HOME
INTRODUCTION
QUICKSTART
USE CASES
DOCUMENTATION
PERFORMANCE
POWERED BY
PROJECT INFO
ECOSYSTEM
CLIENTS
EVENTS
CONTACT US
APACHE

Quickstart

This tutorial assumes you are starting fresh and have no existing Kafka or ZooKeeper data. Since Kafka console scripts are different for Unix-ba and Windows platforms, on Windows platforms use `bin\windows\` instead of `bin/`, and change the script extension to `.bat`.

Step 1: Download the code

Download the 2.5.0 release and un-tar it.

```
> tar -xzf kafka_2.12-2.5.0.tgz
> cd kafka_2.12-2.5.0
```

Step 2: Start the server

Kafka uses ZooKeeper so you need to first start a ZooKeeper server if you don't already have one. You can use the convenience script packaged with kafka to get a quick-and-dirty single-node ZooKeeper instance.

```
> bin/zookeeper-server-start.sh config/zookeeper.properties
[2013-04-22 15:01:37,495] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper
...
```

Now start the Kafka server:

```
> bin/kafka-server-start.sh config/server.properties
[2013-04-22 15:01:47,028] INFO Verifying properties (kafka.utils.VerifiableProperties)
[2013-04-22 15:01:47,051] INFO Property socket.send.buffer.bytes is overridden to 1048576 (kafka.utils.Verif
...
```

Step 3: Create a topic

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

```
> bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --top
```

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

```
> bin/kafka-topics.sh --list --bootstrap-server localhost:9092
test
```

Alternatively, instead of manually creating topics you can also configure your brokers to auto-create topics when a non-existent topic is publishe to.

Step 4: Send some messages

Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. default, each line will be sent as a separate message.

Run the producer and then type a few messages into the console to send to the server.

```
> bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic test
This is a message
This is another message
```

Step 5: Start a consumer

Kafka also has a command line consumer that will dump out messages to standard output.

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
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 termin and see them appear in the consumer terminal.

All of the command line tools have additional options; running the command with no arguments will display usage information documenting the in more detail.

Step 6: Setting up a multi-broker cluster

So far we have been running against a single broker, but that's no fun. 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 (on Windows use the `copy` command instead):

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

Now edit these new files and set the following properties:

```
config/server-1.properties:
    broker.id=1
    listeners=PLAINTEXT://:9093
    log.dirs=/tmp/kafka-logs-1

config/server-2.properties:
    broker.id=2
    listeners=PLAINTEXT://:9094
    log.dirs=/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's data.

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

```
> bin/kafka-server-start.sh config/server-1.properties &
...
> bin/kafka-server-start.sh config/server-2.properties &
...
```

Now create a new topic with a replication factor of three:

```
> bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 3 --partitions 1 --top
```

Okay but now that we have a cluster how can we know which broker is doing what? To see that run the "describe topics" command:

```
> bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-replicated-topic
Topic:my-replicated-topic   PartitionCount:1    ReplicationFactor:3 Configs:
    Topic: my-replicated-topic  Partition: 0    Leader: 1   Replicas: 1,2,0 Isr: 1,2,0
```

Here is an explanation of output. The first line gives a summary of all the partitions, each additional line gives information about one partition. S we have only one partition for this topic there is only one line.

- "leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion o partitions.
- "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive
- "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

Note that in my example node 1 is the leader for the only partition of the topic.

We can run the same command on the original topic we created to see where it is:

```
> bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic test
Topic:test  PartitionCount:1    ReplicationFactor:1 Configs:
    Topic: test Partition: 0    Leader: 0   Replicas: 0 Isr: 0
```

So there is no surprise there—the original topic has no replicas and is on server 0, the only server in our cluster when we created it.

Let's publish a few messages to our new topic:

```
> bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-replicated-topic
...
my test message 1
my test message 2
^C
```

Now let's consume these messages:

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-top
...
my test message 1
my test message 2
```

```
^C
```

Now let's test out fault-tolerance. Broker 1 was acting as the leader so let's kill it:

```
> ps aux | grep server-1.properties
7564 ttys002    0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/bin/java...
> kill -9 7564
```

On Windows use:

```
> wmic process where "caption = 'java.exe' and commandline like '%server-1.properties%'" get processid
ProcessId
6016
> taskkill /pid 6016 /f
```

Leadership has switched to one of the followers and node 1 is no longer in the in-sync replica set:

```
> bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-replicated-topic
Topic:my-replicated-topic   PartitionCount:1    ReplicationFactor:3 Configs:
    Topic: my-replicated-topic  Partition: 0    Leader: 2   Replicas: 1,2,0 Isr: 2,0
```

But the messages are still available for consumption even though the leader that took the writes originally is down:

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-top
...
my test message 1
my test message 2
^C
```

Step 7: Use Kafka Connect to import/export data

Reading data from the console and writing it back to the console is a convenient place to start, but you'll probably want to use data from other sources or export data from Kafka to other systems. For many systems, instead of writing custom integration code you can use Kafka Connect import or export data.

Kafka Connect is a tool included with Kafka that imports and exports data to Kafka. It is an extensible tool that runs *connectors*, which impleme the custom logic for interacting with an external system. In this quickstart we'll see how to run Kafka Connect with simple connectors that impo data from a file to a Kafka topic and export data from a Kafka topic to a file.

First, we'll start by creating some seed data to test with:

```
> echo -e "foo\nbar" > test.txt
```

Or on Windows:

```
> echo foo> test.txt
> echo bar>> test.txt
```

Next, we'll start two connectors running in *standalone* mode, which means they run in a single, local, dedicated process. We provide three configuration files as parameters. The first is always the configuration for the Kafka Connect process, containing common configuration such a the Kafka brokers to connect to and the serialization format for data. The remaining configuration files each specify a connector to create. Thes files include a unique connector name, the connector class to instantiate, and any other configuration required by the connector.

```
> bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-source.properties confi
```

These sample configuration files, included with Kafka, use the default local cluster configuration you started earlier and create two connectors: first is a source connector that reads lines from an input file and produces each to a Kafka topic and the second is a sink connector that reads messages from a Kafka topic and produces each as a line in an output file.

During startup you'll see a number of log messages, including some indicating that the connectors are being instantiated. Once the Kafka Conne process has started, the source connector should start reading lines from `test.txt` and producing them to the topic `connect-test`, and sink connector should start reading messages from the topic `connect-test` and write them to the file `test.sink.txt`. We can verify th data has been delivered through the entire pipeline by examining the contents of the output file:

```
> more test.sink.txt
foo
bar
```

Note that the data is being stored in the Kafka topic `connect-test`, so we can also run a console consumer to see the data in the topic (or u custom consumer code to process it):

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test --from-beginning
{"schema":{"type":"string","optional":false},"payload":"foo"}
{"schema":{"type":"string","optional":false},"payload":"bar"}
...
```

The connectors continue to process data, so we can add data to the file and see it move through the pipeline:

```
> echo Another line>> test.txt
```

You should see the line appear in the console consumer output and in the sink file.

Step 8: Use Kafka Streams to process data

Kafka Streams is a client library for building mission-critical real-time applications and microservices, where the input and/or output data is stor
in Kafka clusters. Kafka Streams combines the simplicity of writing and deploying standard Java and Scala applications on the client side with t
benefits of Kafka's server-side cluster technology to make these applications highly scalable, elastic, fault-tolerant, distributed, and much more.
quickstart example will demonstrate how to run a streaming application coded in this library.