Setting up a simple Kafka cluster with docker for testing

February 02, 2020 3 minute read

In this short article we'll have a quick look at how to set up a Kafka cluster locally, which can be easily accessed from outside of the docker container. The reason for this article is that most of the example you can find either provide a single Kafka instance, or provide a way to set up a Kafka cluster, whose hosts can only be accessed from within the docker container.

I ran into this issue when I needed to reproduce some strange issues with the Kafka instance provided by our private cloud provider. When maintenance happened and the nodes were being cycled, some topics and consumers seemed to completely loss track, and were unable to recover. They needed a restart of the service, before messages were being processed again.

Docker / Kafka setup

The main setup here is just a simple docker-compose file based on the great set of docker images from https://github.com/wurstmeister/kafka-docker (https://github.com/wurstmeister/kafka-docker). So without much further introduction, we'll look at the `docker-compose` file I used for this:

```yaml
version: '2'
services:
  zookeeper:
    image: wurstmeister/zookeeper
    ports:
      - "2181:2181"
  kafka-1:
    image: wurstmeister/kafka
    ports:
      - "9095:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: kafka1.test.local
      KAFKA_ADVERTISED_PORT: 9095
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LOG_DIRS: /kafka/logs
      KAFKA_BROKER_ID: 500
      KAFKA_offsets_topic_replication_factor: 3
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ${KAFKA_DATA}/500:/kafka

  kafka-2:
    image: wurstmeister/kafka
    ports:
      - "9096:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: kafka2.test.local
      KAFKA_ADVERTISED_PORT: 9096
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LOG_DIRS: /kafka/logs
      KAFKA_BROKER_ID: 501
      KAFKA_offsets_topic_replication_factor: 3
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ${KAFKA_DATA}/501:/kafka

  kafka-3:
    image: wurstmeister/kafka
    ports:
      - "9097:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: kafka1.test.local
      KAFKA_ADVERTISED_PORT: 9097
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LOG_DIRS: /kafka/logs
      KAFKA_BROKER_ID: 502
      KAFKA_offsets_topic_replication_factor: 3
```

```
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ${KAFKA_DATA}/502:/kafka
```

What we see here is a simple `docker-compose` file where we define a single Zookeeper node and three kafka nodes. Note that I've also expect the `KAFKA_DATA` variable to be set, which is used as an external volume. That way we don't lose the data when we remove the cluster. Let's look a bit closer at the individual Kafka nodes:

```
  kafka-3:
    image: wurstmeister/kafka
    ports:
      - "9097:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: kafka1.test.local
      KAFKA_ADVERTISED_PORT: 9097
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LOG_DIRS: /kafka/logs
      KAFKA_BROKER_ID: 502
      KAFKA_offsets_topic_replication_factor: 3
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ${KAFKA_DATA}/502:/kafka
```

Here we see the following:

- We expose the Kafka port `9092` on the external host on a unique port `9097` (we do this for each Kafka node in the cluster).
- To make this work correctly we also set the `KAFKA_ADVERTISED_PORT` to `9097`, so clients can connect to the nodes correctly after discovery.
- We need a unique host name for each node, if not. The cluster will complain the it already has a node with that name. So we set the unique name using the `KAFKA_ADVERTISED_HOST_NAME`. We use `kafka1.test.local`, `kafka2.test.local` and `kafka3.test.local` as host name for our cluster.
- Besides the settings above, we also give each node a unique id, using the `KAFKA_BROKER_ID` property.

The final step to take to get this working is having to make sure we can resolve the hosts (`kafka1.test.local`) specified here correctly. We can run our own dns server for this, but it is easier to just update the local host file `/etc/hosts`.

```
10.82.6.17 kafka1.test.local
10.82.6.17 kafka2.test.local
10.82.6.17 kafka3.test.local
```

Running the cluster

At this point we can simply start the cluster using `docker-compose` :

```
$ export KAFKA_DATA=/Users/jos/dev/data/cer/kafka
$ docker-compose -f ./docker-compose-local-kafka-cluster.yml up -d
Starting local_zookeeper_1 ... done
Creating local_kafka-3_1   ... done
Creating local_kafka-1_1   ... done
Creating local_kafka-2_1   ... done
 $ docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED
STATUS              PORTS                                             NAMES
da9d108cbc0e        wurstmeister/kafka       "start-kafka.sh"         7 seconds ago       Up
6 seconds        0.0.0.0:9095->9092/tcp                             local_kafka-1_1
3819d7ca3d7c        wurstmeister/kafka       "start-kafka.sh"         7 seconds ago       Up
6 seconds        0.0.0.0:9096->9092/tcp                             local_kafka-2_1
f8dc6ff937c6        wurstmeister/kafka       "start-kafka.sh"         7 seconds ago       Up
6 seconds        0.0.0.0:9097->9092/tcp                             local_kafka-3_1
62dfc7ea32c6        wurstmeister/zookeeper   "/bin/sh -c '/usr/sb…"   2 days ago          Up
6 seconds        22/tcp, 2888/tcp, 3888/tcp, 0.0.0.0:2181->2181/tcp   local_zookeeper_1
```

We can quickly check which nodes are part of the cluster by running a command against zookeeper:

```
$ docker exec -ti 62dfc7ea32c6 ./bin/zkCli.sh ls /brokers/ids
...
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[501, 502, 500]
```

And that's it. We've now got a cluster up and running, which we can use from outside the docker container, by just connecting to one of the three hosts. From the outside world, it'll look like a valid cluster and we can test failover scenarios or other settings by simply bringing nodes down and seeing how the clients react.

Final note

On a final note, you might notice the `KAFKA_offsets_topic_replication_factor: 3` setting. This setting defines the replication factor of the topic used to store the consumers offset. In the default case this is set to 1. So the consumer offsets for a particular topic will only be present

on a single node. If that node goes down, consumers will lose track of where they are, since they can’t update the consumer offsets. This was the main problem in our case. By setting this to `3` we could safely cycle the nodes of the cluster, without it affecting the coonsumers.