UNIVERSITÀ DI PISA

Master Degree in Artificial Intelligence and Data Engineering
Internet of Thing

Smart Study Room

Leo MALTESE
Giulio FEDERICO

Link to the Github project: https://github.com/GiulioFede/SmartStudyRoom

Contents

Chapter 1

Introduction

1.1 Description

Smart Study Room is an IoT project which consists on monitoring the study rooms of the University of Pisa. In particular, the idea of the project consists on installing a series of sensors, in order to evaluate the quality of the study rooms in terms of noise, temperature and number of free seats. Indeed, as we can see in the Figure 1.1, we image to install sensors with the following functionalities:

- **PIR Sensors**: installed in each seat, in order to check if a seat is occupied by a student or not. If a sensor does not perceive a certain number of movements in a certain period of time then the seat is set to FREE, otherwise the seat is considered busy. There is also the possibility for the student to set on a seat that is free clicking the button present on seat in order to notify, in an immediate way, that the seat is free.

- **Temperature Sensors**: installed in each room, in order to monitoring the temperature of study room. That sensor, apart from monitoring the temperature, is used to receive the information about the possible modification of temperature of the study room. Indeed we implemented an application in which everyone , in the room, can choose its preferred temperature and, after a series of preferences, the temperature of the room will be modified with the mean of the preferences of temperature.

- **Noise Sensors**: installed in each table, in order to evaluate if a room is nosy or not. This measure can be useful for the students because they can choose the study room based on their preferences.

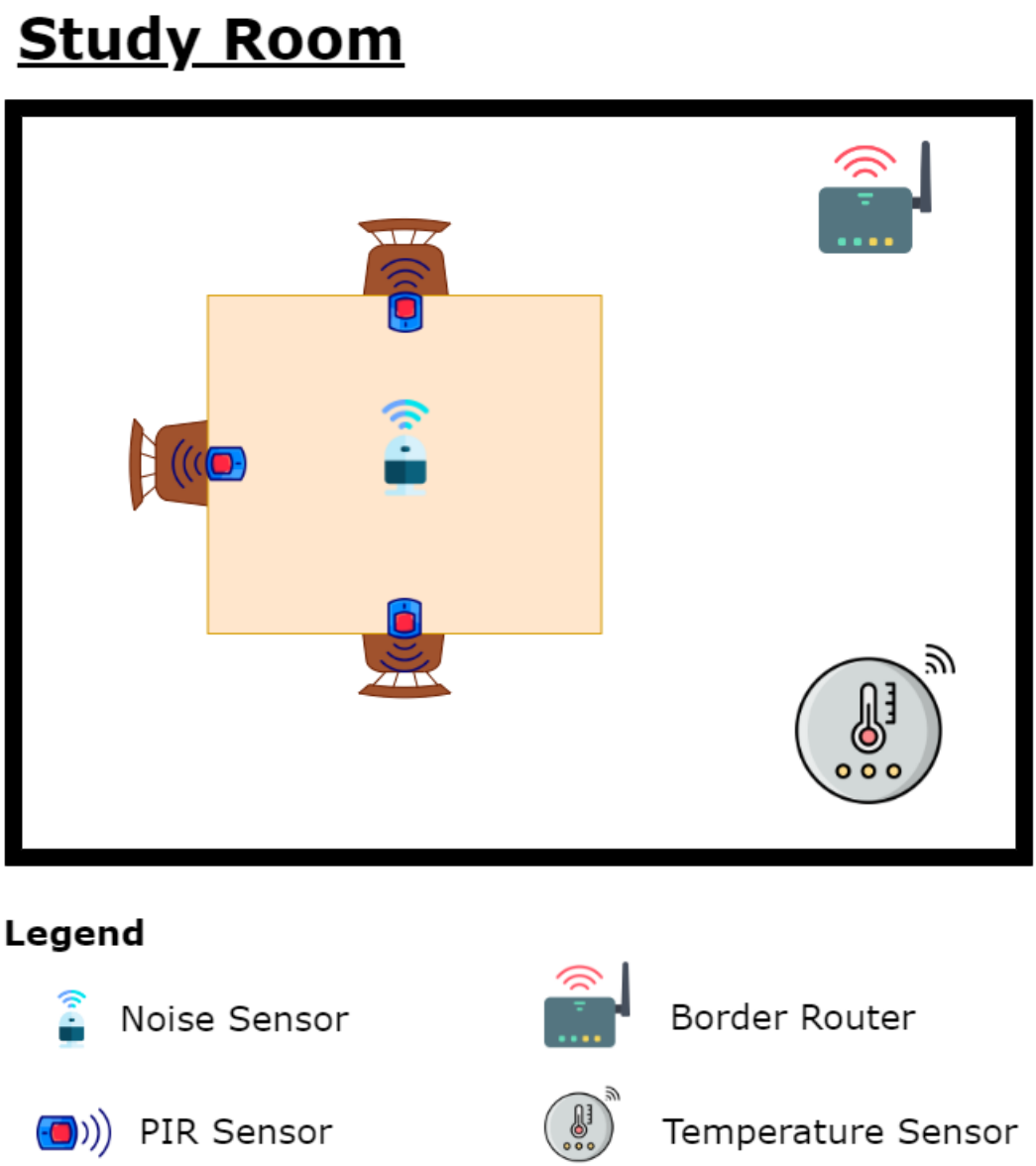

Figure 1.1: Representation of a study room with the sensors

Chapter 2

IoT Architecture

2.1 Schema IoT Architecture

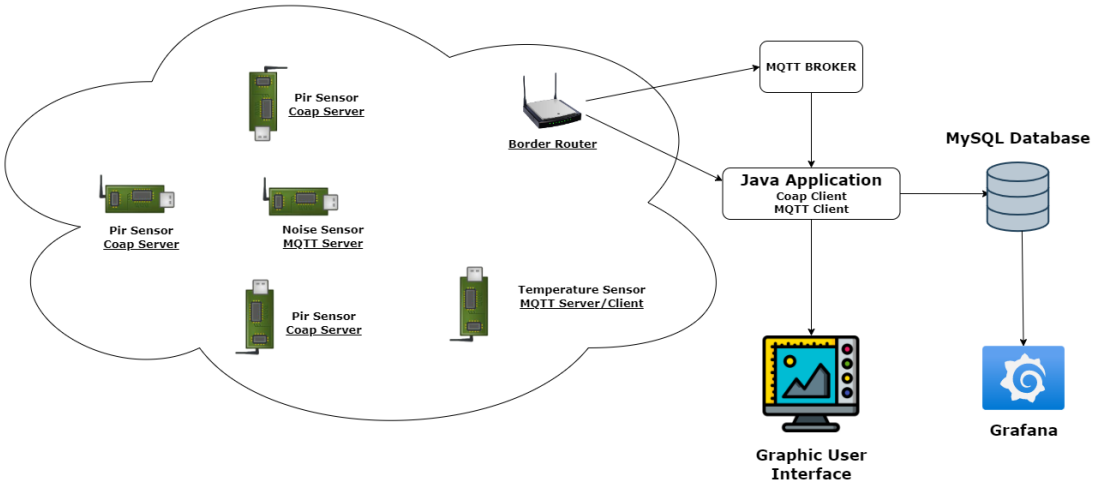

Figure 2.1: Representation of a study room with the sensors

In the figure above, Figure 2.1, is represented the IoT architecture. We have a series of sensors that exploit different protocols to communicate and receive data. These data is received/sent also by a Java Application. It collects the received data and sends these data to a database and to a GUI. Moreover, the data is retrieved by the Grafana to visualize some telemetry information about the study rooms (temperature and noise). Obviously the Border Router is used to forward the packets from the IoT Network to the MQTT Broker / Java Application, performing the role of RPL root.

2.2 Sensor and Application Protocols

The **temperature server** was implemented on the nRF52840 dongle board in Figure 2.2.

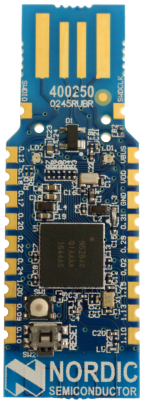

Figure 2.2: nRF52840 dongle board

It uses the MQTT protocol to post messages and subscribe to a topic. In particular, it publishes the average of the samples to the topic `"actuator_temperature_IDBUILDING_IDROOM"` and subscribes on the topic `"temperature"` to receive commands as actuator. The ID_BUILDING and ID_ROOM will be replaced by each sensor with the value of the building and room in which they are located.

The **noise server** was implemented on the CC2650 Launchpad in Figure 2.3.

Figure 2.3: CC2650 Launchpad

It also uses the MQTT protocol to post messages. In particular, it publishes the average of the samplings on the topic `"noise"`.

Both the two servers subscribe to the topic `"time_synch"` to receive, from the client, the notification to set the correct reference time so as to correctly calculate the offset which it will then send to the client to reconstruct the sampling timestamp.

Finally the three **PIR servers** have been implemented both in the dongle (two of them) and in the launchpad (one of them). They are the only ones that implement the CoAP protocol by exposing the resource `"/seat"` which contains the value of the state (1 if free, 0 if occupied) of the seat in which it is mounted. This resource is *observable*.

On the *client side*, the Java app subscribes to topics `"actuator_temperature_IDBUILDING_IDROOM"` and `"noise"` and publishes temperatures in the topic `"temperature"`; it then performs the observation on the `"/set"` resources exposed by the three PIRs.

2.3 Data Encoding

For the data exchange we used the well known JSON data format. The main reason is that, being in an IoT environment, therefore with constrained devices, the JSON format allows us to send less and faster data, which translates into lower energy consumption and the possibility of real-time monitoring. However, the energy constraint can be overcome since in the environment where the sensors work, we foresee the possibility of a continuous power supply. Furthermore, the data we exchange is primitive, simple data and does not require validation given the environment lacking requirements such as data integrity and security.
The server that samples the temperature uses this scheme for publishing the temperature value:

Listing 2.1: JSON format for the publication of the average temperature

```
{
  "idBuilding": 1,
  "idRoom": 12,
  "idDevice": 28763,
  "temperature": 23.4,
  "timestamp": 45
}
```

In the real case we expect the temperature to be sampled every 15 minutes and sent every hour. In the simulation, we set the sampling times every 5 seconds, and the sending every 10 collected samples.

However, the server also acts as an actuator and therefore it is the client, in this case, that sends it a temperature value (the average of the 10 user preferences). This value has not been encoded, and is a normal real value.

The server that samples the noise uses this schema for publishing the noise value:

Listing 2.2: JSON format for the publication of the average noise

```
{
  "idBuilding": 1,
  "idRoom": 12,
  "idTable": 38617,
  "value": 67,
}
```

In the real case we expect the noise to be sampled every minute and a half and sent (the average obtained) every 5 minutes. In the simulation, the server samples every 2 minutes and sends every 15 minutes.

The server that monitor the state of a seat uses this schema for publishing the state for a seat value:

Listing 2.3: JSON format for the publication of the seat state

```
{
  "idBuilding": 1,
  "idRoom": 9,
  "idDevice": 12214,
  "state": 1,
}
```

In the real case we will consider the seat as "occupied" (state 0) if the PIR detects a sufficient number of movements in 30 seconds. We will consider the seat as free (state 1) if the PIR does not detect any movement for 15 minutes. This last time has been designed to allow the user to take a break without fear of losing his seat. In the simulation we used a threshold of 7 movements to be overcome in 20 seconds to occupy the seat, and a threshold of 10 movements to be overcome in 30 seconds to keep the seat occupied.

2.4 Device Settings

In our project we have set a series of specific settings to fix some problems. In particular we will describe the following problems: Time synchronization, Max Age setting and Chunk size.

- **Time Synchronization:** as described in the previous section, in the messages about the temperature and the noise is present the information about the 'timestamp' in which the data is generated. This information cannot be taken directly by the sensors because they do not have the appropriate hardware. We understand that the available source of time is hardware dependent, and the library is hardware independent. Indeed when we tested the code with cooja, we did not have any problem about retrieving the timestamp information, instead, when we flashed the code in the sensors, we were not capable to retrieve this information and the flashing of code in the sensors failed. We resolved the problem in this way:

  1. The sensors of noise and temperature subscribe, through MQTT application protocol, to a topic, called "time_synch".
  2. The Java Application, when it starts, store the timestamp and public a message on the topic "time_synch".
  3. The sensors, when receive this message, store the "clock_seconds()" that represents the reference time on which the sensor can calculate the offset time. Indeed, in the field "timestamp" of the message, the sensor put the difference between the current clock_seconds() and the clock_seconds() stored when it received the message published by the Java publisher.
  4. In the Java Application, when the payload is extracted from the message, to compute the time in which the measure of temperature or noise was

sampled, it sums the value of "timestamp" extracted from the payload with the timestamp stored when the Java Application has been started.

When we store the timestamp value in the Java Application, we store the milliseconds from 01-01-1970.

- **Max Age Option:** in the Coap Server we modify this option so that the server does not send informations about the status of a seat if it has not been modified. This change was made to prevent the server from sending packets that are not useful for the purpose of monitoring seats.
- **Chunk Size:** we modified the *REST_MAX_CHUNK_SIZE* setting, presents in the project-conf.h file of Coap Server, because we need to increment the size of chunk to send the entire information.

Chapter 3

Data Storing and Visualization

In this chapter we will talk about how the data was stored and then retrieved for their graphical visualization.

3.1 MySQL

All the data collected by the client is stored on the SmartStudyRoom database using the well-known relational database management system MySQL.

The SmartStudyRoom database consists of two tables:

- **TEMPERATURE**
  1. *id* (int(11)): self-generated, uniquely identifies each sample.
  2. *timestamp* (timestamp): represents the time in which the data was sampled..
  3. *idBuilding* (int(11)): uniquely identifies the building that houses the study room in which the data was sampled.
  4. *idRoom* (int(11)): identifies the room inside that building in which the data was sampled.
  5. *idDevice* (int(11)): identifies the device involved in sampling the data.
  6. *temperature* (double): the value of the data sampled.
- **NOISE**
  1. *id* (int(11)): self-generated, uniquely identifies each sample.
  2. *timestamp* (timestamp): represents the time in which the data was sampled..
  3. *idBuilding* (int(11)): uniquely identifies the building that houses the study room in which the data was sampled.
  4. *idRoom* (int(11)): identifies the room inside that building in which the data was sampled.
  5. *idTable* (int(11)): identifies the table involved in sampling the data. In reality, in our simulation, this id corresponds to that of the PIR.
  6. *noiseValue* (double): the value of the data sampled.

7. *noiseLevel* (varchar(255)): the band to which this noiseValue belongs. The possible values are: MINIMUM (0-15 db), LOW (15-45 db), MEDIUM (45-60 db), HIGH (from 60 db onwards).

3.2 Grafana

We used Grafana, a web application for interactive data visualization and analysis. In particular, we have created two dashboards to report the data stored in the two tables described in the previous paragraph.

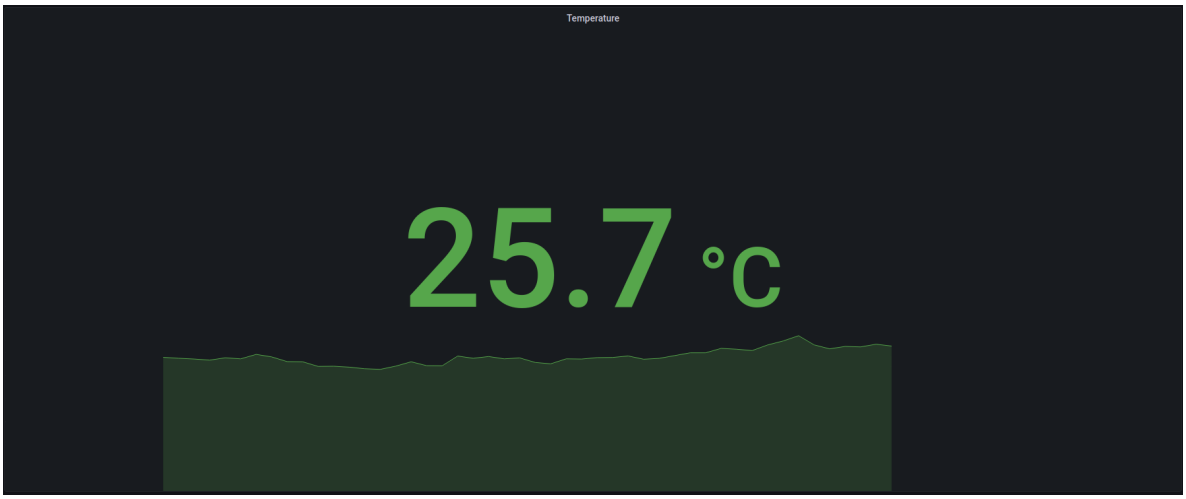

Figure 3.1: The trend of temperature over time. In particular, the last value sampled and stored by the client is shown.

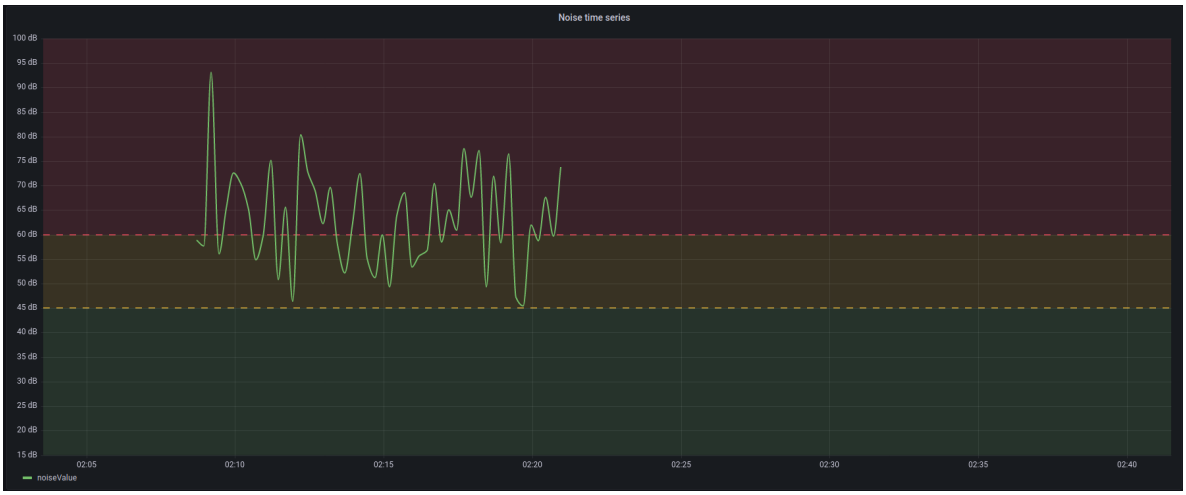

Figure 3.2: The trend of noise over time.

3.3 Graphic User Interface

The client is a Java app that collects data sampled from the various servers in the IoT network. To allow a correct visualization in real time, we have created a friendly graphical interface that shows the data.

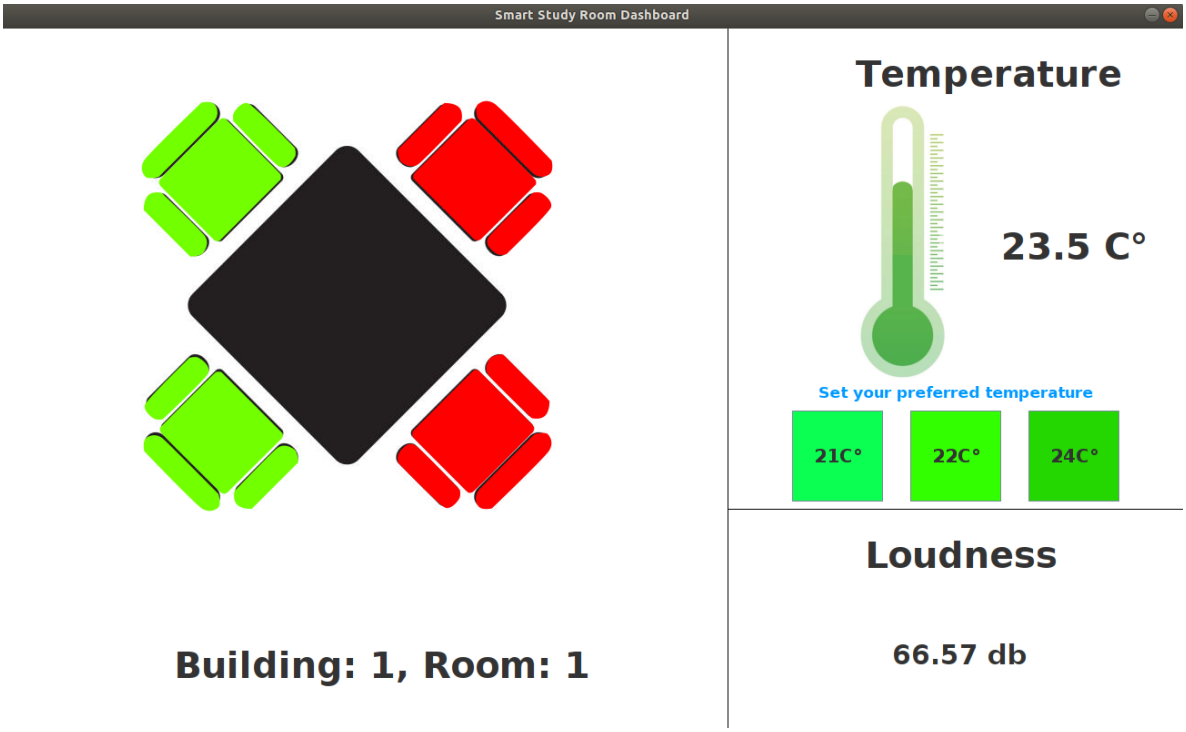

Figure 3.3: The graphical interface of the Java client. The panel on the left shows the 4 seats of the table he is monitoring, of which two have been occupied (in red), while the remaining two (in green) are still free. In the panel on the right we have at the top the room temperature (20 Celsius degrees), while at the bottom the current degree of loudness (80 decibels).

The temperature display panel allows the user to select their desired temperature. To prevent one person from choosing for everyone, a collection of 10 preferences is expected before actually sending the setting request to the server. Indeed, the average of the 10 preferences will be sent.

In our simulation, having a limited number of sensors available, we monitored only one table, but nothing prevents us from creating, for a large-scale production project, an application that allows us to monitor a particular study room, showing several tables, of a particular building.