Embedded Systems and Internet-of-Things
-
Third Assignment

Kimi Osti

May 23, 2025

Contents

Chapter 1

System Requirements

The system is a smart IoT-based temperature monitor. In particular, it measures a closed environment's temperature at any given time, and controls a window connected to a motor to properly ventilate the room in case of critical temperatures. It also implements a manual mode, which can be activated via a button on the *Window Controller*, or via a web-based *Operator Dashboard*, that allows an operator to physically control the window opening angle thanks to a potentiometer attached to the *Window Controller*.

1.1 Temperature Monitor

It's the main system component. It periodically samples the room's temperature, and communicates it to the *Control Unit*, which is responsible of controlling the other component's behavior according to the value recorded by this subsystem. The frequency of the measurements depends on the state of the system, which is stored in the *Control Unit* and communicated to this component in real-time. It communicates with the *Control Unit* via the *MQTT* protocol, and must include a LED signaling whether the connection is properly established.

1.2 Window Controller

It's the component which physically controls the window movement. Its opening level is actuated as indicated by the *Control Unit*, and it depends on the temperature when in automatic mode. It also exposes an in-place operating dashboard, consisting of an LCD screen to check some info on the system at any time, a button to switch between manual and automatic mode, and a potentiometer to control the opening level when in manual

mode. All information exchange with the *Control Unit* is based on *Serial Line* communication.

1.3   Operator Dashboard

It's a web-based user interface that allows operators to work remotely on the system. It shows a graph representing the current state of the system and a brief history of the last measurements, sided by a statistic showing the average, minimum and maximum values in the last period of time. In addition to this info, it exposes a simple operator interface which allows the user to switch between manual and automatic mode, as well as to restore the system status in case an alarm was triggered. It communicates with the *Control Unit* via the *HTTP* protocol in order to reflect user actions on the actual in-place subsystems.

1.4   Control Unit

It's the core of the entire system, and it serves as a mediator between subsystem interactions. Its main function is to track the system state and all info related to the measurements and to the current operating mode (manual or automatic). It also determines the sampling frequency for the *Temperature Monitor* according to the current measure, and the window opening percentage according to the system state (if in automatic mode). It's also responsible of storing all system data, including the last measurements that the Dashboard shows in its graph, and the average, minimum and maximum values that are shown to the operator.

Chapter 2

System Architecture

The system architecture can be analyzed splitting the system in the four main sub-systems highlighted previously.

2.1 Temperature Monitor

This subsystem's main feature is sampling the room's temperature with a given frequency. It's also responsible for sending the collected data to the *Control Unit*, so it must include a component handling the subsystem connectivity to the back-end.

Its behavior can be modeled in four Tasks running in parallel, that can all be modeled as Finite State Machines. In particular, all Tasks can be modeled in a synchronous way, running at a fixed period that dictates their steps.

2.1.1 Temperature Measuring Task

The central Task is the one responsible of actually measuring the temperature. Its main feature is the variable execution period, here depicted with the *freq* parameter. In particular, the frequency is dictated by the *Control Unit* in response to the measured temperature.
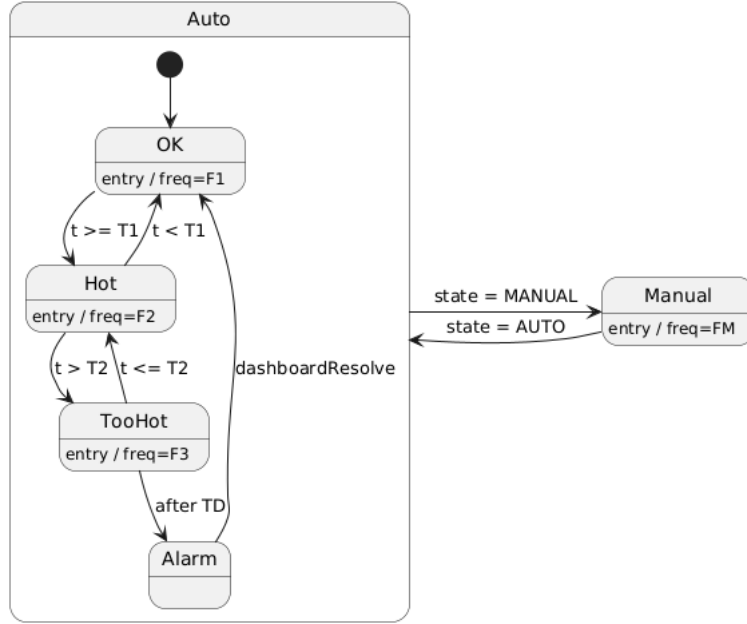
Figure 2.1: FSM modeling the temperature measuring task behavior

It is clearly depicted how the state transitions in automatic mode are determined by the temperature values. But in reality, all state transitions are demanded by the *Control Unit* according to the values that it receives from all components, since it is the central Controller for the application.
Also, it is clearly modeled to be network-agnostic. Indeed, this Task is supposed to measure the room's temperature at all times, and relevant data is retrieved and shared by a network-related Task, also responsible of resolving connectivity issues.

2.1.2 Connection Monitoring Task

This task is the one responsible for monitoring the subsystem connection state. It's supposed to check whether the component is still connected to the *Control Unit*, and when it's not, to try and reconnect. It works on two layers: the first one checks whether the system is online, and the higher one checks whether the *MQTT* subscription is still functioning.
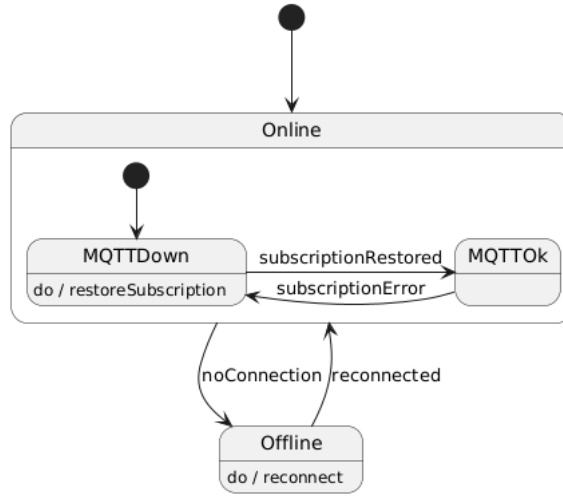
Figure 2.2: FSM modeling the connection monitoring task behavior

2.1.3 Communication Task

This task is responsible of communicating with the *Control Unit* via the *MQTT* connection set up by the *Connection Monitoring Task*. It ensures that data is properly collected, assembled to form a message and sent to the *Control Unit*. On the other hand, it's also responsible of receiving the response messages published by the *Control Unit*, which can dictate the *Temperature Measuring Task* state, and consequently its sampling frequency.
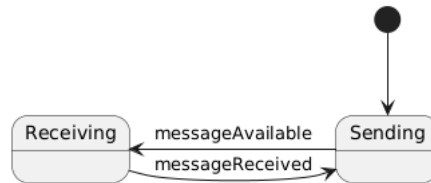

Figure 2.3: FSM modeling the communication task behavior

It's here clear how this task too is network agnostic: simply, when there is no connection or no active *MQTT* subscription no messages are received - and the system therefore never goes to the *Receiving* state - and the ones that are supposed to be sent are actually lost.
Additionally, it's noticeable how this Task has at least a part that runs asynchronously: whenever a message is received, regardless of how much time has elapsed since the last one, a response is triggered in this Task. But this is not an issue since the Task is only meant to receive messages in

response to the ones it sends: so the message receiving can be considered as a part of the single step of a synchronous Finite State Machine running at the same variable period of the *Temperature Measuring Task*.

2.1.4 LED Task

This task's main responsibility is to represent to the user whether the subsystem is online.

Figure 2.4: FSM modeling the LED task behavior

In this scheme, it's obvious how this Task's state depends directly on the state of the *Connection Monitoring Task*. In particular, the LED is shown as green only when the device is connected to the Internet and the *MQTT* subscription is properly established.

2.1.5 Shared Data

In this subsystem, shared data is used to ensure consistency in the behavior of different Tasks. In particular, it's used to demand state transitions to Tasks that depend on other Tasks' state.
To achieve inter-Task communication, all Tasks store a reference to a shared object. Its main feature must be thread-safety: all Tasks can access shared data at any time, also concurrently, so the shared data carrier has to be implemented in such a way to prevent dirty reads, race conditions and deadlocks.

2.2 Window Controller

This component's main responsibility is to actuate the window opening level, based on the values sent by the *Control Unit* via *Serial Line*. Furthermore, it exposes a small operating dashboard to switch between operative modes (automatic or manual) and to control the window's opening level, if in manual mode. Its behavior can be modeled with a task-based architecture, giving

each task a small set of features to handle. Each one of these tasks can be modeled as a *synchronous Finite State Machine*, timed by a central scheduler whose only responsibility is to keep everything temporized.

2.2.1 Window Controlling Task

This Task is the one actually responsible of actuating the window opening level. It has a period of 200ms and its only job is to set the window opening percentage as demanded by the *Control Unit*, regardless of the overall system state.
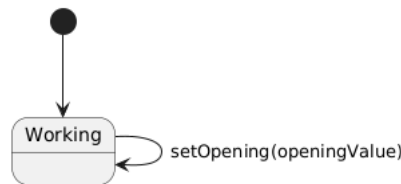

Figure 2.5: FSM modeling the window controlling task behavior

It's obvious in this representation how this *Finite State Machine* actually only has one state, and repeatedly sets the window's opening level based on the system's demands.

2.2.2 Operator Input Task

This Task is responsible of collecting user input, and of communicating it to the *Control Unit* to modify the entire system's state accordingly. It has a period of 100ms: it is a time short enough to never lose button pressing events, but long enough to avoid *bouncing* side effects when pressing the button, which might lead to unpredictable behavior.
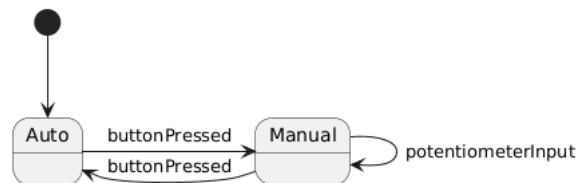

Figure 2.6: FSM modeling the operator input task behavior

In this scheme it's underlined how this tasks accepts operator input via the potentiometer only in manual mode. For better consistency though,

data is here not directly sent to the *Window Controlling Task*, but rather a decoupling level is inserted between the two, centralizing state handling responsibilities in the *Control Unit*.

2.2.3 Operator Output Task

This task is responsible of showing the operator some information about the system's current state. Obviously, for the reasons described above, the system's state is described as received from the *Control Unit*, delegating it all data consistency issues. It has a period of 200ms.
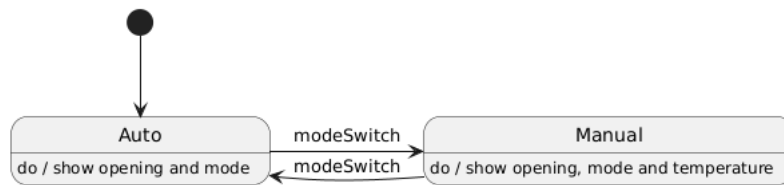

Figure 2.7: FSM modeling the Operator Output Task

In this scheme, it's depicted how this task is showing different information according to the state of the system. On the other hand, state transitions are not handled in any way by this task, and the *modeSwitch* depends on some input received by the *Control Unit*.

2.2.4 Communication Task

This task is responsible of communicating via the *Serial Line* with the *Control Unit*. It's a core feature of this sub-system, since correct communication ensures data (and more importantly behavior) consistency when translating computation into actuation. It has a period of 200ms.
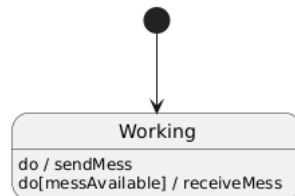

Figure 2.8: FSM modeling the communication task behavior

Each period this Task (which is clearly depicted as state-agnostic) sends a message containing this sub-system's demands for the *Control Unit* - in

particular whether the mode was switched and the last input opening level - and checks for a response's availability. Whenever an incoming message is completely received, it is analyzed by this Task, which stores the current state of the system as demanded by the *Control Unit*.

2.2.5 Shared Data

In this sub-system, shared data between Tasks is crucial for behavior consistency. For this purpose, two objects are exploited: a *Dirty State Tracker* which saves the current system input requests, and a *State Tracker* which saves the last *Control Unit* demands, considered to be the only source of valid data in the system.
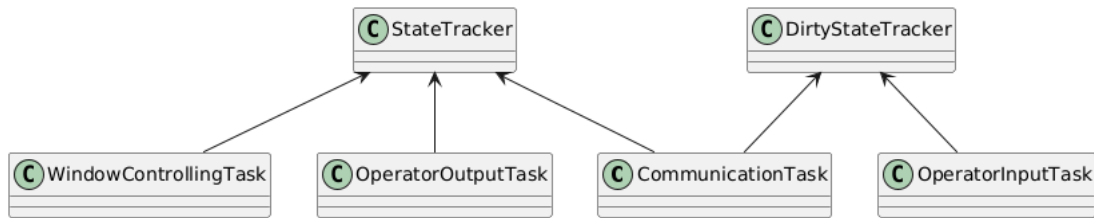

Figure 2.9: Dependencies between tasks and shared data

The two state tracking objects have to be *Singletons*, since they store the only valid sub-system state.

2.3 Operator Dashboard

The *Operator Dashboard* serves as a remote user interface to control the system. In particular it offers a graphical representation of the system's current state - which includes a graph of temperature history, the current average, maximum and minimum temperature values in the last period of time, the system operating mode and the window opening level. In addition to this, it allows the operator to switch between automatic and manual modes, as well as a button to restore the system after solving an issue which caused an alarm. It's connected to the *Control Unit* via *HTTP*.

2.4 Control Unit

This sub-system is the core of the application. It exchanges data with all other sub-systems, and ensures data consistency storing internally all relevant

values. Its main responsibility is to bridge between protocols to coordinate all system components, and to actuate the system's behavior on the real world.

This sub-system itself is made out of various components.
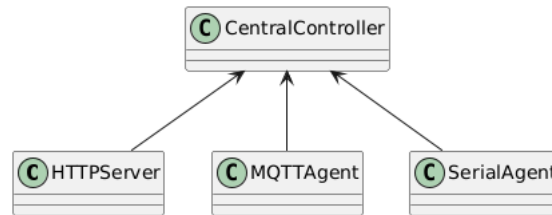
Figure 2.10: UML class diagram showing the architecture scheme for the Control Unit sub-components

In this scheme, the *CentralController* is the one responsible of ensuring proper data consistency, and it cooperates with all the other sub-components to properly communicate with each sub-system.

The main concern implementing this sub-system is to ensure that data is properly stored and can be safely accessed, preventing race conditions and other concurrency issues.

Chapter 3

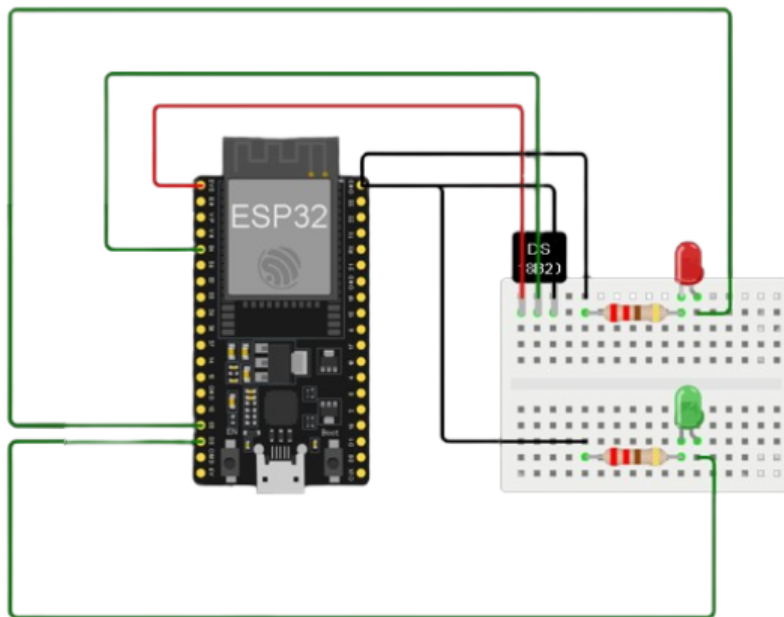# Implementing Solutions

3.1   Temperature Monitor

Figure 3.1: Detailed circuit for the temperature monitor sub-system

In this scheme si depicted the detailed circuit for the *Temperature Monitor*, deployed on an *ESP32 System-on-a-Chip* and programmed relying on the *Wiring* framework.
The *ESP32* holds enough program memory to host a lightweight Operating

System, which means that there is no need to implement a scheduler from scratch. For this project, *the FreeRTOS Operating System* was used, allowing to run Tasks in parallel on both cores of the *ESP32* board.

Specifically, both network-related tasks are pinned to *Core 0*. This implementation was chosen because, using *FreeRTOS* on *ESP32*, the *event Task* is automatically pinned to *Core 0*, and so all WiFi events are dispatched on that core. On the other hand, the *LED Task* and the *Temperature Measuring Task* are left unpinned, delegating to the OS the responsibility to balance load, making the usage of hardware resources as efficient as possible.

For what concerns communication, since the system relies on the *MQTT* protocol, the *PubSubClient library* was used. This particular library allows the programmer to subscribe to specific topics from a specific broker, without having to deal with protocol-specific low level aspects. The message content is represented in JSON format. In particular, the back-end periodically sends to this sub-system the requested sample frequency, and in response this component sends back the current temperature measure, together with the date and the time of the measurement, for history-keeping reasons.

To synchronize the ESP time with current date and time I relied on the *NTP Pool Project time servers*, as suggested by *this guide page*.
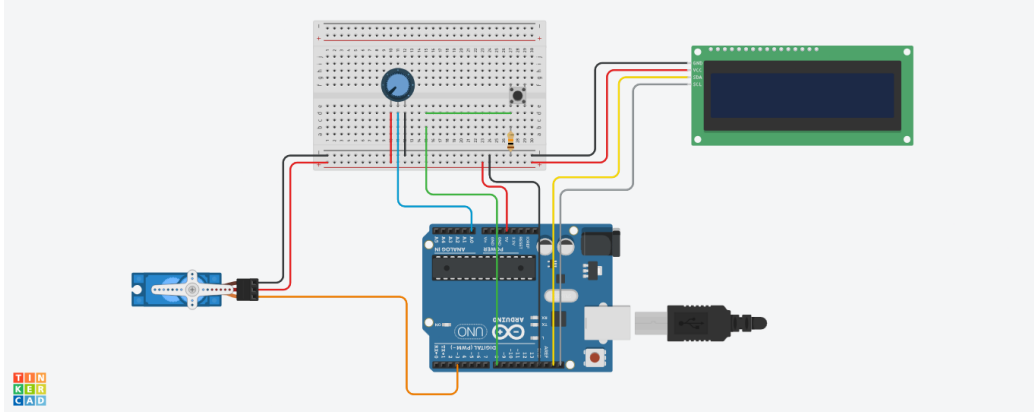
3.2 Window Controller

Figure 3.2: Detailed circuit for the window controller sub-system

In this scheme, available at *this link*, is depicted the detailed circuit for the *Window Controller*, deployed on an *Arduino UNO Board* and programmed relying on the *Wiring* framework.

Working on an *Arduino UNO Board* relieves the programmer from all kind

of concurrency-related concerns (which would be crucial in this environment, where all tasks work on the same two shared *Singletons*) since the Board is equipped with a single-core micro-controller.

On the other hand, one of the main concerns is to implement our own scheduler, since Arduino UNO doesn't support any kind of Operating System. This can be easily achieved implementing a *Cooperative Round Robin Scheduler*, since all Tasks have finite - and relatively short - execution routines. The only things to care for are the scheduler period and the implicit Task priorities that this kind of scheduler defines, since it runs some Tasks before others.

For what concerns the scheduler period, the two mutually exclusive aspects to keep in mind are *deadlines* and *event loss*: the ideal period would be short enough to never lose events, and long enough to allow all Tasks to execute in its duration, so to avoid failing deadlines. The ideal choice for the scheduler length would be the Tasks periods least common divisor, which is in my case 100ms, long enough to allow for all tasks to execute and short enough to read all events, since the *Operator Input Task* has a period of 100ms itself.

For what concerns implicit Task priorities, it's not really an issue in this subsystem. This because data is always valid, since the only valid source of data is considered to be the *Control Unit* (and therefore the reads performed by the *Communication Task* on the *Serial Line*), and because possible delays introduces by Task execution order are in the magnitude of tens of milliseconds, not relevant since the unit interacts with human operators and controls a servo motor, which has longer operating times.

Another aspect to deal with is data format to be sent on the *Serial Line*. Messages on the Serial Line are meant to be as light as possible, to lower as much as possible transmission times and parsing times in a resource-constrained environment such as the Arduino UNO Board.

Outgoing messages have a fixed length of 5 characters: the first represents with a letter (t or f) whether the operator has requested a mode switch pressing the tactile button, then a space and the percentage represented as a three digit number. For example, a message requesting a mode switch and inputting an 80% opening level would be

```
t 080
```

Incoming messages also have a fixed length of 14 characters, and are made of three parts. The overall structure is

```
Txx.xx Mx Oxxx
```

where the first part represents the temperature truncated to the second decimal digit, the second one represents whether the system is currently in manual mode, and the third one represents the last valid opening level as recorded on the system database.

3.2.1 Libraries and External Dependencies

For the scheduler timing purposes, I relied on this *Timer library* to avoid relying on system timer interrupts.
The LCD Screen, which works with the I^2C protocol, is controlled by the *LiquidCrystal I^2C library* to avoid having to handle all the low-level aspects of that communication protocol.
The servo motor is controlled via the *ServoTimer2 library*, which exploits one of the 8-bit system timers leaving the only 16-bit timer (Timer 1) free for other purposes.

3.3 Operator Dashboard

This sub-system is a web-based application that allows an operator to inspect the current state of the system, and to perform some simple actions on it. The core of this system is the user interface, realized with *HTML, CSS* and *JavaScript*.
In particular, *HTML* and *CSS* are used to structure the page, and are retrieved whenever a new dashboard window is open, and then *JavaScript* handles automatic refresh and user operations. Automatic refresh is needed to seamlessly retrieve - thanks to the *AJAX* paradigm - the data to be shown to the operator, while on the other hand user input is handled via *HTTP POST* requests to send data to the *Control Unit*.
Like other sub-systems, it considers only the system state sent by the *Control Unit* to be consistent, but it also keeps a cache of the history of the system in order to properly show the history graph to the user: it will be empty on startup, and it will be gradually drawn while refreshing the page increasingly adding the new data it receives.
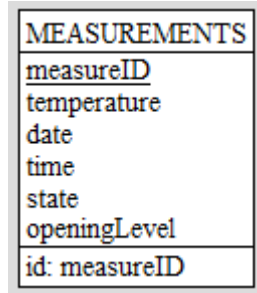To draw the measurements history, the *Chart.js library* was used.

3.4 Control Unit

The *Control Unit* is the core sub-system, that keeps track of the whole system's state in a consistent way. It's on its own divided in a few parts.

3.4.1 Database

The first component is a rather trivial, yet essential, database implemented in *mySQL*. It's made out of only one table, but it allows the system to save data persistently and to keep an history of the measurements even if the *Control Unit* stops its execution.

Figure 3.3: Scheme of the database table

Since the system is interfaced to a database hosted in *mySQL*, which by default runs in *autocommit mode*, all transactions are guaranteed to be atomic, as if they were included in a commit sequence (*as stated in this mySQL guide page*). Therefore, the *Control Unit* can delegate concurrency-related issues to the database Server.

3.4.2 Serial Agent

This component actively communicates with the *Window Controller*. It's implemented in Java, relying on the *JSSC library* for Serial Line Communication. It's responsible of gathering the in-place control unit input, communicating it to the *Central Controller* in order to reply effectively to the *Window Controller* to actuate the - eventually - mutated state of the system.

3.4.3 MQTT Agent

This component actively communicates with the *Temperature Monitor*. It's also implemented in Java, relying on the *VertX library* to implement an event-based asynchronous component. Its main activity is to wait for the *Temperature Monitor*'s messages, replying with the - eventually - mutated state of the system after communicating the measurements to the *Central Controller* and retrieving the new state of the system. For *MQTT* communication, I relied on the *HiveMQ Public Broker*. This was the final choice, after experimenting with a locally hosted broker and a tunneling application

to open the corresponding port. The latter, indeed, caused some issues in case of failed connection to the broker, not delivering the error response and therefore causing the *ESP32 board* to crash and reboot for a failed reset of the *Watchdog Timer* by the *Connection Monitoring Task*.

3.4.4   HTTP Server

This component acts as a very simple Server to answer the *HTTP* requests coming from the *Operator Dashboard*. It's also implemented in Java relying on the *VertX library*. Similarly to the *MQTT Agent*, it waits for the *Operator Dashboard*'s requests, retrieving then valid data from the database thanks to the *Central Controller*, and then communicating it as a response.

3.4.5   Central Controller

This component is the actual core of the whole system. It's responsible of all access to the database, and therefore must implement strong thread-safe procedures in order to ensure that data is handled properly. It's implemented in Java, relying on the *native JDBC API* to connect to the database and query it for data management.