IoT-Project:
Sending data to cloud

Andrew Gagliotti

July 11, 2021

Contents

Chapter 1

Introduction

1.1 What to do

So, in this chapter we're going to talk about what is needed to do in order to complete the tasks of this application: let's imagine having a **SMART METER** which is capable of measuring temperature and humidity of the terrain. The **SMART METER** is divided in 4 devices, the detectors, a gateway which collects all data detected, and a cloud server, which receives them.
Goal is to realise all of this components and sending data from the devices to the gateway using an **UDP** connection and from the gateway to the cloud server using a **TCP** connection. For each connection established it must be present the print() of the buffer used and the time that has been required to send the packages.

1.2 Additional info

- The 4 IoT devices have a class C addressing, **192.168.1.0/24**.
- Gateway has 2 web interfaces: one towards the devices, one towards the web server.
- The IP address of the server is the class A type **10.10.10.0/24**, which is local and inaccessible from internet.
- This application must emulate, in Python, the behaviour of this system, using the loopback interface of our PC.

1.3 Commission

- Project is made for obtaining the access to the exam of the Web course of Engineering and Information Science university in Cesena.
- Responsible of the course is Professor **Giovanni Pau**. Responsible of the practical part of the course is Professor **Andrea Piroddi**.
- This code has been designed, written and maintained by me and it has been realised after studying all arguments of the course.

1.4 Modelling the problem

The system is composed by 4 collectors, the devices; a bridge, the gateway; a destination, the server. It is immediately clear that the application must contain different parts based on the scheme provided by Professor Piroddi:

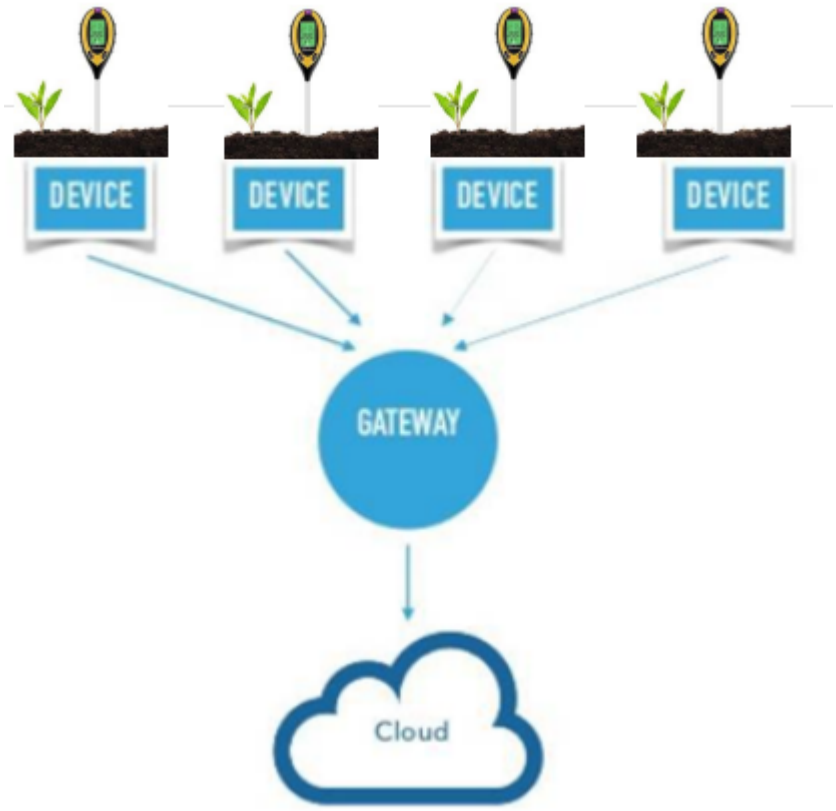

Figure 1.1: The system we must implement

Now let's think about realising this using the concepts of Python programming: we must realise different modules which will represents our entities and giving them a specific behaviour which will make the application functional according to the tasks required.

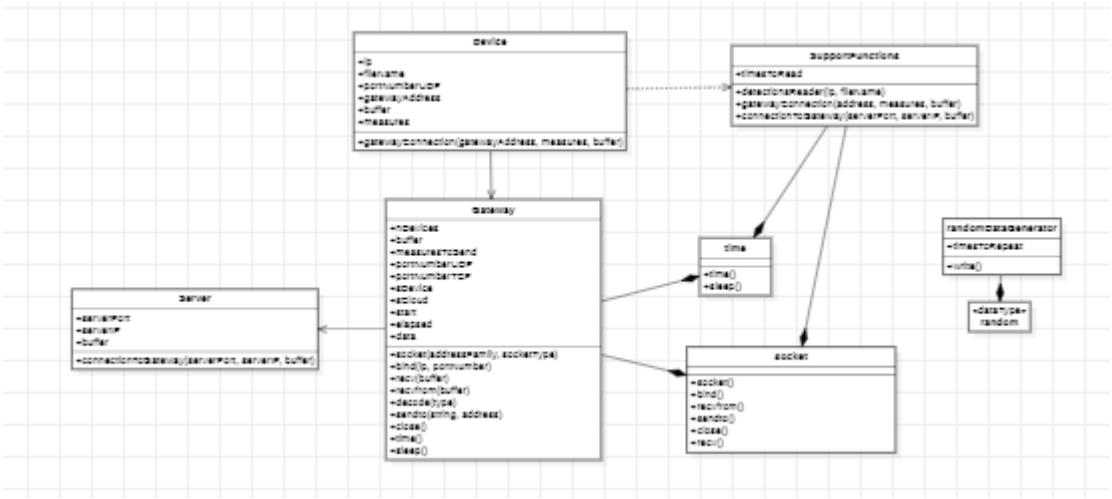

Figure 1.2: Basic UML structure: it'a bit unreadable but it will be all clarified later

Chapter 2

Design

This chapter contains how the project has been thought and designed in all his aspect. All section follows my line of thought for structuring the application.

2.1 Devices

Starting from the collectors, we can think them as real object with a clear purpose: they need to retrieve their data and encapsulate those in order to create a message that must be delivered to the gateway.
Thinking them as an Objects, they need:

- their own identification, which is their **IP** address: **'192.168.1.x'**
- and associated to the previous one, they also need a port number **100xx**. This one in particular must be the same as the gateway one, due to establish a correct **UDP** connection between the devices and the gateway.
- a place where to retrieve the data (*a file? why not*)
- the real message to send via **UDP**, which are the measures taken from the file.

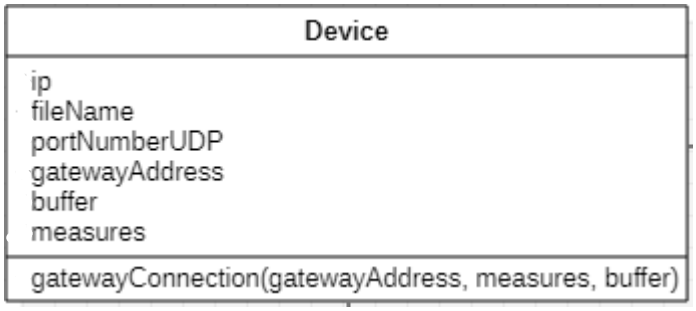

Figure 2.1: UML design for the devices

2.2 SupportFunctions

2.2.1 Reading detections

In this second istance I decided to create some useful functions for supporting my application and giving my code a nice look. This means, for now, that all devices use this module in order to work properly.
I start by defining the real core of the application, which means reading the detections coming from the devices and finding a way to establish a **UDP** connection with the Gateway.
We will think how to establish a connection to the server later.
For reading the detections we need to open the file where they are, then reading each line of it and at the end save all data in a field that must be returned to a device. Also, always remember to close a file when you're not using it anymore.

2.2.2 Establish connection from Device to Gateway

After all data have been collected and packed, it's now the time to send them to the Gateway via **UDP** connection.
First of all we are defining client-side of a **UDP** connection:

- Reserving a **UDP** type socket, specifying the **IPV4** address family and the socket type.
- Sending the message ...
- ... and then checking if message has been received. This is a precaution due to the not reliability of **UDP** connections.

- At the end it is needed to print buffer used and time occurred for transmitting data.
- The last function will be described later.

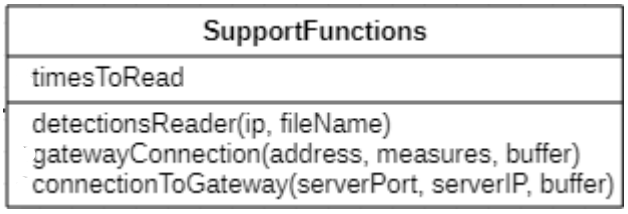

Figure 2.2: UML design for SupportFunctions module

2.3 Gateway

The real core continues to grow and now it's time to think about receiver side of the **UDP** connection and the client side of the **TCP** connection. As I thought it would have been nice to see, I imagined the Gateway as a bridge for sending data from a device to the server. I designed this large functional module with two behaviours.

2.3.1 UDP receiver

The first half of the module is a simple **UDP** receiver: like it was before we must reserve a socket (at the same way as the sender) and put the gateway into a listening status on the same port that the devices use. After data has been received we must save those into a variable that must be sent to the server later. According to the client side of the connection, it is required to send a message back to the client for the correct reception of the data. At the end it is needed to print buffer used.

2.3.2 TCP sender

The second half of the module it's the structure of a client side **TCP** connection component. In order to define this we need to:

- Reserve a socket specifying as always the **IPV4** address family but, this time, a different socket type.
- Establish firstly the connection to the client (reliable connection)

- After this, it's time to send the data and waiting for the confirmation message of correct receiving.
- At the end it is needed to print buffer used and time occurred for transmitting data.

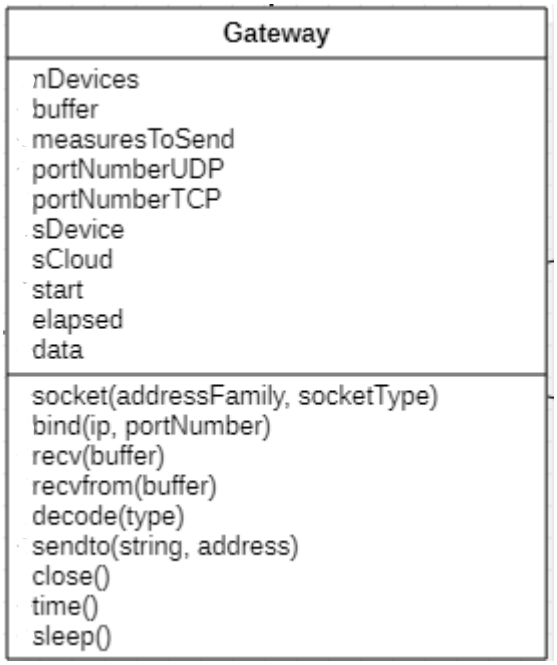

Figure 2.3: UML design for the Gateway

2.4 Server

This is the last component of the application, and it simply defines it's characteristic and implements the receiver side of the **TCP** connections.

2.4.1 The receiver

The receveiver part needs some components in order to work: a **TCP** port, a personal **IP** address (the given one - **10.10.10.x**) and a buffer size.

2.4.2 SupportFunctions - Establish connection from Device to Gateway

Do you remember the last part that I've skipped in explaining the SupportFunctions module? I've decided to encapsulate the Server as a receiver device, with the only purpose of calling a function for retrieving data. The function it's defined in the SupportFunctions module and it works as a **TCP** receiver:

- It reserve a socket with the same **IPV4** family address and the **TCP** type port
- It waits for the connection request from the gateway.
- After receiving the request, accepts it and then we must receive and save the data on a variable to print.
- At the end it is needed to print buffer used.

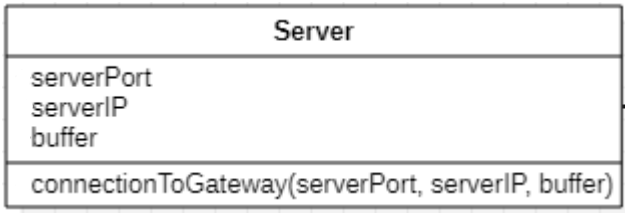

Figure 2.4: UML design for the Server

Chapter 3

Development

In this part i will show how I structured my code following the **Design** chapter written before. The code also contains some operations for time measuring and it contains also some data printing in order to understand better how the connections works.

3.1 Devices

Device is an object defining its own data and calling the detections retriever for sending them to the Gateway. So, I considered a Device as an **UDP** client. This is the related code:

```
# We are defining ip address, name of the related file, gateway address to
# connect with.
# After this we start reading the file content and send it to the gateway
ip = '192.168.1.1'
fileName = 'Measures01.txt'
portNumberUDP = 10024
gatewayAddress = ('localhost', portNumberUDP)
buffer = 4096

measures = sf.detectionsReader(ip, fileName)
sf.gatewayConnection(gatewayAddress, measures, buffer)
```

3.2 Detections reader

This function, which is in the SupportFunctions module, opens a device related file and reads its data, after this, saves all contents on a variable(String type). This is the related code:

```
# TImes to read the file
timesToRead = 4

# In this second istance I decided to create some usefull functions for
# my application and giving my code a nice look.
#
# I start by defining the real core of the application, which means rea
# coming from the devices and finding a way to establish a UDP connecti
# the Gateway

# I start defining the way the application reads the detections:
# first we start opening the file, then reading each line of it and at
# the function saves all data in a field that must be returned, which a
#
# Why not use the list of string with the readlines() method? because t
# works with strings could be sometimes unreadable: lists are a little
# read and manage
def detectionsReader(ip, fileName):

    measures = ''

    #Opening the file
    path = '../Data/' + fileName
    file = open(path, 'r')
    print('Reading available data ...')
    time.sleep(2)

    read = ''
    #Reading all measures and creating a string
    for i in range(timesToRead):

        read = file.readline()
        measures = measures + '{}: {}' .format(ip, read)

    # Closing the file and returning the string
    file.close()
    print('Data of {} has been read. Closing related file' .format(ip))
    return measures
```

3.3 UDP Client

This function, which is in the SupportFunctions module, establish a **UDP** connection to the Gateway. So, each Device establish a **UDP** connection to the Gateway. I have explained before how this connection type works. This is the related code:

```
# Now it's time to establish the UDP connection and sending all colle
# the Gateway: in order to do this I need the Gateway address and the
# that have been collected from a specific device
def gatewayConnection(address, measures, buffer):

    # Establishing UDP connection - starting by creating the socket
    # and then sending info using a try statement for controlling any
    # exceptions due to UDP not reliability
    print('Opening socket ...')
    mySocket = sck.socket(sck.AF_INET, sck.SOCK_DGRAM)

    try:
        # Sending data and start measuring time occurred
        time.sleep(2)
        start = time.time()
        print('Sending ...')
        mySocket.sendto(measures.encode(), address)

        # recvfrom() reads a number of bytes sent from an UDP socket,
        # we are reading data from the Gateway - it's like waiting fo
        print('Waiting ...')
        data, server = mySocket.recvfrom(buffer)

        #elapsed time and printing info
        elapsed = time.time() - start
        time.sleep(2)
        print('Message: {}' .format(data.decode("utf8")))
        print('Size of used buffer is {}' .format(buffer))
        print('Time occured for UDP sending: {}' .format(elapsed))

    except Exception as e:
        print(e)
    finally:
        print('Closing socket ...')
        mySocket.close()
```

3.4 UDP Receiver

Now I'm gonna show how I structured the receiver of the **UDP** connection, which code is in the Gateway object: all measures are regrouped in a unique variable to be sent to server. This is the related code:

```
# Libraries used
import socket as sck
import time

# After the design of the basic fuctions and the creation of the
# now it's time to create the gateway, which purpose is to firstl
# data collected by all devices and then eastablish a TCP connect
#
# After establishing the connection now it's time to send the dat
# for the confirm of correct reception.

# Variables
nDevices = 4
buffer = 4096
measuresToSend = ''
portNumberUDP = 10024
portNumberTCP = 8080

# First we start to listen for the devices: so we need the host i
# the port number we are using
sDevice = sck.socket(sck.AF_INET, sck.SOCK_DGRAM)
sDevice.bind(("localhost", portNumberUDP))

# Now it's time to wait for the data - UDP connection
for i in range(nDevices):
    data, address = sDevice.recvfrom(buffer)
    measuresToSend = measuresToSend + data.decode('utf8') + '\n'
    time.sleep(2)
    print('Data receveived from {}' .format(address))
    sDevice.sendto('Data arrived.'.encode(), address)

# Closing socket after receiving the data collected
sDevice.close()
```

3.5 TCP Client

After collecting the data to a unique variable, the Gateway establish a **TCP** connection to the server,and this one will be already active because he needs to check if there is someone who wants to connect to it. This is the related code, which is in the second part of the Gateway:

```
# Resetting the buffer
buffer = 4096

# Now it's time to send all collected data to cloud server and in
# we need to establish a TCP connection between the gateway and th
# gateway is the sender
print('... it\'s time to open interface 10.10.10.5')
sCloud = sck.socket(sck.AF_INET, sck.SOCK_STREAM)
sCloud.connect(('localhost', portNumberTCP))
start = time.time()

# Sending data to server
sCloud.send(measuresToSend.encode())

# Now we have to wait for server response
print('Waiting ...')
data = sCloud.recv(buffer)

#Data printed
elapsed = time.time() - start
print('Message received: {}' .format(data.decode('utf-8')))
print('Size of used buffer is {}' .format(buffer))
print('Time occured for TCP establishing: {}' .format(elapsed))

# Closing
print('Closing socket ...')
sCloud.close()
```

3.6 Server

This is the last component of the application: it contains all identifications of the server and it acts as a **TCP** client of the connection to the Gateway. It calls a function in order to do that. This is the related code:

```
serverPort = 8080
serverIP = '10.10.10.5'
buffer = 4096

# Establishing the connection
sf.connectionToGateway(serverPort, serverIP, buffer)
```

3.7 TCP Receiver

This code is in the last section of the SupportFunctions module and it simply acts as a **TCP** Receiver, accepting the request of connection from the gateway and then printing, on the console and in a file, the received data. This is the related code:

```
def connectionToGateway(serverPort, serverIP, buffer):

    print('Establishing TCP connection ... \n')
    sSocket = sck.socket(sck.AF_INET, sck.SOCK_STREAM)
    sSocket.bind(('localhost', serverPort))

    # Listening the request for connection
    print('Interface: {}, port: {}' .format(serverIP, serverPort))
    sSocket.listen(1)

    # Accepting the connection and then it's time to receive data
    gatewayConnection, address = sSocket.accept()
    print('Gateway connected! \n')
    print('Data received are:\n')
    serverMessage = gatewayConnection.recv(buffer)

    # Printing data
    print(serverMessage.decode("utf8"))
    print('Size of used buffer is {}' .format(buffer))
    gatewayConnection.send(("Data received").encode())

    # Closing
    print('Closing connection and socket ...')
    gatewayConnection.close()
    sSocket.close()

    file = open('../Data/Output.txt', 'w')
    file.write(serverMessage.decode('utf8'))
    file.close()
```

3.8 Additional parts: Random Data Generator

This is an extra not required from the task. It simply generates a new random dataset on each device-related file. This is the related code, contained in the randomDataGenerator module:

```
import random

# Change here if you want more devices: remember to create new devices!
timesToRepeat = 4

# A dummy file content generator, it literally has a for loop with write method
for j in range(timesToRepeat):
    file = open('../../Data/Measures0{}.txt' .format(j + 1), 'w')
    for i in range(timesToRepeat):
        file.write('Time: {}:00 - ' .format(00 + i * timesToRepeat * 2) +
                   'Temperature: {} C -' .format(round(random.uniform(20, 40), 1)) +
                   ' {} % of humidity' .format(round(random.uniform(40, 80), 1)) +
                   '\n')

    file.close()
```

Chapter 4

Considerations

4.1 Self-evaluation and personal comments

This was a simply project for showing if the students are capable of structuring a correct connection between entities, even if it simulated on a local address and not via internet. I think this is the sum of all aspects of the connection types and it is good to understand that changing some variables may let this application even for online connection.

4.2 Problems occurred and questions

4.2.1 SupportFunctions

Why not use the list of string with the readlines() method? Because the way python works with strings could be sometimes unreadable: lists are a little tricky to read and manage for this project so I thought it would be better to operate with loops, even if they are more onerous in terms of time for doing an operation.
Due to the fact that we are operating with an high level language, it's sometimes difficult to operate with low level functions and returning some data types in a proper and good looking way.

4.2.2 Type of connections used and why

User datagram protocol (**UDP**) operates on top of the Internet Protocol (**IP**) to transmit datagrams over a network. **UDP** does not require the source and destination to establish a three-way handshake before transmission takes place. Additionally, there is no need for an end-to-end connection.

Since **UDP** avoids the overhead associated with connections, error checks and the retransmission of missing data, it's suitable for real-time or high performance applications that don't require data verification or correction. If verification is needed, it can be performed at the application layer

TCP is used for organizing data in a way that ensures the secure transmission between the server and client. It guarantees the integrity of data sent over the network, regardless of the amount. For this reason, it is used to transmit data from other higher-level protocols that require all transmitted data to arrive.
Let's notice that for each connections established, each component share the same port number and the same IP address.

4.2.3 Modules visibility

I had a really bad struggle into visibility of connected modules: it's my practice to give to different modules or packages their own directory and then use a correct wording to import them. Apparently my Python knowledge it's not so developed to make an ordinate application but it took all my efforts for making it good looking. I hope it will suits the requirement of good structuring.

Appendix A

User guide

In order to run the application you firstly need to open all files in Spyder and then choose to open other 5 consoles. We need a total of 6 consoles:

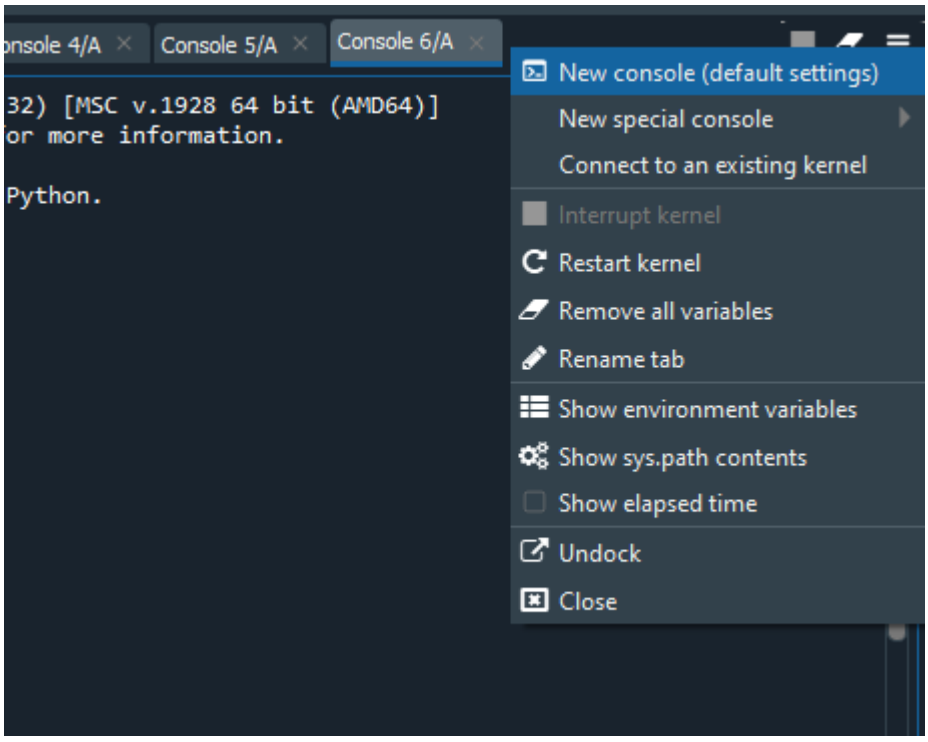

Figure A.1: Right click on the bar next to a console to show this menu, choose "New console (default settings)" and reach a total of 6 empty consoles

After this, you have to start the modules one for each consoles in this order:

- gateway.py
- Device01.py
- Device02.py
- Device03.py
- Device04.py
- server.py

You have 2 seconds delay to run each module so don't worry about rushing. As far as the application works, you will find on output all requirements, like this:

```
In [1]: runfile('C:/Users/andre/Desktop/ProgettoReti/IoT-Pro
Desktop/ProgettoReti/IoT-Project/src')
Data receveived from ('127.0.0.1', 61201)
Data receveived from ('127.0.0.1', 61202)
Data receveived from ('127.0.0.1', 61203)
Data receveived from ('127.0.0.1', 61204)
... it's time to open interface 10.10.10.5
Waiting ...
Message received: Data received
Size of used buffer is 4096
Time occured for TCP establishing: 0.0010008811950683594
Closing socket ...
```

Figure A.2: Gateway

```
In [1]: runfile('C:/Users/andre/Desktop/ProgettoReti/IoT-Proje
andre/Desktop/ProgettoReti/IoT-Project/src')
Reading available data ...
Data of 192.168.1.1 has been read. Closing related file
Opening socket ...
Sending ...
Waiting ...
Message: Data arrived.
Size of used buffer is 4096
Time occured for UDP sending: 2.0144731998443604
Closing socket ...
```

Figure A.3: Devices

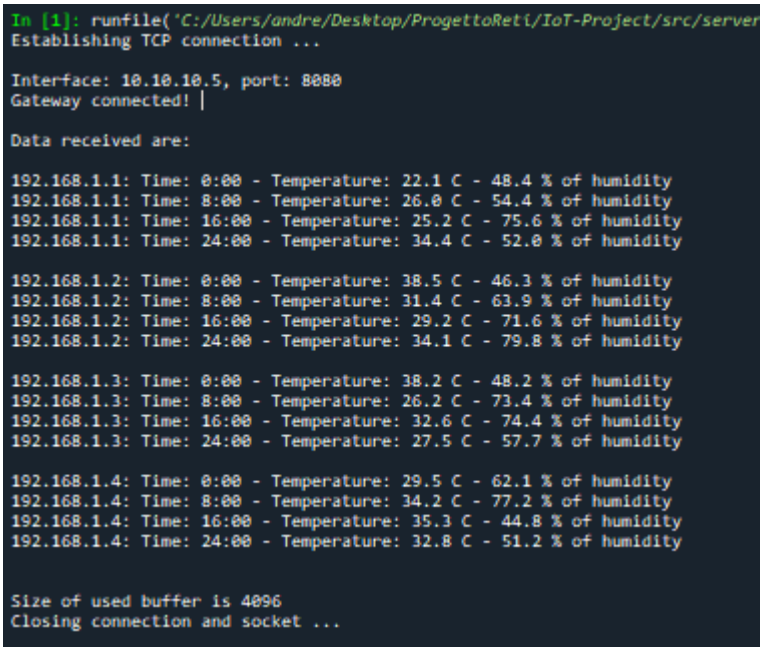

Figure A.4: Server

Why we need to start the modules in this order? That's because we must put the Gateway on a listening status and it can stay in this state indefinitely; then the devices will send to it all their data.
fter this, we put the server in listening status because, after all data have been received, the Gateway immediately asks for establish a TCP connection to the server, in order to send to it all data that have been collected. This output is based on the dataset you will find on GitHub.