Introduction to Kafka and Spark Streaming

Master M2 – Université Grenoble Alpes & Grenoble INP

2023

This lab is an introduction to Kafka and Spark Streaming. The lab assumes that you run on a Linux machine similar to the ones available in the lab rooms of Ensimag. Some information about accessing remotly to the machines of Ensimag is provided in appendix.

1 Apache Kafka

In this first part, we are going to execute basic operations with the Kafka message broker to understand how it works.

The documentation on Kafka is available here: `https://kafka.apache.org/`. It can be a good idea to refresh your mind about the description that was made of Kafka during the lectures by having a look at the corresponding slides.

Note that this first part of the lab is strongly inspired from the Kafka documentation. You can refer to it when some explanations are unclear: `https://kafka.apache.org/quickstart`

1.1 Running a Kafka broker

Installation The first step to use Kafka is to download the archive including all binary files[1] and extracting this archive:

```
> wget https://archive.apache.org/dist/kafka/3.6.0/kafka_2.12-3.6.0.tgz
> tar zxvf kafka_2.12-3.6.0.tgz
```

Note that on some systems, simply using the tar command to extract the files from the archive may not work. In this case, one can use the following commands instead:

```
> gunzip -dc kafka_2.12-3.6.0.tgz | tar xf -
```

At this point, Kafka is ready to be used.

[1]The version 2.7.0 of Kafka is not the most version but we recommend it for this lab, as all proposed exercises have been heavily tested with this version.

Starting a Zookeeper service Kafka relies on Zookeeper to reliably store information about the configuration of a Kafka Cluster, about the messages that have been delivered to clients, etc.

In a realistic setup, the Zookeeper service should be run on multiple nodes, that are not the ones where Kafka is going to execute. However, in this lab, we are going to simply make some tests with a local deployment. As such, we are going to run a single-node Zookeeper instance, as follows:

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

Starting the Kafka broker In the first step of the lab, we are going to work with a single Kafka broker. To launch the broker, run in a new terminal:

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

1.2 First operations with Kafka

Messages in Kafka are published in a topic. The following command can be used to obtain the list of topics that already exist in a Kafka cluster:

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

At the beginning, you should observe that no topics exist. Hence we are going to create one (replace `[MY_TOPIC_NAME]` with the name you like):

```
> bin/kafka-topics.sh --create --bootstrap-server localhost:9092 \
                      --replication-factor 1 --partitions 1 \
                      --topic [MY_TOPIC_NAME]
```

You can then verify that this new topic exists.

It is now time to publish the first messages in the topic. To publish messages, we are going to use the Kafka client console. To start the console, run:

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 \
                                --topic [MY_TOPIC_NAME]
```

From this point on, you can start publishing messages on the topic by entering messages in the console. However, the setup is not yet very interesting as there are no processes reading the messages published on the topic.

To read the messages published on the topic, we are going to start a consumer console:

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
                                --topic [MY_TOPIC_NAME] --from-beginning
```

Note that since we used the flag `-from-beginning`, the consumer receives all the messages published since the topic has been created. To create a client that only receives messages published after it connected, simply remove this flag.

1.3 Understanding the semantic of operations with Kafka

We list below a set of questions that you should try to answer by running new tests with Kafka.

- What happens if the Kafka server crashes and needs to be restarted? Is it possible for clients that connect after the server has restarted to retrieve the messages that were published before the crash? (Try to explain)
  - To simulate a crash of the server, we are simply going to send a signal to the server process by typing `CTRL+C` in the corresponding terminal (i.e., sending a `SIGINT` signal to the process)[2].
- What happens if multiple consumers are registered to the same topic? When a message is published, which consumer receives the messages?
- What happens if multiple producers produce messages on the same topic? What is received by the consumers?
- During the lecture on Kafka, we introduced the notion of consumer group. To associate a consumer to a specific consumer group, the following comment should be used:

  ```
  > bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
                                  --topic [MY_TOPIC_NAME] \
                                  --from-beginning \
                                  --group [MY_GROUP_NAME]
  ```

  What happens on message publication when two consumers belonging to the same group are registered to the same topic? (try to explain)

  To run this test, it is better to create a new topic.
- During the lecture on Kafka, we also introduced the notion of `partitions`. In the first topic we created, there was only one partition. Create a new topic that is divided into 2 partitions[3].

  Assuming that the new topic you created is called `[MY_PARTITIONED_TOPIC]`, you can verify that your new topic includes two partitions using the following command:

  ```
  > bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 \
                        --topic [MY_PARTITIONED_TOPIC]
  ```

  What happens when new messages are published on this topic if 2 consumers belonging to the same group are registered to this topic? (try to explain)
- Same question as before but this time with 3 clients?

[2] Students familiar with UNIX systems can try obtaining the PID of the server process using `ps aux | grep server.properties`, and can then use `kill -9` to kill the corresponding process (i.e., sending a `SIGINT` signal to the process)

[3] Only the value of the `-partitions` option needs to be changed in the command to create a topic

1.4 Setting up a Kafka cluster

Until now we have worked with a single Kafka broker. In practice, a Kafka cluster composed of multiple Kafka brokers is deployed both to improve the performance and the reliability of the system. We are going to create our own Kafka cluster composed of 3 brokers on a single machine.

Cleaning To avoid clutter, we suggest to start this new step from a clean environment. To this end:

- Stop the existing Kafka broker by sending a `SIGINT` signal to the process.
- Stop the Zookeeper service by sending a `SIGINT` signal to the process. (note that it is important to stop the Kafka broker first)
- Delete the logs of Zookeeper and Kafka using the following command:

```
> rm -rf /tmp/zookeeper/* /tmp/kafka-logs/*
```

Starting a Kafka cluster To create a Kafka cluster, we need to start multiple brokers, each with a different broker identifier. To this end, we should create 3 new configurations files, on the same model as the one we used for the first broker (file `config/server.properties`).

More specifically, you need to create 3 new files that are copies of the file `config/server.properties`. In each new file, 3 configuration options need to be updated (`broker.id`, `listeners`, `log.dirs`) so that the values are different in each file. Here is the summary of the modifications for file file `config/server-1.properties`:

- `broker.id=1`
- `listeners=PLAINTEXT://localhost:9092` (only the port number should be changed in the other files, for instance to `9093` and `9094`).
- `log.dirs=/tmp/kafka-logs1`

Once the new configuration files are created, you can:

- Restart the Zookeeper service using the instructions presented in Section 1.1.
- Start the three brokers in new terminals:

```
> bin/kafka-server-start.sh config/server-1.properties
> bin/kafka-server-start.sh config/server-2.properties
> bin/kafka-server-start.sh config/server-3.properties
```

Creating a replicated topic Once the Kafka cluster is created, new topics can be created, now with a replication degree of 3:

```
> bin/kafka-topics.sh --create --bootstrap-server localhost:9092 \
                      --replication-factor 3 --partitions 1 \
                      --topic [MY_REPLICATED_TOPIC]
```

You can observe the state of the new topic using the same command as introduced earlier:

```
> bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 \
                      --topic [MY_REPLICATED_TOPIC]
```

Some questions related to the Kafka cluster

- Explain the output of the command that describes the replicated topic
- Can a producer connect to any broker to publish a message on the topic? (To change the broker to connect to, simply change the port number in the url of the `-bootstrap-server` option)
- Create 3 consumers, each connected to a different broker. What happens when a message is published on the topic?
- Kill the broker that plays the role of leader for the replicated topic.
  - Can we still publish and consume messages on the topic?
  - Observe the new state of the replicated topic and explain what happened.
- Restart the killed broker and observe the state of the replicated topic. Explain.
- Create a new topic with a replication degree of 3 and 2 partitions. Register 3 clients belonging to the same consumer group to this topic.
  - Observe the state of the new replicated topic and explain the difference with the previous case.
  - What happens when a producer publishes messages on the topic?

2 Spark Streaming

important notice: the following sections give instructions for older versions of the Spark streaming and Kafka frameworks. You will need to reingeneer the lab to adapt it to the current versions or the versions you have installed. Another possibility is to install the old versions.

In this part, we are going to execute our first Spark Streaming application. The code of the application is provided to you in the archive published here: `https://tropars.github.io/teaching/#data-management-in-large-scale-distributed-systems`.

The provided Spark Streaming application is a simple version of the famous Word Count application adapted to the streaming context[4]. The application counts the occurrences of words inside each micro-batch.

Installing Spark This lab is based on a recent stable version of Spark (3.0.1)[5] and works with Scala 2.12. Furthermore the `sbt` built manager is used for compilation.

If Spark v3.0.1 is not available on your machine, you should download and *install* it using the following commands in your working directory:

```
> wget \
  https://archive.apache.org/dist/spark/spark-3.0.1/spark-3.0.1-bin-hadoop2.7.tgz
> tar zxvf spark-3.0.1-bin-hadoop2.7.tgz
```

Compiling the Spark Streaming application To compile the Spark Streaming application, move into the directory `code/stream_app` of the provided material and run:

```
> sbt package
```

Take the time to read and understand the provided source code of the application, that is accessible in the file `code/stream_app/src/main/scala/NetworkWordCount.scala`.

Running the application To test the application, we are going to use the network utility `Netcat`, that is going to allow us to send messages to the Spark Streaming application through the console. To start the `Netcat` server, and make it listen for new connections on port 9999, run in a dedicated terminal:

```
> nc -lk 9999
```

To start the Spark Streaming application, run:

```
> [SPARK_DIRECTORY]/bin/spark-submit --master local[*] \
    ./target/scala-2.12/stream-word-count_2.12-1.0.jar localhost 9999
```

Some observations To easily observe what is happening in the Spark Streaming application, we suggest you to modify the application to increase the batch interval to 10 seconds.

After restarting your application, you can observe its state by connecting to the graphical user interface of Spark at the following url: `http://localhost:4040`. We are more specifically interested in the `Streaming` page.

You can observe that the Streaming statistics of Spark report 4 main metrics (Input rate, Scheduling delay, Processing time, Total delay):

[4]The original version of the code can be found in the Spark repository: `https://github.com/apache/spark/tree/branch-2.4/examples/src/main/scala/org/apache/spark/examples/streaming`. It corresponds to the tutorial example given in the documentation of Spark Streaming: `https://spark.apache.org/docs/latest/streaming-programming-guide.html#a-quick-example`

[5]It seems that the lab also works with the version of Spark used for the previous labs (3.3.0).

- Observe how these metrics evolve depending on the input data.
- What information does each of these metrics provide us?

3 Kafka and Spark Streaming

Now that we have managed to have a first streaming application running, we are going to build a Spark Streaming application that reads data from a Kafka broker.

Running the provided application We provide you with another version of the streaming Word Count application, that is available in the directory `code/kafka_stream_app`:

- Compile the new application using `sbt`
- Start a Kafka broker and create a new topic
- Start the new Spark application using the following command:

```
> [SPARK_DIRECTORY]/bin/spark-submit \
    --packages org.apache.spark:spark-streaming-kafka-0-10_2.12:3.0.1 \
    --master local[*] ./target/scala-2.12/kafka-stream-app_2.12-1.0.jar \
    localhost:9092 [CONSUMER_GROUP] [KAFKA_TOPIC]
```

Modifying the Word Count application: displaying trending words As a last step for this lab, we suggest you to modify the Word Count application to create an application that displays the most frequent words over hopping windows. To this end, you need to:

1. Aggregate the data received from the Kafka broker in overlapping windows. We want to create a new 40-second window every 10 seconds.
2. Sort the words appearing in each window in descending order according to their number of occurrence.

Most information regarding the mechanisms required to implement this application are available on this web page: `https://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams`.