An Accessible DSL for Deployment and Testing in the Cloud

Josselin Enet
josselin.enet@etu.univ-nantes.fr

Nicolas Hawa
nicolas.hawa@etu.univ-nantes.fr

Tristan Lenormand
tristan.lenormand@etu.univ-nantes.fr

Gaël Lodé
gael.lode@etu.univ-nantes.fr

Matthias Nantier
matthias.nantier@etu.univ-nantes.fr

Kévin Roy
kevin.roy2@etu.univ-nantes.fr

May 2020

Foreword

The work we present here is carried out as part of the master's degree in computer science from the University of Nantes. The goal of this work is to introduce research to students, by working on a common research subject and at the end, produce a scientific article describing their work.

Abstract

This Paper presents a way to deploy and test such applications by providing a Domain Specific Language (DSL) that makes the engineer in charge at ease when deploying or testing. The DSL presented in this paper improves the process of deploying and testing applications on the Cloud.

1 Introduction

Cloud computing is nowadays a popular technology for hosting software applications. However, deploying and testing such applications can be challenging because it involves many virtual machines, operating systems, programming languages, and libraries.

Distributed Software Testing is about running tests on the same software, but using different supports and languages. For instance, a software implemented in different languages, such as Python, C, or Java, and deployed on different operating systems, such as macOS, Linux, or Windows. Moreover, tests should be run in parallel and be synchronized to eventually return to the distributed testing application and give proper information.

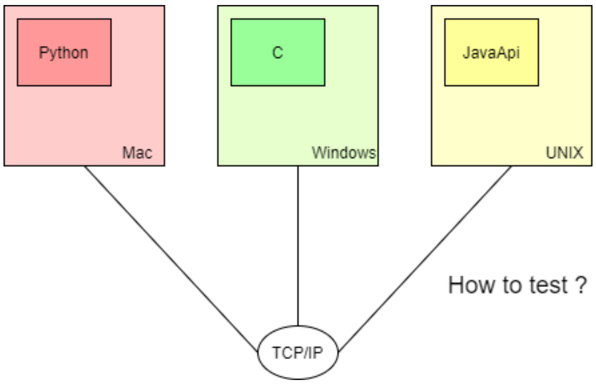


Figure 1: Cloud Testing Problematic

This raises several problems. First, the Distributed Testing Framework must run in parallel with the Software Under Test (SUT), requiring synchronization between them. The Framework must interact with the SUT by sending messages and gathering answers. Second, the specification of test cases becomes complex. On one hand, the Framework aims to be used by software developers and should be as usable as possible. On the other hand, as the tests might run on different operating systems and implementations, a way to communicate efficiently and globally with all those systems has to be found.

So far, some tools dedicated to Distributed Software Testing exist, but they are rather complex and inconvenient to use. Moreover, they require considerable knowledge about the actual design and implementation of the System Under Test. In this paper, we propose a way to conduct distributed testing with little configuration and effort. The contribution of this paper is two fold. First, we propose a Domain-Specific Language (DSL) for deploying of the SUT [1]. Second, we propose another DSL for writing test cases[2].

This paper is organized as follows. The next section provides a background on DSLs and on cloud systems. Section 3 presents an approach for the deployment and test of distributed applications. Section 4 details the solution proposed to solve this problem. Finally, Section 5 presents the related work and Section 6 concludes.

2 Background

2.1 Domain-Specific Languages

The term “Domain-Specific Language”, as opposed to “General Purpose Language”, describes a programming language that relies on a particular application domain. In other words, it is a language that responds to specific needs and allows to express specific problems. In the present case, a language that is easy to use for everyone, including for the ones who are not familiar with hard coding, in order to specify distributed tests. This can be achieved in many ways, but as we will described later, we chose to use command chain expressions from the Groovy language[3].

2.2 Distributed Software

Distributed software is a software in which components are spread across several computers nodes and/or networks, but is perceived by users as a single one. Conversely, Non-Distributed software run on a single device instead of many. Devices are in communication with Distributed software, involving transfer of information between them. Common examples are software applications that share media like music, for instance Apple's iTunes or the zero-configuration network: Zeroconf [4].

In these cases, when one computer misses a soundtrack, it will therefore search if any other computers has it over the local network the first computer is part of. However, those computers might be running on different operating systems and the software might also be implemented in a different language.

2.3 Distributed Testing

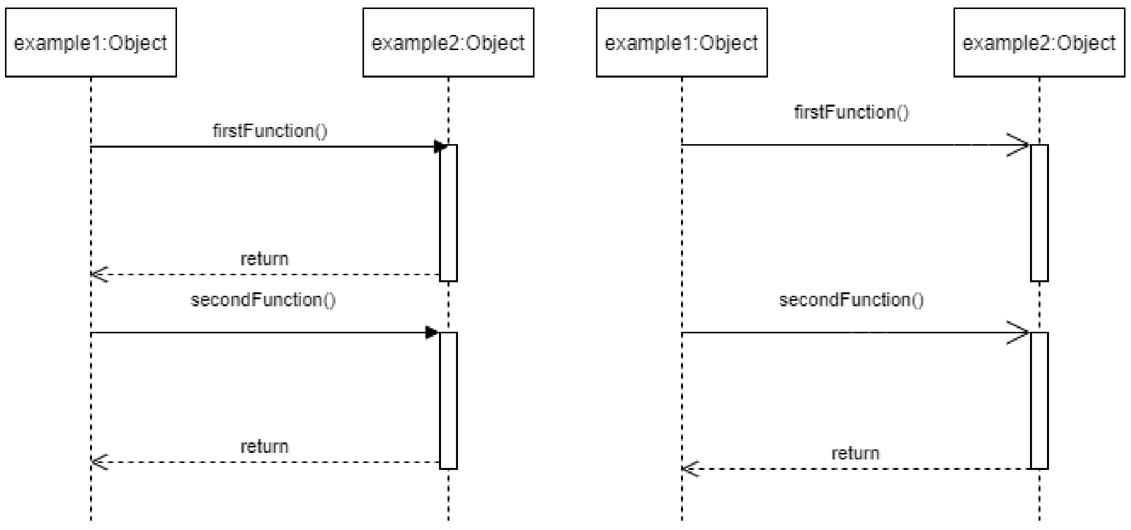
Distributed testing means testing several parts of the application located on different systems. Unlike parallel testing, distributed testing suggests that these nodes interact with each other during a test run. Therefore the coordination and synchronization of these interactions are the crucial points in distributed testing because each test should be deterministic.

2.4 Nodes and Messages

The components that are spread across the network, as described above, are called nodes. They are often applications that contain functions that are called through messages. Hence, the messages are actually requests from a component to others called nodes. These requests contain the name of the method to call in the target application and also the parameters that come along with it.

2.5 Synchronous and Asynchronous Messages

A message can be either *synchronous* or *asynchronous*, the main difference between these being the wait of an answer. When sending a *synchronous* messages, the application must wait for an answer before moving to another action, where sending an *asynchronous* message does not suspend the execution.



(a) Synchronous Messages

(b) Asynchronous Messages

Figure 2: Synchronous and Asynchronous Messages

As shown on Figure 2a, `firstFunction()` is a *synchronous* message, meaning that the object `example1` needs the return from `example2` to proceed to `secondFunction()`. However, on Figure 2b, the *asynchronous* message `secondFunction()` can be executed right after `firstFunction()`, without waiting for a return. Figure 2b shows that an *asynchronous* message can also have a return message, but the object which send it must not necessarily wait this answer before processing another statement. In this case, return messages are also asynchronous.

3 Approach

3.1 Deployment of an application on the Cloud

The objective of the deployment on the Cloud is to execute or install an *element* or a predefined cluster of elements called *node* onto an off-site Virtual Machine *VM*. An *element* is on par with the application and can be for example a compiler, a library, or another application. The virtual machine is given via a cloud provider such as Google Cloud, AWS (Amazon Web Service) or others. In order to solve this problem, we need to build a language that is in charge of deploying or executing the different required resources.

As mentioned earlier, the DSL in charge of the deployment must contain all the information needed to complete the desired deployment. Thus, the language requires either a node name or an element identifier, a cloud provider and the VM specifications. To deploy the node, the name should be already known by the script, due to the nature of the language it cannot be introduced later on. A solution to make this unknown node deployment possible, would be to deploy the elements one by one, if and only if those elements are already known.

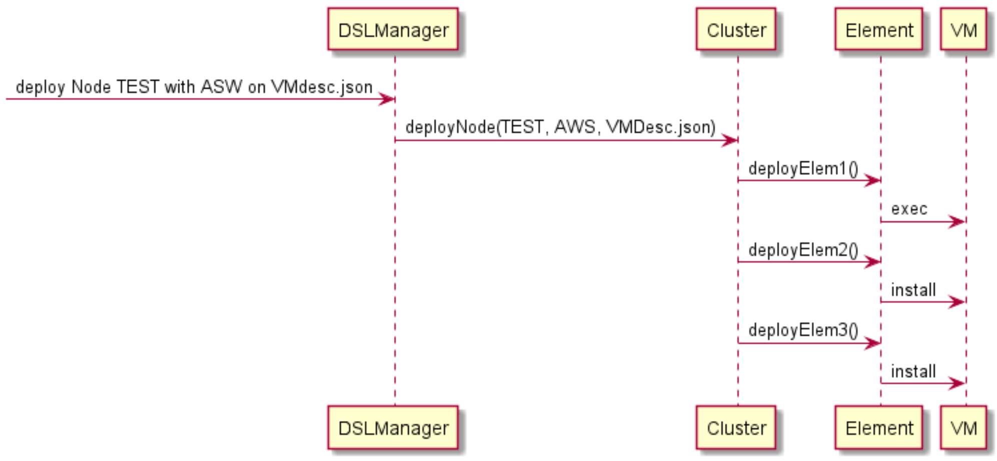


Figure 3: Deployment Sequence Diagram

3.2 Distributed Test Execution and Specification

Figure 4 presents the overall architecture of the test phase. The applications constituting the software under test are deployed on servers referred as Nodes. They receive instructions from a Test Controller, which keeps track of the execution of the tests. The communication between the Controller and the application is ensured by an Adapter.

The test approach is organized as such: specific test methods are implemented in the Nodes. The required tests are described in a script passed to the Controller, which handles the synchronization wit the Nodes. These tests are composed of multiple calls to the Adapter's methods. These calls may be *synchronous* or *asynchronous*, meaning that the Controller can wait for a specific call to respond or not before executing the next instruction. The keyword *send* refers to asynchronous messages, while *call* is used for synchronous messages. For a test to be complete, all responses should have been received, regardless of the nature of the messages.

Therefore, tests are divided into two parts: the test requirements, which gives a list of the different nodes used during the test, and a sequence of instructions which represents a functional test of the application. Each test is given a name so it can be added multiple times, but this feature will be discussed in Section 4.

Listing 1: Test Definition using DSL

```
need 2, "BasicNodeType", asName "node1", "node2"

send "testMethod1" withArgs 1 on "node1"
call "testMethod2" withArgs 1, "hello" on "node2"
```

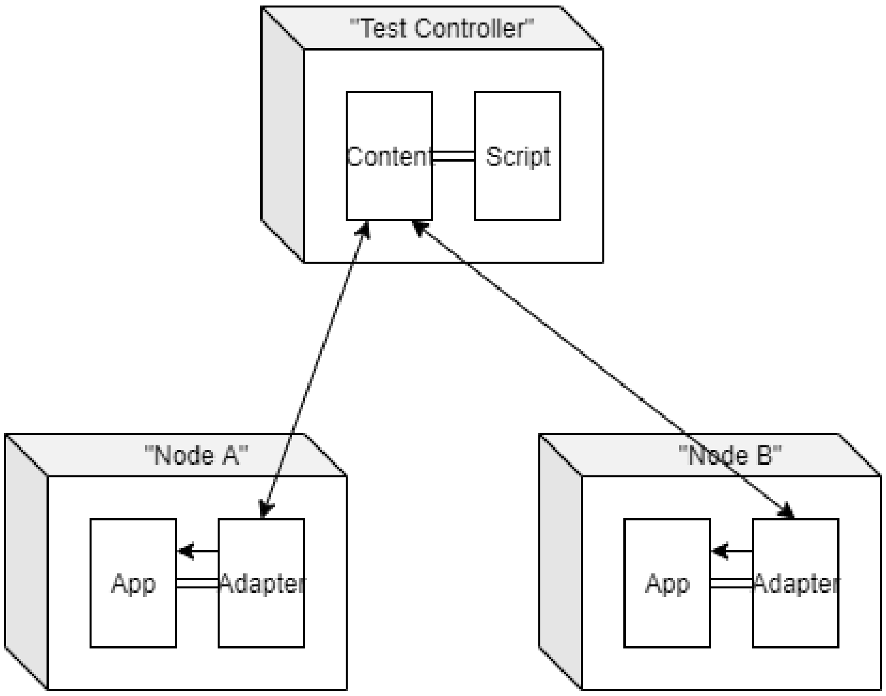


Figure 4: Test Architecture

The controller is composed of a test case to be executed, given at the start of the execution, and a list of available nodes. When a Node starts running, it notifies its presence to the Controller, which adds it to its list. Each Node is assigned a specific type, corresponding to the test methods it is implementing. When all the required nodes for a specific test are available, the test is executed and the Controller keeps looking for the others.

Listing 2: Test Addition

```
addTest "BasicTest"

tryExecute
```

The adapter is a class, implementation-specific, written in the language of the node's application, that will make the connection between the controller and the application. It receives messages from the controller and translate them into requests that are sent to the application that need to be tested. We chose to use an adapter as an intermediate as it avoids having to modify the source-code of the node's application directly. As this is not really advisable, we introduce the concept of an Adapter, enabling an easy way to communicate with the controller and sending afterwards requests to the nodes without having to modify them.

Another purpose of the DSL is to give the tester an easy way to inform the needs of his tests. The person in charge of testing the application should be able to say in the test controller what type of nodes do he needs, and for each type the number of instance, as well as a name for every instance to be able to use them in the execution part. The DSL also provide an explicit syntax for communicating with the different nodes of the application, especially with the *synchronous* or *asynchronous* property.

4 Implementation

4.1 Groovy

The Test Controller is implemented in Groovy, an object-oriented programming language that uses the Java Virtual machine (JVM), ans is inspired by Python, Ruby and Smalltalk. The main reasons why Groovy was chosen over others languages to implement the DSL is because Groovy has a refined and flexible syntax. Moreover, the goal was to have the tester write the less amount of code possible to execute his tests. Another reason why Groovy is a good option is because the language allows to run external scripts into the main program.

Java code :

```
call("testMethod1").on("node1");
send("testMethod2").on"(node2");
```

Groovy code :

```
call "testMethod1" on "node1"
send "testMethod2" on "node2"
```

The Groovy code seems more user-friendly even for a non-programmer. Even though, Groovy is compiled into Java during execution, the language allows to bypass the generic rules of object-oriented language like the `object.method()` syntax illustrated in the Java code example above.

4.2 Deployment Specification and Execution

The DSL used for the deployment is as follows:

- To deploy a node:
  deploy Node the **NodeName** with **Provider** on **VM description**
- To deploy an element:
  deploy Elem the **ElementName** with **Provider** on **VM description**

The deployment can be realized by using our DSL either by modifying the main method inside the DSLManager class, this solution is fairly inconvenient and should not be used. Or, when executing the Deployment script the deployment sentence can be added as a parameter when the script is called. The deployment script could also be part of a Gradle project.

Listing 3: Example of the script execution

```
groovy DeployScript deploy Node TEST with AWS on VMdesc.json
```

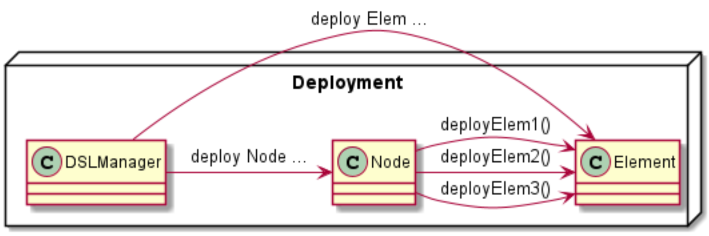


Figure 5: Deployment Diagram

As mentioned earlier, nodes are composed of elements, thus the node class is where nodes are defined. The `Element` class is either called by the `Node` or the `DSLManager` class and is used to deploy each specific node given by the user. Each node has its own deployment method, thus the deployment can be different from one node to another. For example, one node may check the OS or the version when another does not. That way Node definition is extremely flexible.

Listing 4: JSON VM

```json
{
    "Client":{
      "provider" : "local",
      "os": "Linux",
      "VM" : {
          "Java": {
            "version": "1.8"
          },
          "Gradle": {
            "version": "6.1.1"
          },
        "NodeServer": {
          "home_path": "Path␣to␣the␣directory␣of␣the␣node␣application"
          }
        }
    }
}
```

We use a JSON files to specify the description of virtual machines [5]. The description contains all the information about the element and nodes in a given VM but could also include its configuration, for example the operating system, or any information used in order to make the deployment possible. In the Listing 4 the VM has the elements Java and Gradle and the node NodeServer but we also have the information about the OS and the provider.

The choice of using the JSON format is led by the fact that it is easy to change, manage and is commonly adopted. The adopt of other similar formats [6] would require the development of new parsers, which is time-consuming and expensive. Listing 4 presents a model that uses only one VM called “Client” but this model can easily be changed and contain many other specifications.

As for the providers, a provider can easily be added and a VM can easily be accessed as long as the user can communicate with the VM or provider using the console. Groovy is able to execute any shell command, however if the user needs to execute more than one command he must create first a shell script with the correct execution rights that can then be executed by Groovy. Each provider is different and so are the communications, thus it is not possible to make a tool that would work for any provider. Some configuration files may already exist [7], but with this implementation the user is able to interact with any existing or future provider having its communication available through a shell script.

Listing 5: Groovy code to create and execute a shell script

```groovy
def path = scriptPath.home_path
File cmd = new File(path+'/cmd1.sh')
cmd.write( "cd␣$path␣\n")
cmd <<  "gradle␣clean␣build␣\n"
cmd<< "gradle␣runServer\n"
def exec1 = "chmod␣+x␣$path/cmd1.sh".execute()
exec1.waitFor()
"$path/cmd1.sh".execute()
```

The use of `waitFor()` in the Listing 5 is due to how groovy executes commands. Groovy does not wait for the command to be execute to continue the execution. Thus, in this case the file exists but may not be executable.

4.3 Test Specification and Execution

4.3.1 Test Controller

Groovy is used to create the test controller, which is a static super-class that contains all the objects needed to execute the tests: a container for the tests to execute and a container of all the available nodes. The DSL implemented inside the controller allows the tester to instantiate the starting configurations of the controller, such as the tests to execute.

Listing 6: Controller Initialisation using DSL

```
init "https://mqtt-broker.org", 3000
addTest "BasicTest"
tryExecute
```

Figure 6 shows an example of what an initialisation looks like. We use a MQTT server to communicate with the different nodes of the application, the communication protocol is explained more thoroughly in Section 4.3.5. First of all there is the keyword `init` to give the MQTT server address and the port used to communicate with the different nodes. The keyword `addTest` is used in the second line to add a test given its class name to the controller test case. When a test is added, its requirements are read by the controller by a specific method detailed in Section 4.3.2. Finally the keyword `tryExecute` is used to force the controller to try to execute all of its recorded tests, however it is an optional command since the controller will check the possibility of executing each test of the test case every time a node notify itself.

This component also uses multi-threading for improved performance. Every time a test is launched, a new thread dedicated to its execution is created. The original thread then checks if remaining tests can be executed. If not, it waits until new Nodes become available.

4.3.2 Test Class

The Test class is an abstract class that implements a part of the DSL through some keywords. A custom test needs to extend the Test class and implements two methods: `readNeeds()`, whose purpose is to inform the different nodes needed for its execution and to give a name to each one, and `execute()` which contains the code of the functional test.

Listing 7: Test definition using DSL

```
class BasicTest extends Test {

    readNeeds() {
        need 1,"BasicNodeType" asName "node1"
    }

    execute() {
        send "testMethod1" withArgs 4 on "node1"
        5.times{ call "testMethod2" withArgs "a" on "node1"}
    }
}
```

The proposed DSL has few words, as showed in Listing 7. The `need` keyword is used inside the `readNeeds()` method to specify the number of all the different node types used, and the keyword `asName` is used to give a name to each one of them. In the `execute()` method, the tester can write the functional test in Groovy. As Groovy is a completely refine language, it allows to write code such as `5.times` that would take few lines in Java to implement. The only keywords given in addition are `need`, `call` and `send` that allows *synchronous* and *asynchronous* method calls on the nodes (see part 4.3.4). And since Groovy is compiled into Java, it handles directly Java code for Groovy-reticent programmers.

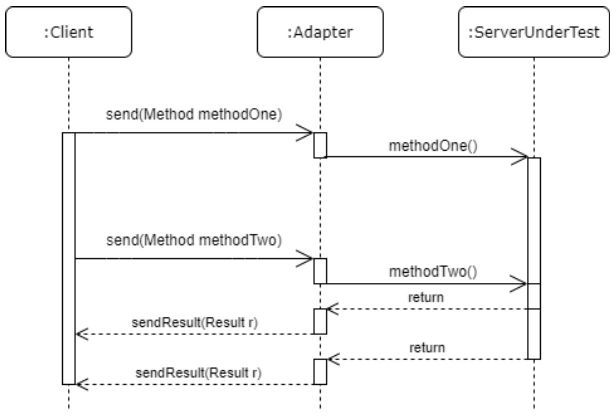
4.3.3 Adapter

An adapter is composed of two parts : one handles communication with the controller, while the other implements application-specific test methods. Along with the Controller, the Adapter could be developed in Groovy. However, this would make the communication with other languages turned out to be overly complicated. To solve this problem, the decision was made to instead use the language of the System Under Test meaning that the communication part has to be language-specific, and the test methods are implementation-specific. At this time, communication can be handled in Java. When adding new test methods, no further configuration is required: once implemented, it can directly be called by the controller.

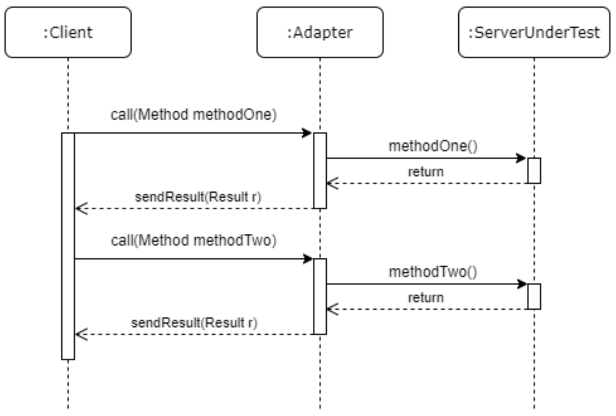
4.3.4 Message

One of the main concerns is how to be able to send unified messages from the controller that can be understood by all the adapters, even if they are implemented in different languages. In order to do that, we need to use serializing language, such as JSON, allowing to create a simple message template that can be easily sent to the adapter, deserialized, and understood by the application. The most efficient way to achieve this is to use Protocol buffers, often called Protobuf[8], a language-neutral extensible mechanism developped by Google for serializing structured data. It is therefore possible to create Protobuf messages that contain the name of the method to call in the nodes and its parameters, and then generate the message class in multiples languages including Java or C.

As said in the Background part, yet it is important to choose which type of message will be sent at which moment, depending of the answers needed. Then there are two different functions, `send()` and `call()`. The first one is *asynchronous* and the other is *synchronous*, which implies, that `send()` will wait result of the method send before running another `send()`, while `call()` allows the client to call other method even if result are not yet received.



(a) Send Message Example



(b) Call Message Example

4.3.5 MQTT Broker

Message Queuing Telemetry Transport [9], also called MQTT, is a message protocol based on the TCP/IP protocol. The MQTT broker is a server which receives message from a client and distributes it to clients that subscribed to that topic. This solution is appropriate in this case, as it is efficient at sending a message from one source to multiple targets.

Figure 6 shows a Deployment Diagram where the client sends a message to an MQTT Broker which can distribute this message to nodes that subscribed to MQTT. Then, in each node, the adapter that receives message can applies specific method as described in Section 4.3.3. In this Figure, arrows are double-headed, meaning that that nodes also send message to the client with MQTT Broker, corresponding to an answer.

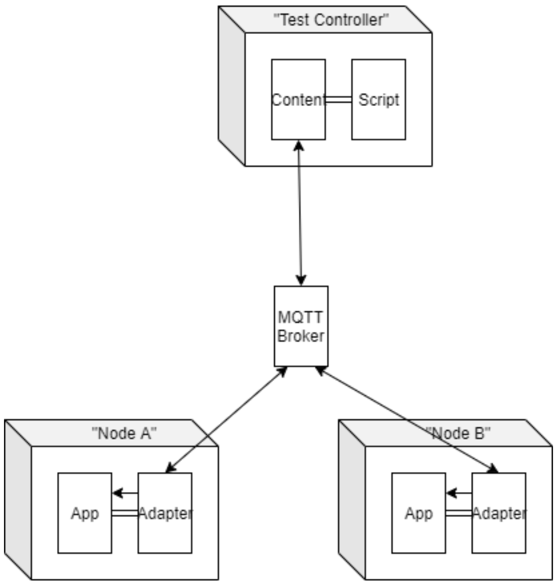


Figure 6: Diagram Client Server With MQTT

In order to start the test phase, an MQTT server must be running and all nodes must be initialized with its address and port. There are currently multiple implementations of this service, such as Mosquitto [10].

4.3.6 Observer

In addition to the console feedback during a test execution, an Observer pattern has been used in order to generate a persistent trace of the execution. It allows users to customize the testing phase output according to their needs and generate accurate reports. In practice, every time the controller sends or receives a message, a log is created and is added to a container that notify every recorded observers.

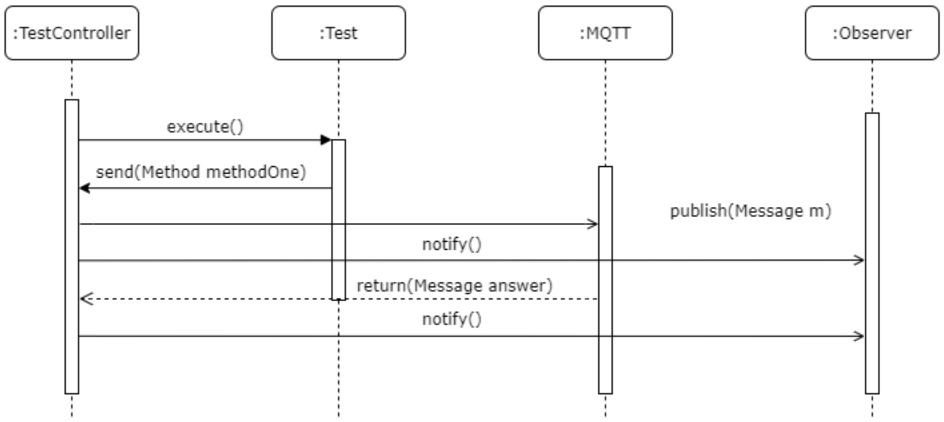


Figure 7: Observer Sequence Diagram

Because every method call induces an answer from the concerned node, there are 4 time-stamps related to each call: when the method call is sent from the controller, when the message is received by the adapter, when the response is sent from the adapter and finally when the controller receive the answer. A logs file printer can be obtained by an already implemented generic Observer. Each observer has to be declared during the initialisation of the controller via the keyword `addObserver ObserverClassName`.

5 Related Work

5.1 DSLs for Deployment

Even though deployment and more specifically cloud deployment is an essential part of any online application, it is still hard to find ways to make it more simple in terms of implementation or use and maintenance. Many research efforts have tried to find alternative ways, and many showed some DSLs but they were not easily usable or they were hard to implement [7, 11, 12]. The use of configuration files for providers is interesting [7] as it would be easily usable, however this information could also easily be contained in the aforementioned JSON configuration file.

5.2 Terraform

Terraform [13] is a Cloud management service that can manage different virtual machines on different providers. Terraform has the ability to generate execution plans and has many utilities to monitor each virtual machine you may have. Terraform could be used to handle the deployment but may not be used efficiently for testing purposes due to its main purpose which is to build large-scale multi-cloud infrastructures. Furthermore the use of a Graphical user interface may be easier for an end-user but slower than shell commands.

5.3 TTCN-3

TTCN-3 is a testing language for testing communicating systems[14]. Tests are done by declaring a message with a request and an answer. Then you are able to use a `send` method on the message based communication port, and a `receive` method giving respectively the Request and the Answer. In the receive method if the answer received is different from the one provided in parameter, the code continues to a branch, otherwise it goes to the other one.

Eclipse Titan is a tool made with TTCN-3 compilation and execution environment. It allows his users to write test case in TTCN-3 with an IDE Eclipse-based.[15] Even if it exists some practical tools to write code in TTCN-3, the programming language stay the same, and TTCN-3's writing itself is a problem, the way this programming language was made looks heavy and difficult to understand. Create a clear and concise language would make our language unique and innovative, in fact a software engineer that experiences this new fancy language should have an efficient use.

6 Conclusion

6.1 Main result

This paper shows a way to easily deploy and execute nodes and elements on any provided VM. Furthermore, this project provides a DSL, using Groovy's command chain expressions, that allows to execute distributed testing in an easy way. The tests are also running in parallel and returns information to the controller in the shape of a flexible report using the observer pattern. However, the adapter remains implementation-dependant and no global solution were found to have an unified adapter able to translate serialized messages to each specific implementation.

6.2 Open issues

Adding a provider or defining a node may not be as easy as it should because of the nature and Independence of each provider. Executing commands with Groovy is OS-dependant because of the execution of the same task is different depending on the operating system, thus limiting the use of this script. Externalising the use of the DSL to external Groovy scripts would be a great improvement to the existing work.

6.3 Future research directions

Cloud computing is developing rapidly and the need for efficient testing solutions is evolving accordingly. However, testing objectives are numerous and providing a unified mean of handling them is challenging [16]. This approach mainly focuses on asserting functional properties of the system under test. Other properties such as security are not taken into account, as their testing requires further configuration and would interfere with the simplicity of the DSL.

We are planning to add some more features to the DSL, particularly some tools to handle the response of a method call inside a test execution. The tester should be able to cast the response in a variable and then use it for the rest of the test. Also, since there isn't any real indication that tells if a test is succeed or not, besides the tester decision given the execution trace, we are planning to add optional assertion features that will determine the output of a test.

References

[1] Gaël Lodé Nicolas Hawa. *TER-Deployment*. `https://github.com/Oskilla/TER-Deployment`.

[2] Matthias Nantier Kévin Roy Josselin Enet, Tristan Lenormand. *TER - DSL for Cloud Testing*. `https://gitlab.univ-nantes.fr/E15G074H/ter---dsl-for-cloud-testing`.

[3] *Groovy*, accessed 8/5/2020. `https://groovy-lang.org/`.

[4] IETF. *ZeroConf*, accessed 8/5/2020. `http://www.zeroconf.org/`.

[5] Douglas CrockFord. *JSON*, accessed 8/5/2020. `https://json.org/json-fr.html`.

[6] Didier Donsez Vincent Zurczak Pierre-Yves Gibello Noel de Palma Linh Manh Pham, Alain Tchana. An adaptable framework to deploy complex applications onto multi-cloud platforms. 2015.

[7] Christina Thorpe Gerson Sunyé Adrien Thiery, Thomas Cerqueus and John Murphy. A dsl for deployment and testing in the cloud. 2014.

[8] *Protobuf*, accessed 11/5/2020. `https://developers.google.com/protocol-buffers`.

[9] Edited by Andrew Banks and Rahul Gupta. *MQTT Version 3.1.1*, accessed 8/5/2020. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`.

[10] Roger Light. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13):265, 2017.

[11] Behzad Bordbar Krzysztof Sledziewski and Rachid Anane. A dsl-based approach to software development and deployment on cloud. 2010.

[12] Tony Clark Dean Kramer and Samia Oussena. Mobdsl: A domain specific language for multiple mobile platform deployment. 2010.

[13] HashiCorp. *Terraform*, accessed 21/2/2020. `https://www.terraform.io/`.

[14] TTCN-3.org. *TTCN-3.org*, accessed 8/5/2020. `http://www.ttcn-3.org/index.php/about/why-use-ttcn3`.

[15] Eclipse Titan. *Eclipse Titan*, accesed 8/5/2020. `https://projects.eclipse.org/projects/tools.titan`.

[16] Antonia Bertolino, Guglielmo De Angelis, Micael Gallego, Boni García, Francisco Gortázar, Francesca Lonetti, and Eda Marchetti. A systematic review on cloud testing. *ACM Computing Surveys (CSUR)*, 52(5):1–42, 2019.