PhyXIT project
Final report

Distributed Interaction

Subject's teacher: Ph. Philippe Truillet
Students: Sara Messara - Clément Pagès

January 2023

Table of contents

List of Figures and Tables

1. Introduction

The objectives behind the PhyXIT project are to implement a solution that allows the integration of both physical and virtual sensors, centralize the data on an online server and provide display by a graphical interface.

In the scope of this report we will present the hardware and software application proposed and implemented by our team as a solution to the imposed requirements.

The hardware application integrates 4 physical sensors (temperature, humidity, range and presence) wired to an ESP32 board. The communication to the online server was achieved using the MQTT protocol. Data was converted to Json objects before being published on the UPSSITECH broker @51.178.50.237 under the topic name "esp32/sensors".

The software application is composed of 2 layers. The first layer integrates virtual sensors which retrieve the real time weather condition of a desired localisation obtained using Visual Crossing API. The second layer subscribes to the MQTT broker to retrieve the data collected by the ESP32 sensors. Finally, the display of data was made possible through a Discord Bot, which sends data whenever requested by the user by commands. This solution combines both the benefits of sensors and API data centralisation as well as the functionalities of the chat server.

For the physical sensors, we imagined a scenario of house monitoring. We used indoor humidity and temperature sensors whose statistics can be used for instance to automatise the scheduling of the heating system of the house. The range sensor can be used to detect a near presence around a place, for instance, to prevent children from getting close to dangerous places such as an operating stove or electrical outlets. Our solution is then suitable for domotic applications.

2. Application design

a. Application architecture

The general architecture of the application showing the link between elements is described in the following figure.

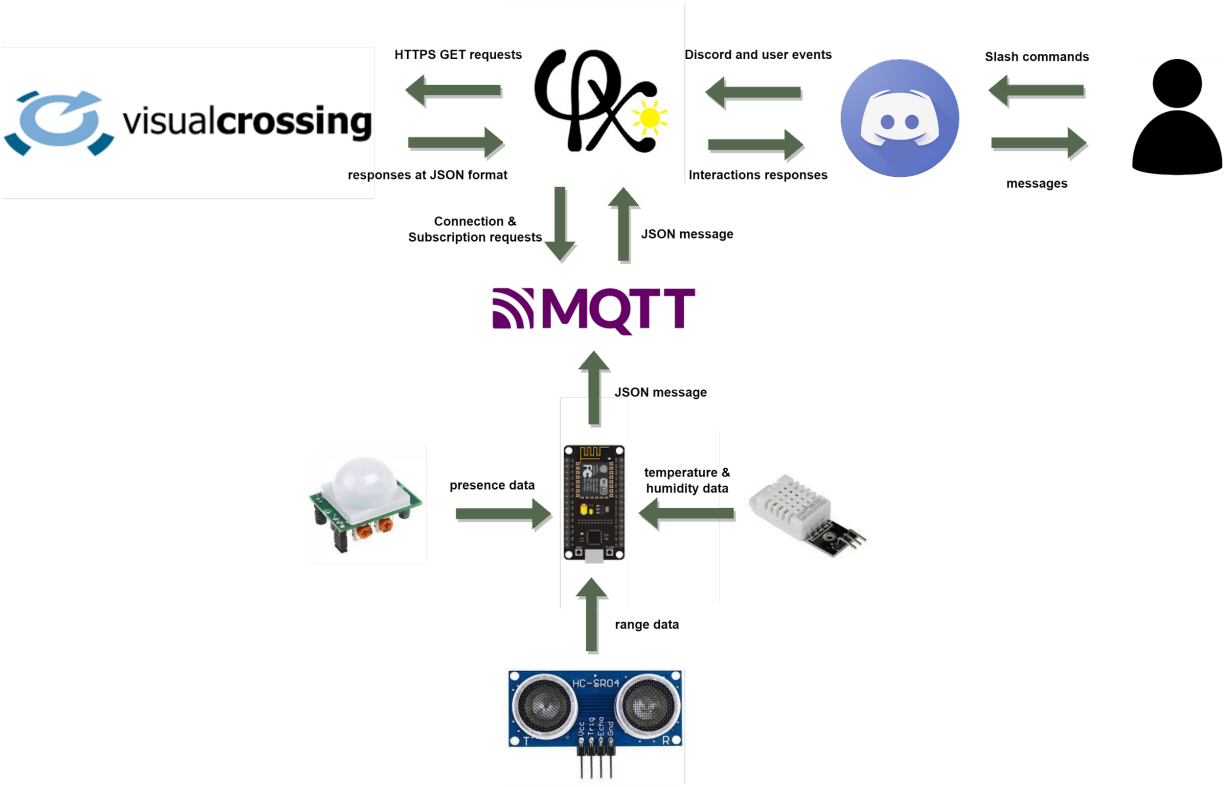


Figure 1: General application architecture

b. Real sensors integration

i. Hardware wiring

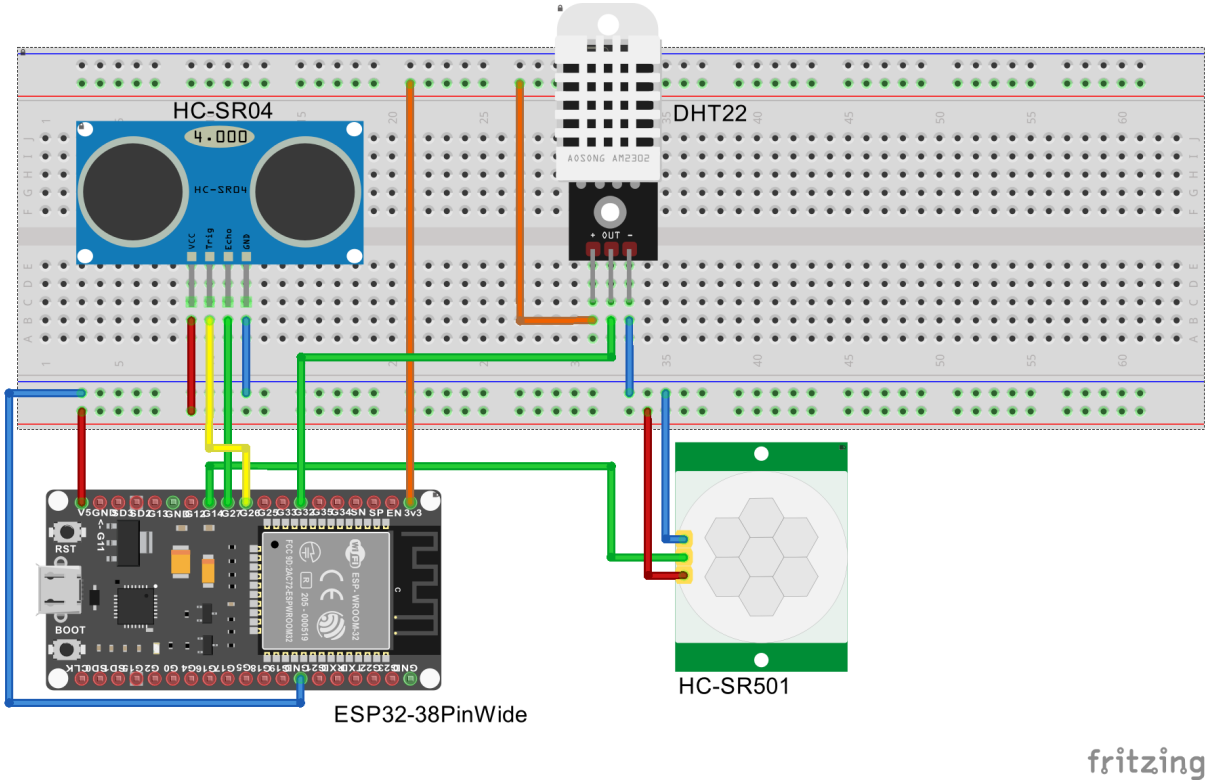


Figure 2: Hardware wiring for ESP32 part

In the scope of this project, 4 sensors have been integrated:

- DHT22: outputs 2 measures, namely, the ambient temperature as well as the humidity. Both data are of type real (float).
- HC-SR04: measure the linear distance (in the frontal direction only) of an object to the sensor. The return value is a float.
- HC-SR501: detects the presence of an object in a perimeter of 3 to 7 m (in all directions). The return value is a boolean.

As a microcontroller, we used the ESP32-38PinWide board.

ii. Challenges regarding sensors data handling

There have been several challenges regarding data collection and storage from sensors listed as follow:

- Sensors output heterogeneous data types, for instance, the presence detector outputs boolean information whereas the DHT sensor gives real data types. This poses a cast problem if stored in the same structure.
- The scale and the precision of data are different, boolean holds on a bit while data of real type have a double precision. As data is operated on an embedded board, the memory size must be well controlled and optimized.
- The C++ Standard Library does not offer a stack structure suitable for embedded systems. The “std::deque” is not suitable as it does not allow fixing the stack size at the start. In

practice, this is problematic since the embedded system has limited memory resources, thus the stack that aggregates data must be limited in size since creation.

iii. Code presentation and communication

In order to implement a stack structure compatible with embedded programming, a Fifo template class has been coded from scratch. It allows the creation of a stack containing any data type (template) with a fixed size. Unlike the C++ Standard Library “deque”, if the stack is full, the oldest data is overwritten by the new entries, which prevents extending the allocated memory. The Fifo code is this link and can be used for a broader range of projects.

To handle the heterogeneity in sensors’ data, an Arduino Library have been created, named, SensorData; a double pointer attribute of Fifo type allows to aggregate a table of Fifos for each sensor that can be from different types (in our application, there are 3 sensors; humidity - float, temperature - float, range - float). Note that the constructor sets the number of sensors at creation as well as the size of the FIFO for each sensor.

In order to control the precision of data (especially for the float), all data are casted into integers then converted to a Json structure. This solution is relevant for the following reasons:

- During testing, MQTT showed limitations in tram size, if sending crude float/double, the precision of the floating point is not controlled and the tram size might explode (as experienced during testing). We propose to keep a precision up 10^-3, the number is then multiplied by 10^3 and casted into int in the subscriber side to recover the original value. This means we fully control the max size of the Json structure and thus the MQTT tram.
- The second reason is to make the data homogeneous. Although we created a template data structure that can handle any base as well as complex type, it is more convenient to manipulate int data then several types of data, it is also more efficient regarding memory.

The Json object is constructed to scale to a variable number of sensors. The chosen structure is given as follow:

```
{   Server : “ESP32”,
    Localisation : “Home”,
    dTypes : “f-f-f-”,
    Sensors : {
            Temperature : [2536, 2698, …],
            Humidity : [1451, 1542, …],
            Range : [2544, 2588, …],
            … }
}
```

The “Server” field refers to the name of the device, in our case, an ESP32 is used.
The “Localisation” field refers to the current localisation of the reception device.
The name of the sensors is given as a variable which allows its modification (Humidity, Temperature, …).
The size of the array is defined by the size of the Fifo when creating the SensorData object.
The dTypes refers to the type of the data contained in each Json array; “f” refers to “float”, “i” to “int” and “b” to “boolean”. Specifying the type helps recovering the data at the subscriber side.

For communication, the MQTT protocol is used. The Json object is converted to a String and then published each 10 min (imposed time interval) in an MQTT broker (we choose the UPSSITECH broker @51.178.50.237). The topic can be subscribed to under the name "esp32/sensors".

The UML diagram is summarized in the below figure:

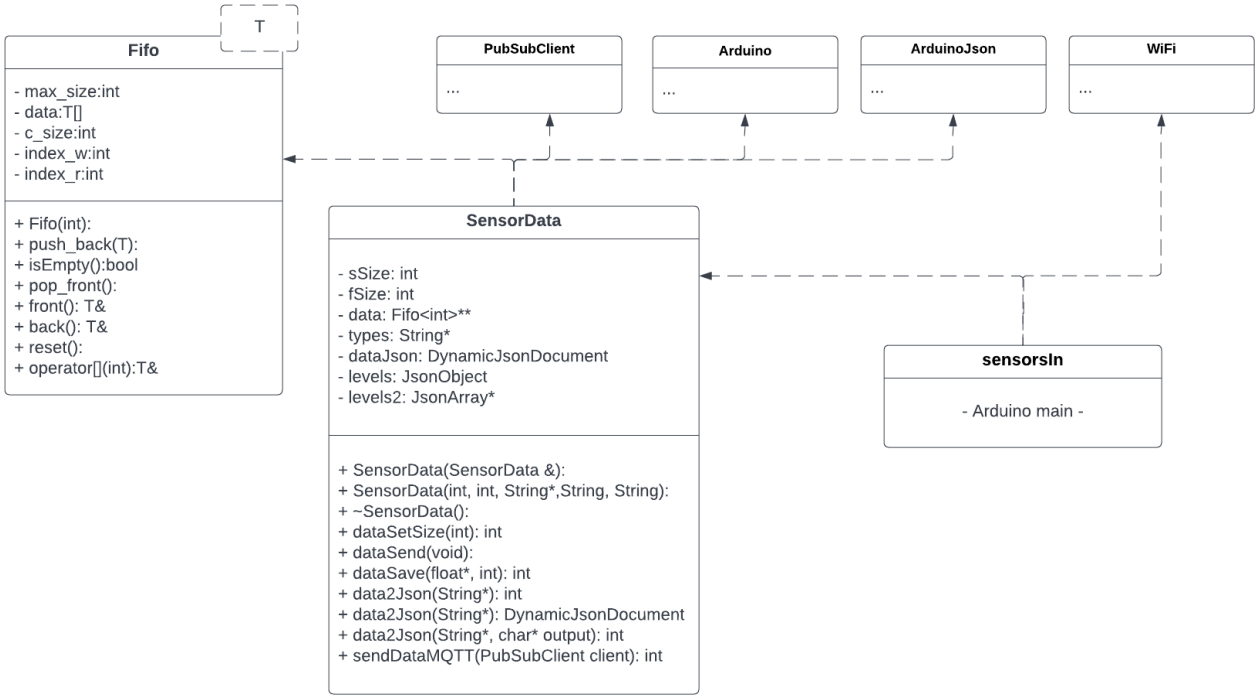


Figure 3: Hardware side UML diagram

c. API retrieving

One of the features of our application is to allow the user to automatically retrieve the weather for any location in the world. To do this, we use the API of Visual Crossing. The API and how we get data from it is presented in this part.

i. Visual Crossing: weather data and API

Visual Crossing is a weather data provider founded in 2003 that allows the user to access a wide range of data, including historical weather data, current weather conditions, weather forecasts and alerts, and climate summaries for most cities around the world. For our system we only get current weather conditions for specified cities by the user. One of the particularities of Visual Crossing compared to other similar services is that it offers a free plan to use its API. Another API that gives free access is OpenWeather, but it is more limited in terms of available data. The free plan permits the user to call the API up to 1000 times per day. Because of this limitation, and with an automatic update of the weather for a city every 10 minutes, the maximal number of locations for which the weather is retrieved is fixed to 6. This limitation is encoded in the application. The way in which the weather data is obtained is described in the following section.

ii. API handler

In order to retrieve current weather data for a specific location, we have developed an API handler whose aim is to generate a HTTPS request for the indicated city and to process the response from the Visual Crossing's API. To build a request, there are mainly two things needed: the name of the location for which we want to retrieve current weather, and a token for authentication. An example of a query is given in figure 4.

```
https://weather.visualcrossing.com/VisualCrossingWebServices/rest/services/timeline/Toulouse/today?unitGroup=metric&include=current&key={os.environ['VC_TOKEN']}&contentType=json&lang=id"
```

Figure 4: Example of HTTPS request to the Visual Crossing API

Thus, this request allows us to get current weather data for Toulouse. The response from Visual Crossing is in JSON format and the language used is "id", which grants access to the current weather code. For example, type 43 indicates that it is currently sunny, and type 21 is for rainy conditions[1]. For security purposes, the token is hidden in an environment variable.

[1] All weather conditions type is available here

The JSON response from the API contains data about current weather data for indicated location, but also information about the location and how the data was acquired. In most of the cases, data is provided by one or more weather stations, which can be seen as a group of real sensors (thermometer, barometer, anemometer, humidity sensor, rain gauge and so on). So data in the JSON response is an aggregation from these stations.

When a response to a query is received, the API handler generates a data structure that keeps only the relevant information from the JSON response. This data structure is a dictionary whose keys are the weather element, and the value measure for this element. The weather elements retrieved are:

- Current absolute and feels like temperature
- Probability of precipitation and rain or snow fall
- Wind speed and direction, gust speed
- Atmospheric pressure
- Visibility and nebulosity
- Humidity rate
- UV index
- Current weather type

The data structure is then used to generate an image of the current weather conditions for a location. Afterwards, the image is sent to a discord channel using the Discord API.

d. Discord graphical interface

In order to present the acquired data from the ESP32 sensor or from the Visual Crossing API, we have decided to use Discord as a graphical interface. Indeed, through the discord API and a bot application, it is easy for a user to get wanted data without any installation of a software. So the user can mix chat activities, get weather data for her/his location and information and alerts from his / her home. Moreover, Discord is a cross-platform application and can be installed on Smartphone, Iphone, tablet, computer (Windows, Mac, Linux…) or just used in a navigator. The implementation of our application into Discord is described in this part.

i. PhyXIT bot: a Discord application

With the aim to integrate our application to Discord, we have developed a Discord application bot named PhyXIT bot. A Discord bot is an entity that acts in the same way as a user. So a bot takes part in a Discord server, like human users. Depending on its authorisation, it can write and edit messages, answer to other users, send files and images or react to messages. To do these actions, the bot source code interacts with the Discord API, which grants access to all the entities on Discord such as users, channels, guilds (servers) or other bots. That API allows also to get events in the program, for instance when a message has been sent by a user on a guild to which the bot belongs. A specific message that the user can send is command. A command is a way to indicate the

bot to do something, for example retrieve weather data for a location. In Discord, there are two kinds of commands: prefix and slash commands. As their name suggests, prefix commands use a specific prefix at the start of a message to indicate that this message is a command. After the prefix, the user gives a keyword that corresponds to the command name, with zero, one or more arguments in the following. Slash command is a command which starts with a "/" character. This type of command is further described in the following section.

ii. Slash commands and autocompletion

To interact with the PhyXIT bot, we have decided to implement slash commands because that type of command is more usable and easier to handle for the user than the prefix command. In fact, when a user types a "/" character in a text channel, Discord provides him with the list of available slash commands that he can call. Here is an example of what the user sees:

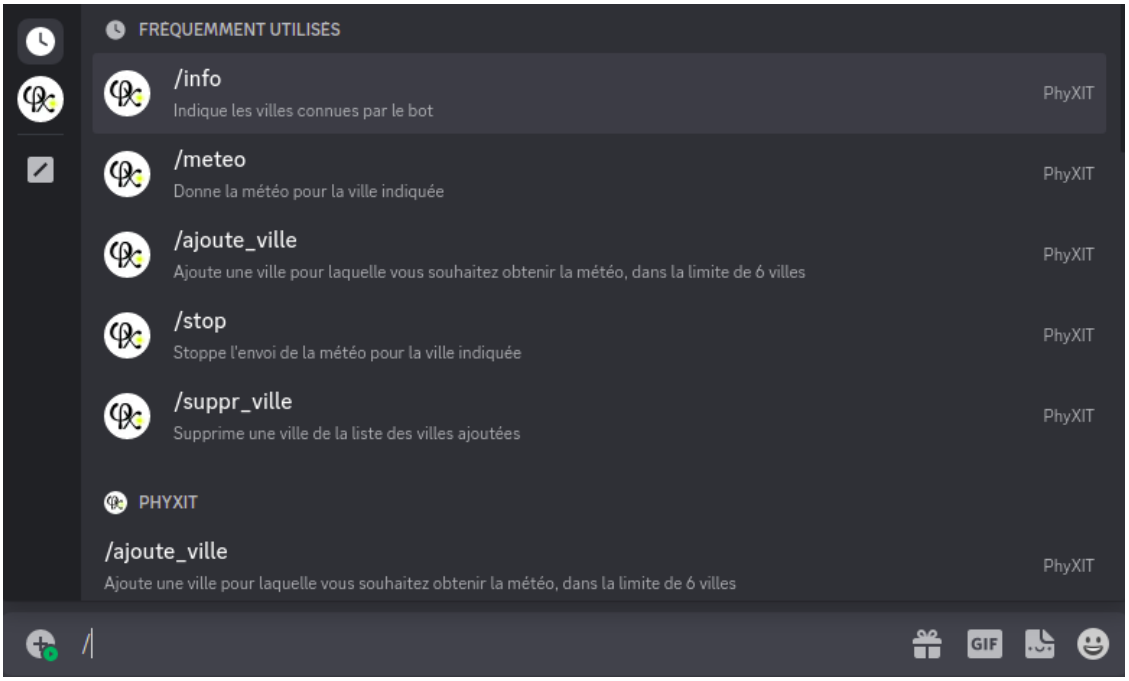
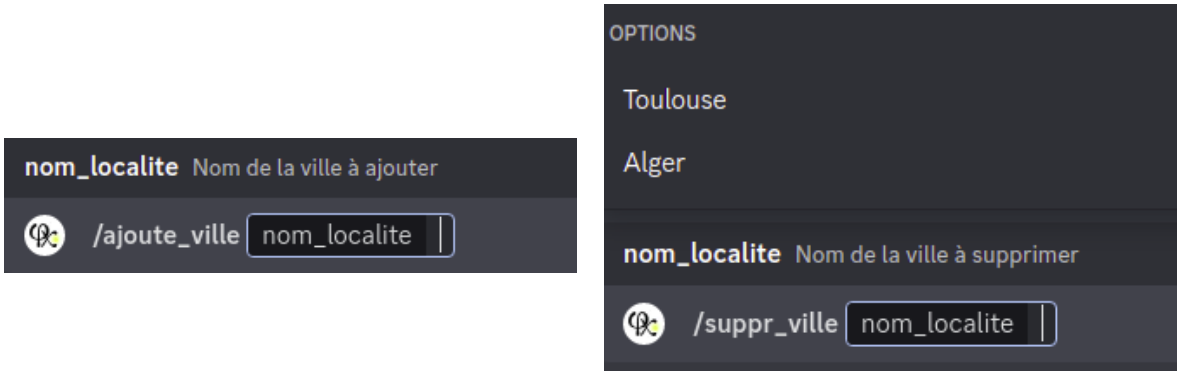


Figure 5: User interface on Discord when typing "/" in a text channel

On this screenshot, we can see the list of available slash commands for the PhyXIT bot. For each command, a brief description of what the command does is given. Because the application is mainly designed for French users, this description is in french.

The user can then select the command he wants to execute or continue to type the name of the command. After that, if the command needs one or more arguments, the Discord interface lets the user fill them with the appropriate value. For each parameter, the interface indicates the role of the argument, as shown in figure 6 (A). According to the selected command, an autocompletion is proposed. The user can then select the value for the parameter from proposed ones (B).



A: User interface when filling command argument

B: User interface with autocompletion for current argument

Figure 6: : User interface on Discord when filling slash commands arguments

In the example above, the interface indicates to the user that the *ajoute_ville* needs one argument: *nom_localite* and lets the user type a value. On the right screenshot, Discord interface proposes a list of possible city names which correspond to the city that can be deleted, because they were previously added to the bot.

In response to the call of a slash command, the bot provides feedback to the user according to the success of the execution of the command. The figure 7 shows some bot responses.

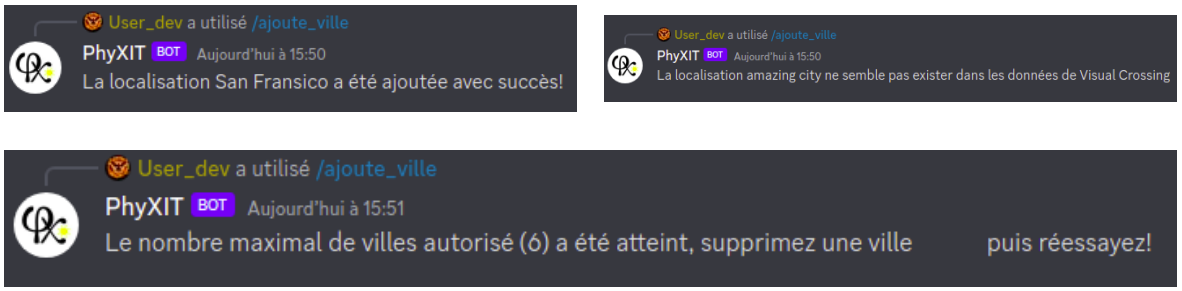


Figure 7: Different feedback sent by the bot for the slash command "ajoute_ville"

iii. Generation of weather images and real sensors data graphs

The response to the call to of a slash command can also take the shape of an image or a graph. This is the case for *meteo* and *esp32* slash commands (see commands user guide for more details on how to use it). Example of image returned by a call to the "meteo" slash command is given in the following figure:

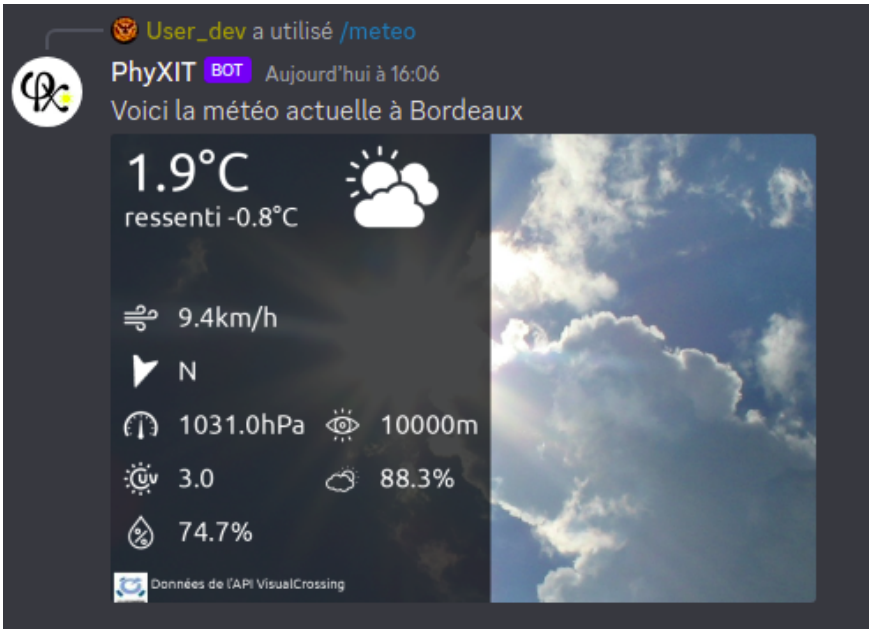


Figure 8: Example of an image representing current weather condition

In this image, we retrieve all the previously presented weather elements, such as temperature, wind speed and direction and so on. This image is automatically updated every 10 minutes with new data for the current location, until the user calls the *stop* slash command. The update sends a new image and removes the previous one, so the image remains at the bottom of the channel. Another idea was to update the weather image by editing the first image, but Discord does not allow editing a message with a new image.

In the same way, when the user calls the *esp32* slash command, the bot returns a figure representing variation according to the time for the specified data type. An example of a graphic for humidity data is shown in figure 9. The user can also retrieve data from all the sensors, using the "all" argument. The result is shown below .

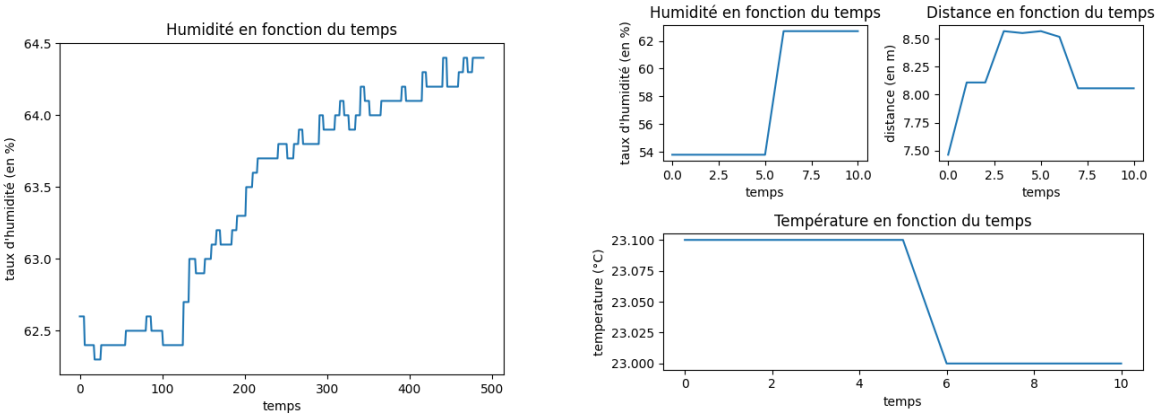


Figure 9: Examples of figures returned by the bot for humidity data (left) and "all" data (right)

e. Further ideas

Here are some ideas for going further :

- Add personal user space for saving the preferences of each user. This is not possible for the current version as the data is centralized on the bot and not on an external database
- Smart Home for monitoring the evolution of the internal house temperature for planning the central heater functioning
- Offers alert notification on Discord in case of presence detection

3. User guide

a. Prerequisites

Before using the ESP32, it must be configured to connect to a WiFi network since the MQTT broker is not local. The name and password of the network must be changed in the source code. The MQTT broker can be chosen differently, note that the IP of the UPSSITECH broker might vary, it must then be reconfigured in the source code of the ESP32.

Before building the project in Arduino, the SensorData library must be added to the Arduino Library folder on the local computer. Otherwise the dependencies will not be found and the compilation will crash.

There is no prerequisite for the use of Discord, the only condition is to be invited to join the server.

b. Discord commands

Descriptions for every Discord commands that the user can call are provided in the following tables. For each command, the table indicates what the command does and returns, how to use it and the description of the expected arguments.

i. Ajoute_ville command

Command name	English name	French name
	Add_city	Ajoute_ville
Command description	This command allows you to add a city to the bot, within the limit of 6 cities, because of limitations of Visual Crossing free plan. The bot checks if the location was not already added and if	

	it is known by the weather API. Finally the bot gives feedback about the success or not of the addition.	
Usage	/ajoute_ville <location_name>	
Expected argument(s)	Argument name	Description
	Nom_localite / location_name	Name of the location to add
Return	Return type	Return description
	Discord message sent by the bot	Feedback about the addition

Table 1: ajoute_ville command

ii. Suppr_ville command

Command name	English name	French name
	del_city	suppr_ville
Command description	This command removes a city from the bot list of known cities. The bot indicates to the user if the city was successfully removed or not. In particular, this operation fails if the city is not in the bot data. To avoid that, the user can use the proposed autocompletion which gives all saved locations.	
Usage	/suppr_ville <location_name>	
Expected argument(s)	Argument name	Description
	Nom_localite / location_name	Name of the location to remove
Return	Return type	Return description
	Discord message sent by the bot	Feedback about deletion

Table 2: suppr_ville command

iii. Meteo command

Command name	English name	French name
	weather	meteo
Command description	The weather command allows the user to retrieve weather data for a previous added location (using add_city command), in the	

<table>
<tr><td></td><td colspan="2">form of a weather image. This image contains data about current weather conditions, namely:
Current absolute and feels like temperature
Probability of precipitation and rain or snow fall
Wind speed and direction, gust speed
Atmospheric pressure
Visibility and nebulosity
Humidity rate
UV index
Current weather type
The weather image is then updated every ten minutes, until the user uses the stop command. The update removes the old message sent for the location. The bot informs the user if the specified location is not known in the bot data. The user can use proposed autocompletion to avoid mistakes.</td></tr>
<tr><td>Usage</td><td colspan="2">/meteo <location_name></td></tr>
<tr><td rowspan="2">Expected argument(s)</td><td>Argument name</td><td>Description</td></tr>
<tr><td>Nom_localite / location_name</td><td>Name of the location for which the user wants get current weather conditions</td></tr>
<tr><td rowspan="2">Return</td><td>Return type</td><td>Return description</td></tr>
<tr><td>An image representing current weather or a message from the bot in case of fail</td><td>If the operation succeeded, an image representing current weather conditions for the specified location, a bot message explaining why the operation failed otherwise.</td></tr>
</table>

Table 3: meteo command

iv. Stop command

<table>
<tr><td rowspan="2">Command name</td><td>English name</td><td>French name</td></tr>
<tr><td>stop</td><td>stop</td></tr>
<tr><td>Command description</td><td colspan="2">The user can use this command to stop the automatic sending of the weather image (see weather command) for the specified location..</td></tr>
<tr><td>Usage</td><td colspan="2">/stop <location_name></td></tr>
<tr><td>Expected argument(s)</td><td>Argument name</td><td>Description</td></tr>
</table>

	Nom_localité / location_name	Name of the location for which the user wants to stop retrieving current weather conditions or sensor graph.
Return	Return type	Return description
	Discord message from the bot	The bot reports whether it has successfully stop the sending of the weather or not

Table 4: stop command

v. Info command

Command name	English name	French name
	info	info
Command description	This command gives information about the cities known by the bot. In particular, it indicates whether the bot sends weather for each city or not.	
Usage	/info	
Expected argument(s)	Argument name	Description
	-	-
Return	Return type	Return description
	A discord embed message	This embed message contains data about known cities and for each one, if the bot sends current weather conditions or not.

Table 5: info command

vi. Esp32 command

Command name	English name	French name
	esp32	esp32
Command description	With this command, the user can retrieve data from physical sensors connected on an esp32. More precisely, the user can get temperature, humidity and range data from the location where esp32 has been placed, using the proposed autocompletion. In	

	response to this command, the bot sends a graph corresponding to data produced by the esp32 sensors. The graph is then updated when the bot receives new data from the ESP32, until the user calls the stop command (see stop command).	
Usage	/esp32 <data_type>	
Expected argument(s)	Argument name	Description
	Type_donnes / data_type	Data type for which the user wants to generate a figure. If the “all” option is selected, the bot sends a figure that contains graphics for all data (temperature, humidity and range).
Return	Return type	Return description
	A figure as an image, or a feedback from the bot	If the command succeeded, the bot returns an image that contains a graphic for the specified data type. If the operation has failed, the bot gives feedback to the user.

Table 6: esp32 command

vii. Ping command

Command name	English name	French name
	ping	ping
Command description	This command checks whether the bot is alive or not, mainly for debugging purposes.	
Usage	/ping	
Expected argument(s)	Argument name	Description
	-	-
Return	Return type	Return description
	A discord message	The bot responds “pong!” if it is alive

Table 7: ping command

4. Conclusion

The objectives of the project have been successfully reached and resulted in the implementation of a hardware-software solution able to combine both physical and virtual sensors and to centralize them in an online server before being displayed at the request of the user in a graphical interface. In total, there have been 4 physical sensors integrated on the ESP32 board. The aggregation of data caused multiple challenges including memory storage and types heterogeneity. The latter have been tackled by creating a C++ template class (Fifo) which was used to construct an Arduino library (SensorData) able to handle the data coming from various sensors. Data is sent to the online server using the MQTT protocol. An MQTT client was created at the ESP32 level and connected to the UPSSITECH broker. In order to send the data, the SensorData object is converted to a Json object. At the sending time (each 10 min), the ESP32 client publishes the Json object (converted to String) in its topic.

Virtual sensors were represented by the online open APIs. For this project, Visual Crossing API has been used to retrieve information on the real time weather condition in a desired localisation specified by the user.

The user can easily access all this data via the Discord server, using a bot designed specifically for the application, by inserting slash commands. An auto-completion functionality was also added for a better user experience. Using Discord eliminates the need for the user to install a third-party application and also allows the user to access all of their data from any device that can host Discord, while still being able to use the other features provided by the server.

5. Github link to the project

https://github.com/SaraMess/PhyXIT