Control software for robotic observatories

Francesc Domene

Resum– L'Institut d'Estudis Espacials de Catalunya (IEEC), i l'Institut de Ciències del Espai (ICE-CSIC) estan especialitzats entre altres coses, en la robotització i automatització de telescopis. L'edifici de l'ICE al campus de la UAB, disposa d'un observatori com a laboratori de millores d'aquests sistemes. Actualment el seu control és parcial i manual, tot i que existeix una base en ROS v1.0, un framework de robòtica open source. L'abast del projecte consistia a adaptar i actualitzar el programari actual de ROS 1.0 a ROS 2.0 i desenvolupar una interfície de control que permeti el control remot de l'observatori. En el projecte es realitzen totes les etapes pròpies de l'enginyeria de software (anàlisis, disseny, desenvolupament i test) seguint una filosofia àgil basada en Kaban i iteracions. Com a resultat s'ha aconseguit un producte mínim que permet la continuació del desenvolupament i demostra el funcionament de tota l'arquitectura.

Paraules clau– Astronomia, Observatori, Desenvolupament de software, ROS, Pylons, Pyramid, Polymer, SQL, Python, Docker, Linux, Raspberry Pi.

Abstract– The Institut d'Estudis Espacials de Catalunya (IEEC) and the Institut de Ciències del Espai (ICE-CSIC) specialize, among other things, in the robotization and automation of telescopes. The ICE building on the UAB campus has an observatory as a laboratory for the improvement of these systems. Currently, its control is partial and manual, although there is a basis in ROS v1.0, a robotic open source framework. The scope of the project was to adapt and upgrade the current software from ROS 1.0 to ROS 2.0 and develop a control interface that allows the remote control of the observatory. In the project, all the stages of software engineering (analysis, design, development and testing) are carried out following an agile philosophy based on Kaban and iterations. As a result, a minimal product has been achieved that allows the development to continue and demonstrates the operation of the entire architecture.

Keywords– Astronomy, Observatory, Software Development, ROS, Pylons, Pyramid, Polymer, SQL, Python, Docker, Linux, Raspberry Pi.

1 Introduction

In astronomy, specialized infrastructures are required for the investigation of the cosmos. These infrastructures can be more or less complex and consist of observatories and telescopes. There are a variety of observatories that could be classified in those that use radio telescopes designed to capture radio waves and those that use optical telescopes that capture images of the visible light spectrum. The use of these observatories goes from the use of a single optical telescope to the use of multiple radio telescopes that can act as one or independently.
The most recent examples are the use of the Event Horizon Telescope (Figure 1) to capture the shadow of a black hole or the SKA (Figure 2) telescope being built in South Africa and Australia.

Fig. 1: Pico Veleta observatory, member of the Event Horizon Telescope.

Fig. 2: Square Kilometer Array telescope artist's impression.

• Contact e-mail: francisco.domene@e-campus.uab.cat
• Mention: Software Engineering
• Academic advisors: Lluís Gesa Bote, Xavier Otazu Porter
• External advisor: Francesc Vilardell Sallés (IEEC)
• Course 2019/2020

In order to control observatories efficiently, it is necessary to develop robotic automation systems that provide remote and planned control of those observatories, which is why the astronomical community uses a variety of software for the control of these systems. Hence, they must specialize in order to have the support of those systems, which implies a high cost and the dispersed use of this software.

1.1 Document structure

The following section of the article describes a comparison of current control systems for observatories and a brief description of the observatory used as a case study with the proposed architecture. The next section describes the project objectives. The following chapter explains the methodology and planning, followed by the development cycle. Then a discussion of the results and to finalize, the conclusions and future lines of work of the project.

1.2 State of the art

Currently, in the astronomy field, there are more than 130 professional observatories around the world that can work autonomously [1] and have different control systems.
Gradually, the open-source philosophy is spreading, which has given more versatile and adaptable control systems involving INDI Library and RTS2. These frameworks provide control over different astronomical instruments, especially for telescopes. However, all these systems are not well known outside the astronomy field, which means having less support depending on which software is used due to the need for more specialized staff in order to maintain it.

An example of a control system is the OpenROCS (Robotic Observatory Control System) [2], an open-source software developed by the Institute for Space Studies of Catalonia (IEEC) [3] that controls, among others, the observatory and the telescope in (Figure 3).

Fig. 3: The Joan Oró Telescope (TJO) situated at the Montsec Astronomical Observatory (OAdM), Catalonia.

The use of these control systems is very dispersed and varied, so building new systems and making all the devices work takes a long time. For that purpose, the IEEC made an approach and developed a control system based on the ROS [4] standard.
ROS stands for Robotic Operating System [5]. ROS is a flexible framework for writing robot software that aims to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms. This framework has large flexibility and a large community.

The IEEC has a laboratory (The AstroEarth Laboratory) that is a facility devoted to the development and testing of new equipment, either hardware or software for ground-based telescopes. The laboratory consists of the following components:

- 3.5m Baader Planetarium AllSky dome
- Meade 10 "LX200GPS Schmidt-Cassegrain telescope
- SBIG ST-7 CCD camera
- Vaisala PTU200 term-hygrometer
- APC AP7920 Rack PDU
- CCTV camera to supervise the observatory operations.

Fig. 4: IEEC AstroEarth lab observatory.

The system developed in ROS 1 controls all the devices in the observatory.
This project proposes to upgrade the version of ROS 1 to the current and improved version ROS 2 [6] and the creation of a web portal in order to control the IEEC observatory devices trough the ROS system remotely. As well as the functionalities that a web portal includes, such as an authentication and authorization system to grant control over the devices and the information displayed.

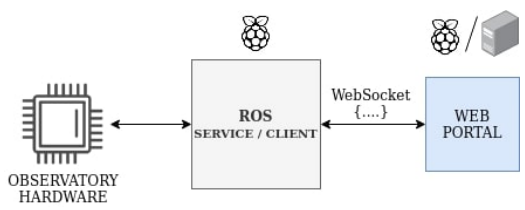

Fig. 5: High level diagram showing the global architecture communication.

1.3 Objectives

The objectives follow the pattern TFG-OBJ-X enumerating and describing its priority over the project.

- TFG-OBJ-1: Analyze the project's viability and plan its development trough an agile philosophy.
- TFG-OBJ-2: Investigate and study the current technology involved in the existing ROS 1.0 software. In order to understand from firsthand how ROS is designed, its applications and how it works. Then understand the current ROS 1.0 base.
  - TFG-OBJ-2.1: Investigate ROS 1.0 framework.
  - TFG-OBJ-2.2: Compare ROS 1.0 and ROS 2.0.
  - TFG-OBJ-2.3: Investigate methods for the current ROS usage and improvement to ROS 2.0.
- TFG-OBJ-3: Design and develop an open-source solution to control robotic observatories using ROS 2.0 once ROS and the previous base of ROS v1.0 are understood.
- TFG-OBJ-4: Design and develop a web portal in order to control the ROS software remotely and enable users to manage it. Once the ROS part is finished, the objective is to develop a web site that allows users to control the observatory remotely.
- TFG-OBJ-5: Design and develop the connection module that will enable the communication between the ROS module and the web portal.

2 METHODOLOGY

The project development followed an agile philosophy [7] basing the project planning on iterations.
Some methodologies, such as SCRUM, Lean, or Kanban [8] are the best known and used in the world of software development. All of them oriented to development teams, with their similarities and differences. That is why a single methodology cannot be appropriately chosen, especially when this project is developed individually.
Seeing that, it was decided to use a little of each to define a methodology that suits the project and the type of development that was carried out.

2.1 Description

A mix of Kanban and Scrum has been used in this project. It consists of the visual management section that Kanban provides, together with the Scrum iterations and the continuous workflow of Kanban, something like Scrumban. Therefore, tasks are going to be planned for an actual date. Still, at the same time, it remains flexible when adding, prioritizing, or modifying tasks during the development. The latter refers to the fact that in Scrum, it is not possible to alter the work done in a sprint once planned, however with Kanban, it is. For example, having to prioritize a new requirement in the middle of an iteration, unexpected scenarios or change the planning given a poorly organized calendar or initially projected optimistically.

2.1.1 Workflow

The task flow goes from the Backlog, To Do, Doing, and from there to either Done, Review, or Block if a problem may occur. If the latter happens, the tasks should go back to the beginning and go to the process again.
A short description of these phases:

- Backlog: The backlog column is full of the tasks that need to be completed. Unlike SCRUM, this backlog is not sprint focused. Instead, it contains all the tasks to be done, and it can be modified at any time.
- To Do: As the name implies, the tasks that are planned to be done in the current iteration are going to be there.
- Doing: Tasks that are currently in development.
- Revision: If some tasks need additional testing or reviews (like the bachelor's thesis goals).
- Blocked: If some problem happened during the development or after and it is blocking the development somehow.
- Done: When the task is fully finished, it reaches the Done and last phase.

2.1.2 Iterations

Moreover, the methodology uses iterations or sprints. Each iteration consists of the four typical stages of software development; Analysis, Design, Develop, and Testing.

- Analysis (What we want): Define the requirements to be done at the beginning and check them every sprint, gather data from stakeholders and plan the development.
- Design (How to get what we want): Investigate and come up with solutions to the tasks planned.
- Develop (Create what we want): Developing the software following the previous steps.
- Testing (Did we get what we want?): Test what has been developed and see if it matches the requirements.

Fig. 6: Agile software development life cycle.

2.2 Tools

The Trello [9] online software was used to keep precise and visual control of the workflow, where both the stakeholders and the Bachelor's thesis tutor could view it at any time. Furthermore, the GitLab repository was used to upload and keep track of the work done along with the test environment.
An example of the project's Trello board can be found in appendix section A.1.

2.3 Planning

In preparation for the project development, the planning was done following a milestone deadline and a sprint definition.

Sprint	Date	Milestone
3	19/04/2020	ROS2 MVP
4	10/05/2020	ROS-Web Connection Module MVP
5	24/05/2020	ROS-Web Connection Module Completed
		WEB based architecture
6	14/06/2020	Deliver final product (ROS, ROS+WEB Connection and WEB MVP)

Fig. 7: Initial milestone table regarding project's module milestones.

Once the milestones were decided and approved, a Gantt chart for the plan was prepared. The structure followed was a sprint focused one, where each sprint had one or multiple milestones.
The Gantt chart can be found in the appendix section A.2.

3 Software development cycle

3.1 Analysis

There were a set of meetings with the stakeholders to collect and define the tasks and the work to be carried out, identifying the proper project requirements. For this, the Software Requirements Specification Document (SRS) [10] was created, where all the requirements are written. Also, during the development of the project, the document was iterated in order to ensure that there are no ambiguities and that everything is clearly described.
In the document, each requirement is specified following the structure of (Figure 8).

Requirement ID	Uniquely identifies the requirement.
Group	Defines the functional group of the requirement.
Description	The definition of the requirement.
Priority	Defines the requirement's implementation priority. Priorities are (highest to lowest) 1, 2, 3... Requirement of priority 1 must be implemented in the first system release. From priority 2 and lower, the requirements are implemented in further releases.
Risk	Specifies risk of not implementing the requirement. These are the different risk's levels listed: • Critical (C) - will break the functionality of the system. The system can not be used without this functionality. • High (H) - will impact the main functionality of the system. Some function of the system could be inaccessible. • Medium (M) - will impact some system features. The system can be used with some limitation. • Low(L) - the system can be used without limitation, but some workarounds.
References	Related use cases or requirements.

Fig. 8: Requirement structure.

3.1.1 Requirements

The list of requirements is somewhat extensive, so the most relevant requirements are listed below and are ones related to the development of the TFG-OBJ-4/5 objectives extracted from the SRS document. Some of them are summarized.

Functional requirements

- R2.01.01: The system shall support the access of users with the properties; Username, Password. Each user is uniquely identified by its name within the system.
- R2.01.02: The system shall provide at least three users with the following roles; Guest, Operator, Admin.
- R2.01.03: The system shall store the list of users, roles and permissions in the database.
- R2.01.09: If successful login, the system shall associate the user with the user roles/privileges and configure the GUI according to the user's profile.
- R2.03.01: The system shall provide the authorized user with the ability to view the list of all the public data available in the system.
- R2.05.01: The system shall provide the authorized user with the ability to view the list of the status of all the devices available in the system.
- R2.06.01: The system shall provide the authorized user with the ability to list what devices are available to the user's access level.
- R2.0X.0X: The system shall provide the authorized user with the ability to manage the devices with all the options available to the user's access level.
- R2.11.04: The system shall provide the admin user with the ability to re-establish the connection between the devices and the ROS module.

Non-functional requirements

- R1.01.01: The system shall support user interfaces for standard computers and mobile phones.
- R1.04.02: The web system shall communicate with the ROS module via websockets using the transport level protocol TCP/IP.
- R3.01.01: The system shall support concurrent users logged in the system.
- R3.03.01: The system shall implement a Role access model.
- R3.03.02: The system shall provide a user authentication mechanism.
- R3.03.04: The system shall secure the data exchange between the web server and the ROS module using SSL/TLS encryption protocol.
- R3.03.05: The system shall be able to secure the user password using the bycrypt hashing.

- R3.03.06: The system shall secure the data sent from the client to the server using POST request method.
- R3.03.07: The system shall secure the access to the web portal data from XSS and SQLi attacks.
- R4.01.01: The system shall be available for use at 24 hours a day, 7 days a week.
- R4.04.01: The system shall conform to the Polymer and a template standard.
- R4.06.01: The system shall provide the possibility to install it automatically.

Lastly, an investigation was also carried out on the hardware and tools needed to the development of the project that later were specified in the requirements document. Then, a preliminary study was made to check how to integrate them, so everything worked well. More details in the development section 3.7 in the tools 3.7.1 subsection.

3.2 Unexpected scenario

During the analysis stage, Spain entered in a confinement situation. Due to the COVID scenario, the access to the IEEC laboratories was restricted, so it was not possible to advance with the TFG-OBJ-3 objective.

Due to this situation, there was a meeting with the stakeholders in order to analyze the objectives and reschedule the project plan, prioritizing the web module (TFG-OBJ-4) over the ROS module (TFG-OBJ-3), thinking that the laboratories could be accessed after a few weeks. Seeing that this could not be possible, it was decided to use a miniature observatory prototype.

3.2.1 Robotic prototype

This observatory would be controlled in the same way as the one developed in ROS1 in C / C ++. However, this would be developed from scratch in ROS2 and Python. In this way, it could be proved that the entire system would work in the case of not being able to access the IEEC facilities due to being in confinement, as has happened.
Therefore, this would give a solution for the TFG-OBJ-3 and consequently TFG-OBJ-2.X. However, in this document, we will refer to the ROS1 system, as it is the one used in production.

3.3 Design

The project can be divided into three large modules; the ROS module, the web module and the connection module between ROS and the web.

The system was designed to be portable and flexible, so it was decided to use Docker[11] since containers enable the application portability and faster software delivery in addition to isolating containers from possible problems caused by others.
The diagram in (Figure 9) shows the connection of the two main modules, ROS and WEB. The web application and the database are located in the environment of the web portal. Then the web connects to the ROS module through the internet using the ROS-Web module.

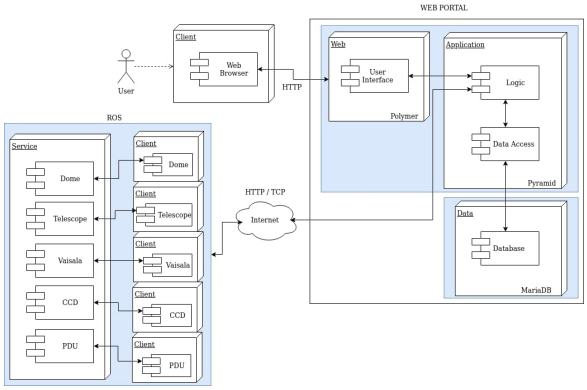

Fig. 9: Deployment diagram. The blue colored boxes are docker containers.

During the design stage, several UML diagrams have been made to define and illustrate the functionality of the systems. The Software Design Description (SDD) document [12] was also created and contains all the diagrams and the system design explanation.

The following sections go into detail about the design of each module. To make the examples more understandable and to have traceability in the information described, the same case is used in each module, the example of the observatory dome device, since all the devices follow a similar structure.

3.4 ROS Module

The current ROS1 system has a client-service architecture [13]. The ROS services are the ones interacting with the hardware. Meanwhile, the clients send petitions to the services to perform several actions.
Each device has its own service - client packages so that they can be controlled independently. However, there is also a global service that controls all devices in one. This is done for the sake of modularity and simplicity.

For example, a user can perform different actions regarding the dome control, as shown in the (Figure 10).

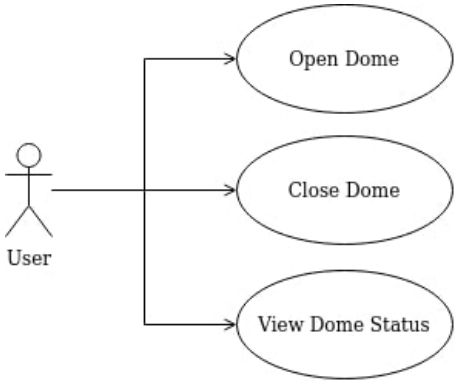

Fig. 10: ROS dome use case diagram.

In the following class diagram (Figure 11), the DomeServer contains a Dome Object. This object interacts with the driver's interface that controls the hardware.

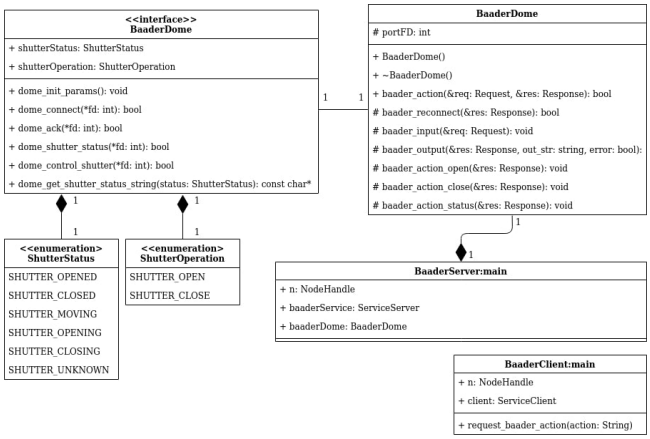

Fig. 11: ROS dome class diagram.

The (Figure 12), shows how a user asks for the status of the dome and then opens the dome. The interaction with the system is done using the following commands:

Once the server has started.

```
$ rosrun ice dome_server
```

Then perform actions through the client.

```
$ rosrun ice dome_client status
$ rosrun ice dome_client open
```

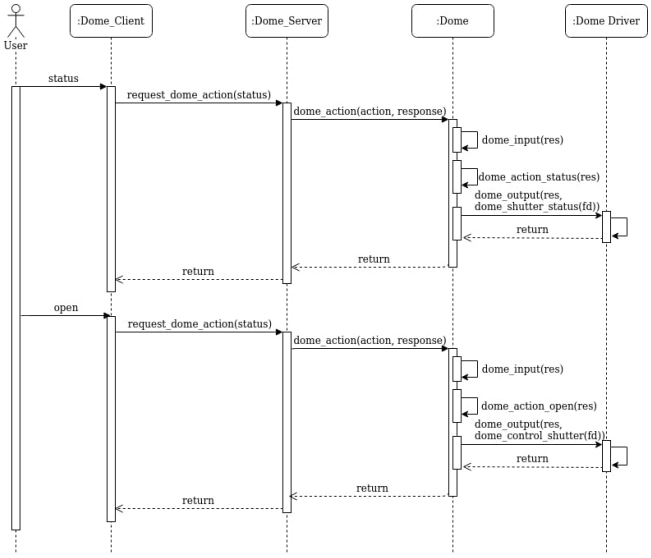

Fig. 12: ROS dome sequence diagram.

The ROS2 prototype [14], used in the development of the project, provides the same functionality as the current ROS1 software but has fewer devices and functionalities. However, it is enough to prove the system because the service-client architecture is the same and works with similar commands.
The (Figure 13) shows the 3D printed miniature prototype.

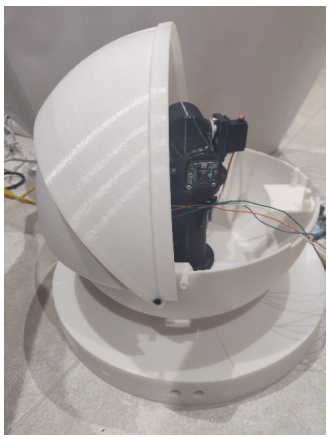

Fig. 13: ROS2 prototype.

3.5 Web portal Module

The web portal is the most specified module in the requirements. The web is designed as a portal for the interactions of different types of users. Users can access the system with the username and password and can access a different kind of views and actions according to their permissions.
There are three archetypes of users:

- Guest: This user can see the public information that is provided about the devices, but in any case, interact with them.
- Operator: User who can manage the different devices.
- Admin: User who can do everything that other users do besides manage the connections with ROS and manage users.

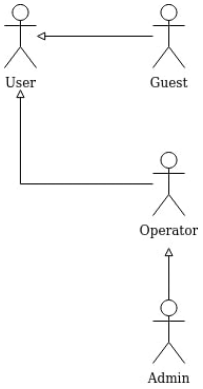

Fig. 14: Web portal user hierarchy.

The complete use case diagram can be found in appendix section A.6.

3.5.1 Backend

The web portal requires authentication and authorization in order to provide an access system with the accordingly security measures based on roles and permissions.
For this reason, a Role-Based Access Control model (RBAC)[15] has been designed to structure the user - role - permission design. Consequently, a relational SQL database has been used to apply this design and build it with high modularity, as shown in (Figure 15).

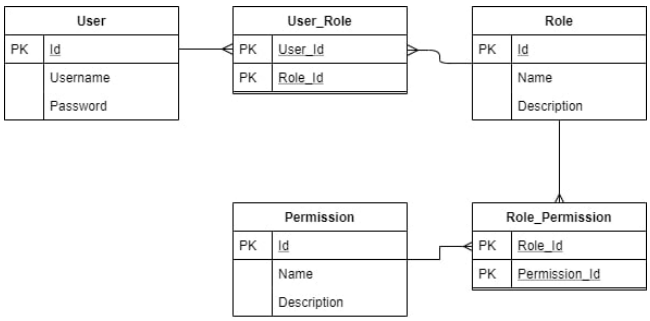

Fig. 15: Database relational model.

The database is used as persistent storage for user information and privileges. In order to access and manage the information, it was decided to use the Object-relational mapping (ORM) technique [16] to take advantage of an object definition data structure, keeping things tidy and well defined.

To manage the web portal views and actions, an Access Control List (ACL) [17] is used to map the user with its role and permission associated to each view or resource, as shown in (Table 1) and (Table 2).

TABLE 1: BASIC ACL TABLE

Action	Principal	Permission
Allow	Everyone	View
Allow	group: Operator	Use
Allow	group: Admin	Admin

TABLE 2: URL, ACTION AND PERMISSION ASSOCIATION

URL	Action	View	Permission
/	Redirect to /login		
/home		home	view
/login	Display login form. If auth succeeeds redirect to /home	login	
/logout	Redirect to /	logout	
/devices	Display all devices status	devices	view
/dome	Display dome status and actions	dome	use dome
/telescope	Display telescope status and actions	telescope	use telescope
/weather	Display weather status and actions	weather	use weather

3.5.2 Frontend

In order to design the front end, there were a couple of meetings with the stakeholders talking about the layout and page design. During these meetings, a paper prototype [18] was discussed and iterated in order to fit the requirements and the stakeholder's view of the product. That gives a highly consistent design ready to be implemented.

The (Figure 16) shows an example prototype of one of the web pages.

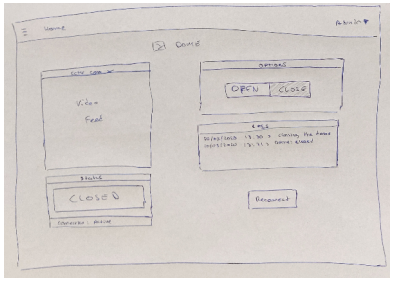

Fig. 16: Paper prototype for the dome device.

The frontend aims to be highly modular using web components and templates. These components encapsulate the client code (HTML, CSS, JavaScript) so it can be reused and scale depending on which functionalities are going to be added. An example of a component could be a login module, a toolbar, a list, etc.

3.6 ROS-WEB Connection Module

In order to receive live data and interact with the ROS system, a communication module was implemented based on WebSockets [19].
The system consists of a server in the ROS system and clients in the web system that communicates with each other using JSON [20] standard messaging, through a websocket connection using SSL/TLS protocol [21].
The (Figure 17) shows the message communication between clients and the server.

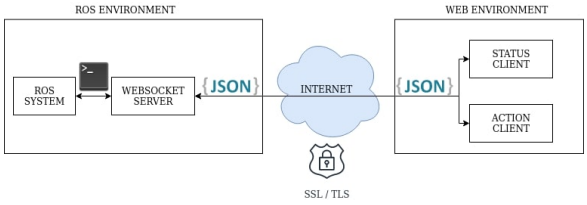

Fig. 17: WebSocket connection diagram.

3.6.1 Server

The server is responsible for handling the commands that need to be used in order to perform an action and gather data. These commands are the ones used in the ROS module (3.4) that interact with the system. Different commands can be used depending on which ROS distribution / framework is in use. That's why the commands are defined in a JSON file, giving more flexibility when switching systems.
This file is responsible for mapping the device with the commands available, as shown in appendix section A.3.

The server creates a list of devices from the JSON file. Then, assigns the commands and answers to each device. These devices are controlled by the Observatory-Handler class, that manages the interaction between the ROS module and the requests given by the clients from the Web module. The (Figure 18) shows the class diagram of the server module.

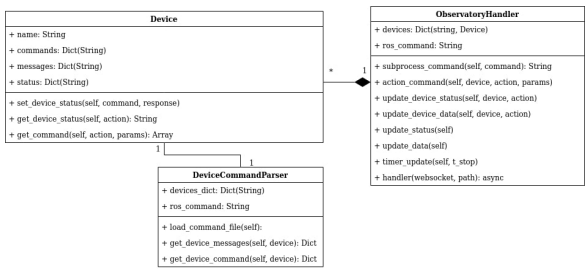

Fig. 18: Websocket server class diagram.

3.6.2 Status client

The status client asks for information every second to update the system variables regarding the data received from the devices.
The (Figure 19) shows the structure of the status message sent by the client every second and getting back the status data of all devices. This messaging is pictured in (Figure 20).

```
{'msg': 'status'}
```

Fig. 19: Websocket status message.

The status data that travels from the server to the status client (data.json) can be found in appendix section A.4.

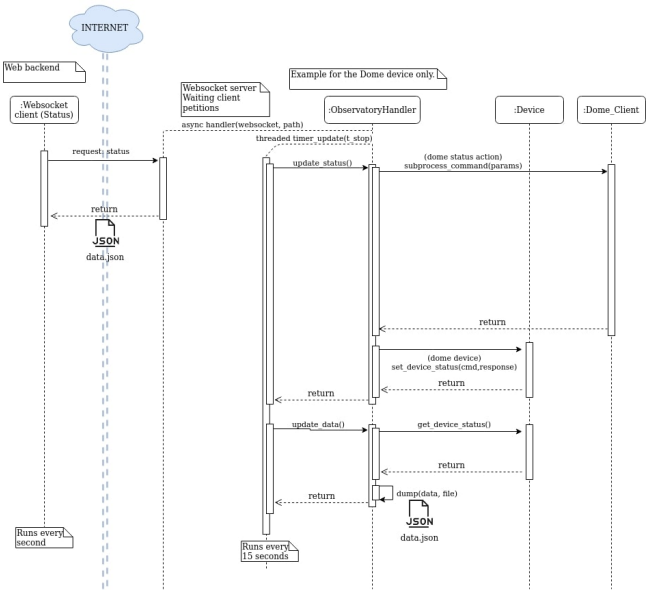

Fig. 20: Sequence diagram of the status message.

3.6.3 Action client

The action client is activated when a user makes an action, like opening the dome. The client sends this action to the server. The server receives the action, performs it and answers with the appropriate response, updating then the status of the system.
The (Figure 21) shows the structure of an action message. It uses a pattern where the device is specified, and then the main action followed by optional parameters which trigger the sequence shown in (Figure 22).

```
{
'msg': {
  'action': {
   'dome': {'action': {'open': ' '}}
   }
  }
}
```

Fig. 21: Websocket action message for opening the dome.

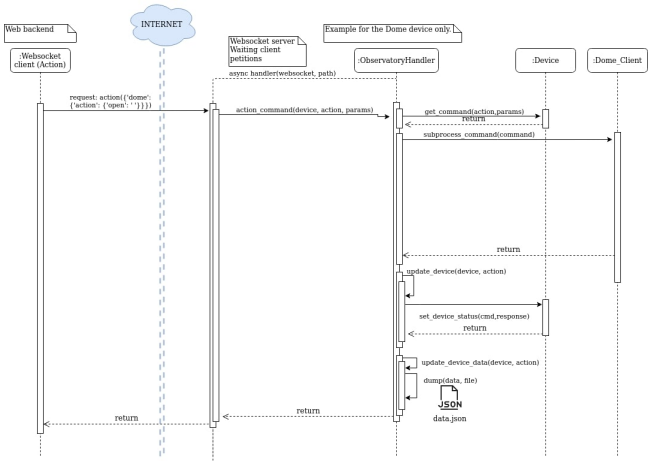

Fig. 22: Sequence diagram of an action message.

Finally, a complete sequence diagram can be found in appendix section A.7 showing the whole sequence of the data updates and an operator user performing an action.

3.7 Implementation

As an agile philosophy has been followed during the overall development, multiple design and development stages have occurred throughout the project development. That means having at least three stages of development.

Firstly, there is the ROS2 system developed to match the current ROS1 system in production. This system was developed in parallel to the analysis and design of the web portal and the ROS-Web connection module.
Secondly, once the design was finished, the ROS-Web connection module development started and was developed in parallel to the web portal.
Finally, the web portal development started once the minimal functionality of the connection module was achieved.

There was a continuous study and investigation of the technologies that were going to be used during the development due to the lack of knowledge of these and having in mind the requirements specification. This lasted from the analysis stage to the beginning of each implementation.
The next section 3.7.1 explains which tools were used during the development.

3.7.1 Tools and technologies

All three main modules were required to be developed in Python due to the great flexibility. That means setting up an infrastructure capable of working with Python. This brings

the following tools and technologies used to develop each module.

- Docker: To automatize and deploy modules enabling faster software delivery and portability.
- MariaDB [22]: Relational Database used to store the users and RBAC information.
- Pyramid [23]: The Python web framework, member of Pylons Project [24], used for the development of the web portal backend.
- Jinja2 [25]: The template language for Python used for the frontend development.
- Polymer[26]: A JavaScript framework for frontend development.
- Visual Studio Code [27] was used as the main IDE, with the addition of some plugins such as the Docker plugin, SQLTools and Remote Explorer.
- The repository working structure has been defined to have the master branch for the stable project updates / releases and a branch for each functionality in development.

3.8 Test

During the development, tests have been made to verify that the project was free from errors and to ensure that the build was solid.
Consequently, different unit tests have been designed, which check the following things:

- Functional tests to check the functionalities of each page. For example, successful or unsuccessful login or user permissions when accessing different pages.
- Tests that verify the connection and use of the data models, for example, the connection to the database, check that information can be queried from the database, check that the hash of the passwords works.
- Tests the page views, check the initialization of the pages, their content and the routes they can take.
- Tests that checks the ROS command system modularity.

Additionally, a coverage of the code was made using the previous tests to know what has been tested and what is missing to be tested.

```
---------- coverage: platform linux, python 3.6.9-final-0 ----------
Name                          Stmts   Miss  Cover   Missing
-----------------------------------------------------------
__init__.py                       9      0   100%
models/__init__.py               25      0   100%
models/meta.py                    5      0   100%
models/page.py                   10      0   100%
models/user.py                   17      0   100%
pshell.py                         7      5    29%   5-13
routes.py                        33      0   100%
scripts/__init__.py               0      0   100%
scripts/initialize_db.py         28     20    29%   11-24, 28-33, 37-46
security.py                      28      0   100%
tests.py                         39     39     0%   1-66
tests/__init__.py                 0      0   100%
tests/test_functional.py         92      0   100%
tests/test_initdb.py              7      0   100%
tests/test_security.py           17      0   100%
tests/test_user_model.py         44      0   100%
tests/test_views.py             113      0   100%
views/__init__.py                 0      0   100%
views/auth.py                    26      0   100%
views/default.py                 42      0   100%
views/notfound.py                 4      0   100%
-----------------------------------------------------------
TOTAL                           546     64    88%
```

Fig. 23: Test coverage.

A continuous integration (CI) and continuous delivery (CD) pipeline have also been integrated into the project in order to automate the execution of tests. This pipeline is executed in the GitLab repository and consists of the following steps:

- Docker-build: Checks that the image of the pyramid application is built correctly.
- Docker-compose: Checks that the image of the database is built and starts together with the pyramid application to provide service.
- Test: Run the application's tests and show the code's coverage.
- Deploy: Check that the application can produce a production build.

The pipeline configuration can be found in appendix section A.5.

The pipeline was configured to run every time a push is made to the remote repository. It is also executed when a merge of a branch is done. If the merge does not pass the test, it is cancelled. Moreover, it is designed to run all tests when merging to the master branch, which is where everything should work fine. When changes are made in other branches, the deploy phase is not executed, so it does not take too long to run unnecessary tests.

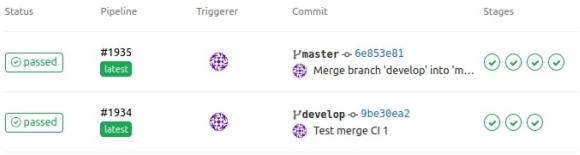

Fig. 24: GitLab CI project pipelines in a merge situation and a branch update.

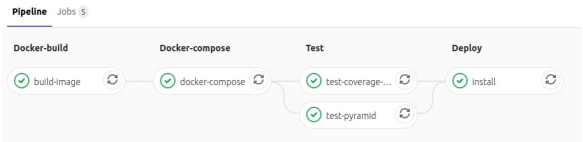

Fig. 25: CI / CD pipeline stages successfully executed.

Finally, the Software Test Description (STD) document [28] has been written describing different tests to be taken into account so that the product achieves the level described in the requirements defined in the SRS document.

4 Results

The project has lived a reschedule and a reorganization of the objectives at the middle of the development, which affected the objectives and therefore, the results.

The (Table 3) exposes the level of completeness of each objective showing its current status regarding the requirements associated to them.

TABLE 3: OBJECTIVES STATUS

Objectives	Status
TFG-OBJ-1	100%
TFG-OBJ-2 and 2.X	100%
TFG-OBJ-3	*%
TFG-OBJ-4	30%
TFG-OBJ-5	90%

The original plan was to have an improved version of the software that currently exists at IEEC, upgrade it from ROS1 to ROS2 and be able to communicate with it through a minimum web interface. Then the objectives changed to prioritize this website, making it a more extensive system along with its connection module.

However, the prototype in ROS2 has partially generated a result for the TFG-OBJ-3 objective, since it uses the new version of ROS and is based on the system that is in production.
As for the web portal, the TFG-OBJ-4 objective could not be completed due to the delay that occurred. However, the portal does the basic login functionality, discerns between users and authorizes or prohibits access according to functionalities depending on users permissions.

Finally, the connection module between ROS and the website (TFG-OBJ-5) it is almost finished. It lacks more testing to check that if the whole system works properly. The module is capable of working regardless of the ROS version or the commands used. It is able to establish a connection between the ROS system to interact with the observatory through the web portal.
Moreover, during the development, all the tasks that have been carried out are the typical ones that a software engineer does, due to the proper software development cycle. This tasks involve the creation of the documentation of the project, going from the software requirements specification, which involves an in-depth analysis of the project and stakeholders meetings, to the software design description specifying the software's architecture design with its diagrams and dependencies and finally, to the software test description where the test cases check the requirements of the project.

5 CONCLUSIONS

To conclude, it should be mentioned that the development process has been slow and time expensive because almost all technologies were unknown and had to be learned to use. Also, the creation of all the documentation and the test pipeline took a long time.

Despite not having been able to develop a front end for the web, or having completed certain functionalities of the web portal (TFG-OBJ-4), the project reached an MVP status (Minimum Viable Product) regarding the functionalities, and it is fair to say that this brings an easy continue for the development of the complete prototype. Since the MVP shows how to be done, and the design is consistent, the completion of the first product version should be as easy as reproducing the functionalities of each device.

5.1 Frontier and future development

About the project itself, the future work relays on the completion of all the device functionalities from the web portal side, and the development of a proper front end.

From a higher perspective, the ROS system could be further developed to the point of having all the functionalities that a robotic observatory should have, not only the control of the different devices but also an automatic planner to observe the sky through requests in an autonomous way and security and meteorological prevention systems. This could lead to a restructuring of the current ROS system and the upgrade to the newest version ROS 2. Furthermore, the current ROS-Web connection module could be replaced by a ROS node that interacts with the web backend or event better have the web backend in a ROS node, that way the middleman is removed and the system is a bit more compact. In that case, the web portal is more of a versatile system, so in the end, it would only be a matter of adding more functionalities.

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere thanks to my advisor Lluís Gesa for continuous support and advice. His guidance was very useful both in the management of the thesis and in the development of the project.
I would also like to thank Xavier Otazu for helping in the last phase of the thesis and special thanks to Francesc Vilardell for all the support given during the development.

REFERENCES

[1] A. J. Castro-Tirado, "Robotic Autonomous Observatories: A Historical Perspective," in , vol. 2010 of *Advances in Astronomy*, p. 8, Apr. 2010.

[2] J. Colomé, J. Sanz, F. Vilardell, I. Ribas, and P. Gil, "OpenROCS: a software tool to control robotic observatories," in , vol. 8451 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, p. 845127, Sept. 2012.

[3] "Institut d'Estudis Espacials de Catalunya." http://www.ieec.cat/en/content/18/the-ieec.

[4] F. Vilardell, G. Artigues, J. Sanz, Á. García-Piquer, J. Colomé, and I. Ribas, "Using Robotic Operating System (ROS) to control autonomous observatories," in , vol. 9913 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, p. 99132V, July 2016.

[5] O. Robotics, "About ROS and documentation." https://www.ros.org/about-ros/.

[6] O. Robotics, "ROS2 documentation." https://index.ros.org/doc/ros2/.

[7] J. Highsmith, A. Cockburn, M. C. Paulk, and P. E. McMahon, "Agile software development," Oct. 2002. http://agilesweden.com/doc/oct02.pdf.

[8] CbtNuggets, "How do Kanban, Scrum and Lean relate?," Oct. 2017. https://www.cbtnuggets.com/blog/career/management/how-do-kanban-scrum-and-lean-relate.

[9] "Trello list-making application." https://trello.com/en.

[10] J.-P. Eisenbarth, "A latex template for a software requirements specification that respects the IEEE standards," May 2020. https://github.com/jpeisenbarth/SRS-Tex/blob/master/srs.tex.

[11] "Docker, why docker?." https://www.docker.com/why-docker.

[12] A. Sankarana, A. Samsonyuka, and M. Attarm, "Software design document, example," Dec. 2010. https://arxiv.org/pdf/1005.0595.pdf.

[13] O. Robotics, "ROS service-client communication." http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams.

[14] F. Domene, "ROS2 observatory," June 2020. https://github.com/fdomf/ROS-observatory.

[15] "Role-based access control model." https://en.wikipedia.org/wiki/Role-based_access_control.

[16] "Object-relational mapping technique." https://en.wikipedia.org/wiki/Object-relational_mapping.

[17] "Access control list." https://docs.pylonsproject.org/projects/pyramid/en/latest/glossary.html#term-acl.

[18] "Paper prototyping technique." https://en.wikipedia.org/wiki/Paper_prototyping.

[19] I. Fette and A. Melnikov, "Websocket protocol." https://tools.ietf.org/html/rfc6455.

[20] D. Crockford, "Introducing JSON." https://www.json.org/json-en.html.

[21] T. Dierks and E. Rescorla, "Transport layer security (tls) protocol," Aug 2008. https://tools.ietf.org/html/rfc5246.

[22] "MariaDB: The open source relational database." https://github.com/MariaDB/server.

[23] "The pyramid web framework." https://docs.pylonsproject.org/projects/pyramid/en/1.10-branch/.

[24] "Pylons project." https://pylonsproject.org/about-pylons-project.html.

[25] "Jinja temmplating language." https://jinja.palletsprojects.com/en/2.11.x/.

[26] "Polymer project." https://www.polymer-project.org/.

[27] "Visual studio code." https://code.visualstudio.com/docs.

[28] L. Ericson and S. Shine, "Minutia deviation tool: Software test description (STD), example," Dec. 2015. https://www.ncjrs.gov/pdffiles1/nij/grants/249555.pdf.

APPENDIX

A.1 Trello Board

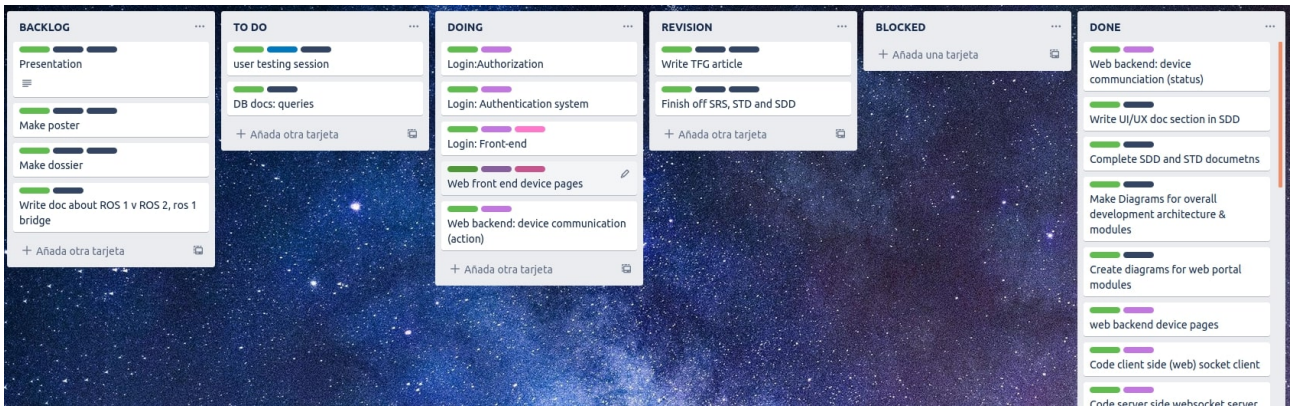

Fig. 26: Trello board status during development.

A.2 Gantt chart

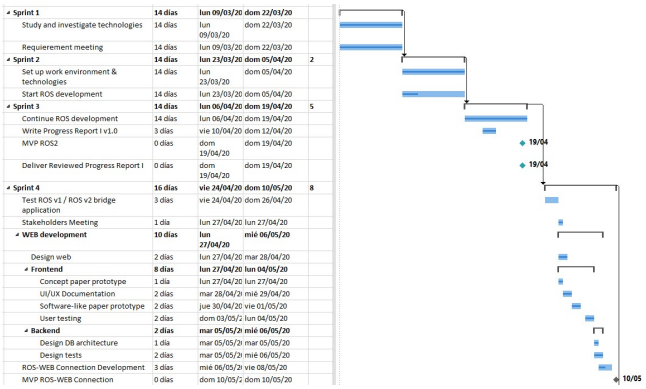

Fig. 27: Gantt chart sprints 1 to 4.

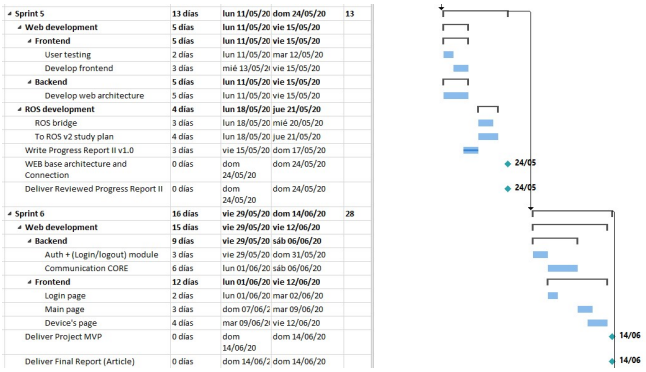

Fig. 28: Gantt chart sprints 5 and 6.

Fig. 29: Gantt chart sprints 7 and 8.

A.3 JSON command file


```
{
    "command": {
        "ros": "ros2 run"
    },
    "devices": {
        "dome": {
            "commands": {
                "info": {
                    "status": "dome_servcli client status",
                    "rt": "dome_servcli client test"
                },
                "open": "dome_servcli client open",
                "close": "dome_servcli client close"
            },
            "messages": {
                "status": {
                    "opened": "the dome is opened",
                    "closed": "the dome is closed",
                    "moving": "the dome is moving"
                },
                "rt": {
                    "test": "the dome is test"
                }
            }
        },
        "weather": {
            "commands": {
                "info": {
                    "status": "weather_servcli client info"
                }
            }
        },
        "telescope": {
            "commands": {
                "info": {
                    "getcoord": "telescope_servcli client getcoord"
                },
                "goto":"telescope_servcli client goto"
            }
        }
    }
}
```


Fig. 30: JSON file where the commands are specified for each device.

A.4 JSON data file


```
{
    "status": {
        "dome": {
            "status": "closed",
            "rt": " Invalid dome action\n"
        },
        "weather": {
            "status": " ['Clouds', 'broken clouds', '18.74', '1016', '77',
        },
        "telescope": {
            "getcoord": " Alt:10.000000  Az:10.000000\n"
        }
    }
}
```


Fig. 31: JSON data message (data.json) that travels from the server to the status client.

A.5 CI / CD pipeline


```
stages:
    - docker-build
    - docker-compose
    - test
    - deploy

build-image:
    stage: docker-build
    script:
        - docker build -t rocs_webp:python .

docker-compose:
    stage: docker-compose
    needs: ["build-image"]
    before_script:
        - docker info
        - apk add py-pip python-dev libffi-dev openssl-dev gcc libc-dev make
        - pip install docker-compose
    script:
        - docker-compose -f docker-compose.yml config
        - docker-compose -f docker-compose.yml build

test-pyramid:
    stage: test
    needs: ["docker-compose"]
    before_script:
        - apk add python3-dev libffi-dev openssl-dev gcc libc-dev make
        - pip3 install --upgrade pip
        - pip3 install -r requirements-ci.txt
    script:
        - pytest -q --disable-pytest-warnings

test-coverage-pyramid:
    stage: test
    needs: ["docker-compose"]
    before_script:
        - apk add python3-dev libffi-dev openssl-dev gcc libc-dev make
        - pip3 install --upgrade pip
        - pip3 install -r requirements-ci.txt
    script:
        - pytest --cov=app app/tests.py -q --disable-pytest-warnings --cov-report=term-missing

install:
    stage: deploy
    needs: ["test-pyramid"]
    only:
        - master
    before_script:
        - apk add python3-dev libffi-dev openssl-dev gcc libc-dev make
    script:
        - python3 setup.py sdist
        - pip3 install dist/*.tar.gz
```


Fig. 32: GitLab CI / CD pipeline structure.

A.6 Use case diagram

Fig. 33: Complete use case diagram. The blue colored use cases are not considered in the current SRS document.

A.7 Dome example sequence diagram

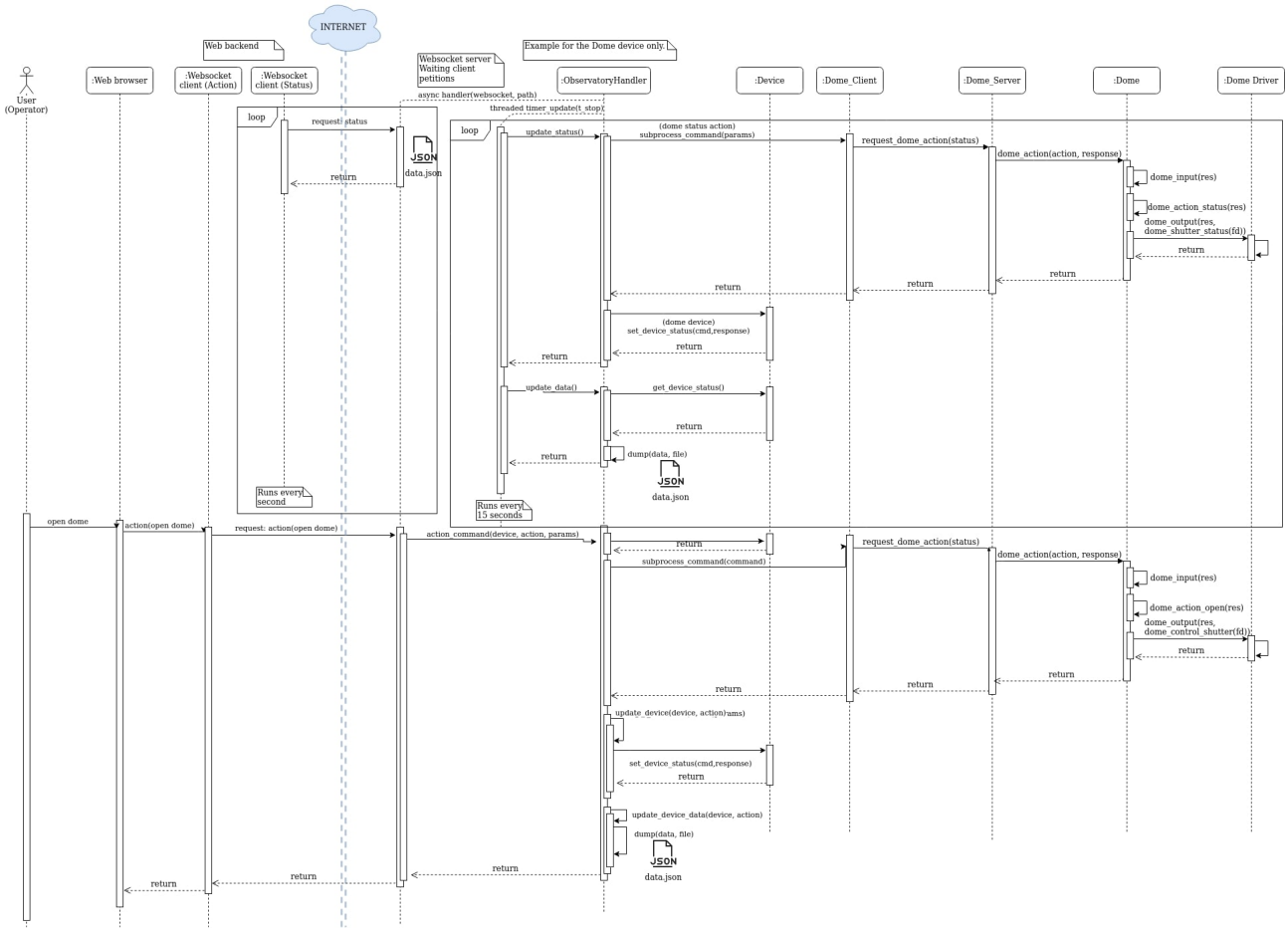

Fig. 34: Sequence diagram showing the whole process of updating data and performing an action.