IOT#BC-TRACE

Pablo Santiago Guilarte
Hugo de la Camara Saiz

DESCRIPTION AND OBJECTIVE

Objective: define a prototype tool to store traces of IoT data on a public platform like Ethereum.

Description:

- Data are provided from: IoT message generator
- Data are ingested in real time from MQTT broker and analysed with Apache Kafka, guaranteeing scalability and performance.
- Output analytics was saved on an external system: Ethereum Blockchain

TECHNOLOGIES

- Ubuntu 18.04
- Java 8
- MQTT
- Apache Kafka
- Ethereum
- Git and GitHub

LOGICS REQUIRED

Logic 1 - Write a Kafka Stream component that subscribes on the topic MQTT and:

Implements a window function (eg. 5 minutes) to group all the messages that belong to the same time window

Produce a new message on topic IOTRACE with the following JSON format:

{

"tz" : "2020-04-06T10:50:39.137+02:00", // ISODATE of the kafka message

"uuids" : ["UUID1","UUID2", ...]

{

LOGICS REQUIRED

Logic 2 - Write a prototype (e.g. in node.js) that subscribes to IOTRACE topic and persist the message on Ethereum

Logic 3 - Configure a Console consumer that dumps the IoT message from MQTT topic to a file. This file will be used to retrieve the original IoT messages from the UUIDs stored in Ethereum

TECHNOLOGIES

Thanks to MQTT we are able to complete the 1st and the 3rd logic

Thanks to Apache Kafka we are able to complete all the logics

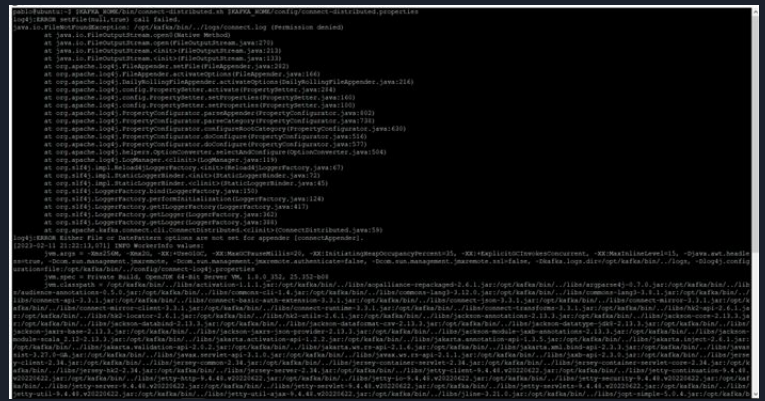
Thanks to Ethereum we are able to complete the 2nd logic

TECHNICAL IMPLEMENTATION

First we need to run Kafka:

KAFKA_HOME=/opt/kafka

$KAFKA_HOME/bin/connect-distributed.sh $KAFKA_HOME/config/connect-distributed.properties



TECHNICAL IMPLEMENTATION

Then, go to the simulator and run it too:

docker-compose up -d

docker-compose logs -f

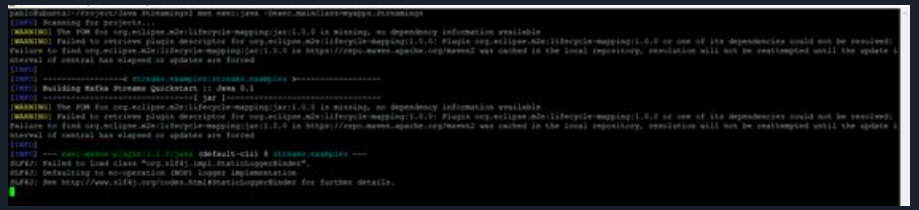
TECHNICAL IMPLEMENTATION

Then connect the connector MQTT of the simulator with Kafka:

curl -s -X POST -H 'Content-Type: application/json' http://localhost:8083/connectors -d @mqtt_connect.json

```
pablo@ubuntu:~$ curl -s -X POST -H 'Content-Type: application/json' http://localhost:8083/con
nectors -d @mqtt_connect.json
{"name":"mqtt-source","config":{"connector.class":"io.confluent.connect.mqtt.MqttSourceConnec
tor","tasks.max":"1","mqtt.server.uri":"tcp://localhost:1883","mqtt.topics":"#","kafka.topic"
:"mqtt.echo","value.converter":"org.apache.kafka.connect.converters.ByteArrayConverter","key.
converter":"org.apache.kafka.connect.storage.StringConverter","key.converter.schemas.enable":
"false","value.converter.schemas.enable":"false","confluent.topic.bootstrap.servers":"localho
st:9092","confluent.topic.replication.factor":"1","confluent.license":"","name":"mqtt-source"
```

TECHNICAL IMPLEMENTATION



Run the app that filters the streamings:

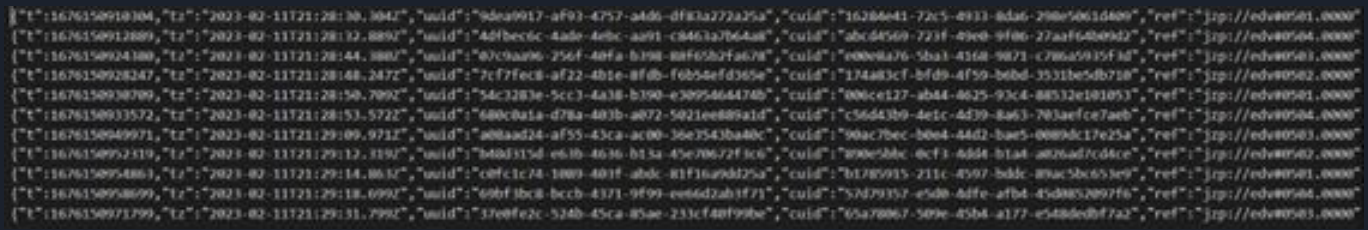
mvn clean package COMPILE

mvn exec:java -Dexec.mainClass=myapps.Streamings EXECUTE

TECHNICAL IMPLEMENTATION

At this moment MQTT is going to start generating messages:

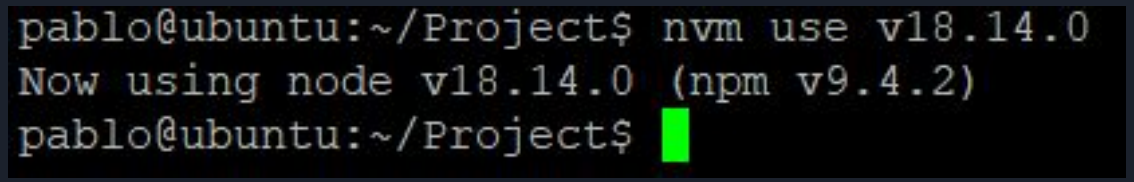
```
pablo@ubuntu:~/Project$ $KAFKA_HOME/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic mqtt.echo > Topic.txt
```



TECHNICAL IMPLEMENTATION

In order to execute the smart contract we must use the command:

nvm use v(Version)



TECHNICAL IMPLEMENTATION

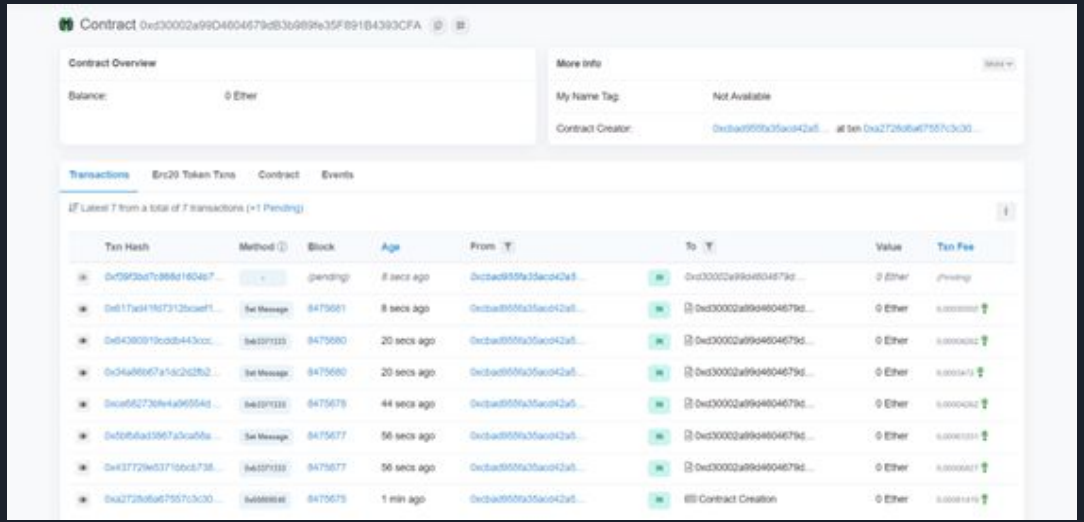
Later we go to the carpet where the deploy of the contract and the smart contract are:

node deploy.js

TECHNICAL IMPLEMENTATION

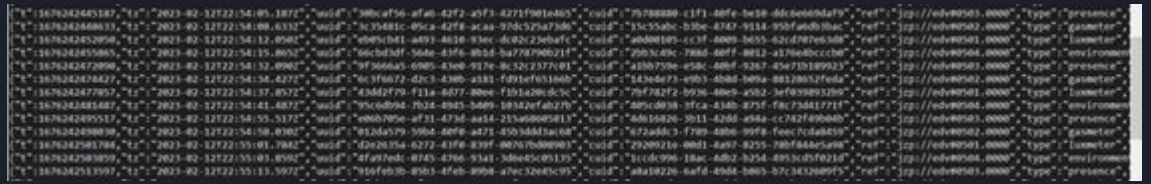
In the console the address of the contract is going to be spawned.

With the address we can go to the network Goerli to see its traceability:



ACHIEVED RESULTS

The smart contract in Goerli show us this trace:



FUTURE IMPROVEMENTS

- The contract could be verified with EtherScan, showing the functions in a public way
- The streamings could have other type of data
- A replication factor to work in parallel could be implemented
- An interface that show the messages in real time could be implemented