An Investigation of Testing Environments for Backend/Cloud-Services

Increasing Efficiency of Testing in the Automotive Industry

Computer Science Algorithms, Languages and Logic

Denise Glansholm
Patrik Ingmarsson

Department of Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

An Investigation of Testing Evironments for Backend/Cloud-Services

Denise Glansholm
Patrik Ingmarsson

Department of Computer Science and Engineering
Division of Computer Science
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

An Investigation of Testing Environments for Backend/Cloud-Services
Denise Glansholm
Patrik Ingmarsson

Supervisor: Koen Claessen, Department of Computer Science and Engineering
Supervisor: Jörgen Börjesson, Delphi Automotive
Examiner: Mary Sheeran, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Division of Computer Science
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Department of Computer Science and Engineering
Gothenburg, Sweden 2016

An Investigation of Testing Environments for Backend/Cloud-Services
Denise Glansholm
Patrik Ingmarsson
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

Products using cloud services are often problematic to prove correctness and quality of. In automotive industries specifically, the clouds are commonly developed alongside the products and controlled by separate companies. If the clouds are inaccessible or uncontrollable, efficient testing using both positive and negative testing is impossible.

This Master's Thesis strives, in cooperation with Delphi Automotive, to improve the testing efficiency in the automotive industry by designing a testing environment for communication from an application to back end/cloud-services. To resolve the issue with non-thorough testing, a cloud simulation environment was to be developed. Furthermore, to enable portability to different projects, the simulation should have complete configurability.

The result was a general testing environment that can be applied to almost any product using HTTP/HTTPS-communication. An evaluation of efficiency of the new testing procedure was performed and the efficiency was found to increase due to the overall higher thoroughness of the testing. However, the time to write tests and configure the environment was found to increase, as a result of the open and general nature of the testing environment. Worth mentioning is that this setup time is predicted to decrease in a real-world project. The testing environment created a time consumption overhead of 1-2 hours. By writing targeted test cases, several previously undiscovered bugs were found relating to retry-handling.

Overall, spending some time developing a general testing environment seems to be profitable. Furthermore, extending the project with an easier configuration interface might be worthwile to further mitigate the setup time.

Keywords: Test , Testing, Mock, Cloud, Back end, Services, Automotive.

Acknowledgements

We would like to thank Delphi for the opportunity to perform the Master's Thesis work in cooperation with them, and carry out the project in their offices. We highly appreciate the Delphi personnel for all the technical guidance and social support we received. Furthermore, we would specifically like to express our gratitude towards our supervisor at Delphi, Jörgen Börjesson, for helping us plan and prioritise during the different phases of the projects, as well as continuously providing feedback and helping keeping our spirits up. We would also like to thank our Chalmers supervisor, Koen Claessen, and our Examiner, Mary Sheeran, for keeping us on track at all times and giving helpful advice and tips throughout the thesis work.

Denise Glansholm, Gothenburg, August 2016
Patrik Ingmarsson, Gothenburg, August 2016

Glossary

Back-end The part of a system that indirectly supports the front-end application by, e.g., performing heavy operations and accessing remote resources.

Cloud A Distributed System that provides some service, e.g., storage.
Cloud Service A resource that is provided over the Internet.

Front-end The part of a system that users (humans or other programs) directly interact with.

HTTP Hyper Text Transfer Protocol.
HTTPS Hyper Text Transfer Protocol Secure.
HU Head Unit.

JSON JavaScript Object Notation.

LTS Long Time Support.

NUC Next Unit of Computing. A small-form-factor PC designed by Intel.

OEM Original Equipment Manufacturer.

SSL Secure Sockets Layer. An older standard protocol, now often replaced with TLS.

TLS Transport Layer Security. A newer standard security protocol, often replacing SSL.

Vertical Slice A milestone in a software management project, focusing on demonstrating functionality across all components of the project.

XML Extensible Markup Language.

Contents

1

Introduction

In this chapter, an introduction to the Master's Thesis is given. Section 1.1 provides background and context, as well as a description of a practical and academic void that exists today. In Section 1.2, the purpose of this thesis is given. A problem statement is defined in Section 1.3 and under Section 1.4 related work are discussed. Lastly, Section 1.5 details the limitations of the thesis.

1.1 Background

The testing phase often accounts for about half the time and cost when developing a system [1]. Despite this, software testing is still far from a refined science and the gap between the industrial and academic world is still noticeable [2].

One of the industries where this scientific gap is prominent is the automotive industry. There, many premium original equipment manufacturers (OEMs) now make use of back-end/cloud services for several in-car functions. These services include, but are not limited to, booking car service, over-the-air software updates and route planning. The cloud services are often developed alongside the head unit (HU), the centrepiece of a car's multimedia system and the unit that utilises these cloud services.

Figure 1.1 shows an example of the HU booking a car service and the interaction between the HU, cloud and driver. Here it might, for instance, be important to test that the HU only notifies the driver and presents time slots (step 4) when it is safe (for example when the car is stationary). However, in order to do this, a cloud is necessary to perform steps 2 and 3. This clearly shows the importance of having access to a cloud during testing.

During the development of the cloud, its services may not be available for testing the HU, and thus delaying the testing until later stages in the development process. Furthermore, when the cloud service is available, it still only offers the possibility of Positive Testing, that is, testing to verify that the HU acts as intended when given valid data, which does not help produce fault tolerant systems.

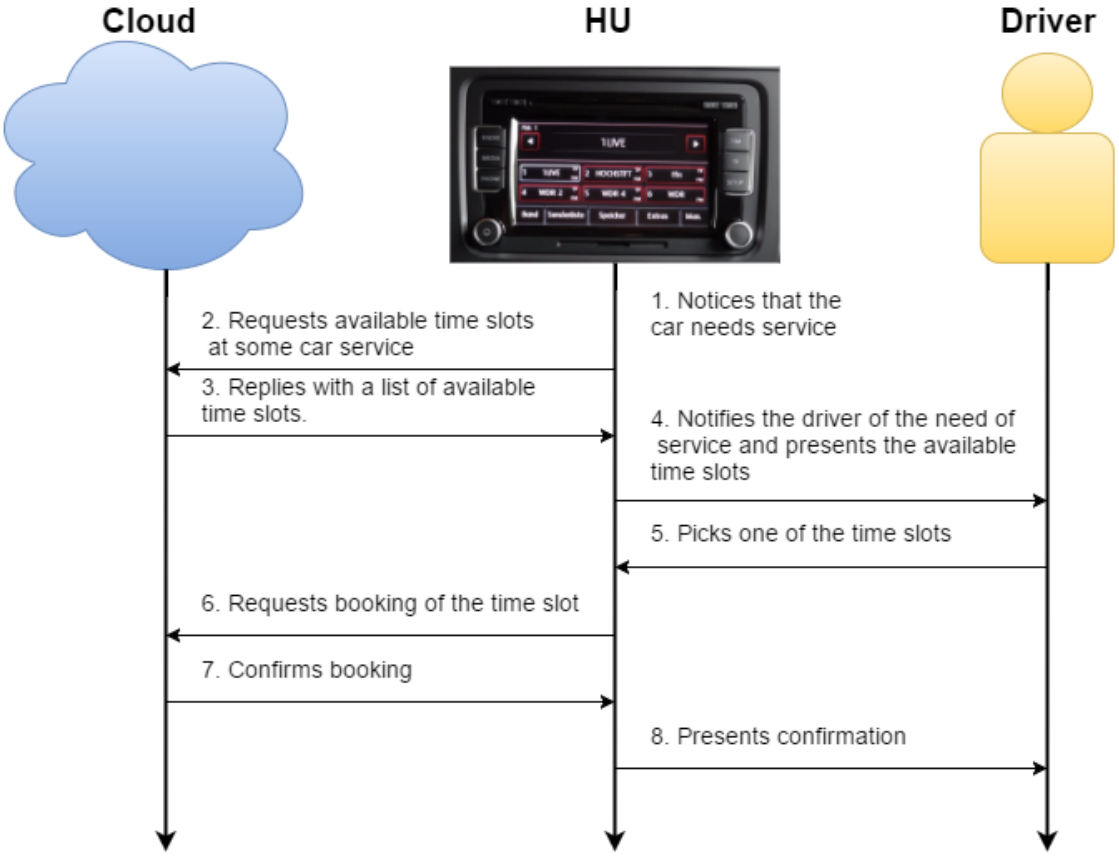

Figure 1.1: The interaction between Driver, HU and Cloud when booking car service.

Statistics show that the cost of correcting code defects increases exponentially with time as the software project moves on to later development stages [3]. Furthermore, research indicates that when returning to previously written code, one has trouble achieving the same level of immersion [4]. Therefore, supporting testing of smaller parts of a system on earlier stages in the development process should make the debugging cheaper and more efficient. Moreover, allowing Negative Testing should produce more fault tolerant systems, an important criteria in the automotive industry.

1.2 Purpose

The purpose of this Master's Thesis is to investigate the testing procedures in the automotive industry, with focus on the parts where communication with backend/cloud-services are handled. The project concentrates on implementing a fully controllable cloud simulation that is able to simulate an arbitrary online service. It also analyses how to integrate it into a testing procedure and how to evaluate the efficiency of it, in relation to the currently existing testing.

Thorough testing is meaningful in most systems, but can be considered absolutely critical in the Automotive Industry. This project seeks to improve testing in the following five ways:

- The ability to perform testing on an environment similar to the environment the end product should be run on. This creates more thorough testing, as it can show bugs that might not otherwise appear.
- Complete control of the cloud, which results in easy Error Injection testing (in other words, Negative Testing, explained in Section 3.2.3).
- Decoupling of the development teams. Smaller, independent teams are more manageable and mitigate the planning and syncing time between the teams.
- Automated testing on code changes that affect cloud services. A consequence of this is that more thorough testing will be performed, which generates more robust systems.
- Mitigate the time to configure the test environment and write tests. Leaving tests out due to time consumption allows for more bugs and longer time debugging which in turn yields even less time to write proper tests

1.3 Problem Statement

Questions connected to the purpose of this project include:

Q1 Is it possible to create a testing service for client-server communication that is easily adaptable to different products?

Q2 Can such a service be efficient? Or rather, does it allow a testing procedure that is more efficient than the procedure currently adopted by Delphi Automotive?

Sub-questions to elaborate the two above:

Q3 Is it possible at all to simulate a cloud that satisfies the testing needs of the cloud services used in the automotive industry?

Q4 Can the new simulation service be incorporated in existing testing frameworks?

1.4 Related Work

Many have found that Clouds are useful to test applications by simulating real-world environments and traffic [5, 6, 7], however, these tests and simulations are often targeted for back end applications and services. Many others have studied how to test the Clouds themselves in an efficient and correct way [8, 9].

However, simulation of the back end and cloud services themselves, in order to properly test front end clients, seems to be largely left out in academia.

MockServer is the only project found having capabilities nearly fitting the demands of this paper. It is a configurable server for testing services using HTTP and HTTPS communication [10]. One major difference is that the MockServer is still a remote server, whereas the test server for this project will be run locally, allowing even more control. Furthermore, this project's server supports several special actions not accepted in the MockServer (or HTTP/HTTP communication in general), such as waiting, easy repeating, closing connections etc. The full list of actions supported can be found in Table 5.2. Lastly, the MockServer is written in Java, while this project is written in C++, according to the limitations set in Section 1.5.

1.5 Limitations

Due to a limited time frame, only a part of the new testing environment was implemented as a proof of concept. The focus was on testing the component foundation services in the Volvo project at Delphi.

Furthermore, as a request from Delphi, the new testing environment runs on a Linux machine running Ubuntu 14.04 LTS and developed in C++; Exploration of other Operating System and Code Language combinations has been left out.

The testing environment will be exclusive to services using the HTTP or HTTPS protocol for communication.

2

Method

This chapter describes the methods used to fulfil the project aim. In Section 2.1, the methods used for the implementation of the environment are detailed. How the testing of the environment was performed is described in Section 2.2, and how the evaluation of success was defined is detailed in Section 2.3.

2.1 Implementation

As described in section 1.5, only a proof of concept was implemented. This was done using the method of Vertical Slicing [11]. This means that support for only one function is implemented to prove that the goal can be achieved and how. In this case, this means that the testing environment only connects to one single project, namely the Volvo infotainment project at Delphi. It also only supports testing of a few services in the project.

The process of implementing the design was divided into different parts. The first goal was to implement the software to work manually, meaning that a user excites a testing suite. When that was done, the work of incorporating the testing software into the current automated testing service at Delphi was assessed and if it was deemed too great, it was to be left out of the project.

The software was implemented according to scrum with biweekly sprints. Informal meetings between the project members was held daily, and in the end of each sprint a meeting was held with the supervisor from Delphi.

2.2 Testing the Testing Environment

Testing a product using a testing environment, such as the one described in this Master's Thesis, is helpful to facilitate the testing process. However, this requires that the testing environment in question behaves correctly as well.

Naturally, an option would be to write another intricate test environment to test the environment in this project's scope. However, this means that the need for testing just propagates. For example, how does one know that the code that tests the testing environment is correct. In order to avoid this problem, a very simple testing program, highly unlikely to contain bugs, was implemented. This consists of a dummy client that connects to the testing server, exhausts the rule set and checks for "correct" response from the server. This "correct" response might in fact be invalid to a real client, but may be implemented in the rule set to enforce Negative Testing (see Section 4.2.6), and is therefore considered desired behaviour from the testing server.

2.3 Evaluation of Efficiency

With the new environment and procedure, an evaluation, as well as a comparison to the old method, was performed. The aspects which define efficiency in this project and how to compare the two procedures is described in the sections below.

2.3.1 Similarity to Native Environment

As mentioned in Section 1.2, it is desired to perform testing in an environment similar to the environment that the end product should be run within. This factor is divided into the following categories, in order of increasing superiority:

1. Testing the software in an environment different than target environment.
2. Testing the software in a simulation of target environment.
3. Testing the software in target environment.

2.3.2 Code and Input Coverage

The code and input coverage is measured in three different ways:

Firstly, the thoroughness of the testing performed is determined by running both the new and the existing testing environment on the same software version, and counting how many bugs that can be found by each. Instinctively, finding a higher number of bugs yields a higher score.

Secondly, the diversity of tests that are supported is a factor. Supporting both positive and negative tests is more valuable than only being able to perform one or the other.

Thirdly, being able to perform the same tests on a developer PC and in an automated testing service is preferable due to consistency.

2.3.3 Time Consumption

An estimation of the time consumption of both the current and the new test environment should be made. This estimation should include the time consumption of setting up the environment, writing test cases and implementing necessary support components. Naturally, the lower time is more efficient in this aspect.

Another important aspect of time consumption is when testing can be performed. As mentioned earlier, it is highly beneficial to find bugs as early on in the development phase as possible. Therefore, a list sorted by increasing priority is constructed as follows:

1. Ability to perform tests only on a developer PC, or only on a remote server.
2. Ability to perform some tests on a developer PC and some on a remote server.
3. Ability to perform all tests on a developer PC and on a remote server.

3

Testing Theory

The first part of this chapter gives some basic knowledge of testing. Section3.1 gives a short look-back on the views of testing. Fundamental testing types are presented in Section 3.2 followed by methods using them in Section 3.3.

3.1 The View of Testing Through History

In the early days (1950s) testing and debugging were the same thing. Back then, much of the testing was done by hand, and was not as much Dynamic Testing (Section 3.2.1) as it was studying information flow and scrutinising code, in other words, highly manual Static Testing (also Section 3.2.1).

In the mid 1980s, the art of testing was still not a strict science. For example, the belief in crystals and energies of that time allowed paid software designers to search for bugs in their programs by dangling crystals over source listings [12]. Although this may not have been common practice, the anecdote divulges the attitude towards testing in this time period.

Today, software testers may not put as much faith in bohemian powers, but despite this, software testing still has a long way to go before it is a refined science [12]. Although software development seems to have its roots in mathematics [13], testing appears to exist only as a necessary evil in the industry, which might explain the historically slow advancement of the field.

It seems as if the percentage of time used for testing has been constant, at about 50%, through history. Despite this it was not until 1975 that it was proposed to actually plan for this amount of testing [14]. The same proportion of testing is used even today [1], although modern tools have made the testing overall more efficient.

3.2 Types of Testing

Today, there exists different types of testing to ensure desired behaviour and robustness of software. The sections below (3.2.1, 3.2.2 and 3.2.3) explain some major testing concepts and how they relate to each other.

In order to exemplify different testing methods, first consider a simple program, whose only function is to calculate the sum of two integers. Once started, it should prompt the user to enter two integers. When provided with two integers it should return the sum. In Figure 3.1, the user enters the two integers 4 and 5, and the program returns 9, which is the sum of the two. An example code that would produce the behaviour from Figure 3.1 can be seen in Listing 3.1. In the following sections, this program will be tested in a variety of ways.

```
Please enter two integers to be summed.
> 4 5
9
```

Figure 3.1: An example of a successful run of a simple summation program.

```
1#include <iostream>
2using namespace std;
3
4int main()
5{
6  unsigned int a,b;
7  cout << "Please enter two integers to be summed.\n> ";
8  cin >> a >> b;
9  cout << a + b << "\n";
10  return 0;
11}
```

Listing 3.1: Code version 1 of a simple summation program in C++.

3.2.1 Static and Dynamic Testing

Dynamic and Static Testing (or Static Analysis) are the two main ways to test software. The dynamic approach entails execution of the program to be tested. Generally, the tester gives the program some test input, executes it and records its behaviour and response [15].

Static Testing involves analysing the code without execution, either manually (which can also be called program understanding) or using an automatic tool to create, for example, a data-flow graph [16]. Unlike with Static Testing, Dynamic Testing does not necessarily require insight into the code and can be performed automatically, saving precious time for developers and testers.

The advantage of using Static Testing is that it can be performed very early in the code life-cycle, which leads to lower rework-costs. Furthermore, some types of defects may be easier to find using static testing. These include: deviation from standards, missing requirements, design defects, inconsistent interface specifications and non-maintainable code [17].

An observant reader may have noted that the code in Listing 3.1 makes use of *unsigned* integers. The specification of the program in Section 3.2 allows negative integers; however, using unsigned integers to represent negative integers may result in undesired behaviour. This is an error which could be found by only inspecting the code, and, in other words, an example of Static testing.

An ambitious reader may have compiled the code, tested various inputs and observed incorrect behaviour, as in Figure 3.2. They have then performed Dynamic Testing and have stumbled, maybe mindlessly, upon some deviation from the program specification. However, where the error occurs is still unknown. To find and correct the error, one might resort to selective Static testing or debugging.

```
Please enter two integers to be summed.
> -2 1
4294967295
```

Figure 3.2: An example of an unsuccessful run of a simple summation program.

3.2.2 White Box and Black Box Testing

Two versions of Dynamic Testing (mentioned in Section 3.2.1) are White Box and Black Box Testing. In White Box testing, one can observe the organisation of the software and maybe even the code. With this knowledge, a tester might be able to figure out a combination of inputs, that exercises every part of the software and, ideally, exhausts every possible path in the software. This can be very time consuming for complex software.

Table 3.1: White Box test cases for exhausting the code of a simple summation program.

Input	Expected Output	Generated Output
1 1	2	2

When testing the software as a black box, one does not know anything about the internal functions. The only knowledge a tester possesses is what output the software should produce given some input. In order to guarantee correct implementation, the tester must exhaust all possible inputs, both valid and invalid, that is, both positive and negative testing (Section 3.2.3). To avoid an infinite number of test cases, one can categorise the input into a finite number of categories and select a few cases out of each category.

Table 3.2: Black Box positive test cases of the input of a simple summation program.

Input		Expected Output	Generated Output
-1	-3	-4	4294967292
-1	0	-1	4294967295
0	-3	-3	4294967293
0	0	0	0
1	0	1	1
0	3	3	3
1	3	4	4
1	-3	-2	4294967294
43	106	149	149
73499	-314	73185	73185
1398	-4921	-3523	4294965171
4294967295	1	0	0

Often, both White and Black Box testing are performed to a certain degree. A developer might perform some White Box testing to check basic functionality and a few special cases before sending the product to a designated tester, performing quantitative Black Box Testing.

In the previous section, an error in the code was pointed out, and it was explained how one could find it using Dynamic Testing. Now, using a more structured approach, the White Box test cases in Table 3.1 are first used to exhaust the code. Seeing that the test is successful, Black Box testing can now be used to further test the software.

Since there are an infinite amount of integers, the set of possible input is infinite (not accounting the restrictions of integer representations in computer systems). A limitation of the input is a must. The extreme cases are those which must be tested most thoroughly. Here, they are combinations where the two integers and the result have a negative sign or different signs, or when the capabilities of the type representing the integers is surpassed. Observe, only valid test cases are tested here, invalid test cases are the subject of Section 3.2.3. In Table 3.2 a collection of tests and their results are shown.

```
1#include <iostream>
2using namespace std;
3
4int main()
5{
6   int a,b;
7   cout << "Please enter two integers to be summed.\n> ";
8   cin >> a >> b;
9   cout << a + b << "\n";
10  return 0;
```

```
11 }
```

Listing 3.2: Code version 2 of a simple summation program in C++.

It is observed that the software does not behave as desired, more specifically, the software seems not to be able to correctly represent negative results. The origin of the issue is located to the use of *unsigned int* instead of *signed int*, and has been corrected in Listing 3.2. The test cases for Black Box testing are revised to contain relevant extreme cases. The new test results in Table 3.3 are successful.

Table 3.3: Black Box positive test cases of the input of a revised simple summation program.

Input		Expected Output	Generated Output
-1	-3	-4	-4
-1	0	-1	-1
0	-3	-3	-3
0	0	0	0
1	0	1	1
0	3	3	3
1	3	4	4
1	-3	-2	-2
43	106	149	149
73499	-314	73185	73185
1398	-4921	-3523	-3523
2147483647	1	-2147483648	-2147483648

3.2.3 Positive and Negative Testing

In the previous Section, examples of White and Black box testing using only valid input data were shown. They are examples of Positive Testing. More formally defined, Positive Testing is a testing process where a system is validated against a valid input data set. This is generally the absolute first type of testing that a developer/tester resorts to.

It may be trivial to see the importance of a system working as intended with valid input; however, it is equally, if not more, important that a system can handle invalid input as well. Of course, the simple summation program cannot have serious consequences, but consider a brake system in a car: if the driver in a moment of panic steps on both the brake and the gas pedal at the same time, which can be considered as invalid input, it is crucial that the brake system starts braking and not crashing.

In Negative Testing, the tester studies the behaviour of the program when it is given incorrect or invalid data. The goal is to ensure that it raises errors when it is

supposed to and handles faults quietly when it is appropriate. Only the imagination of the tester sets the limits to what is tested. This is an important step in creating a stable program [18].

The description of the simple summation program does not mention any desired behaviour when faced with invalid input; however, it may still be interesting to study. In Table 3.4 a small collection of negative test cases are seen together with their respective generated output. The only entries that make somewhat sense are the first and the last; the rest do not produce anything useful. It might be wise to consider to validate the user input and prompt instructions how the summation program is used properly.

Table 3.4: Black Box negative test cases of the input of a revised simple summation program.

Input	Generated Output
1 1 1	2
a a	32765
! <	32767
103.212 12	103
Hello World	32764
3, 2	3
3 + 4	3
1 [Enter] 3	4

3.3 Testing Methods

This section presents some testing methods and how they relate to the types of testing in Section 3.2. It starts by introducing three fundamental methods of testing. In the other sections, other common testing methods are briefly explained.

3.3.1 Unit, Component and Integration Testing

As seen in Figure 3.3, a software entity can have both incoming dependencies, which call or trigger the entity, and outgoing dependencies, which are called or triggered by the entity. A software entity must be triggered in order to test it. When triggering the entity under test (EUT) from another entity, they are forced to be tested as one, since there is no possibility to precisely distinguish where an error originated. Similarly, if the EUT has outgoing dependencies which are needed to execute properly, it might be impossible to construct tests where the origin of an error is located. Such dependencies might result in that the entire project must be tested as a whole.

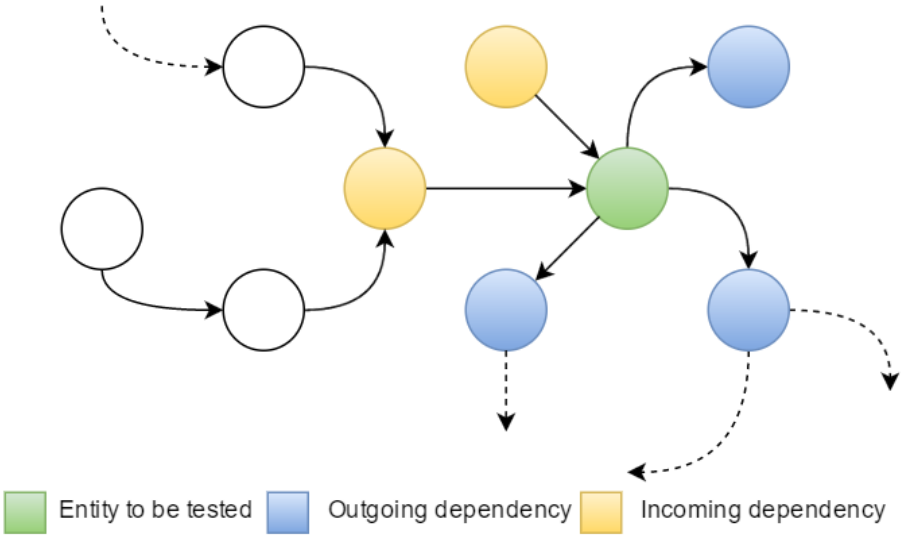

Figure 3.3: An entity to be tested in its "natural" environment.

To avoid revisiting old code, it is desirable to thoroughly test each new piece of code before moving on to the next. However, these dependencies also prevent testing since testing an entity requires having all its dependencies already implemented. The solution is to isolate the EUT by substituting all incoming dependencies with drivers and mocking all outgoing dependencies, see Figure 3.4. Substituting dependencies allows for test construction for arbitrary large entities and entities with unimplemented dependencies.

Drivers are entities that triggers execution of the EUT, so that it can be analysed. A mocked entity contains minimal functionalities to allow execution of the EUT and can assert if called correctly [19].

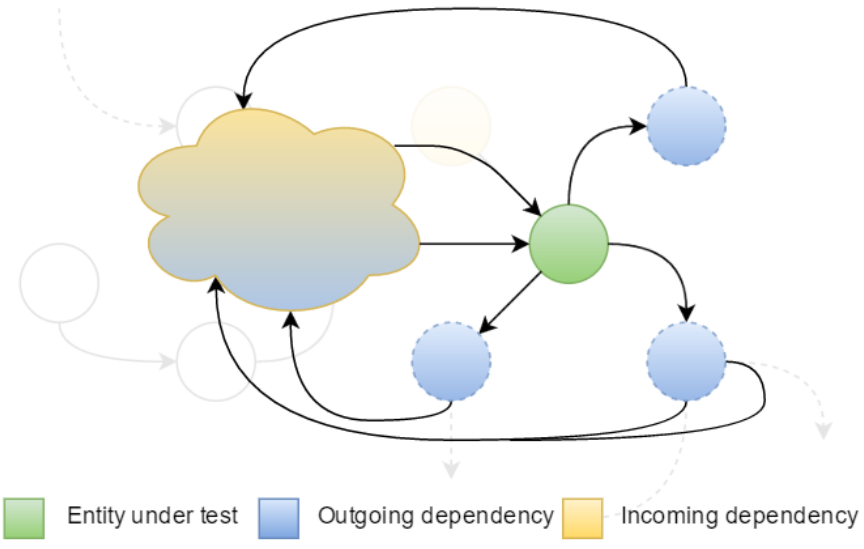

Figure 3.4: An entity under test in a mocked environment.

Commonly, there are three levels of entity sizes. Indivisible software components are called units and testing of such a unit is simply called Unit Testing [20]. In C++, methods and simple classes can be considered units. Unit testing is often made by programmers and seeks to ensure that newly developed code works as intended [21]. Furthermore, 100% test coverage of code is optimal to lessen debugging time when finding undesired behaviour on higher levels of testing.

Closely related to unit testing is component testing. A component consists of several units and forms a module or program [22]. When each component has been tested in isolation, integration testing is performed to verify that the components can operate correctly together. Integration tests are done on systems or subsystems and aim to exhaust the interfaces between components [23].

3.3.2 Regression Testing

Typically, unit, component and integration tests are written and run when new functionality has been added. As important as it is to verify new behaviour, it is equally important to verify that old behavior is not broken by introduced code. This is called Regression Testing [24]. Generally, regression testing consists of unit tests, component tests and integration tests, and in its simplest form, it is the collection of all tests written in a project [25]. However, running all previously written tests becomes time consuming. For example, running all tests in a large project might take up to 7 weeks [26].

There are several techniques that try to increase the efficiency of regression testing. *Regression Test Selection* is a group of techniques which strive to select test suites for only the affected parts [27]. *Test Suite Minimisation* techniques have a slightly different approach and try to have complete coverage with a minimal number of tests [28]. A third category is *Test Case Prioritisation*, which ultimately runs all tests but runs prioritised tests first [29].

3.3.3 System Testing

The purpose of system testing is to validate that a system fulfils the given requirement specification and to provide a quality assurance [30]. This quality assurance consists of the following:

Load Testing	Measures the performance under normal circumstances [31].
Scalability Testing	Measures the performance when a system is deployed in a larger scale [32].
Reliability Testing	Measures how fault tolerant a system is [33].
Stress Testing	Measures how a system performs under extreme conditions [34].
Interoperability Testing	Measures how well a system can operate with other systems [35].
Localisation Testing	Measures how well a system can operate with different languages [36].

These tests must be performed by an unbiased party and, therefore, are often performed by independent test teams [37]. This is important to avoid compromising the integrity of the tests if an conflict of interest arises.

System testing can put high demand on resources, especially during stress testing and load testing of large scale web applications. As stated in Section 1.4, clouds are used for testing applications and it is often system testing that is performed. The massive computational potential of clouds is able to simulate for example a high number of users to load test a web application [38].

3.3.4 Acceptance Testing

These tests are carried out by a customers themselves or hired representatives of a customer and are performed to map the capabilities of a system and if it can satisfy the current needs of the customer [39]. However, they are also executed in order for the customer to determine of they are ready for this system. For instance, the customer might need a specific type of infrastructure and users of the system might need training [40].

4

Testing Environment

To be able to use the testing environment explained in this thesis in several different projects (with different communications), one might produce a highly modular software, with modules coded specifically for each project. This would yield easy to use software, as long as the client-server communication looks the same. If some requirements differ, one would need to either modify or create a new module, and depending on how large the differences are it can be very troublesome to get familiar with the code again and time consuming to implement new and/or different functionality. Section 4.1 addresses the modularity theory of the testing environment in more detail.

To avoid the trouble of implementing new and/or different functionality, one can try to generalise. That way one might use the software in different environments without creating huge amounts of new modules. The price to pay, however, is to provide more information to the system in order to configure it the right way. The testing environment in this Master's Thesis aims to be general in order to minimise the effort to port it to other projects. Furthermore, with the high need of configuration, the goal was to encourage understanding and inspection of client-server communication which could reveal weaknesses in the specification. The details of the configurability theory are found in Section 4.2.

Furthermore, Section 4.3 presents the testing procedures currently adopted in the Volvo project at Delphi and the integration of the new testing environment into the Delphi project is discussed in Section 4.4, along with how the new environment is hoped to exceed the old in efficiency in Section 4.5.

4.1 Modularity

The design of the Test Environment must be fairly easy to modify in order to port it to other projects. To achieve this one can try to categorise and collect project specific functions into modules which can be swapped when the Test Environment is ported [41].

4.1.1 Identifying components/entities of a testing environment

It is a necessity to control an entity when testing it, in order to get the entity to actually execute something. This controller could be either a programmed unit, for instance, a driver in component testing, or a human doing some manual system testing.

Moreover, the entity under test may have dependencies to other entities, such as other classes, components or services, and it is therefore necessary to create an environment where the entity can execute. In more complex tests there may be a need to configure or control the environment, for example, if the behaviour of process A, during communication with process B, is to be tested, a simulation of process B must be configured.

There must also be a controller of some kind to excite tests and collect the results. In total, five separate entities are identified:

- **Test Target** - The entity to be tested. Could be either an unit, a component or a complete system.
- **Target Control** - The entity which can excite some functions of the Test Target and record the results.
- **Environment Simulation** - The entity that simulates the cloud/back-end service and communicates with the test target.
- **Environment Control** - Configures the Environment Simulation and then validates the communication after the test is run.
- **Test Control** - The entity which synchronises actions of the Target Control and configurations of the Cloud Simulation in order to run tests.

A schema of a generic test environment can be seen in Figure 4.1. The steps in the process of performing a test in this environment are the following:

1. When the test control is directed to start testing, it provides the Environment Control with configuration instructions suitable to the pending test.
2. The Environment Control initiates the Environment Simulation with given configurations.
3. The Test Control provides instructions to the Target Control, stating what the Test Target is expected to do.
4. The Test Control excites the Test Target according to the provided instructions.

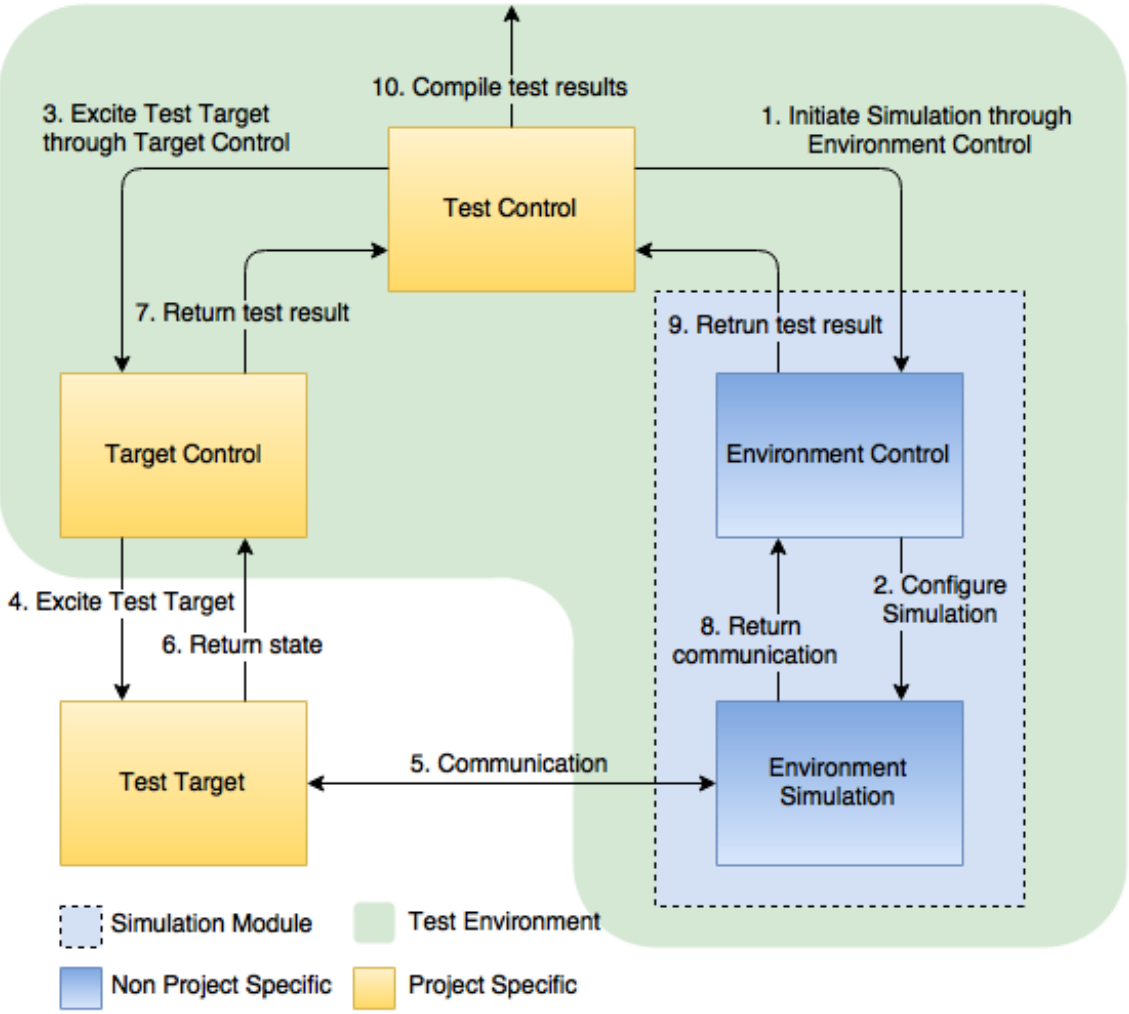

Figure 4.1: Schema of a Test Target and the components in a test environment.

5. The Test Target interacts with the Environment Simulation.
6. The Target Control reads the state of the Test Target.
7. The Target Control returns the relevant data to the Test Control.
8. The Environment Simulation returns test data to the Environment Control.
9. The Environment Control returns the test result to the Test Control.
10. The Test Control examines the results, evaluates the test and presents a complete test result.

Now, imagine that it is desired to test step 2 and 3 of Figure 1.1 using this modular testing environment. Then the process above translates to the following:

1. When the Test Control is directed to start testing, it provides the Environment Control with instructions to reply with a specific list of time slots when the

HU sends a request of available time slots, according to Figure 1.1. Here, the HU wants to have a secure connection, so the Test Control also provides certificate and key information (according to Section 4.2.5).

2. The Environment Control initiates the Environment Simulation with the given instructions. Since security information was provided in the previous step, the Environment Control will initiate the Environment Simulation with, for instance, SSL.

3. The Test Control provides instructions to the Target Control, stating that the Test Target (HU, in this case) should send a time slot list request.

4. The Test Control forces the HU to send a time slot list request.

5. The HU sends a time slot list request. The Environment Simulation analyses its instructions, confirms that it got a time slot list request and sends a list of time slots in response.

6. The communication is finished and Target Control reads the state of the Test Target. It can check, for example, that the HU accepted the response and did not crash.

7. The Target Control returns the result to the Test Control.

8. The Environment Simulation returns the test data to the Environment Control.

9. The Environment Control checks that the Environment Simulation accepted the communication and returns the test result to the Test Control.

10. The Test Control examines the results from the Target Control and the Environment Control. If both are OK with the communication, the test was successful.

4.1.2 General and Project Specific Modules

If the Environment Control and Simulation are configurable enough and able to not contain any project specific functionality, they do not need to be altered when porting the Test Environment. It is therefore desirable to create a Simulation Module that is able to be configured to simulate any HTTP communication.

Target Control is highly dependable on the Test Target and is therefore an entity that must be changed or swapped when porting.

Test Control may be generalised if the Target Control conforms to a specified API and if one is able to configure the Test Control in a manner that specifies what and how tests should be run. However, Target Control can be contained in a single Bash

Shell script, and therefore, it is not a large component and can be written for each new project.

4.2 Configurability

This section details the many different configuration options that were implemented for the testing server in order to make it diverse and easily portable to different projects.

4.2.1 Main Specification of Behaviour

The server needs to have a set of rules to match against incoming requests and create appropriate responses to send to the client. In order to create configurability, this rule set was chosen to consist of a stand-alone file, written in some markup language.

Research showed that the two most popular markup languages applicable to this task were JSON and XML. In Listings 4.1 and 4.2, an example of a simple phone book can be seen in XML and JSON, respectively. It demonstrates that XML is more verbose, taking longer for a human to both write and read. A drawback stemming from JSON's clearer notation is that only a handful of data types are supported.

Considering that the only data that will be stored is HTTP messages, which can be stored as simple strings, JSON seemed like an appropriate choice. Furthermore, the simplicity of JSON offers light weight and self-contained serialisers and deserialisers, which is desirable since that means that the binary, executable, file of the testing environment has no outgoing dependencies and can be run anywhere.

```
1<Contacts>
2    <Contact>
3        <FirstName>John</FirstName>
4        <LastName>Smith</LastName>
5        <PhoneNumber>(020)-6378-2583</PhoneNumber>
6    </Contact>
7    <Contact>
8        <FirstName>Jane</FirstName>
9        <LastName>Lewis</LastName>
10       <PhoneNumber>(029)-5423-8634</PhoneNumber>
11   </Contact>
12   <Contact>
13       <FirstName>Robert</FirstName>
14       <LastName>Jones</LastName>
15       <PhoneNumber>(0131)-368-8426</PhoneNumber>
```

```
16      </Contact>
17</Contacts>
```

Listing 4.1: An example of a phone book markup in XML.

```
1 "Contacts" : [
2     {
3         "FirstName" : "John",
4         "LastName" : "Smith",
5         "PhoneNumber" : "(020)-6378-2583"
6     },
7     {
8         "FirstName" : "Jane",
9         "LastName" : "Lewis",
10        "PhoneNumber" : "(029)-5423-8634"
11    },
12    {
13        "FirstName" : "Robert",
14        "LastName" : "Jones",
15        "PhoneNumber" : "(0131)-368-8426"
16    }
17 ]
```

Listing 4.2: An example of a phone book markup in JSON.

A visualisation of the server workflow between an incoming request to an outgoing response can be seen in Figure 4.2. The system allows the user to easily modify, remove or add rules to test the specific needs of their service.

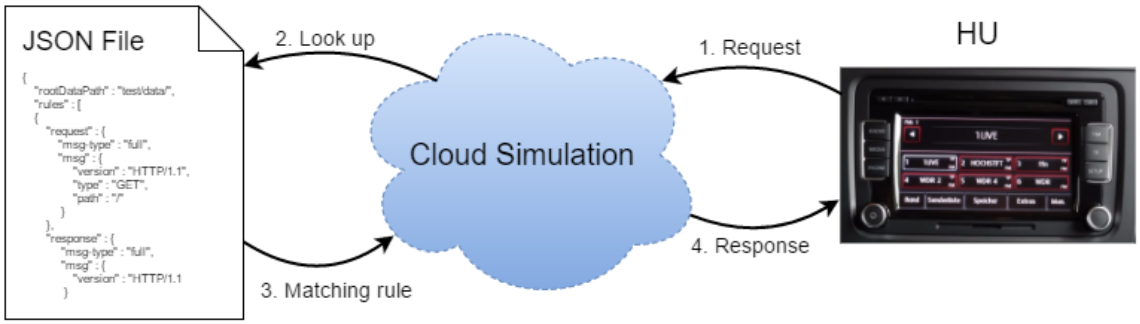

Figure 4.2: Schema of the flow from an incoming request to an outgoing response.

4.2.2 Rule Specification

As mentioned in 4.2.1, the server must have a set of rules to match against in order to create an appropriate response to incoming requests. The rules should be able to be added, modified or removed by the user to configure the behaviour of the server.

The server should designed to only expect requests that can be matched to at least one of the rules provided in the rule set. If no matching rule is found, the server should view it as incorrect behaviour from the client. This produces a very strict predetermined behaviour from the server and the developer has complete control of and responsibility for what is considered correct behaviour of the client.

However, such strict behaviour can be hard to predict when values of fields in the header or data are produced at run time, for example dates and process IDs. Even so, the format of such values can often be predicted, and using regular expressions when configuring a rule in the rule set can accommodate this. Additionally, with regular expressions the strictness of rules can be alleviated by matching requests by pattern instead of exact values.

Furthermore, the user can choose to put the body of a request or response in a separate file, to save space in the JSON file and ease modification of server behaviour. The user provides the path to the document containing the body instead of typing out the actual data body in the JSON file.

4.2.3 Stateless and Stateful Environment

Software architecture can be either stateless or stateful and thus, in order to create a testing environment that can be used with any HTTP service, both a stateful and a stateless environment should be implemented.

In the case of the stateless environment, no attention should be paid to the order of incoming requests and a specific request should always trigger the same response. If no matching rule can be found, an error is reported.

In Figure 4.3, an example of a stateless workflow can be seen. This is useful, for instance, when wanting to test a website without having to press buttons and navigate through it in a certain order.

Sometimes, a stateful environment might be preferable. It is stricter in the sense that the environment will expect everything to happen in a certain order, but less strict in the sense that different responses can be given to the same request, depending on the current state. An error is reported if the client sends a request that the server did not expect.

An example of a stateful workflow can be seen in Figure 4.4. This could be useful when order is important, for example, when a HTTP application should send initial requests upon startup.

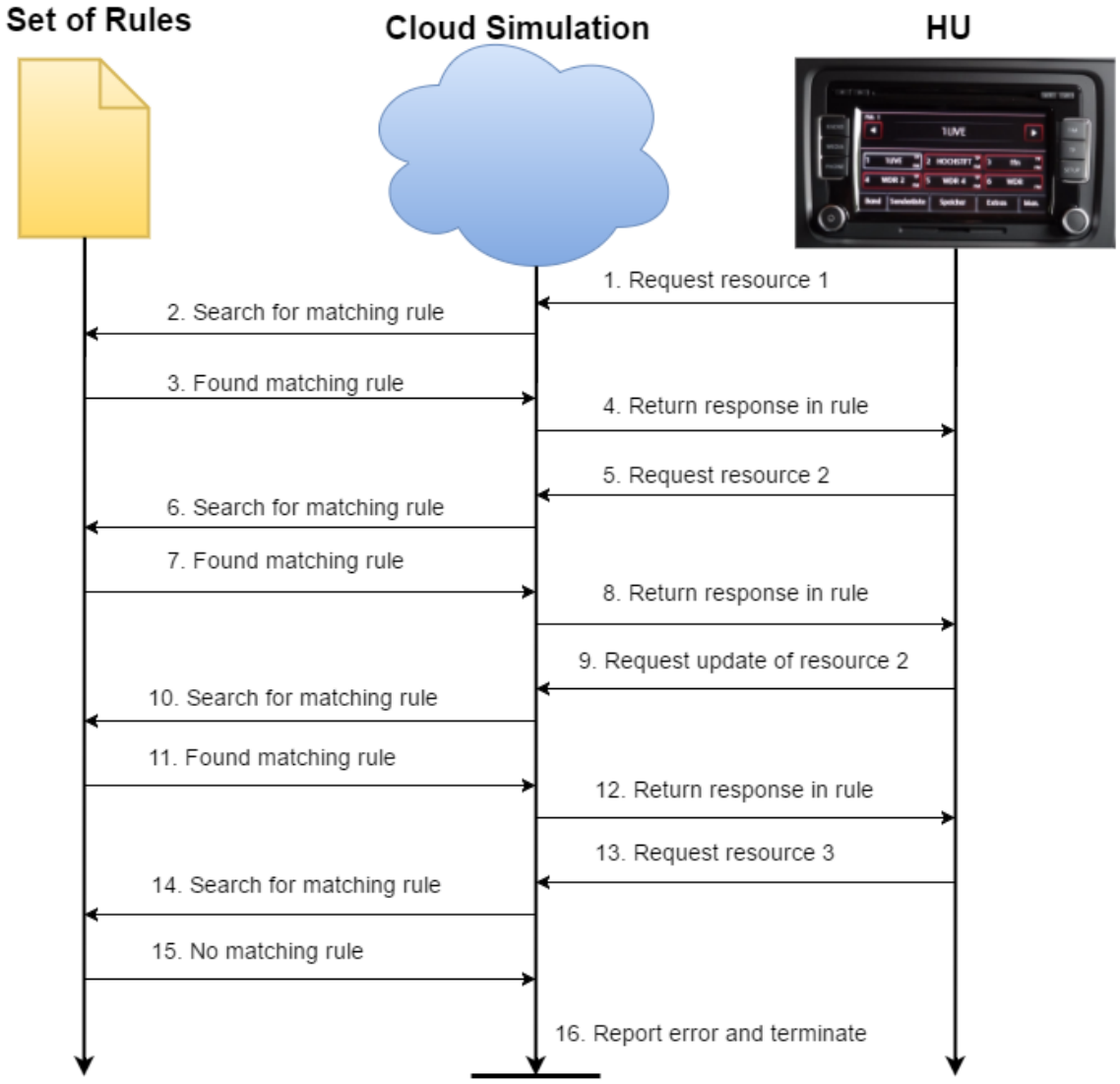

Figure 4.3: An example of a stateless workflow.

4.2.4 Additional Actions

Occasionally, the functionality or configuration of the application to be tested might make it close the current connection (possibly to open a new one) or wait a finite amount of time (that could cause a timeout in the cloud).

Creating a general testing environment should involve an implementation handling this type of special action. It should be able to expect that the client closes the connection and/or opens another connection. It should also be able to wait a certain amount of time before expecting anything. Furthermore, a timeout in the testing sever should be possible to avoid by specifying the desired time before timeout upon startup.

Another functionality that should be supported by the server is the possibility to re-

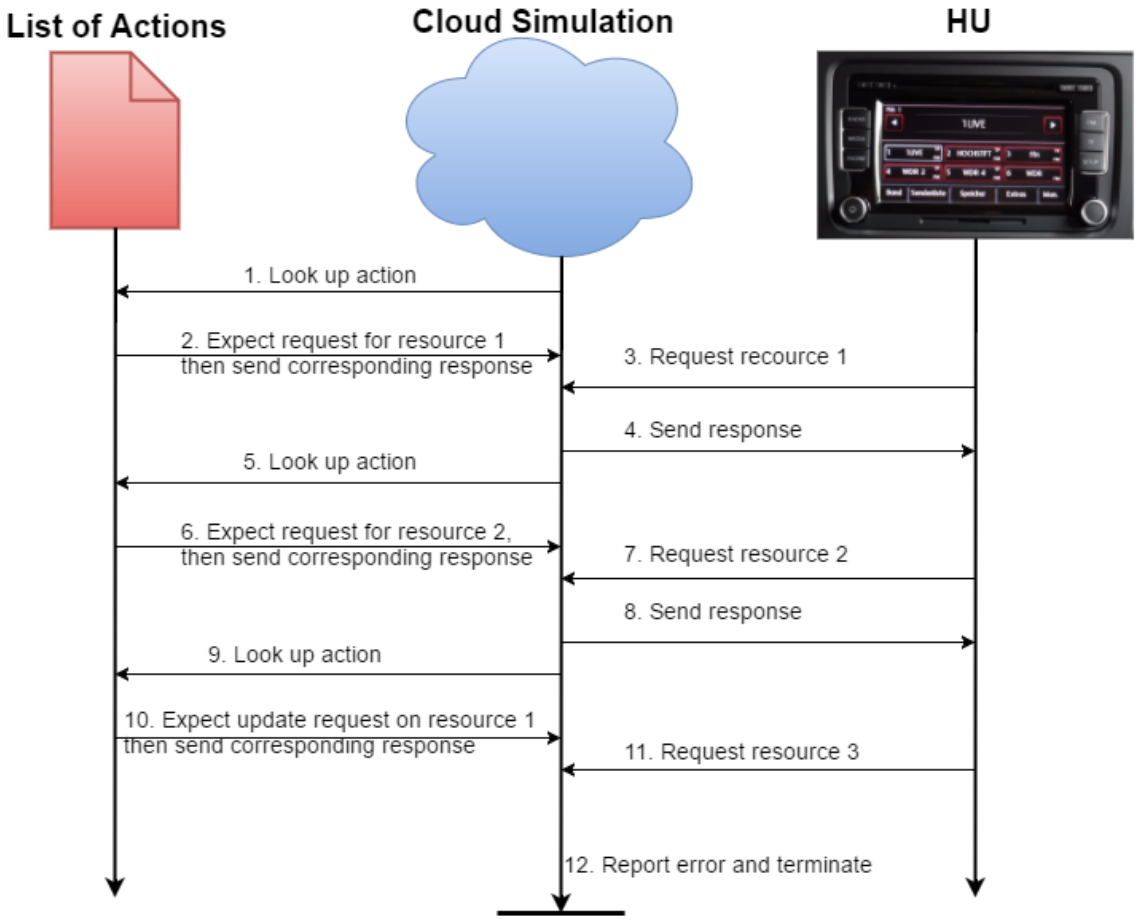

Figure 4.4: An example of a stateful workflow.

ceive a message without creating a response, in contrast to the regular rules detailed in Sections 4.2.1 and 4.2.2. Likewise, the server should be able to send a message without expecting a response from the client.

Furthermore, a HTTP application might sometimes expect the server to close the connection, so functionality for this should also be implemented.

In order to uphold consistency, and make configuration easy, this type of special action should be able to be specified analogously to regular rules, in the JSON document described in Section 4.2.1.

4.2.5 Optional Communication Through SSL/TLS

Most services incorporate security measures in their communications with the cloud. This means that the cloud needs to handle authentication using certificates and keys.

The standard at Delphi is to use the toolkit “openSSL”. It is a popular option and provides services using both SSL (Secure Socket Layer) and TLS (Transport Layer

Security). Furthermore, it is easily incorporated with C++, and thus, seemed like a clear choice.

In order to be configurable and generic, the cloud should also be able to handle a connection without security, to support services that do not want or need security, or simply have yet to implement it.

4.2.6 Configuring Different Testing Types and Methods

Since the testing environment requires execution of the client, it falls under the category of Dynamic Testing, as opposed to Static Testing mentioned in Section 3.2.1. Using the configurable rules described in Section 4.2.1, both Black Box and White Box Testing (Section 3.2.2), as well as Positive and Negative Testing (Section 3.2.3), can be performed.

Black Box Testing can be performed by writing many (ideally all possible) rules in the rule set and looping through them. In White Box Testing, fewer rules are needed, but instead it requires knowledge of the client code. The goal would be to write very diverse rules in the rule set, in order to exhaust the client code.

By writing rules with a valid request format (that is, a request that the client could actually send) and a corresponding valid response, Positive Testing is performed. If instead Negative Testing is desired, only the response needs to be changed into something that the client might not be expecting (that is, some invalid response).

In addition to these different testing types, the environment has support for a variety of testing methods (see Section 3.3). By performing tests using different Target Controls, the target can trigger units, components or the whole system, and thus performing Unit, Component and System tests (explained in Section 3.3.1). Since Regression Testing, Section 3.3.2, is a collection of the mentioned testing methods, it can be performed as well.

This way, the configurability of the testing environment enables diverse testing possibilities, to satisfy different tester and client needs.

4.3 Current Testing in Volvo Project

Currently, unit tests are performed automatically in the Gerrit/Jenkins system, described in Appendix A. However, the tests contain external dependencies and can therefore not be run on target, but are instead run directly on a Test Slave. They use a cloud provided by Volvo.

Component tests are currently only performed manually on the developer PC and,

like unit tests, they are executed without a target. Additionally, these tests are performed using an existing test environment to simulate the cloud, which enables negative testing. This environment, however, is outdated and cannot be used in the Gerrit/Jenkins system or on a target. Furthermore, it lacks modularity and generality, making it inconvenient to port to any other project.

Integration testing can be performed on a simulation of the target, for example, a NUC (Next Unit of Computing). An image of the software to be tested is installed on the NUC, and the tester can then use a graphical interface to manually test different parts of the software.

4.4 Integration with Delphi

The new testing environment will need to be integrated with the current system at Delphi. The test should be runnable in a manual context, explained in Section 4.4.1, as well as in an automatic context, explained in Section 4.4.2.

4.4.1 Manual Testing

The new testing environment should be run directly on a target, connected to the developer PC. Figure 4.5 shows the theoretical workflow that should be performed manually by the developer.

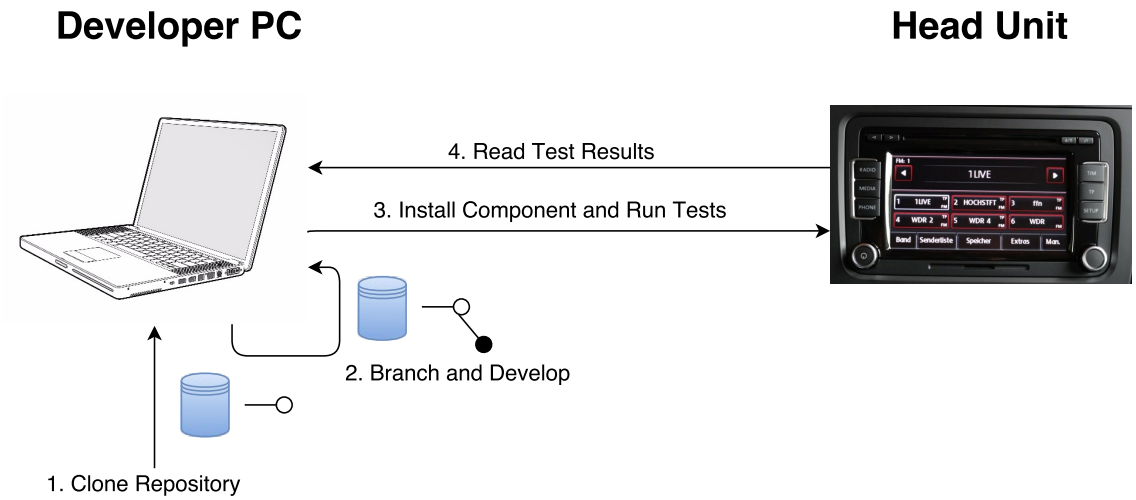

Figure 4.5: Schema of the manual testing procedure.

4.4.2 Automatic Testing

Delphi already has a system for automatic testing that can be studied closely in Appendix A. Figure 4.6 shows how the testing environment of this thesis should be incorporated into the existing routine.

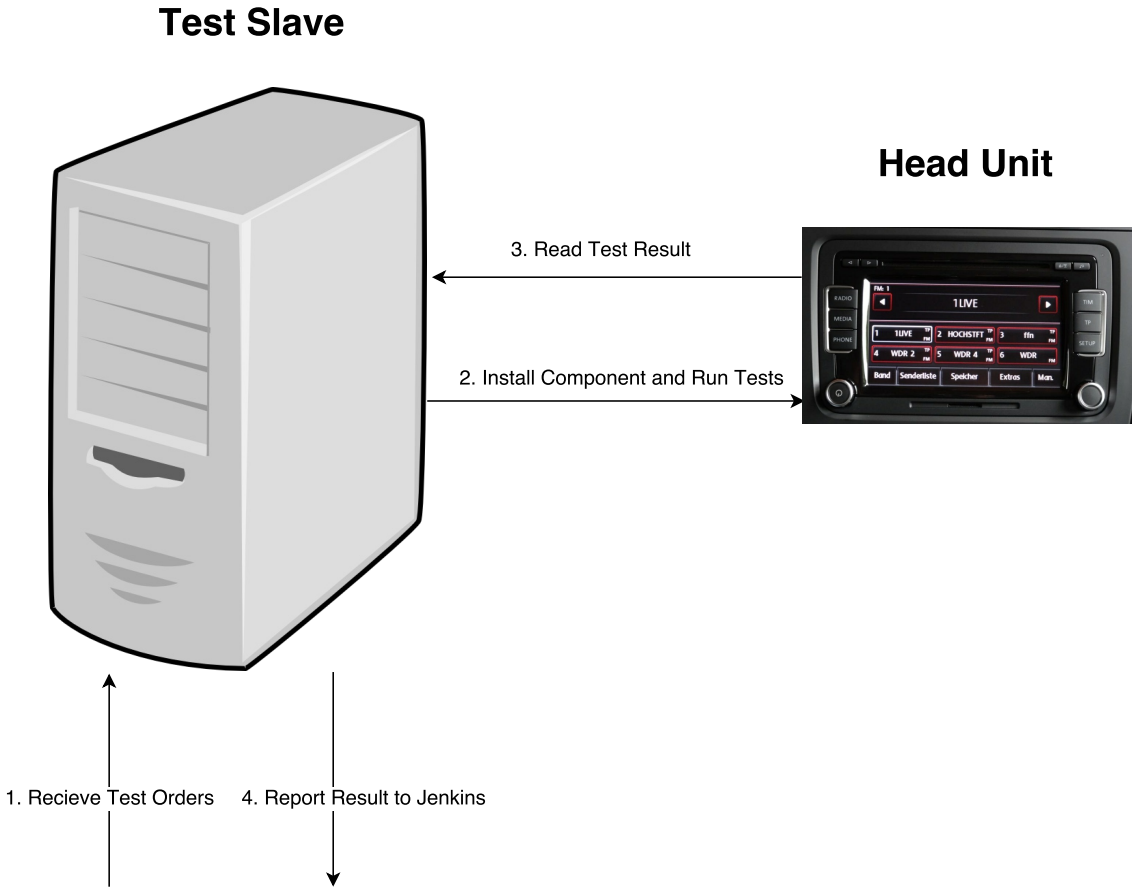

Figure 4.6: Schema of the automatic testing procedure.

The procedure is analogous to the manual testing, shown in Figure 4.5, except that everything is handled automatically when a developer tries to push software changes to Git.

4.5 Exceeding Current Testing Efficiency

This section details how the new testing environment, explained in Section 4.4, in theory, should outperform the old one in terms of the factors of efficiency listed under Section 2.3.

4.5.1 Similarity to Native Environment

As mentioned in Section 4.3, no testing is performed on an actual target. However, integration testing can be performed manually on a simulation of the target , which falls into category 2 according to the specifications in Section 2.3.1. On the other hand, all other testing types are performed on a environment different from the target environment, in other words, category 1 in the specification.

The testing environment in this thesis should be performed directly on a target, linked to either a test slave or developer PC. This means that it falls into category 3 of the above mentioned specification for all testing types.

A higher category number means that the testing is performed in a more similar context, and thus, the testing procedure detailed in this thesis should outrank the current one.

4.5.2 Code and Input Coverage

The current testing at Delphi only supports positive testing, except in the case of component testing that could possibly be written to perform negative tests. However, no such tests are performed today. Furthermore, unit testing is the only testing method that can be performed automatically.

The new testing environment should allow any dynamic tests, both positive and negative. Additionally, all tests should be runnable in both an automatic and manual fashion.

This means that, according to the specifications in 2.3.2, the new testing should outrank the old in this category.

4.5.3 Time Consumption

As mentioned in Section 4.3, the only tests that currently can be performed automatically are the unit tests, although all tests can be run manually in different manners.

The new testing environment should support automatic and manual tests for any testing method.

According to the specifications in 2.3.3, providing both manual and automatic testing gives an advantage in this category of efficiency, meaning that the new testing environment is favorable over the old. However, configuring the new environment and writing tests might prove to consume more time than with the old method.

5

Results and Discussion

The results of this project will be explained in this chapter, along with accompanying discussions. The outcome of the manual and automatic testing goals are detailed in Section 5.1. The modularity aspect of the result is described in relation to the previous theory on the subject (Section 4.1) in Section 5.2. In Section 5.3, the configuration functionalities are portrayed, also in relation to previous theory (Section 4.2).

5.1 Using the Testing Environment

This section describes an overview of how the resulting testing environment is used. Section 5.1.1 describes the target, while Sections 5.1.2 and 5.1.3 details how the testing could be integrated at Delphi.

5.1.1 NUC Instead of Target

The Volvo project at Delphi was still early in development, and no actual HU was available for testing yet. Instead, a NUC that simulated the hardware and software environment of a HU was used. However, since a NUC and a HU behave the same from the testing environments point of view, there should be no issue simply running the tests directly on HU, when one is available.

5.1.2 Manual Testing

The manual test suite was implemented as a script that runs the implemented tests and returns a green "**OK**" for each successful test. If the test was run but the target did not behave according to specifications, an orange "**NOK**" (Not OK) is returned. Furthermore, in case of total failure (the target crashed, the JSON specification is invalid etc.), a red "**FATAL ERROR**" is returned. In both the case of NOK and FATAL ERROR, an error message is also returned, explaining what went wrong.

5.1.3 Automatic Testing

Due to security issues and absent key personnel, the integration with the automated testing service (Jenkins) at Delphi failed. However, the only task of this service is to run scripts, something that can easily be done locally on a PC as well.

Considering this, a script was written to simulate the automated testing service. Since, there is no difference between this and the real one from the testing environment's point of view, there should be no problem integrating the new testing environment into the automated testing service.

5.2 Modularity

The new testing environment consists of three components:

- Target Control
- Simulation Module
- Test Control

The relationship between these components can be seen in Figure 5.1. Comparing this with the theoretical modularity in Figure 4.1, one will notice that they are similar, but not completely identical.

Target Control and Test Control work the same as shown in Figure 4.1 and, as explained in Section 4.1.2, they are specific for the Volvo Project. In the proof of concept, the Target is a component and, therefore, Target Control is a driver component. Test Control is a simple bash script, which starts the simulation with a configuration file, asks Target Control to excite the Target and then collect test results from Target Control and the Simulation Module.

The Simulation Module is designed to be reusable by configuring it with configuration files. It is divided into two sub-modules: an Environment Control and an Environment Simulation, much like in Figure 4.1. However, unlike in that figure, the Environment Simulation is divided into three "subsub-modules". There is one type of module for each of the two modes (more about simulation modes in Section 5.3) that the program can be executed in. These two modules are the ones that handle all the communication between the simulation and the client, and since the two modes are quite different it was decided to keep them apart in separate "sub-modules". The third "subsub-module" consists of a library containing all the classes used in the two modes, as well as the Environment Control. This adds even more modularity and mitigates code repitition.

This quite high level of modularity should enable easy replacement of modules,

should the need arise. For example, if OpenSSL, or anything regarding the HTTP(S) protocol, should need to be replaced, then only the Stateful and Stateless modules would need replacement or configuration.

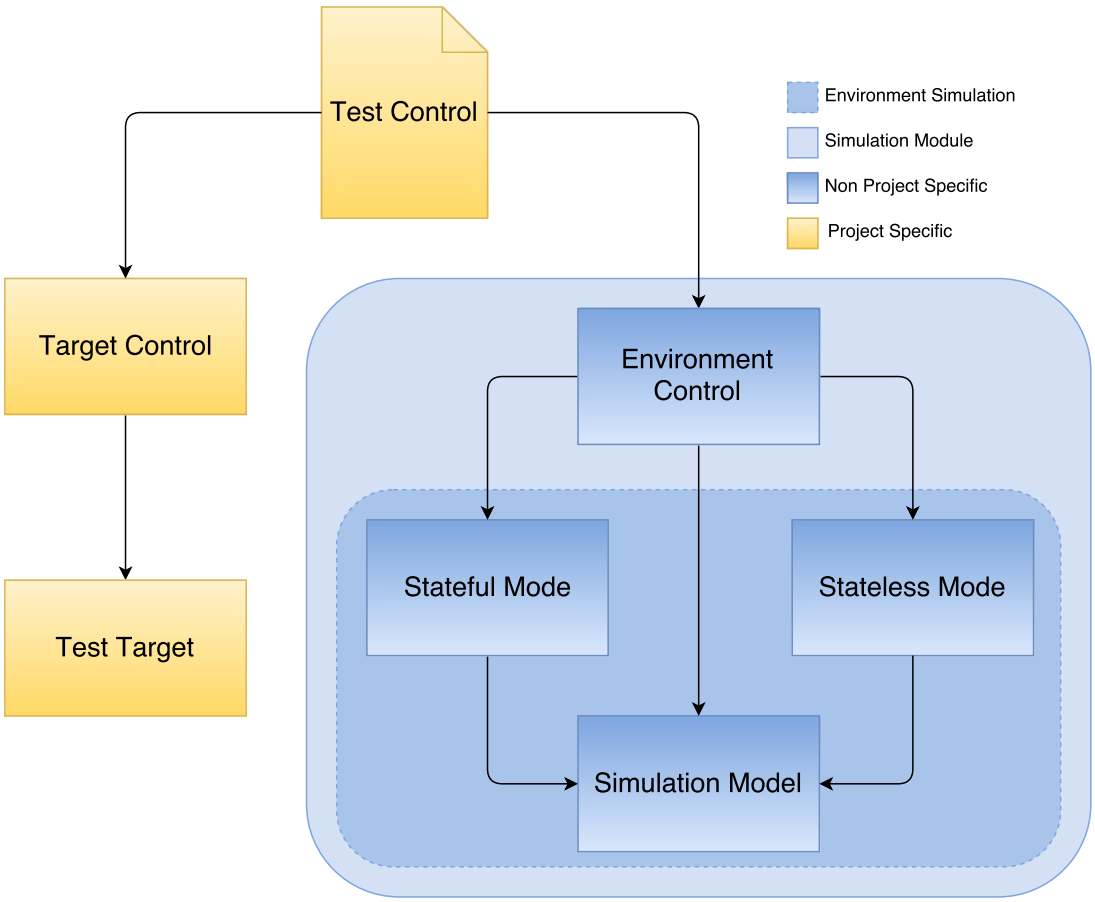

Figure 5.1: Test environment call hierarchy.

5.3 Configurability

This begins by describing the results of the Stateless and Stateful modes portrayed in Section 4.2.3. This is done in Sections 5.3.1 and 5.3.2. The outcome of the option to open a secure connection (detailed in Section 4.2.5), is explained in Section 5.3.3.

5.3.1 Stateless Mode

In Stateless mode, the simulation receives, in the provided configuration file, a list of rules. These rules describe a request and a response. When the simulation receives a request, it searches the rules to match a request, and if one is found, it returns its corresponding response.

This mode is useful, for example, when a developer has implemented several simple functionalities and want to test them in any order using the same cloud, for simplicity. Figure 5.2 shows such a case.

However, the restriction with the stateless mode is that is unable to have different responses to the same request. This is demonstrated in Figure 5.3, using a car service booking example. Since the requests in steps 1 and 9 of the figure is the same, it maps to the same response, even though the HU has actually already booked one of the time slots. Clearly, this type of workflow would be more suited to a stateful environment.

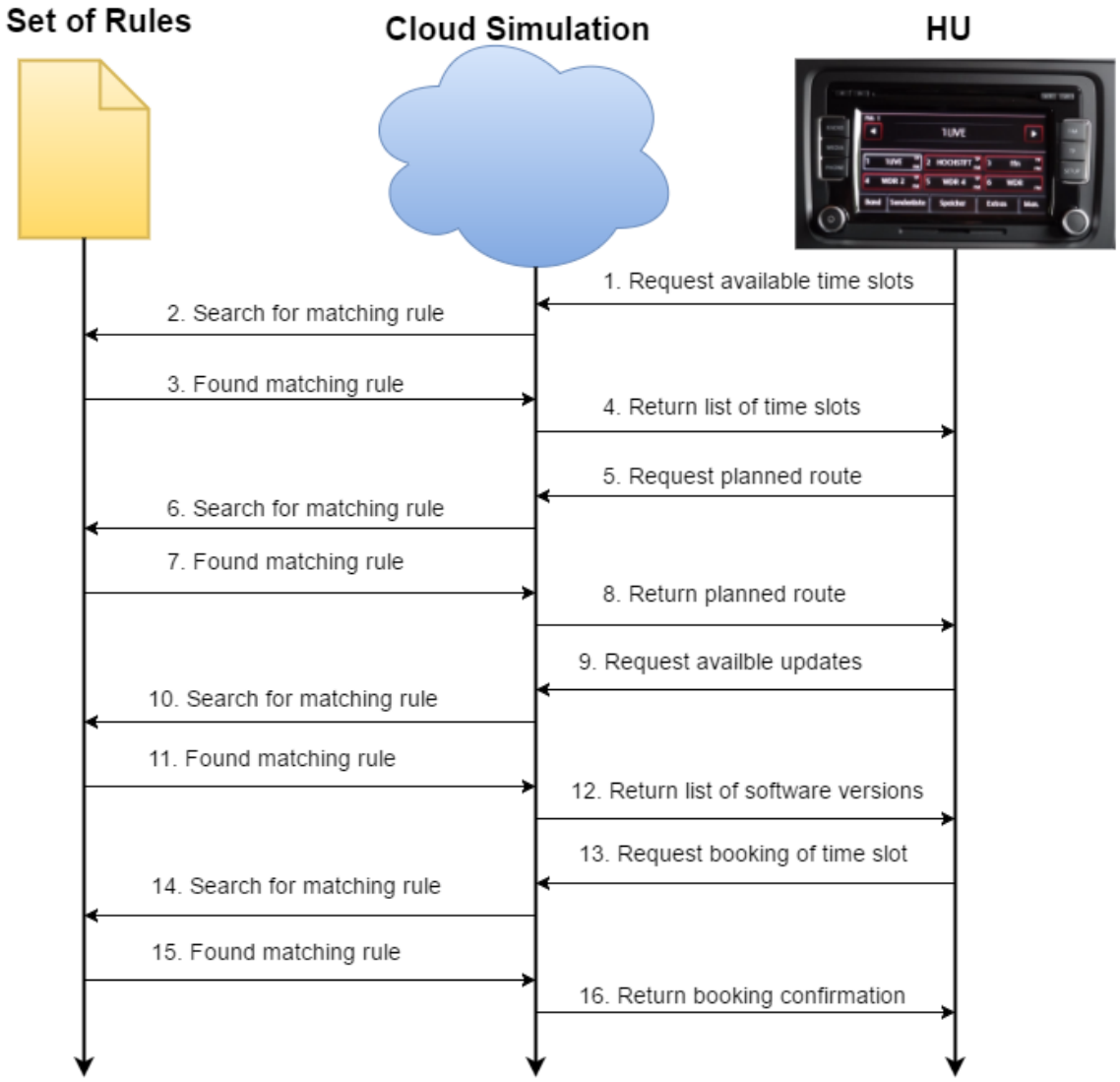

Figure 5.2: An example of a stateless use case, showing the usefulness of the mode.

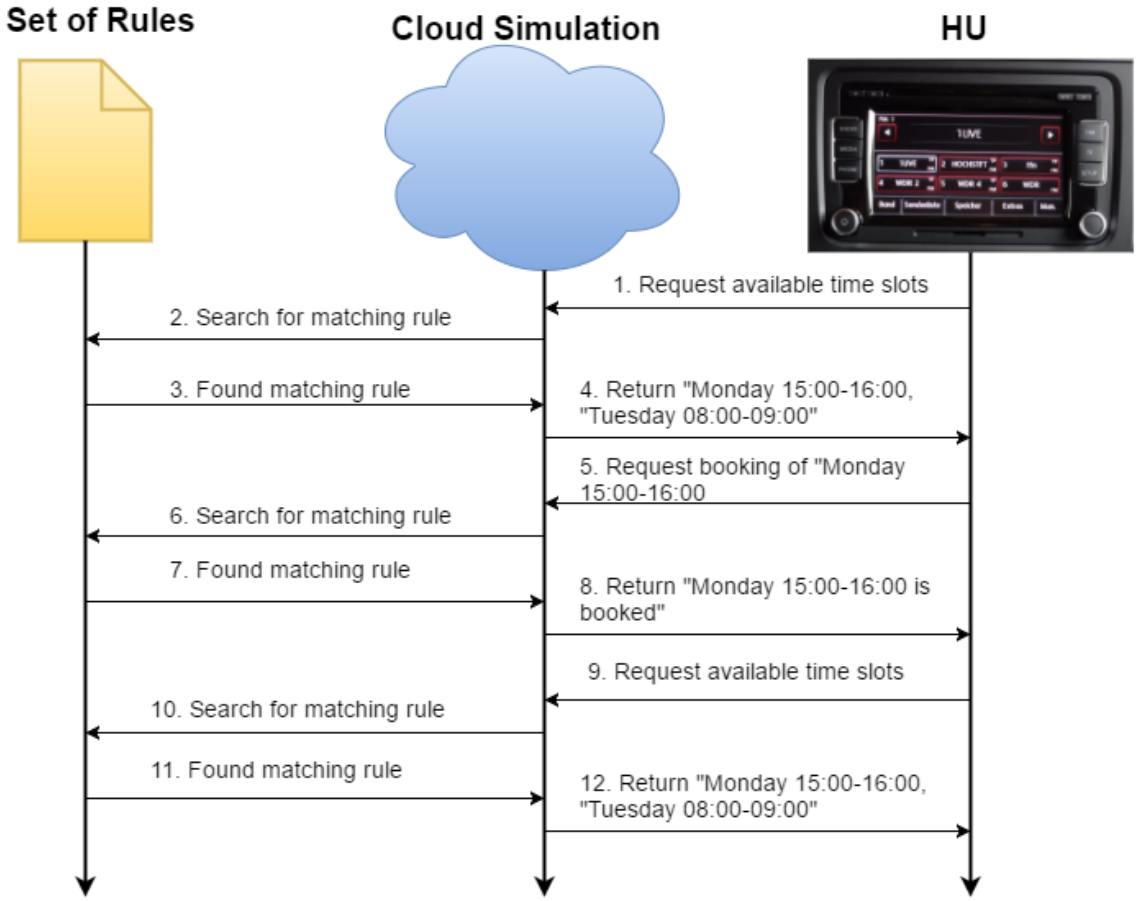

Figure 5.3: An example of a stateless use case, showing its restriction of write operations.

5.3.1.1 Rules Set

A set of rules describing a simple web server can be seen in Listing B.1. These requests and responses can be of three types, see Table 5.1.

Listing 5.1 shows an example of a message of type 'full'. It is strict and contains the whole message. If the data to be sent in the message becomes large, the readability of the file is compromised. To accommodate this, the message type 'dataRead' was created. This type takes a path to a data file rather than the data itself. This also provides the possibility to easily use the same data in multiple messages. An example of a 'dataRead' message can be seen in Listing 5.2. This is the same message as in Listing 5.1, except that the body is contained in a file called "start.html". This demonstrates the enhanced readability of this type of message. There is also a 'regex' type. The type uses POSIX Extended Regular Expression [42]. An example of a regex type can be seen in Listing 5.3.

Moreover, another functionality is that if the fields 'Date' and 'Content-Length' are absent, the Simulation will add them dynamically. This is because 'Date' should be

Table 5.1: The different types of messages in a rule.

Type	Description
full	This message is fully contained by the JSON object
dataRead	The header of this message is contained by the JSON object, but the data must be read from the path given in the JSON bject
regex	This message has some part(s) of it described with regular expressions. It is not meant to be sent, only to be matched against
raw	Sometimes, for instance in negative testing, it is useful to be able to describe raw messages. In other words, create the possibility to send messages that doesn't conform to HTTP protocol.

the actual date when the message is sent and 'Content-Length' might be tricky to calculate.

Listing 5.1: An example of a message of the type 'full'

```
1 "response" : {
2     "msg-type" : "full",
3     "msg"      : {
4         "version" : "HTTP/1.1",
5         "code"    : "200",
6         "status"  : "OK",
7         "fields"  : {
8             "Content-Type"     : "text/html; charset=UTF-8",
9             "Content-Encoding" : "UTF-8"
10        },
11        "data"    : "<html><head><title>This is \"/post\".</
              title></head><body>Hello World, This is \"/post
              \". <a href=\"http://127.0.0.1:30000\">Return to
              start</a></body></html>"
12    }
13 }
```

Listing 5.2: An example of a message of the type 'dataRead'

```
1 "response" : {
2     "msg-type" : "dataRead",
3     "msg"      : {
4         "version" : "HTTP/1.1",
5         "code"    : "200",
6         "status"  : "OK",
7         "fields"  : {
```

```
8               "Content-Type"      : "text/html; charset=UTF-8",
9               "Content-Encoding" : "UTF-8"
10          },
11      "dataPath"      : "start.html"
12      }
13}
```

Listing 5.3: An example of a message of the type 'regex'

```
1"request": {
2       "msg-type": "regex",
3       "msg": {
4           "type": "GET",
5           "version": "HTTP/1.1",
6           "path": "/",
7           "fields": {
8               "Host": "127.0.0.1:30000",
9               "Accept": "([[:alnum:]]|[[:punct:]]|[[:space:]])
                    *",
10              "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
                    alnum:]]|[[:punct:]]|[[:space:]])*"
11          }
12      }
13}
```

5.3.2 Stateful Mode

The Stateful mode has a bit more functionality. In its configuration file, it receives a list of Actions. This list of actions determines how and when the simulation should act, in other words, a scenario. If the simulation cannot act as specified due to the client not doing as expected, it terminates and reports an error.

Figure 5.4 shows the same use case as in Figure 5.3, but used in the stateful mode instead. This demonstrates that the same request can map to different responses, which is useful in this test case. Another use of being able to respond differently to the same request can be seen in Figure 5.5. First, the simulation responds with "Bad Request" when the HU tries to access the available time slots. The HU will retry for the same resource after some internal timer, and this time the simulation will send a positive response.

The stateful also expands the space of negative testing. Figure 5.6 shows a test case using most of the functionality of the stateful mode. See Table 5.2 for the complete list of actions. The test case begins with the simulation sending a request for the resource "/start.html" (1-3), which is typical behaviour of a web browser. Obviously, since the HU is a client, it should simply discard this request (4). Then,

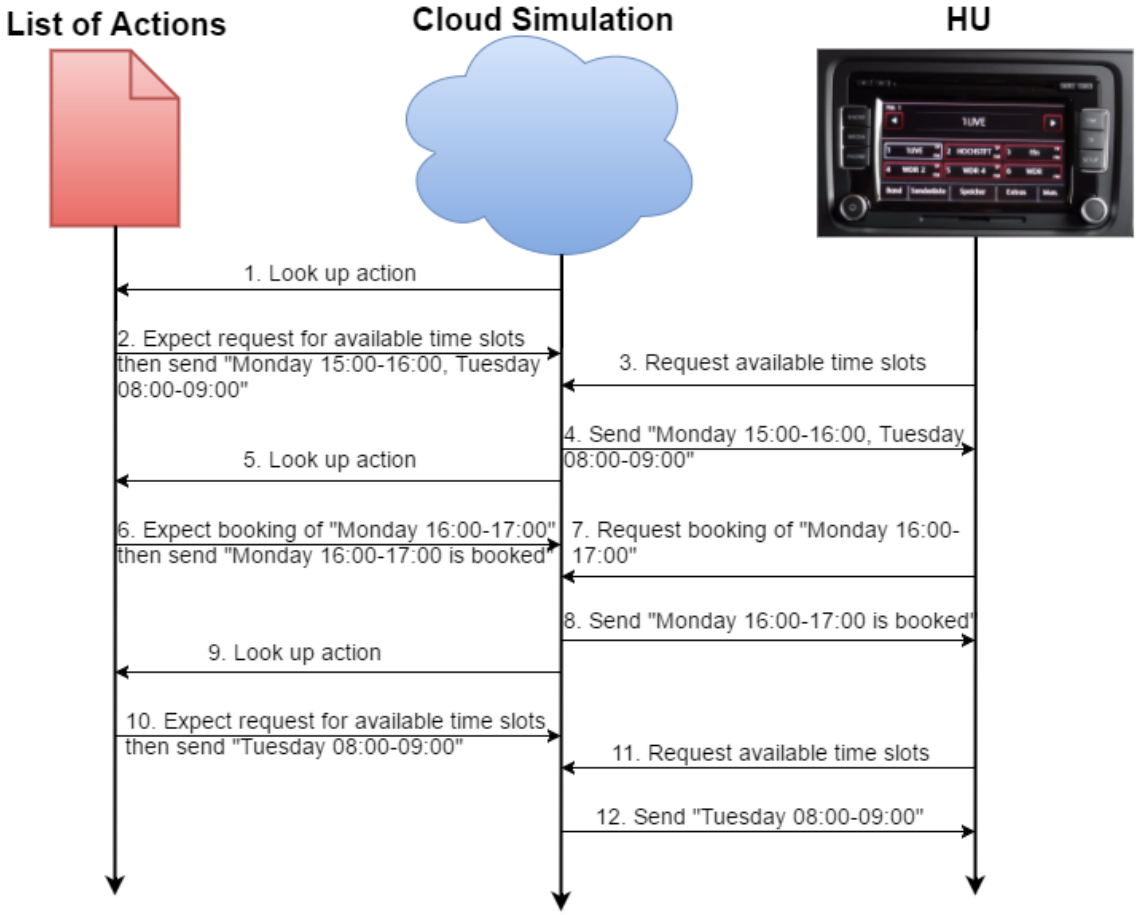

Figure 5.4: A diagram of the same test case as in Figure 5.3 but using the stateful mode which has better support for write operations.

the cloud expects a request for available time slots but it will not respond (5-6). When the HU sends its requests it will wait until it suspects that something went wrong and retry the request (7-8, 11). Now, the cloud responds by deviating from the HTTP-protocol and sends 10 messages only containing the word "spam" (12). The HU should discard also these messages and retry one last time for the available time slots (13). This time the cloud simply closes the connection (14-15, 17) and the HU should decide that the current entry point of the cloud is corrupted and try an other (18).

5.3.2.1 Scenarios

A simple scenario can be constructed by picking and arranging rules from a rule set in the order a user is expected to communicate with the back end. However, a communication entails more than just exchange of data and therefore, the additional actions, previously described in Section 4.2.4, are implemented. They are also presented in Table 5.2. A scenario using the rules for the stateless web server (in Listing B.1), along with some of these additional rules, can be seen in Listing B.2.

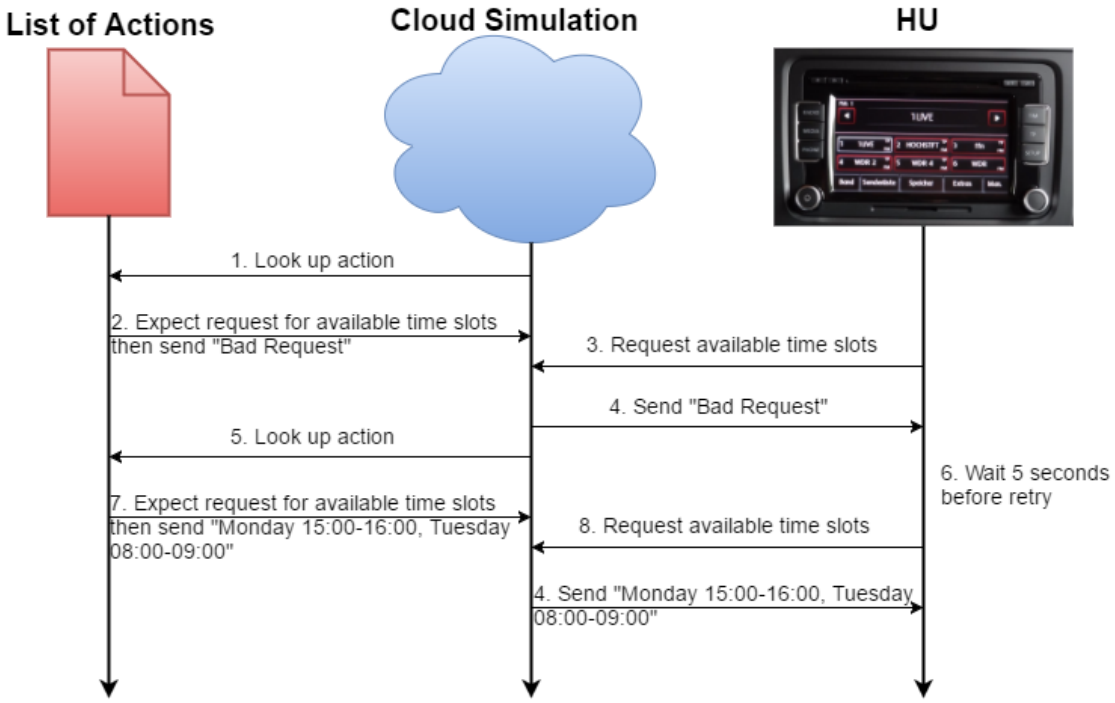

Figure 5.5: A diagram showing other uses of the ability to respond differently on the same request that comes with the stateful mode.

These scenarios can also simulate statefulness. Actions 2-4 in Listing B.2 show a part of the scenario where an example of statefulness can be observed. The user is expected to access a web page, which says "Hello World", and send a POST-request containing their name (dave) to that URL. When the user then request the web page again, it will say "Hello Dave".

Actions 5-11 start to challenge the user with abnormal behaviour from the server. Actions 5-7 simulate a scenario with high latency between the user and the back end service. Examples of negative testing can be seen in actions 8-11, where action 10 returns a malformed response and action 11 spams the client with nonsense.

5.3.3 Choosing secure or unsecure connection

As mentioned in Section 4.2.5, the server should be configurable to use SSL/TLS or simply have an unsecure connection.

The test environment manages this by accepting, but not demanding, the path to a server key and certificate as input upon start up. The server then automatically sets up a secure server, using openSSL and HTTPS, if security information was provided. Otherwise, a standard HTTP server is started.

Note that if a secure server is set up, and the client tries to connect without security information, it will not work, nor will a server configured without certificate and

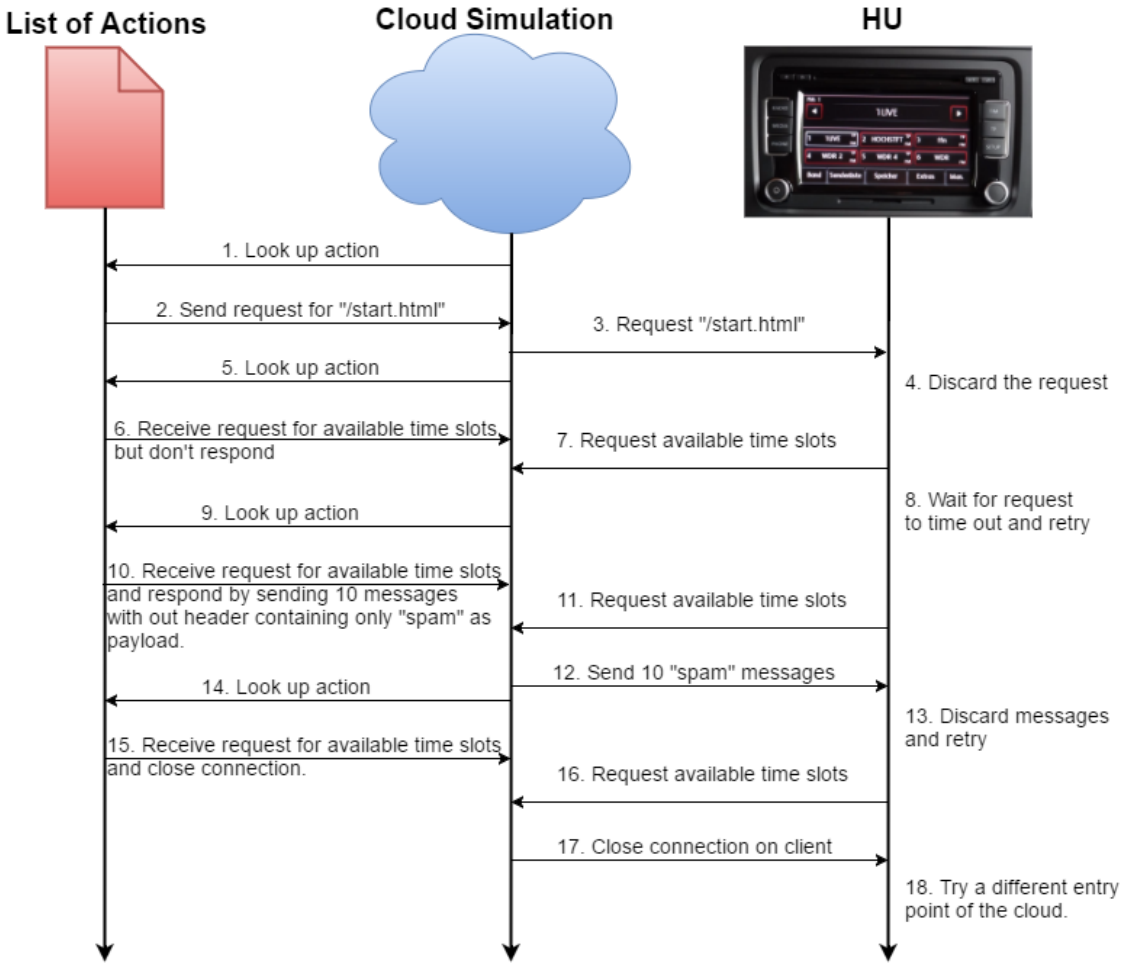

Figure 5.6: A diagram showing a negative test case using the stateful mode.

key accept a client supplying security information. This design choice was made considering that the user of this testing environment should know what type of connection they want their client to have. Thus, strictness in this area might reveal unwanted behaviour by the client.

5.4 Efficiency

This section describes the evaluation of efficiency, detailed in Section 2.3, in relation to the results. It compares the old testing procedure at Delphi with the one in this thesis in terms of each of the efficiency categories.

Table 5.2: The different types of Actions

Action	Description	Usage
rule	Wait for given request and send the given response.	{ "action": "rule", "request": { ... }, "response": { ... } }
receiveMsg	Receive a message.	{ "action": "receiveMsg", "request"/"response"/"msg": { ... } }
sendMsg	Send a message	{ "action": "sendMsg", "request"/"response"/"msg": { ... } }
wait	wait for i seconds	{ "action": "wait", "time": i }
closeConnection	Wait for client to close connection.	{ "action": "closeConnection" }
openConnection	Wait for client to open a connection.	{ "action": "openConnection" }
serverCloseConnection	Close the connection with the client.	{ "action": "serverCloseConnection" }
repeat	Repeat the given actions i number of times.	{ "action": "repeat", "repetitions": i, "actions": [...] }

5.4.1 Similarity to Native Environment

As explained in Section 4.3, the only testing in the old environment that can be performed on a target simulation is integration testing. All other testing can only be performed on the developer PC, since it has dependencies on packages and libraries that do not exist on a target (simulation).

The new test environment has one external dependency, namely OpenSSL. However, this is a very common program and is often installed on target anyway. So, provided that OpenSSL is installed, the test environment can run on the target simulation, and could most likely be easily run on a HU as well. This enables testing closer to the native environment for the foundation services. As of now, only component tests are written, but there is no foreseeable problem to implement other tests such as unit tests and integration tests.

5.4.2 Code and Input Coverage

This section describes the evaluation of the Code and Input Coverage separately.

5.4.2.1 Code Coverage

Due to the fact that the current test environment mostly relies on a remote cloud provided by Volvo, which cannot be (easily) configured, it is troublesome to test the parts of the component that handle unsuccessful requests. With the introduction of negative tests in the new testing environment, the code coverage, naturally, increased.

Furthermore, new capabilities of the environment, such as denying the client requests, as seen in Figure 5.5, revealed previously undiscovered bugs related to retry-handling. It was found that the software did not retry at all for some resources, while it waited longer than expected to retry for other resources.

5.4.2.2 Input Coverage

With negative testing, the possible input data set grew, and with it came the possibility to verify desired behaviour of the component that otherwise did not exist. In some XML data transfers the new test environment could send faulty data, such as excessive tags or empty tags, and the component did what was expected. This is a behaviour that could not have been confirmed with the old test environment.

5.4.3 Time consumption

Construction of JSON-files is necessary in order to use the new test environment and is a bit time consuming. One of the authors of this Master's Thesis spent 1-2 hours on each test configuration file, which each contains 2-10 actions, and in addition, 8 hours were spent on writing scripts executing test suites. The last piece of the test environment is the Target Control and it took roughly 16 hours to complete. In total, the construction of a test suite containing 10 tests took 40 hours.

These are rough estimations and probably do not translate well into a "live" project. Some of the tests were previously constructed for the current testing environment and only had to be ported, which lowered the time consumption somewhat. However, the construction time of the test suite script includes research of bash scripts and the implementation time of the target control includes study of the test target, both of which would be lowered in a “real life” project. Furthermore, many of the scenarios can be reused between different testing type (UT, CT, IT) which cause very little overhead. With this in mind, the new environment would be equally time consuming in this regard, or at least not much more than the current.

The new environment can be run manually on a developer PC and automatically through Jenkins/Gerrit. Furthermore, it can run on both target system and "ordinary" systems, which is beneficial for the closeness to the developer. The developer

can build and test the new changes on their own computer and if it passes that, they can build and test it in target, which will increase the thoroughness of testing on an earlier stage. These are functionalities not provided by the old testing environment, where each type of testing is restricted to either manual/automatic testing and target/PC testing.

Therefore, it scores higher than the old environment, according to the specifications in Section 2.3.3.

6

Conclusion

We believe that a simulation tool such as the one described in this thesis answers the questions in Section 1.3 in a positive manner. This is mainly because of the high level of configurability but also the modular architecture of the software.

The high level of configurability satisfies **Q2**, **Q3** and mostly **Q1**. In the results of the evaluation of efficiency, in Section 5.4, the new environment clearly outperforms the current in the first two categories. In the case of "Time Consumption", the results are a little more ambiguous, although the ultimate verdict, in consideration to the discussion in Section 5.4.3, is that the new testing environment triumphs also in this regard. All though it is difficult to say whether the tool can successfully simulate every possible cloud, we can determine that the modes in Section 5.3 provides the functionality to simulate the cloud used in the project we worked with.

The architecture satisfies **Q4** and solves the remaining issues of **Q1**. In Section 5.1, it is shown that the simulation tool can be incorporated in the existing testing framework currently adopted at Delphi. If the framework is updated or replaced in such a way that it becomes incompatible with the simulation tool, the modularity of the architecture, described in Section 5.2, assists in locating and updating or replacing functionality.

One can argue that MockServer, described in Section 1.4, already provides sufficiently good answers to these questions. And indeed, the simulation tool in this thesis has similar capabilities as MockServer in its stateless mode. However, by accepting that input that does not conform to the HTTP/HTTPS standards, the input coverage is increased in the tool described in this thesis. Furthermore, with the introduction of the stateful mode even more functionality is supported, further increasing the set of possible positive and negative test cases.

All in all, it can be concluded that the efficiency increased with the new testing environment, and the configurability should satisfy the testing needs of the cloud services and provide support for most projects using HTTP/HTTPS, while the modular architecture permits easier modification of the tool in order to add even more support.

6.1 Future Work

Mitigating the time needed to configure the environment and writing tests would be beneficial. One way of achieving this could be to implement a support program with a graphical user interface that can help with the construction of JSON-files. It could enable a more “human” notation of the rules and actions, making it less verbose, and eliminating the requirement of insight into the JSON notation.

Another possible extension of this project could be to implement a program with the capability to generate negative tests, given a positive testing JSON-file. Since humans may not be very apt to construct "every possible" fault, computer generated negative tests would probably increase fault tolerance and mitigate time spent on brainstorming and writing negative tests.

Furthermore, the Test Control component in this testing environment is project specific. But, as briefly discussed in Section 4.1.2, one could create an API that the Target Control conforms to. If the Test Control is then made highly configurable by the user, for example, by accepting different input data, it would make the Test Control general, which could lower the time to setup the test environment for a new project. However, it adds a requirement on the developer/tester to study the API instead, so it might not mitigate the time by much.

Bibliography

[1] Myers GJ, Badgett T, Sandler C. The art of software testing. 3rd ed. Hoboken, N.J: Wiley; 2012.

[2] Chauhan N, Knovel. Software Testing: Principles and Practices. Oxford: Oxford University Press; 2010;2015;.

[3] McConnell S. Software Quality at Top Speed. Software Development. 1996;.

[4] Leeds HD, Spicer JaC, Weinberg GM. Computer programming fundamentals. New York: McGraw-Hill; 1961.

[5] Riungu-Kalliosaari L, Taipale O, Smolander K. Testing in the Cloud: Exploring the Practice. IEEE Software. 2012;29(2):46–51.

[6] Batra R, Sharma N. Cloud Testing: A Review Article. International Journal of Computer Science and Mobile Computing. 2014;3(6):314–319.

[7] Yu L, Tsai WT, Chen X, Liu L, Zhao Y, Tang L, et al. Testing As a Service over Cloud. In: Proceedings of the 2010 Fifth IEEE International Symposium on Service Oriented System Engineering. SOSE '10. Washington, DC, USA: IEEE Computer Society; 2010. p. 181–188. Available from: `http://dx.doi.org/10.1109/SOSE.2010.36`.

[8] Sivapathi A, Karthikeyan M, Saravanan K, Prakash V. Cloud testing: The cloud and our testing practices. Research Journal of Applied Sciences, Engineering and Technology. 2014;7(23):4940–4944.

[9] Nachiyappan S, Justus S. Cloud testing tools and its challenges: A comparative study. Procedia Computer Science. 2015;50:482–489.

[10] Bloom JD. MockServer. 2016;Available from: `http://www.mock-server.com/`.

[11] Clay B. Vertical Slicing; 2013. Available from: `http://www.slideshare.net/skydiver34275/vertical-slicing-v2`.

[12] Jorgensen P. Software testing: a craftman's approach. 4th ed. Boca Raton, Florida: Auerbach; 2013.

[13] Mahoney MS. The Roots of Software Engineering. CWI Quarterly 3. 1990;p. 325–334.

[14] Brooks FP. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Publishing Company, Inc.; 1975.

[15] Butterfield A, Ngondi GE. Dynamic Testing. A dictionary of computer science. 2016;.

[16] Butterfield A, Ngondi GE. Static Analysis. A dictionary of computer science. 2016;.

[17] Wichmann Ba, Canning Aa, Marsh Dwr, Clutterbuck Dl, Winsborrow La, Ward Nj. Industrial perspective on static analysis. Softw Eng J UK Software Engineering Journal. 1995;10(2):69–75. Available from: https://web.archive.org/web/20110927010304/http://www.ida.liu.se/~tddc90/papers/industrial95.pdf.

[18] Nadig S. What is Negative Testing and How to Write Negative Test Cases? Software Testing Help. 2016 May;Available from: http://www.softwaretestinghelp.com/what-is-negative-testing/.

[19] Freeman S, Pryce N. Growing object-oriented software, guided by tests. Addison Wesley; 2010.

[20] ISO/IEC. unit test. Systems and software engineering: vocabulary. 2010;p. 386.

[21] Link J, Frölich P, Books24x7 I. Unit testing in Java: how tests drive the code. San Francisco, Calif: Morgan Kaufmann; 2003.

[22] What is Component Testing. ISTQB Exam Certification;Available from: http://istqbexamcertification.com/what-is-component-testing/.

[23] Butterfield A, Ngondi GE. testing (dynamic testing). A dictionary of computer science. 2016;.

[24] Leung Hkn, White L. Insights into regression testing (software testing). Proceedings Conference on Software Maintenance. 1989;p. 60–69.

[25] Collofello JS, Buck JJ. Software Quality Assurance for Maintenance. IEEE Software. 1987;4(5):46–51.

[26] Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering. 2001;27(10):929–948.

[27] Rothermel G, Harrold MJ. Selecting Tests and Identifying Test Coverage Requirements for Modified Software. In: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis. IS-

STA '94. New York, NY, USA: ACM; 1994. p. 169–184. Available from: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/186258.187171.

[28] Chen TY, Lau MF. Dividing strategies for the optimization of a test suite. Information Processing Letters. 1996;60(3):135–141.

[29] Falah B, Marghabi S. Towards Regression Testing Constraints. International Journal of Modeling and Optimization. 2014;4(6):504.

[30] Desikan S, Ramesh G. Software testing: principles and practice. Dorling Kindersley (India); 2008.

[31] Wescott B. The every computer performace book. Fraser Publishing Company; 2013.

[32] Chen Y, Sun XH. STAS: A Scalability Testing and Analysis System. In: 2006 IEEE International Conference on Cluster Computing. IEEE; 2006. p. 1–10.

[33] Elsayed EA. Overview of Reliability Testing. IEEE Transactions on Reliability. 2012;61(2):282–291.

[34] Meier JD, Farre C, Bansode P, Barber S, Rea D. Chapter 18 – Stress Testing Web Applications. Chapter 18 – Stress Testing Web Applications. 2007 Sep;Available from: https://msdn.microsoft.com/en-us/library/bb924374.aspx.

[35] Kindrick JD, Sauter JA, Matthews RS. Improving Conformance and Interoperability Testing. StandardView. 1996 Mar;4(1):61–68. Available from: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/230871.230883.

[36] Ping TP, Chan CP, Sharbini H, Julaihi AA. Integration of cultural dimensions into software localisation testing of assistive technology for deaf children. Software Engineering (MySEC), 2011 5th Malaysian Conference in. 2011;p. 136–140.

[37] Naik K, Tripathy P. Software testing and quality assurance: theory and practice. 1st ed. Hoboken, N.J: John Wiley & Sons; 2008;2011;.

[38] Kumar R, Singh S. Cloud Testing: Perspectives and Challenges. International Journal of Computer Applications. 2014;106(17).

[39] USER ACCEPTANCE TESTING (UAT) GUIDES & CONCEPTS. TestingBrain; 2011. Available from: http://www.testingbrain.com/blackbox/user-acceptance-testing.html.

[40] Hambling B, van Goathem P. User Acceptance Testing: A Step-By-step Guide. 1st ed. Biggleswade;Swindon;: BCS, The Chartered Institute for IT [Imprint]; 2013.

[41] Boudreau T, Tulach J, Wielenga G. Rich Client Programming: Plugging into the Netbeans™Platform. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press; 2007.

[42] The Open Group Base Specifications Issue 7, 2013 Edition. The Open Group; 2013. Available from: http://pubs.opengroup.org/onlinepubs/9699919799/.

A

The Delphi Work Flow

The work flow adopted by Delphi uses Gerrit and Jenkins. Gerrit is a free web-based team code collaboration tool in which the team members review each others's code changes. It also features close integration with git, a distributed version control system. Jenkins is a Continuous Integration tool which automatically builds and tests source code.

A high-level view of the typical work flow of a developer at Delphi can be seen in Figure A.1. First they (1) clone the repository of the project of interest, hereby referred to as project A, and (2) create a new branch on the local copy. They then make some changes to the code and proceed by (3) pushing them to Gerrit for review.

In Gerrit, other developers review the code and vote to accept or reject the proposed changes. Jenkins act as one of these users. When a change in project A is pushed to Gerrit, Jenkins automatically (4) starts testing by (5) running the testing scripts for project A. These scripts allocate and order test slaves to (6) start testing.

When a test slave receives an order to start testing, it clones project A, builds an image of it and installs the image on a Head Unit (HU). Then, it (7) runs the test by (8) initiating actions on the HU and (9) measures the response, e.g., the new state of the HU. When the test is finished, the test slave (10) returns the result to Jenkins.

Jenkins final job is then to collect all test results and (11) return the overall result to Gerrit, i.e., if the tests went well Jenkins will accept the changes, otherwise they will be rejected. Lastly, if no user as rejected the changes to the code, the developer is able to (12) commit them to the master branch of project A.

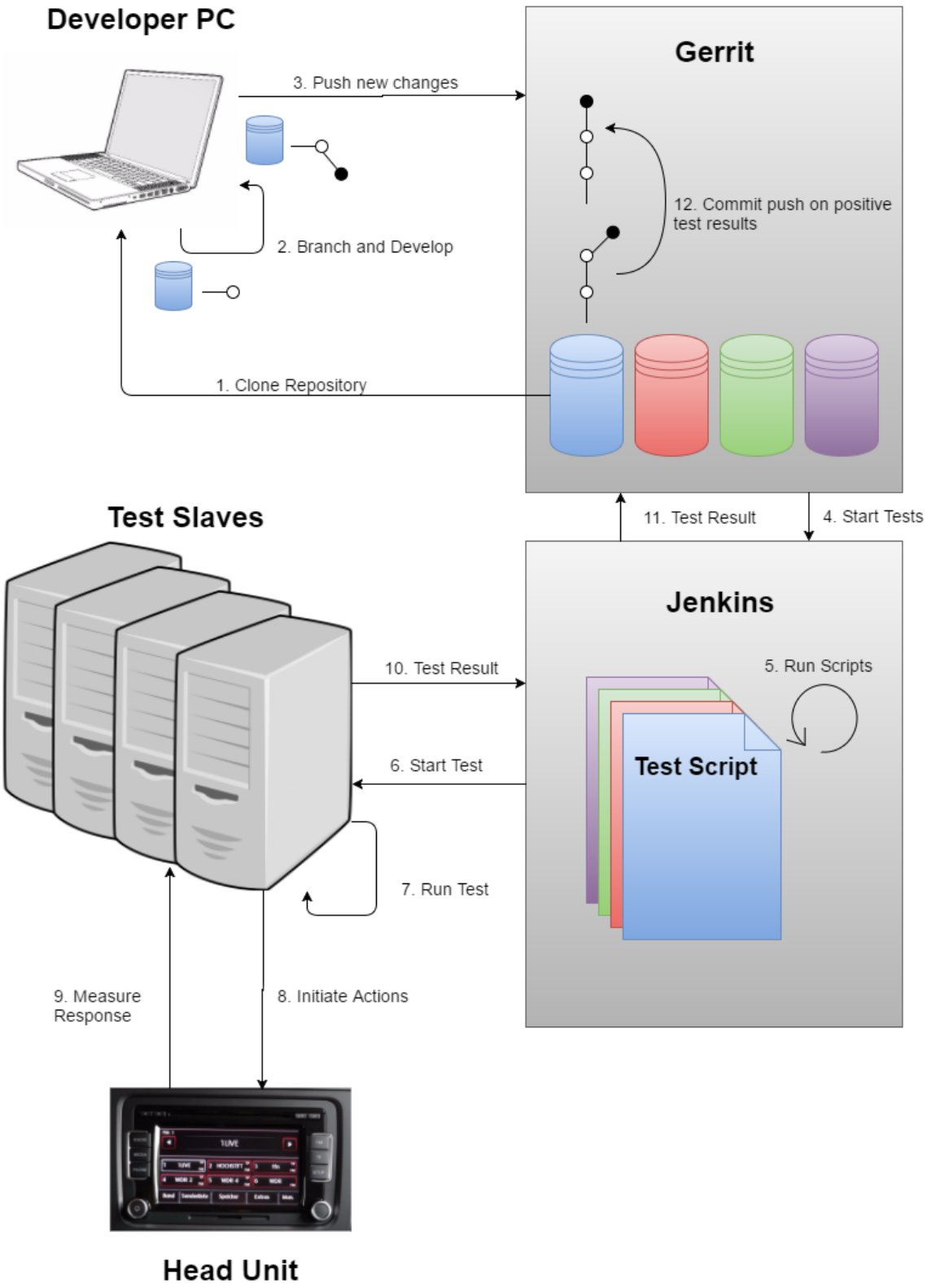

Figure A.1: Work flow using Gerrit and Jenkins.

B

Website Simulation

For demonstration purposes we wrote a rule set for a simple web site, see Listing B.1. This web site is a couple of pages and a form to post data to the server. All the requests are of type 'regex' to be able to serve most web browsers. The responses are of type 'full' or 'dataRead'. The path leads to the HTML file seen in Listing B.3.

In B.2 we have created a manuscript to test how a web browser's actions. First, we expect the web browser to request the start page of the web site, and then to navigate to /path. Secondly, we expect the browser to navigate to the start page again, but this time the server will wait for 60 seconds before replying. Thirdly, the browser is expected to close the connection and then to retry for the start page, this time, we reply immediately.

Listing B.1: Rule set for a simple web site

```
{
  "rootDataPath": "website",
  "rules": [
    {
      "request": {
        "msg-type": "regex",
        "msg": {
          "type": "GET",
          "version": "HTTP/1.1",
          "path": "/",
          "fields": {
            "Host": "127.0.0.1:30000",
            "Accept": "([[:alnum:]]|[[:punct:]]|[[:space:]])*",
            "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:alnum:]]|[[:punct:]]|[[:space:]])*"
          }
        }
      },
      "response" : {
                  "msg-type" : "dataRead",
                  "msg"      : {
```

```
21                      "version" : "HTTP/1.1",
22                      "code"    : "200",
23                      "status"  : "OK",
24                      "fields"  : {
25                          "Content-Type"      : "text/html;
                               charset=UTF-8",
26                          "Content-Encoding" : "UTF-8"
27                      },
28                      "dataPath"    : "start.html"
29                  }
30              }
31      },
32      {
33        "request": {
34          "msg-type": "regex",
35          "msg": {
36            "type": "GET",
37            "version": "HTTP/1.1",
38            "path": "/path",
39            "fields": {
40              "host": "127.0.0.1:30000",
41              "Accept": "([[:alnum:]]|[[:punct:]]|[[:space:]])
                   *",
42              "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
                   alnum:]]|[[:punct:]]|[[:space:]])*"
43            }
44          }
45        },
46        "response" : {
47                  "msg-type" : "full",
48                  "msg"      : {
49                      "version" : "HTTP/1.1",
50                      "code"    : "200",
51                      "status"  : "OK",
52                      "fields"  : {
53                          "Content-Type"      : "text/html;
                               charset=UTF-8",
54                          "Content-Encoding" : "UTF-8"
55                      },
56                      "data"    : "<html><head><title>This is
                           \"/post\".</title></head><body>Hello
                           World, This is \"/post\". <a href=\"
                           http://127.0.0.1:30000\">Return to
                           start</a></body></html>"
57                  }
58              }
```

```
59      },
60      {
61        "request": {
62          "msg-type": "regex",
63          "msg": {
64            "type": "POST",
65            "version": "HTTP/1.1",
66            "path": "/post",
67            "fields": {
68              "host": "127.0.0.1:30000",
69              "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
                  alnum:]]|[[:punct:]]|[[:space:]])*"
70            },
71            "data": "firstname=([[:alpha:]])*&lastname=([[:
                  alpha:]])*"
72          }
73        },
74        "response" : {
75                  "msg-type" : "full",
76                  "msg"      : {
77                      "version" : "HTTP/1.1",
78                      "code"    : "200",
79                      "status"  : "OK",
80                      "fields"  : {
81                          "Content-Type"     : "text/html;
                              charset=UTF-8",
82                          "Content-Encoding" : "UTF-8"
83                      },
84                      "data"    : "<html><head><title>This is
                          \"/post\".</title></head><body>Hello
                          World, This is \"/post\". <a href=\"
                          http://127.0.0.1:30000\">Return to
                          start</a></body></html>"
85                  }
86           }
87      },
88      {
89        "request": {
90          "msg-type": "regex",
91          "msg": {
92            "type": "POST",
93            "version": "HTTP/1.1",
94            "path": "/post",
95            "fields": {
96              "host": "127.0.0.1:30000",
97              "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
```

```
                alnum:]]|[[:punct:]]|[[:space:]])*"
98          },
99          "data": "firstname=([[:alnum:]])*&lastname=([[:
              alnum:]])*"
100       }
101     },
102     "response" : {
103               "msg-type" : "full",
104               "msg"      : {
105                   "version" : "HTTP/1.1",
106                   "code"    : "200",
107                   "status"  : "OK",
108                   "fields"  : {
109                       "Content-Type"     : "text/html;
                            charset=UTF-8",
110                       "Content-Encoding" : "UTF-8"
111                   },
112                   "data"    : "<html><head><title>This is
                      \"/post\".</title></head><body>Hello
                      World, This is \"/post\". Form had
                      number in values. <a href=\"http
                      ://127.0.0.1:30000\">Return to start
                      </a></body></html>"
113               }
114          }
115   },
116   {
117     "request": {
118       "msg-type": "regex",
119       "msg": {
120         "type": "GET",
121         "version": "HTTP/1.1",
122         "path": "([[:alnum:]]|[[:punct:]]|[[:space:]])*",
123         "fields": {
124           "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
                alnum:]]|[[:punct:]]|[[:space:]])*"
125         }
126       }
127     },
128     "response" : {
129               "msg-type" : "full",
130               "msg"      : {
131                   "version" : "HTTP/1.1",
132                   "code"    : "400",
133                   "status"  : "BAD REQUEST",
134                   "fields"  : {
```

```
135                         "Content-Type"     : "text/html;
                               charset=UTF-8",
136                         "Content-Encoding" : "UTF-8"
137                     },
138                     "data"     : "<html><head><title>This is
                           \"/test\".</title></head><body>Nope.
                           Not a valid path. <a href=\"http
                           ://127.0.0.1:30000\">Return to start
                           </a> </body></html>"
139                 }
140             }
141     }
142   ]
143 }
```

Listing B.2: Action set to test a simple web site

```
 1 {
 2   "rootDataPath": "website",
 3   "actions": [
 4     { /* Action 1: User requests start page */
 5       "action": "rule",
 6       "request": {
 7         "msg-type": "regex",
 8         "msg": {
 9           "type": "GET",
10           "version": "HTTP/1.1",
11           "path": "/",
12           "fields": {
13             "Host": "127.0.0.1:30000",
14             "Accept": "([[:alnum:]]|[[:punct:]]|[[:space:]])
                  *",
15             "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
                  alnum:]]|[[:punct:]]|[[:space:]])*"
16           }
17         }
18       },
19       "response" : {
20                   "msg-type" : "dataRead",
21                   "msg"      : {
22                       "version" : "HTTP/1.1",
23                       "code"    : "200",
24                       "status"  : "OK",
25                       "fields"  : {
26                           "Content-Type"     : "text/html;
                                  charset=UTF-8",
27                           "Content-Encoding" : "UTF-8"
```

```
28                      },
29                      "dataPath"     : "start.html"
30                  }
31              }
32      },
33      { /* Action 2: user navigates to /path */
34        "action": "rule",
35        "request": {
36          "msg-type": "regex",
37          "msg": {
38            "type": "GET",
39            "version": "HTTP/1.1",
40            "path": "/path",
41            "fields": {
42              "host": "127.0.0.1:30000",
43              "Accept": "([[:alnum:]]|[[:punct:]]|[[:space:]])
                  *",
44              "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
                  alnum:]]|[[:punct:]]|[[:space:]])*"
45            }
46          }
47        },
48        "response" : {
49                  "msg-type" : "full",
50                  "msg"      : {
51                      "version" : "HTTP/1.1",
52                      "code"    : "200",
53                      "status"  : "OK",
54                      "fields"  : {
55                          "Content-Type"     : "text/html;
                                charset=UTF-8",
56                          "Content-Encoding" : "UTF-8"
57                      },
58                      "data"    : "<html><head><title>This is
                          \"/post\".</title></head><body>Hello
                          World, This is \"/post\". <a href=\"
                          http://127.0.0.1:30000\">Return to
                          start</a></body></html>"
59                  }
60              }
61      },
62      { /* Action 3: user posts its name (Dave) to /path */
63        "action": "rule",
64        "request": {
65          "msg-type": "regex",
66          "msg": {
```

```
67            "type": "POST",
68            "version": "HTTP/1.1",
69            "path": "/path",
70            "fields": {
71              "host": "127.0.0.1:30000",
72              "Accept": "([[:alnum:]]|[[:punct:]]|[[:space:]])
                  *",
73              "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
                  alnum:]]|[[:punct:]]|[[:space:]])*"
74            },
75            "data": "name=Dave"
76          }
77        },
78        "response" : {
79                  "msg-type" : "full",
80                  "msg"      : {
81                      "version" : "HTTP/1.1",
82                      "code"    : "200",
83                      "status"  : "OK",
84                      "fields"  : {
85                          "Content-Type"     : "text/html;
                              charset=UTF-8",
86                          "Content-Encoding" : "UTF-8"
87                      },
88                      "data"    : "<html><head><title>This is
                          \"/post\".</title></head><body>POST
                          accepted. <a href=\"http
                          ://127.0.0.1:30000\">Return to start
                          </a></body></html>"
89                  }
90           }
91    },
92    { /* Action 4: user requests /post again. Now the body
         says "Hello Dave" */
93      "action": "rule",
94      "request": {
95        "msg-type": "regex",
96        "msg": {
97          "type": "GET",
98          "version": "HTTP/1.1",
99          "path": "/path",
100         "fields": {
101           "host": "127.0.0.1:30000",
102           "Accept": "([[:alnum:]]|[[:punct:]]|[[:space:]])
                  *",
103           "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
```

```
                alnum:]]|[[:punct:]]|[[:space:]])*"
104           }
105         }
106       },
107       "response" : {
108                 "msg-type" : "full",
109                 "msg"      : {
110                     "version" : "HTTP/1.1",
111                     "code"    : "200",
112                     "status"  : "OK",
113                     "fields"  : {
114                         "Content-Type"     : "text/html;
                              charset=UTF-8",
115                         "Content-Encoding" : "UTF-8"
116                     },
117                     "data"    : "<html><head><title>This is
                          \"/post\".</title></head><body>Hello
                          Dave, This is \"/post\". <a href=\"
                          http://127.0.0.1:30000\">Return to
                          start</a></body></html>"
118                 }
119           }
120     },
121     { /* Action 5: User request start page again, but server
            does not reply */
122       "action": "receiveMsg",
123       "request": {
124         "msg-type": "regex",
125         "msg": {
126           "type": "GET",
127           "version": "HTTP/1.1",
128           "path": "/",
129           "fields": {
130             "Host": "127.0.0.1:30000",
131             "Accept": "([[:alnum:]]|[[:punct:]]|[[:space:]])
                    *",
132             "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
                    alnum:]]|[[:punct:]]|[[:space:]])*"
133           }
134         }
135       }
136     },
137     { /* Action 6: Server waits for 60 seconds */
138       "action": "wait",
139       "time" : 60
140     },
```

```
141     { /* Action 7: Now the server replies with the start
            page */
142       "action" : "sendMsg",
143       "response" : {
144         "msg-type" : "dataRead",
145         "msg"      : {
146             "version" : "HTTP/1.1",
147             "code"    : "200",
148             "status"  : "OK",
149             "fields"  : {
150                 "Content-Type"     : "text/html; charset=UTF
                      -8",
151                 "Content-Encoding" : "UTF-8"
152             },
153             "dataPath"    : "start.html"
154        }
155        }
156     },
157     { "action": "closeConnection" }, /* Action 8: Server
          expects user to close connection */
158     { "action": "openConnection" }, /* Action 9: Server
          expects user to open connection */
159     { /* Action 10: user requests start page. The server
          responds with gibberish in head and body*/
160       "action": "rule",
161       "request": {
162         "msg-type": "regex",
163         "msg": {
164           "type": "GET",
165           "version": "HTTP/1.1",
166           "path": "/",
167           "fields": {
168             "Host": "127.0.0.1:30000",
169             "Accept": "([[:alnum:]]|[[:punct:]]|[[:space:]])
                    *",
170             "([[:alnum:]]|[[:punct:]]|[[:space:]])*": "([[:
                    alnum:]]|[[:punct:]]|[[:space:]])*"
171           }
172         }
173       },
174       "response" : {
175         "msg-type" : "full",
176         "msg"      : {
177             "version" : "HTTP/1.1",
178             "code"    : "666",
179             "status"  : "GIBBERISH",
```

```
180             "fields"  : {
181                 "Content-Type"     : "text/html; charset=UTF
                        -8",
182                 "Content-Encoding" : "UTF-8"
183             },
184             "data"    : "GIBBERISH"
185         }
186       }
187     },
188     { /* Action 11: The server will send 20 messages only
           containing spam to the user. */
189         "action": "repeat",
190         "repetitions": 20,
191         "actions": [
192             {
193               "action": "sendMsg",
194               "msg" : "spam"
195             }
196         ]
197     },
198     { "action": "serverCloseConnection" } /* Server closes
           the connection */
199     { "action": "openConnection" } /* Server expects user to
            reconnect */
200   ]
201 }
```

Listing B.3: The read HTML file of the web site.

```
1
2<html><head><title>An Example Page</title></head>
3<body>
4         <p>Hello World, this is a very simple HTML document.
             </p>
5         <ul>
6                 <li><a href=path>An other path</a></li>
7                 <li> <p> A POST request: </p>
8                         <form action="post" method="post">
9                                 First name:<br>
10                                <input type="text" name="
                                     firstname" value="Dan"><
                                     br>
11                                Last name:<br>
12                                <input type="text" name="
                                     lastname" value="Smith"><
                                     br><br>
13                                <input type="submit" value="
```

```
                                        Submit">
14                              </form> </li>
15                      <li>place holder adhaspi d</li></ul>
16</body></html>
```