IOT Report
IoT-Based Indoor Air Quality Monitoring System

Pozzoli Davide

GitHub:
https://github.com/treyvian/IOT-unibo-project

Project report for
IOT exam

Artificial Intelligence
University of Bologna
Italy

Contents

1 Introduction

This the report for project of Internet of Things for the academic year 2021-2022. The project that I'm going to describe corresponds to the first proposal: **IoT-Based Indoor Air Quality Monitoring System**.

The project consists in the deployment of an IoT application for smart home scenarios, including functionalities of IoT-based monitoring of indoor environmental parameters such as temperature, humidity and gas concentration, data collection and data forecasting.

In next section, *Project's Architecture*, I will briefly describe the architecture of the pipeline required and then each component will be deepen in the chapter called *Project implementation*.

2 Project's Architecture

2.1 IoT smart device

As suggested in the project proposal for the hardware side of the project I choose:

- **ESP32** board: a of low-cost, low-power system on a chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth.
- **DHT11** sensor: a commonly used Temperature and humidity sensor that comes with a dedicated NTC to measure temperature and an 8-bit microcontroller to output the values of temperature and humidity as serial data.
- **MQ-2** sensor: gas sensor sensitive for LPG, propane, hydrogen. It outputs voltage boosts along with the concentration of the measured gases increases.

Everything is of course connected through a breadboard and cables.

The esp32 board can connected trough WiFi, Bluetooth or a micro-usb cable. For the project the only connection used to transfer data and communicate with other devices is the WiFi connection.

I implemented two communication protocols for sending data from the sensor to the data proxy: *HTTP* and *MQTT*. The only protocol supported is the MQTT for the communication with the device, in order to change values at runtime. The esp32 plays the role publisher and subscriber for the MQTT protocol, even though the topics are different for the different roles.

2.2 Data Proxy

The data Proxy is constituted by two parts:

- Http REST server: which is in charge of receiving the POST calls from the IOT device

- Mqtt broker: which is in charge of receiving the messages published by the IOT device

The data is collected in both cases by the Data Proxy, which is the same software that runs also the http server and send it to the Data Management part. In this case the data proxy is a subscriber of the Mqtt broker which listens to the topic on which the data is published.

2.3 Data Management

As described by the project proposal this part is managed by:

- **InfluxDB**: a high-performance open source Time Series Database with a SQL-like query language, called InfluxQL. It has been designed to handle high write and query loads and is part of the TICK Stack, a set of open source projects which will help you manage huge amounts of time-stamped information, giving you all the tools for your analysis needs, in real-time.
- **Grafana**: a multi-platform open source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources.

InfluxDB collects the data from the Data Proxy and then the requested values(temperature, humidity, gas, AQI, rssi) are shown in their own graph inside a Dashboard.

2.4 Data Analysis

The data analysis is done using FB Prophet to predict the the values as requested. The result is then stored in influxDB and can be visualized through the Gafana dashboard. This is the only technique that I implemented through the a few scipts in python.

3 Project's Implementation

In this section I will describe the pipeline presented before in more technical details.

3.1 IoT smart device

Everything is organized and managed via Arduino for which a vast documentation is present online.

In the setup function the pin connection are initialized with the WiFi and the mqtt connections.

In the loop function the all the variables are collected, the AQI index is calculated and a message using either HTTP or MQTT (depending on the state of the variable *is_http*), is sent every *SAMPLE_FREQUENCY* seconds.

WiFi

The WiFi connection is dealt with the library *WiFi.h*. In order to connect a file called env.h must be present in the same folder of the arduino program containing the ssid and the password of the hostspot to connect to. The two values must be specified inside two variables *WIFI_SSID* and *WIFI_PASS* both as strings.

In the Arduino code there is the presence of two WiFiClients. Each client has a limit of being able to sustain only one TCP connection at a time, this means that with the two TCP protocols implemented using one client would cause the connection to always fall. Thats the reason behind the need of double clients even though they refers to the same WiFi connection.

Changing values at runtime

During the runtime of the board some values can be changed, as requested, using the mqtt protocol. The ip of the mqtt broker must be set in the variable *broker*. The variables that can be changed are:

- **SAMPLE_FREQUENCY**: via the topic *esp32/frequency*. It changes the sample frequency of the sensors. The value send must be an integer representing the number of seconds between two measurement.
- **MAX_GAS, MIN_GAS**: via the topic *esp32/(max_gas or min_gas)*. they are the values on which the AQI is calculated. The value are interpreted as an integer and in case the value is not correctly formatted the value set will be 0.
- **protocol**: via the topic *esp32/protocol*. The protocol to use to transfer data. There are two options http by sending *http* or mqtt sending *mqtt*.

3.2 Data Proxy

The principal component of the data proxy is the REST server deployed using Java spring boot. Sping framework is able to deal with the HTTP and the MQTT protocol.

Http

The REST server created is able to deal only with the POST request on the *serverip/endpoint* which is where the esp board sends the signal. The message is then automatically casted to create a Point object which is a class created in java that has as attributes the values sent by the board: temperature, humidity, etc.... When the message is receive and the Point object created also the time stamp is set as an attrubute of the object. The timestamp refers to the UTC timezone. The Java API of InfluxDB is then used to write the information to the influx database.

Mqtt

The path for the mqtt message is a little more complicated since Java boot supports only the client instance, so a separate broker is needed. I used Mosquitto which is an open source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 5.0, 3.1.1 and 3.1.

The message is published by the esp board to the mosquitto broker under the topic esp32/point, the message is then notified to the client defined inside the Java Spring application which is subscribed to the same topic. Once the message arrive to the client a new Point instance with the information inside the body of the message and the same API of Influx are used to load the data to the influx db.

How to excecute

After installing the mosquitto broker to lanch it the command to from the root directory of the project run it is:

```
mosquitto -c data_proxy/mosquitto.conf
```

It will launch and start to listen to all the connections coming from the port 1883

To execute the Java spring boot application the command is:

```
java -jar target/server-1.0.0.jar
```

3.3 Data Management

InfluxDB

The data is saved in the bucket called **IOT_exam**, under the measurement Point there are the fields containing all the time series requested. Under the measurement there are some also the prediction created in the data analysis part of respectively gas, temperature and humidity.

InfluxDB need to be launched before it can be used with the command:

```
influxd
```

and the UI can be accessed through the localhost at the 8086 port.

Grafana

The grafana dashboard can be accessed through the localhost at the port 3000 in the folder *IOT_exam* where all the time series are shown one for each value for the last hour of measurement. If available also the predictions lines will be shown along side with the real values.

Alert

Based on the project description an alert should be triggered in case the AQI value is greater or equal to 1. Now based on the specification the AQI should be 1 in case the moving average of the gas of the last 5 measurements is inside the range of MAX MIN gas set in arduino, it should be 0 in case the average goes beyond the MAX set and 2 in all the other cases. Now my assumption is that an alert should trigger when an extraordinary event happens like for example the moving average for the gas goes above the maximum set that could mean a gas leakage is happening somewhere near. Based on the report the alert should be triggered also when the value is 1, basically in a normal condition which seems to me to be an error.

In the project I decided to set the alert when the value of the AQI goes to 0, so basically when the average of the last 5 measurement of gas goes above the Maximum set.

3.4 Data Analysis

The data analysis is done completely in Python. I divided the analysis in 3 parts to be executed in order based on the number at the beginning of the file name.

00_get_data

The main function of this file is to retrieve the data need to execute the forecasting.

Before executing some things need to be set, starting from the *BUCKET_NAME, TOKEN, ORG* which are constants defined inside the file utils.py. Those information need to be set based on the InnfuxDB configuration. The other thing that needs to be set is the variable *query* which of course is a string corresponding to an Influx query and it is used to download the data from the InfluxDB.

Thanks to the *influxdb_client* library the data can be directly downloaded from the script and saved inside the folder *data* in the csv format.

As presented the script will download the data of temperature, humidity and gas and saving each timeseries in its own csv file.

01_fbprophet

The second file is responsible for execute the Prophet algorithm on the data previously downloaded. I moved the training of the model inside the function *run_prophet* for better clarity and organization. In this file everything is basically already set if the previous file had been already executed.

As requested by the project when launching this script an argument must be specified at command line. With the argument *–x* the number of prediction to be made. The algorithm is set to forecast one point every hour so the x corresponds to the number of hours in the future you want to predict.

The prediction are both saved in the folder *predictions* in a csv format but also sent to InfluxDB since they are required to be showed inside the Grafana dashboard.

02_mse

This file as the name suggests is responsible for the calculation of the *Mean Squared Error* of the forecasted values. In order to calculate it the real values must be present in InfluxDB in order to confront them.

The script execute another query in the DB in order to obtain the real values, after that a few operations are done in order to match the predicted values with the closest measurement present in the DB. Then the *mse* is found thanks to the sklearn function.

The mse values are simply printed inside the terminal and are recorded inside the file called *mse.pdf*

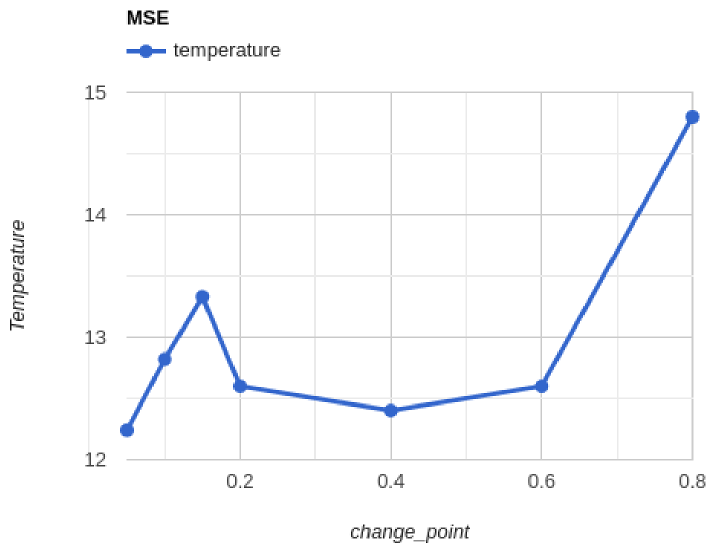
4 Results

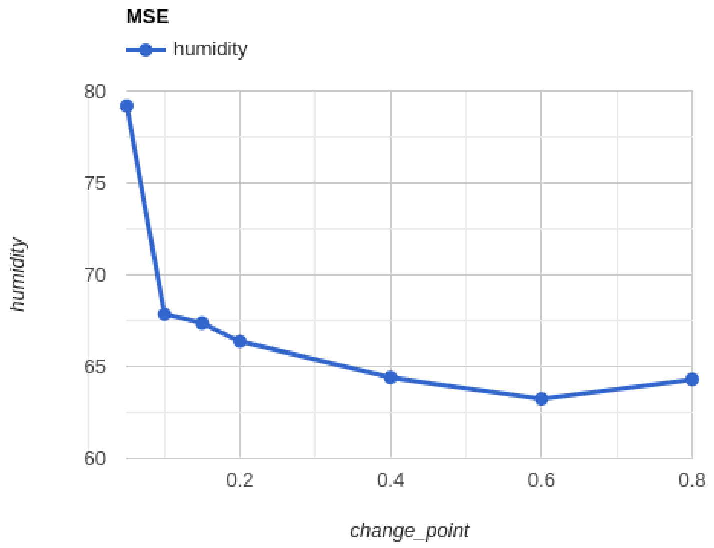
In this chapter I will analyze the results of the *MSE* for the predictions and the stress test on the server and broker.

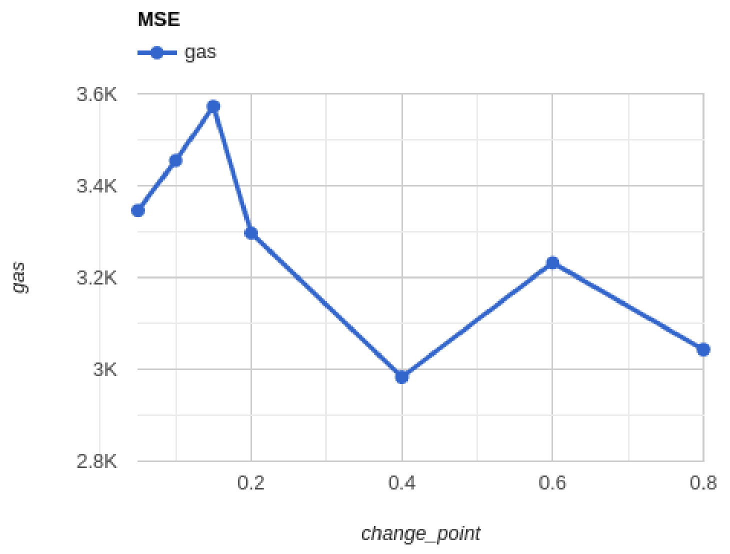
4.1 MSE

In order to evaluate the prediction algorithm I let the board collect data for 3 days and 12 hours at a sample frequency of 15 seconds. In particular from the 8pm of the 6^{th} of June to the noon of the 26^{th}. The data used for training was unitil the midnight of the 26^{th} and the goal was to predict the following 12 hours.

The dataset created is classified by Prophet as sub-daily dataset and the daily seasonality is automatically enabled. Since the dataset is not big enough to have a precise prediction I tried to change the parameter *changepoint_prior_scale* during training. This parameter can significantly impact the prediction: A very small value will force a linear trend, while a large value will allow the trend to fluctuate quite a bit. The default value is 0.05 and since the data is abundant I tried to increased it step by step.


MSE
temperature
15
14
13
12
0.2
0.4
0.6
0.8
Temperature
change_point


MSE
humidity
80
75
70
65
60
0.2
0.4
0.6
0.8
humidity
change_point



As we can see from the images we have an improvement of the mse while increasing the change point whit a minimum around 0.4, but if it is increased too much (ex. 0.8) the mse will actually degrade since there is too much degree of movement.

The average mse for the different configuration tested is:

- *humidity*: 67.53
- *temperature*: 12.97
- *gas*: 3275,6

4.2 Performance evaluation

For the performance evaluation I tested the two protocols separately, for the HTTP server I used a software called Locust while for the mqtt broker I used Mqtt stresser.

In the report it was asked to calculate also the packet delivery ratio, but in the next part of this chapter you will only find the average delay, that's because both protocols run over TCP and loss shouldn't be a problem (apart from a performance impact) as they will get re-sent even if the connection is momentarily lost.

All the test were conducted from a machine on the same network but different from the one hosting the server and the broker.

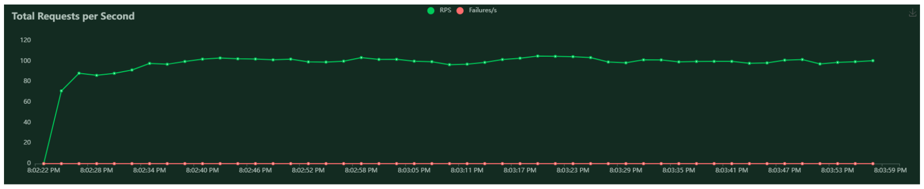
4.2.1 Locust

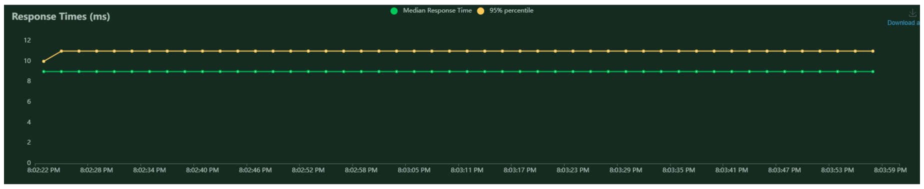
All the information described in this section can be found inside the folder *locust* in the root folder of the project. For all the test done there is the complete report in html format and a csv containing all the data collected.

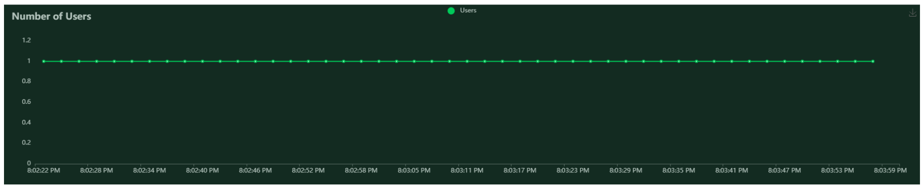
Locust is simple program that works with python code and is able to generate a load from a swarm of user simultaneously and report back the information about the performance of the server.

In the test collected I tried increasing the number of user submitting request to the server to understand how it would react.

1 user

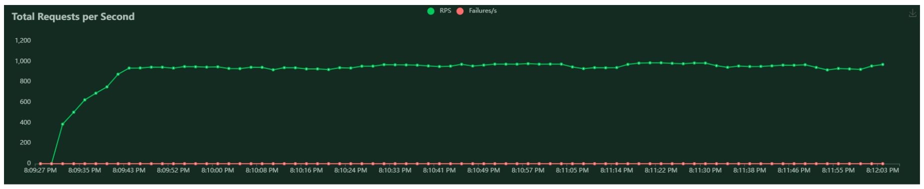


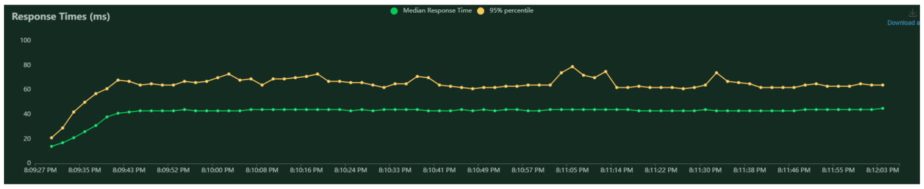


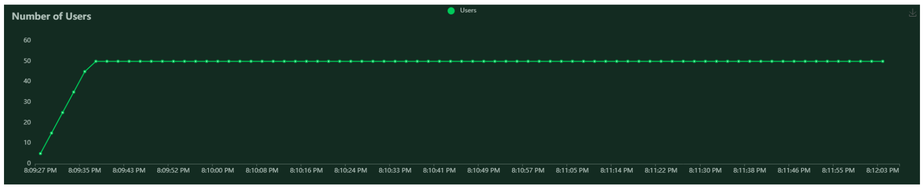


With a single user we can see that the response time of course is keep under the 12 ms which is expected by this amount of requests.

50 users

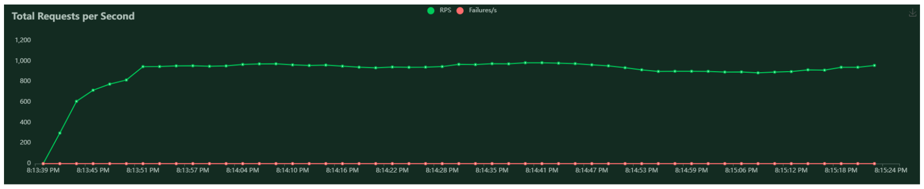


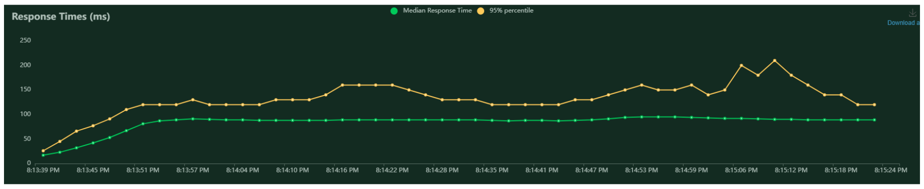


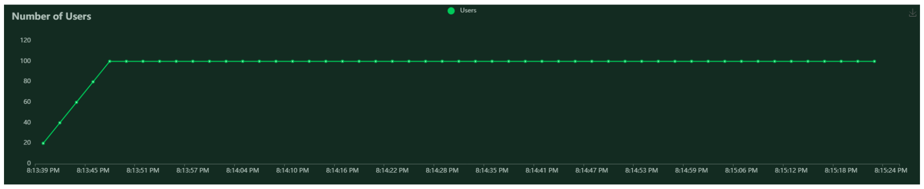


With 50 users we can see that even if the server is more than capable of supporting all the load and no request has been lost the response time start to increase, not the median one that remains fairly stable, but the 95% percentile which start to wobble a bit especially in some instants.

100 users







With 100 users the performances take a hit by doubling the response time and having some response time above the 150ms on the 95% percentile. Still no request were lost and the server continued to operate without a problem.

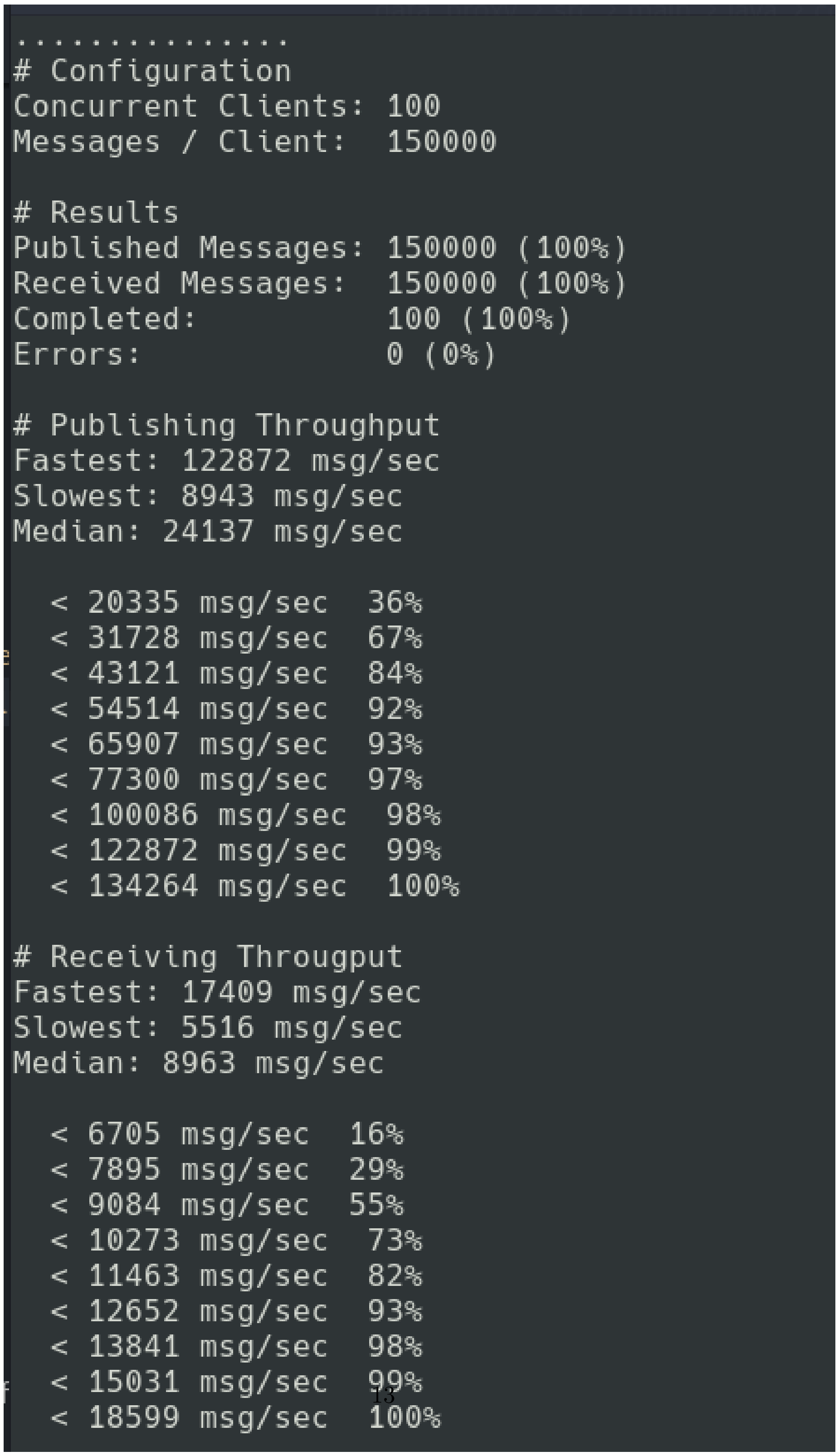
We also need to consider that this was a synthetic load and even with 100 clients connected all together the server would still receive an amount of request an order of magnitude inferior. That said these kind of tests are always useful to test the limit of the implementation provided.

4.2.2 Mqtt stresser

Mqtt stresser is a github project which allows to stress an mqtt broker in almost the same manner as Locust, by sending a huge number of messages with a swarm of users and then record the time taken for the broker to send back the acknowledgement. In this case the stresser is both subscriber and publisher on the same topic to record the full rountrip time of the message.

I tested it with 100 user and 1500 messages per client and even with this synthetic load and this huge amount of messages the broker was able to keep it up without errors and with a reasonable amount of time to respond.

The image with the results can be found in the next page.



```
...............
# Configuration
Concurrent Clients: 100
Messages / Client:  150000

# Results
Published Messages: 150000 (100%)
Received Messages:  150000 (100%)
Completed:          100 (100%)
Errors:             0 (0%)

# Publishing Throughput
Fastest: 122872 msg/sec
Slowest: 8943 msg/sec
Median: 24137 msg/sec

  < 20335 msg/sec  36%
  < 31728 msg/sec  67%
  < 43121 msg/sec  84%
  < 54514 msg/sec  92%
  < 65907 msg/sec  93%
  < 77300 msg/sec  97%
  < 100086 msg/sec  98%
  < 122872 msg/sec  99%
  < 134264 msg/sec  100%

# Receiving Througput
Fastest: 17409 msg/sec
Slowest: 5516 msg/sec
Median: 8963 msg/sec

  < 6705 msg/sec  16%
  < 7895 msg/sec  29%
  < 9084 msg/sec  55%
  < 10273 msg/sec  73%
  < 11463 msg/sec  82%
  < 12652 msg/sec  93%
  < 13841 msg/sec  98%
  < 15031 msg/sec  99%
  < 18599 msg/sec  100%
```