WWW.JAMK.FI

ENHANCED TESTING AUTOMATION PROOF OF CONCEPT FOR FREENEST

Niko Korhonen

Bachelor's thesis
November 2012

Software Engineering
The School of Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) KORHONEN, Niko	Julkaisun laji Opinnäytetyö	Päivämäärä 20.11.2012
	Sivumäärä 75	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty (X)
Työn nimi PARANNELLUN TESTAUSAUTOMAATION ESIMERKKI FREENESTILLE		
Koulutusohjelma Ohjelmistotekniikka		
Työn ohjaaja(t) PIETIKÄINEN, Kalevi		
Toimeksiantaja(t) RINTAMÄKI, Marko		

Tiivistelmä

Työssä oli tarkoituksena luoda toimiva testausautomaation esimerkki käyttämällä TestLink, Robot Framework ja Git -työkaluja. Testausautomaatin on tarkoitus ajaa TestLinkistä valitut testit automaattisesti pilviympäristössä käyttämällä Robot Frameworkia ja raportoida tulokset takaisin TestLinkille. Testien uusimmat versiot haetaan Git-repositoriosta ajon aikana.

Työn teoriaosassa on esitelty ohjelmiston testausta, pilvipalveluita, sekä projektissa käytettyjä työkaluja. Käytännön osassa on esitelty työssä tehdyn testausautomaatin elementtejä, arkkitehtuuria ja komponentteja, sekä itse automaation toimintaa. Asennusohjeet ovat työssä mukana liitteenä.

Testausautomaation perustoiminnallisuus saatiin rakennettua ajallaan siten, että sitä pystyi käyttämään FreeNestin testaamiseen. Osa suunnitelluista lisäominaisuuksista jäi kuitenkin keskeneräisiksi tai puuttumaan kokonaan ajan puutteen vuoksi, mutta niiden puuttumisesta ei kuitenkaan koitunut huomattavaa haittaa testausautomaation toiminnalle.

Avainsanat (asiasanat)
FreeNest, SkyNest, Robot Framework, Git, TestLink, Selenium, testausautomaatio

Muut tiedot

DESCRIPTION

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Author(s)	Type of publication	Date
KORHONEN, Niko	Bachelor´s Thesis	20.11.2012
	Pages 75	Language English
		Permission for web publication (X)

Title
ENHANCED TESTING AUTOMATION PROOF OF CONCEPT FOR FREENEST

Degree Programme
Software Engineering

Tutor(s)
PIETIKÄINEN, Kalevi

Assigned by
RINTAMÄKI, Marko

Abstract

The purpose of this thesis was to create a working testing automation proof of concept by using TestLink, Robot Framework and Git. The automation is supposed to run the tests given from TestLink in a cloud environment by using Robot Framework and report the results back to TestLink. Git is used for getting the most recent versions of tests during the testing run.

Software testing, cloud services and the tools used in the project were introduced in the theory part of this thesis. The elements, architecture, components and the functions of the testing automation are described in the practical part of this thesis. Installation instructions are included in appendices.

The basic functionality of the testing automation was built in time well enough for it to be used for testing FreeNest. However, a part of the planned features was left either unfinished or completely missing due to limited time; however, they did not cause any noticeable trouble for the testing automation.

Keywords
FreeNest, SkyNest, Robot Framework, Git, Testlink, Selenium, testing automation

Miscellaneous

CONTENTS

FIGURES

APPENDICES

TERMINOLOGY

FreeNest A project management environment developed in SkyNest -project.

SkyNest A project hosted at JAMK University of Applied Sciences.

Cloud Software Finland
A program of TIVIT that aims to improve Finnish competitiveness in software development.

Tekes The Finnish Funding Agency for Technology and Innovation.

MIDEaaS Mobile IDE as a Service, a project where testing automation is used.

NSN Nokia Siemens Networks, the supporter of Robot Framework.

Robot Framework A generic testing automation framework supported by NSN.

TestLink	A web based test management tool.
Git	A code version control system.
Python	An object-oriented, high-level programming language that emphasizes code readability. (What is Python? Executive summary, Python.org, 3.10.2012)
PHP	Hypertext Preprocessor, A general-purpose server-side scripting language used for dynamic web pages. (PHP, Wikipedia, 3.10.2012)
HTML	HyperText Markup Language, mainly used for displaying web pages and documents. (HTML, Wikipedia, 3.10.2012)
OpenStack	An open source cloud computing platform.
JunkCloud	Projects' own OpenStack cloud built from scrap computers.
Selenium	A tool suite for automating web browsers.
Selenium Grid Hub	The central point of the grid, distributes all the tests for nodes to run.
Selenium Grid Node	A slave machine in the grid, does all the hard work and reports the results to the hub.

1 INTRODUCTION

1.1 FreeNest

FreeNest is an open source application life cycle management environment developed in the SkyNest -project at the School of Technology at JAMK. Since the project is hosted at JAMK, most of the workers in the project are students either performing their internships or working on their bachelor's thesis projects. SkyNest is part of the Cloud Software Finland -program and it is funded by Tekes, the Finnish Funding Agency for Technology and Innovation. (Projects, Jyväskylä University of Applied Sciences)

FreeNest is meant to be used for managing the project life cycle from the initial idea to the development of the application, to support after the release. It integrates several widely used open source tools and acts like glue between them, making it possible for the tools to communicate between each other. It is built in a way that one FreeNest should be used in one project and when the project ends and a new one starts, the environment gets taken down and a new one is launched for the new project. If there are several projects, there have to be several FreeNests. (FreeNest, 2012)

FreeNest includes more or less all the tools that are needed in a software project and it can be configured for users' own needs. The current version at the time of writing this thesis, 1.3, is available as a virtual machine image, but 1.4 will be available as Debian packages, making the installation more flexible. The user can then decide which tools to install and the environment is ready for use in a short time. (FreeNest, 2012)

1.2 Cloud Software Finland

Cloud Software Finland is Tivit's four-year program initiated in 2010 and aiming to “significantly improve the competitive position of Finnish software intensive industry in global markets” according to Janne Järvinen, the Focus Area Director in the program. The most important factors are operational efficiency, user experience, web software, open systems, security engineering and sustainable development, and those are the areas the project focuses on. Tivit was founded in 2008 for the purpose of predicting the products and services of the future. It is owned by 46 companies and public research communities and funded by Tekes. (Cloud Software Finland brochure, August 2011)

1.3 Objective

The objective in this thesis is to create a proof of concept of a working testing automation tool chain. The tool chain should include a test management tool for all the test cases, a version control repository for the test scripts and a testing framework for running the tests on virtual machines. The integration should run all the tests related to the test case when the test run is started in the test management tool. It should pull all the latest test scripts from the version control repository and give them for the testing framework to take care of. After testing is finished, the test results should be combined and collected from the testing framework and finally sent pack to the test management tool in correct format.

2 MAIN CONCEPTS

2.1 Cloud computing

A cloud can be considered to be a large pool of resources, like a cluster of computers connected to each other, acting as one very powerful and flexible computer. All the data and power is spread on multiple nodes, so if some nodes break, their tasks are moved on to other nodes and the cloud is still working without any data loss. The computing resources can be distributed as services in the form of applications, servers, virtual machines or platforms over the Internet where ever they are needed and they can be scaled up or down depending on demand. The user does not have to buy or maintain new hardware since all the hardware is located in large data centers maintained by the service provider. (What is cloud computing?, IBM; Cloud Computing, Wikipedia)

There are several types of cloud services, the most notable ones being SaaS (Software as a Service), PaaS (Platform as a Service) and IaaS (Infrastructure as a Service)(See figure 1).

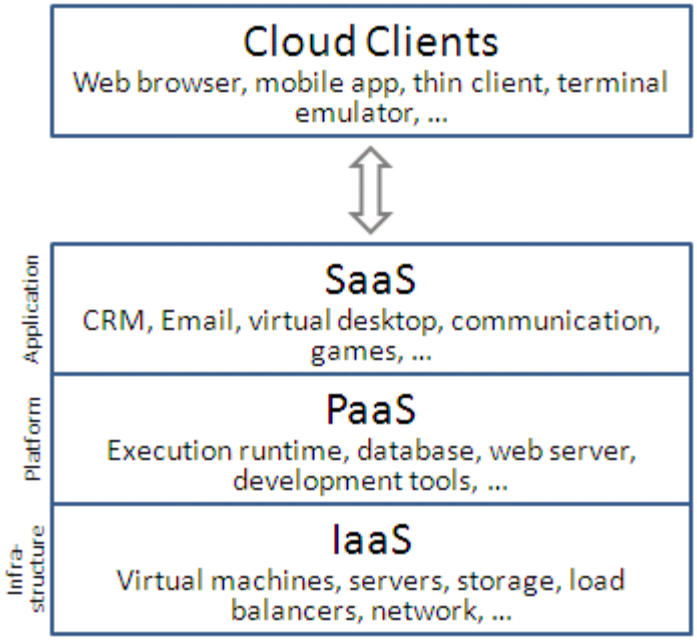

FIGURE 1. Cloud computing layers (Cloud Computing, Wikipedia)

2.1.1 Software as a Service

Software as a Service means a single application accessible via a client interface, such as a web browser. The application can be anything from project management tools like Trac to email services like Gmail. The users do not have to install or maintain anything on any of their computers since the software is installed in the cloud and maintained by the service provider. The user only needs to log in to the service from any computer and start using the application. (Cloud Computing, InfoWorld; What is cloud computing?, IBM; Cloud Computing, Wikipedia)

2.1.2 Platform as a Service

Platform as a Service means a software development environment complete with an operating system, development tools, database and a web server. Developers can use these tools to create their own software, which is then available for the end users directly from the web server. What development tools and technologies are available depends on the service provider, so the developers will not have complete freedom over what they can create; however the system is more predictable and manageable that way. Good examples of this type of cloud service are Google's App Engine and Salesforce.com's Force.com. (InfoWorld; IBM; Cloud Computing, Wikipedia)

2.1.3 Infrastructure as a Service

Infrastructure as a Service is the most basic type of cloud services. The user only receives machines or virtual machines including web servers, data storages and networking which are maintained by the service provider and it is the user's own responsibility to install the operating system and all the required software and keep them up to date. The user does not have to buy his own hardware to run his software and he can request more computing power if necessary. (IBM; Cloud Computing, Wikipedia)

2.2 Software testing

2.2.1 General testing

Software testing is a process to detect differences between the given input and expected output. It is used for assessing the quality of the

software and should be handled throughout the software's development. It is a verification and validation process, making sure the product behaves the way it is wanted to behave and that the product is built according to customer requirements. (CodeProject, March 2012)

Manual software testing is performed by a human sitting in front of the computer, going through the application, trying different usage and input combinations and comparing the results to the expected results. Testers follow a test plan and execute the tests according to the detailed test cases assigned to them. The results are then gathered from all testers and compiled into a test report. (SmartBear; Manual testing, Wikipedia)

2.2.2 Testing approaches

White-box testing

There are three approaches for testing software: white-box, black-box and gray-box testing. White-box testing, also known as glass box or structural testing, means that the tester has full understanding about what happens inside the program and test cases are designed using that knowledge and programming skills. The tester chooses his inputs in a way that he can check all the paths through the code and determine appropriate results. (CodeProject, March 2012; Software testing, Wikipedia)

Black-box testing

Black box testing, also known as functional testing treats the program like a black-box. The tester has no knowledge about the internal

functions of the program; he only knows what the program is supposed to do and not how it does it. The tester takes the role of an end user; tries different inputs and checks that the program returns appropriate outputs. (CodeProject, March 2012; Software testing, Wikipedia)

Gray-box testing

Gray-box testing is a combination of white-box and black-box testing. The tester can have access to the internal code of the program and has some knowledge about the functions of the program. The test cases are designed using that knowledge; however all testing is still done in black-box level. The testers can for example manipulate the data inside a database, which would not be possible in black-box level, or use reverse engineering to determine boundary values or error messages and then test the program like they would normally do in black-box testing. (Software testing, Wikipedia; Software Testing Fundamentals)

2.2.3 Testing Levels

Software testing can be categorized into testing levels based on the depth, scale and subject of the tests. There are three main levels of software testing: unit, integration and system testing as shown in figure 2. Acceptance testing is the last level of testing where the functionality of the software is compared to the business requirements and while it is an important part of testing, this thesis will not go into specifics or acceptance testing. (Software testing Fundamentals; Software Testing, Wikipedia)

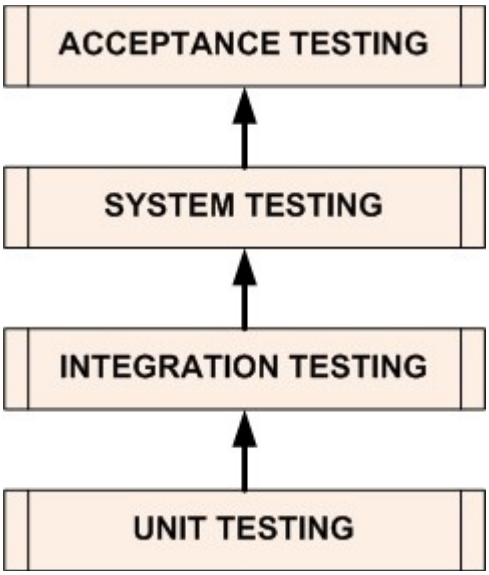

FIGURE 2. Major software testing levels (Software testing Fundamentals)

Unit testing

Unit testing means testing an individual unit or a group of units. This can be a simple method, a module or a class, basically the smallest testable part of the application. Unit testing is usually conducted by the programmer during the development process to determine if the unit he implemented produces the expected output. They need to be built in a way that they go through different inputs and can force errors and exceptions to be raised, they should only cover one unit at a time and they should be fast and small enough to be run while developing the software. They are meant to be used for catching errors as accurately and early in the development as possible, meaning that if a unit test fails, the developer most likely knows almost immediately what went wrong. Since knowing the system is required and the testing is usually carried out by the developers, unit testing falls under the white-box

testing approach. (CodeProject, March 2012; CodeThinked, June 2009; Unit Testing, Wikipedia)

Integration testing

Integration testing focuses on testing the interaction between a group of units of the system. While unit testing makes sure that individual units work as they should, integration testing makes sure they also work well together. If the software and hardware have any relation, that is also tested in integration testing. The testing is usually handled using the unit interfaces, simulating different scenarios by using different inputs, therefore integration testing usually falls under the black-box testing approach. (CodeProject; Software testing Tutor; Integration Testing, Wikipedia)

There are three approaches how to carry out integration testing: Top-down, Bottom-up and Big bang. In top-down approach the integration starts from the graphical user interface and follows the architectural structure to the bottom level, while in bottom-up approach the integration starts from the bottom level. In big bang approach all the developed modules are integrated together to form a complete system. This can save time compared to other approaches, but if the tests are not designed and recorded correctly, the testing process becomes more complicated and might even prevent testing the system properly. (Software testing Tutor; Integration Testing, Wikipedia)

System testing

The purpose of system testing is to verify that the software works as expected as a whole, fully integrated system. The software should be fully installed on different platforms with different configurations to

make sure that all scenarios are covered. Testing that the system installs correctly and uninstalls without leaving any traces are also covered in system testing. Test case designs are based on the original requirements from the end user's perspective, thus system testing falls under black-box testing approach. (Guru99; System Testing, Wikipedia)

System testing includes over 50 types of testing from usability and interface testing to stress, load and hardware/software testing, which means testing the interactions between the hardware and software. In most cases it is impossible to use all testing types for system testing due to limited resources and huge number of testing types, thus the tester needs to decide which parts are important enough for full testing to be necessary. (Guru99; System Testing, Wikipedia)

Regression testing

Regression testing, also known as verification testing is a special type of testing executed on all testing levels. It means repetitive tests that are run every time the application source code has changed. Its purpose is to verify that recent code changes have not affected the existing features of the application, created new bugs or resurfaced old bugs. Regression testing should cover features that undergo frequent changes and are most visible for users, and a good amount of test cases, including but not limited to all integration and complex test cases and the test that verify the core features of the application. Due to the repetitive nature of regression testing it can take up huge amount of resources depending on the frequency of the changes, so testing automation would be a valid solution to save testing time. (Guru 99; What is regression testing, Webopedia, 2012)

2.2.4 Automated testing

Automated testing means that instead of a human going through the software clicking buttons and collecting results, a machine can go through the tests and report the results to the test manager after it is done. It can be used in all testing levels from unit tests to full system testing on different environments. While it can be used for almost all kinds of testing, it is mostly used for regression testing, meaning the tests that need to be run frequently to make sure the software is working as it should.

Benefits

Making regression tests automated usually reduces the time and resources required for testing the system, since they have to be run every time someone makes changes to the source code. Running those tests manually multiple times is costly and time consuming, while automated tests can be run over and over again once the corresponding test scripts have been created. A machine also follows the given scripts precisely making the tests more reliable, while a human can and most likely will make mistakes during long testing runs. (SmartBear, 2012; Testing automation, Wikipedia)

There are also some test types that cannot be run manually. For example, they can take extremely long times, like when testing system stability, they can be too complex for a human to run perfectly or they might require huge amounts of resources like hundreds of users using the system at the same time to test the system performance. Testing automation can be left running over night and results can be collected in the morning, which saves working time. It can also simulate multiple users running the system at the same time using virtual machines.

(SmartBear, 2012; Testing automation, Wikipedia)

Downsides

While test automation can make a huge difference in bigger projects, it cannot be used efficiently in some cases and it can be more harmful than beneficial. Setting up the tool chain in the middle of the project can take a great deal of time and resources, and testing scripts need to be written for all the tests that need to be automated. This can be a serious problem if the project has been going on for a while and hundreds of tests need to be converted into automated tests. That is why automation must be considered at the beginning of the project, tools can be set up at the same time with the development environments and scripts are faster to create while developing new features.

3 TOOLS

3.1 TestLink

TestLink is a web based test management system developed by the TestLink community. It is designed to provide support for test specification and execution, and monitor test activities. It helps the testers to keep track of which tests are assigned to which testers, which tests are supposed to be run in the build and the total progress of the tests. In general it keeps the testing process under control. A screenshot of TestLink's interface can be seen in figure 3. (TestLink user Manual, March 2012)

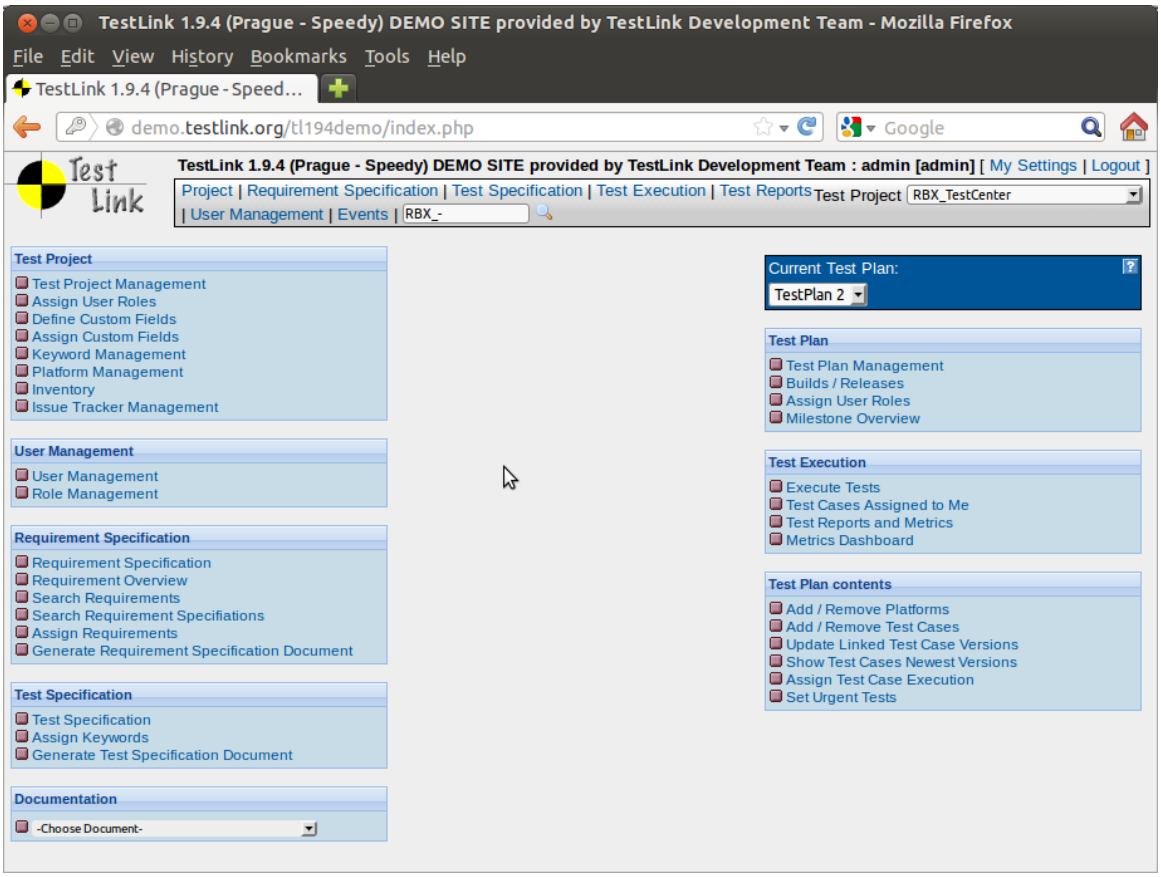

FIGURE 3. A screenshot of TestLink 1.9.4 demo.

Test projects

A test project is the base of the whole testing process in TestLink, without it there is nothing to be tested. It can mean a product or solution to be tested and it contains the requirements documentation, test specification, test plans and project specific user rights. Test projects do not share data, so each project should be used for only one product and by only one testing team if possible. Creating a new test project is usually the first thing that needs to be done after logging in with administrator's rights for the first time. This can be done by clicking “Test Project Management” -link in the front page after logging

in. When creating a project, there are two mandatory fields: name and prefix, which is used for test case IDs. The user can give some additional info or enable some features for the project, or use old projects as templates for the new project. (TestLink user Manual, March 2012)

Test plans

Test plans are the basis of the new testing activity. They hold the information about what needs to be tested and one project can contain several test plans, for example one plan for integration testing and another for system testing. New test plans can be created in “Test Plan Management” and they need a name, description and the status if the plan is active or not. The description should include information about the plan, like the scope of the plan, what needs to be tested, what should not be tested, risks, tools, references to documents etc. Test plans can also be removed or deactivated; however, removing is not recommended since it wipes all the data related to the plan, including test cases and their results. Deactivating only hides the plan from everywhere else except the overall test report. (TestLink user Manual, March 2012)

Test cases

Test cases are testing tasks that consist of test steps and their expected results and they are located in test suites like files inside folders. Test suites should be used for sorting the test cases into categories depending on the test subject for easier test management. Each test case can be considered as one test that verifies the functionality of a small part of the system under test and they can be executed either manually or automatically. Test cases are project

specific and they can be added into more than one test plan. (TestLink user Manual, March 2012)

Test cases can be created inside test suites right after the project has been created. The user can give a name, summary, preconditions and keywords for the test case, from which only the name is required. Summary and preconditions however are recommended since they give additional information for the tester and that can be crucial for the test to pass. After the test case has been created, the user can start adding steps for the test by clicking “Create step” -button. A step consists of the step actions and the expected results, and both are required for a proper test. Adding test steps is not required for automated tests, but the testing automation needs to be enabled from the TestLink configuration file. See an example of a passed test in Figure 4. (TestLink user Manual, March 2012)

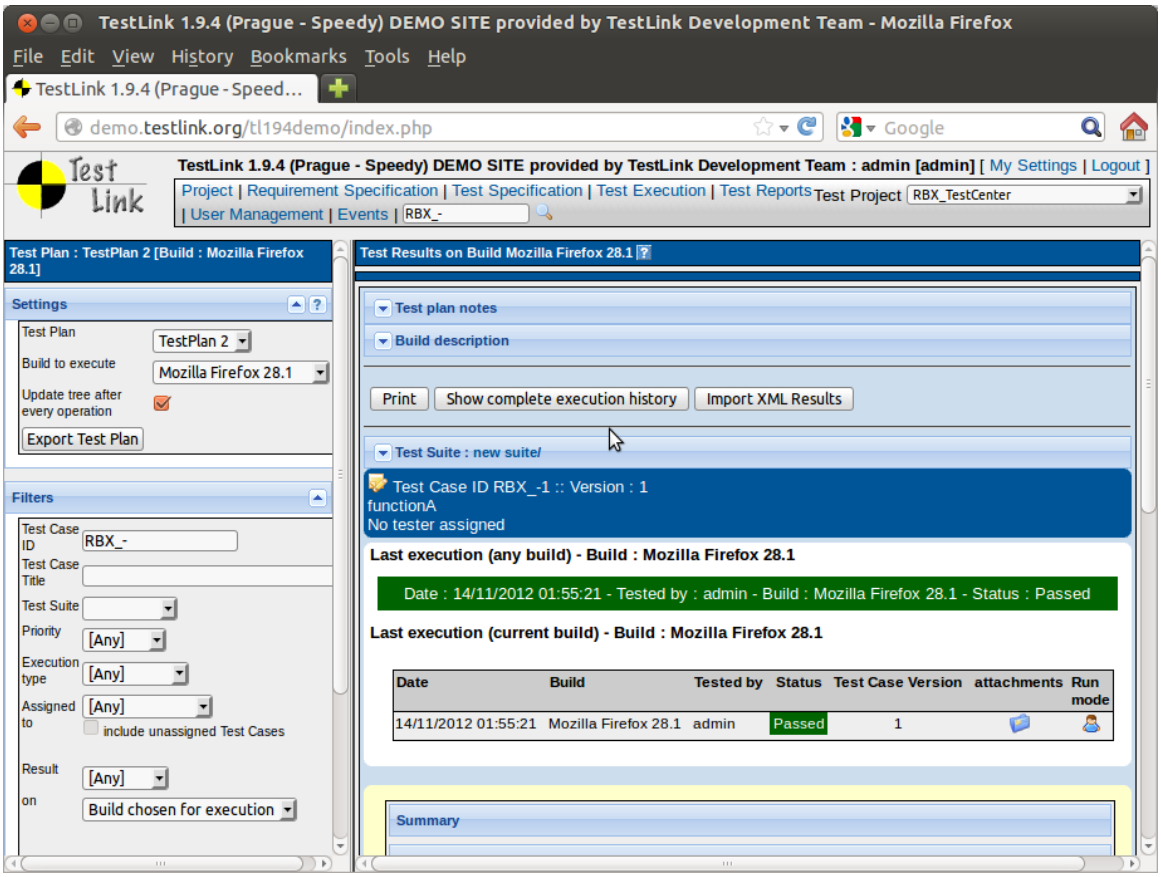

FIGURE 4. An example of a passed test case.

Before test cases can be executed, they need to be added into test plans by clicking “Add to Test Plans” -button in the test case overview. The button will not show up if there are no test plans in the project. Tests can be run from Test Execution -section in TestLink and the tests will not show up in the list before they are added into the test plan. Test execution also requires a build to be active in the plan. A build can be considered as a software release that needs to be tested, thus every time a new version of the software needs to be tested, a new build should be created for it and only the tests related to the changes should be run in the build. New builds can be created and activated from “Builds / Releases” in the project main page. (TestLink user

Manual, March 2012)

Custom fields

The user can modify the test cases to be more accurate by defining additional custom fields into the test project. They can be used for giving additional data while specifying new tests cases, making test reporting more accurate during the test execution, for example by saving execution times or additional notes, or for passing test parameters for testing automation. The custom fields can be visible for all projects, but they need to be assigned to a project before they can be used. By assigning the custom fields properly, each project can have its own set of custom fields in use. (TestLink user Manual, Mrch 2012)

Custom fields need to be defined by navigating in “Define Custom Fields” -link in the project main page in TestLink. The user can give the custom field a name, field type, availability and time allowed for editing, like during test specification or execution. Before the custom field can be used, it needs to be assigned to the test projects. This can be done by navigating in “Assign Custom Fields” in the project main page. They can only be assigned into the project that is currently selected in TestLink, thus they have to be assigned separately in each project. (TestLink user Manual, March 2012)

3.2 Robot Framework

Robot Framework is a generic test automation framework for end-to-end acceptance testing and acceptance-test-driven development. It can be used for all testing levels from running groups of simple unit tests to full scale system tests, and several types of applications like

simple command line applications as well as web sites. Its modular architecture allows it to run tests without the core knowing anything about the target system, all interaction between Robot Framework and the target system are handled by test libraries and lower level testing tools. See figure 5 for Robot Framework's architecture. (Robot Framework User Guide, September 2012)

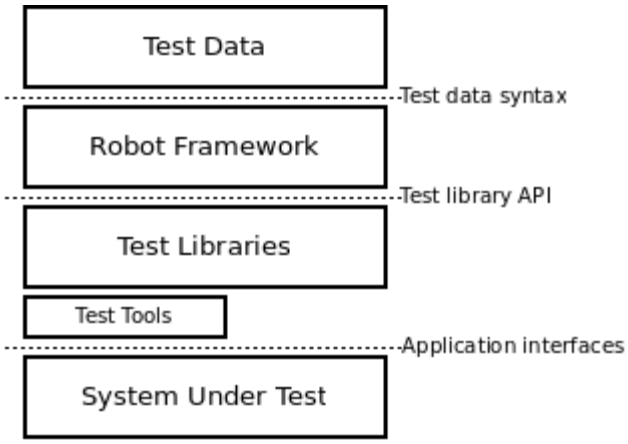

FIGURE 5. Robot Framework's high-level architecture. (Robot Framework User Guide, September 2012)

Robot Framework runs on Python and Jython, which is based on Java, and at least one of them is required to be installed. Robot Framework 2.7 and newer also have support for IronPython which is based on .NET. New keywords and functionalities can be enabled by importing libraries in tests, new libraries can be created using the provided library API and they can be implemented with Python or Java. (Robot Framework User Guide, September 2012)

Creating tests

Robot Framework uses test case files that are treated as test suites for all test cases in the file. The directories that contain test case files are considered as higher level test suites and they can also contain test suite directories, forming a deep hierarchical structure. One test case file can contain several test cases and each one of them is run when the script is executed. The test case file can be in HTML, TSV or plain text format, each one having slightly different structures. Their functionality however is the same, so the format to be used depends on the tester's preferences and the tools available. (Robot Framework User Guide, September 2012)

A test script can have tables with different purposes. The tester can change the test behavior by importing libraries, resource files and variable files, defining metadata for test cases or suites, or setting tags in the Settings -table. The variables used in the tests are defined in the Variables -table and user made keywords are created from lower level keywords in the Keywords -table. The test cases are created using keywords in the Test cases -table. All keywords are written in English, one keyword being in one row with its arguments. This makes the test scripts easily readable and manageable. See Appendix 1 for an example of a test script. (Robot Framework User Guide, September 2012)

Testers can create their tests using any editor they want, but there is also a tool called RIDE, which is designed for creating test case files more easily. Instead of writing all the keywords with a text editor, the user can create tests using a graphical interface shown in figure 6. RIDE can import libraries automatically when specific keywords are added, test settings are written in their own text boxes and keywords

can be chosen and added into the table from a list. The user can also run the test directly from RIDE to make sure it is working properly and export the test when it is finished. However, unlike Robot Framework, RIDE does not support Jython or Iron-Python, the regular Python is required for using RIDE. (How To, RobotFramework/RIDE Wiki, October 2011)

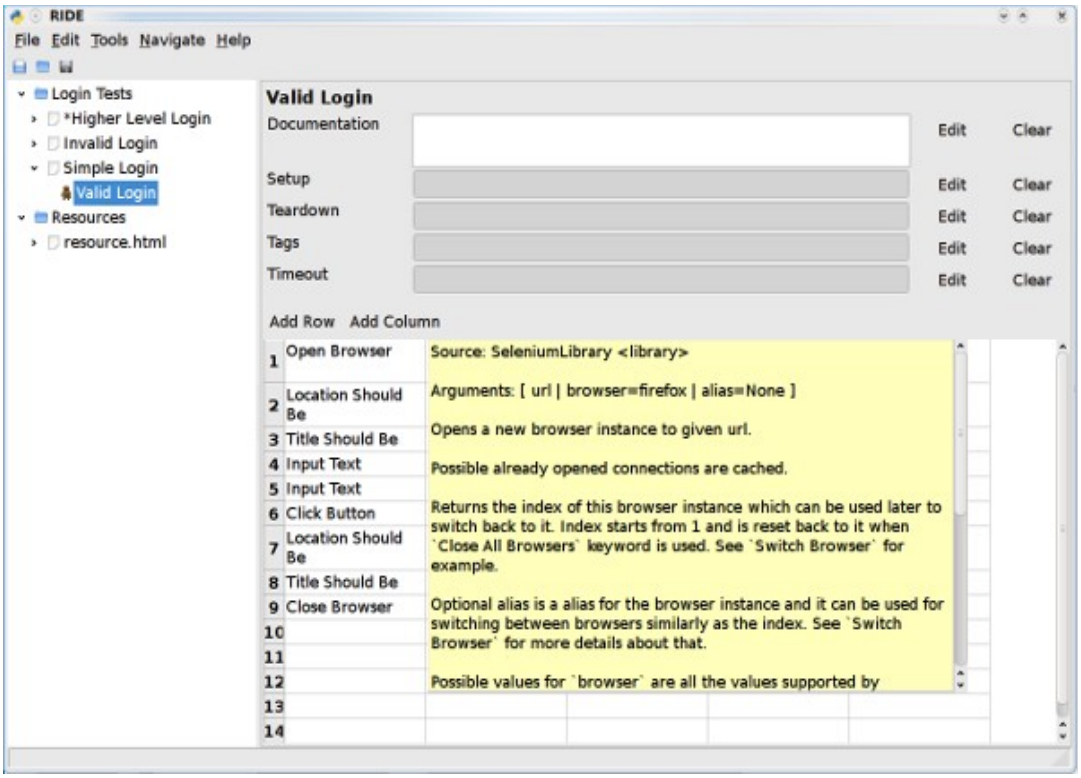

FIGURE 6. A screenshot of RIDE's interface (How To, RobotFramework/RIDE Wiki, October 2011)

Usage

Once Robot Framework is installed and tests are created, they can be executed from the command line using command “pybot [options] testScript1.txt”, or jybot with Jython or ipybot with IronPython. It also accepts several scripts at once or folders that contain scripts. The user can control Robot Framework's behavior using additional arguments in the command, for example redirecting or disabling output files with “-r NONE -l NONE”, giving meanings like criticality for certain tags with “--noncritical noncrit” or giving the suite a unique name for better readability with “--name MySuite”. The user can also pass variables for the script by giving the variable names and values in front of the test script, for example “--variable USERNAME:johndoe”. If the user installed RF as an executable Jar-file, tests are run using command “java -jar robotframework-2.7.5.jar [options] testScript1.txt”. (Robot Framework User Guide, September 2012; Robot Framework Introduction, 2012)

Robot Framework prints the test steps and their results to the command line when it starts going through the tests. If one step fails, by default the whole test fails and Robot Framework continues running the remaining tests. Unless a critical tag was set, all tests are treated as critical tests and failing one test will result failing the whole suite. If a non-critical test fails, the test suite can still pass, thus it is recommended that all tests that are still in development or have a huge chance to not pass are tagged as non-critical. (Robot Framework User Guide, September 2012)

Once all tests are run, Robot Framework prints a short compilation of the test results and creates three output files: log.html, report.html and output.xml. The log and report can be viewed using a browser and they

contain detailed information about all test steps and pictures of the failed steps, and they are meant for the tester to go through manually after running the tests. Output.xml is an XML-file containing all data from the tests. The XML-file can be used for importing the test results into the test management tool, or the data can be easily parsed for other tools to use. See figure 7 for the report files. (Robot Framework User Guide, September 2012)

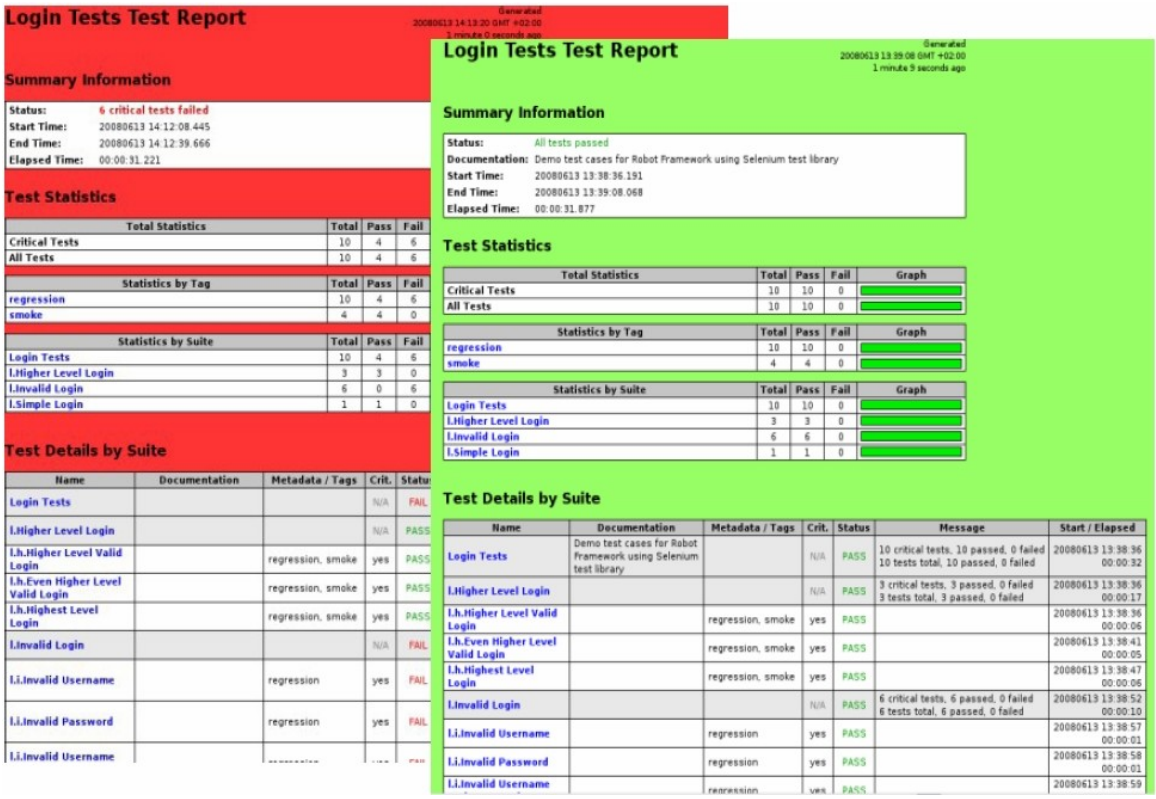

FIGURE 7. Failed and passed reports from Robot Framework (Page 11, Robot Framework Introduction, 2012)

Rebot

Rebot is an output post-processing tool included in the Robot Framework installation. The tester can use it to generate the log and report files if they were not generated during testing, or most importantly combine several output.xml -files into one. This is useful when tests have been executed on several machines or instances, for example when the same tests were run on different environments, and the test data should be readable from only one file. The output files can be combined easily by running a command “rebot [options] output1.xml output2.xml”. Since the names Robot Framework generates are not user friendly in the case of huge test suites, it would be a good idea to use the “--name MySuite”-option when combining the output files. (Robot Framework User Guide, September 2012)

3.3 Selenium

Selenium is a suite of tools for automating web browsers by Sauce Labs. It is mostly used with various testing frameworks for automatically testing web applications, but it can be used for example for automated web administration tasks as well. It also has support for multiple platforms, like Windows, OS X, Linux and Solaris, and it can be controlled by many programming languages (C#, Java, Perl, Python etc.) and testing frameworks (Bromine, Junit, Nunit, Rspec, Robot Framework etc.). (Selenium, August 2012)

There are several components of Selenium, each one having slightly different purposes, thus the users can choose which ones to use based on their needs.

Selenium IDE

Selenium IDE is an add-on for Mozilla Firefox, which allows users to record, edit, debug and run tests straight in the browser. The user can record his actions on the browser while he goes through the tasks that need to be automated, let it be some repetitive maintenance task or a complex website interface test, and turn it into an automated test script. He can save all the steps into an HTML -file, Ruby script or some other format and the task can then be run automatically whenever the user wants. The users can also edit their scripts if needed, in some cases some steps might not be recorded correctly and they need to be fixed manually in order to run the task properly. Tasks can also get changed for example due to interface updates, therefore occasional manual editing is always required. (Selenium IDE Plugins, August 2012)

Selenium RC / WebDriver

Selenium Remote Control (RC) is a website testing tool. It consists of two parts, Selenium Server and Selenium Core. Selenium Server acts as a receiver between Selenium Core and the testing program like Robot Framework; it receives the commands, translates and gives them to Selenium Core, and reports the results back to the testing program. Selenium Core attaches itself into the browser and runs the translated “Selenese” commands in the browser. See Figure 8 for the architecture.

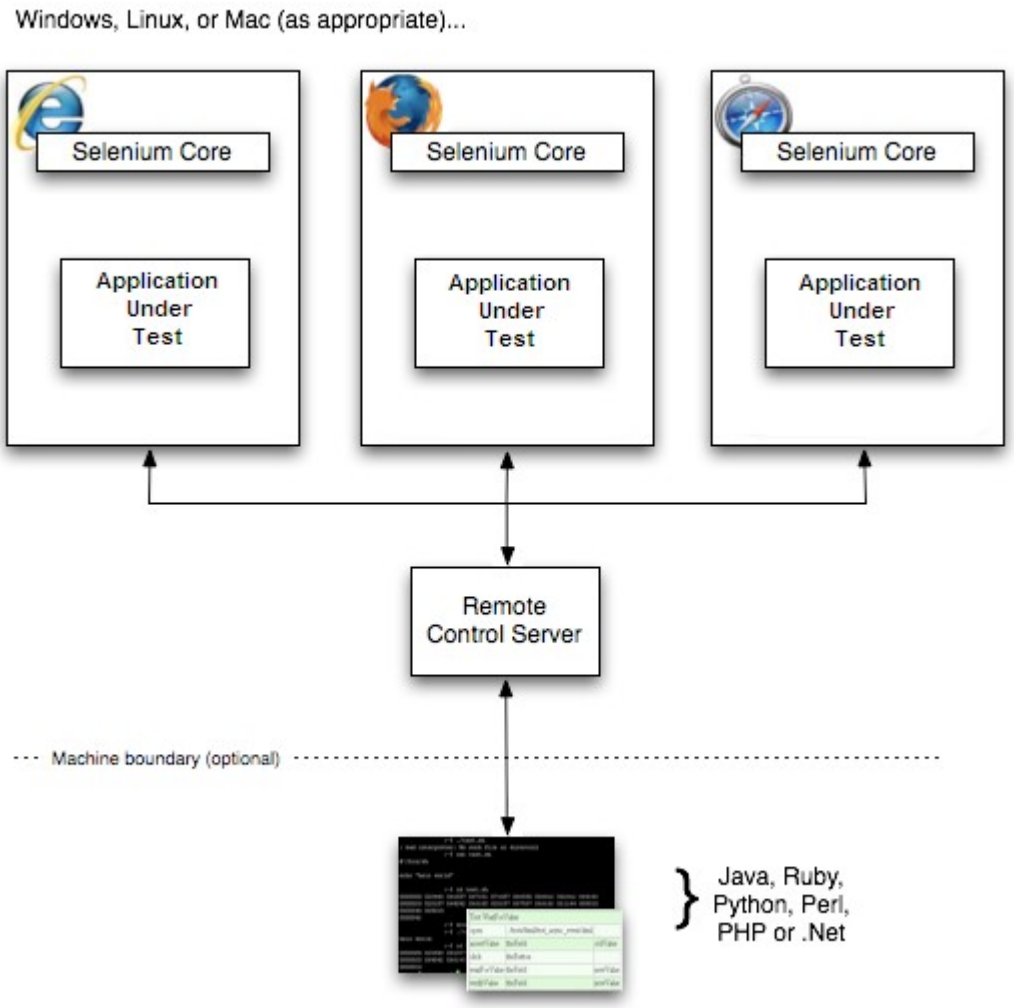

FIGURE 8. Simplified Selenium architecture diagram. (Selenium Documentation, August 2012)

Selenium WebDriver is a more advanced version of Selenium Remote Control. The main difference between them is that selenium RC uses JavaScript functions for running all the supported browsers and finding the correct elements from websites, while WebDriver uses each browsers built-in support for automation, running the browser directly instead of using JavaScript. This makes it more reliable to test dynamic websites where the content can change without the page being reloaded. It also fixes some limitations RC has and makes tests easier

to read and maintain than with RC. Although RC can be considered deprecated, it is still actively supported in maintenance mode due to providing support for more languages and most browsers. (Selenium Documentation, August 2012)

Selenium Grid

Selenium Grid is a built-in Selenium Server functionality. While RC / WebDriver usually runs on only one machine, Grid can form a network full of machines connected to each other. It consists of two kinds of servers: Hubs and nodes. A hub acts like central point in the network; it receives all the test requests and distributes them for all the nodes to execute. Nodes are the slaves in the network; they are all connected to the hub and when they receive tests, they execute them and report the results to the hub. In other words, test suites are not run on a single machine one at a time, they are run on several machines in parallel. This does not have much effect when the tests are simple and there are not many of them; however, Grid can run the tests several times faster than a single server when used with large and complex test suites, in theory twice as fast with two nodes, four times as fast with four nodes and so on and so forth. (Selenium Documentation, August 2012)

3.4 Git

Git is an open source distributed version control system. It was originally developed to hold the source code of the Linux kernel since the free-of-charge status of the original version control system, BitKeeper, was revoked in 2005. The design was based on the lessons the developers had learned while using BitKeeper and the main goals

of the new system were speed, simple design, strong support for non-linear development, fully distributed and ability to handle large projects. Git is used in some of the largest FOSS projects like Linux kernel, Ruby on Rails etc. (Getting Started - A Short History of Git)

Distributed version control system

Distributed version control system means that every copy of the repository is a full clone of the contents and the history of the repository. Instead of having all the files on a central version control server, everyone in the project has a full local copy of the project as in figure 9. In case the server dies, each one of the copies can be used as a backup for restoring the data into a new server. Furthermore, Git does not require network connection to work. The user can work on features and commit changes while being offline and push the work on the server when network comes available. Since the repository history is also saved locally, the user can compare the current version to a version committed a month ago for example, unlike with centralized version control systems where the project history is located on the server. (Getting Started – Git Basics)

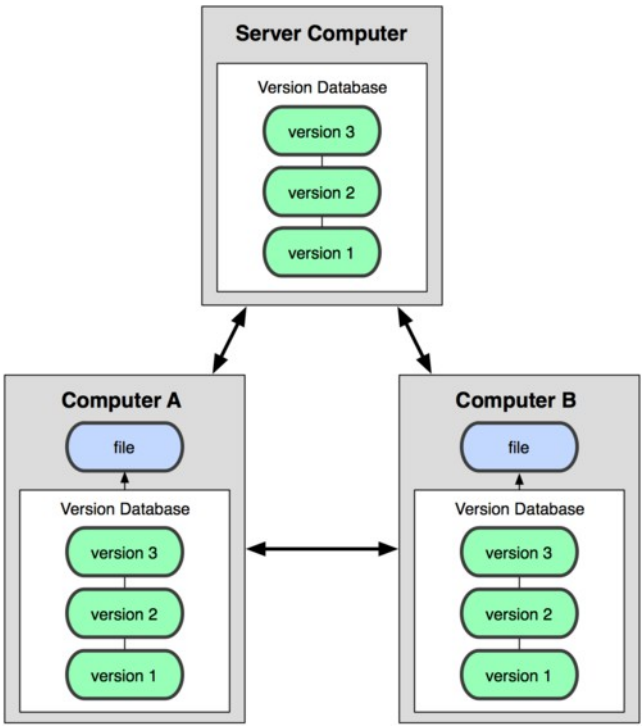

FIGURE 9. Distributed version control diagram (Getting Started - About Version Control)

Unlike most version control systems, Git does not save keep track of files as a list of changes to the original files. With each commit, Git saves snapshots of the current state the project and treats them like a small file system. Git only takes snapshots of the changed files while it creates a link to the previous snapshot of the unchanged files. (Getting Started – Git Basics)

Branches

Branching is one of the special features of Git. The project is always saved in a default branch called master. Branches are meant to be used for developing new features without interfering with the master branch and when the feature is finished, it can be merged to the

master branch and removed. Each branch is independent of other branches. For example a developer can work on new features in a development branch, checkout to master and create a new branch for a hotfix. After the hotfix is done, it can be merged with the master branch and the development of new features can continue from where the developer left off. This way the developers can work on their own parts of the project safely without causing conflicts with the rest of the team. (Git Branching - What a Branch Is ; About Git)

Usage

When the user starts working on a new feature, the first step would be creating a new branch for the feature by using a command “git branch featureX”. At this point the user is still in the master branch, so the branch needs to be changed with command “git checkout featureX”. The user can switch the branch at any point, but if there are untracked changes, Git will not allow the switch before the changes have been committed. The new branch is a complete copy of the master branch from the moment when the branch was created, so the user can start working on the feature. (Getting Started – Git Basics)

Whenever the user has made progress that needs to be saved, the work has to be added into the staging area first. This can be done with command “git add index.html”, index.html being the changed file. This does not mean that the file is saved; it only means that git will be tracking its changes. The file can be saved into the repository by committing it with command “git commit -m 'edited index.html'”. Now the changes have been saved locally into the branch where the user is working currently. (Getting Started – Git Basics)

After enough development the user can merge the branch to the master branch. This is done by switching to the master branch and using command “git merge featureX”. Git will automatically try to merge the changes but in some cases there have been changes to the same file in two branches. In these cases git might not be able to merge them automatically, thus the user has to open the conflicting file and fix the conflicts manually. Git adds conflict markers to the areas where the conflicting changes are located to make it easier for the user to solve the conflicts. After the file has been fixed, the changes must be committed again and the merging should work properly. (Getting Started – Git Basics)

The project can be shared either by cloning it, pulling the changes or pushing it to a remote server. Cloning is the first time operation when the project needs to be downloaded and it can be done with command “git clone server:path/to/repository.git” provided that the user has sufficient rights for cloning the repository. After the project has been cloned, the original repository is remembered as origin. The changes done to the repository after the cloning or latest pull can be pulled using command “git fetch origin”. This will only get the latest changes but it will not try to merge them. Using “git pull origin” however pulls the latest changes to the current branch and the files are merged, thus it can be more user friendly in some cases. The latest changes can be sent to the repository by using command “git push origin”, but it requires that the latest changes have been pulled from the repository and all conflicts have already been solved. (Getting Started – Git Basics)

3.5 OpenStack

OpenStack is an open source cloud operating system founded by Rackspace Hosting and NASA. Their mission is to produce a massively scalable but easy to use cloud computing platform that will meet the needs of public and private clouds. The system can be installed on standard hardware, like a group of desktop computers and there are no system requirements. The cloud architecture consists of three main parts; compute, networking and storage, and it all can be controlled via browser by using the dashboard, which is also known as Horizon. See figure 10 for the architecture. (Software, OpenStack)

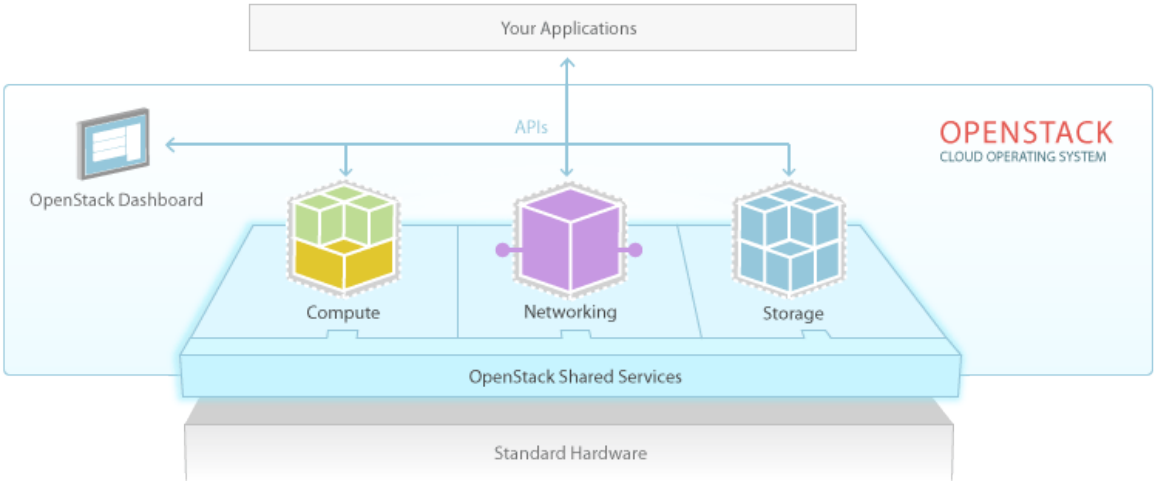

FIGURE 10. OpenStack operating system diagram (Software, OpenStack)

Compute

OpenStack Compute, which is also known as Nova, handles the computing resources of the cloud. It can manage large networks of

computers and turn them into computing power that can be distributed on demand. It is designed to scale horizontally on standard hardware and it can use MaaS, Metal as a Service automatically when a new computer is plugged into the network and use it as additional computing resources. (Compute, OpenStack)

Networking

OpenStack Networking, also known as Quantum, manages the device networking in the cloud. Administrators and users can use it for creating and managing internal networks and IP addresses, and controlling traffic. Quantum allows users to use floating IP addresses for dynamically rerouting the traffic for other instances during system maintenance or a system failure. Its purpose is to ensure that the network will not be the limiting factor in the cloud deployment. (Networking, OpenStack)

Storage

OpenStack Storage supports two types of storage; Object storage, also known as Swift, and Block storage which is also known as Cinder. Object storage means a distributed, static storage, like backups and archives. Swift stores the data across the cloud on multiple drives and if a server or a hard drive breaks, the data is replicated on other active nodes. This ensures the integrity of the data even if the cloud consists of cheap common hardware instead of expensive equipment. Block storage is the opposite of Object storage; it means more dynamic type of storage like databases and expandable file systems. Users can attach blocks into server instances by using the dashboard, allowing them to manage their own storage needs. The storage can be unified with enterprise storage platforms and backed up by using snapshot

management. Snapshots can then be used for restoring data or creating new block storage volumes. (Storage, OpenStack)

JunkCloud

JunkCloud is SkyNest's own OpenStack cloud. It has been mostly built out of low powered computers that were either salvaged or donated for the project, hence the name JunkCloud. It is mainly used for researching how the cloud behaves, and testing FreeNest and other applications in a cloud environment. The project has its own Cloud team assigned to maintaining and researching JunkCloud and further developing new features to it. One example of the developed features is CloudNest -tool, which can be used for executing commands in the cloud from external machines.

4 TARGET OF WORK

Problem

Since FreeNest is being developed using agile working methods and new features and tools are added and developed, all components need to be tested frequently to make sure everything works as intended. Every tool needs to be tested separately to make sure the tool still has everything working, and with other tools to make sure that installing one tool does not break any other tools or their functionalities. Some of the tools are also connected to each other enabling some additional features, thus the connections between those tools need to be tested as well.

Previously all testing was conducted by a testing team, where each team member picked a suite full of test cases and used the instructions in the test cases to run the tests and check the results. Even though there are only a few hundred test cases, it still took a lot of time to go through each one of them by hand. Considering that in bigger companies some projects can have tens of thousands of test cases, going through all of them manually would be a huge waste of time and resources.

Solution

The solution would be setting up a testing automation tool chain to make testing faster. Instead of having a team of testers going through the features of FreeNest, a group of virtual machines would go through it several times faster and more accurately. New tests could be created by the developers at the same time when they are working on new features and after that the scripts can be run whenever a feature needs to be tested. With the testing automation up and running there would be no need for a testing team, which has not been active in months at the time of writing this thesis.

5 ARCHITECTURE OF FNTC

5.1 Development

When the project started in February 2012, only a few requirements were set: TestLink should be used as the test management tool since it is already in FreeNest and widely in use, and Robot Framework should be used as the testing framework. It started as a research about how they could be integrated together so that when the user presses the

“Execute and save results” -button in TestLink, it starts Robot Framework automatically and saves the results in TestLink. When no ready-made solutions were found, the development of TestLink - Robot Framework -integration was started and Python was chosen to be used as the programming language.

After the first version that could run the tests was completed, Git was added to the requirements to be used as a test version control system. It was chosen because of the same reasons as TestLink, It is widely used and already integrated in FreeNest. Even though it was added to the requirements at the early stages of the integration, it was not implemented before the core of the integration was further developed and refined.

During May 2012 the integration was chosen to be included in Vaadin's MIDEaaS -project, which means Mobile IDE as a Service and is part of Tivit's Digital Services -program (Digital services SRA, Tivit, December 2011). The software they are developing, Arvue, allows users to design, create and test their own mobile applications using a web browser, and Testlink-Robot Framework -integration was chosen to handle the automatic testing. Mikko Ojala joined the team to work on the integration for MIDEaaS and mainly helped to rethink the structure of the integration for better modularity and develop new features like logging, configuration and the plug-in system. At the same time the integration name was changed to FNTC, FreeNest Test Controller.

The current version of FNTC is almost completely class based and most of the parts can be swapped to similar ones with some modifications. When the testing run is started, it asks for testing data from TestLink's custom fields and decides the testing engine by the received data. It comes with two testing engines by default, the simpler Robot engine

and more advanced Grid engine. FNTC can also pull the newest test versions from the Git repository just before running the tests making sure that no outdated tests are used in the testing run. See figure 13 in Appendix 2 for FNTC's class diagram.

5.2 Environment elements

The tool chain is spread across several virtual machine instances for better flexibility and performance. There are four types of machines used in the tool chain, each one having their own tools and configurations to serve their purpose better. The purposes of the virtual machines have been defined more accurately since the original plans and a new type of virtual instance, Grid Slave, had to be added into the tool chain, but the original presentation picture about the architecture of the test automation is still mostly accurate. See figure 11 for the original architecture plan.

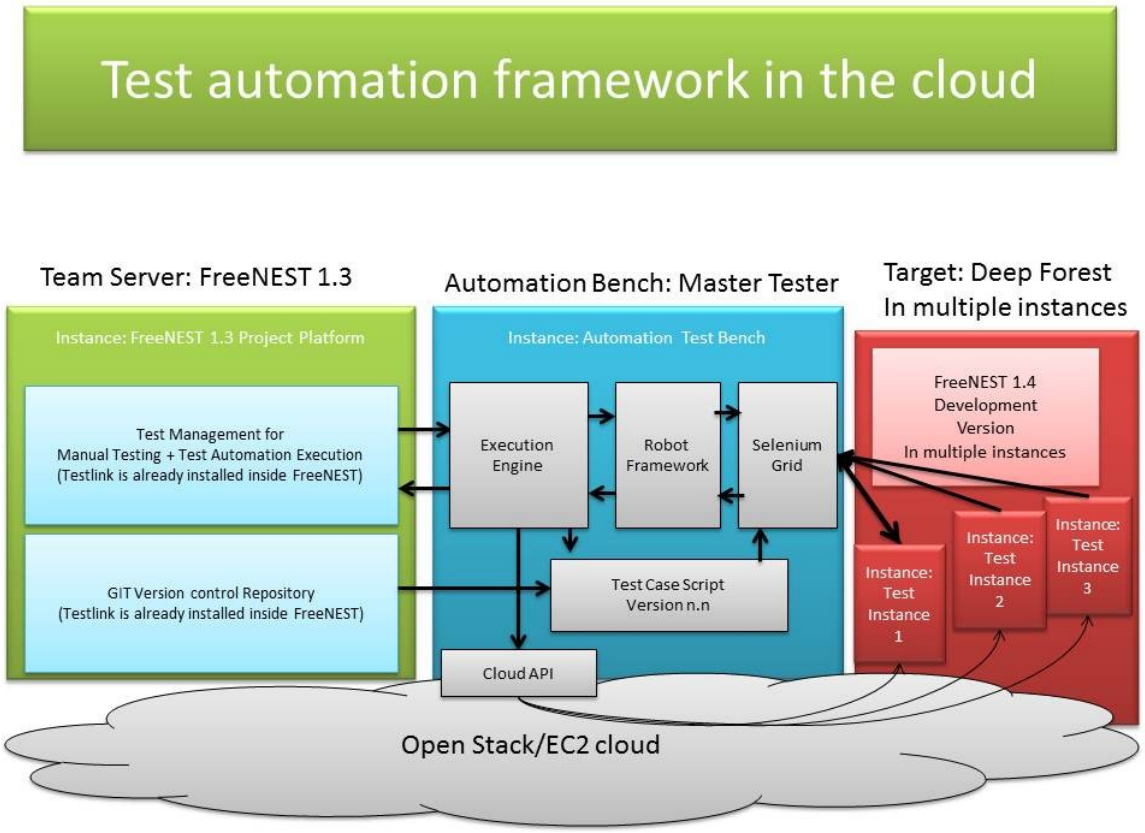

FIGURE 11. The original testing automation architecture plan (Marko Rintamäki, Q1 2012)

Team Server

The first instance in the tool chain is called Team Server. Its purpose is to act as a base of operations where the testing team can keep track on, modify and execute the tests, thus it needs to contain a test management tool and a version control repository. In this case an instance of FreeNest 1.3 is being used as a Team Server, more precisely SkyNest's own FreeNest instance called Strongbow. TestLink and Git are included in the FreeNest installation, leaving only some configuration to be done in TestLink in order to enable the testing

automation. Team server is the only machine that is located outside the JunkCloud and there should be only one Team Server per one project.

Master Tester

The second type of virtual machines is called Master Tester. Its purpose is to process and run all the tests it receives from Team Server, or distribute the tests for slave machines to run if the testing framework allows it. The machine has Robot Framework, Selenium Server, Git and FNTC installed, but Robot Framework and Selenium can be replaced with the testing frameworks of tester's choice with some modifications in FNTC. Depending on the testing frameworks used and the complexity of the tests, Team Server and Master Tester are the only machines required in the tool chain.

Grid Slave

The third type of virtual machines in the tool chain are the slave machines that run the tests, Grid Slaves. They are the dumb slaves in the tool chain, they run the tests they receive from Master Tester and return the test results after they are done testing. The tool chain is designed in a way that there are several Grid Slaves, each one of them has one Deep Forest to run the tests against and they can be taken down if they are not needed. The connection between the slaves and Master Tester is handled using Selenium Grid, the hub being located in Master Tester and Slaves containing the nodes. When tests are executed, the hub chooses those slaves that are currently inactive and uses them for running the tests.

Deep Forest

The systems under testing are located in the last type of virtual machines which are called as Deep Forests. Their only purpose is to be on-line while tests are run against the systems installed in them and when the system becomes too corrupt due to all the testing, it can be taken down and replaced with a fresh system. All Deep Forest instances can have different configurations like operating systems or different tools to bring more variety and coverage into testing. In this case the system under testing is an instance of FreeNest 1.4, which is supposed to be installed right after launching the instance the first time. At the time of writing this thesis the installation has to be done manually with command “sudo apt-get install freenest” in every instance of Deep Forest; however, it is planned to be automatic in the future.

5.3 Components

5.3.1 Core

The core of FNTC consists of two files: fakeXMLRPCTestRunner.php and fntc.py. The PHP file is only used as a middle hand, converting data from TestLink for FNTC and the other way around. The file used to be only an example of XML-RPC usage developed by the TestLink team. It acted as a fake server inside TestLink; when it received some test data from TestLink, it returned hard coded results depending on the test data it received. When the development of FNTC began, the hard coded results were replaced with a Popen-function that calls fntc.py. The same method is still used for running FNTC, but it will be changed in the future.

The python script is the main part of the core. It handles all the data received from arguments and TestLink API and chooses the testing engine and test scripts depending on the data it receives. The testing itself is handled by the testing engines and when the engine has done testing, FNTC takes the results and sends them back to Testlink.

FNTC comes with a simple logging feature. It launches a Python logging facility that writes logs about everything that happens in FNTC and its components for the user to keep track on. The logging system prints the exact time of the event, the class where the event took place, the type of the event and a short message that explains what happened. The type can be info, log, debug, warning, exception, error or critical, but only debug and critical are used in FNTC at the time of writing this thesis. Full logs can be found from testlink_robot_client.log, which is located in the FNTC installation folder.

The behavior of FNTC can also be controlled using a configuration file called testlink_client.conf. The data from the configuration file is loaded using PyYAML -library's load -function and the loaded data is handled as key – value pairs. The values are then assigned to variables for easier data handling. An example of loading the data is shown below:

```
# getting variables from config
f = open('testlink_client.conf')
conf = yaml.load(f)
f.close

SERVER_URL = conf['general']['serverURL'] + "lib/api/xmlrpc.php"
devKey = conf['general']['devkey']
vOutputdir = conf['general']['outputdirectory']
testdir = conf['general']['testingdirectory']
logdir = conf['general']['logdir']
```

The configuration file also contains some data used in the testing engines. Since every engine behaves slightly differently, the data they need is not consistent, thus the configuration file has separate sections for every engine with the engine specific data. The correct sections are then loaded inside the engines and the correct data is assigned to variables.

FNTC also has error handling throughout the script. If something goes wrong, for example if TestLink API cannot establish a connection to TestLink or an engine fails to load, the rest of the functions are aborted and a blocked result is returned to TestLink. The error message is then displayed in TestLink and in logs, making troubleshooting easier.

5.3.2 TestLink API

In order to successfully run the correct tests FNTC needs to ask for some additional data from TestLink. The information FNTC needs is the engine, runnable tests, the amount of times the tests are run and test failure tolerance, and those are given separately using the custom fields in every automated test case in TestLink. The only way FNTC can access that data is by using the TestLink API.

The API used in FNTC is TestLinkAPI.py, a python class written in 2011 by Olivier Renault and based on James Stock's testlink-api-python-client R7. It contains methods for connecting to TestLink's XML-RPC interface, some basic methods for verifying the functionality of the API and several methods for polling additional information from TestLink and creating new test projects, plans, builds and cases. There are also two additional methods for reporting test results and uploading attachments into TestLink, added by the author of this thesis. The method for reporting test results is shown on the next page.

```
def reportTCResult(self, tcid, tpid, status):
        """ reportTCResult
        Report test result directly into Testlink
        """
        data = {"devKey":self.devKey, "testcaseid":tcid, "testplanid":tpid,
        "status":status, "guess":True}
        return self.server.tl.reportTCResult(data)
```

The API needs two arguments for setting up the connection to TestLink: The server URL and an API access key. The server URL needs to point to the specific file in the TestLink installation, xmlrpc.php in <TestLink installation path>/lib/api -folder, which handles the connection on TestLink's end. The developer's key is user specific and needs to be generated in the user profile in TestLink, so it would be a good idea to have a separate profile for FNTC. Both the server URL and the access key are written in testlink_client.conf -file in FNTC installation and they are used automatically after FNTC has been set up properly. They are given for the API when the class is initialized and the API is fully functional after that.

At FNTC's current state the API is only used for polling the custom field information from TestLink. However, the API requires information for polling and some of the information might not be available before asking for it, therefore FNTC needs to execute five polls before it has all the data needed for polling the custom field values, and additional four polls for getting all the custom field values. An example about polling the custom field value can be found below.

```
client = TestLinkAPI.TestlinkAPIClient(SERVER_URL, devKey)

--- Other polls are required here for getting all the information ---

cfEngine=(client.getTestCaseCustomFieldDesignValue(prefix + "-" + tcidlist[0]
['tc_external_id'], tcinfo[0]['version'], sys.argv[4], "testingEngine", ""))

logger.debug('Got engine from custom field: %s', cfEngine)
```

5.3.3 Git wrapper

Git wrapper is a small class used for interacting with Git. The main purpose of Git in FNTC is holding a repository of all test case versions, testers are able to write and update their tests on their own computers, push the changes to the repository and the tests will always be updated to the latest versions just before running them. It has also been planned that certain test versions could be tagged using Git's tags and the Git wrapper would be able to pull those specific test versions when needed.

For now the Git wrapper consists of two parts: git_puller.py and git_wrapper.sh. The sole purpose of the python file is to run the shell script using the Popen -function of the subprocess and catch the possible exceptions that can occur during the process. At the current state the error situations are slightly unclear, which lead to disabling the error handling. However, a possible exception is unable to break anything, the only thing it affects is failing to pull the newest test versions and the failure can be seen from the logs as blank Git output. The pull -method in git_puller.py as it is now can be found on the next page.

```
def pull(self, testdir):
        try:
                self.logger.debug('Starting GIT process')
                gitpull = subprocess.Popen(["sh",
                "git_wrapper.sh"],cwd=testdir,stdout=subprocess.PIPE,stderr=s
                ubprocess.PIPE).communicate()

                #self.logger.debug('GIT output list: %s', str(gitpull))
                # check and raise an exception if errors occur , disabled for now
                if str(gitpull[0]).find == ' ':
                        raise Exception(str(gitpull[1]))
                else:
                        self.logger.debug('GIT output: %s', gitpull[0])
                        return "ok"
                except Exception as e:
                        # if something weird happens, a message will be
                        # returned and old tests are run
                        self.logger.critical('GIT process exception: %s', str(e))
                        return str(e)
```

The shell script is located in the base folder of the Git repository that contains all the test scripts. Its purpose is to run the actual Git commands inside the repository and use the correct SSH key for authenticating. The script uses SSH-agent for finding the correct key and then runs the Git command. In the future the alternate commands could be run in this same file based on additional arguments it would receive from git_puller. Here are the contents of git_wrapper.sh as they are now:

```
#!/bin/sh

ssh-agent bash -c 'ssh-add /var/www/.ssh/wwwdata; git pull'
```

5.3.4 Plug-in system

One of the key features making FNTC more flexible is the plug-in system. It allows users to use their own testing frameworks by creating their own engine plug-ins. This way FNTC can be relatively easily

extended to cover more frameworks and technologies than just Robot Framework and its libraries, and the engine can be swapped by just changing the custom field value in TestLink. The code for the plug-in system was found from a tutorial “Python Style Plugins Made Easy” at LuckyDonkey and it was slightly modified to fit in FNTC.

There are three things that are considered when looking for the right engine. The file that contains the engine has to be named in a specific way, it has to start with “engine_” and end with “.py”. After the file has been found, the class name inside it must contain the word “Engine”, in this example it is “exampleEngine”, and it must be a child class of the Engine -class. Other than that, the class can behave however it pleases and it can be located in any folder inside FNTC. Of course the class must have some specific methods for it to work correctly with FNTC, see the required structure of the class in Figure 12.

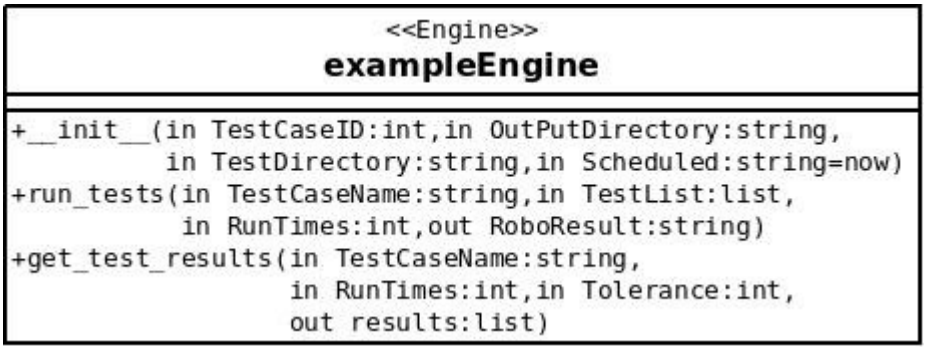

FIGURE 12. The required structure of the Engine -class.

The first thing the plug-in system does is looking for all the available engines. It starts from the folder where FNTC is located and continues looking through all the sub folders for files that match the name requirements. Here is the code of the part that walks through the directories looking for correct files:

```
engine_defined = False
cls = Engine
engines = []

path = "./"     # will start from the folder where the script is installed
for root, dirs, files in os.walk(path):
        for name in files:
                if name.endswith(".py") and name.startswith("engine_"):
                        path = os.path.join(root, name)
                        modulename = path.rsplit('.', 1)[0].replace('/', '.')
                        modulename = modulename.rsplit('.', 1)[1]
                        result = look_for_subclass(modulename)
                        if result == "error":
                                break
```

Every time a file with correct type of name is found, it is be checked for classes. The script walks the dictionaries inside the file to get to the last one and checks if their names have “Engine” in them. When a class with the name Engine is found, the script checks if the class is a sub class of Engine. If everything matches, the engine is added into the list of found engines. The code for checking classes can be found on the next page.

```
def look_for_subclass(modulename):
        try:
                logger.debug("Checking module: %s", modulename)
                module = __import__(modulename)

                # walk the dictionaries to get to the last one
                d = module.__dict__
                for m in modulename.split('.')[1:]:
                        d = d[m].__dict__

                #look through this dictionary for the correct class
                for key, entry in d.items():
                        if key == cls.__name__:
                                continue

                        try:
                                if issubclass(entry, cls):
                                        logger.debug("Found engine: %s", key)
                                        engines.append(entry)
                        except TypeError:
                                #this happens when a non-type is passed in to
                                #issubclass. Non-types can't be instances, so they
                                # will be ignored.
                                continue
                return "ok"

        except Exception, e:
                # if something goes wrong while loading modules, loading is
                # aborted
                logger.critical('Error while loading modules: %s', str(e))
                return "error"
```

After all usable engines are found, their class names are compared to the engine name that was polled from TestLink's custom fields. The engine name in the custom field must be a match to the class name of the engine, thus the engines in the custom field are named the same way as “exampleEngine”. Once a matching engine is found, it is loaded immediately and the script proceeds to testing. However, if there are no matching engines, engine_defined will remain false and the testing run will be aborted. The code for loading the engine can be found on the next page.

```
for e in engines: #scroll through all found engines and find the correct one
        if e.__name__ == cfEngine:
                engine = e(sys.argv[2], vOutputdir, testdir, sys.argv[8])
                engine_defined = True
                break
```

5.3.5 Robot engine

Robot engine is the default engine used in FNTC. It used to be a solid part of the main script, but when the plug-in system was implemented, Robot engine was moved into its own class. Its purpose is to simply use Robot Framework for running the given tests one by one on a single machine, process the test results and send them for FNTC to take care of.

The engine contains two important methods: run_tests and get_test_results. The run_tests -method takes three arguments; the test case name, a list of tests and the amount of times to run the tests, and uses them to build a command for starting Robot Framework. First it uses the test case name as the test suite name in reports and sets the non-critical tag in the case of non-critical tests. Then it goes through the test list making sure that the test scripts exist and adds each found script into the command. The command is finally executed using the Popen -function of the subprocess, Robot Framework runs the tests, the results are written into different folders and the engine will not proceed before all related tests have been executed for required number of times. Error handling is used whenever there is a possibility for something to go wrong, and the error message and a blocked result are returned to TestLink when an error occurs.

After Robot Framework has finished testing, get_test_results is called. It takes three arguments: test case name and amount of times to run the tests, which are used for finding correct output folders, and tolerance which is used for deciding if enough non-critical tests have passed. The method parses the test results from the XML-files Robot Framework generated and combines them into notes that are returned to TestLink. The notes include the number of times the tests were run, how many of the total, critical and non-critical tests passed, did the test suite pass and the reason if it did not pass and the results separately from each test case and test run. The method also generates links that lead to the original reports, but it only works if the tester has access to the same network where the testing environment is located.

There is also one experimental method called upload_results. It was used for packing Robot Framework's report and log files and uploading them as an attachment in the corresponding test case in TestLink, but it came with two problems. The first problem was a design flaw; while it does sound tempting to have the test results quickly at hand in TestLink and it would work even if the tester did not have access to the testing network, the attachments from hundreds of test cases and multiple test runs would fill up the database rather quickly and that could lead to performance issues. The second problem is a bug that was left unfixed due to the realization of the first problem. For some currently unknown reason the uploaded attachments get corrupted while uploading them, making it impossible to unpack the packages and view the results.

5.3.6 Grid engine

Grid engine is a more advanced version of Robot engine. Unlike Robot engine, Grid engine separates the given tests into blocks and executes them on separate virtual machines in parallel. Tests that can not be executed in parallel are not tagged and they will be executed afterwards on a single machine using serial execution. It then compiles the test results into one XML-file and the results are parsed from that file.

The base of the engine is almost the same as in Robot engine, but most of the functionality is handled using Parabot. Parabot was found from SeleniumLibrary's Google Code page “Advanced test distribution utilizing Selenium GRID”, and it was originally written by Thomas Klein in 2009 and heavily modified to fit in FNTC by the author of this thesis. Parabot.py was turned into a class that can be imported directly into the engine and the functionality was modified for more suitable test handling and test execution. Due to different ways of handling the tests and several other modifications, Grid engine is not fully functional at the time of writing this thesis; however the development will continue in the near future.

Instead of forming the command, the engine makes sure that the test scripts exist and send them to Parabot for execution. Parabot checks all the tests it receives, divides them into parallel tests and serial tests depending on the tags and the parallel tests are then further divided into test blocks. Parabot then starts a pybot instance for every test block, polls when each one of them is finished testing and kills the instances when they are finished. Serial tests, meaning the tests that would have conflicts with parallel tests, are then executed one by one on a pybot instance. After all tests are finished, the results are

collected and combined into one XML-file. The results are parsed the same way as in Robot Engine, but since Parabot disables Robot Framework's report and log files, links cannot be generated.

5.4 Installation

At the current state the installation process can be very long and tedious for a new user. It requires installing several tools, setting up connections and manual configuration file editing on three different machines, and multiple test runs before there is even hope for FNTC to work as it should. This problem could be fixed by carefully building Debian packages that would automatically handle the installation on all machines, but until then, the installation will be the hardest part in using FNTC. The instructions can be found in Appendix 4, however it should be noted that they might not be perfect.

5.5 Usage

Testing is started by creating a new test case in TestLink. It is recommended that all fields are filled to clarify what is supposed to be tested in the test case, however only the test case name and custom fields are required for FNTC to work correctly. The test case name should not contain any spaces since they break the parameters and the test run will fail due to incorrect data. Once all fields are filled, the test case should be added to the test plan for it to show up in test execution.

FNTC is started by clicking the “Execute and save results” -button in the test case in TestLink. TestLink will use XML-RPC -protocol for contacting FNTC and it sends eight parameters for it: Test case name,

internal ID, version ID, test project ID, test plan ID, platform ID, test build ID and execution mode. These are received by fakeXMLRPCTestRunner.php, which passes them for FNTC. After FNTC is launched, it will use the parameters for polling the remaining data from TestLink by using TestLink API. All results are returned in lists and it will take several polls before FNTC has all the data required for running the tests.

Once all data has been gathered, FNTC makes sure all test scripts are up to date by using the Git wrapper. At this point it would also create new Grid Slaves by using the CloudNest -tool, since it already has sufficient data for calculating the need for new nodes. CloudNest would ask OpenStack for running instances and it would create new ones if there are not enough nodes online. Each node would register itself automatically to the grid hub and FNTC would not continue before CloudNest has verified that all nodes are online. However, CloudNest is not implemented yet.

After updating the tests and setting up nodes, FNTC looks for the correct engine plug-in by comparing the class name to the custom field value that was polled from TestLink. In this case Robot engine will be used for running the tests. It launches Robot Framework and gives the correct tests for it, which will use Selenium Grid for running the tests against Deep Forest, the test instance of FreeNest. Even though Grid has the capability to run the tests using several nodes, in this case only one node is used. After Robot Framework has finished testing, FNTC uses the same engine for getting the test results. The results are returned as a list and they will be passed on to fakeXMLRPCTestRunner.php and then to TestLink, which updates the test case with the results.
In case errors happen during some part of the execution, FNTC will

always wait for a result string from classes and other parts. If everything is in order, the string contains only a word “ok”. If the string is not “ok”, it is assumed that something is not working as it should and the string is returned to TestLink as an error message along with a blocked result. The whole process can be seen in Figure 14, which is located in Appendix 3.

6 RESULTS

6.1 Current state

The testing automation built in this project was mostly finished in time. Some of the features did not get further than the planning phase and some features were left unfinished; however, the basic functionality is works properly. The core parts of the automation were built to be relatively modular, thus creating and implementing new features should be easy. If there is need for other testing frameworks, the users can create their own plug-ins and new frameworks are taken into use when the plug-ins are loaded. The testing automation can be used as it is for any kind of testing Robot Framework and selenium are capable of, and since Robot Framework and Selenium are sufficient enough for testing FreeNest 1.4, it can be used for internal software testing in SkyNest, provided that the developers get some training for creating new test scripts.

It was also noticed during the project that moving from manual testing to automatic testing is not easy or quick. The project can contain hundreds of tests and even when they are simple tests, converting them into automatic tests is a huge and very time consuming task. Even when the test automation is taken into account at the beginning

of the project, additional measures are still needed. The developers need some training to be able to write the testing scripts for their code and extra knowledge is also needed for using and maintaining the testing automation. The test automation tool chain contains more tools than just one and if one of them stops working, the maintainer will need to know where to look in order to find and fix the problem quickly.

6.2 Future improvements

While FNTC works well enough for it to be used for internal testing in SkyNest, it can not be considered as a valid testing automation solution yet. Since the author of this thesis did not have much experience of Python when the project started, most parts of FNTC could be built differently and more efficiently. The core could be written to be truly modular since some parts are currently hard coded to support only specific tools, TestLink API and Git wrapper being examples of this. TestLink API will not have any use if FNTC is not used with TestLink, and Git should be replaceable with Subversion if the user prefers.

TestLink could also be modified to better support testing automation. The interface looks messy when the results are returned, notes being just raw text without any kind of formatting unless the user realizes to press one small button for the formatted notes to show up. The interface does not give any kind of feedback about when the tests are being executed; it just waits for them to be finished and then updates the page with results. This can lead to the user thinking that nothing is happening and pressing the execution button again, possibly creating conflicts in the scripts. The interface should be more user friendly and provide sufficient feedback for the user to prevent misunderstandings.

It was planned that the testing automation could be integrated into JunkCloud in a way that it could control the cloud resources. Implementing the CloudNest -tool into FNTC would allow it to launch new Grid Slaves into the cloud depending on the amount of tests to be run. It would also be able to kill unneeded instances after running the tests. This would make FNTC more automated, removing the need to manually launching and setting up new nodes for it to use.

At the time of writing this thesis, Mikko Ojala has already started refactoring and improving the core parts of FNTC for better modularity and fit in MIDEaaS -project. Instead of using the fakeXMLRPCTestRunner.php, Twisted Framework is used for creating a server that receives the test requests and launches FNTC. It can be used as a daemon, meaning that the server starts up automatically when the machine starts, and it runs in the background listening for test requests. This makes it possible to have several FNTC instances running at the same time in separate threads, and it is not as strict towards the data it receives unlike fakeXMLRPCTestRunner.php. This way PHP can also be removed from the software requirements.

The installation process is also unnecessarily complex and has several possible points of failure if the user does not know exactly what needs to be done. That is why FNTC will be Debian packaged properly in the future, either by Mikko Ojala or the Cloud team. The file locations will be changed to more suitable ones, for example FNTC would be located in AdminUser's home folder instead of /var/www/ -folder, which can be considered as a security issue.

REFERENCES

Cloud Software Finland brochure. August 2011. Referred on November 8, 2012. http://www.cloudsoftwareprogram.org/rs/2226/6e620c3b-438c-425c-bfcc-a70731023c59/8b3/fd/1/filename/cloudbroch-aug2011-net.pdf

CodeProject. 20 March, 2012. What is software testing? What are the different types of testing? Referred on October 16, 2012. http://www.codeproject.com/Tips/351122/What-is-software-testing-What-are-the-different-ty

CodeThinked. June 30, 2009. What is Unit Testing? Referred on October 25, 2012. http://www.codethinked.com/what-is-unit-testing

FreeNest.org. 2012. Portable Product Platform. Referred 4.10.2012. http://freenest.org/about

Git. n. d. About Git, getting started. Referred on November 14, 2012. http://git-scm.com/about , http://git-scm.com/book/en/Getting-Started

Guru 99. n. d. What is System Testing? Referred on October 26, 2012. http://www.guru99.com/system-testing.html

IBM, n. d. IBM Cloud Computing: What is cloud computing? Referred on October 24 - 25, 2012. http://www.ibm.com/cloud-computing/us/en/what-is-cloud-computing.html

InfoWorld. n. d. What cloud computing really means. Referred on October 24, 2012. http://www.infoworld.com/d/cloud-computing/what-cloud-computing-really-means-031?page=0,0

JAMK. n. d. Projektit, SkyNest. Referred on October 31, 2012. http://www.jamk.fi/projektit/1233

LuckyDonkey. January 2, 2008. Python Style Plugins Made Easy. http://www.luckydonkey.com/2008/01/02/python-style-plugins-made-easy/

OpenStack. n. d. Software, Compute, Networking, Storage. Referred on November 12, 2012. http://www.openstack.org/software/

Python. n. d. What is Python? Executive Summary. Referred on October 3, 2012. http://www.python.org/doc/essays/blurb.html

RobotFramework-SeleniumLibrary. February 1, 2011. Advanced test distribution utilizing Selenium GRID. Referred on November 7, 2012. http://code.google.com/p/robotframework-seleniumlibrary/wiki/UseSeleniumGRIDwithRobotFramework

Robot Framework. 2012. Robot Framework Introduction. Referred on November 4, 2012. http://wiki.robotframework.googlecode.com/hg/RobotFrameworkIntroduction.pdf

Robot Framework. September 30, 2011. Robot Framework User Guide. Referred on November 9, 2012. http://robotframework.googlecode.com/hg/doc/userguide/RobotFrameworkUserGuide.html

Selenium Documentation. August 26, 2012. Selenium WebDriver, Selenium 1 (Selenium RC). Referred on October 24, 2012. http://seleniumhq.org/docs/

SmartBear. 2012. Why Automated Testing? Referred on October 10 – 24, 2012. http://support.smartbear.com/articles/testcomplete/manager-overview/

Software Testing Fundamentals. n. d. Gray Box Testing. Referred on October 17, 2012 http://softwaretestingfundamentals.com/gray-box-testing/

Software testing Fundamentals. n. d. Software Testing Levels. Referred on November 14, 2012. http://softwaretestingfundamentals.com/software-testing-levels/

TestLink. March 18, 2012. TestLink User Manual. Referred on November 14, 2012. http://www.teamst.org/_tldoc/1.9/testlink_user_manual.pdf

Tivit. December 15, 2011. Strategic Research Agenda for services. Referred on November 19, 2012. http://www.tivit.fi/file_attachment/get/Services_SRA_2011-12-14.pdf?attachment_id=57

Webopedia. 2012. Regression testing. Referred on November 1, 2012. http://www.webopedia.com/TERM/R/regression_testing.html

Wikipedia, n. d. Cloud computing, Integration testing, Manual testing, Software testing, System testing, Testing automation, Unit testing. Referred on October 10 – 26, 2012.
http://en.wikipedia.org/wiki/Cloud_computing ,
http://en.wikipedia.org/wiki/Software_testing

APPENDIX 1. EXAMPLE OF A ROBOT FRAMEWORK TEST SCRIPT

This test case verifies that the AboutFreeNEST -page in FosWiki works correctly. It logs in FreeNest as AdminUser, clicks the link in the Dashboard and verifies that the page is correct by checking if it contains text “About FreeNEST”. The test was originally created by Teemu Ojala and it has been used for verifying the testing automation functionality, hence the simplicity of the test.

```
*** Settings ***
Library         SeleniumLibrary  5
Suite Tear Down  Close Browser

*** Test Cases ***
WikiwordDashboard
    Open Browser  http://192.168.42.111/  ff
    Maximize Browser Window
    Input Text  username  AdminUser
    Input Password  password  adminuser
    Click Button  Log in
    Click Link  FreeNEST
    Page Should Contain  About FreeNEST
    Click Element  logoutBtn
    Close Browser
```

APPENDIX 2. THE CLASS DIAGRAM OF FNTC

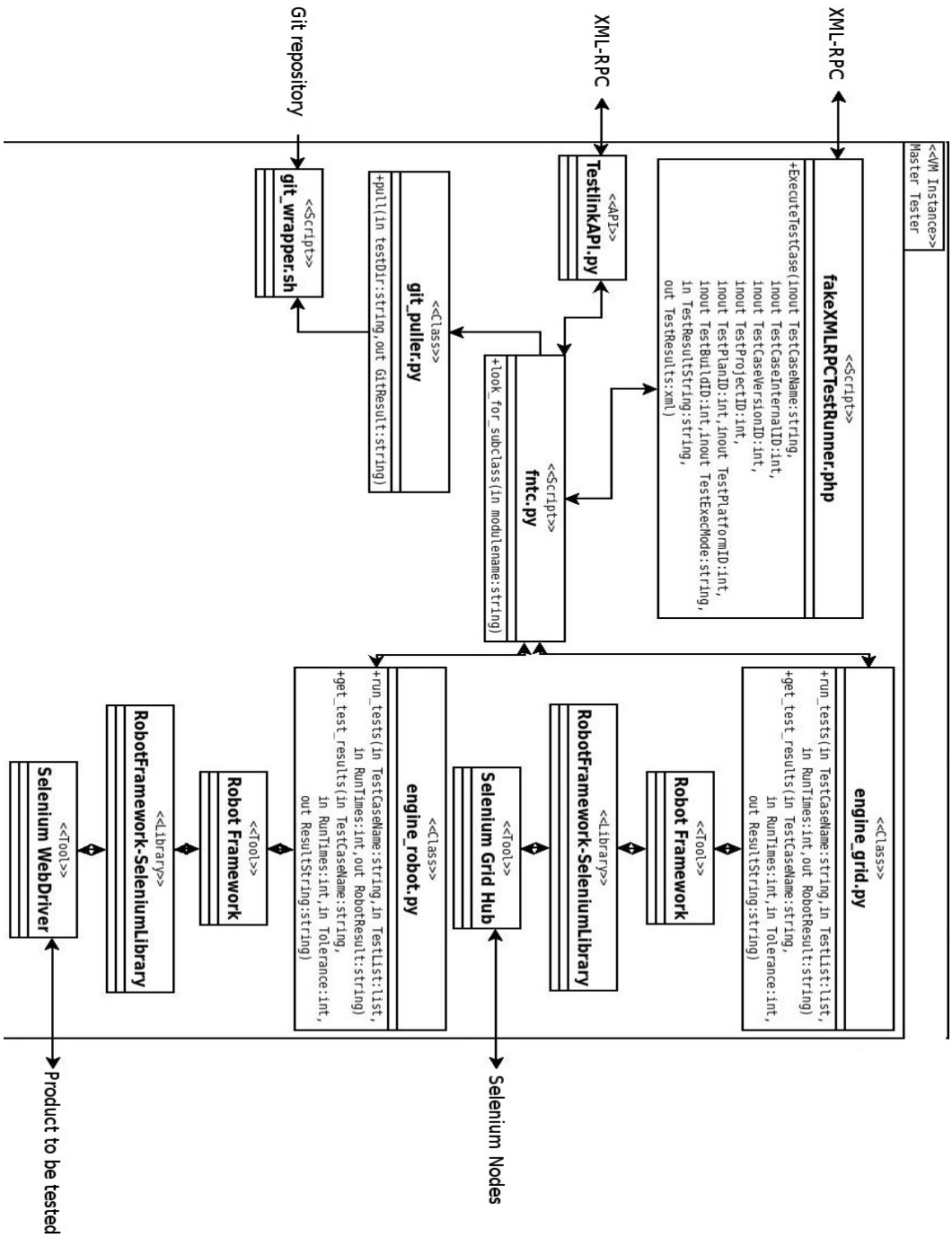

FIGURE 13. The class diagram of FNTC. The picture had to be rotated due to the large size.

APPENDIX 3. THE SEQUENCE DIAGRAM OF FNTC

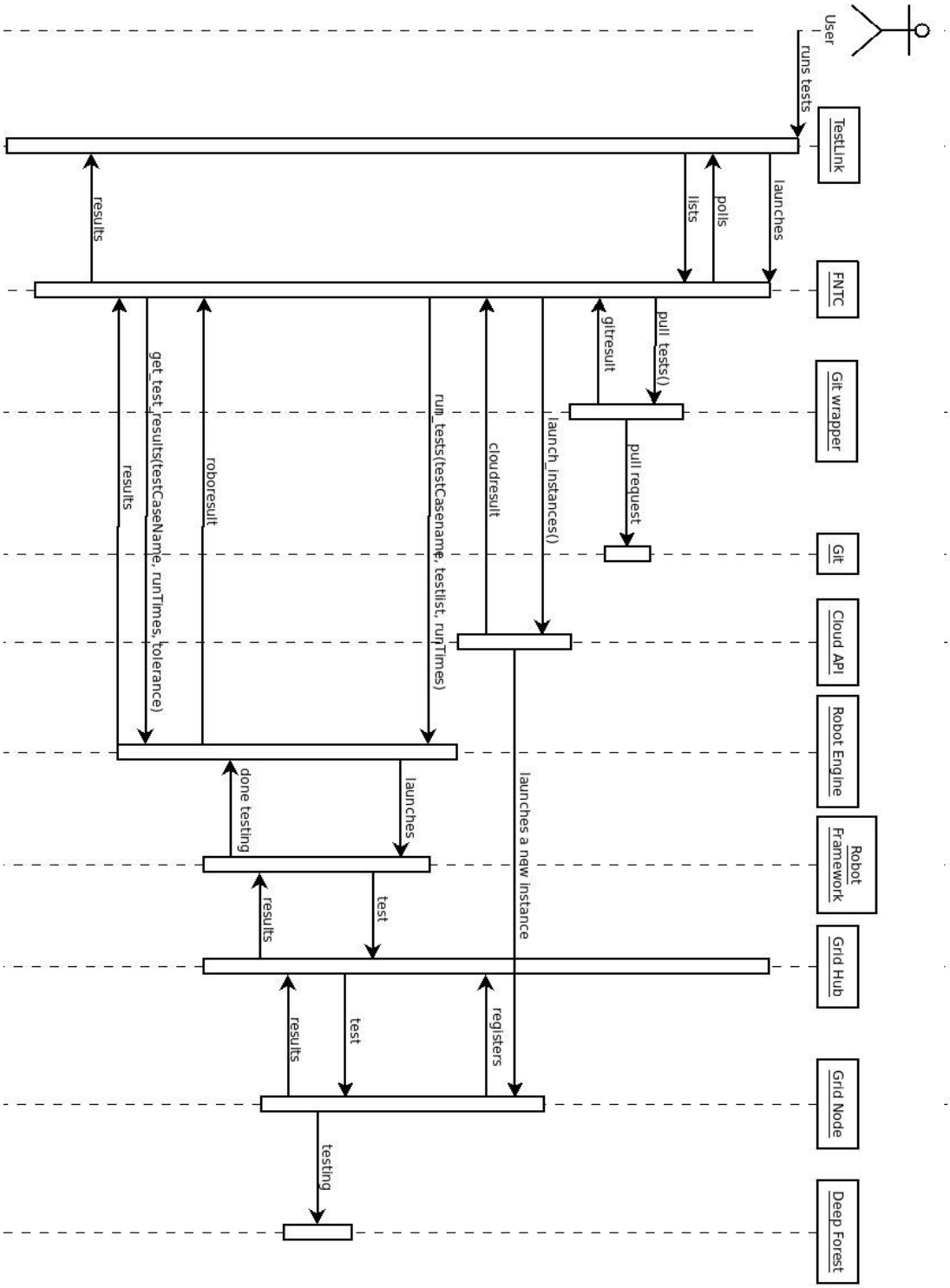

FIGURE 14. The sequence diagram of FNTC. The picture had to be rotated due to the large size.

APPENDIX 4. FNTC INSTALLATION INSTRUCTIONS

Team Server

The easiest way of setting up the Team Server is installing FreeNest 1.4 from the package repository, since it has TestLink and Git already installed and configured. FreeNest 1.4 is not yet released for public, thus it can only be installed from SkyNest own private repository. When having access to SkyNest's private network, this can be done by adding the repository into the following line into /etc/apt/sources.list -file:

deb http://192.168.42.104/packages/ubuntu/precise precise main

After the file has been updated, it is time to run three commands to install FreeNest:

sudo apt-get update
sudo apt-get install freenest-preseed
sudo apt-get install freenest

After running those commands FreeNest should be ready for use.

TestLink is the only tool in Team server that needs to be configured. The first this that needs to be done is editing the config -file. The custom_config.inc.php can be located in TestLink's installation folder, which is usually /var/www/ProjectTESTLINK. There are two lines that need to be added into the config:

$tlCfg->exec_cfg->enable_test_automation = ENABLED;
$tlCfg->api->enabled = TRUE;

These values can also be edited in config.inc.php but according to TestLink documentation, that is strongly discouraged. There are also three values that need to be increased in PHP in order to prevent timeout problems during testing. The php.ini can be located in /etc/php5/apache2 -folder, and the values that need to be edited are session.gc_maxlifetime, max_execution_time and default_socket_timeout. Their values need to be much higher, for example 360000 has worked well enough in this project.

Now TestLink can be configured using its own interface. First thing to do after logging in and creating new test projects and plans is creating new custom fields. There are three custom fields that are required for testing automation and four that are required for FNTC. The first three custom fields can be imported using the XML-file included in TestLink installation, the file can be found in ProjectTESTLINK/docs/file_examples and it is named as RE-XMLRPC-customFields.xml. The custom fields can be defined by clicking “Define Custom Fields” in TestLink project main page. The additional custom fields are testingEngine as a list, scriptNames as a test area, runTimes as a string and tolerance as numeric value. They need to be enabled on test specification and displayed in test execution, nowhere else. After the custom fields have been defined, they need to be assigned to the project by clicking “Assign Custom Fields” in the project main page. The last step in TestLink is creating the API access key. This can be done by clicking “My Settings” in TestLink's top bar and then clicking “generate a new key” -button under API interface. The key is then printed above the button and it can be added into the configuration file in Master Tester.

The last step in Team Server is creating a new Git repository for the test scripts. This can be done either by using the Control Panel in FreeNest 1.4 or by using Gitolite. In FreeNest it should be as easy as going in Administration, then Control Panel and then Git Admintools and adding a new repository under repositories -tab. Adding an new repository is not hard with Gitolite either, only a few lines need to be added into gitolite-admin/conf/gitolite.conf. Those lines consist of the repository name and user rights as shown below:

```
repo    robot_testing_scripts
        RW+     = @admin
        R       = gitweb
```

Master Tester

The Master Tester is the most complex machine in the tool chain and while it is mostly straight forward, there are some parts that most likely will not succeed the first time. The installation is started by installing the required software by using the command:

```
sudo apt-get install apache2 php5 openjdk-7-jre git
```

After they have been installed, Robot Framework, Selenium server and the library between them can be downloaded and installed. They can be found here:
http://code.google.com/p/robotframework/downloads/list
https://github.com/rtomac/robotframework-selenium2library/downloads
http://seleniumhq.org/download/
The newest versions at the time of writing this are robotframework-2.7.5.tar.gz, selenium-server-standalone.2.25.0.jar and robotframework-selenium2library-1.1.0.tar.gz. Robot Framework and SeleniumLibrary can be installed by unpacking them with command

“tar -xzf <package name>” and running the script inside them with command “python setup.py install”. Selenium Server is a runnable jar-file, thus it can be moved somewhere safe. PyYAML is also required for FNTC to work, it can be found here: http://pyyaml.org/wiki/PyYAML and it is installed the same way as Robot Framework.

Before FNTC can be installed, a folder needs to be created to hold all FNTC related files. The folder can be created in /var/www -folder with command “sudo mkdir Testlink-Robot” and FNTC can be cloned inside that folder from Strongbow's repository with command “git clone strongbow:testlink_robot”. This will only work if Git has been set up correctly and the user has enough rights for cloning the repository. Testing scripts are cloned into the same folder and one more folder is created for the output data with command “sudo mkdir robot_testing_output”. Now there should be testlink-robot, robot_testing_scripts and robot_testing_output -folders inside Testlink-Robot -folder.

Before FNTC can be used, it needs to be configured properly. First thing to do is creating the log-file for FNTC with the command “touch testlink_robot_client.log” in testlink_robot -folder. Testlink_client.conf needs to be updated with Team Server's IP address and the API key generated in TestLink, the rest of the options should work as they are by default. To ensure that FNTC can be run remotely, the owner of all the folders needs to be changed to www-data. This can be done with command “sudo chown -R www-data:www-data Testlink-Robot”.

Now Selenium hub can be started with command “java -jar selenium-server-standalone-2.25.0.jar -role hub”.

Grid Slaves

The Grid Slaves are the easiest virtual machines to set up in the tool chain. They only contain Firefox, Selenium Server for setting up the node, Java, which is required for Selenium, and VNC4Server, which acts as a virtual display. VNC4Server is not required if the machine is a desktop Ubuntu, but it is required for Ubuntu Server and it is useful for remotely checking the status of the node machine.

The installation should be started by installing Java, VNC4Server and Firefox:

sudo apt-get install openjdk-7-jre vnc4server firefox

Firefox requires a display before it can be used at all, thus starting the VNC4Server would be a good idea. It can be started using command “vnc4server”, it will ask for a password and finally give a number of the display. The virtual display can then be viewed from another computer using Vinagre. After starting Vinagre, the user can create a new connection by pressing “connect”, changing the protocol to VNC and giving the IP address and the port of the display. For example, if the IP is 192.168.42.104 and the display number is 1, the address would be “192.168.42.104:1”. Then Vinagre will ask for password and after that the connection should be working.

The next step is important for escaping the SSL certificates, since there are no keywords for them in Robot Framework, all tests get stuck in the certificate dialog and fail due to timeout. For that selenium needs a Firefox profile, thus Firefox profile manager needs to be started using

command “firefox -Profilemanager” in the command prompt visible in Vinagre. It should give a file path for the profile while creating it; it will be needed later on. The only way to get past the certificate is using the new profile for visiting the site and accepting the certificate manually so the choice will be remembered. Firefox can be safely closed after accepting the certificate.

The last step is starting the Selenium Server. The server can be downloaded here, 2.25.0 being the latest one at the time of writing this: http://seleniumhq.org/download/
It does not matter where selenium-server-standalone-2.25.0.jar is saved as long as it can not disappear; AdminUser's home folder was used in this case. Now the server can be started using command:

DISPLAY=:1 java -jar selenium-server-standalone-2.25.0.jar -role node -hub http://192.168.42.104:4444/grid/register -firefoxProfileTemplate “/home/adminuser/.mozilla/firefox/f02n6b5q.Selenium"

“DISPLAY=:1” defines the display that is used for running Firefox, “-role node” defines the role of the server, “-hub (IP:port)” defines the location of the hub and “-firefoxProfileTemplate (path)” is the profile that is used for running Firefox. After the connection has been established, the tool chain is ready for testing.