**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”: “t1.4”, “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**.

Examples of messages received in a Kafka broker (sent in JSON format):

{“Discounts”: {“Value”: {“Token”: “t3”, “Discount Value”:”1”}}}

1. **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 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

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

1. **Definition of the fault tolerance requirements for Kafka**

In order to create **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 and added a **replication factor** of 3 to every topic. This way we can have a **leader of each topic in every broker** and **two ISR** in the others, providing a normal service to our users even if two of our brokers fail. In addition, we will have 3 zookeeper brokers to avoid single points of failure, which could happen if we had only one zookeeper broker.

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.

1. **Kafka installation**

We followed these steps from the tutorials for practical classes:

P2. B. Creating and launching an AWS EC2 instance

P2. C. Access the AWS EC2 instance using PuTTY

P2. E. Access the AWS EC2 instance using FileZilla

P2. F. Install Kafka in the AWS EC2 instance

Creating 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**

|  |  |  |
| --- | --- | --- |
| Broker-1:  config/server-1.properties:  **broker.id=0**  **listeners=PLAINTEXT://*<YourIP\_or\_DNS>*:9093**  **offsets.topic.replication.factor=3**  **transaction.state.log.replication.factor=3**  **transaction.state.log.min.isr=3**  **log.dir=/tmp/kafka-logs-0**  **log.retention.hours = 48** | Broker-2:  config/server-2.properties:  **broker.id=0**  **listeners=PLAINTEXT://*<Public DNS>*:9094**  **offsets.topic.replication.factor=3**  **transaction.state.log.replication.factor=3**  **transaction.state.log.min.isr=3**  **log.dir=/tmp/kafka-logs-1**  **log.retention.hours = 48** | Broker-3:  config/server-3.properties:  **broker.id=0**  **listeners=PLAINTEXT://*<Public DNS>*:9095**  **offsets.topic.replication.factor=3**  **transaction.state.log.replication.factor=3**  **transaction.state.log.min.isr=3**  **log.dir=/tmp/kafka-logs-2**  **log.retention.hours = 48** |

We added these commands to .bash\_profile:

**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**

Opened the in-bound ports 9093, 9094 and 9095 in the AWS EC2 console.

Created the 4 Topics:

**sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper *<Public DNS>*:2181 -replication-factor 3 --partitions 3 --topic T0\_METRO**

**sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper *<Public DNS>*:2181 -replication-factor 3 --partitions 3 --topic T1\_UBER**

**sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper *<Public DNS>*:2181 -replication-factor 3 --partitions 3 --topic T2\_GIRA**

**sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper *<Public DNS>*:2181 -replication-factor 3 --partitions 3 --topic Discounts**

And verified**: ps -ef |grep java |grep server**

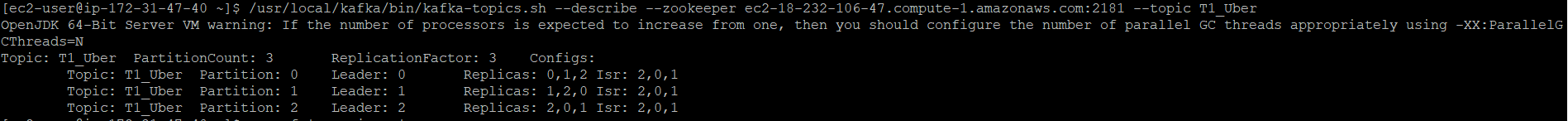
**sudo /usr/local/kafka/bin/kafka-topics.sh --list --zookeeper *<Public DNS>*:2181**

**sudo /usr/local/kafka/bin/kafka-topics.sh --describe --topic Topic\_Name --zookeeper *<Public DNS>*:2181**

1. **Kafka parametrization**
2. **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:

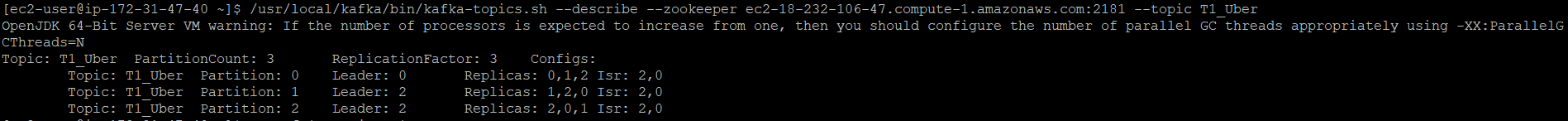
/usr/local/kafka/bin/kafka-console-consumer.sh --bootstrap-server <Public\_DNS>:9093, <Public\_DNS>:9094, <Public\_DNS>:9095 --topic T1\_Uber --group g1

* 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:

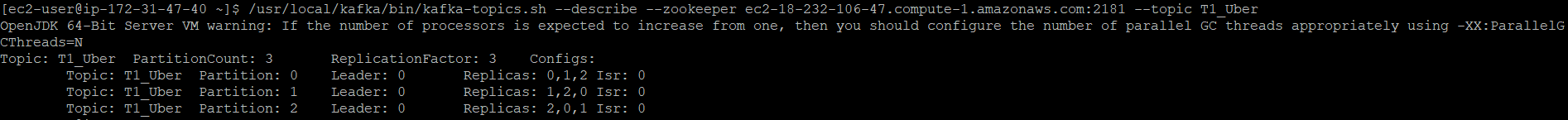
sudo kill <Broker\_2\_PID>

* And checked the change in the leaders:

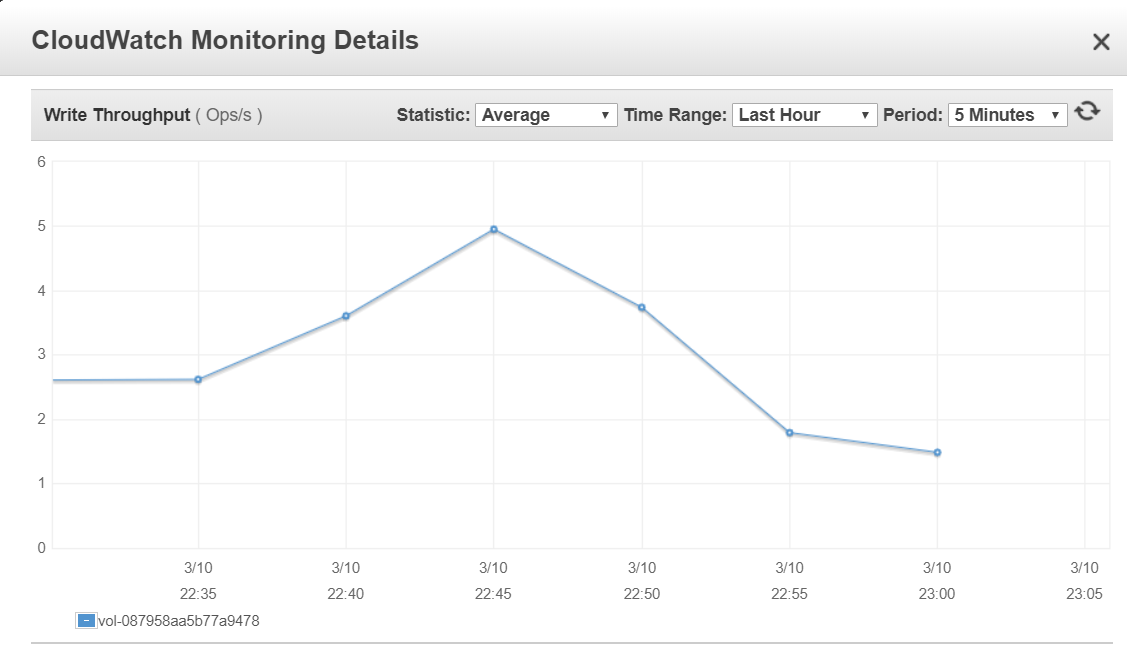
The consumer continued to receive the messages

* After some time, we stopped broker 3:

sudo kill <Broker\_3\_PID>

* And checked the change in the leaders:

**Write throughput**



1.4866666

2.62

3.60666

1.79333

3.74

4.95333333

22:35 – Messages started being sent

22:45 – Broker 2 stopped

22:50 – Broker 3 stopped

23:00 – We stopped the producer

By analyzing this graph, we can notice a drop of the throughput when the first broker went down (4.95333 to 3,74 ≈ 25%). When the second broker went down there was a drop as well (3.74 to 1.4866 ≈ 60%). 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.