Internship Report

Mobile Application for Monitoring Battery Charge at a Wind Turbine

Author:
Pablo Donato

Supervisors:
Andrew Donnellan
John Fox
Jean-Michel Bruel
Isabelle Clavel

April 11, 2016 — June 17, 2016

Institut Universitaire de Technologie de Blagnac — Toulouse II Département Informatique

1, place Georges Brassens - BP 73 -
31700 Blagnac Cedex

Institute of Technology Tallaght Dublin — Department of Electronic Engineering

Tallaght
Dublin 24
Ireland

Internship Report

Mobile Application for Monitoring Battery Charge at a Wind Turbine

Author:
Pablo Donato

Supervisors:
Andrew Donnellan
John Fox
Jean-Michel Bruel
Isabelle Clavel

April 11, 2016 — June 17, 2016

Institut Universitaire de Technologie de Blagnac — Toulouse II Département Informatique

1, place Georges Brassens - BP 73 -
31700 Blagnac Cedex

Institute of Technology Tallaght Dublin — Department of Electronic Engineering

Tallaght
Dublin 24
Ireland

Acknowledgement

First, I would like to thank Isabelle Clavel and Jean-Michel Bruel for giving me and Alexandre the opportunity to complete our internship in Tallaght, but also for their support and follow-up by email during these ten weeks. Special thanks to Ms. Clavel who came all the way to Dublin and attended our intermediate project presentation in May.

I would also like to thank our supervisors Andrew Donnellan and John Fox, who were always present to answer our questions and help us in our work, as well as for their guidance about life at the institute.

Of course I am grateful to my family and friends who kept supporting me even across the ocean, and to the new friends I made here who made a great experience out of this internship.

Finally, I wish to thank my teammate Alexandre Bontems for his great work and collaboration on this project.

Contents

Introduction

As part of my graduation from the Institut Universitaire de Technologie (IUT) Blagnac, I had to perform, along with my classmate Alexandre Bontems, a ten weeks internship in the Department of Electronic engineering of the Institute of Technology Tallaght (ITT), located in the city of Tallaght in the Dublin area. The goal of this internship was to apply the knowledge acquired during our studies at the IUT to a real research project with concrete applications.

Since the internship took place in a university, discovery of the world of work as well as integration into a company were not part of our objectives, unlike most of our french schoolmates; instead, the challenge was about work and integration inside an english-speaking, foreign academic environment. Another peculiarity is that both of us were assigned to the same project, therefore we had to share out the tasks among ourselves.

The general purpose of the project was to design a system that would allow researchers of the ITT to monitor remotely and simultaneously multiple inputs from various sensors placed in a wind turbine, which is located in front of the ITT's main building. Our work consisted in choosing the hardware and technologies most suited to the task, as well as setting up the network infrastructure and developing the software required to implement such a system.

In the first part of this report, I will present the context of the project, that is who it served, what was needed and why. I will next make an inventory and comparison of the available technologies, leading to the technological choices that were made and their justification. Third part will be dedicated to the design of the different parts of the system, with a special emphasis on the server side, which constituted most of my work. After comes the implementation part, also on the server side, and a fifth part about the future work that still needs to be done. I will finally talk about the results achieved in the whole project, on a professional, but also more personal level.

Please note that chapters 1 to 3 were written collaboratively by me and Alexandre, and therefore figure in both of our reports.

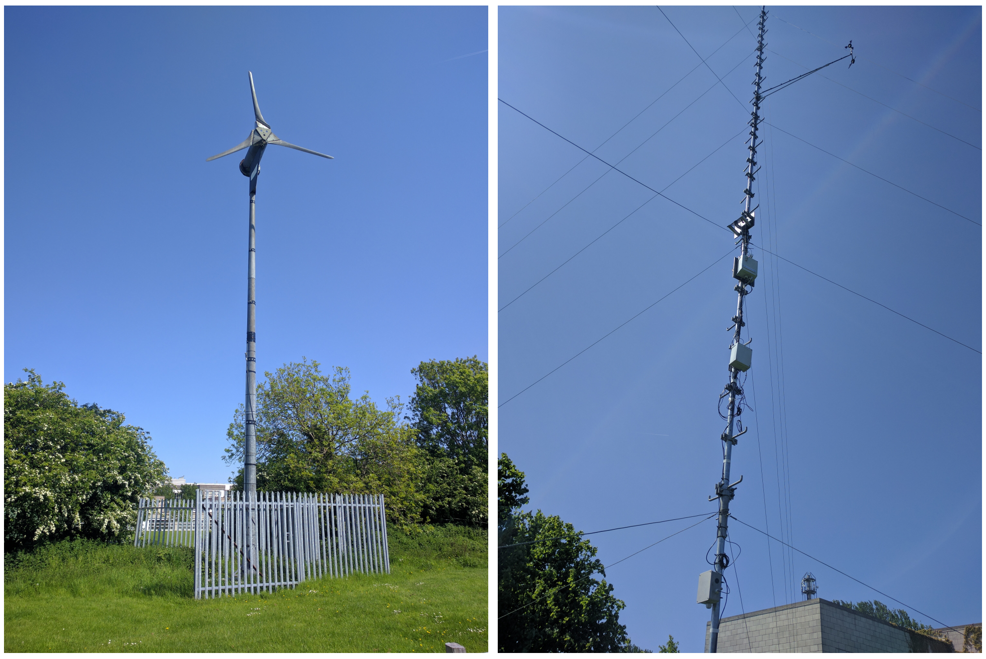
1 Context

1.1 The Wind Energy Technology Centre

The Wind Energy Technology Centre, involves a focus by ITT Dublin's School of Engineering, towards **research within the field of Wind Energy**. As a vast area of research, the School of Engineering current research has been attentive to Wind Energy at a Micro-generation level.

Continuing on from a Sustainable Energy Authority of Ireland (SEAI) initiative in 2011, surrounding wind data collection, (as applicable to the on-campus **6kW Wind Turbine** — see figure 1.1a). The Wind Energy Technology Centre is dedicated to continued research in this field while establishing links with industry and fellow institutes involved in this area. **The aim is to study the effects of an urban environment on the wind conditions** (speed, direction, etc) and consequently on the effectiveness of the turbine.

In conjunction with the 6kW Wind Turbine, a 15 meter **Meteorological Mast** — see figure 1.1b — resides in the vicinity of the Wind Turbine to allow comprehensive research to develop in this area. Weather stations as located on site within the campus, allow comparative data analysis. While two grid tied parallel inverters are in use to feed power back to the grid. The use of industry standard power analysing equipment as available within the School of Engineering gives the advantage of specific power analysis.



(a) 6kW wind turbine (b) Meteorological mast

Figure 1.1: Wind turbine and meteorological mast

1.2 Problem definition

In order to precisely mesure the wind conditions near the turbine, two anemometers have been placed on the top of the meteorological mast: one is located at the height of the turbine's hub and one at the height of the bottom radius of the blades. The difference of speed between these two points is a valuable source of information which is recorded with a high resolution frequency.

A battery (located in one of the boxes of the met-mast — see figure 1.1b) is also hooked up on the met-mast and is charged by these two devices. For maintenance and troubleshoot purposes, **it is necessary to be able to monitor the battery level at its source and be able to tell if the battery is charging**. The only way for the researchers was to climb up the met-mast which is time consuming and according to a direct testimony from one of the researchers: "It's frankly a pain".

There was therefore a need for a system capable of monitoring the battery and accessible from the ground. Specific needs for the battery involved displaying real time data of the voltage output and displaying an history of values over the last minutes. The system also had to be transferable with other sensors in the future, like the two anemometers on the met-mast.

1.3 Solution

The solution devised was to build a **node monitoring system to facilitate maintenance and troubleshooting scenarios**. An embedded system board would take the battery data and store it locally. Then an application would display this data in real time (and over the last few minutes) on a client device.

This monitoring system was to be point to point and enable quick observation of whether the battery was being charged and at what level the battery was being charged at. It was to be accessible **via a secured wireless medium that could be viewed on a tablet or phone device from ground level**.

Hence, the following general layout:

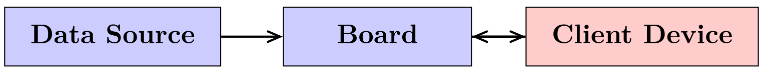


Figure 1.2: Global system layout

When the solution was implemented, the extensibility of the system was meant to be tested with the addition of pressure sensors to the board.

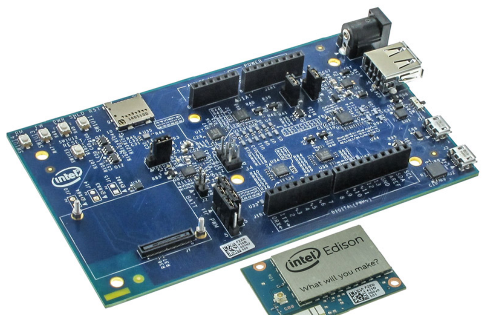
2 Background

One of the important goals of this project was the **choice of technologies for the different blocks of the system**. Specifically it involved picking an embedded board capable of sampling data from different sensors concurrently, choosing the most suited network topology and later on choosing appropriate pressure sensors. This part will detail which technologies were available and provide insight on the choices eventually made.

2.1 Embedded boards

In terms of embedded boards, two options were available: the **Raspberry Pi B+** and the **Intel® Edison**.

(a) Raspberry Pi model B+



(b) Intel® Edison Kit for Arduino

Figure 2.1: Embedded boards

It was decided to implement the system on the Raspberry Pi first and then test the implementation on the Intel® Edison to compare the results between the boards. A configuration guide was written for each board in this regard, and also as documentation in case the setup needed to be done again; those are available in the `board` git repository as *config/rpi.md* and *config/edison.md*.

2.1.1 Intel® Edison

The Intel® Edison is an Internet of Things (IoT) board whose design is oriented towards low power consumption. Out of the box, it contains an integrated Wi-Fi chipset and an Analog-to-Digital Converter (ADC), both needed for the implementation of this project. Consequently, the initial setup of these components is pretty easy as they are preconfigured in the Edison's operating system and libraries.

However the design of the board and the apparent inaccuracy of the ADC don't play in favor of the Edison.

While experimenting with the board, the values collected from the integrated ADC were off by several digits from what could be measured with a multimeter, despite the 12-bits resolution of the ADC. It turned out that the library used to collect data from the ADC, `mraa`, was configured on a 10-bits resolution by default. But even after changing this resolution in the program to the maximum of 12 bits, the values were still off up to 0.2V, **which was not an acceptable error margin**.

Also, despite the wireless setup being drastically simpler on the Edison (just a matter of pressing a button for 4 seconds), **the package manager, `opkg`, is very limited**, and all the libraries used had to be installed manually from source in order to dispose of the latest versions, while it took only a few commands on the Raspberry Pi.

2.1.2 Raspberry Pi B+

The Raspberry Pi Model B+ is an embedded board designed as a low-cost computing terminal. It features a relatively powerful processor and an **interesting hardware hackability for this project**. Its price and the large community behind the hardware — providing a wide array of resources and documentations — made the board a relevant choice for this project.

The model B+ does not, however, feature an integrated ADC nor an integrated Wi-Fi chipset. That means the implementation of the system required a more complex setup than the Edison: **installation of a Wi-Fi dongle and installation of an ADC**. The Wi-Fi dongle setup was especially complex, notably because of drivers specific to the chip that were not installed on the operating system, nor available in the package repositories.

Now both cards still need external storage space if values from the ADC are to be stored accross several years. On the Raspberry Pi, it has to be USB storage as the micro-SD card is already used by the system. Fortunately, this is very simple to achieve as the drive is instantly recognized, and just need to be mounted in the right place. USB storage is not supported by default by the Edison despite one port being available, so it would require a micro-SD card that may not have as much storage available.

MCP3008 ADC

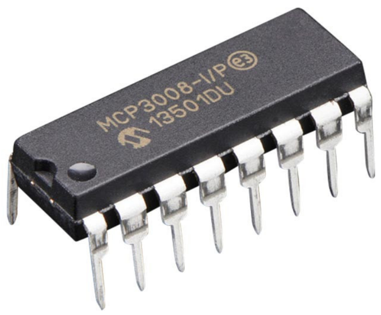


Figure 2.2: MCP3008

For this project, the `MCP3008` ADC was provided as it was necessary to convert values from analog sensors. It can take an analog input voltage and yield a digital value representative of the input.

This ADC requires a voltage supply between 2.7V and 5.5V: perfect for the Raspberry Pi which can provide either 3.3V or 5V.

Its **10-bits resolution** means it will provide a value between 0 and 1023 which can then be converted to a voltage value by comparing it to the **reference voltage** provided to the ADC. With the Raspberry Pi, a reference voltage of 3.3V can be provided so there is a step of 3.22mV between each value (3.3V/1023 = 3.22mV).

The `MCP3008` communicates through a Serial Peripheral Interface (SPI) bus which is relatively fast (compared to other bus types like I^{2}C) but requires a lot of wiring to the embedded board.

2.2 Network topologies

Another important choice was the network topology: what kind of network is the most suited to communicate with the client application?

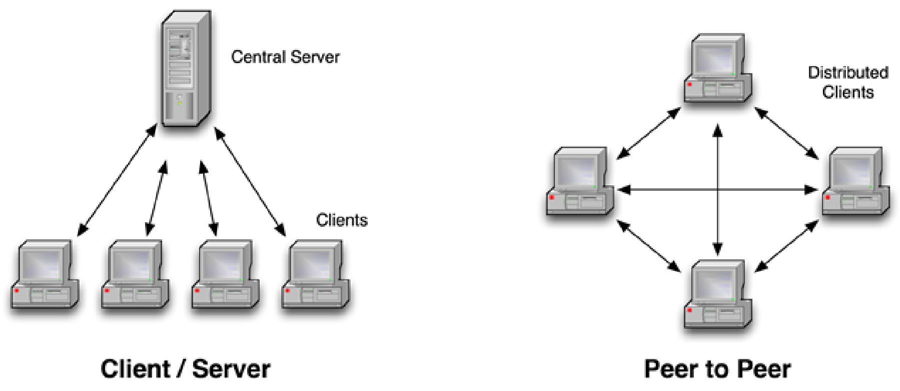


Figure 2.3: Network topologies

2.2.1 Peer-to-peer (ad hoc network)

The basic requirement was to be able to communicate with at least one client, which could be done with a basic ad hoc network. This is a decentralized network topology that would have greatly facilitated the setup on the server side. Unfortunately, this kind of network is very poorly supported by Android devices, so it could not be used for this project. Anyway, the client/server architecture of the system tends more toward the use of a centralized topology, as clients don't need to communicate between them.

2.2.2 Client-server (Wi-Fi access point)

Using a Wi-Fi access point is the standard way to setup a centralized wireless network. The access point acts as a router that will automatically assign Internet Protocol (IP) addresses to its clients: this allows to easily manage multiple clients requesting data to the server, as any Wi-Fi/Dynamic Host Configuration Protocol (DHCP) enabled device can easily join the network, including Android ones. Hence this is the solution that was chosen to implement the network.

2.3 Pressure sensors

The choice of pressure sensors was greatly influenced by the design of the application as they were chosen later during development. They needed to be easily connectable to the ADC and board used and other specifications were not important: the goal was to test the addition of sensors to the board. Therefore, the pressure sensor `NXP MPXHZ6400AC6T1` was chosen mostly for its appriopriate voltage supply requirements (between 4.64 and 5.36V which can be provided by both the Raspberry Pi and the Intel® Edison).

Figure 2.4: Pressure sensor NXP MPXHZ6400AC6T1

3 Design

3.1 Server side

3.1.1 Serving data on the network

The primary role of the board is to act as a server feeding realtime data from the battery and sensors in the wind turbine to remote client applications, for monitoring and maintenance purpose. Note the plural here for applications, implying that the server should be able to **handle multiple connections asynchronously**. Also, it was requested that the application give the current value of a given source, battery or sensor, but also an history of the latest minutes samples. Eventually other kinds of data would be requested for future needs, so it has been decided to establish a simple request protocol with predifined messages identifying each request.

The protocol is very simple: a request command is a two-characters string, where the first character should identify the source from which we want the data, and the second character its nature. The choice of character is only for semantic purpose, some special commands may not request data from a sensor; the only absolute requirement is the length of the string, so that it will simplify command parsing. Here are the requests defined and supported as of the date of this report:

- `` `bl` `` : current battery output voltage
- `` `bh` `` : battery output voltage history
- `` `sf` `` : status of the storage space on the board

As you can see, the `` `sf` `` request doesn't follow the semantic rule because of its special purpose ("sf" is for "Storage Full"), related to the collector discussed right below: at some point the board's storage space will be full, and the server must notify the monitor app to let the user know he has to free up space, either by going directly to the wind turbine change the external storage, or by downloading files remotely via Secure Shell (SSH) for example.

The way all those features should be implemented in their workflow is summarized in the workflow diagram shown in **figure 3.1**.

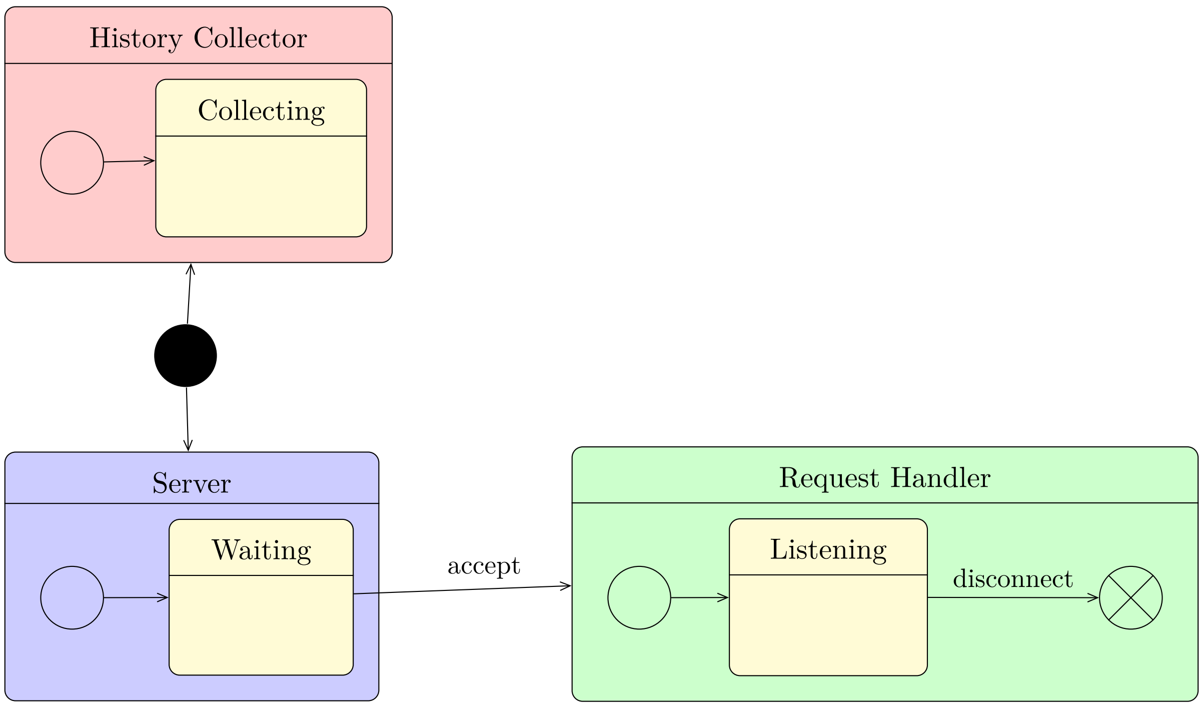


Figure 3.1: Server workflow diagram

We will have on one side processes updating independantly the history of the latest samples for each source, called **History Collectors**. On the other side will be the network part of the program, with the actual **Server** waiting for incoming connections, creating a **Request Handler** for each connection in a separate process that can be closed at any moment without affecting the rest of the program. There is no final state in the diagram, nor exit state for the History Collector and the Server, because they should ideally run indefinitely, unless someone wants explicitly to shutdown the server or the board, or if an unexpected error occurs.

3.1.2 Collecting data

The other role of the board is to log values from the battery and sensors on disk across several years. This will enable researchers to establish statistics in order to optimize some parameters, for example the placement of the wind turbine benefiting the best from the wind speed. This feature is independant from the realtime monitoring handled by the server, and should consequently be implemented as a separate program.

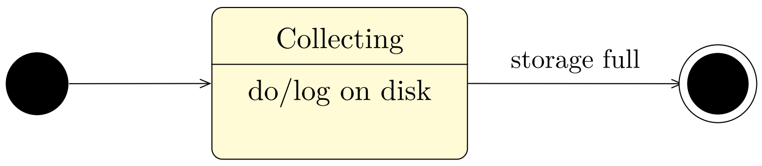


Figure 3.2: Collector workflow diagram

Such long periods of data collecting raise the problem of the storage space available, especially with the requested plain-text Comma Separated Values (CSV) file format, far more voluminous than an application-specific binary format. To overcome this problem, the solution chosen was to run on a regular basis a program that would check the amount of free space available, and stop the collector beyond a defined threshold, as shown in **figure 3.2**. The regularity at which the check is executed depends on a number of variables:

- The initial storage space available
- The average size of one sample log, itself depending on the nature of the sample as well as its formatting
- The number of sources collected
- The sampling rate for each source

The three latter variables determine the speed at which the amount of data will increase. The sampling rate should be 6 Hz as requested by researchers; we can consider 3 different sources: battery output voltage, wind speed and wind direction, though only the battery is supported as of the date of this report; and finally, every sample must be logged along with a timestamp formatted on 19 characters (`YYYY-MM-DD HH:MM:SS`), plus at most 6 significant digits for the value itself, which makes a total of 26 characters with the CSV comma. With these approximations, this gives us a speed of $6 \times 3 \times 26 = 468\,\mathrm{B\,s^{-1}}$. This amounts to $468 \times 3600 \times 24 = 40\,435\,200\,\mathrm{B} \approx 40\,\mathrm{MB}$ of data stored on a daily basis.

Considering a minimum of 4 GB of initial storage, this represents 1% of it, **giving enough margin to perform a daily check for a threshold >1% of free space**.

Finally, we should keep in mind that the server is also collecting data at a given sampling rate to keep its history of the latest samples, hence a common Application Programming Interface (API) should be implemented for both the server and the collector. This also implies that the two programs may want to read the ADC at the same time, which may lead to conflicts in transactions and therefore wrong values. Because all sources are read from the same ADC, this problem affects as well all the independant ”sub-collectors” for each source. It is thereby important to find a way to synchronize all the read procedures in the implementation.

3.1.3 Service scheduling

To collect and serve data on the network automatically just by booting the board, it is necessary to define a scheduling of all the services required, in order to satisfy possible dependencies between them. We must also consider possible events that may disrupt those services, and if not prevent them from happening, at least restart automatically said services so that the system keeps running without the need of manual intervention. This has been summarized in a workflow diagram shown in **figure 3.3**.

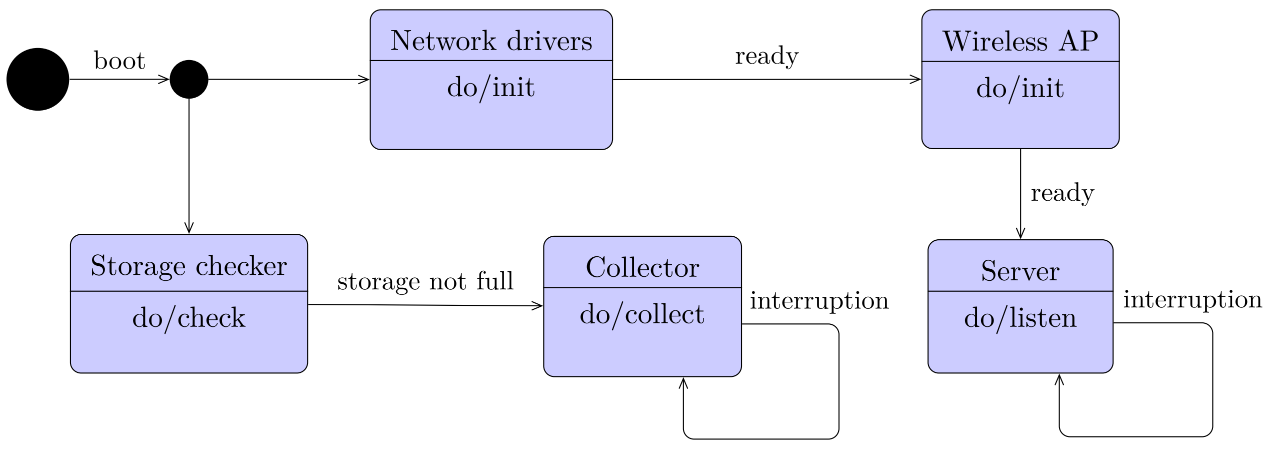


Figure 3.3: Service scheduling workflow diagram

1. First, the board is booted by being powered.
2. The **Collector** service can be started **only** if there is enough free storage space.
3. The **Wireless Access Point (AP)** will be started as soon as the system has finished preparing the **Network drivers**.
4. Finally, the **Server** can be started when the **Wireless AP** is ready.

The **Collector** and most of all the **Server** are the two services the most prone to unexpected events leading to interruptions, and therefore must be restarted automatically.

3.2 Client side

The main purpose of the client application is to **display real time data from the board's network**. Thus, the general workflow of the network actions in **figure 3.4** was planned early on. The application will try to connect to the board on startup until a connection is established. When connected, new data will be requested until the application loses focus or is closed, which means network actions will be performed in a loop while the application is running.

The application will be able to request different kinds of information concurrently and will support multiple sensors. Therefore, the application will feature different screens — one for each sensor — and every screen will be able to request data from the board and display it. To achieve this design, and because the future extensibility of the system is a requirement of this project, the client application will be **somewhat modular** so that the addition of new screens for additional sensors will be as easy as possible. This principle translates itself into the diagram of the **figure 3.5**.

This diagram states that the client application is to be composed of different views (screens) and one network manager shared accross all views. Each view will be able to access this network manager to request and retrieve data from a distant server, and update its content. The network manager will feature an API allowing the views to provide specific configurations like their necessary server commands (`bl, bh, ...`) and the intervals according to which new data

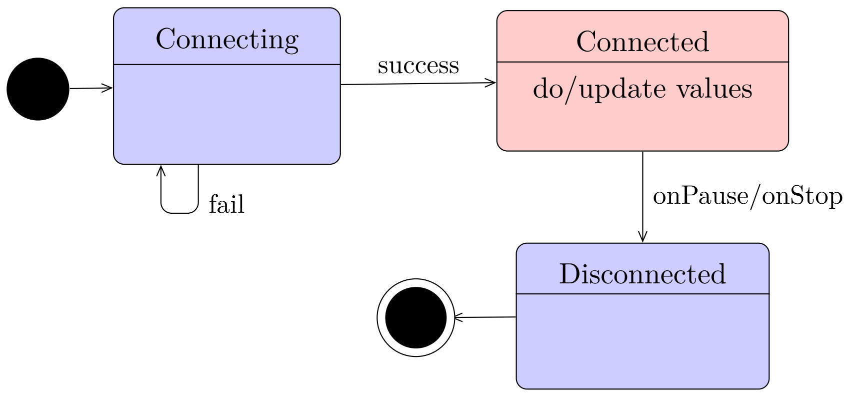


Figure 3.4: Android application general workflow

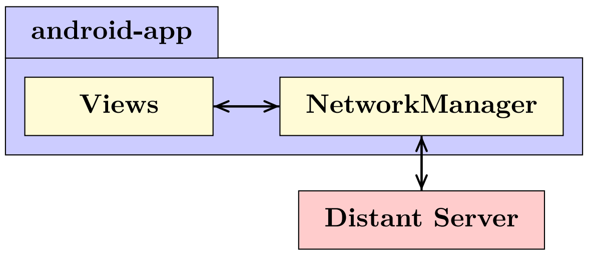


Figure 3.5: Android application general architecture

should be requested. The network manager, with said configurations, will be able to manage every network actions and thanks to the implementation of an **Observer Pattern**, provide the views with new data when a request is resolved.

4 Implementation

4.1 Board setup

The first step to implement the board monitoring system was to choose and setup all the software needed for each service/feature. We will only discuss in this section the technological choices that were done. Detailed installation instructions of the programs and libraries can be found in the configuration guides mentioned in **section 2.1**.

4.1.1 Operating system

The Operating System (OS) is the base on which will run all other softwares, so this is naturally the first piece of the system that was installed. Whatever board would be chosen for the final implementation, the constant was that it would be a GNU/Linux distribution, to maximize software compatibility between the boards (and because this is the industry standard on embedded boards). The Raspberry Pi features a wide range of available OS, and the winning choice was **Arch Linux ARM**; there are multiple reasons for this:

- It has a small size of 276MB, useful to save up space for the collector
- System configuration is simple and straight-forward
- It always features the latest software revisions thanks to its well-furnished package repositories as well as its rolling-release mechanism

As for the Intel® Edison, there are only a few alternative distributions because of the i586 architecture of the System on Chip (SoC), and none with the advantages of Arch Linux, so it was easier to keep the default **Yocto Linux**.

4.1.2 Programming language

It is next necessary to choose the programming language(s) used to implement the services themselves. There are many technologies available thanks to the boards running Linux, including C, Python and Javascript among the most popular.

As a low-level language with lots of libraries available, C would be a good choice if we needed true realtime data. However, the Linux kernel installed is not realtime, and even if it's possible to install one on the Raspberry Pi or to use the one installed on the second SoC of the Intel® Edison with the ViperOS GNU/Linux distribution, this would imply a lot of extra work for a feature that is not a requirement of the system: it is indeed not a problem if the values monitored are given with a few milliseconds, or even seconds delay. Moreover, the low-level nature of the C language is also its weakness, as the development takes a lot more time when high-level features like Object-Oriented Programming (OOP), multithreading and network communications are needed.

If responsiveness is not a problem, and high-level features are a requirement, then Python looks like the perfect fit for the task. Its clear and concise syntax, powerful object oriented capabilities, and very complete standard library, as well as its Test-Driven Development (TDD) model, make it suitable for the embedded system we want to implement. This is therefore the language that was used to program the two main services of the board, the collector and the server. More specifically, the latest version, Python 3.5, was to be used to benefit of the latest features of the language available. However, this version is not installed by default on the Edison, and consequently had to be compiled from source along with the corresponding versions of the external libraries used.

4.1.3 Service manager

To implement the service scheduling discussed in **subsection 3.1.3**, there is two standard ways on Linux:

- Using the legacy SysVInit to schedule programs at boot time, and `cron` to launch the storage checker (**subsection 3.1.2**) on a daily basis
- Using the new `systemd` machinery to implement both features in a unified way

`systemd` is the one installed by default on Arch Linux and Yocto Linux, as it is becoming the new standard in the Linux ecosystem. In addition, it allows a fine-grained management of service dependencies and conflicts that goes beyond a simple ordering of program startup at boot time. This is therefore the software chosen to implement service scheduling.

4.2 Sensor reading

One of the first work of the implementation process concerned the **collection of data**. The goal was to read the output voltage from an electric generator (mimicking the output of the met-mast's battery) and store it on the board.

4.2.1 MCP3008 ADC

To collect data from the generator and sensors, the **analog output** from those devices needs to be converted to **digital values**. Since the Raspberry Pi does not have an integrated ADC, an external one is used: **`the MCP3008`**. For more information on the wiring and communication protocol used with the MCP3008, please refer to **Alexandre Bontems**'s report.

4.2.2 Pressure sensors

Following the collection of battery data was the collection of pressure sensors data. However, because of a lack of time, the software for reading these sensors was not implemented. Informations on the wiring setup can be found in **Alexandre Bontems**'s report.

4.3 Data collection

As discussed in the design in **subsection 3.1.2**, it is required to define a common data collection API for both server and collector; once done, it is trivial to program the collector that uses this API to log samples on disk.

4.3.1 adc module

The fundamental operation when collecting data is reading them from the ADC. We just mentioned how to do this concretely with the MCP3008, which was probably the most challenging part, but what about the integrated ADC of the Edison? In order for the collector to support any possible ADC, it is necessary to abstract it through an API. That is what the `adc` Python module is for.

```python
# Adapt to current platform
try:
    import RPi.GPIO as GPIO
    GPIO.setwarnings(False)
    platform = "rpi"
except ImportError:
    try:
        import mraa
        platform = "edison"
    except ImportError:
        print("Error: the server must run on a Raspberry Pi or Intel Edison board.")
        sys.exit(2)
```

Figure 4.1: Code for adaptive ADC library import

First, we want our code to adapt to the current platform, Raspberry Pi or Intel® Edison, to avoid having to manage two separate versions of the module. For this, we do a so-called ”adaptive import”, as can be seen in **Figure 4.1**: we try to import each possibly needed library for each platform in a `try:` block, so that if the library cannot be found, the program infers that he is not running on the corresponding platform, and tries another in the `except:` block. The program will immediately stop if it cannot find any platform.

We next define the interface containing all the attributes and methods an ADC should possess with the abstract class `ADC`. As can be seen in the package `adc` of the class diagram in **Figure 2**, we use it to derive the two concrete ADC classes that will be used, the `MCP3008` and `ADS7951` classes. Here is the interesting point: we take advantage of the dynamic execution of Python code to define only the class that we will need by checking the current platform determined during the adaptive import, and then assigning the class object to a global variable `ADC_CLASS`, so that the collector will automatically read the ADC correctly by using this variable. The code sample in **Figure 4.2** illustrates this for the ADS7951 ADC of the Edison.

We also defined in the design part the necessity of synchronizing all the read calls from the ADC, to avoid conflicted values. Since this is directly related to concurrency, and more precisely in our implementation **multithreading** and **multiprocessing** as we will discuss later, we use the locking mechanism already implemented in the standard library to that effect. We use more specifically the `Lock` object of the `multiprocessing` Python module, as it allows synchronization between threads **and** processes at the same time. The entire module disposes of a unique instance of a `Lock` object named `readlock` to ensure that one read is made at a time from whichever script imports the module; then each ADC subclass must wrap the whole body of its `read` method in a `with readlock:` block as in **Figure 4.2**, taking advantage of the `with` statement with context managers to greatly simplify the management of the `Lock` object.

```python
elif platform == "edison":
    @ADC.init(10, 3.3, range(0, 5))
    class ADS7951(ADC):
        """
        The ADS7951 ADC integrated in the Intel Edison Arduino breakout board.
        """
        @classmethod
        def read(cls, channel):
            with readlock:
                cls.check_channel(channel)
                aio = mraa.Aio(channel)
                aio.setBit(cls.resolution)
                return aio.read() + aio.readFloat()

    ADC_CLASS = ADS7951
```

Figure 4.2: Code for dynamic definition of ADS7951 class

4.3.2 collector module

Now that the ADC can be read correctly on both the Raspberry Pi and the Intel® Edison, we can implement all the features related to data collection inside the `collector` module.

We basically want to do one thing: collect samples from an ADC, from a certain channel, and at a given sampling rate. And this is done for every input source we want to read, battery or any sensor. So we wrap all this into an abstract class `Collector` that will define 3 attributes and 2 methods as can be seen in **Figure 2**:

- `adc`, `channel` and `sampling_rate`, pretty self-explanatory. Note the `adc` attribute: we assign a specific `ADC` class object to each instance of `Collector`, in case a lot of channels are to be read in the future, and one ADC won't suffice anymore.
- `sample` and `collect`: the first method will return a tuple containing a value read from the collector's ADC channel, along with its timestamp; it can be called outside the class to manually read a value from the ADC, and is also called regularly inside the class at the ADC's sampling rate, with its return value passed to the `collect` method. The latter is what makes the class abstract, as it is to be implemented in each subclass to define what is done with the samples, i.e. how they are collected.

```python
def loop(self):
    self.collect(self.sample())
    sleep(1 / self.sampling_rate)
```

Figure 4.3: Code for Collector.loop(self) method

The `Collector` class extends the `StoppableThread` class of the `stoppable` module, in order to have each collector run independantly from the other, and be stoppable at any time. We only need to implement the `loop` method of the `Stoppable` interface (**Figure 4.3**), and the code will be repeated automatically, which corresponds to the behavior we want to collect data regularly. Then to handle the sampling rate, we just do a blocking wait of $1/sampling\_rate$ seconds at each loop, which also allows us to save a lot of Central Processing Unit (CPU) load. In fact, a first implementation with a non-blocking wait was first done in order to have multiple channels read in the same collector, but in addition to preventing the possibility of different sampling rates, the program was consuming almost 100% of the CPU load, while it takes less than 1% with a simple `sleep`.

```python
class HistoryCollector(Collector):
    """
    Collector handling a limited size history of the latest samples.
    """
    def __init__(self, adc, channel, sampling_rate=1/6, histsize=100):
        super().__init__(adc, channel, sampling_rate)
        self.history = deque(maxlen=histsize)
        self.lock = Lock() # Only one thread can use the history at a time

    def collect(self, sample):
        with self.lock:
            self.history.append(sample)
```

Figure 4.4: Code for HistoryCollector class

It is now easy to create any kind of collector with its own attributes, and a specific `collect` method, by deriving the `Collector` class. The `HistoryCollector` is used by the server to maintain its history of the latest minutes samples. It just adds a deque to its attributes with a given maximum size (defaults to 100). The principle of this data structure provided in the `collections` Python standard module is that when it reaches its maximum size, it will override the values first added in a First In First Out (FIFO) fashion, which corresponds exactly to the way a history works. We therefore just have to add the samples to the deque inside the `collect` method, and everything will be handled by the deque itself. Still, despite the deque data structure being well optimized for modification with atomic operations like the `append` method, it cannot be modified when accessed concurrently through an iterator, which is typically done when a remote client requests the history. This is why each instance of `HistoryCollector` manages a `threading.Lock` object to synchronize history access from different threads.

```python
class CSVCollector(Collector):
    """
    Collector logging samples on disk using the CSVDateLogger class.
    """
    def __init__(self, adc, channel, sampling_rate=6):
        super().__init__(adc, channel, sampling_rate)
        self.csv = CSVDateLogger(LOGS_DIR)

    def collect(self, sample):
        self.csv.log(sample)
```

Figure 4.5: Code for CSVCollector class

The other type of collector needed is the `CSVCollector`, that logs its samples in CSV files. It does so by using the `CSVDateLogger` class, that handles a file hierarchy with one directory per year, one directory per month, and one CSV file per day. In its `collect` method, the `CSVCollector` passes its samples to the `log` method of the `CSVDateLogger` that checks the current system date to determine the path of the file to write to.

Finally, to ease the management of a set of collectors sharing the same class and ADC, a `CollectorManager` class extending the built-in dictionary type was created. Each collector is identified by a string, generally of one character (e.g. 'b' for the battery), and can therefore be accessed with the syntax `collector_manager[<string_ID>]`. Actually, the constructor takes a dictionary associating each ADC channel number to its ID to generate all the collector instances: the main reason behind this is to allow the declaration of the collectors to be done once — which essentially comes down to defining which channel corresponds to which source — so that the server and collector programs share the same set of sources. Collectors can still be added with the `add` method, and removed via the standard Python deletion operator `del`. The `start` and `stop` methods are useful to manage the lifecycle of all the collectors in a unified way. Note that the `stop` method will wait for all the collectors to stop by calling `join()` on all of them.

4.3.3 storage-full script

To implement the storage checker discussed in **subsection 3.1.2**, the solution chosen was to create a small shell script (**Figure 4.6**).

```bash
#!/bin/bash

used_percentage=$(df / | tail -1 | tr -s " " | cut -d " " -f 5 | tr -d "%")
if [ $used_percentage -ge 95 ]; then
    systemctl stop collector &
    exit 1
fi
exit 0
```

Figure 4.6: Code for storage-full.sh

We first get the percentage of total storage space used by parsing the output of the `df` command. Then we compare this with the predefined threshold of 95%, and stop the collector program through `systemd` if storage is almost full; the script exits with 1 in this case, 0 otherwise. The exit code will be used by the server to inform the Android app of the current status of the storage. This is why the command stopping the collector on line 5 is launched in the background with a '`&`' at the end of the line, so that the script returns immediately and the server doesn't have to wait for the collector to stop to send its response to the client request.

4.4 Network and server

The main purpose of the embedded board is still to serve monitoring data on a dedicated network. It is therefore necessary to first setup the said network, as a wireless access point as discussed in **section 2.2**, and then implement the actual server program.

4.4.1 Software access point

Usually, a wireless access point is setup using a dedicated hardware of the same name. However, this is possible to implement all the features of a wireless AP using software. All the programs needed are available on Linux, and this is why the first approach was to configure each of those programs invidually, and then put them all together through scheduling and dependency management with `systemd`. For example, the automatic assignment of IP addresses to the clients would be handled with the standard DHCP server `dhcpd`, that needs to be run before the `hostapd` program that creates the Wi-Fi network and therefore needs the IP address of the board itself to be set. However, the overall setup was getting more and more complex and quite unstable, with the AP not always starting correctly at boot.

```
option domain-name-servers 8.8.8.8, 8.8.4.4;
option subnet-mask 255.255.255.0;
option routers 192.168.42.1;
subnet 192.168.42.0 netmask 255.255.255.0 {
    range 192.168.42.1 192.168.42.255;

    host WindTurbine {
        hardware ethernet b8:27:eb:28:c0:7e;
        fixed-address 192.168.42.1;
    }
}
```

Figure 4.7: Configuration file for dhcpd

Hopefully, a Bash script named `create_ap` had already been implemented by the community to ease the setup of a software AP on Linux. We therefore used this script, simplifying greatly the configuration process with a single `systemd` service file as can be seen in **Figure 4.8**.

```
1  [Unit]
2  Description=Wi-Fi software access point
3  After=network.target
4  Conflicts=netctl@eduroam.service
5  Before=netctl@eduroam.service
6
7  [Service]
8  Type=simple
9  ExecStart=/usr/bin/create_ap -n -g 192.168.42.1 wlan0 WindTurbine monitoring
10
11 [Install]
12 WantedBy=multi-user.target
```

Figure 4.8: create_ap service file

To start the access point, we just launch the `create_ap` command with a few options to configure the network parameters (line 9 in **Figure 4.8**). A short list explaining the role of each argument:

- `-n`: indicates that we don't share internet connection
- `-g 192.168.42.1`: IP address of the board on the network
- `WindTurbine`: SSID of the network
- `monitoring`: password used to secure the network

Note the dependency and conflict management done in the `[Unit]` section of the file: the `After=network.target` line allows us to wait for the network facilities of the system to be ready before starting the AP. As for the next two lines, they indicate that the AP cannot run concurrently with the connection to the `eduroam` network of the ITT providing internet access, because the Wi-Fi dongle supports only one mode at a time, `monitoring` (connection to another network) or `AP`; consequently, `systemd` will take care of automatically stopping the conflicting service when the other is started, would it be automatically at boot or with the `systemctl` command.

4.4.2 TCP server

The server program is scattered in two Python modules: the Transmission Control Protocol (TCP) request handler class, soberly named TCPHandler, located in the `server` module, and the server program strictly speaking implemented as a `StoppableProcess` in the `monitor` module discussed in **subsection 4.5.1**.

```
1  def handle(self):
2      print("[" + self.client_address[0] + "]", "Connected")
3      while True:
4          try:
5              cmd = self.request.recv(2).decode("utf8")
6              # Empty byte means the client closed its socket
7              if len(cmd) == 0:
8                  print("[" + self.client_address[0] + "]", "Disconnected")
9                  break
10             # Log command
11             print("[" + self.client_address[0] + "]", "Received:", cmd)
12             # Build response
13             data = getattr(self, cmd)()
14             # Send data
15             self.request.sendall(data + b"\0\n")
16         except:
17             print(traceback.format_exc())
18             break
```

Figure 4.9: Code for TCPHandler.handle(self) method

The `TCPHandler` consists of two parts: the `handle` method that is called once each time a new client connection is made, and a set of methods to handle each client request. The `handle` method is detailed in **Figure 4.9**, and is basically an infinite loop that will wait for requests from the client associated with the `TCPHandler` instance, execute the corresponding method, and send back the response. To find which method to execute from the command name sent by the client, we simply lookup in the namespace of the class with the standard `getattr` method (line 13). Therefore, methods names must be exactly the request commands defined in **subsection 3.1.1**, and must share the same signature, that is no parameters and the bytes to be sent in the response for the return value, as described in **Figure 2**.

The server is really launched in the `Server` process defined inside the `monitor` module (**Figure 4**). More precisely, this is done in a separate thread executing the `start_server` method, using the `socketserver.ThreadingTCPServer` class to handle each client connection in a separate thread, instanciating a `TCPHandler` for each. We also start (and eventually stop) all the history collectors declared in the `server` module using the `CollectorManager` class (cf end of **subsection 4.3.2**).

4.5 Monitor multiprocessing model

Now that we have all the features we need for the collector and server implemented into multiple modules and classes, we need a main program to be launched by `systemd`. The first approach was to have a script for the collector and another for the server; however this would have implied that they run in two independant processes, cancelling the read synchronization implemented in the `adc` module.

4.5.1 Monitor program

The solution chosen was to create a monitor module/program that would be the main process managed directly by the operating system; the monitor manages each program accessing the ADC as a child process, spawned using the `multiprocessing` Python library, the same library used to lock `read` calls in the `adc` module. We can see in the class diagram in **Figure 2** that each child process — namely the Server and Collector classes — extends the `StoppableProcess` class implementing the `Stoppable` interface, in order to be started and stopped on-demand, and properly thanks to the respective `setup` and `finish` methods.

We next define a `Monitor` class (**Figure 3**) that will manage a dictionary of all the child processes, and defines `start` and `stop` methods to respectively spawn a new child process only if it is not already running, and stop it then wait for its termination. This class also handles a dictionary of `ProcessRestarter` objects for each process.

The `ProcessRestarter` class is used to emulate the `Restart=on-failure` mechanism of `systemd`. Because we cannot manage directly the child processes via `systemd`, and we still want to implement the auto-restart feature described at the end of **subsection 3.1.3**, we have to create another child process for each child process of the monitor, watching it and restarting it whenever it exits with a failure status code.

4.5.2 systemd services

There is a total of 3 `systemd` services with this monitor architecture:

- The monitor program itself with no dependency; should obviously be the first one to be started
- The collector program, that requires only the monitor to be already running
- The server program, that requires the wireless AP to be active in addition to the monitor

The child processes of the monitor, that is the collector and server, are started and stopped through realtime signals sent by `systemd` with the `pkill` command, and handled in the monitor program with the `signal` Python module.

4.6 Android application

More information can be found in the report of **Alexandre Bontems** who was in charge of this part of the implementation.

5 Future Work

5.1 Embedded board

5.1.1 Accurate timestamps

One big problem with the current implementation is that the operating system time of the board is completely off, because it doesn't have an integrated hardware clock to keep track of the time when the system is down.

One solution is to synchronize with online servers using the Network Time Protocol (NTP). This is automatically done on the Raspberry Pi when connecting to the internet through the `eduroam` network. However, the board may not be able to connect to `eduroam` from the wind turbine, and this could hardly be done automatically at boot time anyway. This can be a last resort solution if `eduroam` is accessible from the wind turbine, and with the assumption that the system should never shutdown — in this case the board can't be rebooted to restart the collector after changing the external storage, this must be done manually through an SSH connection by executing the command `systemctl restart collector`.

A more viable and durable solution is to install an external hardware clock to have the correct time without the need for an internet connection. There is a very good tutorial available on the Adafruit website [6] covering the hardware and software parts of the setup, with notably a section about `systemd` (designed for Raspbian, but easily applicable to Arch Linux). The hardware used is the **DS1307 Real Time Clock breakout board kit**.

5.1.2 Supporting new sensors

Adding a new sensor to the monitoring system can be done in three steps.

First, it is necessary to correctly wire the sensor to the ADC, using the sensor documentation as well as the explanation of the ADC wiring provided in **section 4.2**.

Then, a collector can be added on the ADC channel used for the sensor by editing the `CHANNEL_IDS` dictionary variable of the `collector` module. This variable is the default one used by the `CollectorManager` class, documented in **subsection 4.3.2**.

Finally any required client request command concerning the data fed from this sensor can be added as a method of the `TCPHandler` class of the `server` module, as described in **subsection 4.4.2**.

5.2 Android application

Future work concerning the client application mostly involves adding new screens for new sensors. Specifically the screen for the pressure sensors was not implemented due to a lack of time and so a new fragment along with optional preferences must be implemented. For more information about how to do this, please refer to **Alexandre Bontems**'s report.

6 Results

6.1 Professional results

Given that not all objectives were defined from the beginning of the project, but were rather added gradually by our supervisors depending on our progress with Alexandre, we could not really plan in advance all the tasks we needed to achieve, with GANTT diagrams for instance. We therefore had to adopt an Agile development process, and our project was quite a success in that aspect, even if we did not formalize it through written practices like backlogs. Even so, we did share tasks among ourselves with Alexandre and communicated regularly, especially in the beginning during the research and design phases when we had to establish a working communication between the Android application and the server on the board.

In summary, we succeeded in implementing all the features that were asked with reasonable time available. The ones that still need to be done are all described in **chapter 5** and were asked at a very late stage of the project. Here is a short list of the main features with their estimated degree of completion:

Feature	**Degree of completion**
Current battery voltage monitoring	100%
Latest minutes battery voltage monitoring	100%
Concurrent data collection from multiple sources in CSV files	100%
Notification when board storage is full	100%
Accurate timestamps	40%
Support for pressure sensors	50%

Figure 6.1: Table of main features and their final degree of completion

Accurate timestamps are not yet implemented, but the solution has already been designed, which was accounted for an arbitrary 40% of the work. As for the support for pressure sensors, all the effort put into the advanced extensibity of the system should remove another arbitrary 50% of the workload.

In terms of lines of code produced, it would be unwise to judge on quantity, or rather the fewer the lines, the greater the quality of the code. The main focus was indeed to produce a base system architecture with a maximum of extensibility to implement quickly and easily support for new sources of analog data. In this regard, the project was a success: focusing on my part of the work specifically, the board monitoring system, there is only 536 lines of well documented Python code, and adding a new source to the collector can be done in as few as 6 characters; as for handling a new client request, it is just a matter of adding one method returning the required data.

The work accomplished so far, in addition to being usable by researchers to monitor the battery voltage directly from their smartphone, will greatly simplify future work concerning the addition of new sensors to the system. Therefore, this will save a lot of time to both researchers and future trainees, giving them the ability to focus on their core activities.

6.2 Personal results

On the technical level, this project has greatly improved my skills in embedded development, notably when it comes to choosing the technology that suits the best your needs. I was able to use a lot of advanced idioms of the Python language studied this year to optimize my code, combined with OOP concepts that I had to translate from Java to Python. Also, having a personal strong interest in Linux technologies, I was able to learn a new one that is prominent in system administration nowadays: `systemd`. I found somewhat innovative the connection of all those technologies together, communicating with the Android application developed by Alexandre. This brings me to collaborative development in a team, and even though we were only two on this project, this allowed me to better master the Git version control system.

On the contrary, the internship wasn't as fulfilling as I expected in terms of soft skills. Of course, as stated in the introduction, the fact that it happened within a university suppressed the challenges related to integration into a company. I expected however that I would be able to work with other exchange students and improve my communicational and linguistic skills in that aspect; but because we were only french trainees from the IUT in our office, the only opportunities I had to speak in English were the intermediate reviews and presentations with our supervisors, who were of course very educative, but too rare to really immerse myself in the language and significantly improve my level.

In the end, this project has confirmed my taste for embedded computing and all the possibilities it opens in many fields of the IoT, ranging from sustainable energies in this project to live arts, the latter being one of my main interests for my future career. It also raised in me a certain interest in Unix system administration, although I may not turn it into my very specialty.

Conclusion

In the end, this ten weeks internship was a very fulfilling experience on a technical, and to a lesser extent human level.

We succeeded in implementing almost all the features requested, with a solid code base that should be easily extensible for future developments: these include installing a hardware clock on the board to have accurate timestamps for proper monitoring, as well as support for new sensors on the server and client side, notably pressure sensors. Researchers can already use our system to monitor the battery output voltage of the wind turbine in a very practical way from their Android smartphone or tablet, but also establish statistics across several years on the data collected to improve the efficiency of the wind turbine.

This is quite rewarding to be able to participate in a research project on such a promising field as sustainable energies. And even if the linguistic immersion could have been better, I was still able to greatly improve my english in a way that would not have been possible if I had completed my internship in France.

Concerning my career plan, it has just confirmed some of the technologies I like and want to work with in the future, especially with the innovative possibilities of embedded computing. Still, my fields of interest stay unchanged: I strive to work in applied research at the crossroad of computer science, live arts, and cognitive science.

Glossary

Analog-to-Digital Converter Device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude. 9, 33

Application Programming Interface Set of routine definitions, protocols, and tools for building software and applications. 15, 33

Central Processing Unit Electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output operations specified by the instructions. 22, 33

Comma Separated Values File format for storing tabular data (numbers and text) in plain text. 14, 33

Dynamic Host Configuration Protocol Standardized network protocol used on IP networks for dynamically distributing network configuration parameters, such as IP addresses. 11, 33

First In First Out Method for organizing and manipulating a data buffer, where the oldest entry is processed first. 22, 33

Internet of Things Network of embedded devices with electronics, software, sensors, and network connectivity that enables these objects to collect and exchange data. 9, 33

Network Time Protocol Networking protocol for clock synchronization between computer systems. 28, 33

Object-Oriented Programming Programming paradigm based on the concept of ”objects”, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. 18, 33

Secure Shell Cryptographic network protocol for operating network services securely over an unsecured network; the best known example application is for remote login to computer systems by users. 13, 33

Serial Peripheral Interface Synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. 11, 33

System on Chip Integrated circuit that integrates all components of a computer or other electronic system into a single chip. 18, 33

Test-Driven Development Software development process that relies on the repetition of a very short development cycle. 19, 33

Transmission Control Protocol Core protocol of the IP suite providing reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating over an IP network. 25, 33

Acronyms

ADC Analog-to-Digital Converter. 1, 9–12, 15, 19–21, 23, 27, 28, 33, 39, *Glossary:* Analog-to-Digital Converter

AP Access Point. 1, 16, 24, 25, 27

API Application Programming Interface. 15, 16, 19, 20, 33, *Glossary:* Application Programming Interface

CPU Central Processing Unit. 22, 33, *Glossary:* Central Processing Unit

CSV Comma Separated Values. 14, 15, 23, 29, 33, *Glossary:* Comma Separated Values

DHCP Dynamic Host Configuration Protocol. 11, 24, 33, *Glossary:* Dynamic Host Configuration Protocol

FIFO First In First Out. 22, 33, *Glossary:* First In First Out

IoT Internet of Things. 9, 30, 33, *Glossary:* Internet of Things

IP Internet Protocol. 11, 24, 25, 32

ITT Institute of Technology Tallaght. 6, 7, 25, 34

IUT Institut Universitaire de Technologie. 1, 6, 30

I^{2}C Inter-Integrated Circuit. 11

NTP Network Time Protocol. 28, 33, *Glossary:* Network Time Protocol

OOP Object-Oriented Programming. 18, 30, 33, *Glossary:* Object-Oriented Programming

OS Operating System. 18

SEAI Sustainable Energy Authority of Ireland. 7

SoC System on Chip. 18, 33, *Glossary:* System on Chip

SPI Serial Peripheral Interface. 11, 33, *Glossary:* Serial Peripheral Interface

SSH Secure Shell. 13, 28, 33, *Glossary:* Secure Shell

TCP Transmission Control Protocol. 1, 25, 33, *Glossary:* Transmission Control Protocol

TDD Test-Driven Development. 19, 33, *Glossary:* Test-Driven Development

Bibliography

[1] Wikipedia, `http://wikipedia.org`

[2] stackoverflow, `http://stackoverflow.com`

[3] Python 3 documentation, `https://docs.python.org/3`

[4] systemd manpages, `https://www.freedesktop.org/software/systemd/man/index.html`

[5] ITT Dublin, Wind Energy Technology Centre, `http://eng.ittdublin.ie/Wind-Energy-Technology-Centre`

[6] Limor ”Ladyada” Fried, Adding a Real Time Clock to Raspberry Pi, Adafruit, 2016-04-29 11:45:10 PM EDT, `https://cdn-learn.adafruit.com/downloads/pdf/adding-a-real-time-clock-to-raspberry-pi.pdf`

Appendices

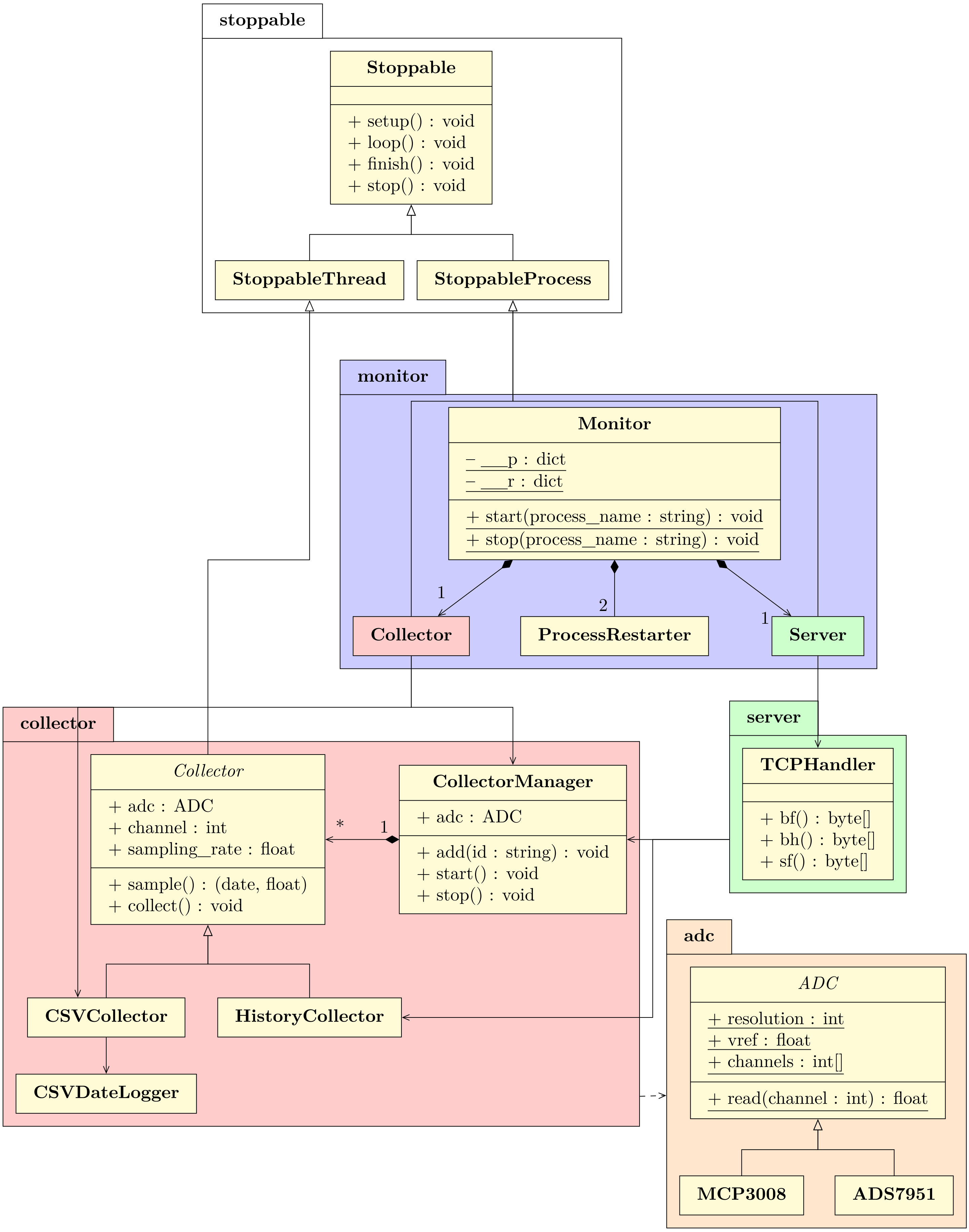


Figure 2: Board monitoring system class diagram

```python
class Monitor:
    """
    Class managing the collector and server processes.

    It emulates systemd Restart=on-failure mechanism
    with the ProcessRestarter class.
    """
    # Processes
    __p = {}
    __p["collector"] = Collector()
    __p["server"] = Server()

    # Process restarters
    __r = {name: ProcessRestarter(name) for name in __p.keys()}

    @classmethod
    def start(cls, name):
        """
        Starts a new instance of the process with
        the given name if it is not already started,
        as well as its corresponding ProcessRestarter.
        """
        if not cls.__p[name].is_alive():
            cls.__p[name] = cls.__p[name].__class__()
            cls.__r[name] = ProcessRestarter(name)
            cls.__p[name].start()
            cls.__r[name].start()

    @classmethod
    def stop(cls, name):
        """
        Stops the process with the given name if it is already started,
        as well as its corresponding ProcessRestarter.
        """
        if cls.__p[name].is_alive():
            cls.__p[name].stop()
            cls.__p[name].join()
            cls.__r[name].stop()
            cls.__r[name].join()

    @classmethod
    def exitcode(cls, name):
        """
        Returns the exitcode of the process with the given name.
        """
        return cls.__p[name].exitcode
```

Figure 3: Code for Monitor class

```python
class Server(StoppableProcess):
    """
    Server feeding various ADC inputs to remote clients.
    """
    def start_server(self):
        """
        Thread target to launch the server without being
        blocked by the serve_forever() call.
        """
        addr = ("192.168.42.1", 8888)
        try:
            self.__server = ThreadingTCPServer(addr, srv.TCPHandler)
            self.__server.daemon_threads = True
            self.__server.serve_forever()
        except OSError:
            srv.collectors.stop()
            print(traceback.format_exc())
            sys.exit(1)

    def setup(self):
        srv.collectors.start()
        Thread(target=self.start_server).start()

    def finish(self):
        srv.collectors.stop()
        self.__server.shutdown()
```

Figure 4: Code for monitor.Server class

List of Figures

Abstract

The aim of this project was to provide an easy way to monitor remotely different sensor values coming from a wind turbine located in the Institute of Technology Tallaght. The request was made by the university to two french students from the IUT Blagnac, as a research project for their internship. Still, there was a real need behind the project, as the resulting application was to be used internally by a group of researchers on a regular basis for their work. Thereafter are presented the tasks performed by the author of this document.

The first step consisted in researching the most suited embedded development board between the Intel® Edison and the Raspberry Pi Model B+, which would be used to gather data from the sensors and provide them to the clients. It came out that both had their load of pros and cons, and it was decided to start with the Raspberry Pi, then try out the Edison, and finally compare the implementation process of the two boards to assess the best choice in the final setup.

Next was the design phase of the server side, which consisted mainly in system-wide and application-focused workflow conception. This basically amounts to specify the future implementation of the server to be completely autonomous and exception-safe to minimize manual maintenance, especially thanks to a self-restart mechanism handling unexpected, fatal exceptions.

Last but not least was the server implementation. The first step was to install the lightweight operating system Arch Linux on the Raspberry Pi, in order to use the external Wi-Fi dongle and run other programs in a scheduled way, while minimizing power consumption. A wireless, software AP was then set up to establish a network with the client devices. The actual server program was implemented in Python, collecting data from the sensors through an ADC connected to the board, and feeding them to the clients through a TCP connection. Finally, scheduling was done with the service manager `systemd`.

TRAITEMENT DOCUMENTAIRE

NOM ETUDIANT : Pablo Donato

Date de Naissance : 29/04/1996

NOM TUTEUR IUT : Jean-Michel Bruel

DEPT. – ANNEE – PROMO : Informatique - 2015/2016

Signature [illegible]

NOM ENTREPRISE : Institute of Technology Tallaght

Ville- Pays : Tallaght - Ireland

NOM TUTEUR ENTREPRISE : Andrew Donnellan

Signature Andrew Donnellan

☐ Confidentiel *

SUJET DE STAGE : Mobile application for monitoring battery charge at a wind turbine

MOTS CLES (5) : embedded systems - wind turbine - android network - raspberry pi

RESUME :

Development of a mobile application to monitor battery charge at a wind turbine