Kafka is designed for distributed high throughput systems. Kafka tends to work very well as a replacement for a more traditional message broker. In comparison to other messaging systems, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications.

What is a Messaging System?

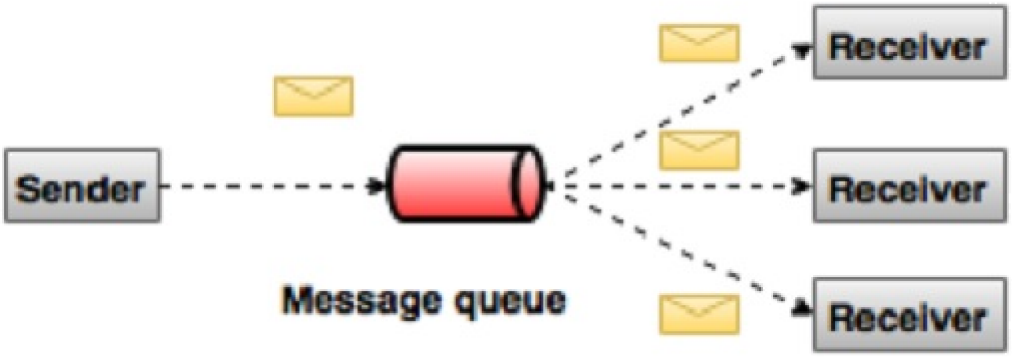
A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it. Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging system. Two types of messaging patterns are available − one is point to point and the other is publish-subscribe (pub-sub) messaging system. Most of the messaging patterns follow pub-sub.

Point to Point Messaging System

In a point-to-point system, messages are persisted in a queue. One or more consumers can consume the messages in the queue, but a particular message can be consumed by a maximum of one consumer only. Once a consumer reads a message in the queue, it disappears from that queue. The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time. The following diagram depicts the structure.

Publish-Subscribe Messaging System

In the publish-subscribe system, messages are persisted in a topic. Unlike point-to-point system, consumers can subscribe to one or more topic and consume all the messages in that topic. In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers. A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



What is Kafka?

Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.

Benefits

Following are a few benefits of Kafka −

- **Reliability** − Kafka is distributed, partitioned, replicated and fault tolerance.
- **Scalability** − Kafka messaging system scales easily without down time..
- **Durability** − Kafka uses "Distributed commit log" which means messages persists on disk as fast as possible, hence it is durable..
- **Performance** − Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even many TB of messages are stored.

Kafka is very fast and guarantees zero downtime and zero data loss.

Use Cases

Kafka can be used in many Use Cases. Some of them are listed below −

- **Metrics** − Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.
- **Log Aggregation Solution** − Kafka can be used across an organization to collect logs from multiple services and make them available in a standard format to multiple con-sumers.
- **Stream Processing** − Popular frameworks such as Storm and Spark Streaming read data from a topic, processes it, and write processed data to a new topic where it becomes available for users and applications. Kafka's strong durability is also very useful in the context of stream processing.

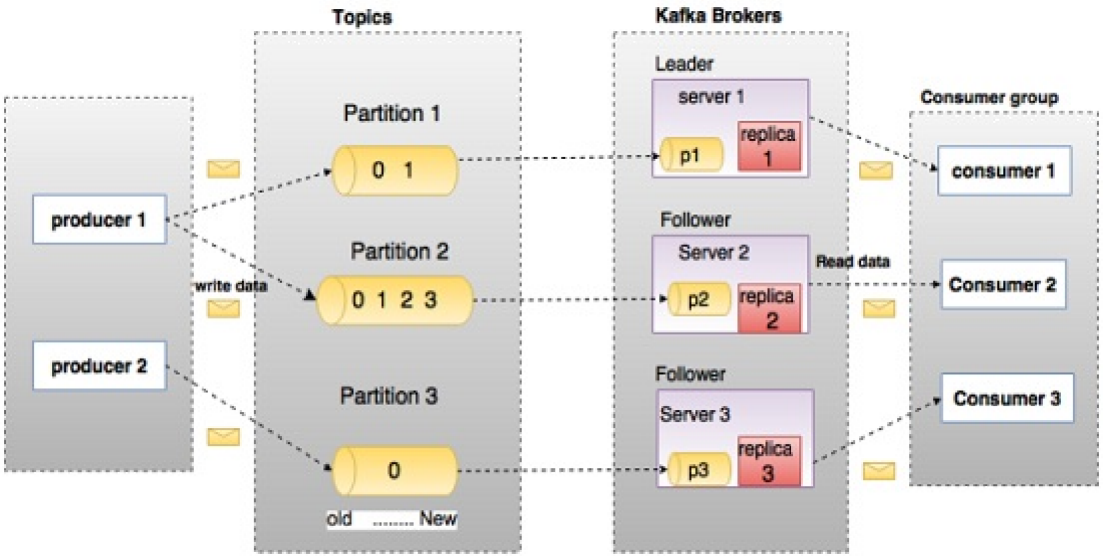
Need for Kafka

Kafka is a unified platform for handling all the real-time data feeds. Kafka supports low latency message delivery and gives guarantee for fault tolerance in the presence of machine failures. It has the

ability to handle a large number of diverse consumers. Kafka is very fast, performs 2 million writes/sec. Kafka persists all data to the disk, which essentially means that all the writes go to the page cache of the OS (RAM). This makes it very efficient to transfer data from page cache to a network socket.

Apache Kafka - Fundamentals

Before moving deep into the Kafka, you must aware of the main terminologies such as topics, brokers, producers and consumers. The following diagram illustrates the main terminologies and the table describes the diagram components in detail.



In the above diagram, a topic is configured into three partitions. Partition 1 has two offset factors 0 and 1. Partition 2 has four offset factors 0, 1, 2, and 3. Partition 3 has one offset factor 0. The id of the replica is same as the id of the server that hosts it.

Assume, if the replication factor of the topic is set to 3, then Kafka will create 3 identical replicas of each partition and place them in the cluster to make available for all its operations. To balance a load in cluster, each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time.

S.No	Components and Description
1	**Topics** A stream of messages belonging to a particular category is called a topic. Data is stored in topics. Topics are split into partitions. For each topic, Kafka keeps a mini-mum of one partition. Each such partition contains messages in an immutable ordered sequence. A partition is implemented as a set of segment files of equal sizes.
2	**Partition** Topics may have many partitions, so it can handle an arbitrary amount of data.
3	**Partition offset**

Each partitioned message has a unique sequence id called as offset.

4 **Replicas of partition**

Replicas are nothing but backups of a partition. Replicas are never read or write data. They are used to prevent data loss.

5 **Brokers**

- Brokers are simple system responsible for maintaining the pub-lished data. Each broker may have zero or more partitions per topic. Assume, if there are N partitions in a topic and N number of brokers, each broker will have one partition.

- Assume if there are N partitions in a topic and more than N brokers (n + m), the first N broker will have one partition and the next M broker will not have any partition for that particular topic.

- Assume if there are N partitions in a topic and less than N brokers (n-m), each broker will have one or more partition sharing among them. This scenario is not recommended due to unequal load distri-bution among the broker.

6 **Kafka Cluster**

Kafka's having more than one broker are called as Kafka cluster. A Kafka cluster can be expanded without downtime. These clusters are used to manage the persistence and replication of message data.

7 **Producers**

Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers. Every time a producer pub-lishes a message to a broker, the broker simply appends the message to the last segment file. Actually, the message will be appended to a partition. Producer can also send messages to a partition of their choice.

8 **Consumers**

Consumers read data from brokers. Consumers subscribes to one or more topics and consume published messages by pulling data from the brokers.

9 **Leader**

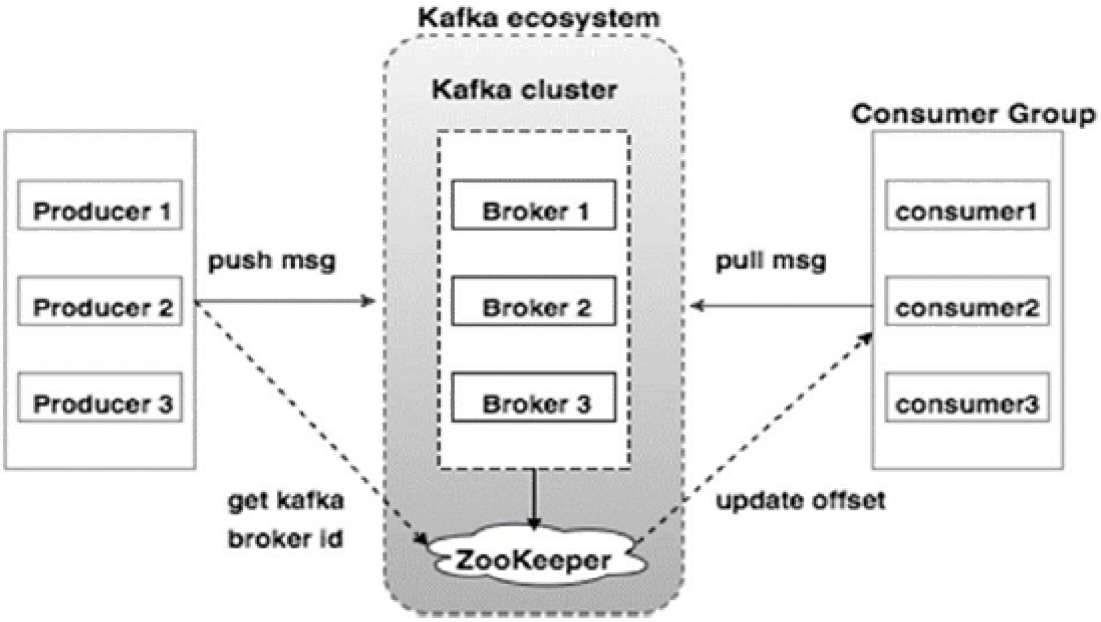
Leader is the node responsible for all reads and writes for the given partition. Every partition has one server acting as a leader.

10 **Follower**

Node which follows leader instructions are called as follower. If the leader fails, one of the follower will automatically become the new leader. A follower acts as normal consumer, pulls messages and up-dates its own data store.

Apache Kafka - Cluster Architecture

Take a look at the following illustration. It shows the cluster diagram of Kafka



The following table describes each of the components shown in the above diagram.

S.No	Components and Description
1	**Broker** Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each bro-ker can handle TB of messages without performance impact. Kafka broker leader election can be done by Zoo-Keeper.
2	**ZooKeeper** ZooKeeper is used for managing and coordinating Kafka broker. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka sys-tem or failure of the broker in the Kafka system. As per the notification received by the Zoo-keeper regarding presence or failure of the broker then pro-ducer and consumer takes decision and starts coordinating their task with some other broker.
3	**Producers** Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn't wait for acknowl-edgements from the broker and sends messages as fast as the broker can handle.
4	**Consumers** Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer acknowledges a parti-cular message offset, it implies that the consumer has consumed all prior messages. The consu-

mer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by ZooKeeper.

Apache Kafka - WorkFlow

As of now, we discussed the core concepts of Kafka. Let us now throw some light on the workflow of Kafka.

Kafka is simply a collection of topics split into one or more partitions. A Kafka partition is a linearly ordered sequence of messages, where each message is identified by their index (called as offset). All the data in a Kafka cluster is the disjointed union of partitions. Incoming messages are written at the end of a partition and messages are sequentially read by consumers. Durability is provided by replicating messages to different brokers.

Kafka provides both pub-sub and queue based messaging system in a fast, reliable, persisted, fault-tolerance and zero downtime manner. **In both cases, producers simply send the message to a topic and consumer can choose any one type of messaging system depending on their need.** Let us follow the steps in the next section to understand how the consumer can choose the messaging system of their choice.

Workflow of Pub-Sub Messaging

Following is the step wise workflow of the Pub-Sub Messaging −

- Producers send message to a topic at regular intervals.
- Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.
- Consumer subscribes to a specific topic.
- Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.
- Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.
- Once Kafka receives the messages from producers, it forwards these messages to the consumers.
- Consumer will receive the message and process it.
- Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.
- Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages.
- This above flow will repeat until the consumer stops the request.
- Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

Workflow of Queue Messaging / Consumer Group

In a queue messaging system instead of a single consumer, a group of consumers having the same "Group ID" will subscribe to a topic. In simple terms, consumers subscribing to a topic with same "Group ID" are considered as a single group and the messages are shared among them. Let us check the actual workflow of this system.

- Producers send message to a topic in a regular interval.
- Kafka stores all messages in the partitions configured for that particular topic similar to the earlier scenario.
- A single consumer subscribes to a specific topic, assume "Topic-01" with "Group ID" as "Group-1".
- Kafka interacts with the consumer in the same way as Pub-Sub Messaging until new consumer subscribes the same topic, "Topic-01" with the same "Group ID" as "Group-1".
- Once the new consumer arrives, Kafka switches its operation to share mode and shares the data between the two consumers. This sharing will go on until the number of con-sumers reach the number of partition configured for that particular topic.
- Once the number of consumer exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing consumer unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition and once all the partitions are assigned to the existing consumers, the new consumers will have to wait.
- This feature is also called as "Consumer Group". In the same way, Kafka will provide the best of both the systems in a very simple and efficient manner.

Role of ZooKeeper

A critical dependency of Apache Kafka is Apache Zookeeper, which is a distributed configuration and synchronization service. Zookeeper serves as the coordination interface between the Kafka brokers and consumers. The Kafka servers share information via a Zookeeper cluster. Kafka stores basic metadata in Zookeeper such as information about topics, brokers, consumer offsets (queue readers) and so on.

Since all the critical information is stored in the Zookeeper and it normally replicates this data across its ensemble, failure of Kafka broker / Zookeeper does not affect the state of the Kafka cluster. Kafka will restore the state, once the Zookeeper restarts. This gives zero downtime for Kafka. The leader election between the Kafka broker is also done by using Zookeeper in the event of leader failure.

Apache Kafka - Installation Steps

Following are the steps for installing Java on your machine.

Step 1 - Verifying Java Installation
Hopefully you have already installed java on your machine right now, so you just verify it using the following command.

$ java -version

If java is successfully installed on your machine, you could see the version of the installed Java.

Step 1.1 - Download JDK
If Java is not downloaded, please download the latest version of JDK by visiting the following link and download latest version.
http://www.oracle.com/technetwork/java/javase/downloads/index.html

Now the latest version is JDK 8u 60 and the file is “jdk-8u60-linux-x64.tar.gz”. Please download the file on your machine.
Step 1.2 - Extract Files

Generally, files being downloaded are stored in the downloads folder, verify it and extract the tar setup using the following commands.

$ cd /go/to/download/path
$ tar -zxf jdk-8u60-linux-x64.gz

Step 1.3 - Move to Opt Directory
To make java available to all users, move the extracted java content to "usr/local/java"/ folder.

$ su
password: (type password of root user)
$ mkdir /opt/jdk
$ mv jdk-1.8.0_60 /opt/jdk/

Step 1.4 - Set path

To set path and JAVA_HOME variables, add the following commands to ~/.bashrc file.

export JAVA_HOME =/usr/jdk/jdk-1.8.0_60
export PATH=$PATH:$JAVA_HOME/bin

Now apply all the changes into current running system.

$ source ~/.bashrc

Step 1.5 - Java Alternatives

Use the following command to change Java Alternatives.

update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_60/bin/java 100

Step 1.6 − Now verify java using verification command (java -version) explained in Step 1.

Step 2 - ZooKeeper Framework Installation

Step 2.1 - Download ZooKeeper

To install ZooKeeper framework on your machine, visit the following link and download the latest version of ZooKeeper.
http://zookeeper.apache.org/releases.html

As of now, latest version of ZooKeeper is 3.4.6 (ZooKeeper-3.4.6.tar.gz).
Step 2.2 - Extract tar file

Extract tar file using the following command

$ cd opt/
$ tar -zxf zookeeper-3.4.6.tar.gz
$ cd zookeeper-3.4.6

$ mkdir data

Step 2.3 - Create Configuration File

Open Configuration File named "conf/zoo.cfg" using the command vi “conf/zoo.cfg” and all the following parameters to set as starting point.

```
$ vi conf/zoo.cfg
tickTime=2000
dataDir=/path/to/zookeeper/data
clientPort=2181
initLimit=5
syncLimit=2
```

Once the configuration file has been saved successfully and return to terminal again, you can start the zookeeper server.
Step 2.4 - Start ZooKeeper Server

```
$ bin/zkServer.sh start
```

After executing this command, you will get a response as shown below −

```
$ JMX enabled by default
$ Using config: /Users/../zookeeper-3.4.6/bin/../conf/zoo.cfg
$ Starting zookeeper ... STARTED
```

Step 2.5 - Start CLI

```
$ bin/zkCli.sh
```

After typing the above command, you will be connected to the zookeeper server and will get the below response.

```
Connecting to localhost:2181
.....
.....
.....
Welcome to ZooKeeper!
.....
.....
WATCHER::
WatchedEvent state:SyncConnected type: None path:null
[zk: localhost:2181(CONNECTED) 0]
```

Step 2.6 - Stop Zookeeper Server

After connecting the server and performing all the operations, you can stop the zookeeper server with the following command −

$ bin/zkServer.sh stop

Now you have successfully installed Java and ZooKeeper on your machine. Let us see the steps to install Apache Kafka.

Step 3 - Apache Kafka Installation

Let us continue with the following steps to install Kafka on your machine.
Step 3.1 - Download Kafka

To install Kafka on your machine, click on the below link −
https://www.apache.org/dyn/closer.cgi?path=/kafka/0.9.0.0/kafka_2.11-0.9.0.0.tgz

Now the latest version i.e., – kafka_2.11_0.9.0.0.tgz will be downloaded onto your machine.
Step 3.2 - Extract the tar file

Extract the tar file using the following command −

$ cd opt/
$ tar -zxf kafka_2.11.0.9.0.0 tar.gz
$ cd kafka_2.11.0.9.0.0

Now you have downloaded the latest version of Kafka on your machine.
Step 3.3 - Start Server

You can start the server by giving the following command −

$ bin/kafka-server-start.sh config/server.properties

After the server starts, you would see the below response on your screen −

$ bin/kafka-server-start.sh config/server.properties
[2016-01-02 15:37:30,410] INFO KafkaConfig values:
request.timeout.ms = 30000
log.roll.hours = 168
inter.broker.protocol.version = 0.9.0.X
log.preallocate = false
security.inter.broker.protocol = PLAINTEXT
…….
…….

Step 4 - Stop the Server

After performing all the operations, you can stop the server using the following command −

$ bin/kafka-server-stop.sh config/server.properties

Now that we have already discussed the Kafka installation, we can learn how to perform basic operations on Kafka in the next chapter.

Apache Kafka - Basic Operations

First let us start implementing "single node-single broker" configuration and we will then migrate our setup to single node-multiple brokers configuration.

Hopefully you would have installed Java, ZooKeeper and Kafka on your machine by now. Before moving to the Kafka Cluster Setup, first you would need to start your ZooKeeper because Kafka Cluster uses ZooKeeper.

Start ZooKeeper

Open a new terminal and type the following command −

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

To start Kafka Broker, type the following command −

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

After starting Kafka Broker, type the command "jps" on ZooKeeper terminal and you would see the following response −

821 QuorumPeerMain
928 Kafka
931 Jps

Now you could see two daemons running on the terminal where QuorumPeerMain is ZooKeeper daemon and another one is Kafka daemon.

Single Node-Single Broker Configuration

In this configuration you have a single ZooKeeper and broker id instance. Following are the steps to configure it −

Creating a Kafka Topic − Kafka provides a command line utility named "kafka-topics.sh" to create topics on the server. Open new terminal and type the below example.

Syntax
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic topic-name

Example
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic Hello-Kafka

We just created a topic named "Hello-Kafka" with a single partition and one replica factor. The above created output will be similar to the following output −

Output − Created topic "Hello-Kafka"

Once the topic has been created, you can get the notification in Kafka broker terminal window and the

log for the created topic specified in “/tmp/kafka-logs/“ in the config/server.properties file.

List of Topics

To get a list of topics in Kafka server, you can use the following command −

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

Output
Hello-Kafka

Since we have created a topic, it will list out "Hello-Kafka" only. Suppose, if you create more than one topics, you will get the topic names in the output.
Start Producer to Send Messages

Syntax
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic topic-name

From the above syntax, two main parameters are required for the producer command line client −

Broker-list − The list of brokers that we want to send the messages to. In this case we only have one broker. The Config/server.properties file contains broker port id, since we know our broker is listening on port 9092, so you can specify it directly.

Topic name − Here is an example for the topic name.

Example
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic Hello-Kafka

The producer will wait on input from stdin and publishes to the Kafka cluster. By default, every new line is published as a new message then the default producer properties are specified in "config/produ-cer.properties" file. Now you can type a few lines of messages in the terminal as shown below.

Output
$ bin/kafka-console-producer.sh --broker-list localhost:9092
--topic Hello-Kafka[2016-01-16 13:50:45,931]
WARN property topic is not valid (kafka.utils.Verifia-bleProperties)
Hello
My first message

My second message

Start Consumer to Receive Messages

Similar to producer, the default consumer properties are specified in "config/consumer.proper-ties" file. Open a new terminal and type the below syntax for consuming messages.

Syntax
bin/kafka-console-consumer.sh --zookeeper localhost:2181 —topic topic-name

--from-beginning

Example
bin/kafka-console-consumer.sh --zookeeper localhost:2181 —topic Hello-Kafka
--from-beginning

Output
Hello
My first message
My second message

Finally, you are able to enter messages from the producer's terminal and see them appearing in the consumer's terminal. As of now, you have a very good understanding on the single node cluster with a single broker. Let us now move on to the multiple brokers configuration.

Single Node-Multiple Brokers Configuration

Before moving on to the multiple brokers cluster setup, first start your ZooKeeper server.

Create Multiple Kafka Brokers − We have one Kafka broker instance already in con-fig/server.properties. Now we need multiple broker instances, so copy the existing server.prop-erties file into two new config files and rename it as server-one.properties and server-two.prop-erties. Then edit both new files and assign the following changes −

config/server-one.properties
# The id of the broker. This must be set to a unique integer for each broker.
broker.id=1
# The port the socket server listens on
port=9093
# A comma seperated list of directories under which to store log files
log.dirs=/tmp/kafka-logs-1

config/server-two.properties
# The id of the broker. This must be set to a unique integer for each broker.
broker.id=2
# The port the socket server listens on
port=9094
# A comma seperated list of directories under which to store log files
log.dirs=/tmp/kafka-logs-2

Start Multiple Brokers− After all the changes have been made on three servers then open three new terminals to start each broker one by one.

Broker1
bin/kafka-server-start.sh config/server.properties
Broker2
bin/kafka-server-start.sh config/server-one.properties
Broker3
bin/kafka-server-start.sh config/server-two.properties

Now we have three different brokers running on the machine. Try it by yourself to check all the daemons by typing jps on the ZooKeeper terminal, then you would see the response.

Creating a Topic

Let us assign the replication factor value as three for this topic because we have three different brokers running. If you have two brokers, then the assigned replica value will be two.

Syntax
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3
-partitions 1 --topic topic-name

Example
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3
-partitions 1 --topic Multibrokerapplication

Output
created topic “Multibrokerapplication”

The "Describe" command is used to check which broker is listening on the current created topic as shown below −
bin/kafka-topics.sh --describe --zookeeper localhost:2181
--topic Multibrokerappli-cation

Output
bin/kafka-topics.sh --describe --zookeeper localhost:2181
--topic Multibrokerappli-cation

Topic:Multibrokerapplication PartitionCount:1
ReplicationFactor:3 Configs:

Topic:Multibrokerapplication Partition:0 Leader:0
Replicas:0,2,1 Isr:0,2,1

From the above output, we can conclude that first line gives a summary of all the partitions, showing topic name, partition count and the replication factor that we have chosen already. In the second line, each node will be the leader for a randomly selected portion of the partitions.

In our case, we see that our first broker (with broker.id 0) is the leader. Then Replicas:0,2,1 means that all the brokers replicate the topic finally "Isr" is the set of "in-sync" replicas. Well, this is the subset of replicas that are currently alive and caught up by the leader.

Start Producer to Send Messages
This procedure remains the same as in the single broker setup.

Example
bin/kafka-console-producer.sh --broker-list localhost:9092
--topic Multibrokerapplication

Output

bin/kafka-console-producer.sh --broker-list localhost:9092 --topic Multibrokerapplication
[2016-01-20 19:27:21,045] WARN Property topic is not valid (kafka.utils.Verifia-bleProperties)
This is single node-multi broker demo
This is the second message

Start Consumer to Receive Messages

This procedure remains the same as shown in the single broker setup.

Example
bin/kafka-console-consumer.sh --zookeeper localhost:2181
—topic Multibrokerapplica-tion --from-beginning

Output
bin/kafka-console-consumer.sh --zookeeper localhost:2181
—topic Multibrokerapplica-tion —from-beginning
This is single node-multi broker demo
This is the second message

Basic Topic Operations

In this chapter we will discuss the various basic topic operations.
Modifying a Topic

As you have already understood how to create a topic in Kafka Cluster. Now let us modify a created topic using the following command

Syntax
bin/kafka-topics.sh —zookeeper localhost:2181 --alter --topic topic_name
--parti-tions count

Example
We have already created a topic “Hello-Kafka” with single partition count and one replica factor.
Now using “alter” command we have changed the partition count.
bin/kafka-topics.sh --zookeeper localhost:2181
--alter --topic Hello-kafka --parti-tions 2

Output
WARNING: If partitions are increased for a topic that has a key,
the partition logic or ordering of the messages will be affected
Adding partitions succeeded!

Deleting a Topic

To delete a topic, you can use the following syntax.

Syntax
bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic topic_name

Example
bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic Hello-kafka

Output
> Topic Hello-kafka marked for deletion

Note −This will have no impact if delete.topic.enable is not set to true

Apache Kafka - Simple Producer Example

Let us create an application for publishing and consuming messages using a Java client. Kafka producer client consists of the following API's.

KafkaProducer API

Let us understand the most important set of Kafka producer API in this section. The central part of the KafkaProducer API is "**KafkaProducer**" class. The KafkaProducer class provides an option to connect a Kafka broker in its constructor with the following methods.

- **KafkaProducer** class provides send method to send messages asynchronously to a topic. The signature of send() is as follows

  producer.send(new ProducerRecord<byte[],byte[]>(topic, partition, key1, value1) , callback);

- ProducerRecord − The producer manages a buffer of records waiting to be sent.

- Callback − A user-supplied callback to execute when the record has been acknowl-edged by the server (null indicates no callback).

- KafkaProducer class provides a flush method to ensure all previously sent messages have been actually completed. Syntax of the flush method is as follows −

  public void flush()

- KafkaProducer class provides partition For method, which helps in getting the partition metadata for a given topic. This can be used for custom partitioning. The signature of this method is as follows −

  public Map metrics()

  It returns the map of internal metrics maintained by the producer.

- public void close() − KafkaProducer class provides close method blocks until all previously sent requests are completed.

Producer API

The central part of the Producer API is "Producer" class. Producer class provides an option to connect Kafka broker in its constructor by the following methods.

The Producer Class

The producer class provides send method to send messages to either single or multiple topics using the following signatures.

public void send(KeyedMessaget<k,v> message)

- sends the data to a single topic,par-titioned by key using either sync or async producer.
public void send(List<KeyedMessage<k,v>>messages)

- sends data to multiple topics.
Properties prop = new Properties();
prop.put(producer.type,"async")
ProducerConfig config = new ProducerConfig(prop);

There are two types of producers – Sync and Async.
The same API configuration applies to "Sync" producer as well. The difference between them is a sync producer sends messages directly, but sends messages in background. Async producer is preferred when you want a higher throughput. In the previous releases like 0.8, an async producer does not have a callback for send() to register error handlers. This is available only in the current release of 0.9.
public void close()

Producer class provides close method to close the producer pool connections to all Kafka bro-kers.

Configuration Settings

The Producer API's main configuration settings are listed in the following table for better under-stand-ing −

S.No	Configuration Settings and Description
1	**client.id** identifies producer application
2	**producer.type** either sync or async
3	**acks** The acks config controls the criteria under producer requests are con-sidered complete.
4	**retries** If producer request fails, then automatically retry with specific value.
5	**bootstrap.servers** bootstrapping list of brokers.
6	**linger.ms** if you want to reduce the number of requests you can set linger.ms to something greater than some value.
7	**key.serializer** Key for the serializer interface.
8	**value.serializer** value for the serializer interface.
9	**batch.size** Buffer size.

10	**buffer.memory** controls the total amount of memory available to the producer for buff-ering.

ProducerRecord API

ProducerRecord is a key/value pair that is sent to Kafka cluster.ProducerRecord class constructor for creating a record with partition, key and value pairs using the following signature.

public ProducerRecord (string topic, int partition, k key, v value)

- Topic − user defined topic name that will appended to record.
- Partition − partition count
- Key − The key that will be included in the record.
- Value − Record contents

public ProducerRecord (string topic, k key, v value)

- ProducerRecord class constructor is used to create a record with key, value pairs and with-out partition.
- Topic − Create a topic to assign record.
- Key − key for the record.
- Value − record contents.

public ProducerRecord (string topic, v value)

ProducerRecord class creates a record without partition and key.

- Topic − create a topic.
- Value − record contents.

The ProducerRecord class methods are listed in the following table −

S.No	Class Methods and Description
1	**public string topic()** Topic will append to the record.
2	**public K key()** Key that will be included in the record. If no such key, null will be re-turned here.
3	**public V value()** Record contents.
4	**partition()** Partition count for the record

SimpleProducer application

Before creating the application, first start ZooKeeper and Kafka broker then create your own topic in Kafka broker using create topic command. After that create a java class named "Sim-pleProducer.java" and type in the following coding.

```
//import util.properties packages
import java.util.Properties;
```

```
//import simple producer packages
import org.apache.kafka.clients.producer.Producer;

//import KafkaProducer packages
import org.apache.kafka.clients.producer.KafkaProducer;

//import ProducerRecord packages
import org.apache.kafka.clients.producer.ProducerRecord;

//Create java class named “SimpleProducer”
public class SimpleProducer {

  public static void main(String[] args) throws Exception{

    // Check arguments length value
    if(args.length == 0){
      System.out.println("Enter topic name”);
      return;
    }

    //Assign topicName to string variable
    String topicName = args[0].toString();

    // create instance for properties to access producer configs
    Properties props = new Properties();

    //Assign localhost id
    props.put("bootstrap.servers", “localhost:9092");

    //Set acknowledgements for producer requests.
    props.put("acks", “all");

    //If the request fails, the producer can automatically retry,
    props.put("retries", 0);

    //Specify buffer size in config
    props.put("batch.size", 16384);

    //Reduce the no of requests less than 0
    props.put("linger.ms", 1);

    //The buffer.memory controls the total amount of memory available to the producer for buffering.
    props.put("buffer.memory", 33554432);

    props.put("key.serializer",
      "org.apache.kafka.common.serializa-tion.StringSerializer");

    props.put("value.serializer",
```

```
        "org.apache.kafka.common.serializa-tion.StringSerializer");

      Producer<String, String> producer = new KafkaProducer
        <String, String>(props);

      for(int i = 0; i < 10; i++)
        producer.send(new ProducerRecord<String, String>(topicName,
          Integer.toString(i), Integer.toString(i)));
            System.out.println("Message sent successfully");
            producer.close();
   }
}
```

Compilation − The application can be compiled using the following command.
javac -cp "/path/to/kafka/kafka_2.11-0.9.0.0/lib/*" *.java

Execution − The application can be executed using the following command.
java -cp "/path/to/kafka/kafka_2.11-0.9.0.0/lib/*":. SimpleProducer <topic-name>

Output

Message sent successfully
To check the above output open new terminal and type Consumer CLI command to receive messages.
>> bin/kafka-console-consumer.sh --zookeeper localhost:2181 —topic <topic-name> —from-begin-ning
1
2
3
4
5
6
7
8
9
10

Simple Consumer Example

As of now we have created a producer to send messages to Kafka cluster. Now let us create a consumer to consume messages form the Kafka cluster. KafkaConsumer API is used to consume messages from the Kafka cluster. KafkaConsumer class constructor is defined below.

public KafkaConsumer(java.util.Map<java.lang.String,java.lang.Object> configs)
configs − Return a map of consumer configs.

KafkaConsumer class has the following significant methods that are listed in the table below.

S.No	Method and Description
1	**public java.util.Set<TopicPar-tition> assignment()** Get the set of partitions currently assigned by the con-sumer.

2 **public string subscription()**
Subscribe to the given list of topics to get dynamically as-signed partitions.

3 **public void sub-scribe(java.util.List<java.lang.String> topics, ConsumerRe-balanceLis-tener listener)**
Subscribe to the given list of topics to get dynamically as-signed partitions.

4 **public void unsubscribe()**
Unsubscribe the topics from the given list of partitions.

5 **public void sub-scribe(java.util.List<java.lang.String> topics)**
Subscribe to the given list of topics to get dynamically as-signed partitions. If the given list of topics is empty, it is treated the same as unsubscribe().

6 **public void sub-scribe(java.util.regex.Pattern pattern, ConsumerRebalanceLis-tener lis-tener)**
The argument pattern refers to the subscribing pattern in the format of regular expression and the listener argument gets notifications from the subscribing pattern.

7 **public void as-sign(java.util.List<TopicParti-tion> partitions)**
Manually assign a list of partitions to the customer.

8 **poll()**
Fetch data for the topics or partitions specified using one of the subscribe/assign APIs. This will return error, if the topics are not subscribed before the polling for data.

9 **public void commitSync()**
Commit offsets returned on the last poll() for all the sub-scribed list of topics and partitions. The same operation is applied to commitAsyn().

10 **public void seek(TopicPartition partition, long offset)**
Fetch the current offset value that consumer will use on the next poll() method.

11 **public void resume()**
Resume the paused partitions.

12 **public void wakeup()**
Wakeup the consumer.

ConsumerRecord API

The ConsumerRecord API is used to receive records from the Kafka cluster. This API consists of a topic name, partition number, from which the record is being received and an offset that points to the record in a Kafka partition. ConsumerRecord class is used to create a consumer record with specific topic name, partition count and <key, value> pairs. It has the following signature.

public ConsumerRecord(string topic,int partition, long offset,K key, V value)

- **Topic** − The topic name for consumer record received from the Kafka cluster.
- **Partition** − Partition for the topic.
- **Key** − The key of the record, if no key exists null will be returned.
- **Value** − Record contents.

ConsumerRecords API

ConsumerRecords API acts as a container for ConsumerRecord. This API is used to keep the list of ConsumerRecord per partition for a particular topic. Its Constructor is defined below.

public ConsumerRecords(java.util.Map<TopicPartition,java.util.List<Consumer-Record>K,V>>> records)

- TopicPartition − Return a map of partition for a particular topic.
- Records − Return list of ConsumerRecord.

ConsumerRecords class has the following methods defined.

S.No	Methods and Description
1	**public int count()** The number of records for all the topics.
2	**public Set partitions()** The set of partitions with data in this record set (if no data was returned then the set is empty).
3	**public Iterator iterator()** Iterator enables you to cycle through a collection, obtaining or re-moving elements.
4	**public List records()** Get list of records for the given partition.

Configuration Settings

The configuration settings for the Consumer client API main configuration settings are listed below −

S.No	Settings and Description
1	**bootstrap.servers** Bootstrapping list of brokers.
2	**group.id** Assigns an individual consumer to a group.
3	**enable.auto.commit** Enable auto commit for offsets if the value is true, otherwise not committed.
4	**auto.commit.interval.ms** Return how often updated consumed offsets are written to ZooKeeper.
5	**session.timeout.ms** Indicates how many milliseconds Kafka will wait for the ZooKeeper to respond to a request (read or write) before giving up and continuing to consume messages.

SimpleConsumer Application

The producer application steps remain the same here. First, start your ZooKeeper and Kafka broker. Then create a "SimpleConsumer" application with the java class named "SimpleCon-sumer.java" and type the following code.

```
import java.util.Properties;
```

```
import java.util.Arrays;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;

public class SimpleConsumer {
  public static void main(String[] args) throws Exception {
    if(args.length == 0){
      System.out.println("Enter topic name");
      return;
    }
    //Kafka consumer configuration settings
    String topicName = args[0].toString();
    Properties props = new Properties();

    props.put("bootstrap.servers", "localhost:9092");
    props.put("group.id", "test");
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("session.timeout.ms", "30000");
    props.put("key.deserializer",
      "org.apache.kafka.common.serializa-tion.StringDeserializer");
    props.put("value.deserializer",
      "org.apache.kafka.common.serializa-tion.StringDeserializer");
    KafkaConsumer<String, String> consumer = new KafkaConsumer
      <String, String>(props);

    //Kafka Consumer subscribes list of topics here.
    consumer.subscribe(Arrays.asList(topicName))

    //print the topic name
    System.out.println("Subscribed to topic " + topicName);
    int i = 0;

    while (true) {
      ConsumerRecords<String, String> records = con-sumer.poll(100);
      for (ConsumerRecord<String, String> record : records)

      // print the offset,key and value for the consumer records.
      System.out.printf("offset = %d, key = %s, value = %s\n",
        record.offset(), record.key(), record.value());
    }
  }
}
```

Compilation − The application can be compiled using the following command.
javac -cp “/path/to/kafka/kafka_2.11-0.9.0.0/lib/*” *.java

Execution − The application can be executed using the following command

java -cp “/path/to/kafka/kafka_2.11-0.9.0.0/lib/*”:. SimpleConsumer <topic-name>

Input − Open the producer CLI and send some messages to the topic. You can put the smple input as ‘Hello Consumer’.

Output − Following will be the output.
Subscribed to topic Hello-Kafka
offset = 3, key = null, value = Hello Consumer

Apache Kafka - Consumer Group Example

Consumer group is a multi-threaded or multi-machine consumption from Kafka topics.

Consumer Group

- Consumers can join a group by using the same"group.id."
- The maximum parallelism of a group is that the number of consumers in the group ← no of partitions.
- Kafka assigns the partitions of a topic to the consumer in a group, so that each partition is consumed by exactly one consumer in the group.
- Kafka guarantees that a message is only ever read by a single consumer in the group.
- Consumers can see the message in the order they were stored in the log.

Re-balancing of a Consumer

Adding more processes/threads will cause Kafka to re-balance. If any consumer or broker fails to send heartbeat to ZooKeeper, then it can be re-configured via the Kafka cluster. During this re-balance, Kafka will assign available partitions to the available threads, possibly moving a partition to another process.

```
import java.util.Properties;
import java.util.Arrays;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;

public class ConsumerGroup {
  public static void main(String[] args) throws Exception {
    if(args.length < 2){
      System.out.println("Usage: consumer <topic> <groupname>");
      return;
    }

    String topic = args[0].toString();
    String group = args[1].toString();
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092");
    props.put("group.id", group);
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("session.timeout.ms", "30000");
    props.put("key.deserializer",
```

```
      "org.apache.kafka.common.serializa-tion.StringDeserializer");
    props.put("value.deserializer",
      "org.apache.kafka.common.serializa-tion.StringDeserializer");
    KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);

    consumer.subscribe(Arrays.asList(topic));
    System.out.println("Subscribed to topic " + topic);
    int i = 0;

    while (true) {
      ConsumerRecords<String, String> records = con-sumer.poll(100);
        for (ConsumerRecord<String, String> record : records)
          System.out.printf("offset = %d, key = %s, value = %s\n",
          record.offset(), record.key(), record.value());
    }
  }
}
```

Compilation
javac -cp “/path/to/kafka/kafka_2.11-0.9.0.0/libs/*" ConsumerGroup.java

Execution
>>java -cp “/path/to/kafka/kafka_2.11-0.9.0.0/libs/*":.
ConsumerGroup <topic-name> my-group
>>java -cp "/home/bala/Workspace/kafka/kafka_2.11-0.9.0.0/libs/*":.
ConsumerGroup <topic-name> my-group

Here we have created a sample group name as "my-group" with two consumers. Similarly, you can create your group and number of consumers in the group.
Input

Open producer CLI and send some messages like −
Test consumer group 01
Test consumer group 02

Output of the First Process
Subscribed to topic Hello-kafka
offset = 3, key = null, value = Test consumer group 01

Output of the Second Process
Subscribed to topic Hello-kafka
offset = 3, key = null, value = Test consumer group 02

Now hopefully you would have understood SimpleConsumer and ConsumeGroup by using the Java client demo. Now you have an idea about how to send and receive messages using a Java client. Let us continue Kafka integration with big data technologies in the next chapter.

Apache Kafka - Integration With Storm

Kafka Tool packaged under “org.apache.kafka.tools.*. Tools are categorized into system tools and replication tools.

System Tools

System tools can be run from the command line using the run class script. The syntax is as follows −

bin/kafka-run-class.sh package.class - - options

Some of the system tools are mentioned below −

- **Kafka Migration Tool** − This tool is used to migrate a broker from one version to an-other.
- **Mirror Maker** − This tool is used to provide mirroring of one Kafka cluster to another.
- **Consumer Offset Checker** − This tool displays Consumer Group, Topic, Partitions, Off-set, log-Size, Owner for the specified set of Topics and Consumer Group.

Replication Tool

Kafka replication is a high level design tool. The purpose of adding replication tool is for stronger durability and higher availability. Some of the replication tools are mentioned below −

- **Create Topic Tool** − This creates a topic with a default number of partitions, replication factor and uses Kafka's default scheme to do replica assignment.
- **List Topic Tool** − This tool lists the information for a given list of topics. If no topics are provided in the command line, the tool queries Zookeeper to get all the topics and lists the information for them. The fields that the tool displays are topic name, partition, leader, replicas, isr.
- **Add Partition Tool** − Creation of a topic, the number of partitions for topic has to be specified. Later on, more partitions may be needed for the topic, when the volume of the topic will increase. This tool helps to add more partitions for a specific topic and also allows manual replica assignment of the added partitions.

Apache Kafka - Applications

Kafka supports many of today's best industrial applications. We will provide a very brief overview of some of the most notable applications of Kafka in this chapter.

Twitter

Twitter is an online social networking service that provides a platform to send and receive user tweets. Registered users can read and post tweets, but unregistered users can only read tweets. Twitter uses Storm-Kafka as a part of their stream processing infrastructure.

LinkedIn

Apache Kafka is used at LinkedIn for activity stream data and operational metrics. Kafka mes-saging system helps LinkedIn with various products like LinkedIn Newsfeed, LinkedIn Today for online message consumption and in addition to offline analytics systems like Hadoop. Kafka’s strong durability is also one of the key factors in connection with LinkedIn.

Netflix

Netflix is an American multinational provider of on-demand Internet streaming media. Netflix uses Kafka for real-time monitoring and event processing.

Mozilla
Mozilla is a free-software community, created in 1998 by members of Netscape. Kafka will soon be replacing a part of Mozilla current production system to collect performance and usage data from the end-user's browser for projects like Telemetry, Test Pilot, etc.

Oracle
Oracle provides native connectivity to Kafka from its Enterprise Service Bus product called OSB (Oracle Service Bus) which allows developers to leverage OSB built-in mediation capabilities to implement staged data pipelines.