FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mock Testing Framework for a Fire Detection System

Guilherme de Castro Oliveira

Mestrado em Engenharia de Software

Supervisor: João Pascoal Faria

Second Supervisor: Ana C. R Paiva

July 30, 2021

Mock Testing Framework for a Fire Detection System

Guilherme de Castro Oliveira

Mestrado em Engenharia de Software

July 30, 2021

Abstract

When working in the area of software development, testing has become one of the most important phases when developing a system. When testing a system with multiple dependencies, the process can turn out to be difficult and time-consuming. To test this type of system, developers can either instantiate all the dependencies or use mock objects to simulate the dependencies. Studies show that although mock testing is widely used, there is a lack of scientific knowledge related to the subject that would be fundamental to guide the creation of tools and solutions to help in the widespread use of mock testing.

As the title suggests, the current document explains the process of creating a framework to test help test and enhance a fire detection system. Most fire detection systems are connected to a central panel; This panel usually has a way of connecting a computer to it, which allows the fire detection system to be configured through software installed in the computer. When this software is connected to the physical system, composed of the panel and the fire detection system, it is called a cyber-physical system. The goal of this project is to use mock testing to test the software referred to before. In order to test software that depends on the physical part of the system to execute most of its features, without it being expensive and time-consuming, developers have to simulate the physical components to allow for a faster and cheaper process.

The project explained in this document is a framework capable of mocking the existence of the physical part of the system. This framework obtains data from the tester, creating rules that are used to define how the framework will respond to the commands received by the software used to configure the fire detection system. The framework is able to create a connection with the configuration software, and imitate the behavior of the physical system, by receiving the commands, decoding them and reply by using the rules previously mentioned.

The present document contains the state of the art; a literature review was made to gather information in the subject, related to mock testing and solutions to assist in the development of mock testing software, followed by a research of the available tools, where each tool is compared against the requirements of the project, leading to the selection of the best option for this specific problem but also briefly describing each tool that was researched. After the state of the art is presented, the implementation of the tool started, the decisions for the technologies and architecture are introduced, each component of the system is thoroughly explained, as well as the interaction between components.

Keywords: Mock Testing, Mock Object, Fire Detection System, Cyber-physical systems, Communication Simulation, API Simulation

Resumo

Na área de desenvolvimento de software, testar é, cada vez mais, uma das fases mais valorizadas ao desenvolver um sistema. O processo de testar um sistema com múltiplas dependências pode ser difícil e demorado, os programadores tendem a encontrar soluções para minimizar essa falta de eficiência. Uma das principais soluções para aumentar a eficiência deste processo nos sistemas referidos é a utilização de objetos simulados, que simulam a presença das dependências necessárias. Estudos demonstram que, embora os testes simulados sejam amplamente utilizados, há uma carência de conhecimento científico relacionado com o assunto, o que dificulta a orientação e a criação de ferramentas e soluções que auxiliem nos testes simulados.

O documento atual foca-se na utilização de testes simulados num sistema específico, um sistema de detecção de incêndios. Um sistema de detecção de incêndio que pode ser configurado através de um software, quando o software está conectado ao sistema físico, é um sistema ciberfísico, e o objetivo deste projeto é testar o software utilizado para configurar o sistema de detecção de incêndios. Para testar um software que depende da parte física do sistema para executar a maioria das suas funcionalidades, mantendo o processo breve e acessível, os programadores necessitam de simular os componentes físicos.

Foi feita uma revisão da literatura para levantamento de informações no assunto e presente documento demonstra o estado da arte relacionado com testes de simulação e ferramentas para auxiliar no desenvolvimento de softwares para testes de simulação, seguida de uma pesquisa que abrange as várias ferramentas disponíveis, onde cada ferramenta é comparada com os requisitos do projeto, concluindo com a seleção da melhor opção para este problema específico, mas também descrevendo resumidamente cada ferramenta que foi pesquisada e experimentada. Após a apresentação do estado da arte, é detalhado o planeamento e implementação da ferramenta, as decisões sobre as tecnologias e arquitetura são apresentadas, cada componente do sistema é explicado detalhadamente, assim como a interação entre os componentes.

Keywords: Testes simulados, Objetos Mock, Sistema de Deteção de Fogos, Sistemas ciberfísicos, Simulação de comunicação, Simulação de API

Acknowledgements

Throughout the last months, while writing this dissertation, I have been strongly supported and I have several parties to thank..

I would first like to thank my supervisor, Professor João Faria Pascoal, and my second supervisor, Professor Ana Cristina Ramada Paiva, whose expertise was invaluable in organizing my ideas, designing the project and writing the dissertation.

I would also like to thank the members from Bosch, Paulo Costa and João Ferreira, for their valuable guidance throughout the project. You provided me with the tools that I needed to choose the right direction and successfully complete my dissertation. Your positive feedback motivated me to work harder and push myself for this project.

In addition, I would like to thank my parents for their wise counsel. You have always been there for me. I could not have completed this dissertation without the support of my sweet girlfriend, Rita Araújo, and all my friends, but especially Tiago Ramalho, who shares the same line of work and provided stimulating discussions about the subject as well as happy distractions to rest my mind outside of my research.

Finally, I would like to acknowledge my colleagues from my masters at FEUP for the wonderful environment felt throughtout the whole course. I would particularly like to single out three of my colleagues, Beatriz Tavares, Rúben Jesus and Francisco Serrão. I want to thank you for your constant and dedicated support.

Guilherme Oliveira

"Simplicity is a difficult thing to achieve."

Charlie Chaplin

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
CPS	Cyber-Physical Systems
DLL	Dynamic-Link Library
FOSS	Free and Open-Source Software
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JSON	JavaScript Object Notation
POC	Proof of Concept
REST	Representational State Transfer
RPS	Remote Programming Software
SMTP	Simple Mail Transfer Protocol
SUT	System Under Test
TCP	Transmission Control Protocol
WWW	World Wide Web

Chapter 1

Introduction

The current document was created to demonstrate the work that have been done to achieve the goals, set by Bosch, of one of the many projects within the scope of their ambitious “Safe Cities” project . “Safe Cities” project, with a partnership with the University of Porto, aims to respond and anticipate the challenges faced by modern urban societies, increasingly dependent on technological evolution in the fields of data transmission, storage and remote intelligent processing, to meet their needs for security, privacy, comfort and efficiency. This project in specific, aims to create a mock testing framework to help test a fire detection system. A fire detection system in a large plant or building is composed of a variety of components, e.g., smoke detectors, actuators, alarms, all centralized by a control panel, which means that every component is connected to this panel. The central panel provides an overall check of the system, for the user to know if any component is activated or not working as it should be. The panel , in some cases, has a keypad that allows for a simple configuration of the physical system, and also provides an input connection to connect to a computer, turning the system into a cyber-physical system, where the software (cyber part) will communicate with the physical part of the system to configure it. An example of a fire detection system in a large building can be seen in Figure 1.1.

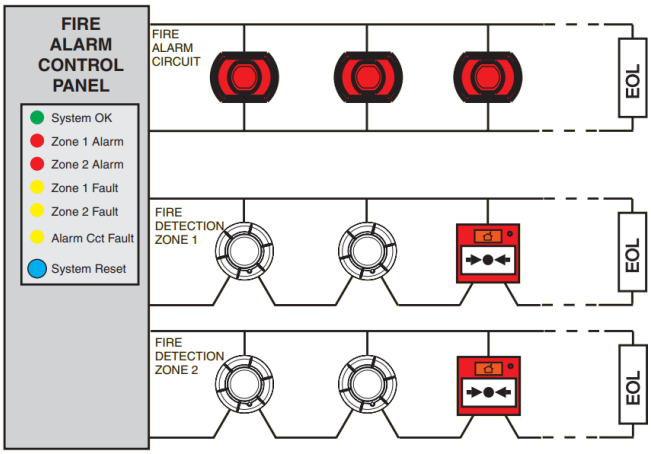


Figure 1.1: Example of a Fire Detection System [19]

In a more complex system, many programmable features are available, and achieving this configuration through a keypad can be challenging and time-consuming. By connecting the computer to the panel and using a specific software to configure the physical system, the process becomes easier and more efficient than executing the process through the panel's keypad, allowing for a better and more intuitive configuration of the system. The configurations done through the computer are stored in the control panel and the computer can then be removed [19].

1.1 Challenge

The software used to communicate with the panel and responsible for configuring the system, which will be referred to as Configuration Software for now, needs to be tested, to prove that it is functional and its features work as they should. To make testing possible, the physical part of the system needs to be present alongside the Configuration Software, so that the commands sent by the Configuration Software are received by the panel and consequently the response from the panel is sent back to the Configuration Software. This necessity for the physical part of the system to be present represents an hindrance for testing the Configuration Software for a number of different reasons [18]:

- Some physical systems are expensive and having a set of devices present for each tester in every occasion the tests need to be run leads to a both greater time-investment and a financial disadvantage;
- Each different set of physical components attached to the control panel represents a different set of tests that must be done. If the software is prepared to communicate with several different sets of components, e.g. through abstraction, it would have to be connected to each of those sets individually to test every feature available in the Configuration Software;
- Having the two boundaries of the Configuration Software and physical system present in the testing scenarios, demands a bigger knowledge and coordination between the team responsible for setting up the physical part and the team developing the tests for the Configuration Software;
- Testing failure scenarios (e.g. timeouts) with a real system is hard. The physical components that compose the system were not made to fail when they were built and put together, therefore simulating this scenarios is difficult.

1.2 Goal

The goal of this work is to solve the issues described before, and in order to achieve that, the implementation of a Simulation component, which will be referred to as Simulator on the rest of the document, needs to be done. The Simulator will work as a Mock, allowing a component, in our case the Configuration Software, to be tested in isolation and eliminating the need for the physical

part of the system to be present during test execution. In order for that to be achieved the Simulator must replace the physical components of the system so it should respond to the commands sent by the Configuration Software, mimicking the scenarios executed when the real physical components are present. To help achieve these goals, Bosch has provided a working document with detailed specifications for the functional objectives of the project, which will be referred throughout the document [17].

1.3 Structure

The rest of this paper is organized as follows. Chapter 2 has the objective of collecting the most recent data related to the project, creating the state of the art used to lead the rest of the process and preventing future mistakes. Chapter 3 has the design and implementation of the project. First, it explains the design of the solution, containing the selected technologies and the designed architecture of the project. Next, the implementation of the solution is described, presenting the different steps throughout the months of implementation and explaining each component of the system with detail, as well as how the different components interact. Chapter 4 has a detailed example of the system usage, explaining each step until the Communication Software has connected and logged in to the Simulator. This chapter also validates the work done, by showing the results that were expected from the framework. In Chapter 5 the reader can find the the Conclusions, where the results are listed and compared with the problems stated in the first chapter, everything is summarized, and the future work is pointed out.

Chapter 2

Background and State of the Art

2.1 Basic Concepts

2.1.1 Mock Testing

As in many other fields, testing in Software Engineering has become one of the most important phases of development [1]. Software testing is divided into several different techniques, Unit Testing being one of the most normally used. It is a functional testing approach that compares the real output of a software component executing a task with the output that was expected and checking that both outputs match, proving that the execution of the task is being successfully done.

While testing software, it is fairly common for an artifact to have dependencies, either external to the software or from other elements in the software. Therefore, it can become demanding to properly create a complete and convenient set of unit tests [2]. As a result, developers began to use mock testing to remove the need of these referred dependencies and allowing for unit testing to be done inside the boundaries of the specific component that is being tested in the system.

When working with mock testing, the developer tries to replace a real software dependency with a mock object that simulates its relevant features [2]. An example of mocking is, while testing a component that accesses an external dependency (a database) and retrieves data, simulating the database and returning mock data, removing the need to connect to the real database, which is a slow process, and, therefore, allowing the tests to run faster and focus on the boundaries of the component being tested, without the risk of test failure coming from the external dependency.

Content repositories, web services and external dependencies in general are the slowest components and consequently tend to be the most mocked dependencies [2] [3].

2.1.1.1 Mock Objects

Mock objects have the objective of replacing dependencies of a software by replicating their most relevant aspects and features. In most cases, dependencies return results based on input values sent to them, therefore, that is the frequent design utilized in a mock object, as we will see in the next example.

In the code example shown in the "Listing. 2.1", the tool Mockito is used, which is a Java framework used for mocking and one of the most used frameworks for this purpose [2]. Each line of code will now be explained:

- In the first block of code, in line 4, the mocked object is created, in this case a Java List Object is mocked called "mockedList" using the mock function provided by Mockito;
- In the second block of code the behaviour of the mocked list is defined. In the line 8 we define that when a function to get data from the first position is called, the mocked object is meant to return a string containing the word "first". In the line 9, it is defined that the mocked list is supposed to return a true value if a function to add data to the list is invoked and receives a string;
- In the third and last block of code, the object and definition made previously are tested, with the help of "assertEquals", in line 14, that compares the returned value from calling the function to get data from the first position in the mocked list with the string containing "first", if they are the same the test is correct. In line 15, by using the "assertTrue" function we check if the returned value from calling a function to add a string to the mocked list is a true value, which will make this test a success.

```
1  @Test
2  public void mockingAListObject() {
3     //1) Creating mocked list
4     List<String> mockedList = mock(List.class);
5
6     //2) Defining the list behaviour
7     when(mockedList.get(0)).thenReturn("first");
8     when(mockedList.add(anyString())).thenReturn(true);
9
10    //3) Testing with the mock object
11    assertEquals(mockedList.get(0), "first");
12    assertTrue(mockedList.add("A string"));
13 }
```

Listing 2.1: Mockito Example

The goal of the previous example, is to make it clearer how the process of mock testing can help analyse a system without the need of having real data or components that need specific inputs. The example demonstrated in "Listing. 2.1" shows how it can be done in a monolithic software system, object-oriented system. But in a distributed system, such as the one envisioned for this project, the mocking of certain components and external dependencies can be a lot harder to achieve, requiring a more dedicated work to ensure that the software under test boundaries are respected and that the simulated components' features match with the original, maintaining the behaviour from the real system. The second type of simulation referred is, for example, frequently

used to mock an external payment service, in a system where the user must process a purchase and receive the successful action from the external service, it is easily understandable why processing a real payment every time the system needs to be tested and the reason why mocking this external dependency is mandatory. This example can be related to the project described in this document, since having the physical system present for each testing process is too expensive and it is also mandatory to simulate the its presence.

2.1.1.2 Mock Testing Challenges

Even though, mocking is a good solution to the problem presented in this document, creating a suitable mocked component may be challenging, especially when the simulated part is a complex system with several different features, which is the case in the system referred to in the present document. The major challenge related to mocking, is being able to transfer the behaviour from the original component to the mock, maintaining their compatibility, especially if the original component is prone to change, forcing the mock to be updated regularly [2].

2.1.2 Cyber-physical Systems

Cyber-physical system (CPS) is a term used to describe complex distributed embedded systems that integrate a physical part (e.g. sensors, actuators and other physical components) and a computing technology (cyber part) that is responsible for the coordination, monitoring and configuration of the physical components through communication and command sending. CPS have an immense range of useful interaction with the real world, especially in the current time where smart solutions and the Internet of Things (IoT) is one of the most discussed topics in software engineering [6]. A fire detection system configured by a software is a CPS, hence the importance of describing what is a CPS, why it is so important to test it and the difficulties attached to it. Examples of CPS usage include aerospace, transportation, factory automation, security, environmental control and medical systems.

With this vast range of applications of cyber-physical systems, and linked with the complexity and diversity associated with CPS, leads to security and reliability risks which can cause massive setbacks on companies and systems relying on this type of technology. This failure rate, allied with the vast range of application of CPS discussed before, makes the assurance of security and reliability a topic of great importance before the system is implemented into the real world, and that is why testing is such a fundamental step in CPS development, as in many other subjects.

There is a dearth of data about cyber-physical systems in general but especially in data related to testing these systems. From the literature review performed for the present document, we found that CPS developers are generally unfamiliar with traditional software verification and validation tools and methodologies and that the current approach used for developing and testing CPS is trial-and-error [5].

When developing software to work side-by-side with physical systems, there are a great amount of common challenges through most of the situations that involve this process:

- Being one of the most obvious, the heterogeneity of existing devices and physical systems. The amount of different devices, and combinations between devices, that can be achieved generates an immensity of different case scenarios that can be tested. The innovative approaches to abstraction and architectures that allow the integration and interoperability of heterogeneous systems have been discussed through the last years as an attempt to bridge this diversity, but there still is a lack of efficient and robust options in this area of work [8]. Developing a software for a specific physical system means the software may have to be updated on the occasion that the physical components are changed.

- In a specific cyber-physical system, several needed components may have limited capabilities. These limitations can become an issue when developing a software that communicates with the components. Some typical examples of limitations can be wireless communication, storage capabilities and processing limitations.

- In most cases, CPS applications are critical, implying that if the system fails there are critical consequences, e.g., a production line that stops leading to a loss of capital. With this in mind, it's critical that cyber-physical systems are reliable, and to achieve this robustness it is important to test both the cyber and physical parts of the system.

- The challenge that supports this project the most, as discussed in the first section of the document, is the expense accompanying the presence of the physical part of the system, each time the test scenarios need to be executed. In order to test the software responsible for coordinating the physical components, the entire system needs to be present. This is not only expensive both financially and in time consumption, but also makes the testing of failure scenarios arduous to achieve. Furthermore, in most cases the software is prepared to manage several different physical systems with different components, meaning that, to test every available feature of the software, there is a necessity of having all the different components, that fill every feature provided by the Communication Software, available to connect to the software and execute the tests.

Focusing on the last challenge mentioned in the previous list, related to the expensive testing in situ, and to strengthen the motivation of the project discussed in the introduction, it becomes clear that, in order to reduce financial costs and time spent on developing and testing software for cyber-physical systems, a reliable and high-quality simulation of the physical components of the system is of the utmost importance. A framework allowing the user to design test cases with virtual simulated responses mirroring the operations of the physical part of a CPS will make developing and testing the software a much more economic and efficient process. Additionally, testing failure scenarios in a real physical system is an obstacle, since the physical components that compose the system are built to work, therefore, having a scenario where failure could be testes would

mean that some alterations would have to be done directly to the components. Having a simulated physical system will remove that obstacle of testing failure scenarios, making it easier to simulate a failure message, e.g., a timeout [5].

2.1.3 Web Service Communication

As will be explained in the next section, the tool envisioned for this project will contain a component that allows the user to communicate with the Simulator through an application programming interface (API) with HTTP requests. Therefore, it is important to understand the typical architecture used for communicating with a web service. The most commonly used architecture for a web service is the representational state transfer (REST). An API that uses this architecture is called a RESTful API. In the API, different Uniform Resource Identifier (URI), also called endpoints, are defined. An example of an endpoint is *"http://api.example.com/"*. The communication with the endpoints is done through HTTP requests sent by the user. The requests have four main different methods in the REST architecture [18]:

- GET - As the name suggests, this method is used to get resources from the service, when the user wants to receive data that is already stored in the service to either use or inspect the GET HTTP method is the one used for that purpose;
- POST - The POST method is meant to be used to generate resources in the server. This HTTP method is frequently attached with data in the form of JavaScript Object Notation (JSON), which the service will use to define the objects containing the data that the user wants to add to the service storage;
- PUT - This method enforces the update of a resource, is also frequently accompanied by data in form of a JSON message, where the data contains the details of what needs to be changed, by sending an HTTP PUT request to an endpoint that specifies the resource that the user desires to update, it aims to change some aspects of that resource that already exists in the service storage;
- DELETE - When using this method the goal is to delete a resource from the server's storage. This HTTP method needs to be sent to an URI that is specifically related to the resource that the user wants to remove from the database;
- PATCH and HEAD - There are two other methods that are not frequently used in the development of web services [18]. The PATCH request is similar to the PUT request, but when a resource is updated with PUT it replaces the whole resource with the new data, unlike the PATCH that only changes properties of a resource instead of fully replacing. The HEAD request is similar to the GET request but only returns data concerning the properties of the resource that would be returned if a GET request is used, for the user to be informed of what will be retrieved from the server if the GET is used. At the present time, these two methods are not planned to be included in the current project.

This basic knowledge of how the architecture of a RESTful API works will be important in the Section IV, more specifically when explaining the experimental evaluation done.

2.2 Background

2.2.1 Remote Programming Software

The Remote Programming Software (RPS) is the software developed and used by Bosch to communicate and configure systems similar to the fire detection system described previously, in the previous sections this software was referred to as Configuration Software but for the rest of the document, will be referred as RPS.

2.2.1.1 Graphical User Interface

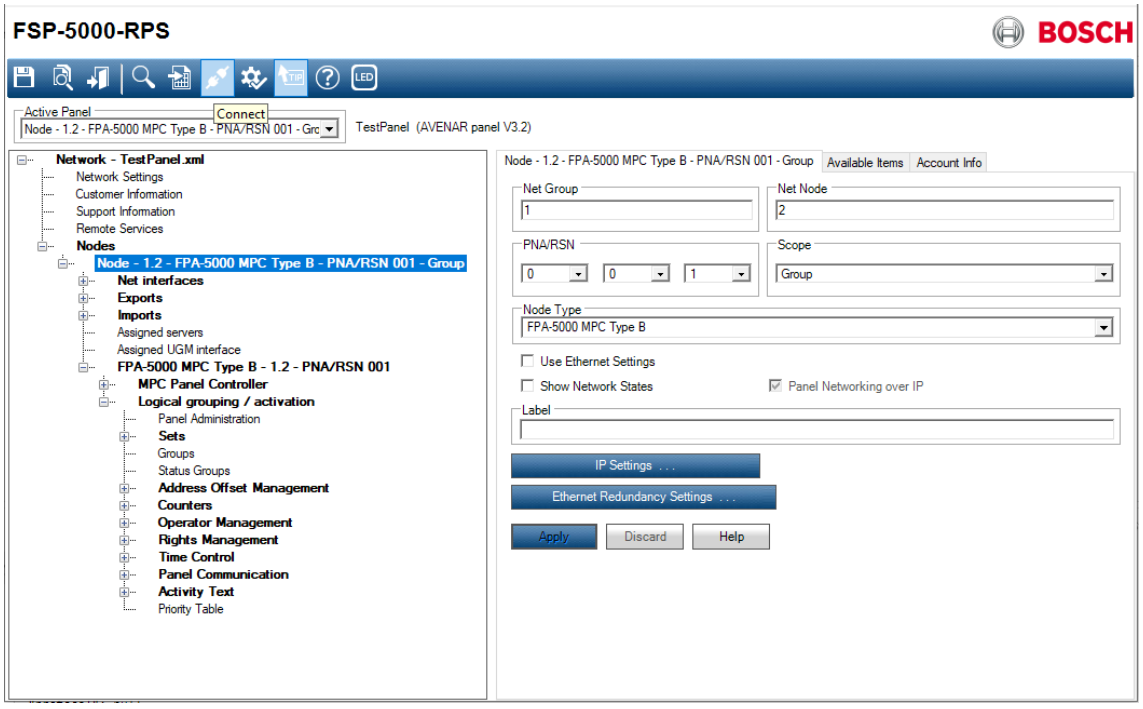


Figure 2.1: Panel Configuration

The RPS has a graphical user interface (GUI) that helps the user throughout the process. The RPS is a legacy tool that uses Windows Forms for the Graphical Interface, this is something that Bosch is currently trying to upgrade gradually, in order to try to prevent that their clients struggle with the changes done to the tool. The home page of the RPS allows the user to create a list of panels, from a different selection of panel generations, panel type and version. This will inform the RPS about the specifications of the panel it is trying to communicate with.

After selecting the desired panel, the user will be shown a window with all the different specifications of the panel, as shown in Figure 2.1. In this page the user is provided with fields to change the predefined values to match the Panel values, or, in this case, the Simulator values.

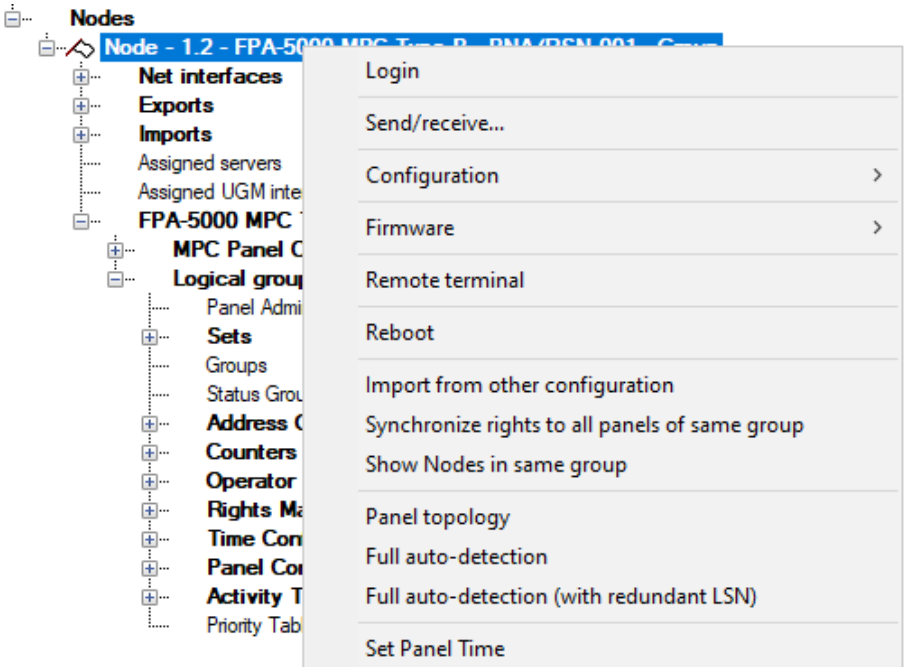


Figure 2.2: Panel Communication

Finally, the user can send the commands to the panel it is communicating with from this window, by initially using the "Connect" button on top of the window shown in Figure 2.1, and, after the connection is successful, which can be confirmed by the flag next to the Node in Figure 2.2, the RPS can send commands from the listed options in Figure 2.2. After the login command is successful the flag signaling the connection with the panel should turn green. This part will be explained in a more detailed manner in the Usage Example Chapter, where most of the validations are done.

2.2.2 RAMV Protocol

Although the project started by using a fictitious protocol to create an initial Proof of Concept (POC), the real protocol is called RAMV Protocol and is what the RPS uses to communicate with the external environment. The simplified protocol to achieve the POC will be explained next, as well as the intentions in having this POC developed. The actual RAMV Protocol has an extensive list of commands and responses, every command follows a structure, with some small differences between each command.

One of the most important phases of the project was to learn the structure of the commands and responses so they could be recreated in the Simulator to allow for smooth communication with the RPS. The two tables above demonstrate those exact structures. As we can see by comparing

Table 2.1: Command Structure

Size	1 bit	7 bits	1 Byte	2 Bytes	<Length> Bytes
Field	Reply Flag	Message Type	Sequence Number	Length	Data

the two tables the first four fields of both the command and response structure are identical as well as the last. Each field will be explained:

- **Reply Flag** — This flag indicates if the message is a reply message to a previously sent command, in which case it will be set to 1, or if it is a command, in which case it will be set to 0;
- **Message Type** — The message type indicates the type of command that is being sent or received. Both the command and the response must share the same message type and are distinguished by the previous field "Reply Flag". There are over 50 Message Types but we chose to focus on the ones that are responsible for initiating a connection, which will be discussed further in more detail;
- **Sequence Number** — The sequence number is incremented with each new command received. For the response of a specific command, the sequence number must be the same as the command but this number is handled independently for commands received in the Simulator side and for responses received in the RPS, allowing for supervision of the communication and ensuring that no command or response is lost in the communication path during the message trade;
- **Length** — This field is used to indicate the size of the Data field in bytes, which varies between different Message Types and even in the same Message Type but in different scenarios. In the response structure, is added 3, which is the size of the Continuous Flag, Status, and LUCA Error fields, and added n, which is the size of the LUCA Error Text field;
- **Data** — Although the name of the field is the same, the content changes between each command and response, this field contains the data sent in the message and each Message Type has a format for this field.

In the Response there are 4 more fields:

Table 2.2: Response Structure

Size	1 bit	7 bits	1 Byte	2 Bytes	1 bit	7 bits	2 Bytes	n Bytes	<Length> Bytes
Field	Reply Flag	Message Type	Sequence Number	Length + 3 + n	Continuous Flag	Status	LUCA Error	LUCA Error Text	Data

- **Continuous Flag** — This flag indicates if an additional response, concerning the same command, will follow, in which case it will be set to 1, otherwise it will be set to 0. The Continuous Flag is usually utilized to signal that the command is still being processed;
- **Status** — Depending on the response message type, status values can be returned. There's a list specifying all the Status Codes that can be sent, which contains over 50 codes. Each code represents a scenario. The most common Status Code is the "CMR_OK" which indicates that the message transaction was successful and no error was detected. As for the rest of the codes, most represent a scenario where an error was detected (e.g. "CMR_TIMEOUT" is used in case of a timeout, "CMR_LOST_MESSAGE" is used when the Sequence Number field of the command does not match the response's Sequence Number).
- **LUCA Error and LUCA Error Text** — Both the fields are only valid if the Status field is set to "CMR_LUCA_ERROR", otherwise they are set to 0 and an empty string respectively. These two fields are used to get more detailed information about an eventual error and are more focused on the lower layer level of the system. The LUCA Error also has a list of codes that can be used similar to the Status field.

Further in the document, some commands will be explained, more specifically, the commands needed to achieve a mocked connection and log in. Completing a version that simulates these two processes will prove that the tool is working properly with the RPS and the RAMV protocol.

2.3 Simulation Tools

Based on the knowledge acquired from the literature review we concluded that to help test the software that communicates with a physical fire detection system, the most profitable solution would be to simulate that physical system. We carried out a tool research in order to understand which available tools currently exist to help develop a software that mocks an external dependency like the physical part of a fire detection system. The software developed should respond to the commands sent by the RPS which will allow the tests to be run without the actual physical system present.

The Simulator must replace the physical components of the system so it should respond to the commands sent by the RPS, and for that to be achieved, based on the working document delivered by Bosch [17], we extracted the following requirements to help define the architecture and the choice of the solution. These requirements, alongside the non-functional aspects defined further, assisted in the definition of the tool that will be used to develop the project:

- **Should be configurable through a REST API.** The objective is to develop a software, which will be called Simulator for the rest of the document, that will have a RESTful API reachable by the tester or developer in order to create simulations of a physical system. The simulations created through the API will be called Mocked Servers for the rest of the document. To do this, the user will send an HTTP Request containing the data needed to

create a mocked server that the RPS will send commands to instead of sending the command to a real physical system. The API will also have endpoints to update and delete the mocked servers created by the user.

- **Mocked servers should communicate through Transmission Control Protocol/Internet Protocol (TCP/IP).** The RPS communicates with the physical system through a TCP/IP connection, therefore the mocked instances must also receive and send messages through the same protocol to replicate the behaviour of the real system. Having a TCP/IP communication also means that, when creating a mock server, the user must state a port number in which the mock server will connect and wait for commands to be received.

- **Definition of test scenarios.** In order for the system to work properly and the testing of the RPS to go smoothly, the user should be able to define test scenarios for the mocked server. Test scenarios are composed of expectations and responses. The expectations will be compared to the commands received, if the command matches any of the expectations then the response attached to that expectation will be sent back to the RPS, simulating the behaviour of the normal system.

- **Creation of different sessions (or mocked servers).** The software will allow the creation of a multitude of mocked servers through the API, each mock server will be assigned to a different port number. To use a different mock server, the user has to be able to change the port in which the communication will be established in the RPS in order for the ports to match on both sides. This is an important requirement, so that the tester, or different testers, can run multiple test scenarios concurrently in the same instance.

In addition to the functional requirements there are other aspects that were considered while conducting the research to assist the selection of a final tool:

- **Client library.** The idea is to build a tool, so the research focus on solutions that provide a client library to support the development of the tool. A solution with a client library allows the developers to build their tool with function and methods from the solution, facilitating the work that needs to be done to achieve certain objectives.

- **.NET Support.** More specifically, the tool will be developed in .NET therefore the search focus on libraries that support that language.

- **Support and Documentation.** Another important aspect of the solutions explored is the quality of documentation, if it is regularly updated and the price involved in using each particular solution.

Once the criteria on which the available solutions that were going to be evaluated upon were defined, it was compared to several different tools. The starting collection of tools selected to research was based on a list of comparison of API simulation tools [9]. With the information made available in the list, most of the solutions present were removed from the research considering they

missed some key aspects. Following this, a more detailed research was performed individually to each tool that seemed to satisfy part the requirements.

2.3.1 Individual Description

In this section, each evaluated tool is explained, the data was retrieved from their documentation and small interactions with the tools.

2.3.1.1 Beeceptor

This was the first tool researched, it is also the simplest one. Beeceptor is a simple API simulation tool, accessible through any browser, where the user is able to define endpoints that, when accessed by HTTP requests, will return a predefined response. This tool is used to test systems that connect to an external API in order to obtain data, but isolates the software being tested, since the requests return values are predictable and chance that the connection with the external API fails for external motifs not related to the software itself. The analysis to Beeceptor was made in an initial phase but, due to its simplicity, this tool was quickly removed from the possible solutions, the TCP/IP communication protocol was not a part of the tool and it does not have a client library [10].

2.3.1.2 MockServer

When searching for API Simulation, MockServer is one of the solutions that appear the most. The tool's website has detailed and extensive documentation, a live community and a lot of different clients. MockServer provides an API to test the system and how the tool works, and also provides client libraries for programmatically developing a software using the functions built for the API referred before. Unfortunately, the libraries made available by MockServer only cover Java and JavaScript and one of the requirements of the project is to develop in .NET which means writing the software in C#. Another dissimilarity between MockServer and the requirements defined is the lack of TCP/IP communication. For these two reasons MockServer was excluded from the possible solutions [11]. The example of one of the usages of the Mockserver tool is represented in the Fig. 2.3, by comparing this example with the desired architecture discussed in the previous section in the Fig. 2.5, we can conclude that this example is closer to the typical architecture defined in the Fig. 2.6 than to the desired architecture, which is another reason to make us remove MockServer from the option pool. In this diagram it can be understood that the HTTP requests are received in step 1, matched with the defined expectation of the API in step 2, and the responses are sent back to the host of the request based on the expectations.

2.3.1.3 Wiresham

Wiresham was an option that drew attention when searching for tools due to the fact that is one of the few options in the list [9] that uses TCP. Unfortunately, after reading the short documentation available in the project's page, it was easy to understand that the tool's features do not match the

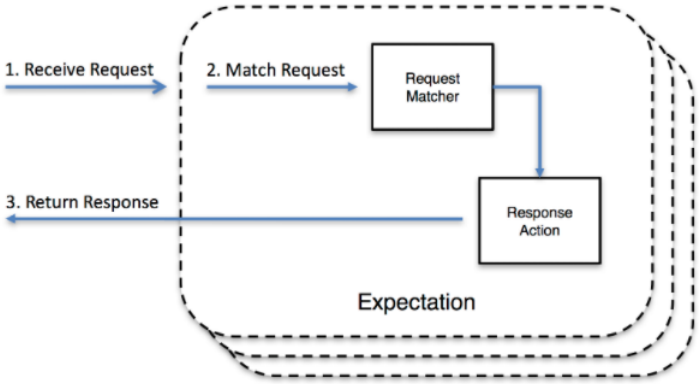


Figure 2.3: Mockserver Architecture Diagram [11]

requirements of the project. The goal is to capture network traffic sent through TCP/IP with other software (e.g. Wireshark) and use Wiresham to replay the traffic, mocking an external component that sends data and helping test a software that needs to receive data through TCP/IP in a desired port. It is an interesting tool especially because Wireshark, being one of the most used tools for traffic detection and capture, does not have this feature, but fails to support the development of our project since it does not allow the user to define expectations and responses, just replays captured network traffic [12].

2.3.1.4 WireMock.Net

WireMock.Net is a library for stubbing and mocking web HTTP responses using request matching and response templating. It's based on the original WireMock but made for .NET framework. Wiremock is another solution that seems to be used a lot when developing applications that use HTTP communications. By creating mocked endpoints for testing requests it accelerates the process of testing the software, removing the need for a real API working at all times and minimizing the chance of errors occurring from external boundaries to the system under test. Once again, the tool does not fulfill the requirements of the project discussed in this document since it does not support TCP/IP communication and due to the fact that mocking responses to HTTP requests is not the objective of the project, the goal is to configure, through HTTP requests, mocked responses to TCP/IP commands sent by the RPS [13].

2.3.1.5 Parasoft Virtualize

Parasoft Virtualize is a tool that seems to meet most of the requirements, with one of the biggest problems being the fact that it is a paid software and it's proven to be difficult to experiment with the tool [14]. Despite the attempts to download and install the Community Version provided, the links sent to download the tool were never functional. Additionally, Parasoft Virtualize is a tool, without an available client library, so it can't be used to develop the desired software and with

no possibility of testing the features of the tool it becomes impossible to do a reliable evaluation. Consequently the tool was removed from the pool of possible solutions to the problem.

2.3.1.6 Mountebank

From the requirements defined from the working document [17], right from the start Mountebank seemed to meet every aspect desired [15]. Unlike any other tool researched, the Mocked Servers to mock the TCP/IP communications (called imposter in the Mountebank documentation) are defined through HTTP requests, as well as the sets of expectations and responses (called stubs in the Mountebank documentation). Mountebank allows for the creation of several mock servers, each associated to a port, and the mock servers have the option of communicating through both TCP/IP and HTTP. The original Mountebank does not support .NET but there is a client library already implemented for that purpose called MbDotNet [16]. The documentation is extensive and complete with several examples provided which helps when developing with this solution. Because of all the similarities between Mountebank and the requirements, this solution turned out to be the preferable one from the selected pool of solutions to research, being selected to assist in the development of the project. The mountebank documentation provides a diagram with the workflow of the tool, which can be seen in Fig. 2.4, allowing for a better perception of how similar the architecture of the mountebank is to the desired architecture presented in the Fig. 2.5. In the diagram we can perceive that a typical test scenario, called test in the diagram, is to first send an HTTP request to the mountebank API with configuration details, followed by the creation of the imposter done inside the mountebank service using the configurations received in step 1. Then, the app, that is the system under test, is activated by the test in step 3, and through this activation the app will communicate with the created imposter, in step 2, to simulate the TCP communication between the app and a real component.

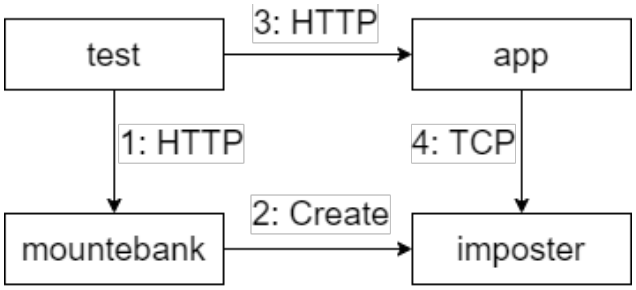


Figure 2.4: Mountebank Architecture Diagram [15]

2.3.2 Architectural Distinction

Since the beginning of the tool research, it was clear that most of the solutions available had a major flaw related to their architecture. To better understand the differences between the typical architecture of the tools and the desired architecture for this project, two images explaining the workflows of both architectures were constructed.

In Fig. 2.5 we have the desired architecture workflow, showing an overview of the steps that are meant to be made in the project being explained in this document:

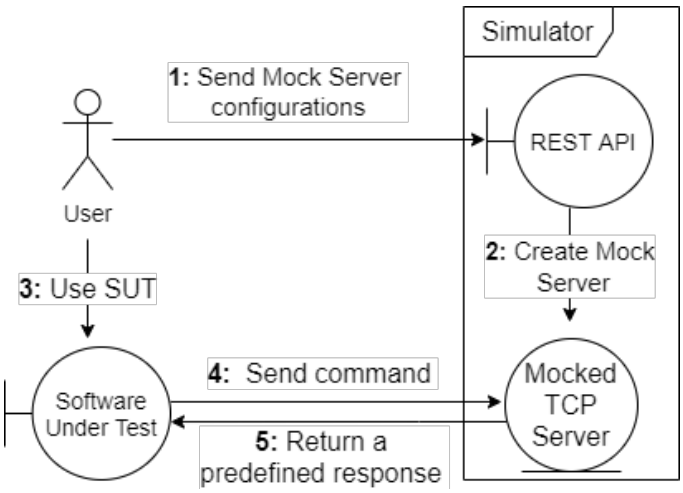


Figure 2.5: Desired Architecture Workflow

- Step 1: the user communicates with the Simulator's API to create a mock server, sending with the HTTP request the details needed to configure it;
- Step 2: Inside the simulator, once the configurations are received by the REST API, the functions to create a new Mock Server will be executed with these configurations, the created mock server will wait for TCP messages and the expectations and responses will be defined using the same configurations provided by the User;
- Step 3: The User now interacts with the Software Under Test (SUT), in our case the Communication Software, to start the exchange of messages between the Communication Software and the Mocked TCP Server;
- Step 4: The Communication Software sends one or several commands, similar to what would happen if it was connected to the real physical system, but in the testing scenario the commands will be sent to the port defined in the mocked server, so that the simulation receives the commands;
- Step 5: After comparing the received command with the expectations defined by the User, and assuming the expectations match with the command, the Mocked TCP Server responds with the responses that were configured for these expectations.

As referred previously, the architecture explained in Fig. 2.5 is the desired architecture for the current project. With the desired architecture explained, and to understand the dissimilarities between the two, it is now important to describe the typical architecture used in most of the tools researched, as shown in Fig. 2.6, being this the reason why most of them were discarded right away.

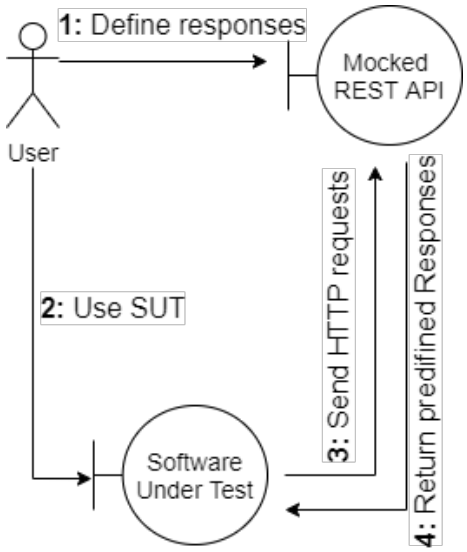


Figure 2.6: Typical Architecture Workflow

- Step 1: The user defines the responses for a mocked REST API, through an interface or programmatically. This will create a simulation API ready to receive HTTP requests and respond automatically to the requests;
- Step 2: The User interacts with Software Under Test (SUT), in order to make it communicate with the Mocked REST API through HTTP requests;
- Step 3: The SUT sends the HTTP requests that would normally send to a real API but this time to the simulated API;
- Step 4: The Mocked REST API receives the requests from the SUT and sends back a response based on the configurations defined by the user in the Step 1 in order to simulate the workflow that would normally take place if the SUT was communicating with the real software.

As we can see through the workflows, illustrated in Fig. 2.5 and Fig. 2.6, the typical architecture and the desired architecture have plenty of differences, the former being used to simulate a more typical communication between a system and a RESTful API, made through HTTP requests. On the contrary, the desired architecture, does not simulate endpoints from an API, the simulators has an API as one of its components that is used to create mock servers that will, in its turn, simulate the communication between two systems made through TCP/IP.

2.3.3 Evaluation Matrix

Succeeding the tool selection and once the documentation of the different solutions was analyzed, an evaluation matrix was created with the tools and the different aspects found relevant based on the requirements as can be seen in Table 2.3.

Table 2.3: Evaluation Matrix

Category	**Criteria**	**Researched Tools**					
		Mountebank	**Wiremock.Net**	**Wiresham**	**Beeceptor**	***Mockserver***	**Parasoft Virtualize**
Support	Active	Yes	Yes	No	Yes	Yes	Yes
	FOSS[a]	Yes	Yes	Yes	No	Yes	No
	Documentation[b]	*****	****	*	***	*****	*****
Architecture	Protocols	HTTP, TCP/IP	HTTP	TCP/IP	HTTP	HTTP	HTTP, TCP/IP
	Container Deployment	Docker	Docker	None	None	Docker	Docker
	Client Library	Yes	Yes	Yes	No	Yes	No
	Graphical User Interface	Yes	No	No	Yes	Yes	Yes
	Language Support	C#, Java and others	C#	None	None	Java, NodeJS	None
Functionality	Expectations and Responses	Yes	Yes	No	Yes	Yes	Yes

[a] Free and Open-Source Software

At the time, because of all the features that matched the requirements of the project, Mountebank was the selected tool to help develop the framework to mock test the RPS.

2.3.4 Experimental Evaluation

Once the simulation tool was selected, as referred in the previous section, a process of experimenting was performed. Mountebank provides an already implemented API to try out the features of the tool, allowing the user to create mock servers (called imposters in the documentation of the solution) with specific sets of expectations and responses (called stubs in the documentation of the solution).

To start the experimentation, the tool was installed and deployed locally to be accessed as a service through the API provided by it, allowing the user to utilize the endpoints developed by the authors of Mountebank. Next, after studying the documentation of the tool, a set of HTTP requests were designed to test the features of the service and tested against the API. Next, there will be examples of the requests used and accompanied by an explanation for each one.

Creating an imposter The first request sent to the service creates an imposter, which is a software component similar to what the mock server is envisioned to be. Assuming the service is running locally the address used to access it will be "localhost", and by sending a POST request with a JavaScript Object Notation (JSON) attached to it, as shown in Listing. 2.2, to the endpoint *"http://localhost:{port}/imposters"*, changing the port for the port used by the mountebank service, an imposter will be created. The JSON message should have the specific configurations to create the imposter and after that it will be waiting for data to be received in the port specified in the request.

In Listing. 2.2, the example demonstrates a JSON containing the configurations normally used to create an imposter, this JSON will be attached to the POST request sent to the service:

- In line 2 the port of the imposter is defined. This is the port to which the RPS will connect to, instead of connecting to the real physical system and the port that will be used to send TCP packets;
- The protocol is defined in line 3, in this case the imposter is using TCP to match the requirements of the project but it could also use HTTP, HTTPS and Simple Mail Transfer Protocol (SMTP);
- In line 4 the "mode" flag configures if the imposter receives data as text or binary. This might be an important feature considering that the physical components often send binary messages and the RPS might send binary commands as well, helping the transmission of commands in case the communication between RPS and physical components use the binary mode;
- Line 5 defined the imposter's name. This is an optional field used to facilitate the user's interaction when working with the tool;
- In line 6 the stubs field is written, inside this parameter, in line 7, is where the expectations and responses will be defined, the "stubs" parameter can have one or more stubs in form of JSON. The JSON composition to create a stub will be explained in detail next.

```
{
  "port": 5555,
  "protocol": "tcp",
  "mode": "text",
  "name": "ImposterName",
  "stubs": [
    (...)
  ]
}
```

Listing 2.2: JSON to create an imposter

In Listing. 2.3, that can be found below, an example on how to configure a stub with expectations (called predicates in the mountebank tool) and responses is shown and described in detail:

- The first line defines that we are configuring the stubs for an imposter, it is important to know that several stubs can be attached to a single imposter simultaneously as explained before, in case the test scenario needs several different stubs, inside this “stubs” JSON configuration there must be represented different JSON Objects each one with predicates and responses, like it can be observed in the example;
- From line 2 to 10 the first stub is defined;
- From line 3 to 5 the response that should be return, in case of this stub is activated, is defined. In this example, if the predicates match with the received message, this stub will return a simple string with the text "Response Ex." to the address where the original message was sent from;
- From line 6 to 9 the predicates are defined. In this case, two predicates are being configured, Mountebank provides various matching alternatives. For the example the “contains” and “startsWith” were used, they search for the existence of the string “test” in the message received and examine if the message received starts with the string "example", respectively. In the event that both conditions are verified, the response will be triggered and the original sender of the message will receive the response defined in line 4.
- From line 11 to 18, the second stub is defined, similar to the first one but with a few changes;
- From line 12 to 14 the response for the second stub is define, this stub return “Response Ex. 2” if it gets triggered by a received message;
- From line 15 to 17 the predicates for the second stub are defined. In this case, only a single predicate was defined, it will check for a message that contains the string “test” in it. The mountebank tool respects the order of the stubs defined, therefore, if a message matches

the first stub it will trigger a response and stop the process of matching, meaning that if a message that would match both stubs is sent, only the first one to be defined is activated.

```
1  "stubs": [
2    {
3      "responses": [
4        {"is": {"data":"Response Ex."}}
5      ],
6      "predicates": [
7        {"contains": {"data":"test"}},
8        {"startsWith": {"data":"example"}}
9      ]
10   },
11   {
12     "responses": [
13       {"is":{"data":"Response Ex. 2"}}
14     ],
15     "predicates": [
16        {"contains": {"data": "test"}}
17     ]
18   }
19 ]
```

Listing 2.3: Stub JSON example

After the request is executed the imposter will be created and the user will now be able to interact with it through the endpoint *"http://localhost:{port}/imposters/{imposterPort}"*, where the "imposterPort" is the port defined in the Listing. 2.2 in line 2; in this specific example it would be the port "5555".

Updating an imposter The service provides an endpoint to update the stubs of an existing imposter, instead of a POST request, a PUT request needs to be sent to the service. When the imposter is created, each stub attached to it has an ID representing it. The endpoint for updating a stub is *"http://localhost:{port}/imposters/{imposterPort}/stubs/{stubID}"*, where "stubID" is the ID assigned to the stub that needs to be changed. The data structure that should be attached to the request is similar to the example in the Listing. 2.3.

Adding a stub to an imposter When configuring an imposter, if the required stubs are not added in the initial request that creates the imposter, the stubs can be added afterwards through the endpoint *"http://localhost:{port}/imposters/{imposterPort}/stubs"*. Once again the data attached to the request should be similar to the example shown in the Listing. 2.3.

Deleting an imposter Another endpoint provided by the Mountebank API allows the user to delete the imposter by sending an HTTP DELETE request to the endpoint *"http://localhost:{port}/imposters/{imposterPort}"*, where, once more, the "imposterPort" is the port selected for the imposter on creation.

Inspecting created imposters Having a way of inspecting the data available in the service is a typical feature of every API, and mountebank is no exception. By sending an HTTP GET request to the endpoint *"http://localhost:{port}/imposters/{imposterPort}"* the service will return all the configurations of the imposter including the stubs. If the user does not specify any imposter port, a list with all the existing imposters will be returned. This list does not contain the configurations of each imposter, but indicates the imposters ports if the user wants to inspect the details of a specific imposter.

Once the API features were learned and clarified, its capabilities were tested to assure that the configured imposter worked as it should. After creating an imposter as specified in the Listing. 2.2, with a stub equal to the example in Listing. 2.3, the imposter needed to be tested to verify that the defined responses were returned when the predicates were matched. With the assist of a well known tool called Netcat, several TCP/IP packages, containing either a string that matched or a string that did not match the predicates, were sent to the imposter's address to test that the response was returned or not, respectively. All the tests were correct.

2.3.5 Gaps

Although the Mountebank tool complies with most of the requirements defined for the project, there are some aspects that are not sufficient for the desired development:

- The different matching alternatives for the expectations that can be defined in the API are not be enough to cover the complexity of the commands sent by the RPS. The commands sent by the RPS, have a specific pattern, described before, and the simple available matching options of the Mountebank do not allow for a complex enough comparison and management of the data received in the commands;
- The same thing happens with the responses, although the basis of the Mountebank tool in terms of communication and architecture is similar to the one envisioned for this project, the responses must have a more suitable way of preparing the content, due to fact that the responses follow a complex pattern described in the Table 2.2;
- The behaviours that can be configured for simulating certain error scenarios only work with HTTP imposters and this behaviour must be perfected when developing the tool to help test failure scenarios, like timeouts, to be executed while testing the Communication Software;
- Configuring imposters and stubs in the Mountebank service is a laborious activity, although the configurations can be reused through different testing sessions. One of the goals of the project is to reduce the time invested in the process of testing the system, therefore the

writing of all the specific configurations in the form of a JSON message, like shown in the Listing examples previously, must be improved to a simpler, more efficient process.

Considering these gaps between the selected tool and final product idealized, it was clear that the tool was not enough to fulfill all the requirements and that a different solution would have to be implemented. Specially in terms of definition of expectations and responses, as referred in the two first items of the list above, the goal should be to create a tool that is ready for scenarios where the communication is done through commands that comply with the patterns defined in Table 2.1 and Table 2.2. The goal was to use the Mountebank C# client library to help implement the basis of the new tool, taking advantage of the already implemented features of the Mountebank in terms of communication and creation of components ready to receive and send TCP packets through an API. However, the client library only provided methods to communicate with the base Mountebank tool and, therefore, the same problems were present. The user must be able to define new expectations and behaviours through the code and create new features that facilitate the definition of mock servers and their reuse throughout different testing sessions. Therefore, a new more sophisticated tool needs to be implemented.

2.4 Summary

With this chapter, through the literature review and by gathering information to build a state of the art of the subjects discussed previously, the intention of using mock testing in our project to help facilitate the work involved in testing the software was reinforced. Also, having researched the challenges that are usually faced during the implementations of a simulation software, to replace the need for real components of a system, will help prevent several scenarios that would normally delay the process. The definition of the requirements based on the working document was definitely one of the most important aspects of the state of the art. This is the information in which the production will be based upon for the rest of the project, and having a set of accurate requirements ensured that the tool research was made correctly, focusing on the most important aspects and that the future work will be well guided throughout the next months. Also, through the tool research, it was concluded that, although there are several different tools and frameworks that can be used when working with mock testing, there is a dearth of solutions for certain languages and scenarios that need to be expanded. With the similarities referred in the Experimental Evaluation section, it was simple to understand why Mountebank was the selected tool among all the other options, the fact that this is the only tool that uses an architecture similar to the one defined in the Architectural Distinction section as the desired architecture, makes the decision of Mountebank almost obvious. And finally, considering the contrasts between the desired product for this project and the aspects explored in the Mountebank API, as explain in the Gaps section, the work planned for the months after the writing of the state of the art was to use the library provided by the Mountebank team to implement a framework that would satisfy all the requirements defined for the project that the current state of the Mountebank tool can not satisfy. Later in the project we discovered the challenge of using the Mountebank tool and altering the definition of expectations and responses, and

decided to create a new tool, maintaining the architecture but making it custom and more specific for the problem presented here.

Chapter 3

Solution Design and Implementation

3.1 Technologies and Architecture

As stated in the working document shared by Bosch [17], the project should be developed in C#, more specifically in .NET Core to allow the solution to smoothly execute in both Windows and Linux systems. Therefore, the Framework and programmatic language were defined since the beginning and other options were not discussed or analyzed. I used Microsoft Visual Studio Community 2019 to implement the project as my selected integrated development environment (IDE).

In the weeks following the conclusion of the state of the art, the meetings with the team responsible for the project began, and the focus was now defining the best architecture for the solution, based on the defined requirements.

Although there were various alterations to the requirements and objectives of the project throughout the months of implementation, the architecture suffered minimal modifications from the initial draft.

Figure 3.1 shows the desired architecture on which the implementation of the solution would be based upon. In this diagram we can see the components that compose the system and the communication that is present between these components.

The next chapter will be dedicated to describing the way each component works, its usefulness to the objective of the project and the communication between components will be explained in detail as well.

Once the literature review was concluded, the practical analysis of the different available solutions, and the initial draft of the architecture were defined in the meetings with the team responsible for the project, the implementation of the solution started, according to what was planned before.

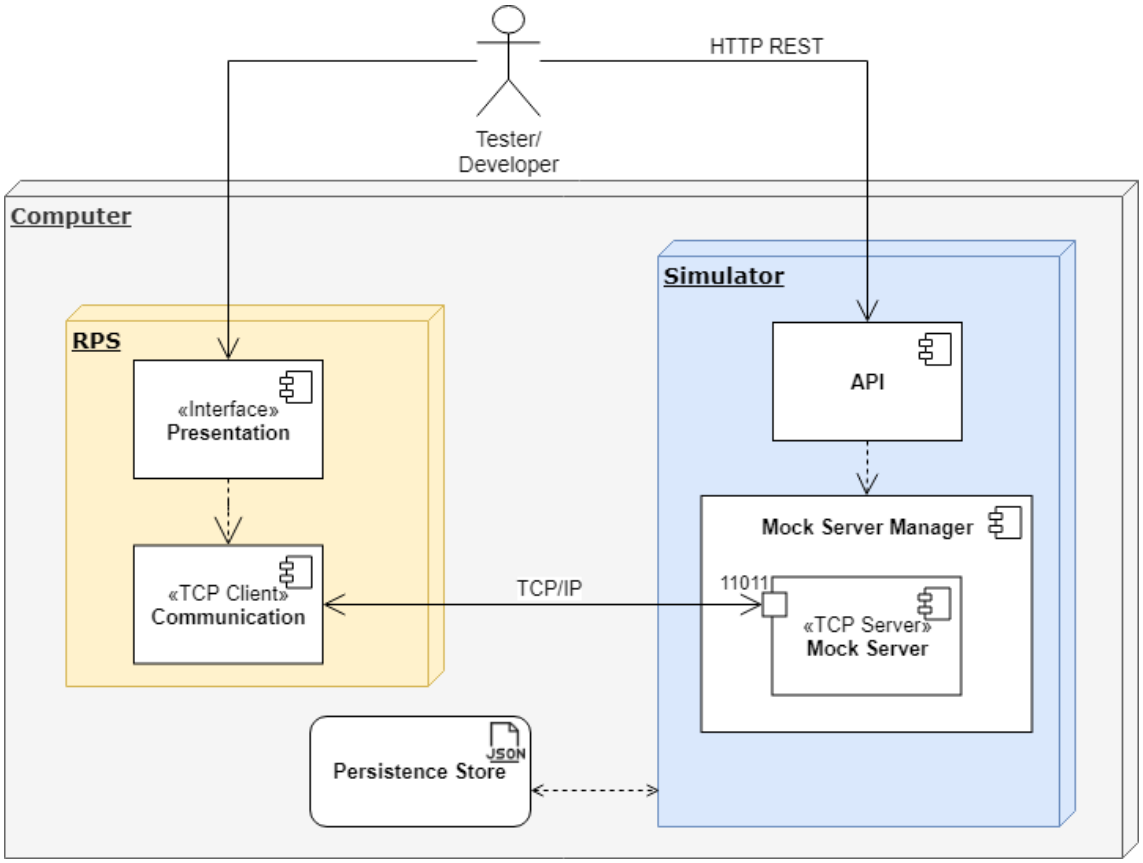


Figure 3.1: Initial Architecture

3.2 Working Example

Table 3.1: RPS Fictitious Command Pattern

Command Pattern				
Content	**Type**	**Unique Id**	**Length**	**Data**
Size	1 byte	1 byte	2 bytes	<*Length*> bytes

Table 3.2: RPS Fictitious Response Pattern

Command Pattern						
Content	**Type**	**Unique Id**	**Length**	**Has Error**	**Error Code**	**Data**
Size	1 byte	1 byte	2 bytes	1 byte	1 byte	<*Length*> bytes

The first step of the implementation was to create a working example based on the protocol described in the working document, this example would work as a proof of concept (POC) and serve as a base for the next steps of the implementation. The working example should follow the architecture shown before but the protocol specified was more simplistic and attainable than the real protocol that is used by the actual RPS to communicate.

The working example commands use a different protocol, similar to the real RAMV protocol but more simplistic. The structure of this fictitious commands can be seen in Table 3.1 and Table 3.2. In this initial phase, the mock server should be able to receive and respond to the following mock commands:

- **Connect** — The first command to create a connection between the two components;
- **Login** — The login command to allow for authentication;
- **Get Custom Values** — Command to retrieve data from the panel.
- **Save Custom Values** — Command to save data in the panel.
- **Set System Date and Time** — Change the date and time of the panel.

These commands simulated the presence of the RPS, since they are similar to the commands from the actual protocol, they helped develop the first version of the testing framework, which provided an API and created mock servers. This initial version was later updated to work with the real

RAMV protocol. Also, since the protocol differs from the real RAMV protocol, the actual RPS could not be used to send the commands to this first draft of the framework and could not prove that the mocked communication was functional. Therefore, and to achieve the initial POC, another solution had to be created. This new solution and the initial draft of the framework used the same command classes, so they could initialize and serialize them, and communicate through TCP/IP with these commands. The successful communication between the two components proved that the workflow, similar to the workflow desired in the communication between the RPS and the tool created, was fully functional.

3.3 Component Interaction

Before explaining each component in the system, I will show a more generic view of the interaction between components. In the example shown as a Sequence Diagram in Figure 3.2, we are testing the connection and login of the RPS with the Simulator.

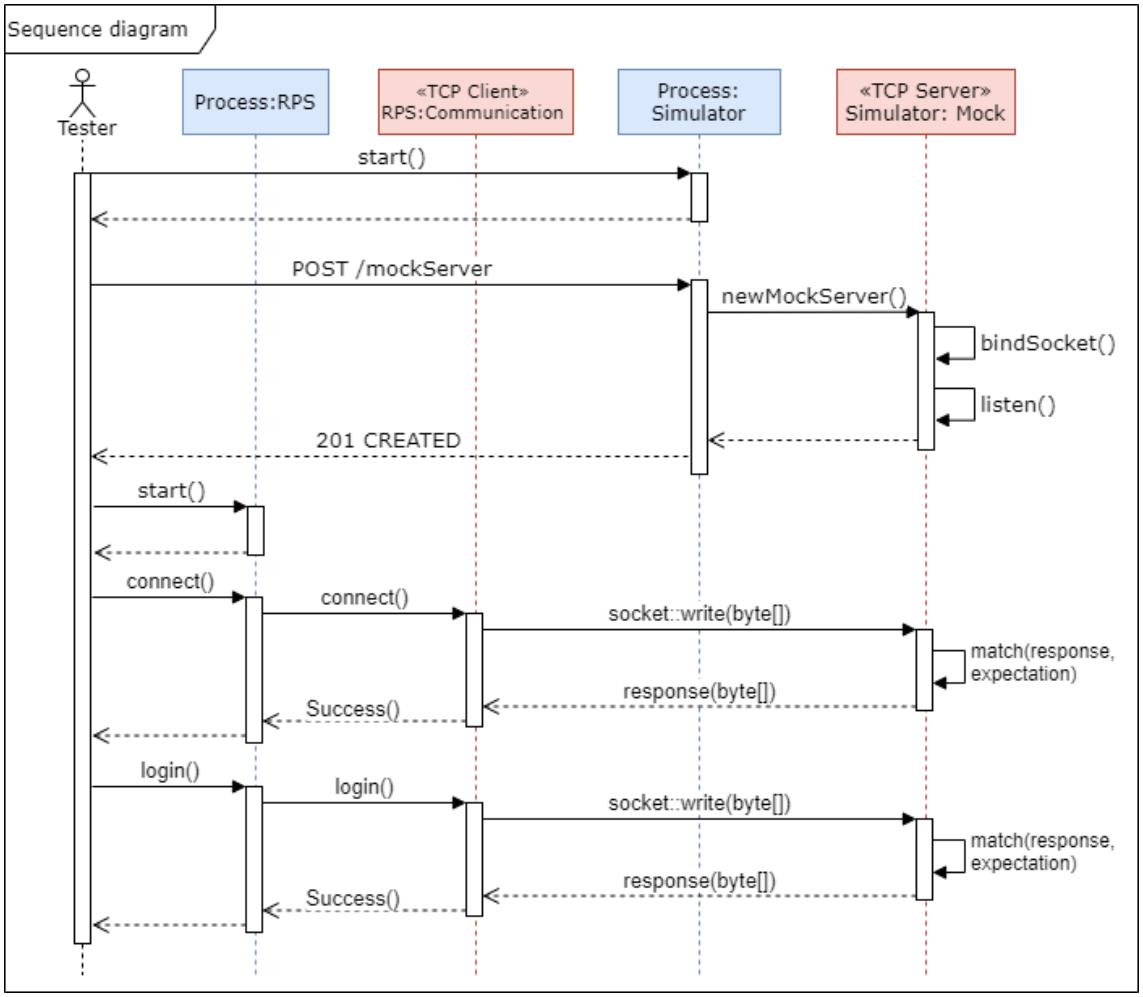


Figure 3.2: Sequence Diagram of a Connection and Login between the RPS and Simulator

The tester should begin by starting the Simulator, after that, the tester sends the configurations to create a mock server through an HTTP POST request. The mock server is created, bound to a specific port and waits for a connection. Inside the mock server, there should be configurations of expectations and responses, sent in the POST request to create the mock server as a JSON message. These configurations will be used to reply to the RPS commands. The tester should now start the configuration tool, called RPS in Figure 3.2, and use the RPS to send the commands it would normally send if it were communicating with a panel. The commands will be sent to the mock server created before instead of the panel, and the mock server is responsible for mimicking the behaviour of the panel so that the RPS act as if it was communicating with an actual panel. As shown in Figure 3.2, the tester sends a "connect" command, received by the mock server, matched with the expectations, the respective response is created and returned to the RPS, which will accept the Success response and assume it is connected to a panel.

This process explains the process of the system broadly. Next, I will describe each component of the system in more detail.

3.4 Simulator

The Simulator is the main part of the system being developed, as we can see in Figure 3.1, which shows the architecture of the system, the Simulator is divided into several components and each of these components will be described next.

3.4.1 Application Programming Interface

The first stage of developing the Simulator was to create an Application Programming Interface (API) capable of receiving Hypertext Transfer Protocol (HTTP) Requests containing, in JavaScript Object Notation (JSON) format, the data required for the configuration of a mock server.

3.4.1.1 Endpoints

The API is prepared to receive requests in various endpoints, to create, update and delete mock servers and to store and retrieve configurations of a mock server.

The API is divided into two main blocks, the first one is responsible for handling the data related to the mock servers. Every endpoint has an URL beginning with "/api/MockServers/", and the second block is responsible for handling the data related to the storage of sessions in the server, where every endpoint URL starts with "/api/MockServers/storage/".

The first block has the following endpoints for handling the mock server data:

- **Get Mock Server or all Mock Servers** — An HTTP GET request to retrieve information of the mock servers that are currently running, to receive the information of a single mock server the URL is followed by the port used by the specific Mock Server desired.

- **Create Mock Server** — An HTTP POST request to execute a new mock server, the data needed to configure the mock server will be sent with the request, this data will be explained in detail in the next section. When sending this command, the server will attempt to find a session saved with the same name, if it can not be found, the configuration sent will be saved.
- **Update Mock Server** — An HTTP PUT request to update a mock server that is currently executing. The URL must be followed by the port that the mock server selected to update is using.
- **Delete Mock Server** — An HTTP DELETE request to stop and delete a mock server that is currently executing. The URL must be followed by the port that's being used by the mock server which is going to be deleted.

The second block has the following endpoints for handling the sessions data that is stored in the server:

- **Get session or all sessions** — An HTTP GET request to retrieve the session configurations stored in the server.
- **Delete session** — An HTTP DELETE request to delete a session configuration, the URL must be followed by the name of the session which will be deleted.
- **Add new session** — An HTTP POST request to add a new session configuration, although this will be done automatically when creating a new mock server.
- **Update session** — An HTTP PUT request to update a session configuration, the URL must be followed by the name of the session which will be updated.

Although every endpoint is useful, the most important endpoint, since it was the most used endpoint while developing the Simulator, is the one to create the mock server. Therefore, this will be the one that we are going to focus on in the next section.

3.4.1.2 JSON Structure

```
{
   "port": 11011,
   "name": "Test session 1",
   "protocol": "RAMV",
   "mockServerStubs": [
      {
         "expectation": "InitV2",
         "response":
         {
```

```
10              "Status": "CMR_OK",
11              "SWVersionHigh": 0,
12              "SWVersionLow": 0
13          }
14      },
15      {
16          "expectation": "ConnectV3",
17          "response":
18          {
19              "Status": "CMR_OK",
20              "LUCAErrorCode":"VE_OK",
21              "PNA": 1
22          }
23      }
24  }
```

Listing 3.1: JSON to create a Mock Server

Listing 3.1 demonstrates an example of the data that is contained inside the HTTP POST request that is sent when creating a mock server and has the following parameters:

- **Name** — The name of the session will be important to store the configuration for reuse and to help differentiate between the different mock servers that can be running simultaneously.
- **Port** — The port defined in which port the mock server will await for commands, there can only be one mock server for each port.
- **Protocol** — This field is used to define what is the protocol required to configure the mock server and the respective expectations. For the scope of this thesis, the only protocol used is the RAMV protocol, but it is separated into an external library, envisioning the possibility of expanding the project and including other protocols.
- **Mock Server Stubs** — The Mock Server stubs contain the expectations and responses defined for the mock server. Each stub has an expectation based on the protocol and the response related to that expectation. For now, the expectation must contain the name of a command that is mapped in the server, otherwise it will not be accepted. The idea of creating a more generic and dynamic way of creating expectations is discussed in the Further Work section. In the response, the different fields need to be specified in the JSON configuration to allow for the creation of the response that is going to be sent to the RPS after it sends the command that is defined in the expectation. The process of stub creation will be explained in detail afterwards.

3.4.1.3 Mock Server Creation

When the Simulator API receives the request to create a mock server containing the configuration data, the first step is to check the availability of the port in the system through the C# library "System.Net.NetworkInformation". After that, the system checks if there is a Dynamic-Link Library (DLL) available for the protocol selected, which will be responsible for the configuration of the stubs and for handling the external communication with the mock server. Currently, the only DLL available is for the RAMV protocol but with the possibility of developing libraries capable of supporting other protocols since, as explained previously, the idea is to create a project to allow an easier way of mock testing a system, which is attained with the Simulator, but the creation of expectations and responses is not completely dynamic, therefore the system still needs the presence of a library that handles the mapping of the commands and responses as well as the communication.

Finally, the server uses the DLL selected previously to create the stubs. This process is done in the external library, in the example, the protocol RAMV is being used, and the library was developed in a way that makes it ready to receive just the name of the command that is to be expected, however, in the response, several fields need to be defined, depending on the command and the data that needs to be sent back to the RPS. The different responses are mapped in the DLL, therefore, if the fields defined in the JSON of the response do not match the actual fields of the response, an error will be returned and the mock server will not be created. As an example, we can look at Listing 3.1 and the last stub creates an expectation for the "ConnectV3", which means the mock server will expect to receive a command of the type Connect Version 3. After receiving the command, the mock server will create an instance of the respective response to this command type, the structure for both the command and response are shown previously. As explained before, the response to the specified command has a Physical Node Address (PNA) field, that is defined in the JSON to allow the mock server to create a response that mimics the response of actual interaction.

3.4.1.4 JSON Reuse

This was one of the requirements defined in the initial proposal of the project, the testing sessions should be easily reusable. As referred before, to achieve this requirement, the Simulator saves, in a .json file, the different configurations sent to it, to reuse it every time the tests for that specific scenario need to be executed.

3.4.2 Mock Server

Once the starting API was developed and working, I focused on implementing the mock server. Each mock server is executed inside a class called "MockServerRunner" that stores the actual instance of the mock server, handles the TCP communication, and calls the methods from the external library that are responsible for handling the deserialization of the commands received and

the creation and serialization of responses that will be sent back to the RPS. An instance of a mock server runner should be able to:

- Execute in its own Thread;
- Store the configurations of the stubs defined before;
- Start a TCP server to receive commands;
- Compare the received messages with the expectations defined in the stubs of the mock server;
- Create the necessary replies based on the configuration of the stubs of the mock server;
- Send the responses back through the TCP server;

Each of the points listed before will be explained next.

3.4.2.1 Execute as a Thread

According to the requirements, the mock server should be scalable, which means it can have multiple instances executing concurrently, and, therefore, should be implemented in a way where the Simulator should not be blocked by the communication process of each mock server. On that account, a new thread, called Task in C#, responsible for handling the execution of the mock server should be created and started when a new mock server instance needs to be created.

3.4.2.2 Store the Mock Server Configurations

When an instance of the mock server is created, the configurations are stored inside the instance, as the data contains the port that should be used for the TCP communication as well as the different stubs that the mock server will use to compare with the received commands and reply with the necessary responses.

3.4.2.3 TCP Communication Module

To handle the communication between the mock server and the RPS, and making use of the C# library “System.Net.Sockets”, a TCP server is started inside the Task. The server will wait for a connection at the port specified in the initial configuration and, once the connection is started, the communication with the client, in this case, the RPS, will begin.

3.4.2.4 Command and Response Handling

The logic of the communication protocol is defined and contained inside the external library that was activated when receiving the initial configuration of the mock server, therefore, whenever the TCP server receives a message, it is redirected to the DLL, which is called to deserialize and

handle the message, and to create the respective response based on the defined stubs. The DLL used to specifically communicate with the RPS will be explained further in Section 4.2.4.

Deserialization

First, the received command needs to be deserialized before it is compared with the stubs. The commands are mapped in the DLL through a group of classes, each class has fields and variables that represent the structure of each command. This structure will be shown next, in the RAMV Protocol section. In some particular cases, the deserialization of the command requires additional steps, such as the Connect Version 3 command that involves decryption of the data, and for these commands, a specific deserialization method is defined in the class. As for the rest of the commands, a specific method for this process is no longer required, therefore, a more broad and dynamic approach was implemented, basing the deserialization process in the content of the class: each variable and the particular type of that variable. This process will be explained further with excerpts of the code. The beginning of the structure of the commands is known and is always similar in every case, as shown before in the Remote Programming Software section. The group of the initial fields of a command that are similar in every command type shall be called the header of the command. It is explained previously that the header always contains the Message Type of the command, therefore, following the deserialization process, the fact that every command respects the same structure, allows the server to discover which command is being received through the Message Type and, subsequently, compare the command to the expectations defined.

Comparison with the expectations

After the message was deserialized and the command type is discovered, the stubs are examined, searching for a stub that contains an expectation that matches the command received. If no stub matches, no response is sent back. Otherwise, if there is a match between the received command and the expectation of a stub in the mock server, the process of creating and serializing the response starts, based on the several fields included in the data received at the creation of the mock server.

Creation and Serialization of Responses

The creation and handling of the response is also part of the communication protocol that is defined in the DLL. The response must follow the structure described before to allow for successful communication between the Simulator and the RPS. Before the response can be sent to the RPS as a reply to the command received, it needs to be serialized, into a format that the RPS is ready to receive and handle. As said before, the different commands and responses are mapped in the DLL, through a group of classes that represent the messages. In some particular cases, the responses have a specific method to handle the serialization, but for most of the cases, the process is done dynamically, through the different variables and variable types of the class that represents the response. Once the response message is serialized, it is sent through the TCP socket that holds the connection with the RPS and the server waits for the next command the RPS sends.

3.4.3 Mock Server Manager

Following the definition and implementation of the mock server, considering that the system should allow the execution of several instances of the mock server working in parallel, it was necessary to create a component responsible for orchestrating the different instances. The Mock Server Manager is responsible for that, as shown in Figure 3.1, the API, after receiving an HTTP request to create a mock server, will handle the creation of the mock server and redirect it to the Mock Server Manager that will handle and store references for the different instances. When an alteration needs to be done to any existing mock server, it is done through the Mock Server Manager. It keeps a list of handlers, and each handler keeps a mock server runner as well as the data necessary to interact with it, either to update the mock server or to terminate it.

3.5 Class Diagram

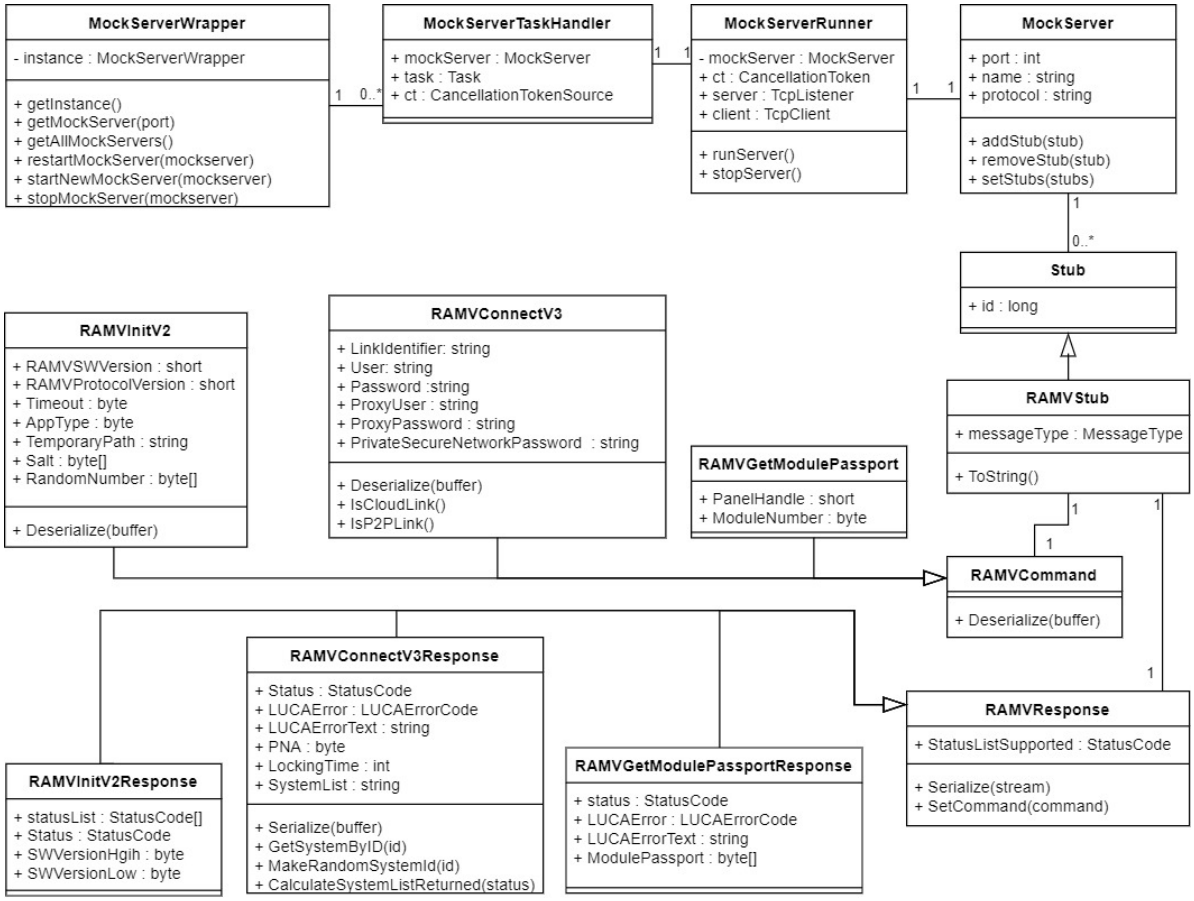


Figure 3.3: Class Diagram

In this section, the Class diagram is shown, in Figure 3.3, to provide the reader with a better understanding how the internal workflow of the system is. The most significant classes responsible for storing and handling the data of the process are described, in detail, in this section.

After this section, we will talk about the Dynamic-Link Libraries (DLL) that have been developed for the system, the Simulator accesses these DLL and uses their classes and methods as extensions. Therefore, I'm going to divide the classes by components, so that the reader can see which class belong to which component:

- **Simulator** - The first three classes on the top of the diagram, Mock Server Wrapper, MockServerTaskHandler and MockServerRunner, are part of the Simulator. In the architecture, shown in the Figure 3.1, the component called "Mock Server Manager" is actually composed by these three classes.
- **Models** - The Models DLL contains the classes that represent the models of the system: the MockServer and the Stub. These classes are needed in both the Simulator and other DLL, hence the reason for its creation.
- **RAMVProtocol** - The RAMVProtocol DLL contains the RAMVStub, RAMVCommand, RAMVResponse and every other class that inherits from the last two. I will not show every command and response derived from the main classes but three examples are shown and will be explained.

The **MockServerWrapper** stores a collection of all the MockServerTaskHandlers and has methods to retrieve, start, restart and stop mock servers. This class is a Singleton, which means there is only one instance of the class and the same instance is activated any time data needs to be accessed;

Each **MockServerTaskHandler** has a thread (called Task in c#), where the MockServerRunner will be executed, and the respective CancellationTokenSource, which is used to stop the thread from the outside, in case the user decides to stop the mock server;

The **MockServerRunner** is where the mock server will be created, this class's method "runServer" is the method that will be running while the communication is active;

The **MockServer** is a simple Model containing the data to execute the server, the port, the name, the protocol and a list with all the stubs;

The **Stub** is the class where the expectations and responses will be stored, but for each different protocol the type of expectation and responses can change, therefore, a class that derives from the original Stub class is created in the RAMV Protocol DLL. That class is called **RAMVStub** and has a value that shows the message type of that stub as well as a command and response;

For the RAMV protocol the expectation is the **RAMVCommand** class and the response is the **RAMVResponse** class. These are abstract classes and for each command type, a different class was implemented, all inheriting from these two initial classes. In this class diagram, at Figure 3.3, the command and response are shown for three types of messages: the InitV2, the ConnectV3 and the GetModulePassport. I will explain what are the differences between them and why I chose this three.

As we can see the class that represents the InitV2 command implements the "Deserialize" method, overriding the method from the class that it derives from. But the response for the same method does not override the "Serialize" method. This means that, when receiving the command, the content is not composed of simple data that remains the same in every communication process, consequently, a more sophisticated method needs to be done. In this case, this happens because the command receive has a Salt value and a random number, which allows for more security in the process but also raises its complexity. On the other side, the response is a simple data transaction with no complex behaviour, therefore, the inherited method to serialize the response can be used to dynamically serialize it.

The ConnectV3 command also needs a specific "Deserialize" method because the data is encrypted when received, with the salt from the previous InitV2 command. And the same has to be done to the response sent, therefore, the response also needs a new implementation of the "Serialize" method.

On the other hand, in both the GetModulePassport command and response, the data transaction is simple, with no complex steps attached to it, and because of that, neither of the cases need to override the methods inherited from the main classes and can use the dynamic approach.

This distinction between the different approach uses was important for the reader to understand the reason why the mapping of expectations and responses wasn't made more dynamically.

3.6 Dynamic-Link Libraries

The inclusion of Dynamic-Link Libraries (DLL) was already briefly indicated, but this section shall expand and amplify the detail and rationality behind the DLLs used to enhance the system. A DLL is an external library that contains code and data which can be used by more than one program concurrently. The main reason to use it, according to Microsoft's documentation [20], is that it helps promote modularization of code, code reuse, efficient memory usage, and reduced disk space. So, the operating system and the programs load faster, run faster, and take less disk space on the computer. For the current system, not only the motive described before applies but also the need to organize the code in a way that allows for easier implementation of support for new protocols.

3.6.1 Models

The first piece of the system that was isolated into a library was the Models DLL, it contains the classes of the models that will be needed both in the base project as well as in the external libraries containing the data for the protocols, hence the isolation of these classes. This small library is composed of:

- **MockServer** — This class is the main class modeling the mock server, it has variables for the different fields of a mock server: an integer for the port, two strings for the name and protocol, and the mockServerStubs, a list of the Stub object, which will be explained next.

- **Stub** — This class represents a stub, each mock server has a list of stubs. This is the main class of the stub, it only contains the id field, but every protocol DLL should have a stub class that derives from this one, and the content of the class should include an expectation, that contains a command object, and a response, that contains a response object. The content of both the expectation and the response depends on the commands from that specific protocol, and where the stub creation is also customised for each protocol.

- **Protocol** — The protocol is a class that will be one of the fields of the MockServer class, represented by a string, and, similarly to the Stub, every protocol should have a class that derives from this Protocol class, this class will be used to allow the Simulator to select the right DLL to use when prompted to start a new mock server, checking the respective Protocol class for each protocol's DLL.

3.6.2 RAMVProtocol

The second DLL that needs to be described is the RAMV Protocol DLL. This is a complex DLL that contains the mapping for every command and response, both the serialization and deserialization methods for every message, the listing for response codes, such as the LUCA error codes mentioned before in the Background section, the method for creating the stubs according to the RAMV protocol, as well as any method required for the communication process with software that utilizes this protocol to go smoothly.

Before developing the classes that map the commands from the RAMV protocol, I had to study the structure of each command. Although the protocol specification lists over fifty commands with the respective responses, while developing the system, we focused on being able to connect and login through the RPS without the presence of a real panel. This would be the proof of concept necessary to verify that the structure of the project functions correctly with the RPS and that it can be upgraded to work with every command from the RAMV protocol. To do so, we needed the following list of commands to be responded to, each command will be explained in detail.

3.6.2.1 Init (Version 2.0)

Table 3.3: Init Command Structure

Size	1 bit	7 bits	1 Byte	2 Bytes	2 Bytes	2 Bytes	1 Byte	1 Byte	n Bytes	16 Bytes
Field	Reply Flag	Message Type	Sequence Number	Length	Data					-
Content	0	0x35	x	5 + n	RAMV SW	RAMV Protocol	Timeout	Application Type	Temporary Path	Random Number (Salt)

After the connection through the socket is established, the Init command, represented in Table 3.3, is sent as the first message by the RPS to the Simulator, this command signals the software and protocol version, a timeout used throughout the communication process, and Application Type

that defined the application. For different applications the type differs, in this case, we are using the RPS, therefore this field will be 0x01, and finally, the command also sends a directory path where the temporary files for the communication process will be stored. Through every command of the process, if no reply is received within the specified timeout after sending the command, the connection is finished and the socket is closed. The RAMV protocol regularly receives updates to make it more secure, hence the version 2 of the Init command, several commands have upgraded versions as we will see next. This version adds a field at the end of the command with 16 Bytes that contains a random number that is used to generate a session key, which will be used in the encryption of some following messages, adding security to the process but also some complexity to the mocking process.

The reply to the Init command follows the structure described above in Table 2.2, and the Data field, which, as explained before, differs between response type, is composed of the "TransferTool Software Version", the Simulator is mocking the behaviour of this "TransferTool", thus the reply message must send a fictitious value in this field.

3.6.2.2 Connect (Version 3.0)

Table 3.4: Connect Command Structure

Size	1 bit	7 bits	1 Byte	2 Bytes	n Bytes	m Bytes
Field	Reply Flag	Message Type	Sequence Number	Length	Data	
Content	0	0x36	x	n + m	Link Identifier	Encrypted Credential

If the Init command receives the expected response, the RPS then sends the Connect command, represented in Table 3.4, this command has the "Link Identifier" in the data field that describes all parameters required to create a connection to a panel, including the hardware and protocols used for the link, the syntax of the content is complex and will not be explained. Considering we used an updated version of the Init command we also need to use a version of the Connect command that matches the first, in this case, we are using Version 3 of the command that includes some extra content in the data field, the "Encrypted Credential" contains the credential needed to connect to the RPS.

As for the response to the Connect command, it follows the same structure described above, adding the "PNA" field, which stands for Physical Node Address, and helps validate that we are connecting to the Panel that is expected. Additionally, the response also sends the "Number Of Systems" and "System List" fields, these two fields define the physical systems present.

3.6.2.3 Login (Version 2.0)

If the two previous commands were successful, the Simulator is now connected to the RPS. Following this initial connection, starts the login process, the RPS sends the Login command, which

Table 3.5: Login Command Structure

Size	1 bit	7 bits	1 Byte	2 Bytes	n Bytes
Field	Reply Flag	Message Type	Sequence Number	Length	Data
Content	0	0x37	x	n	Encrypted Login Information

is represented in Table 3.5, this command will create an authenticated connection, which will allow for callback commands that retrieve information from the panel. In the first version of the command, the data for the login is not encrypted but to match the commands we used before, we use version 2 of the Login command, which sends all the login information encrypted.

The Login reply follows the same structure as described before and if the response is successful, it also returns a "Panel Handle", which contains a handle for the connection, this handle will be used from now on, in every command related to the panel.

3.6.2.4 GetModulePassport

Table 3.6: GetModulePassport Command Structure

Size	1 bit	7 bits	1 Byte	2 Bytes	2 Bytes	1 Byte
Field	Reply Flag	Message Type	Sequence Number	Length	Data	
Content	0	0x07	x	3	Panel Handle	Module Number

Automatically after the login command is received and replied successfully, the RPS will send the "GetModulePassport" command, represented in Table 3.6, which will retrieve data about the different modules of the panel, sending the "Panel Handle" defined in the Login command, and the "Module Number" of which the passport, containing the data of that specific module, will be returned in the response. The response follows the same structure defined before and also sends the module passport, an example of this module was made available by the team working at Bosch to facilitate the process of mocking a real panel.

3.6.2.5 GetImageInfo

To conclude the process of login, the RPS sends the "GetImageInfo" command, represented in Table 3.7, this command will solicit the image information of images that are stored in the panel, each image has a complex group of parameters that represent a node from the panel and describes the firmware or the configuration of the node. Once more, the Panel Handle defined in the login command is sent as part of the Data field; the "Request Type" defines the type of images that should be returned, images can be of type "firmware" or "configuration", but this field can also ask for all types of images; each image has an index, the "Image Index" is used to retrieve a

Table 3.7: GetImageInfo Command Structure

Size	1 bit	7 bits	1 Byte	2 Bytes	2 Bytes	1 Byte	1 Byte	n Bytes
Field	Reply Flag	Message Type	Sequence Number	Length	Data			
Content	0	0x1c	x	4 + n	Panel Handle	Request Type	Image Index	Request String

specific image with the requested index, however, if the user intends to retrieve all the images from the panel, the value of this field can be set to "0xff"; and, finally, the "Request String" lists the parameter names that are requested from the image.

As expected, the response follows the same structure defined before and sends the Image Information required to fulfil the request. An example of an image information string was provided by the team working at Bosch to facilitate the process.

Once these five commands are correctly sent and responded to, the connection and login phase is done and a notification created by the RPS should appear, informing the user that the login is done, as well as a visual alteration that the process was successful and the RPS is ready to send more command to the panel.

3.6.2.6 Disconnect and Abort

If a command was sent to the Simulator and the reply takes too long to reach the RPS, the Abort command is used to cancel the transaction. Once the communication process is complete and every command and response transaction that needed to be tested was done, the Disconnect command is used to terminate the connection. These are two simple commands with no data attached, just the header with the respective Message Type which will trigger the expected action.

3.6.2.7 Command Mapping and Serialization

There's a group of classes that map the different commands and responses shown previously. Each command class derives from the abstract class "RAMVCommand" and every response from the abstract class "RAMVResponse". Each command class should have a deserialization method defined, since the initial class, from which the other classes derive , has the virtual method called "Deserialize". However, if the method isn't implemented, when trying to deserialize the command it will throw an exception and utilize the dynamic method referred to previously. For the serialization of the response classes, the process is identical, each class should have a serialize method implemented, otherwise the dynamic method is used for that purpose. I will now demonstrate and explain the method of dynamic serialization, based on reflection, for a response that does not have a custom method implemented for serialization.

Listing 3.2: Dynamic Deserialization method

```
1 static private void Deserialize(Stream stream, object obj){
```

```
2     Type type = obj.GetType();
3     PropertyInfo[] propertyInfos = type.GetProperties(BindingFlags.Public |
          BindingFlags.Instance);
4
5     foreach (var pro in propertyInfos)
6     {
7        if (pro.PropertyType == typeof(Byte))
8           pro.SetValue(obj, ReadByte(stream));
9        else if (pro.PropertyType == typeof(String))
10          pro.SetValue(obj, ReadString(stream));
11       else if (pro.PropertyType == typeof(ushort))
12          pro.SetValue(obj, ReadUshort(stream));
13       else
14          throw new Exception("Fail in deserializing RAMV command " + type.Name
                + ", RAMVProtocol has no idea how to deserialize field " + pro.
                Name);
15          }
16 }
```

As an example of the dynamic methods referred before, Listing 3.2 shows the method Deserialization of a command that is sent by the RPS and received by the Simulator and which does not have a custom method of deserialization. It is important to understand that the header of the command, as explained before, is similar for every command, therefore it is deserialized before this method and used to uncover the command type and create an empty instance of the class that represents that class. I will explain the code:

- Line 1 — the method receives an object "stream", of type Stream, which contains the serialized command received, minus the header which was deserialized before, and an object "obj", of type object, which is the empty instance of the command created before;
- Line 2 — the type of the object is defined, the reason why the object "obj" is received as an object to this method is that the same method is used for a variety of commands and it has to find the type dynamically;
- Line 3 — in this line, the properties of the class that represents the command type are all listed in a variable, this list will be used next to understand how to deserialize the "stream" object;
- Lines 5 to 15 — in this loop, each property from the list defined in Line 3 is examined. In each iteration, through a set of conditions, the type of the property is discovered. And another method is used to read part of the initial "stream" object based on the type of the property. Finally, the part that was read from the Stream object is stored in the object "obj", this way, when the loop ends, the "obj", which was initially empty, will have all the deserialized data that was serialized in the Stream object. If the property is not defined in the list of conditions, an exception informing the user that the deserialization method failed will be thrown.

3.7 Test Solution

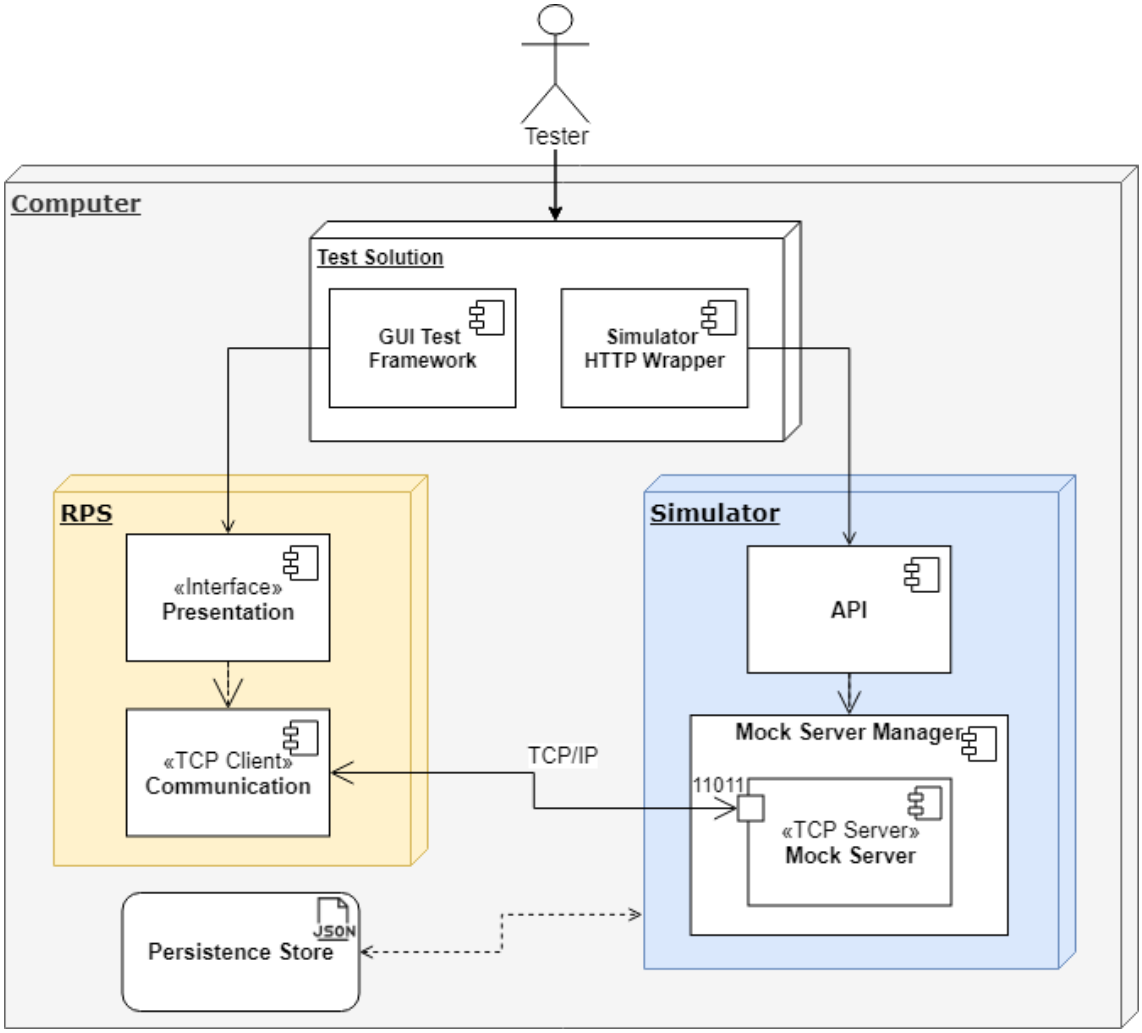


Figure 3.4: Updated Architecture

When the Simulator was implemented and the communication between it and the RPS was fully functional, there was a need to automate this process to facilitate the testing environment of the solution. At the time, to test the process, the user needed to manually send the HTTP request with the data to create the mock server and send the commands through the graphical user interface of the RPS individually to prove that the communication was working and that everything was functional. The idea of creating a test solution to make this process automated emerged immediately after it was functioning, however, to achieve this automation, there was a significant obstacle, the fact that the RPS is a legacy and complex tool without an easy way of implementing automated tests. Bosch's team had already come across the same problem and developed a Test Tool that simulates mouse clicks to interact with the RPS's Graphical User Interface (GUI), therefore, using this tool to overcome the problem was the obvious solution. After meeting with the

team and discussing the best approach method, we accomplished to create an updated version of the architecture which is demonstrated in Figure 3.4.

3.7.1 HTTP Wrapper

To start, I created a simple solution that contains methods to mimic the user action of sending the HTTP requests to the Simulator. To do that, an HTTP Wrapper, containing methods responsible for sending specific HTTP requests, was created, with the help of the C# library "System.Net.Http". Amongst the methods referred to before, there are methods to create mock servers with different configurations, e.g., empty mock server, mock server ready for connection, mock server ready for connection and login.

Listing 3.3: Creation method of a Mock Server ready for Connection

```
1 public HttpResponseMessage createMockServerForConnection(int port, string name)
      {
2         JArray stubsArray = getStubsArrayForConnection();
3
4         MockServer mockserver = new MockServer(port, name);
5         string jsonString = JsonConvert.SerializeObject(mockserver, Formatting
              .Indented);
6
7         JObject jObjectMockServer = JObject.Parse(jsonString);
8         jObjectMockServer["mockServerStubs"] = stubsArray;
9
10        var content = new StringContent(jObjectMockServer.ToString(), Encoding
              .UTF8, "application/json");
11        var result = _client.Post(mockServersURL, content);
12
13        return result;
14      }
```

As an example, Listing 3.3 shows the method for creating a mock server, which contains the stubs necessary to be ready for a connection with the RPS, which implies the transition of both the Init and Connect commands and, therefore, the respective stubs. This method, like many other in the solution, utilizes a C# library called "Newtonsoft.Json", used to handle data in JSON format. To explain this method:

- Line 1 — the method receives the port and name that should be assigned to the mock server;
- Line 2 — an object of type "JArray", from the library referred to before, is created and contains a list with the necessary stubs, retrieved from another method;
- Lines 4 and 5 — the mock server object is created with the corresponding name and port. It is converted to a String

- Lines 7 and 8 — the resulting string, from the previous lines, is parsed into a “JObject”, a class from the Json library referred before, and the “JArray” with the stubs is added to the “JObject” that represents the mock server.
- Lines 10 to 13 — the full data necessary is placed into a “StringContent” object, which will be the content of the HTTP POST request sent, in line 11, to the Simulator API. The result of the HTTP request is stored in a variable called result, which is returned to verify that the process was done according to expectations.

Following the conclusion of the HTTP wrapper and the methods that handle the content of the requests and, through the HTTP wrapper, send HTTP requests to the Simulator, a solution with unit tests that make use of the methods described before was started. This solution calls the methods and receives the responses from the HTTP requests and, through assert methods, commonly used in test projects, it determines if the results pass or fail according to the intended outcome.

3.7.2 GUI Test Tool

The Graphical User Interface Test Tool, which has been referred to before, is a tool created internally by Bosch that allows the user to plan the simulation of clicks from a mouse, in order for them to happen automatically. This tool is perfect to implement automated tests into the RPS, considering it is a legacy tool without a simple way of creating automated tests programmatically. The process required at Bosch to be able to share an internal tool with a person unaffiliated with the company is thorough, and the deadline for the project was near, therefore the team from Bosch made the decision of handling the merging of the HTTP Wrapper explained before with their GUI Test Tool. To facilitate the merging process, I created a simple TCP Wrapper, similar to the HTTP Wrapper, that sends commands equivalent to the commands sent by the RPS, this way, the testing environment can still be executed without the presence of the GUI Test Tool, and the following steps of merging can be done by replacing the method calls to the TCP Wrapper with the method calls to the GUI Test Tool.

Once the TCP wrapper was operational, the previously explained unit test solution, additionally to executing calls to the HTTP wrapper methods, also started using methods from the TCP wrapper, to allow the unit tests to execute the whole process of creating the mock server in the Simulator, through HTTP requests, and sending mock commands equivalent to the ones sent by the RPS, through a TCP connection. Once more, is important to understand that these TCP methods will be replaced by the GUI Test Tool methods in the future.

Chapter 4

System Usage and Validation

With the last chapter complete, the reader should have full knowledge of every component in the system, as well as the way the components interact with each other. To help consolidate the idea, this chapter will demonstrate an example and explain each step of the process of connecting and logging in the RPS with the Simulator. This chapter can also be used as validation, since it shows the full process of connecting and logging in. It is related to the Sequence Diagram shown before in Figure 3.2, but the detail of each step is significantly increased. Every step is going to be done manually instead of using the test solution that makes the process automatic, to make it easier for the reader to understand each step.

4.1 Prepare Environment

First, the environment needs to be prepared:

- the Simulator, which is a C# project, should be executed, so that the API is made available, I used Microsoft Visual Studio to implement the Simulator and use it to execute the solution as well. After executing the Simulator, the API is ready to receive HTTP requests;

- the RPS, provided by Bosch, is an executable. Before executing the RPS, the port defined in the RPS configurations needs to be changed to match the port of the mock server. To do this configuration, the user must access the Windows Registry Editor, go the RPS folder and change the key named "MagicPanelTransferToolSocket". Then, we should run the application and select the desired panel from the list, this is explained in the previous chapter, in the Remote Programming Software section. After that, the RPS is ready to send commands.

- throughout the implementation, debugging and testing, I used Postman to send HTTP requests to the API, and it will be used as well in this example. Postman is one of the most used API test tools, making the process of sending HTTP requests simple.

```
{
    "port": 11011,
    "name": "Test session 1",
    "protocol": "RAMV",
    "mockServerStubs": [
        {
            "expectation": "InitV2",
            "response":{
                "Status": "CMR_OK",
                "SWVersionHigh": 0,
                "SWVersionLow": 0
            }
        },
        {
            "expectation": "ConnectV3",
            "response":{
                "Status": "CMR_OK",
                "PNA": 1
            }
        },
        {
            "expectation": "LoginV2",
            "response":{
                "Status": "CMR_OK"
            }
        },
        {
            "expectation": "GetModulePassport",
            "response":{
                "Status": "CMR_OK",
                "ModulePassport": "..."
            }
        },
        {
            "expectation": "GetImageInfo",
            "response":{
                "Status": "CMR_OK",
                "ImageInfo": "..."
            }
        }
    ]
}
```

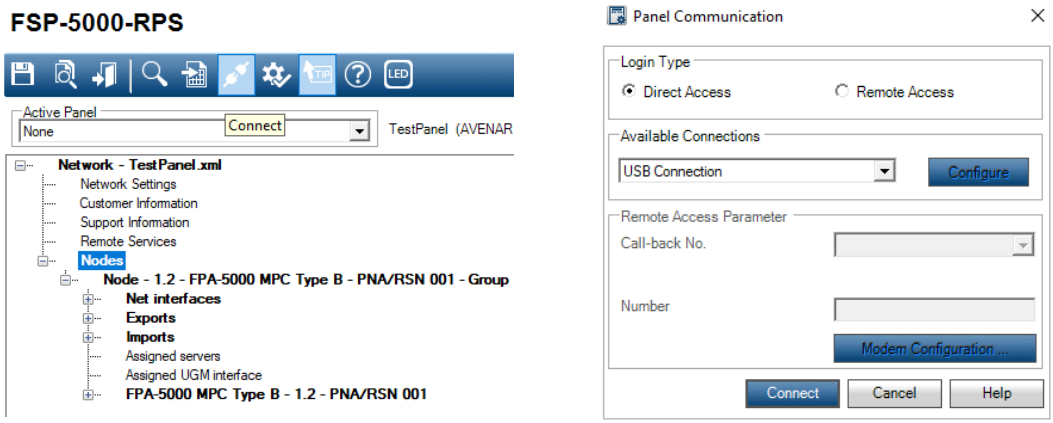
Listing 4.1: JSON for usage example

4.2 Start Mock Server

Once the environment is ready and every component is executed, the interaction between components can start. To begin this process, the mock server needs to be created, so an HTTP POST request with the data needed to create and configure the mock server needs to be sent. The HTTP request should contain the data in JSON format, as shown in Listing 4.1. The mock server will start in port 11011, have the name "Test session 1" and use the RAMV protocol. In the list of stubs, some fields, as the "ModulePassport" and "ImageInfo", have been removed for confidentiality reasons and to facilitate the demonstration. We can also perceive that the mock server, with this configuration, will be ready to receive and respond to the five commands explained before, and the ones needed for the RPS to connect and login. By using the Postman tool to send the HTTP request, the API provided by the Simulator will receive the request, create the mock server with the respective stubs, and start the TCP server which will be ready to receive the commands sent by the RPS. The process of creating the mock server was explained in detail previously in Section 4.3.

4.3 Connect

Next, with the mock server started and waiting for commands, the user should utilize the RPS for the first command transaction between the two components. Once the user has selected the desired configurations, he should press the "Connect" option, which we can see in Figure 4.1a. This option will open a new window, demonstrated in Figure 4.1b, that will attempt to start the connection when the "Connect" button is pressed.



(a) RPS Connect Button

(b) RPS Connect Window

Figure 4.1: RPS Before Connection

When the "Connect" button is pressed, the connection process begins, the RPS tries to communicate to the selected TCP port, to create a TCP connection with the Simulator. If the TCP connection is successful, as explained before, the first command to be sent is the "Init" command.

While developing the system, I tried to make it easier for the user to understand what was happening on the Simulator side by incorporating a text output system. Listing 4.2 demonstrates the output from the Simulator when receiving the "Init" command, there is a similar output for each command, however, I will only show the example in Listing 4.2 to keep the example brief and compact.

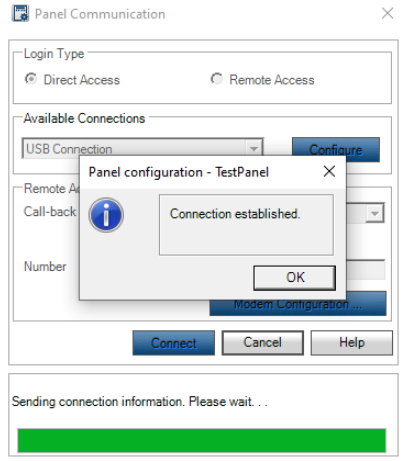
```
1 --------------------------------
2 ------New Command Received------
3 --------------------------------
4 Header bytes: 0x35 0x00 0x00 0x88
5 Header: IsResponse: False; Type: InitV2; Sequence Number: 0; Command Length:
      136;
6 Header Type:InitV2
7
8 Command Body Received:
9 RAMVSWVersion = 769
10 RAMVProtocolVersion = 256
11 Timeout = 3
12 AppType = 1
13 TemporayPath = C:\Users\Guilherme\AppData\Local\Temp\637628981060434499
14 Salt:
15 0xF3 0x45 0x5C 0x5A 0x3B 0x43 0x51 0x42 0x35 0x35 0x35 0x65 0x42 0x35 0x4A 0x66
16
17
18 Comparing the command type with stubs
19 Received command of type: InitV2 and found a Stub for it
20
21 ---------------------- RAMVInitV2Response ----------------------
22 Status = CMR_OK
23 SWVersionHigh = 0
24 SWVersionLow = 0
```

Listing 4.2: Simulator output for Init command

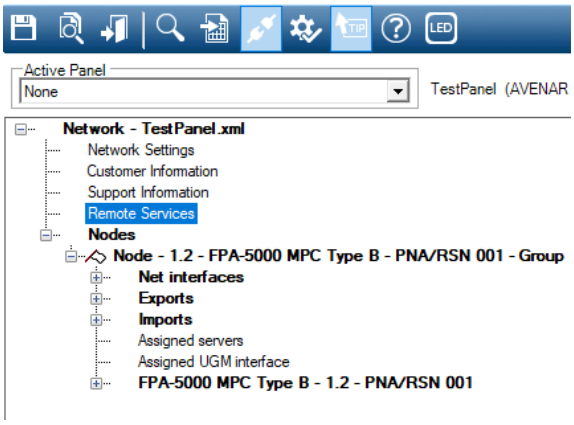
In Listing 4.2, the first 15 lines show that a command has been received, and the data retrieved from the command. From line 4 to 6, the header of the command is processed, where we can see the different fields, including the message type and from line 8 to 15, we can see the contents of the data field from the "Init" command or, as called in the listing, the body of the command. Line 18 and 19 tell the user that the command is being matched with the expectations and that, in this case, one of the stubs match the command, so the replying process starts. The last 4 lines show the response sent back to the RPS, the response to the "Init" command only has the three fields shown in the listing, apart from the header of the response.

After the "Init" command is received and the response is sent back, the RPS now sends the "Connect" command to the Simulator. The process of replying to it is similar to the one described previously. If the transaction is done correctly, the RPS should show a message, which can be seen if Figure 4.2a, that informs the user that a connection was established. After this, a flag

should appear next to the Node, which indicates that it is connected to a panel, demonstrated in Figure 4.2b.



(a) RPS Connection Established Notification



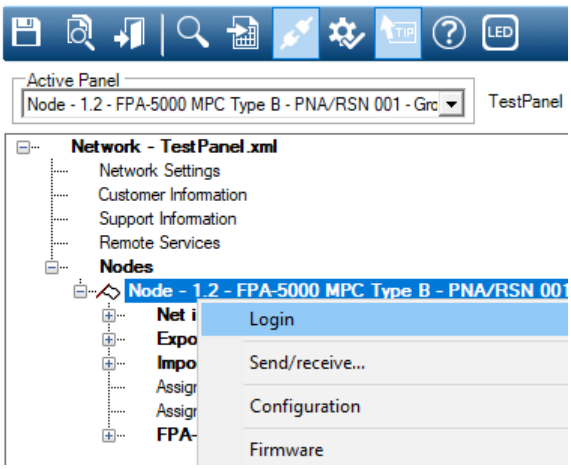
(b) RPS Connected Flag

Figure 4.2: RPS After Connection

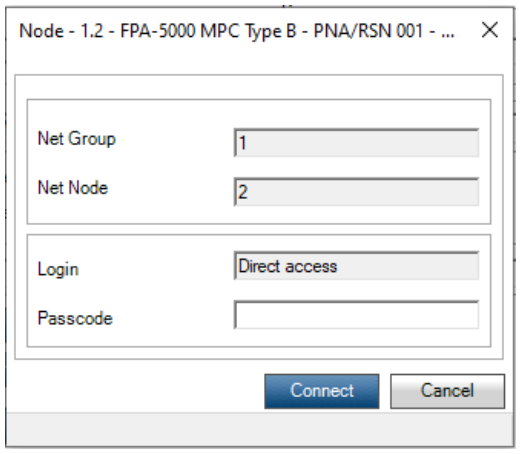
The connection phase is completed and the next step is to login.

4.4 Login

The login phase is similar to the connect phase, the user must select the Login option in the RPS, by right-clicking the Node, which is connected to the Simulator, this option is shown in Figure 4.3a. By pressing the “Login” button, a new window will appear, demonstrated in Figure 4.3b, which has a few configurations and a “Connect” button that will start the transaction of commands to achieve the login.



(a) RPS Login Option



(b) RPS Login Window

Figure 4.3: RPS Before Login

As explained in the previous chapter, the login process is composed of three different commands, “Login”, “GetModulePassport”, and “GetImageInfo” commands. These three commands are sent successively in the order they are listed before. First, the “Login” command is sent, the Simulator compares it with the stubs and replies with the predefined response, then the process is similar to the “GetModulePassport” and finally to the “GetImageInfo”.

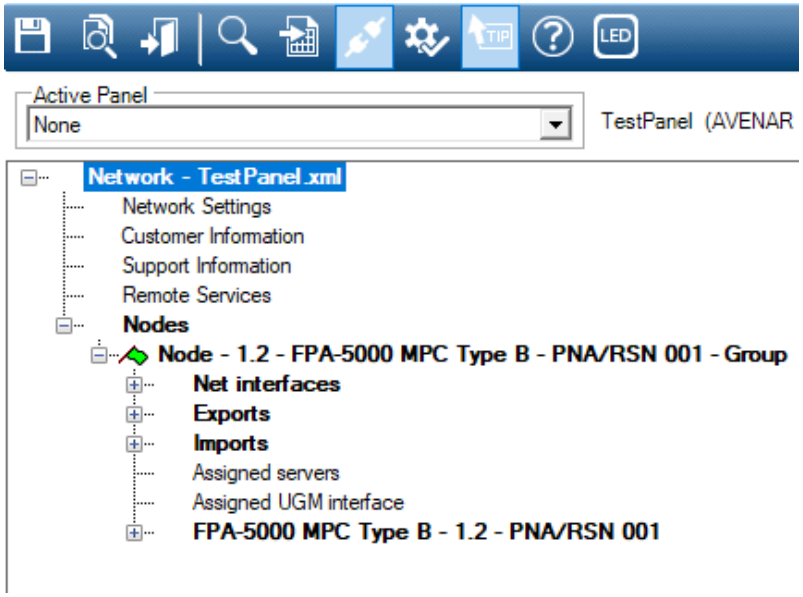


Figure 4.4: RPS Flag indicating Login

With the transaction of the three commands, and if everything was successful, the RPS will now be logged in to the Simulator. The flag next to the Node, signaling that it is connected, should now turn green to indicate to the user that the Node is also logged in. This green flag, which can be seen in Figure 4.4, demonstrates that the process of connection and login was successful and is used as one of the main validation items for the system.

Chapter 5

Conclusions and Future Work

This chapter concludes the document, focusing on results of the solution, based on the requirements defined in the beginning of the project and in the future work that can still be done to the solution.

5.1 Results

5.1.1 Successful Connection and Login

Since the start of the implementation, the Proof of Concept (POC) that was specified was the successful connection and login through the RPS with the Simulator. It has been shown, in the previous chapters, that this process is working correctly and that this POC was accomplished. Additionally, a full demonstration, explaining with detail each step to achieve this POC has been displayed in the previous chapter. With this solution, the problems listed in the introductory chapter, specifically in Section 1.1, are solved:

- Having the physical systems present for each test scenario and every time the test need to be executed is expensive. Our solution removes the necessity of the system being present;
- The RPS (called Configuration Software in the first chapter) is able to communicate with several devices and sets of devices, and to test every feature of the RPS, a large group of devices would be needed. Our solution allows the tester to simulate any set of devices and test every feature of the RPS, as long as the tester has the respective information of the system to be able to simulate it;
- Although the knowledge and coordination between teams still exists, to share for example devices configurations, the test process of the software no longer requires full attention of professionals that focus on the physical systems;
- The testing of failure scenarios was something that was discussed since the beginning and, although a mechanism allowing for other failure scenarios was not implemented, the timeouts can easily be tested by not setting up a stub for a specific command. The creation of

a mechanism ready for other failure scenarios will be discussed next, in the Further Work section.

5.1.2 Test Solution

The Test Solution was the next step that needed to be implemented to accelerate the process of testing the software. The test solution works, but currently is only using the Simulator side correctly and simulating the RPS commands. As explained before, the methods that simulate the commands from the RPS should be replaced by the methods from the Bosch's GUI Test Tool, which will make the test solution operational with the full system, including the RPS. The main goal of the Test Solution was to create a group of methods to help the team at Bosch gather information of how to interact with the Simulator, so they could use it in their own Test Solution, and that has been achieved.

5.2 Further Work

5.2.1 Failure Scenario Mechanism

It has been stated that testing failure scenarios is difficult with an actual physical system and that mocking the system facilitates the process. Although the current solution allows for timeout testing, by not defining any stub for a specific command, a mechanism that allows testing for other failure scenarios would make the system more robust.

5.2.2 Upgrade Test Solution

This has been referred to several times, the Test Solution does not operate all the components of the system, only the Simulator side, and to be ready to have full system coverage, the test solution must utilize the GUI Test Tool to also operate the RPS. The other topics in this section are good features that increase the coverage and robustness of the system, but this topic is crucial, considering it makes the project fully functional and ready to be used in a real working environment.

5.2.3 Adding Protocols

Currently the only implemented protocol is the RAMV protocol, it has a DLL developed specifically for it. In the future, other protocols can be included, by developing a DLL for each protocol.

5.2.4 Dynamic Stub Creation

This topic was discussed in several meetings throughout the last months as an alternative to the previous topic. The idea of making the creation of stubs dynamic, removing the necessity of a specific library for each protocol would drastically increase the possible use of the system in every cyber-physical systems. Although we thought of a solution to achieve this dynamically created stubs, it was later into the implementation and the work needed to achieve this solution

was considerable, that being the reason why it is in the Further Work section. The idea thought of, would remove some complexity from the Simulator side, by not having to create a specific and complex DLL for each protocol, but would increase the complexity in the HTTP request data that would need to be sent to the Simulator API. Seeing as some specific commands need special treatment and some protocols, including RAMV, have data that should be stored and used throughout the whole communication process, include encryption and decryption of data, and decoding the data from the commands would be a challenge without a specific DLL mapping the commands and how to serialize and deserialize.

5.3 Conclusion

To finish this document this section will present a brief conclusion. It will summarize the work done, the results and the work left to be done. Working on creating a framework like the one described in this document proved to be challenging, I had to tackle new fields that I have never had before, e.g., mock testing and such a low level communication protocol. Nevertheless, the process made me learn a lot, motivated me to grow and learn and, ultimately, the requirements defined for the project were almost all completed. As discussed throughout the document, the Proof of Concept was attained, with the presence of the framework, the process of connecting and logging in with the RPS without the actual physical panel was successful. There is still a lot of work to be done for this framework to become robust and applicable to real complex scenarios. The most important one being the feature that allows the tester to create dynamic test scenarios without the need to have to create a complex DLL or class to handle the mapping of each command.

References

[1] D. E. Krutz and M. Lutz, "Bug of the Day: Reinforcing the importance of testing," 2013 IEEE Frontiers in Education Conference (FIE), Oklahoma City, OK, 2013, pp. 1795-1799.

[2] D. Spadini, M. Aniche, M. Bruntink and A. Bacchelli, "To Mock or Not to Mock? An Empirical Study on Mocking Practices," 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, 2017.

[3] S. Mostafa and X. Wang, "An Empirical Study on the Usage of Mocking Frameworks in Software Testing," 2014 14th International Conference on Quality Software, Dallas, TX, 2014, pp. 127-132

[4] R. Rajkumar, I. Lee, L. Sha and J. Stankovic, "Cyber-physical systems: The next computing revolution," Design Automation Conference, 2010.

[5] X. Zheng and C. Julien, "Verification and Validation in Cyber Physical Systems: Research Challenges and a Way Forward," 2015 IEEE/ACM 1st International Workshop on Software Engineering for Smart Cyber-Physical Systems, 2015.

[6] Pallakku, Selvaprasanth & N., Sathiyanathan. "A Brief Study on IoT Applications. Journal of Scientific Research and Development". 2020. 4. 23-27.

[7] Zhou, Xin & Gou, Xiaodong & Huang, Tingting & Yang, Shunkun. "Review on Testing of Cyber Physical Systems: Methods and Testbeds". 2018.

[8] Sanislav, Teodora & Miclea, Liviu. "Cyber-physical systems - Concept, challenges and research areas". Control Engineering and Applied Informatics. 2012. 14. 28-33.

[9] “Comparison of API simulation tools,” Oct. 14, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_API_simulation_tools

[10] "Beeceptor Docs." [Online]. Available: https://docs.beeceptor.com/

[11] James D Bloom, "MockServer documentation". [Online]. Available: https://www.mock-server.com/

[12] "Wiresham". [Online]. Available: https://github.com/abstracta/wiresham

[13] "WireMock.Net". [Online]. Available: https://github.com/WireMock-Net/WireMock.Net

[14] Adam Trujillo, "Parasoft Virtualize Documentation". [Online]. Available: https://docs.parasoft.com/display/VIRT9106

[15] "Mountebank - over the wire test doubles." [Online]. Available: http://www.mbtest.org/

[16] "MbDotNet". Available: https://github.com/mattherman/MbDotNet

[17] Robert Bosch GmbH, “ Use Cases for Panel Simulator Working Document”, Jun. 15, 2020.

[18] A. Neumann, N. Laranjeiro and J. Bernardino, "An Analysis of Public REST Web Service APIs," in IEEE Transactions on Services Computing. 2018.

[19] “Notifier by Honeywell: Fire Detection System”. Available: https://www.notifierfiresystems.co.uk/

[20] "What is a DLL". [Online]. Available: https://docs.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library