ATU Distributed Systems

Lab: Fault Tolerant and Scalable Kafka

Instructions for the Distributed Systems lab **Fault Tolerant and Scalable Kafka**.

Lab Objectives

In this lab you'll:

- Configure and run a Kafka cluster with multiple brokers
- Create a partitioned and replicated topic
- Test the fault-tolerance of the multi-broker cluster

Introduction

In the last lab you set up a simple Kafka cluster consisting of a single Kafka broker. While this was useful as an introduction to Kafka, it's not a very realistic setup, because a single broker Kafka cluster has a single point of failure: if that broker goes down then no data can flow through Kafka, amd the whole system you've built around Kafka does down with it.

In this lab we'll set up a more realistic scenario. We'll expand the cluster by adding more brokers and see that, not only does this prevent our cluster from having a single point of failure, it also allows us to spread the workload across the brokers.

Getting Started

1. Log in to your Azure Lab Services VM.
2. In the VM, open a terminal and navigate to the Kafka install directory.

All subsequent commands and instructions assume you're in the folder
`/home/comp08011/dev/kafka_2.13`

Restarting the Single-Broker Kafka Cluster

First off, let's get back to where we were at the end of the last lab, with a single broker Kafka cluster up and running, and a command-line producer and consumer running. Every time you start up the cluster after your

VM restarts, you'll need to do 2 things:

1. Start Zookeeper (the one bundles with Kafka)
2. Start (at least one) Kafka broker If you're using the command-line consumer and producer (which we are here) then you'll need to start those too. (Use a new terminal tab for each of these operations. You can rename terminal tabs by right-clicking on them, it could be useful in helping to keep track of what's running in each tab).

- Start Kafka's bundled Zookeeper by running the `zookeeper-server-start.sh` script, passing in the zookeeper config file as a command-line argument

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

- Open a new terminal tab (`CTRL-Shift-T`) and start a Kafka broker by running the broker start-up script and passing in the config file we set up last week:

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

- Open a new terminal tab and run `kafka-console-producer.sh`, setting `--bootstrap-server` to the address of our Kafka broker (`localhost:9092`), and `--topic` to the topic we created last week, `atu`:

```
./bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic atu
```

- Run the following command to consume messages from the `atu` topic (it might be useful to split the terminal tab here, right click and select `Split Horizontally`):

```
./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic atu
```

- Publish some messages from the producer and verify that they're received at the consumer.

Tip: Labelling Terminal Tabs

We're going to have several terminal tabs open by the end of this lab, and it'll be easy to lose track of what's running where. To avoid this, let's label our tabs before the tab count gets out of hand.

Double click on the name of the first tab (where Zookeeper is running) and change its name from `comp08011@lab000001: ~/dev/kafka_2.13` to `zookeeper`. Do the same for the Kafka broker that's running, naming it `broker 0`. Call the producer and consumer tab whatever you like (`test` maybe?).

As we start more brokers label the terminal tabs like this to make it easier to navigate the setup.

Exploring the Fault Tolerance of a Single-Broker Cluster

Now that the simple cluster is up and running let's explore its ability to tolerate faults.

- Kill the Kafka broker (`broker 0`) with `CTRL-C`
- Note the Kafka console producer and consumers logging `WARN` messages about `Connection could not be established`, they've detected that the Kafka broker has done down.
- Try to send messages using the command-line producer
    - You should find that it's not possible since the connection between the producer and the Kafka cluster has been lost

We can see that this simple cluster isn't fault-tolerant: the failure of a single broker causes the whole cluster to stop operating.

Scaling the Cluster

Configuring Additional Brokers

To make our cluster capable of withstanding failures, we'll need to add more brokers (we'll start 2 more). When one broker goes down, the others will keep the cluster running. We'll need to configure these brokers in a similar way to how we configured our original broker.

- Restart the broker that you killed in the last step.
- Verify that messages can be sent from the producer and received by the consumer
- Our new brokers will need somewhere to store their logs, so we'll need to create two new logs folders for them.

```
mkdir logs-1
mkdir logs-2
```

- On the command-line go to the `config` folder in the Kafka install directory.
- Create two new configuration files for our two new brokers by copying the first broker's config:

```
$ cp server.properties server-1.properties
$ cp server.properties server-2.properties
```

- Open `server-1.properties` in a text editor (e.g. gedit, run `gedit server-1.properties`)
- Each broker will need a unique ID, so set `broker.id=1`
- Since we're testing our cluster by running all the brokers on a single node (our VM), we'll need to give them unique TCP port numbers to listen on:
    - uncomment the line starting with `listeners=`, and set the port number to 9093, i.e.. `listeners=PLAINTEXT://:9093`
- Set `log.dirs` to point to the first of the two new logs folders you created, i.e. `log.dirs=/home/comp08011/dev/kafka-2.5.0/logs-1`
- Configure the second broker by opening `server-2.properties` in the text editor. You'll update the same fields, but giving them unique values, as follows:

```
broker.id=2
listeners=PLAINTEXT://:9094
log.dirs=/home/comp08011/dev/kafka-2.5.0/logs-2
```

Running Additional Brokers

- Open a new terminal tab or pane (you can make a new pane in the current tab by splitting it)
- Start a new Kafka broker, passing in one of the new config files we just set up:

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

- In another new terminal tab/pane start another new brokerusing the other config file:

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

In the logs output you should see the Kafka brokers running and outputting their ids:

```
[2021-11-14 21:57:10,822] INFO [KafkaServer id=1] started
(kafka.server.KafkaServer)

[2021-11-14 21:57:46,800] INFO [KafkaServer id=2] started
(kafka.server.KafkaServer)
```

- **Don't forget to rename these terminal panes to help you keep track of which broker is running where.**

You should now have a Kafka cluster running with 3 brokers.

Creating a Fault-Tolerant Topic

Now that we have a more robust cluster running, we can set up a new topic that makes use of the multiple brokers. This topic will be partitioned and replicated:

Partitioned: messages in the topic will be divided across multiple brokers for processing

Replicated: messages in each broker will be copied to other brokers as a backup

- Create a new topic called `purchases` using the `--create` command on the `kafka-topics` script. We can set up this topic to be partitioned across all 3 brokers and replicated to all 3 brokers using the options `--partitions` and `--replication-factor`:

```
./bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --replication-
factor 3 --partitions 3 --topic purchases
```

Note that we don't have to provide the addresses of all the servers in the cluster, just a single server (`--bootstrap-server`). This server will let clients know about all the brokers in the cluster.

- Use the `kafka-topics` script's `--list` command to verify that the topic was created:

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

- Verify that connection to any broker in the cluster can connect us to all other brokers by re-running the `--list` command but using each of our brokers as the bootstrap server by changing the value of `--bootstrap-server`:

```
./bin/kafka-topics.sh --bootstrap-server localhost:9093 --list
./bin/kafka-topics.sh --bootstrap-server localhost:9094 --list
```

- `kafka-topics` also provides a `--describe` command which can give us useful details on how a topic is configured. First, run this command on the topic created last week (`atu`):

```
./bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic atu
```

This simple topic isn't partitioned or replicated, it only has one partition which is managed entirely by the single broker we are running. Note how this is reflected in the simple output of `--describe`:

```
comp08011@lab000001:~/dev/kafka_2.13$ ./bin/kafka-topics.sh --bootstrap-server
localhost:9092 --describe --topic atu
Topic: atu  TopicId: xhApybWOQ1-t6vszEcz_zw PartitionCount: 1   ReplicationFactor:
1   Configs:
    Topic: atu  Partition: 0    Leader: 1   Replicas: 1 Isr: 1
```

- Now check the configuration of the `purchases` topic using `--describe`:

```
./bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic
purchases
```

The outout here is much more complex, and should look something like this:

```
./bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic
purchases
Topic: purchases        PartitionCount: 3       ReplicationFactor: 3    Configs:
segment.bytes=1073741824
        Topic: purchases        Partition: 0    Leader: 0       Replicas: 0,1,2
```

```
Isr: 0,1,2
        Topic: purchases        Partition: 1    Leader: 2       Replicas: 2,0,1
Isr: 2,0,1
        Topic: purchases        Partition: 2    Leader: 1       Replicas: 1,2,0
Isr: 1,2,0
```

There's a number of important things here to note:

- At the topic, `PartitionCount` and `ReplicationFactor` give an overall view of how the topic is partitioned and replicated
- Each subsequent line gives details for an individual partition, identified by the `Partition` number
- For example, in the output above (yours may be slightly different), Kafka has set up partition 0 as follows:
    - The broker with `id=0` is the `Leader` for this partition. This means that all writes and reads to/from this partition go through broker 0.
    - There are three brokers which are `Replicas` for this partition (brokers 0, 1 and 2). This means that all the messages written to the partition via broker 0 will be replicated to brokers 1 and 2.
    - `Isr:` stands for in-sync replicas. These are replicas which are currently synchronised with the leader and have a full copy of the data.

Test New Topic

Let's make sure that this topic is set up ok by verifying that we can publish messages to and consume messages from it.

- Stop the console producer and consumer scripts
- Restart them, this time configuring them to point to the `purchases` topic
- Use the producer to send 5 messages, e.g. `purchase1, purchase2..` etc.
- Verify that the messages are received at the consumer Note an important difference here. When we had a single broker (with a single topic partition), messages were received in the other that they were sent. Now that the topic has been partitioned we've lost this global ordering. Messages will be ordered **within** a partition, but not **across** partitions. We've sacrifice global ordering for scalablity.

Testing Fault-Tolerance

Now let's see how fault-tolerant our new cluster is.

- Shutdown the broker with id=1
- Run the `kafka-topics` script's `--describe` command again, and compare its output with the output you got the last time you ran `--describe`.
    - You should see that the 3 topic partitions still exist, but they are now divided across only 2 brokers
    - Note that the partition for which broker 1 was the leader now has a new leader
- Try sending messages again from the producer. They should be received at the consumer as before. Shutting down one of the brokers hasn't impacted the ability of the cluster to process data.
- Kill the consumer and start it again, this time using the `--from-beginning` option (this will try to read all messages sent through the cluster from the start).

- You should see all messages previously sent arrive at the consumer. One of the brokers "failing" hasn't had any impact on the persistence of messages in the cluster.

Conclusion

In this lab you've seen how to set up a more realistic multi-broker Kafka cluster, and explored how to create partitioned and replicated Kafka topics. You've seen that this cluster set up is fault-tolerant (thanks to replication), and scalable (thanks to partitioning).