Enterprise Integration (MEIC-A, 2019-20, 2º semestre)

Instituto Superior Técnico – MEIC-A

Project Proposal

1. Definition of the mobility operators and respective messages

Mobility as a Service is an analogy of the usual Software as a Service model popularized by the Cloud for the software industry. The idea is the same, people will use the transport network they see fitting better their needs, be it the Public Transport Operators like Metro or Buses, Taxis or the new players such as Uber or Cabify and all other innovative alternatives for personal transportation like rental bikes, scooters, motorcycles, etc.

The innovation is the seamless use of all of them without the usual difficulties of different ticketing and payments systems, and the negative incentive for such use due to incompatibilities between cards, apps, tariffs, monthly subscriptions, etc.

The main idea of Maas is that one can take any transportation system and in the background his usage is being registered and one will pay for mobility according to the schema that best suits his needs.

As part of the proposal, we aim to integrate all the different transportation operators that provide services in Lisbon. For that purpose, we've defined the architecture of our Maas Operator based on an analysis we made of the types of services that are provided by each operator. We can generalize these services by gathering all of them in three types:

- **Type 0**: Check in and check out method
  - This method is used by Metropolitano de Lisboa and CP, for example
  - The user must have a ticket or a pass that is validated when he enters and when he leaves the transport and the price is fixed

Types of messages:

{"Metro": {"CheckIn": {"Token": "t1", "Station": "Odivelas", "Timestamp": "2020-02-29 18:23:41.278"}}}

{"Metro": {"CheckOut": {"Token": "t1", "Station": "Alameda", "Timestamp": "2020-02-29 18:23:47.718"}}}

- **Type 1**: Distance and time dependent method
  - This method is used by Uber and Cabify, for example
  - The price of the trip is mainly calculated using the distance and time of the trip

Types of messages:

{“Uber”: {“Usage”: {“Token”: “t2”, “Price”: “70.51901”, “Timestamp”: “2020-02-29 19:45:58.638”}}}

- **Type 2**: Time dependent method
  - This method is used by GIRA and Lime, for example
  - The price of the trip depends on how much time do you spend using the transport

Types of messages:

{“Gira”: {“Usage”: {“Token”: “t1”, “Time”: “3600”, “Timestamp”: “2020-02-29 20:57:10.294”}}}

To have a representative example of how our system would work in the “real world”, we chose one operator of each type to make sure that our system would cope with every type of public transportation operating in Lisbon. We chose **Metro, Uber and GIRA**.

2. Definition of the event queueing integration: Topics, Partitions

In our system we’ve defined that every operator has a topic because of two main reasons:

- One topic for company enforces the event load to be more distributed, one topic for all operators would have too much load
- It helps identifying the operator that sent the message

In addition, we’ve provided an optimization to the system by identifying the type of service in the name of the topic: **T0_Metro**, **T1_Uber** and **T2_GIRA**. This helps the consumer services to know the type of processing that is needed for the event simply by reading the name of the topic, which might be helpful in the future.

We’ve also identified the need to create a topic for discounts of the services to help with the revenue distribution and charging of the clients. The topic has the name **Discounts**.

For a better understanding of how the discounts topic will work we provide a short description of how we idealize the functioning of our system:

- Operators send events for their corresponding topic
- AccountManager Service and RevenueDistribution Service will both consume all the messages in each topic
- The AccountManager Service will have a database with information about the current discounts available in all the operators and the discounts that every user is entitled to
- When AccountManager Service identifies an event that should have a discount, it sends an event for the **Discounts** topic

- RevenueDistribution Service will consume all the messages of the Discounts topic as well and make sure that the user of the event has the discount when it charges him

Discount messages:

{“Discounts”: {“Value”: {“Token”: “t3”, “Discount Value”:”10”}}}

To provide some parallelism to our system we’ve created all the topics with 3 partitions.

3. Definition of the fault tolerance requirements for Kafka

In order to provide **high availability** in our system and as every topic has 3 partitions (number of partitions has to be less or equal to the number of brokers), we’ve created **3 brokers** in our Maas operator, added a **replication factor** of 3 to every topic and configured a **minimum of 2 ISR**. This way we can have a **leader of each topic in every broker** and **we guarantee that there will be another ISR even if the leader fails**, providing a normal service to our users even if two of our brokers fail. In addition, we will have 3 zookeeper nodes to avoid single points of failure, which could happen if we had only one zookeeper nodes.

The Kafka cluster durably persists all published records using a configurable retention period. We set the **retention period** for 48 hours, so for the two days after the record is published, it is available for consumption, after which it will be discarded to free up space. We think that 48 hours is more than enough for the records to be consumed.

We need to have 2 **Consumer Groups**, one for RevenueDistribution Service and other for AccountManager Service, since each one of them must read all the messages from every operator. If these consumer instances were in the same consumer group, then the records would be load balanced between them.

4. Kafka installation

The architecture of our system contains:

- 1 EC2 Instance
- 3 Kafka Brokers
- 3 Zookeeper nodes
- One topic per operator plus one for discounts (4 topics in total), with:
    - 3 partitions per topic
    - Replication factor of 3
- One consumer group for AccountManager Service and another consumer group for RevenueDistribution Service

5. Kafka parametrization

- **We created data directories for all the zookeeper nodes to store data**

```
mkdir -p /usr/local/kafka/data/zookeeper1
mkdir -p /usr/local/kafka/data/zookeeper2
mkdir -p /usr/local/kafka/data/zookeeper3
```

- **We created zookeeper properties files**

```
sudo nano /usr/local/zookeeper/config/zookeeper1.properties
sudo nano /usr/local/zookeeper/config/zookeeper2.properties
sudo nano /usr/local/zookeeper/config/zookeeper3.properties
```

Zookeeper-1:	Zookeeper -2:	Zookeeper -3:
zookeeper1.properties: **dataDir=** **/usr/local/kafka/data/zookeeper1** **clientPort=2181** **tickTime=2000** **initLimit=5** **syncLimit=2** **server.1=localhost:2666:3666** **server.2=localhost:2667:3667** **server.3=localhost:2668:3668** **maxClientCnxns=0**	zookeeper2.properties: **dataDir=** **/usr/local/kafka/data/zookeeper2** **clientPort=2182** **tickTime=2000** **initLimit=5** **syncLimit=2** **server.1=localhost:2666:3666** **server.2=localhost:2667:3667** **server.3=localhost:2668:3668** **maxClientCnxns=0**	Zookeeper3.properties: **dataDir=** **/usr/local/kafka/data/zookeeper3** **clientPort=2183** **tickTime=2000** **initLimit=5** **syncLimit=2** **server.1=localhost:2666:3666** **server.2=localhost:2667:3667** **server.3=localhost:2668:3668** **maxClientCnxns=0**

dataDir – the location to store the in-memory database snapshots and, unless specified otherwise, the transaction log of updates to the database

clientPort - the port to listen for client connections

ticktime - the basic time unit in milliseconds used by ZooKeeper.

initLimit - is timeouts ZooKeeper uses to limit the length of time the ZooKeeper servers in quorum have to connect to a leader

(In this case the timeout for initLimit is 5 ticks at 2000 milliseconds a tick, or 10 seconds)

syncLimit - Amount of time, in ticks, to allow followers to sync with ZooKeeper

maxClientCnxns - Maximum number of concurrent connections that a single client can make to a single member of the ZooKeeper ensemble

(As this vale is set to 0, there is no limit of connections)

- **Opened the in-bound ports 2181, 2182 and 2183 in the AWS EC2 console for the zookeeper nodes.**
- **Opened the in-bound ports 2666, 2667, 2668 in the AWS EC2 console for followers to connect to the leaders**
- **Opened the in-bound ports 3666, 3667, 3668 in the AWS EC2 console for leader election**

- **We created the unique id for each zookeeper instance**

```
echo 1 | sudo tee /usr/local/kafka/data/zookeeper1/myid
```

```
echo 2 | sudo tee /usr/local/kafka/data/zookeeper2/myid
echo 3 | sudo tee /usr/local/kafka/data/zookeeper3/myid
```

- **We created 3 brokers**

```
cp /usr/local/kafka/config/server.properties /usr/local/kafka/config/server-1.properties
cp /usr/local/kafka/config/server.properties /usr/local/kafka/config/server-2.properties
cp /usr/local/kafka/config/server.properties /usr/local/kafka/config/server-3.properties

sudo nano /usr/local/kafka/config/server-1.properties
sudo nano /usr/local/kafka/config/server-2.properties
sudo nano /usr/local/kafka/config/server-3.properties
```

Broker-1:	Broker-2:	Broker-3:
config/server-1.properties: **broker.id=0** **listeners=PLAINTEXT://*<Public DNS>*:9093** **offsets.topic.replication.factor=3** **transaction.state.log.replication.factor=3** **transaction.state.log.min.isr=2** **log.dir=/tmp/kafka-logs-0** **log.retention.hours = 48** **zookeeper.connect=localhost:2181, localhost:2182,localhost:2183**	config/server-2.properties: **broker.id=1** **listeners=PLAINTEXT://*<Public DNS>*:9094** **offsets.topic.replication.factor=3** **transaction.state.log.replication.factor=3** **transaction.state.log.min.isr=2** **log.dir=/tmp/kafka-logs-1** **log.retention.hours = 48** **zookeeper.connect=localhost:2181, localhost:2182,localhost:2183**	config/server-3.properties: **broker.id=2** **listeners=PLAINTEXT://*<Public DNS>*:9095** **offsets.topic.replication.factor=3** **transaction.state.log.replication.factor=3** **transaction.state.log.min.isr=2** **log.dir=/tmp/kafka-logs-2** **log.retention.hours = 48** **zookeeper.connect=localhost:2181, localhost:2182,localhost:2183**

(The parameters not mentioned here remained with the default values)

broker.id – broker id

listeners – the address the broker socket listens on

offsets.topic.replication.factor - specify the replication factor for the __consumer_offsets topic. This topic stores information about committed offsets for each topic:partition per group of consumers

(We've set this value to 3 to take advantage of having 3 brokers, providing more redundancy for this information)

transaction.state.log.replication.factor - the replication factor for the transaction topic. Internal topic creation will fail until the cluster size meets this replication factor requirement.

(We've set this value to 3 to take advantage of having 3 brokers, providing more redundancy for this information)

transaction.state.log.min.isr - minimum ISR for this topic

(All the topics will have at least the leader and one replica in sync to continue to provide service)

log.dir – the directory in which the log data is kept

log.retention.hours - the number of hours to keep a log file before deleting it

(As mentioned before we set it to 48, the messages will be kept for 48 hours before they are deleted)

zookeeper.connect - ZooKeeper connection string

(Contains the addresses for the zookeeper nodes)

- **Opened the in-bound ports 9093, 9094 and 9095 in the AWS EC2 console for the Kafka brokers.**

- **We added these lines to .bash_profile to start the zookeeper nodes and Kafka brokers automatically**

```
sudo nano .bash_profile

sudo /usr/local/zookeeper/bin/zkServer.sh start "/usr/local/kafka/config/zookeeper1.properties"
sudo /usr/local/zookeeper/bin/zkServer.sh start "/usr/local/kafka/config/zookeeper2.properties"
sudo /usr/local/zookeeper/bin/zkServer.sh start "/usr/local/kafka/config/zookeeper3.properties"
sudo /usr/local/kafka/bin/kafka-server-start.sh -daemon /usr/local/kafka/config/server-1.properties
sudo /usr/local/kafka/bin/kafka-server-start.sh -daemon /usr/local/kafka/config/server-2.properties
sudo /usr/local/kafka/bin/kafka-server-start.sh -daemon /usr/local/kafka/config/server-3.properties
```

- **We exited Putty terminal and started it again for the commands to run**

- **We verified that everything was working:**

```
ps -ef |grep java |grep server
```

- **Created the 4 Topics with 3 partitions and a replication factor of 3**

```
sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181,
localhost:2182, localhost:2183 -replication-factor 3 --partitions 3 --topic T0_METRO

sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181,
localhost:2182, localhost:2183 -replication-factor 3 --partitions 3 --topic T1_UBER

sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181,
localhost:2182, localhost:2183 -replication-factor 3 --partitions 3 --topic T2_GIRA

sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181,
localhost:2182, localhost:2183 -replication-factor 3 --partitions 3 --topic Discounts
```

- **Verified if all the topics were created correctly**

```
sudo /usr/local/kafka/bin/kafka-topics.sh --list --zookeeper <Public_DNS>:2181,
<Public_DNS>:2182, <Public_DNS>:2183
```

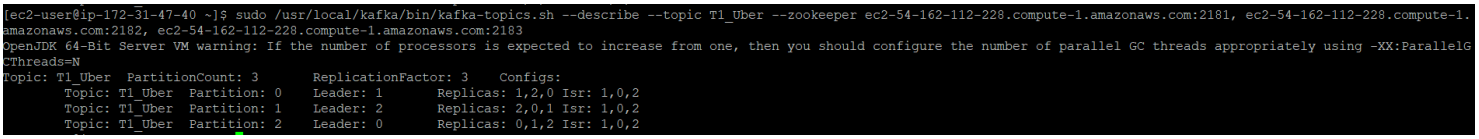
- **We checked the configuration of the topics**

```
sudo /usr/local/kafka/bin/kafka-topics.sh --describe --topic T0_Metro --zookeeper
<Public_DNS>:2181, <Public_DNS>:2182, <Public_DNS>:2183

sudo /usr/local/kafka/bin/kafka-topics.sh --describe --topic T1_Uber --zookeeper
<Public_DNS>:2181, <Public_DNS>:2182, <Public_DNS>:2183

sudo /usr/local/kafka/bin/kafka-topics.sh --describe --topic T2_GIRA --zookeeper
<Public_DNS>:2181, <Public_DNS>:2182, <Public_DNS>:2183

sudo /usr/local/kafka/bin/kafka-topics.sh --describe --topic Discounts --zookeeper
<Public_DNS>:2181, <Public_DNS>:2182, <Public_DNS>:2183
```

6. Test of the integration using applications for event generation

We decided to make a failure test for our Maas Operator using the T1_Uber topic.

- We started by checking that all the 3 brokers were well configured and who were the leaders of the partitions of this topic:



- Then we created a producer with the command:

```
java -jar MaaSMessageTaxiGenerator.jar --broker-list <Public_DNS>:9093,
<Public_DNS>:9094, <Public_DNS>:9095 --topic T1_Uber --token-list jkdjdjs --throughput
2000 --typeMessage JSON
```

(We decided to use a big value for the throughput to test if our Maas Operator could handle a big load)

- Then we created a consumer with the command:

```
/usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server <Public_DNS>:9093,
<Public_DNS>:9094, <Public_DNS>:9095 --topic T1_Uber --group g1
```

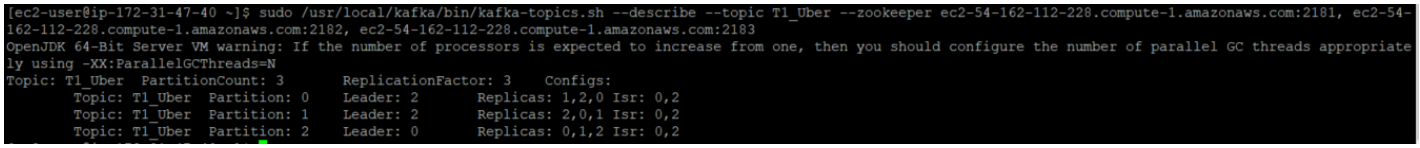
- After some time, we ran this command to check the PID of the brokers:

```
ps -ef |grep java |grep server
```

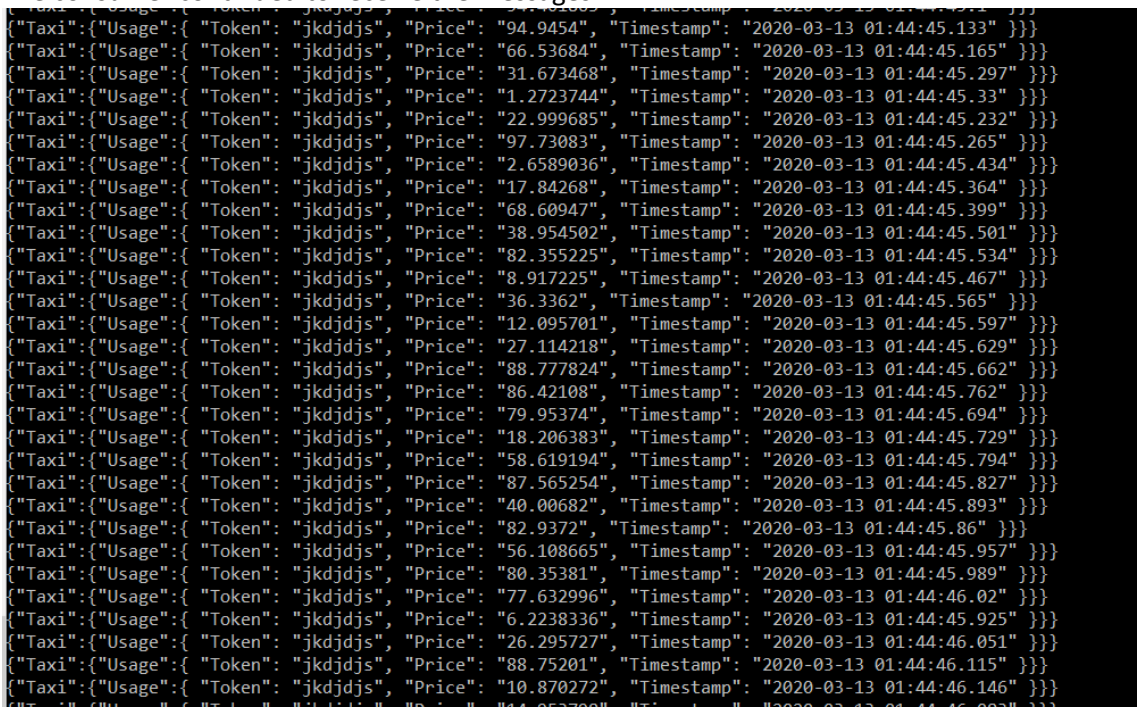
- Then we stopped broker 2:

```
sudo kill <Broker_2_PID>
```

- And checked the change in the leaders:



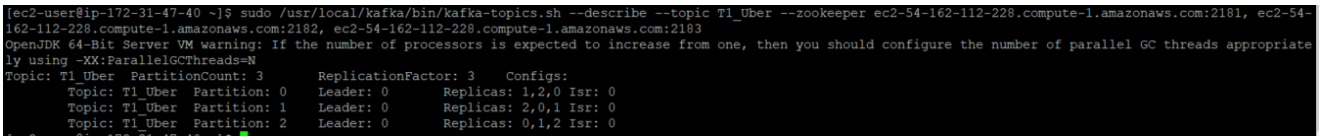
The consumer continued to receive the messages:



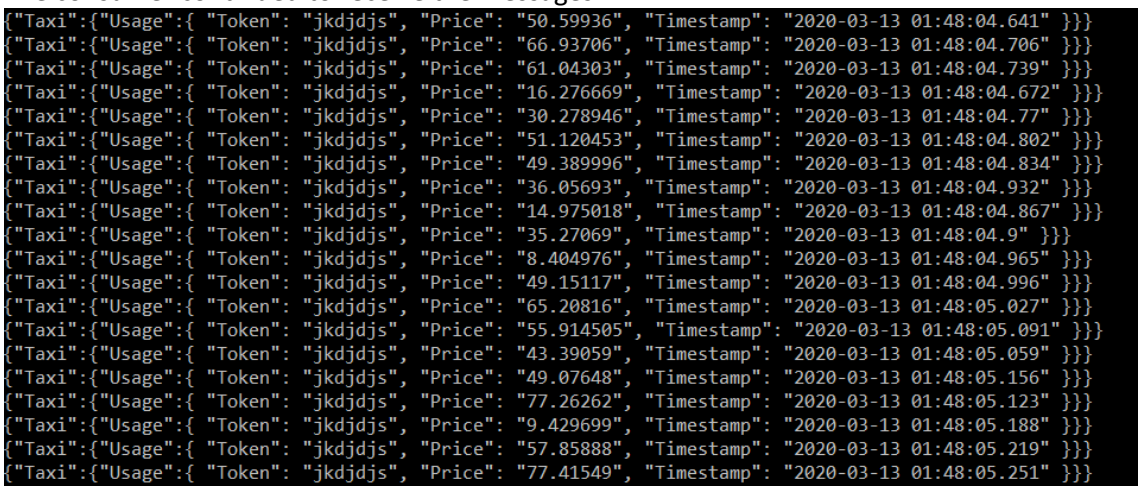
- After some time, we stopped broker 3:

```
sudo kill <Broker_3_PID>
```

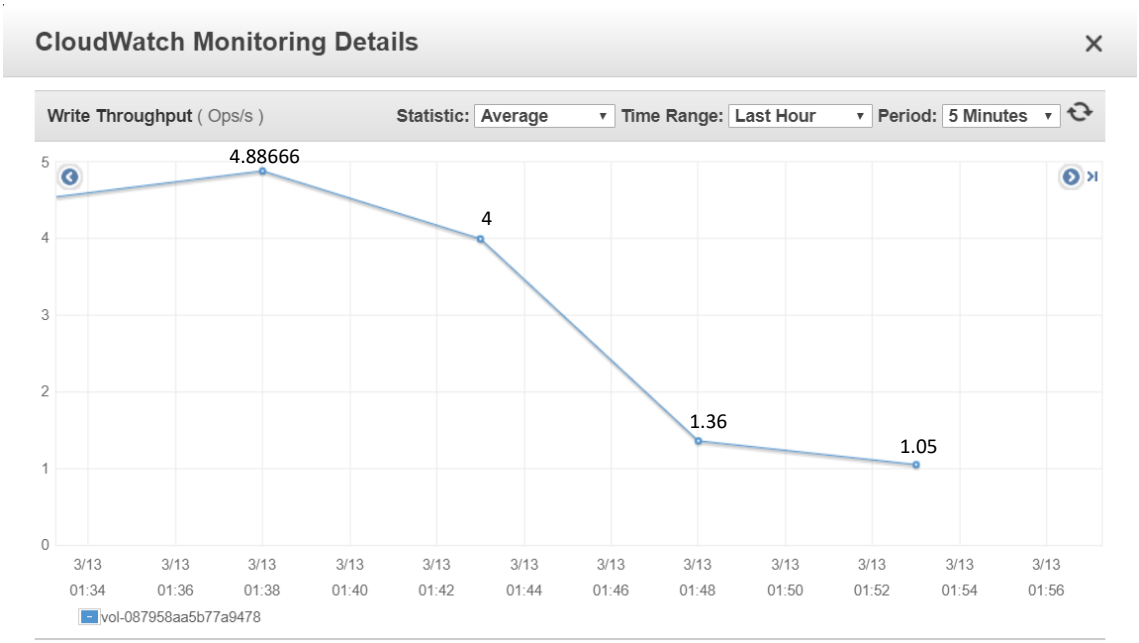
- And checked the change in the leaders:



The consumer continued to receive the messages:



Write throughput



01:37 – Messages started being sent
01:42 – Broker 2 stopped
01:47 – Broker 3 stopped
01:52 – We stopped the producer

By analyzing this graph, we can notice a big drop of the throughput when the first broker went down (01:42) and more subtle drop of the throughput when the second broker went down (01:47). Although there were drops in the throughput, the consumer continued to receive all the messages so we can conclude that our Maas Operator can tolerate two faults maintaining the correct functioning of the system.