.1.1
apache.zookeeper.ClientCnxn)
[2014-12-11 07:06:23,079] INFO zookeeper state changed (SyncConnected) (org.apache.kafka.clients.ZkClient)
[2014-12-11 07:06:23,433] INFO Log directory '/tmp/kafka-logs' not found, creating it. (kafka.log.LogManager)
[2014-12-11 07:06:23,514] INFO Starting log cleanup with a period of 60000 ms. (kafka.log.LogManager)
[2014-12-11 07:06:23,547] INFO Starting log flusher with a default period of 9223372036854775807 ms. (kafka.log.LogManager)
[2014-12-11 07:06:23,643] INFO Awaiting socket connections on 0.0.0.0:9092. (kafka.network.Acceptor)
[2014-12-11 07:06:23,645] INFO [Socket Server on Broker 0], Started (kafka.network.SocketServer)
[2014-12-11 07:06:23,964] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4jLoader)
[2014-12-11 07:06:24,099] INFO 0 successfully elected as leader (kafka.server.ZooKeeperLeaderElector)
[2014-12-11 07:06:24,383] INFO Registered broker 0 at path /brokers/ids/0 with address ubuntu:9092. (kafka.utils.ZkUtils)
[2014-12-11 07:06:24,457] INFO [Kafka Server 0], started (kafka.server.KafkaServer)
[2014-12-11 07:06:24,575] INFO New leader is 0 (kafka.server.ZooKeeperLeaderElector$LeaderChangeListener)
```

Step 2: Create A Topic

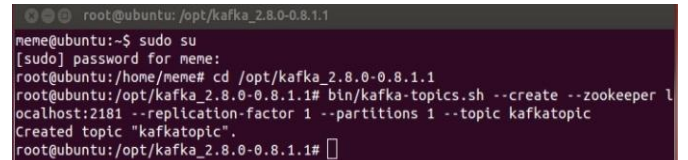
Let's create a topic named "kafkatopic" with a single partition and only one replica:

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
# bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic kafkatopic
```

You should get an output as shown in the following screenshot 3:

Screen Shot 3:



```
root@ubuntu:/opt/kafka_2.8.0-0.8.1.1
meme@ubuntu:~$ sudo su
[sudo] password for meme:
root@ubuntu:/home/meme# cd /opt/kafka_2.8.0-0.8.1.1
root@ubuntu:/opt/kafka_2.8.0-0.8.1.1# bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic kafkatopic
Created topic "kafkatopic".
root@ubuntu:/opt/kafka_2.8.0-0.8.1.1#
```

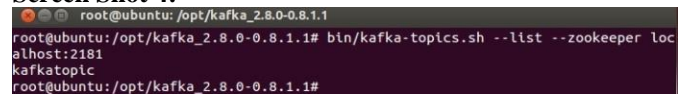
We can now see that topic if we run the list topic command:

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
#bin/kafka-topics.sh --list --zookeeper localhost:2181
```

You should get an output as shown in the following screenshot 4:

Screen Shot 4:



```
root@ubuntu:/opt/kafka_2.8.0-0.8.1.1# bin/kafka-topics.sh --list --zookeeper localhost:2181
kafkatopic
root@ubuntu:/opt/kafka_2.8.0-0.8.1.1#
```

Step 3: Starting A Producer For Sending Messages

Kafka provides users with a command-line producer client that accepts inputs from the command line and publishes them as a message to the Kafka cluster. By default each new line entered is considered as a new message. The following command is used to start the console-based producer for sending the messages.


```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
#bin/kafka-console-producer.sh--broker-list localhost:9092 --topic kafkatopic
```


Apache Kafka: Next Generation Distributed Messaging System

You should get an output as shown in the following screenshot 5:

Screen Shot 5:



```
root@ubuntu: /opt/kafka_2.8.0-0.8.1.1
root@ubuntu: /opt/kafka_2.8.0-0.8.1.1# bin/kafka-console-producer.sh --broker-lis
t localhost:9092 --topic kafkatopic
Welcome to Kafka.
This is a message.
This is another message.
```

Step 4: Start A Consumer

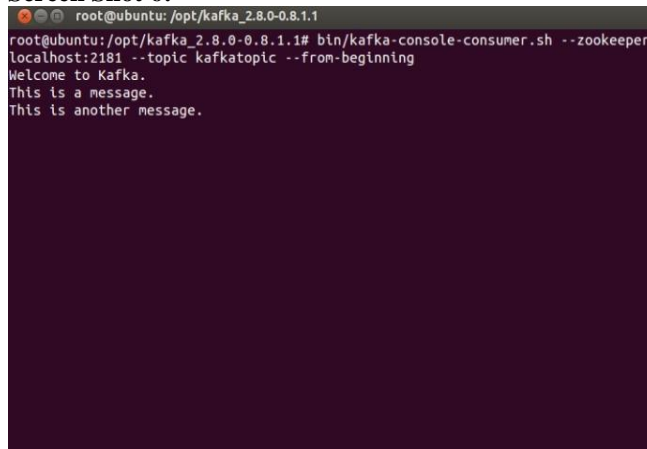
Kafka also provides a command-line consumer client for message consumption. The following command is used for starting the console-based consumer.

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
#bin/kafka-console-consumer.sh—zookeeper local host:
21 81 --topic kafkatopic --from-beginning
```

You should get an output as shown in the following screenshot 6:

Screen Shot 6:



```
root@ubuntu: /opt/kafka_2.8.0-0.8.1.1
root@ubuntu: /opt/kafka_2.8.0-0.8.1.1# bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic kafkatopic --from-beginning
Welcome to Kafka.
This is a message.
This is another message.
```

If you have each of the above commands running in a different terminal then you should now be able to type messages into the producer terminal and see them appear in the consumer terminal.

Step 5: Setting Up A Multi-Broker Cluster

For Kafka, a single broker is just a cluster of size one, so nothing much changes other than starting a few more broker instances. But just to get feel for it, let's expand our

cluster to three nodes (still all on our local machine). First we make a config file for each of the brokers:

```
> cp config/server.properties config/server-1.properties
```

```
> cp config/server.properties config/server-2.properties
```

Now edit these new files by using “vi” command and set the following properties:

config/server-1.properties:

```
broker.id=1
```

```
port=9093
```

```
log.dir=/tmp/kafka-logs-1
```

config/server-2.properties:

```
broker.id=2
```

```
port=9094
```

```
log.dir=/tmp/kafka-logs-2
```

The broker.id property is the unique and permanent name of each node in the cluster. We have to override the port and log directory only because we are running these all on the same machine and we want to keep the brokers from all trying to register on the same port or overwrite each other data.

We already have Zookeeper and our single node started, so we just need to start the two new nodes:

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
#bin/kafka-server-start.sh config/server-1.properties &
```

```
...
```

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
# bin/kafka-server-start.sh config/server-2.properties &
```

```
...
```

Now create a new topic with two partitions and two replicas:

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
# bin/kafka-topics.sh --create --zookeeper localhost:2181 -
-replication-factor 2 --partitions 2 --topic my-replicated-
topic
```

If we use a single producer to get connected to all the brokers, we need to pass the initial list of brokers, and the information of the remaining brokers is identified by

querying the broker passed within broker-list, as shown in the following command to start the producer:

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
# bin/kafka-console-producer.sh --broker-list localhost:9093, localhost: 9094 --topic my-replicated-topic
```

For the consumer, use the previous command to consume messages.

```
[root@ubuntu:/opt/kafka-2.8.0-0.8.1.1]
```

```
#bin/kafka-console-consumer.sh --zookeeper local host:2181 --topic my-replicated-topic --from-beginning
```

V. CONCLUSION AND FUTURE WORK

We present a system called Kafka for processing huge volume of a log data streams. Kafka employs a pull-based consumption model that allows an application to consume data at its own rate and rewind the consumption whenever needed. It achieves much higher throughput than conventional messaging systems. It also provides integrated distributed support and can scale out. Later, we'll test Kafka in API for both producer and consumer by using Scala Programming Language.

VI. REFERENCES

- [1] <http://hortonworks.com/hadoop/kafka/>.
- [2] Jay Kreps, Neha Narkhede and Jun Rao, "Kafka: a Distributed Messaging System for Log Processing", LinkedIn Corp.
- [3] <http://kafka.apache.org>.
- [4] Nishant Garg, "Apache Kafka" , PACKT Publishing.
- [5] <http://www.infoq.com/news/2013/12/apache-afka-messaging-system>.
- [6] <http://zookeeper.apache.org/>.
- [7] <http://zookeeper.apache.org/doc/trunk/>.