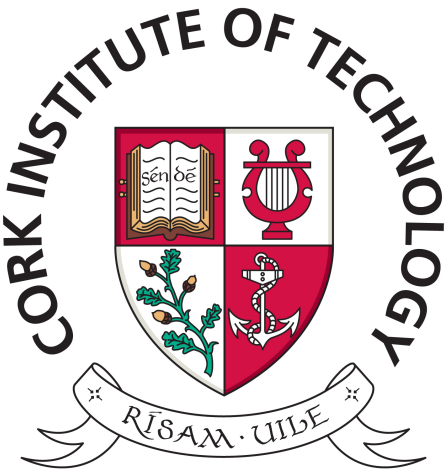


A Comparison of Physical, Virtual and Containerized Environments for UI Test Automation

by

Kevin Geary

This thesis has been submitted in partial fulfillment for the degree of Master of Science in Cloud Computing

in the
Faculty of Engineering and Science
Department of Computer Science

May 2017

Declaration of Authorship

I, Kevin Geary, declare that this thesis titled, 'A Comparison of Physical, Virtual and Containerized Environments for UI Test Automation' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a postgraduate masters degree at Cork Institute of Technology.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Cork Institute of Technology or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

CORK INSTITUTE OF TECHNOLOGY

Abstract

Faculty of Engineering and Science
Department of Computer Science

Bachelor of Science

by Kevin Geary

Agile development methodologies are emerging as the dominant practice in the software industry. One of the driving forces behind its adoption is its ability to respond quickly to changing customer requirements throughout the development lifecycle. A key tenet of this adaptability is achieved by having short iterations of evolving incremental fully-tested functionality, with constant customer feedback and engagement. Such a model necessitates that test activities work within these timeframes. Tests need to execute quickly and reliably within the iterations in order to provide the customer with the data that they need. This is a challenge for test teams and it is a process which strongly lends itself to automated testing. Test teams, short on resources such as time and hardware, need to embrace new ways of testing in order to meet these challenges posed by Agile. This study proposes to build an Automation Test Framework using the Selenium suite and to run tests across three distinct environments in order to see how each platform performs. The first environment is a physical machine with native Linux OS, the second environment is hypervisor with para-virtualised virtual machines, and the third environment is a hypervisor with container-optimized virtual machines. The objective of the study is to determine which environment makes the best use of limited hardware resources when teams are faced with decisions on how to best structure their test framework.

Acknowledgements

I would like to thank Dr Donna O'Shea for her advice, encouragement and expertise. I would also like to thank my employers at Dell EMC for providing the hardware outlined in this thesis. Finally I would especially like to thank my family, my wife Deirdre, and children Lia and Tomas who have always been a great support throughout this masters.

Contents

List of Figures

List of Tables

Abbreviations

SAN	**S**torage **A**rea **N**etwork
AUT	**A**pplication **U**nder **T**est
LAMP	**L**inux **A**ppache **M**ySQL **P**erl-Python-PHP
CLI	**C**ommand **L**ine **I**nterface
W3C	**W**orld **W**ide **W**eb **C**onsortium
DOM	**D**ocument **O**bject **M**odel
LXC	**L**inux **C**ontainers
ISO	**I**nternational **S**tandards **O**rganisation
Xvfb	**X** **V**irtual **F**rame**B**uffer
VNC	**V**irtual **N**etwork **C**omputing
HBA	**H**ost **B**us **A**adpater

Chapter 1

Introduction

Despite ever changing software development methodologies and models, every organization delivering software has various stages of testing that form an integral part of its development process. These types of testing are usually performed at different levels in the development processes and can therefore be described as *Test Levels*. The Institute of Electrical and Electronics Engineers (IEEE) Guide to the Software Engineering Body of Knowledge [1] defines three key Test Levels which are described in Table 1.1.

TABLE 1.1: Test Levels

Test Level Name	Description
Unit Testing	This refers to testing the specific functionality of a piece of code in isolation, usually at a very low level, such as a method or function.
Integration Testing	This is the processes of verifying the workings of the interfaces between the various components of the software program
System Testing	This is the process by which a complete fully integrated system is tested to ensure that it meets its orginal system requiremnts. Non-functional requirements are often tested at this point

A second categorization of software testing is defining tests in view of the objective of the test. These objectives will fall within the scope of the Test Levels. By stating the objective of the test explicitly it gives a greater degree of precision over the result of the test and "allows control to be established over the test process" [1, 54] . Table 1.2 lists some of the different types of testing objectives.

The focus for this study will be on automating Functional Tests for a User Interface. This was chosen as traditionally this has been the hardest layer of tests to automate. Cohn [2] describes a Test Automation Pyramid whereby Automated UI testing "is placed at

TABLE 1.2: Test Objectives

Test Ojectives	Description
Acceptance testing	Verifying that the system behaviour meets customer requirements
Alpha/Beta Testing	Testing the pre-released software at a customer site or bringing a customer on site to run tests. Problems are reported to the development team
Install testing	Verifying that the software can be installed and works correctly on various different environments
Functional testing	Testing the software to verify that it conforms to its specifications
Regression Testing	Testing the software to verify that older functionality has not been broken by the addition of newer functionality
Stress testing	Exercises the software at maximum loads
Performance testing	Verifying that the software meets its performance requirements, for example it must run smoothly once enough memory and CPU are allocated
Usability testing	Evaluates the softwares ease of use for the end-user

the top of the test automation pyramid because we want to do as little of it as possible". It is time consuming and tests can be brittle when faced with changing code. Functional testing itself is a black-box testing process.This means that it examines the functionality of the application without knowledge of the inner workings of the application[3]. It bases its test cases on the functional specification of the software under test. It can refer to verifying the functionality of newly delivered features as well as testing that previously delivered functionality still works correctly. These tests can be done either manually or automatically, however since the spread of the Agile development methodologies, with its short sprint times and incremental feature release, the need for test automation has increased considerably. The reason for this is that a key tenet of Agile is customer involvement, and in order to get feedback on the software as quickly and reliably as possible, teams need and rely on automated testing [4, 178]. As projects grow so do the test suites, requiring test engineers to find newer ways to run ever expanding numbers of tests as reliably and as quickly as possible, often with limited resources. An Australian survey on Software Testing practices found that over 75% of organizations allocated less than 40% of their development budget on software testing activities. Despite this, the survey also reported that four-fifths of these organizations overspent on their testing budgets leading the authors to conclude that software development organizations are not allocating realistic budgets to test [5]. Capgemini et al [6] states that in 2016 the average spend on Quality Assurance and Testing for an organisation is 31%, down 4% from the previous year. One of the reasons given for this is that organizations are uncertain

as to how to control the test budget given the changes in the test process under new paradigms such as Agile and DevOps [7] where test cycles are short and frequent. It can be concluded that there is therefore a need for test teams to find ways maximise their resources, both from time and hardware point of view, in light of these changes to the software lifecycle.

Virtualization technologies can provide one such way to create efficiencies in test runs when physical resources are limited or under-utilized. The physical hardware is segmented into logical virtual machines that each runs an Operating System and applications backed by the resources of the physical machine. Each virtual machine can run tests as an independent entity. This isolation is good from the point of view of test automation as it does not create a single point of failure in the framework. If one virtual machine fails, the others can continue to work and run tests. Another aspect of virtualization that is of benefit is that tests can be run on different operating systems at the same time. For example, in Cross Browser Testing [8] a User Interface may be rendered differently depending on the browser type and Operating System combination. It is often a requirement for test teams to complete Cross Browser Testing whereby the functional consistency of a web application is checked across multiple browser versions and across multiple Operating Systems .

Linux containers are a lightweight operating-system-level virtualization method used for running multiple isolated Linux systems(containers) on a control host using a single Linux kernel. While containers have been around for a number of years they have recently gained much more prominence since the advent of Docker [9]. Docker is an open-source program that easily enables a Linux application and its dependencies to be packaged as a container. One of the benefits of containerization proposes that you can pack many more containers on a host than you can virtual machines, so it is possible that this may lead to even greater efficiencies when running large suites of long running tests by amplifying the benefits already described in the virtualization section. As regards browser testing, it is possible that even more operating systems and browser combinations can be opened up as long as they can exist within a container. Therefore the test team is not constrained by the Operating Systems physically installed or those supported for install on the hypervisor. One of the perceived drawbacks of using containers is that the shared kernel prevents the hardware isolation that virtualization gives and that this can be a security concern, especially if running such a solution from a

Cloud Service provider. A solution to this is to run containers within lightweight virtual machines optimized for that purpose.

With the benefits of virtualization and containerization just described in mind, this study proposes to run a series of UI automated tests on a physical machine, a hardware virtualized environment and a container optimized hardware virtualized environment in order to evaluate how well the test framework performs on each environment.

1.1 Motivation

To the best of the authors knowledge, there is no performance evaluation of this kind in the specific area around how test automation performs using hardware virtualization versus a physical machine; and now that container optimized virtual machines are becoming a more prevalent technology it also possible to introduce that comparison into the mix. As previously described, the impact of Agile Development has led to a greater need to find ways to test software in less time, get quicker feedback and do it for less cost to the organization. The results of this study should be interesting to any Software Testers or Engineers who are looking to implement solutions to the problems posed by Agile.

For many test engineers the words "Test Automation" have often held a fear. Crispin et al [10] identified many problems faced when introducing automation including the "Hump of Pain", that being the steep learning curve for non-programming test engineers, as well as the fear of writing code for non-programmers. A second motivation for this study is that the final report may possibly dispel some of those fears for testers who have this mindset.

1.2 Research Question and Objectives

1.2.1 Research Question

Most of the previous studies, as will be described in a Chapter 2 [11] [12] have shown that there is some overhead incurred when moving from virtualized to containerized infrastructures. These studies are primarily conducted on Linux, Apache, MySQL, and

PHP/Python/Perl (LAMP) stacks between Virtual Machines. This study aims to add to that research body but specifically in the area of UI Test Automation, whilst also including a physical host in the comparison. The Research Questions are:

TABLE 1.3: Research Questions

Question	Question
How will the performance of a Test Automation suite be affected by being run on : a) A physical machine b) A virtual machine c) A container optimized virtual machine	What other factors should be considered, outside of performance, when deciding which environment best suits a Test Automation framework. The environments are: a) A physical machine b) A virtual machine c) A container optimized virtual machine

1.2.2 Research Objectives

With the research questions in mind, the following objectives have been formulated.

1. For a given Test Automation Framework, run a series of test scenarios, with each scenario introducing increasing levels of concurrent tests, on each test environment and record the pass rates for each test run.

2. For a given Test Automation Framework, run a series of test scenarios, with each scenario introducing increasing levels of concurrent tests, on each test environment and record the time it takes to complete each test run.

3. Formulate an opinion on the ease of use and the operational flexibility of each solution.

1.3 Executive Summary

The proliferation of Agile Development practices raises some interesting challenges for test teams. The need to create test automation to run quickly and reliably has arguably never been greater but it comes at a time when test budgets are shrinking and the demand for quality is as high as ever. There is an imperative on teams to make the best possible use of all of their resources. The first objective of the thesis is to create a Test Automation Framework using various components of Selenium[13] so as to conduct a

series of experiments and then analyse how this performs from a reliability and execution speed point of view across three different environments.

- The first environment will be a physical machine with Linux OS
- The second environment will be a virtual machine with a standard para-virtualized Linux OS
- The third environment will be a virtual machine that is optimized for running containers running a very minimal Linux OS.

The second objective of the thesis is to give an opinion on the ease of use and complexity of each solution from a configuration point of view. The research method will take the form of a case study as quantitative data is collected. Measurements of pass rates and execution times will be collected and analyzed and the final report will make recommendations based on this. There is also a qualitative aspect to the study as an opinion on the operational flexibility of the different environments and how they combine with the Test Framework will be submitted based on the user experience throughout the study.

1.4 Contribution

A survey of software trends in 2016 by the Atlassian company of over 18000 customers found that 56% of them were using containers specifically to spin up test, staging and production environment [14]. It is a huge growth area and this thesis contributes to the field of study surrounding comparisons between virtual machines and containers. There are a number of studies in the area but none that the author can find relating specifically to how Test Automation performs over these different environments. The principal contribution of this report will be a conclusion as to which type of environment is best suited to running UI Test Automation. A secondary contribution will give an opinion on the operational flexibility of each type of environment given the demands of running UI Test Automation for a test team.

1.5 Structure of This Document

Chapter 1 of this document describes the overall reasons for undertaking this thesis, defines the research question and motivation, and outlines the contribution it will make to the research. Chapter 2 will describe the necessary background research undertaken for this study. It will describe the technologies involved, which are virtualization, containerization, test automation and storage area networks. It will also give an appraisal of the current state of the art with respect to the research question. Chapter 3 describes the design and implementation of the research, including the research parameters and the method by which this implementation takes place. Chapter 4 details the results of the tests, and includes observations based on these results. It also gives an opinion on the operational flexibility of each solution. Chapter 5 describes the studies conclusion and findings from the study and details areas of that may be of value for future research.

Chapter 2

Background

In this chapter the background for the thesis will be described. This will include an overview of the current technologies in the area, the technologies used in the study as well as a review of the current research.

2.1 Background Research

2.1.1 Test Automation

Dustin et al [15] describes Software Test Automation as a means to automate software testing activities including the development and execution of test scripts, verification of testing requirements, and the use of automated testing tools. Agile Software Development methodologies, such as Extreme Programming [16], and especially Scrum [17], that are common today deliver functionality in short iterations, typically a week or two, which need to be verified quickly and often. Having lots of short iterations with new functionality being added continuously means that having automated tests becomes essential to having a smooth development process. It is simply not feasible to manually test new features and verify that older functionality still works in these timeframes. One way of addressing this issue is through the use of automated testing tools [18]. The purpose of this study is to construct and evaluate a test automation solution using one of these tools, which will run tests across three distinct hardware and software platforms in order to see which environment performs best in terms of reliability and speed. The

exact requirements that will be used to measure this reliability and speed will be expanded upon later in this report. There are various types automation test tools on the market that help test teams to build and execute suites of automated tests which can be run whenever a new build is delivered to the test team.

The type of test tool that a team chooses will critically depend on the type of Application Under Test(AUT). Examples include, Web Apps, Mobile Apps, Desktop Applications and CLI-based Applications. Some tools will not be suitable for certain types of application. This study proposes to evaluate User Interface Test Automation, against a Web Application that is used for managaing Storage Area Networks (SANs).

The tools considered were SeleniumWebDriver, Protractor and TestComplete and the features of these are compared in Table 2.1. A greater discussion on each takes place in the following subsections.

TABLE 2.1: Test Automation Tools Comparison

Feature	Selenium	Protractor	TestComplete
Test Development Platform	Cross-platform	Cross-platform	Windows
Application Under Test	Web Apps	Web Apps	Windows desktop Apps, Web Apps, Mobile Apps
Scripting Languages	Java, C#, Perl, Python, JavaScript, Ruby, PHP	JavaScript	JavaScript, Python, VBScript, JScript, Delphi, C++, and C#
Learning Curve	Medium	Medium	Medium/High
License Type	Open source	Open source	Proprietary
Cost	Free	Free	License Fee
HTML5 and Angular Support	Yes/Custom	Yes	Yes

2.1.1.1 Selenium WebDriver

Selenium is an open source Test Framework with several components. One of the components of the framework is the WebDriver API which very recently has become a candidate for recommendation as a World Wide Web Consortium (W3C) standard [19]. It provides a platform and programming language agnostic way to control and interact with the Document Object Model (DOM) [20] elements in Web Applications by communicating with the browser using the browsers native support for automation. It provides

support for most browsers and has a very active development community. WebDriver is the name of the key interface against which tests are written [21]. It has become something of a standard for Test Automation and many other solutions are now wrapping the WebDriver code to create their own tools and products.

2.1.1.2 Protractor

Protractor is an end-to-end test framework for AngularJS applications [22]. It is a Node.js program built on top of WebDriverJS. Tests are written in Javascript and are run against the application in a browser as previously described in the WebDriver section. It has a few advantages that should be considered when choosing a Test Automation tool. Its principal added value is that is knows about Angular so that the end user does not have to build in custom wait statements to wait for angular processing to complete, because Protractor knows when Angular is still busy and waits for it. It also has some Angular specific WebElement locators such as *model, repeater* and *binding*, over and above regular WebDriver. During initial investigations, a small test case was built to run some simple Protractor tests and it was found that a poller is running continuously on the Application Under Test(AUT) using the Angular `$timeout` function. By default Protractor waits for these timeouts to complete and this was causing tests to hang as the function kept polling. The developers at Protractor do not see this as a bug [23]. Their solution is for your product to use the Angular `$interval` service to do the polling, and in this case Protractor will work as normal as `$timeout` and `$http` will run to idle. This is not an option for this study as the AUT code is not available. There are two possible solutions/workarounds for this.

- Ensure that Protractor never waits for a `$timeout` to finish on the page using the `ignoreSynchronization()` function that Protractor has. It would also be necessary to use a `Thread.Sleep()` in conjunction with this for long loading pages [24]. From initial investigations this solution has been shown to work but in general using sleep calls in this way is bad practice writing automated tests [25].
- Rewrite one of the Protractor functions `waitForAngular()` in such a way that it will somehow deal with the `$timeout` never going idle. The return on investment for this is high.

2.1.1.3 TestComplete

TestComplete is a Test Automation tool from the SmartBear company. It has support for running automated tests on many different types of Applications, including Web Applications. It has many built in features to handle different kinds of HTML5 popups and dialogs which don't require the user to write code to handle such events, therefore leaving the user to concentrate on the tests [26]. From initial investigations, that being running some tests written against the AUT, it was found that the tool itself had its own learning curve with the various ways to locate objects, handle events and so on. Tests need to be written in a scripting lanaguage such as Javascript, Python or JScript which would require some time to become familiar with . On the test environment it was found that it used a lot of memory and sometimes hung when using the functionality that is provided to locate elements on the page. This necessitated closing the application by killing the process. This happened on numerous occasions. Also, there is a licence fee associated with the product.

2.1.1.4 Choice of Framework

Following careful evaluation it was decided, due to the relative maturity of API, its large and active development community, as well as the support for writing code in Java rather than a scripting language, to use Selenium as the Automation Framework for this study.

2.1.2 Containers and Virtualization

Containers and Virtualization are often pitted against one another as competing technologies. Both are key enablers in the various Cloud Computing service models[27]. However in recent years there has also been a trend to merge the capabilities into a solution that may provide the benefits of both. In this subsection we will look at Virtualization and Container technologies as a whole, as well as some of the specific solutions that will be used in this study. Figure 2.1. shows the software stacks four principal approaches to the deployment of software and applications.

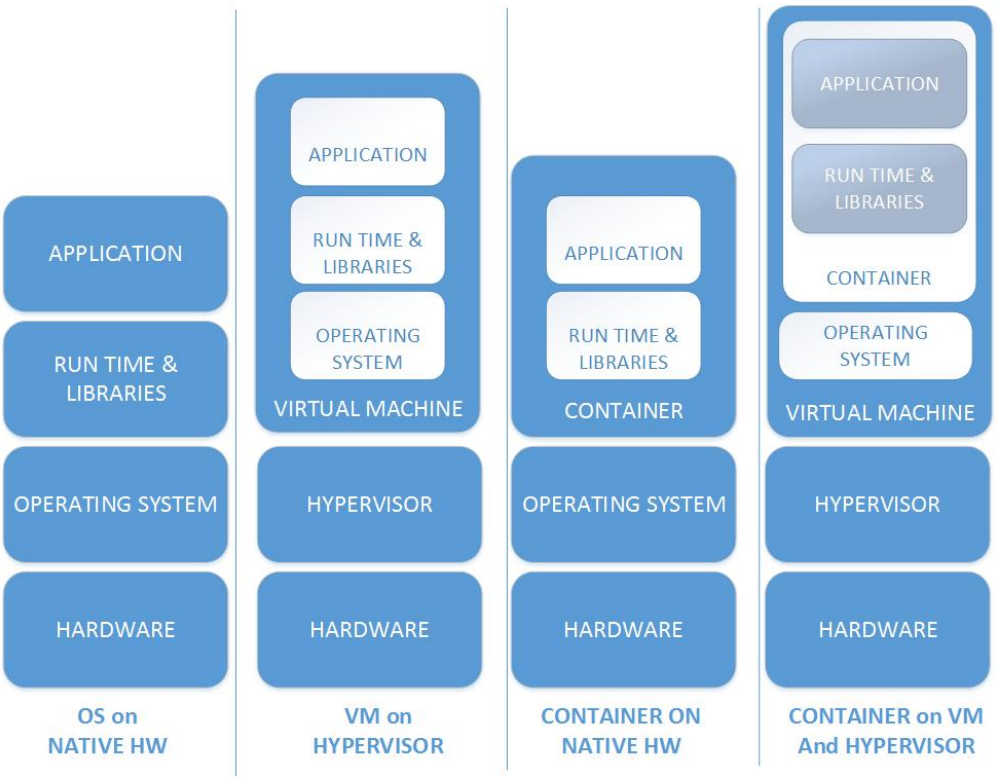


FIGURE 2.1: VirtualizationApproaches

The first shows the traditional native setup on a physical machine, then a virtual machine setup, a traditional containerized setup, and then a container within a virtual machine. In this study we will examine performance on the first, second and fourth silos.

2.1.2.1 Virtualization

Virtualization as a concept has been around since the 1960s when researchers at IBM were developing time sharing solutions for the Mainframe, however it was really the 1990s before the widespread adoption of what is known as Hardware Virtualization really took hold and initiated somewhat of a revolution in computing. This type of virtualization creates an abstracted management layer known as a hypervisor between the physical hardware and the operating system. It enables many virtual operatings systems, of different types, to exist on the physical machine and enables optimum use of the hardware resources. The technology is mainly focused on the Linux kernel and can be split into three categories: full-virtualization, para-virtualization and container-based virtualization [11]. Full virtualization does not require any virtual machine kernel

adjustments to optimize performance, para-virtualization does make minor adjustments to optimize performance, and container based virtualization does not use a kernel at all.

2.1.2.2 Containers

Containers, primarily in Linux, are often referred to as means to achieve lightweight virtualization[28]. Containers provide isolation and management of resources without any hardware emulation. Containers can run their own operating system, and are generally used to package an application and everything that it needs to run, but they share the same kernel. This is a key point as having a complete kernel process per small application can be seen as an unneeded overhead. There are three main components that facilitate this isolation.

(i) *chroot* is a utility and system call in Linux that allows us to isolate a process from the host filesystem by specifying a directory other than root. The process and its children are then forbidden access outside its own root directory while other programs can still see inside the new root.

(ii) *namespaces* enable each container to receive its own network communication and inter-process communication.

(iii) *cgroups* or Control Groups provides a way for admins to control access to the systems resources. It is then possible to set limits on things like CPU and Memory for specific containers. [29]

The traditional way of setting this up was to have a container engine such as Docker running on the physical host operating system as can be seen from the third silo in Figure 2.1 and from there containers would be created using that software. Another representation of this can be seen in Figure 2.2 where the Docker container platform runs as a process alongside the containers on the host.

Architecture Diagram of Container Stack on Physical Machine

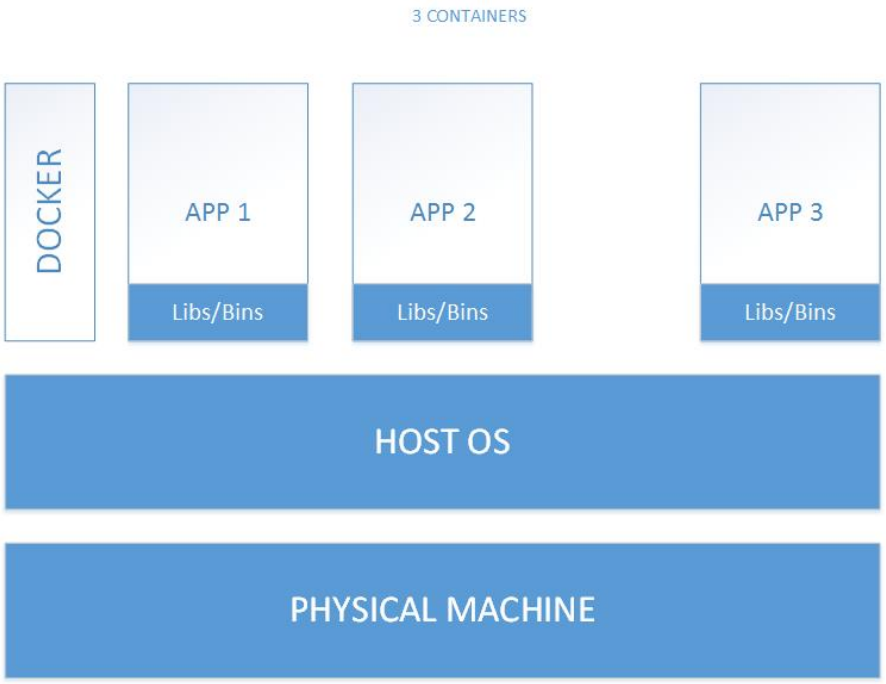


FIGURE 2.2: Docker Stack

Some recent trends in this area have been the creation of streamlined minimalistic Linux distributions such as CoreOS [30] and PhotonOS which are designed specifically with the idea of containers in mind [31]. The idea is to trim down the kernel and reduce all overheads as much as possible. This will be described in more detail in section 2.2.

2.1.2.2.1 Docker Docker is a technology that enables development, packaging and running of applications and all their dependencies within containers[9]. It enables separation of applications from their infrastrucure and as a result it can speed up the time between writing code and running it in production. Docker is written in the Go programming language and uses Linux Kernel features to deliver its functionality, some of which have been described in the container section[32]. Table 2.2 shows a comparision between Docker, rkt and LXC container runtimes. It is important to note that there isn't a one to one correlation between these technologies as they all offer something different. In fact Docker started life as a project to build Linux Containers (LXC) before evolving into its own container runtime environment. The table will compare criteria that are important in the pursuit of this particular investigation.

TABLE 2.2: Container Runtimes

Feature	Docker	rkt	LXC
Full System Virtualization	No (Application Level)	No (Application Level)	Yes
Platform Indpendence	Yes (w Docker Engine Support)	Yes	No (Cannot be easily ported)
Security And Isolation	Good	Good	Not Good
Industry Adoption	Very high	Low	Medium
Image Repository	Yes (Docker Hub, 100k images)	Yes (github - small)	Yes (image repository - small)
Support for Docker Images	Yes	Yes	Yes but needs config
License Type	Open source (Apache 2.0)	Open source	GNU LGPLv2.1+ license
Cost	Free	Free	Free
Built in Orchestration Engine	Yes (SWARM)	No	No

A key consideration for choosing Docker as the container platform technology for this study was its widespread adoption in the technology space. A Container Market Adoption survey in 2016 stated that 94% of those organizations using containers are using Docker, while for LXC this is 15%, and for rkt this is 10% [33]. Perhaps as a result of this, the Docker Image Repository, *DockerHub* is huge, with over 100,000 images and over 6 billion pulls [34]. The Linuxcontainers.org website provides a list of basic images for LXC. rkt does not have an image repository in the way that the others do but it is possible to run Docker Containers using rkt with some added configuration [35].

The rest of this subsection will describe how Docker implements the Linux kernel features that enable containerization.

Docker uses *Namespaces* to provide the isolation of workspaces for each container. Table 2.3 shows the namespaces created in Docker containers and what they do.

TABLE 2.3: Namespaces in a Docker Container

Namespace.	Namespace Description
The pid namespace	Process isolation (PID: Process ID)
The net namespace	Managing network interfaces (NET: Networking)
The ipc namespace	Managing access to IPC resources (IPC: InterProcess Communication)
The mnt namespace	Managing filesystem mount points (MNT: Mount).
The uts namespace	Isolating kernel and version identifiers. (UTS: Unix Timesharing System)

Docker also uses Control Groups, *cgroups*, to control how the host resources are shared. There are two *cgroup* drivers that come with Docker, legacy LXC driver and libcontainer [36]. The libcontainer driver is the default driver in the newest releases of Docker. In systems with high resource usage it is possible to set limits on workloads on a container by container basis [37]. This can prevent some containers hogging resources and even causing containers to be shut down due to resource starvation.

Union File Systems, also known as *UnionFS*, are a type of layered lightweight file system that have the benefit of being very fast. Docker uses this to layer Docker images. A Docker image consists of many layers to make up the final image. If a new image is needed that shares layers, it is only necessary to pull the differences [38]. One of the key concepts around this is that just one copy of an image is needed; say an Operating System image, which can be used as the basis for many containers. This gives us savings in terms of storage space and memory, and also speeds up container deployment times. [11].

Docker Images are templates or instruction sets from which running containers are created. Often images are based on other images, the most common example being that of an Operating System, but that has installed a WebServer. Users can build their own images but there is also a repository of well-known official images to be found at DockerHub website [39]. Included amongst these are a set of images for the *DockerSelenium* project which will be used in this study.

DockerCompose is a tool for running multi-container Docker Applications. It simplifies setup even further by letting the user specify in a configuration file which containers they would like to set up, how containers can communicate with one another and it also lets the user handle application scaling up and down in a very user-friendly manner. If the container images do not exist on the host, it will even download them from the repository. Full Examples of this setup can be found in Appendix B.

DockerSelenium is an open source project of Docker images for the Selenium Standalone Server, Selenium Hub and Selenium Node configurations. It supports Chrome and Firefox browsers [40].

2.1.3 Operating Systems

The next section contains a brief outline of the operating systems that will be used for this study.

2.1.3.1 Photon OS

Photon OS is an open source minimal Linux container host operating system that is optimized to run on VMWare platforms with a kernel specifically tuned for vSphere. It is part of a wider VMWare movement around vSphere Integrated Containers to make container solutions available to its customers and stresses that virtual machines and containers need not be competing technologies but rather can be complimentary ones [41]. It has a very small footprint, coming in three distribution formats, minimal International Organization for Standardization (ISO) image, full International Organization for Standardization (ISO) image, and Open Virtual Appliance (OVA). Photon OS supports Docker and rkt so the user is ready to go almost as soon as they have done some basic configuration. It also contains a yum-compatible package manger for life cycle management[42] which can be used to keep the software on the machine up to date.

2.1.3.2 RedHatEnterpriseLinux

Red Hat Enterprise Linux often abbreviated to RHEL is a distribution of Linux developed for the business market. The operating system supports various workloads and is designed for mission-critical enterprise computing and is certified by top enterprise software and hardware vendors. [43]

2.1.3.3 SuseLinuxEnterpriseServer

Suse Linx Enterprise Server often abbreviated to SLES is a distribution of Linux developed by SUSE. It is mainly designed for server, mainframes and workstations. It is a similar operating to RHEL in many ways with the main difference for most being the package management software. SLES uses YAST and RHEL uses YUM. One other difference that comes within the scope of this study is that SLES has support for LXC, while RHEL does not.[44]

2.1.4 Enterprise Storage Systems And Concepts

While not the subject of the research itself, the application under test is a management application for storage. The basic concept of these storage systems to take a bunch of storage devices (disks), and add some layers of intelligence so that the storage can be presented for use to hosts, which are attached to these systems.The data on these disks can protected in a reliable and performant way. This section will give a very brief overview of some of these technologies.

2.1.4.1 Storage Area Network

A Storage Area Network (SAN) is a high speed network of storage devices, which use the Small Computer System Interface (SCSI) standard for communication between connected devices. A variety of media can be used to transfer this Small Computer System Interface (SCSI) [45, 29] information with the two most popular being Fibre Channel (FC) or Internet Small Computer System Interface (iSCSI) [45, 133]. SANs provide storage at what is known as the Block Level. A piece of storage is carved out from the disks on the array and is presented to a host which can then format it in whatever way it wants to use as a hard drive. A host can connect directly to a storage array but it is most often connect through a SAN switch. [46]

2.1.4.2 Zoning

Fibre Channel (FC) Zoning is a switch function that segments the SAN into logical groups to restrict communication between devices. Ports are added to zones and within this communication is possible. There are different types of zoning [45, 115]. Port zoning refers to ports on the switch being zoned together, which means that anything plugged into this port will be a member of the zone. World Wide Name (WWN) zoning zones the world wide names of the endports, such as the Host Bus Adapter (HBA) [47] on the host, or Front End Apdater (FA) [48, 5] on the storage array. A WWN is a unique SAN identifier similar in concept to that of MAC addresses in networking. The zone is made up of these WWNs and it not concerned with the ports on the switch.

2.1.4.3 LUN Masking and Storage Groups

Logical unit number (LUN) Masking is another way of controlling what level of access a host has to the storage array. Even after zoning, several Logical unit numbers (LUNS) may be accessed through the same port on the storage array. LUN Masking enables each host to only view the LUNS which the administrator specifies. On EMC's VMAX Storage Array, this is achieved in an auto-provisioned way by using creating what is known as a "Masking View". Table 2.4 shows the masking view contents

TABLE 2.4: Masking with VMAX

Construct Name	Description
Initiator Group	Contains hosts initiator e.g. HBA wwn
Storage Group	Contains storage devices (LUNS)
Port Group	Contains the storage array front end port eg VMAX FA
Masking View	IG+SG+PG

Related initiators are grouped together, for example a host with multiple Host Bus Adapters (HBAs) used for redundancy , storage array front end ports are grouped together for the same reasons, and devices on the array are grouped together in a storage group.[49] A masking view is created by the combination of these constructs and the storage is provisioned to the host. It is important to be aware of this concept as the tests that are being run in the study are concerned with creating Storage Groups. A storage group is a logical grouping of devices from the array. LUN addresses are assigned to these devices once they are masked.

2.1.4.4 Unisphere For VMAX

Unisphere for VMAX is Dell EMCs GUI web based management application that allows you to configure and manage the VMAX Storage Array. In this study, this is the Application Under Test (AUT). Its features include the ability to provision storage, protect storage with local and remote replication as well as to monitor the array performance. It has a front end written in the AngularJS Javascript framework. This study is not concerned with the server and infrastructure code only that it communicates with an API known as Symmetrix Application Programming Interface (SYMAPI) which in turn that communicates with storage array. It also has a set of Representational State Transfer (REST) services which offer a large subset of the features of the UI.

2.2 Current State of the Art

There have been a relatively small number of studies in recent years comparing the performance of hardware based virtualization to container based virtualization and none that stand out doing a performance comparison for running test automation within such set-ups. The following subsections will describe the existing state of the art for such comparisons as well as our automation test framework.

2.2.1 Containers and Virtualization

Scheepers [11] presented one such study comparing the results of a set of macro-benchmarks between the Xen hypervisor and Linux Containers (LXC), as well as a discussion on the operational capabilities of each. A XenServer 6.2 and CoreOS 324.3.0 with Docker 0.11.1 are set up on two identical physical machines. Each runs two Ubuntu 12.04 virtual machines. The first virtual machine operates the application server running Apache, PHP and WordPress. The second virtual machine functions as the database server running MySQL database and is filled with content provided by WordPress. The first benchmark focused on the application performance when the web server needs to service increasing numbers of concurrent requests. The main findings showed that the LXC setup was able to process up to four times more requests than the XenServer 6.2. It also showed that Xen takes more time to process a single request and that it started swapping memory before the LXC. The second benchmark was in regard to inter-virtual machine communication using SQL queries. A PHP script was used to query the database and it found that LXC experiences less networking and CPU utilization overhead than Xen. However a second test was constructed to test this communication when the generation of data consumed all available resources and in this scenario Xen outperformed LXC considerably. Scheepers concludes that this shows LXCs inability to successfully isolate resources. The paper also examines the operational and ease of use capabilities of each system in light of the rise of automated data centers. The three areas of comparison are provisioning, service discovery and high availability. Scheepers states that from a provisioning point of view, LXC solutions using Docker's ability to build container images make this a better option. XenServer can provide snapshots and images but that this lacks the simplicity of Docker. For service discovery both use metadata services but

CoreOS uses a clustered solution requiring some configuration, which can be a problem to debug when wrongly configured. CoreOS has since gone production ready and this may not be as much of an issue as it was then as the technology has matured. In the area of High Availability, Xen provides comprehensive set of features including live migration capabilities which CoreOs does not have. CoreOS being a minimal OS does boot a lot quicker giving it an advantage in terms of failover.

Aspernas et al [31] did a study to using macro benchmarks on a LAMP application stack over CentOS, CoreOs and Photon OS in order to evaluate how the choice of container host (including virtualized and non-virtualized) effects application performance. Results showed that there was 18-20 percent decrease in the number of answerable HTTPS Requests when running a container host as a virtual machine, which was an expected outcome given the extra abstraction layer added when virtualizing an operating system. They also ran two sets of SQL tests. Running a single SQL SELECT between containers running on virtualized operating system took longer due to the nature of it being virtualized, whereas the SQL INSERT tests found that it was the choice of operating system rather than whether the operating system was virtualized or not that was the limiting factor.

A study from IBM by Felter et al [50] questions the practice of deploying containers inside virtual machines and claims that the extra abstraction layer of hardware virtualization only ever adds overhead and does not provide additional benefits over deploying containers on physical machines.

This study used KVM and Docker and in response to this VMWare did a comparison study to do similar experiments with VMWare vSphere. The findings from this showed that running Docker containers in a vSphere VM compared with a native configuration, delivered near native performance in most of the micro benchmark tests with generally less than 5 per cent overhead. It also showed that running an application in a Docker container in a vSphere VM has a very similar overhead to that of running containers on a native OS on a physical server.[12]

2.2.2 Test Automation and Selenium

To the best of the authors knowledge, there are no empirical performance studies of this kind done using the Selenium Framework. There are several studies where contributors have created Test Frameworks for the purpose of showcasing a particular approach to Automated testing [45] [51], comparing Test Automation tools, including Selenium against each other [52] , or demonstrating the benefits of Test Automation in general [53]. These types of studies, whilst not performance comparisons on different environments, are relevant to this one in that they show how Test Automation can address some of the problems that come from moving to Agile Development. It is an evolving field that is increasing in importance. One interesting development is that the WebDriver API which is the backbone of the Selenium framework is currently at Candidate Recommendation stage to become a W3 Web Standard [19]. This API is also being used for functional test cloud services such as Sauce Labs and BrowserStack. Support for WebDriver is being integrated into numerous other products, such as Appium for mobile web app testing, Protractor for AngularJS testing and SmartBear's TestComplete and QAComplete, and the Sencha Test Framework.

Chapter 3

Design and Implementation

The objective of this chapter is to first provide an overview of the problem and the objectives of the study including the functional and non requirements of the Test Framework to be constructed. The second part of this chapter shows the various hardware and software architectures and configurations involved in the implementation of the experiments, and the final section shows how this implementation (and subsequent analysis) fits in within the research parameters and methodology.

3.1 Problem Definition

The study investigates how best to use physical hardware at the disposal of a team when there is a limit on the amount of hardware available due to budget constraints and a limit on the time that an engineer can spend on repeating the same tests given shortened time frames associated with modern software development models. The emphasis focuses specifically on UI Automation testing where the tests needs to simulate the actions of a user against a real browser. In such a scenario there is a need to start a real browser session per test, each of which has an associated CPU and memory overhead. Starting multiple browsers sessions in parallel and executing tests concurrently so as to get the most from the physical hardware is a crucial part of this exercise. In order to carry out these experiments, a Test Framework needs to be built which can run tests in this fashion. The following section details these requirements.

3.2 Test Framework Requirements

The functional and non-functional requirements in this case refer to what is needed in terms of the functionality of the Test Framework to be written in order to be able to conduct the experiments across the different test environments so that it is possible to evaluate its performance.

3.2.1 Functional Requirements

1. Write a test plan to test the "Storage Groups" feature of a the Unisphere for VMAX Management Application.
2. Create a suite of automated test cases that has the capability to start a browser session, simulate point, click and write operations like a human user.
3. Have a mechanism for executing the tests.
4. Have a mechanism for verifying test data.
5. Have a mechanism in the framework that has has the capability of measuring the performance of each environment in terms of passed tests and time that it takes to complete a test.

3.2.2 Non-Functional Requirements

1. Ensure that the framework can scale up to run test cases reliably with increasing levels of concurrency.
2. The framework should usable and intuitive with the target audience being test engineers looking to implement agile test automation

3.3 Solution Architecture

3.3.1 Technologies Used

Table 3.1 shows a list of the various technologies that were used in this study.

Software And Programming	
Selenium Webdriver	Version 2.53
SeleniumGrid	
Java Programming Language	Java 7/8
TestNG Test Framework	Version 1.6.9
Docker	
DockerSelenium	
Unisphere For VMAX	Version 9.0
Hardware	
Physical Host Machine	Cisco R210-2121605W
Cores	12 CPUs x 2.666 GHz
Chip	Intel Xeon CPI X5650 @ 2.67 GHz
RAM	32 GB
DISK	131 GB
Physical Host Machine	DELL PowerEdge R430
Cores	12 CPUs x 2.4 GHz
Chip	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
RAM	32 GB
DISK	550 GB
Storage Array	VMAX 200k
Cores	128 Cores
Chip	2.6 GHz Intel Xeon E5-2650-v2
RAM	1958 GB
DISK	4186 TB
FC Switch	cisco MDS 9222i
Port	4x1GE IPS, 18x1/2/4Gbps FC/Sup2
RAM	Motorola, e500v2 with 1036300 kB of memory
Processor	Board ID JAF1524CTMJ
Host Operating Systems	
VMware ESXi	5.1.0 build-3872664
Red Hat Enterprise Linux Server	release 6.7 (Santiago)
SUSE Linux Enterprise Server	11 SP4
PhotonOS	1.0
Other Operating Systems	
FC Switch	Cisco Nexus Operating System (NX-OS) Software 6.2(9b)
Storage Array	HyperMax OS 5977.1112.1113

Table 3.1: Technologies Used In this Study

3.3.2 Storage Area Network Architecture

Figure 3.1 shows a simplified layout of the Storage Area Network that has been built out. This is the environment where the tests cases which will be creating storage devices are run. Fibre Channel Adpaters (FAs) on the VMAX Storage Array connect to the switch. Two Host Bus Adapters (HBAs) connect to the switch. These adapters are

Fiber Channel Zoned enabling FC communication between the hosts and the storage array. Both hosts are running the ESXi hypervisor and have virtual machines installed on them. There are three instances of the Unisphere for VMAX application, one on each VM, which enables an end user to manage the storage array. Three instances are installed so as not to have a bottleneck for tests in the application code. The best practice document for Unisphere for VMAX recommend no more than ten concurrent connections but this study will show the limits of this concurrency are far greater. Having three instances means the experiments can be run without issue on the application side and that resource usage on the browsers client machines will be the limiting factor.

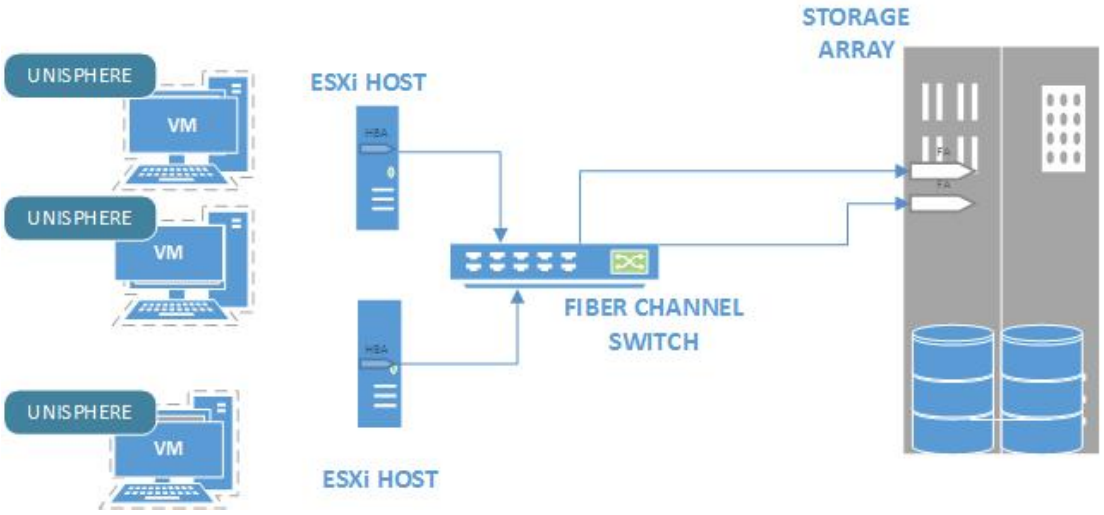


FIGURE 3.1: Storage Area Network layout

3.3.3 Virtual Machiness

For the purposes of this study we will be using VMWare's ESXi hypervisor which is a paravirtualized hypervisor.

Architecture Diagram of Stack of Para-Virtualized ESXi Hypervisor

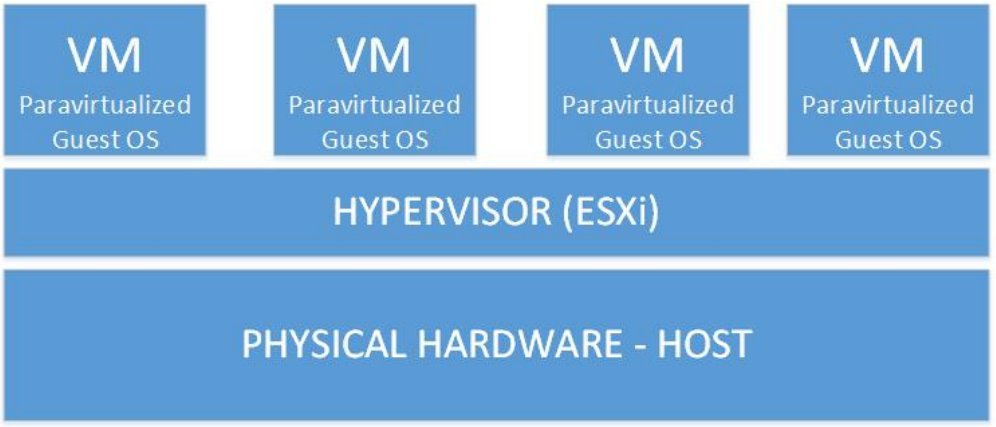


FIGURE 3.2: ESXi Stack

3.3.4 Selenium Grid Architecture

Figure 3.3 shows a simplified layout of a Selenium Grid. The Grid is the mechanism which facilitates distributed parallel testing in the framework. The basic architecture consists of single Hub which acts much like a test scheduler and is the central point into which tests are loaded. It receives information about a test, such as which browser to run it on, which Operating System to run it on, the IP address of the Application Under Test and so on from the test code. Selenium Nodes are registered with the hub. The Nodes are test machines which run the browser sessions. The Selenium hub selects which node should run each test based on the requirements of the test. For example, the test may need to be run on Chrome on Windows, and point to a particular instance of the AUT. Once a test reaches a node, it will carry out the test by starting a browser session, logging in to the application under test and carrying out the test. Tests are written in Java using the Selenium Webdriver API which enables calls to be made to the browser, simulating the actions of a human user.

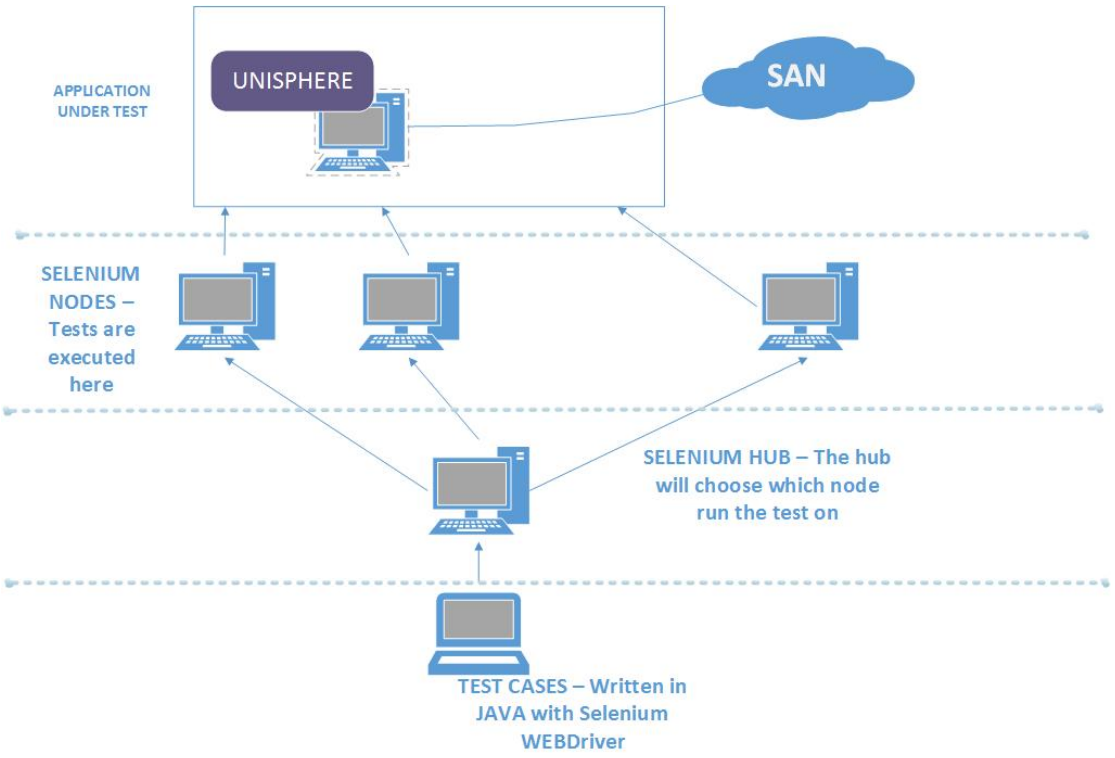


FIGURE 3.3: Selenium Grid Basic Architecture

Figure 3.4 shows the flow of information through the framework for a small test suite containing three tests. Some of these components will be described later but it is important to understand how this works. The TestNG XML file specifies the tests to be run and their configurations, as well as whether tests should be run concurrently. Table 3.2 shows an example of the type of data that would be supplied with the file.

TABLE 3.2: TestNG XML File Data

Run Tests Concurrently	Yes	
Threads to Run	3	
Test Name.	Browser	Application Under Test
Test 1	Firefox	10.10.10.1:8443
Test 2	Chrome	10.10.10.1:8443
Test 3	Firefox	192.10.10.10:8443

The test classes each contain one Test annotated by *@Test*, but there is a precondition for each class annotated by *@BeforeClass* to use the `WebDriverManager` create an instance of the `RemoteWebDriver` [54] which is an implementation of WebDriver that allows execution of tests on remote machines. These tests are then distributed by the

Selenium hub to a test machine that matches the configuration of the test. In the Example Test1 is sent to a Node that can support Firefox and runs its test on AUT1.

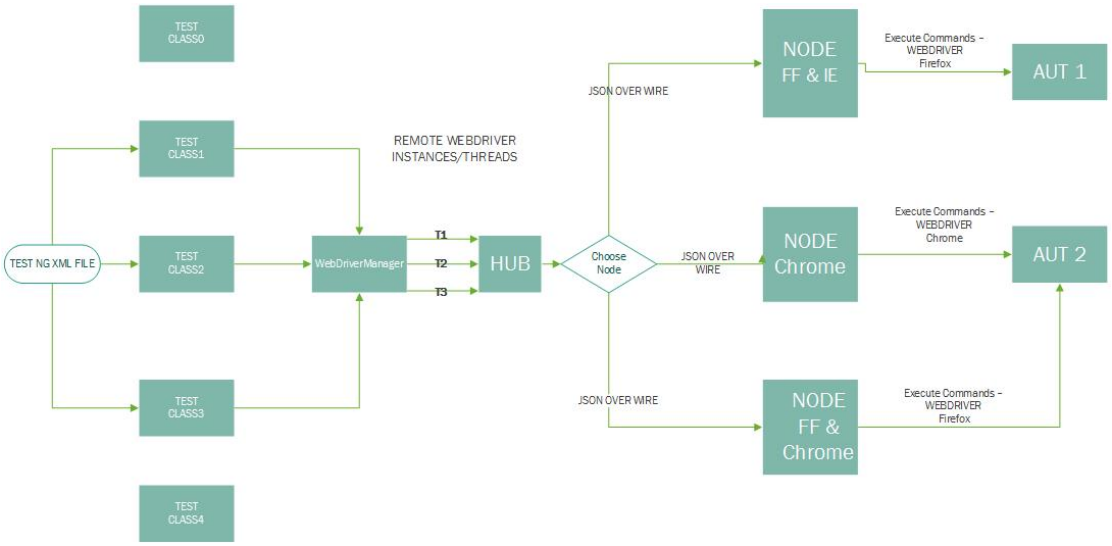


FIGURE 3.4: Information Flow Through the Grid

In this diagram there is only one Hub managing 3 Nodes, but it is possible that many more Nodes can be registered to this hub. It is also possible to have more than one Hub, as will be seen in one of the Docker configurations, and to split the tests across multiple Hubs, which in turn distribute tests to their nodes. An example of a TestNG.xml file is available in Appendix A.4. The important thing to note is that all of this configuration is done through the XML file. The threads themselves are initiated through TestNG, however the test framework code must be capable of running this, so drivers are created using the ThreadLocal [55] variable keeping each test threadsafe. This code is available in Appendix A.1.

The remainder of this section shows the five test environments that will be used in this study and how they each implement Selenium Grid. There are slight differences in the setup depending on the type of hardware/software involved. It can be assumed from the following diagrams that the virtual machines are sitting on top of a hypervisor. The hypervisor layer is not shown in these diagrams.

Figure 3.5 shows the layout of the Physical Machine which is running SLES 11. It has 32 GB RAM, 24 Cores and 550 GB disk space.

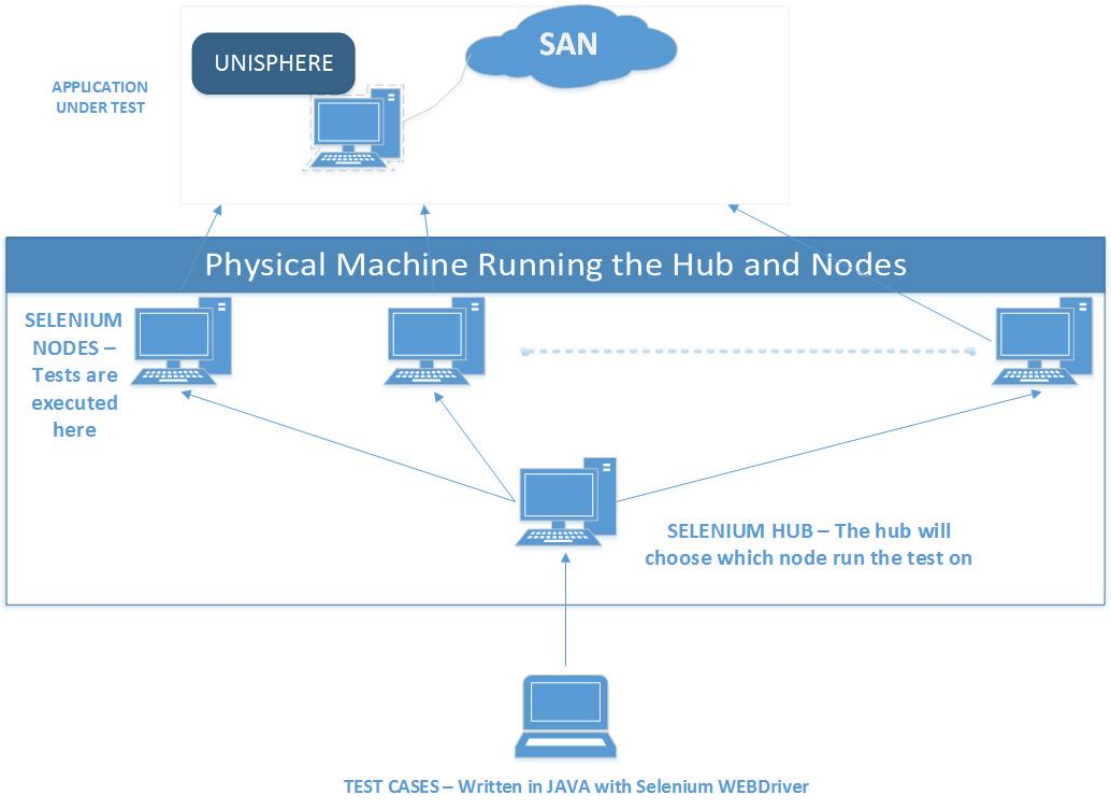


FIGURE 3.5: Physical Machine Grid Basic Architecture

Figure 3.6 shows the Virtual Machine running RHEL 6.7 with all host resources. It has 32 GB RAM, 24 Cores and 131 GB disk space.

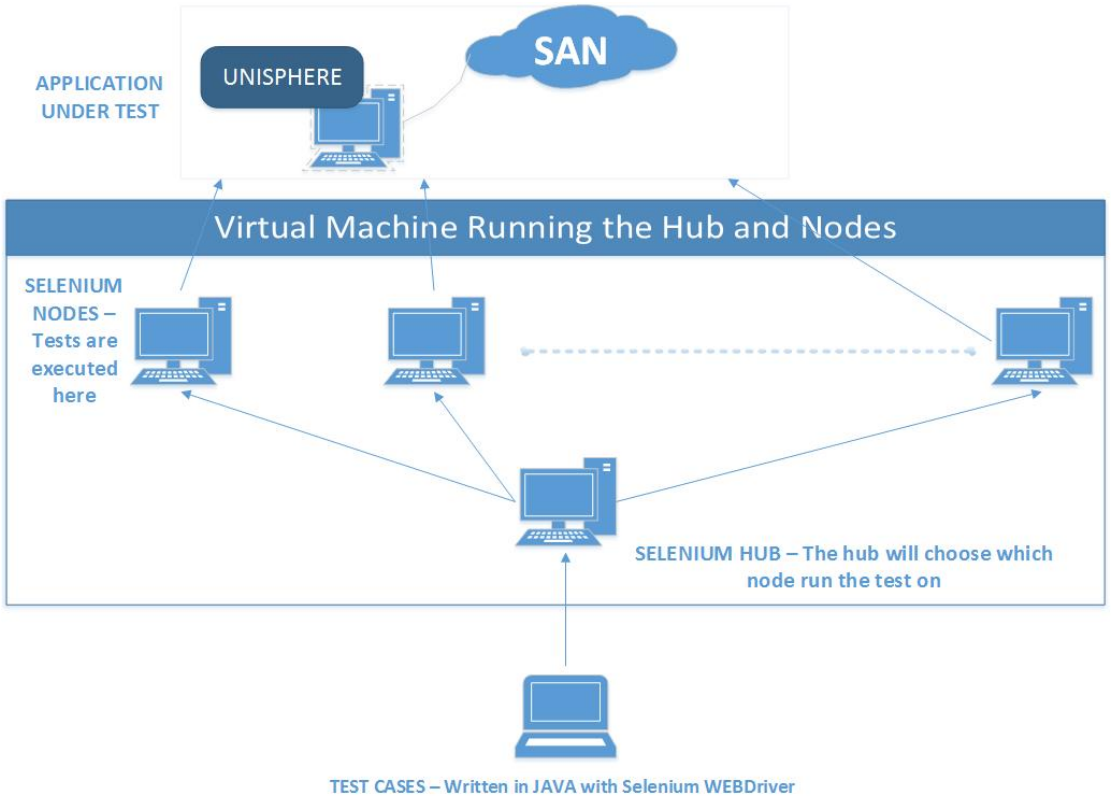


FIGURE 3.6: Standalone Virtual Machine Grid Basic Architecture

Figure 3.7 shows the Four Virtual Machines each running RHEL 6.7. Each VM has the same resources, 8 GB RAM, 6 Cores and 20 GB disk space. The remaining disk space is left free for vSphere snapshot management. An important point to note here is that one hub installed on one VM which manages all others.

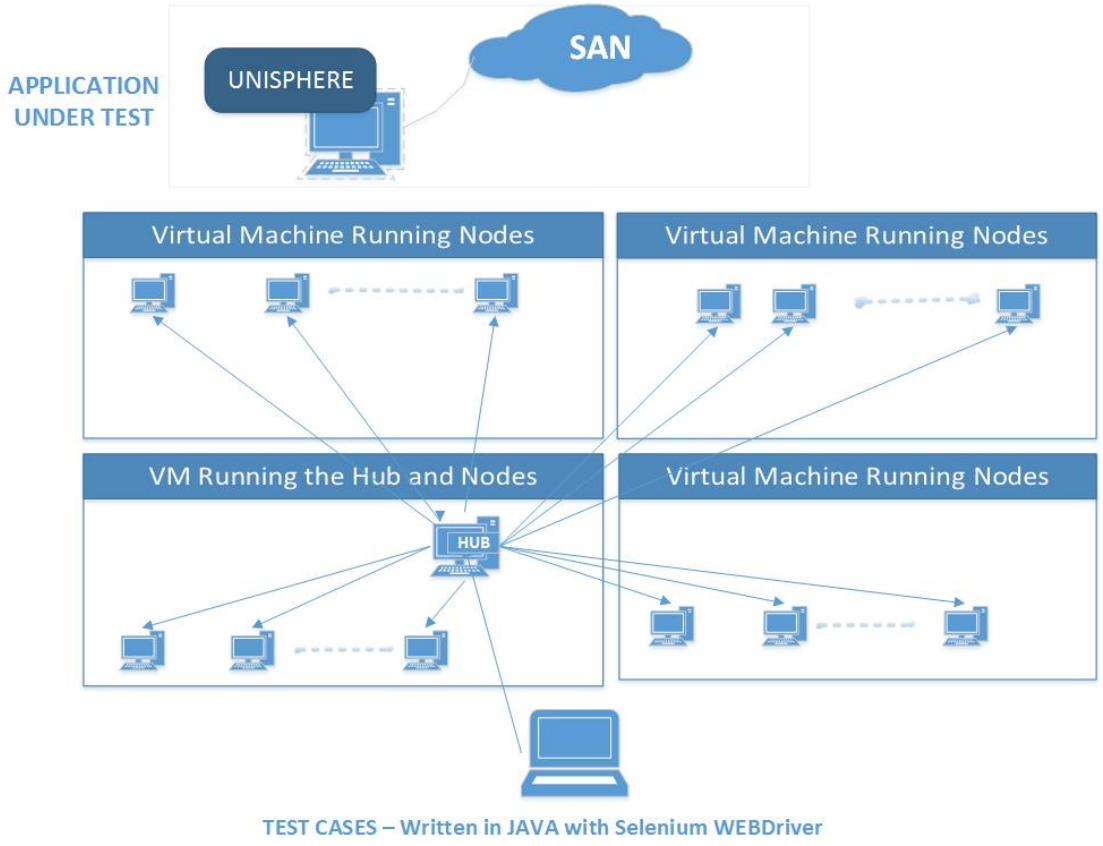


FIGURE 3.7: Four Virtual Machines Grid Basic Architecture

Figure 3.8 shows the Virtual Machine Optimized for running Containers with all host resources. It has 32 GB RAM, 24 Cores and 131 GB disk space.

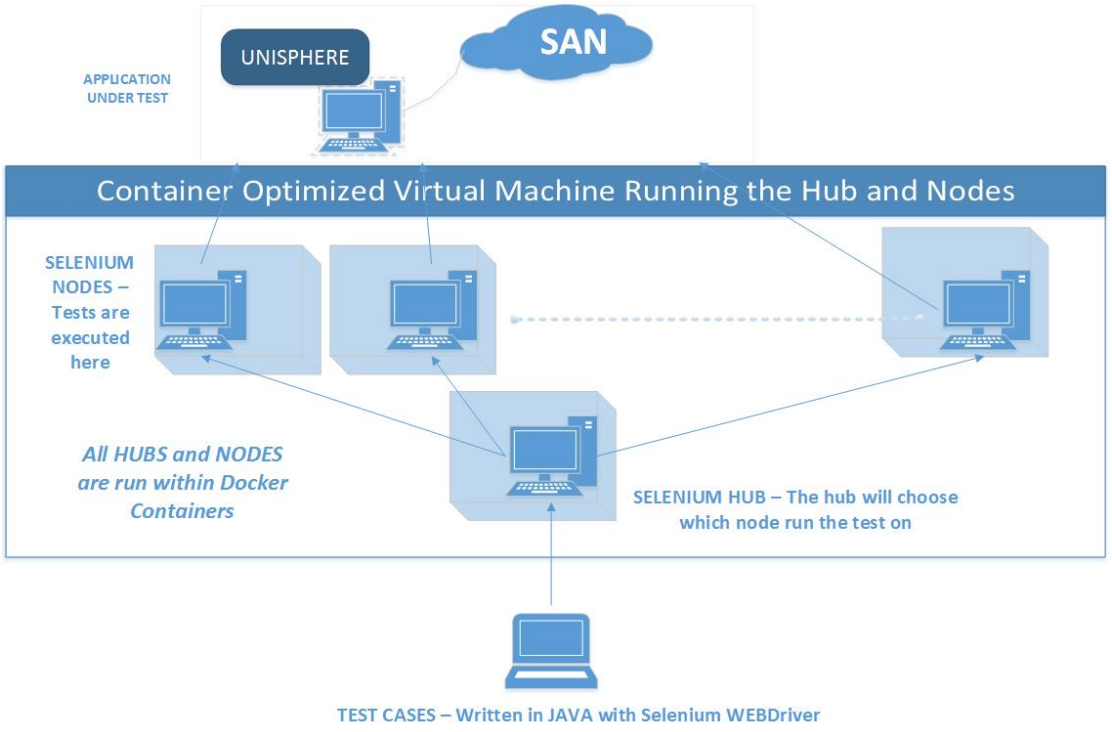


FIGURE 3.8: Standalone Virtual Machine Grid Basic Architecture

Figure 3.9 shows Four Virtual Machines Optimized for Running Containers sharing Host Resources equally. Each VM has the same resources, 8 GB RAM, 6 Cores and 20 GB disk space. The remaining disk space is left free for vSphere snapshot management. Each hub and node is run within a container. There is a single hub on each VM which manages all nodes on that VM.

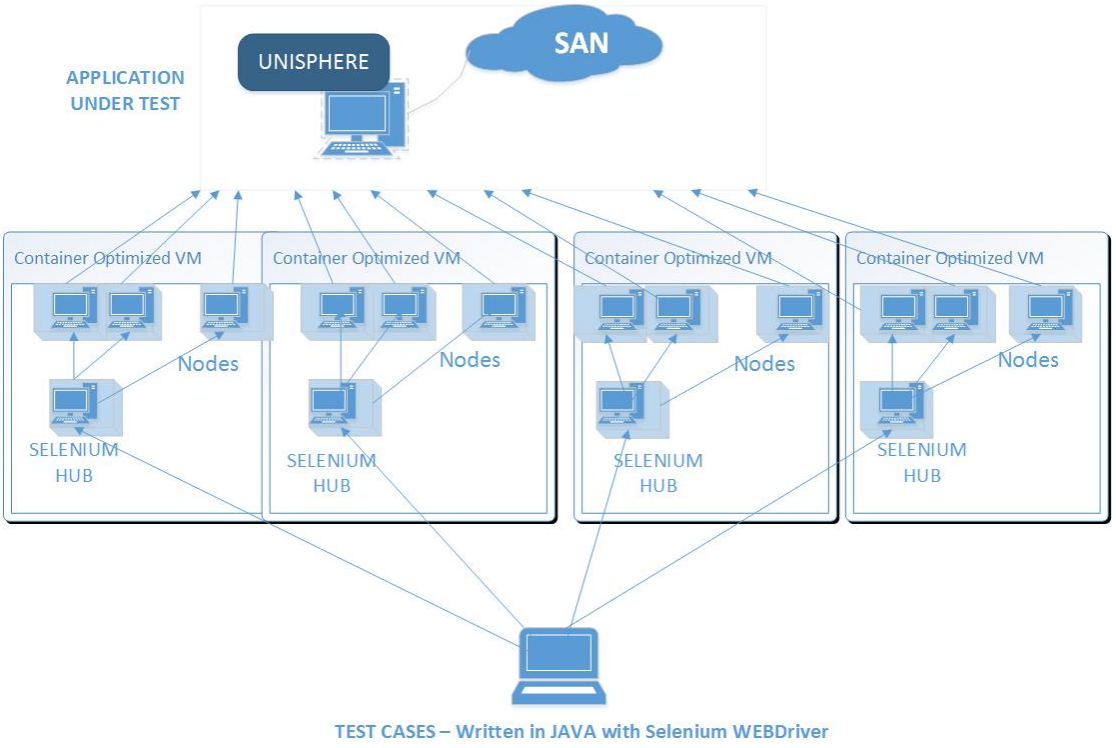


FIGURE 3.9: Four Virtual Machines Grid Basic Architecture

3.4 Test Automation Framework Code

3.4.1 Page Object Model Design Pattern

One of the most common ways to write UI Test automation code is to use the Page Object Model design pattern. The idea behind this is to separate the logic of the test case from that of interacting with the Application Under Test by creating a repository of elements for each page used in the tests [56]. In this way, it creates robustness to the test case code whereby changes in the UI can be more easily absorbed as tests interface with the decoupled page object elements rather than directly with the HTML of the web page. A study by Leotta et al [57] concluded that using this design pattern reduced

framework maintenance time by over 65% and reduced Lines of Code (LOC) in the project by over 87%.

PAGE OBJECTS

A set of page objects are created to be able to run the StorageGroups Test Plan. Six Page Object classes are needed for this, whose layout can be seen in Figure 3.10

Page Object Package Layout

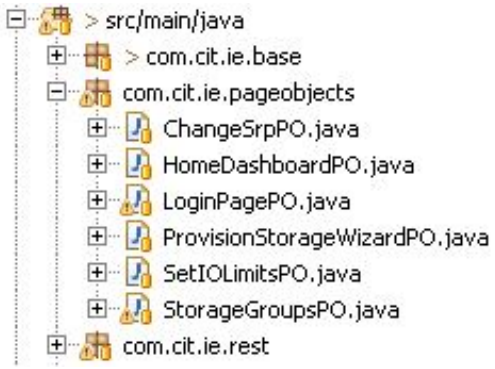


FIGURE 3.10: Page Object Package Layout

Each Page Object contains the information for locating the `WebElements` on that page, such as a button, textfield, label, etc. These are located in this instance by using xPath strings, the following code shows the locators for the buttons located along the top of the Home Dashboard.

```
//TOP BUTTONS
public final String USER_PROFILE_XPATH="//button[@aria-label='User Profile']";
public final String LOGOUT_XPATH="//button[@aria-label='Logout']";
public final String LOGOUT_CONFIRM_XPATH="//button/span[text()='OK']";
```

`PageFactory` is a factory class provided by Selenium WebDriver to support the Page Object design pattern [58]. This is used to initialize `WebElements` on each page. The example below shows how this class intializes the `WebElements` using the locator strings already described.

```
@FindBy(xpath=USER_PROFILE_XPATH)
public WebElement userProfileButton;
@FindBy(xpath=LOGOUT_XPATH)
public WebElement logoutButton;
```

```
@FindBy(xpath=LOGOUT_CONFIRM_XPATH)
public WebElement logoutConfitmButton;
```

Also created for each Page Object are sets of services that are accessible from each page such as navigation and logout functions. The following code shows examples of navigation and logout functions provided by this Page Object.

```
//Navigate to the Storage Groups List View
public void navigateToStorageGroups() throws InterruptedException{
navigateToSideMenu();
jsClickElement(storageGroupsSideMenuItem);
}

//Logout of the Unisphere For VMAX Application
public void doLogout() throws InterruptedException{
jsClickElement(userProfileButton);
jsClickElement(logoutButton);
jsClickElement(logoutConfitmButton);
}
```

Selenium WebDriver instances are managed by a `WebDriverManager` class which has a `launchBrowser()` method that is run before each test. Using the `@Parameters` annotation it is possible to specify the browser type, the Selenium Grid Hub IP Address and the URL of the Unisphere installation. These parameters are passed at runtime. This method also instantiates the WebDriver instance for this test.

```
@Parameters({"browser","hubIP","applicationURL"})
@BeforeClass()
public void launchbrowser() throws MalformedURLException, InterruptedException
    ↪ {
appURL=applicationURL;
if (browser.equalsIgnoreCase("firefox")) {
DesiredCapabilities cap = DesiredCapabilities.firefox();
cap.setBrowserName("firefox");
cap.setCapability("marionette", false);
cap.setCapability(FirefoxDriver.PROFILE, new FirefoxProfile());
```

```
cap.setCapability(CapabilityType.ACCEPT_SSL_CERTS, true);
cap.setCapability(CapabilityType.UNEXPECTED_ALERT_BEHAVIOUR,UnexpectedAlertBehaviour.ACCEPT);
String Node = "http://"+hubIP+":4444/wd/hub";
threadDriver = new ThreadLocal<RemoteWebDriver>();
threadDriver.set(new RemoteWebDriver(new URL(Node), cap));
}
```

TESTS

Each test is written as a separate class. This is just one approach that can be taken as it also possible to put all the tests in one class, it is really a matter of personal preference. The TestNG `@Test` annotation is used to specify a test method. For each test a new `WebDriver` object is created, browser session is started and Page Objects are created as needed. Tests then interact with the elements on a page by page basis just as a user would; pointing, clicking, writing data, reading data and so on.

Table 3.3 shows a simplified test

TABLE 3.3: Test Case - Create an Empty StorageGroup.

Step No.	Step Description
1	Launch Firefox Browser
2	Enter Unisphere For VMAX URL
3	Enter Valid Username and Password and Click Login
4	Click Storage SideMenu Item
5	Click StorageGroups SideMenu Item
6	Click Create StorageGroup Button
7	Click StorageGroup Name Textfield on Wizard
8	Write the StorageGroup name
9	Click the RunNow button to Create Empty StorageGroup
10	Wait Until Success Message is visible
11	Close Browser
12	Verify that the StorageGroup is created with REST or CLI
13	Cleanup Testdata

The following Test code shows how the test is implemented. The test creates the `WebDriver` instance, creates a `LoginPagePO` object, does a login, and navigates to the Storage Groups list view page. It then creates a `StorageGroupsPO` object which enables it to click the create new Storage Group button. From there it creates a `ProvisionStorageWizardPO` object which gives it all the elements needed to finish creating the Storage Group. Finally it verifies the information using using REST and deletes it using REST.

```
@Test
private void _002_CREATE_EMPTY_SG_WITH_SRP() throws JSONException,
    ↪ IOException, InterruptedException {
```

```
sgName="00DC02";
if(threadDriver!=null)
{
findRemote(threadDriver.get());
}
//CREATE LOGIN PAGE OBJECT
LoginPagePO lppo=new LoginPagePO(getDriver());
lppo.gotoStorageGroupsPage(lppo,sgName);
//CREATE STORAGE GROUPS PAGE OBJECT
StorageGroupsPO sgpo=new StorageGroupsPO(getDriver());
sgpo.waitForStorageGroupsPageObjects();
sgpo.jsClickElement(sgpo.createStorageGroupButton);
//CREATE PROVISION STORAGE PAGE OBJECT
ProvisionStorageWizardPO pswpo=new    ProvisionStorageWizardPO(getDriver());
pswpo.waitForElementVisiblity(pswpo.WAIT_FOR_PAGELOAD);
pswpo.jsClickElement(pswpo.storageGroupNameTextField);
pswpo.storageGroupNameTextField.sendKeys(sgName);
pswpo.setSrpInformation(pswpo,"default_srp");
pswpo.jsClickElement(pswpo.selectRunMethodMenu);
pswpo.jsClickElement(pswpo.createSgRunNow);
sgpo.waitForElementToDisappear(pswpo.TASK_IN_PROCESS_XPATH);
sgpo.quitWebDriver();
//VERIFY DATA AND DELETE
pswpo.verifyAndCleanup(sgName);
}
```

3.4.2 Verification of the Data and Cleanup

A simple Representational state transfer (REST) Client has been written to call the Unisphere for VMAX RESTAPI. The code for this can be found in Appendix A. The response status of a successful GET call is 200. Using TestNG the test can assert for this to verify if a Storage Group has been created. If the response status is 200 then pass the test, if it is not, fail the test. Once verified successfully it will delete the Storage Group.

```
public void verifyAndCleanup(String sgName) throws InterruptedException {
//VERIFY THAT GROUP HAS BEEN CREATED WITH REST
```

```
RESTClient.refreshRestDB(baseURL);
RESTClient.GET(baseURL+sgName);
if (RESTClient.responseStatus!=200)
{
RESTClient.printResponses();
}
Assert.assertEquals(RESTClient.responseStatus,200);
//CLEANUP BY DELETING THE STORAGE GROUP
RESTClient.DELETE(baseURL+sgName);
}
```

3.4.3 Running Tests

TestNG

TestNG is a JUNIT-like test framework that is used to run tests [59]. It gives an extensive set of annotations to define tests, setup test preconditions and specify test input parameters. Some examples of this have already been shown in the preceding code.

```
@Test:         A piece of code that is a test
@BeforeClass:  A piece of code to be run before the class (Pre-conditions)
@BeforeTest:   A piece of code to be run before each test (Pre-conditions)
@AfterMethod:  A piece of code to be run before each method (Post-conditions)
@Parameters({firefox",applicationURL"}):  Test Inputs
```

TestNG also gives a set of *Assertions* that can be used to validate data. These assertions verify the conditions of the test and to decide whether it has passed or failed. See the examples below.

```
Assert.assertEquals(500,responseStatus);
Assert.assertTrue(responseOutput.contains(SomeData));
```

Finally it gives a way to run the tests in a parallel fashion which is a crucial part of this experiment. Using an XML file it is possible to specify which tests to run, which parameters to set, and how many threads to run in parallel. In the example below the

user is running 20 tests in parallel as set by the *thread-count*. This test run is also pointing at a Selenium Hub (see Fig3.7) which contains the RedHat Virtual Machines Grid as set by *hubIP*. It is also possible to spread the load across many instances of Unisphere through the *applicationURL* parameter and set firefox as the *browser* for this test run. This approach gives great flexibility in the configuration of test runs.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite verbose="1" name="FF VM GRID Suite" parallel="tests" thread-count="20">
<test name="TEST0">
<parameter name ="browser" value="firefox"/>
<parameter name="hubIP" value="10.73.29.47"/>
<parameter name ="applicationURL" value="https://10.73.28.70:8443/univmax"/>
<classes>
<class name ="com.cit.ie.storagegroups.Test000"/>
</classes>
</test>

<test name="TEST1">
<parameter name ="browser" value="firefox"/>
<parameter name ="hubIP" value="10.73.29.47"/>
<parameter name ="applicationURL" value="https://10.60.141.19:8443/univmax"/>
<classes>
<class name ="com.cit.ie.storagegroups.Test001"/>
</classes>
</test>
```

3.5 Other Configuration

The project is organized using the well known Maven directory structure [60] and makes use of the Project Object Model (POM) file to manage its dependecnies for Selenium, TestNG and Jersey RestClient. The code is implemented using Java JDK version 8. At the time of writing it is using Selenium WebDriver 2.53.0 and TestNG 6.9.10.

Github is being used for version control and as the code repository. The code for the Automation Framework can be found on Github [61]. There were some constraints in

the operating systems that were available for the paravirtualized VMs. RHEL6.9 images are used which do not support Chrome. The Docker Selenium images do not yet support Internet Explorer out of the box. There is also a bug in the Firefox Marrionette driver that was not resolved when the project started details of which can be found here [62]. For these reasons it was decided that the best workable solution was to use the Firefox only for the test runs and to use the ESR 45.6 driver.

3.6 Research Parameters and Methodology

This section describes the method by which the research questions will be answered. The two areas to be examined are the parameters for the research and the research methodology. In order to maintain the logical flow and consistency of the chapter headings, some of the methodology section will be placed in Chapter 4.

Henrichsen at al [63] describes four clear parameters for research, it is important to be aware that making a choice in one affects the options in another. The parameters are described in Table 3.4

TABLE 3.4: Parameters of Research.

Research Parameter	Type
General Approach	Synthetic : A holistic view of the whole system is taken Analytic : A constituent view of the system is taken.
Research Aim	Deductive : Top down, a hypotheses is tested and examined then proven/disproven Heuristic : Bottom up, a hypothesis is generated as questions are answered
Control over the Research Context	The researcher observes and does little to effect the context of the research The manipulates, changes, re-organizes the context of the research
Explicitness of Data Collection	Low- Less formal methods of collecting data High Precise and restricted methods of collecting data

Based on the information above it is determined that this study will fall to the analytic side of the spectrum when taking a general approach. Individual components, for example CPU and memory and possibly even disk space and networking, will be looked at when trying to find relationships between them, and to see what affects performance. It is mainly deductive in its research aim. As regards the control over the research,

this project falls somewhere in the middle. The research question is well established and the initial plan is to set the tests up and observe, however depending on how the process evolves there could be be some element of fluctuation as to how this work is organized. Finally, as regards Data Collection, the experiments will follow explicit line of questioning and testing. Tests will only be run at nights and evenings when it can be guaranteed a minimum level of disruption and ensuring consistency in terms of network traffic as much as is possible, and results will be generated and correlated. A minimum of three tests run for each scenario will be run and average results will calculated. It is important that this is highly structured and controlled in order to generate good data. For theses reasons the study ranks very high in this category.

There will be both quantative and qualitative aspects to the study. Quantitative Research, focuses on what can be measured. Data is collected and from this data we can make conclusions. Test cases will be run against various environments to gather results and this data will be used as the basis for the analysis. This will be the bulk of the work. There is also a qualitative aspect concerning the usability and flexibility of each environment from a test engineers point of view.

The research methodology being used is the *Empirical Research Process for Software Engineering*[64]. There are six phases to this methodology, the first two of which will now be described in the context of this study. The remaining phases will be discussed in Chapter 4.

3.6.1 Experimental Context

When a new software engineering technique or study is done, in this case examining Selenium performance across five environments, background information regarding the circumstances should be recorded. This will include data on all the hardware involved, all the software involved and any other information relevant to the experiment. This has been described in the technoloies used section in Chapter 2, as well as in the Background chapter. In this regard as much context information as is possible has been detailed. Also in Chapter 2, the research in the area has been described and an explanation as to how this project differs from others is given.

3.6.2 Experimental Design

Identify the Population: This has been achieved in the cataloging and description of Test Automation solutions, hardware choices, software choices including - Virtualization, Containers, Container Engine.

Process for Selection: This has been described in detail in Chapter 2. Each subject will receive the same treatment. The test cases do not change, the SAN does not change, only the environment where the browsers are run changes.

Avoiding bias, user controlling the task and introducing levels of blinding: This is hard to do in this study but hopefully due to the automated nature of the tasks, then the chances of directly influencing results should be reduced.

Chapter 4

Results and Observations

In this section the results of the experiments are presented and analyzed. There are two sections, the first focuses on how the experiment was conducted, results recorded, and data was analyzed. This is examined within the context of the Research Methodology "Guidelines for Empirical Research in Software Engineering".

The second section focuses on the operational flexibility observations from a user point of view for each solution.

4.1 Research Methodology

This section details the final four parts of the *Empirical Research Process for Software Engineering*. The Analysis And Presentation of Results sections are generally considered separate sections under the methodology but are combined to enhance the flow of this report.

4.1.1 Conduct of the Experiment and Data Collection

A test suite containing 66 Test Cases has been written using the Selenium Framework. The tests are concerned with testing the functionality of the Storage Group feature in Unisphere For VMAX. There are 5 separate test environments that are set up across the 3 different environments (Physical, Virtual, Containerized). These are outlined in Table 4.1.

TABLE 4.1: Test Environments

Environment Name.	OS	CPU	MEMORY	DISK
Physical Machine	SLES 11 SP4	24 Core	32GB	550GB
Single Virtual Machine	RHEL 6.7	24 Core	32GB	131GB
Virtual Machine x 4	RHEL 6.7	6 Core	8GB	20GB
Single Container Optimized VM	PhotonOS 1.0	24 Core	32GB	131GB
Container Optimized VM x 4	PhotonOS 1.0	6 Core	8GB	20GB

4.1.1.1 Test Scenarios

This is an important section as these *Scenarios* will be referenced throughout the remainder of the report. While the ultimate barometers of performance for the study will be pass rates and execution speed, with the behaviour of CPU and Memory during the suite elucidating the relationship between these resources and test performance. In order to measure these metrics the *top* [65]and *esxtop* [66] linux utilities will be used.

For the physical machine *top* will be run in batch mode for the duration of the test runs. Its output data is recorded in 10 second intervals. For the virtual machine environments the ESXi utility *esxtop* is run for the duration of the test run and its data is also recorded in 10 second intervals. On the vSphere management software, real-time performance graphs are supplied for CPU and Memory so these graphs will also be collected. There is no such facility for the physical machine. Each test requires at least one instance of a browser and java selenium process. The test methodology seeks to start with a single threaded execution of the test suite on each environment, behaviour is observed and results will be recorded. Then repeat this exercise, running the test suite with 10 tests executing concurrently. Then repeat the exercise for 20, 30 and 40 tests in the same way.

TABLE 4.2: Test Scenarios

Name.	Concurrency
Scenario 1	Single Test Execution
Scenario 2	Ten Tests Executed in Parallel
Scenario 3	Twenty Tests Executed in Parallel
Scenario 4	Thirty Test Executed in Parallel
Scenario 5	Fourty Tests Executed in Parallel

A final "unofficial" *Scenario 6* was added at a later time, simply to test the framework at what is 100% capacity. In this Scenario, every test is run in parallel and 66 threads

are executed simultaneously. For completeness this Scenario, although originally not included in the study, will be included in the final results.

Finally, in the last days of this study, a physical RHEL machine with an identical configuration to the Cisco hardware except that it had 16 cores instead of 24 was made available. Scenario 4 was executed once on this machine. This run will be mentioned briefly but will not be included in any tables or graphs as it has not been tested according to the methodology defined.

4.1.1.2 Reliability

In order to improve reliability and validity of the data, each test suite is run three times for each scenario and average results are calculated. If there are outliers, further investigation will take place and the sample will be increased to five test runs.

The hosts are all present in the same lab, running on the same SAN and Local Area Network (LAN). There should not be any significant latency issue between them. All hardware is running the same software where applicable, such as browser versions, docker versions, selenium components. All of the physical hosts have the same 32G RAM (and 4 GB Swap space) and all have 12 dual core CPUs giving a 24 core total.

As the Photon OS is a minimal Linux distribution it was necessary to remove the Graphical User Interface (GUI) packages from the Linux RHEL VMs in order to create a like for like comparison as much as was possible for the test runs. The upshot of this is that Firefox needs to be run in headless manner [67]. This means that the browser is run without a graphical user interface. In its place an emulated Graphical User Interface (GUI) is run with a utility called *Xvfb* which lets the user run an X server with no physical display associated with it. Appendix B.2 contains the steps needed to configure this. The Virtual Machines have each been created with a 20GB hard-drive.

The biggest potential for discrepancy is the different chipsets on the CPUs. The Physical Suse Linux host has the E5-2620v3 @ 2.40GHz chip which has a CPU benchmark score of 9976 [68] , while the Cisco Servers where the ESXi runs have X5650 @ 2.67 GHz have a benchmark score of 11589 [69] . This will be taken into consideration when analyzing results.

4.1.2 Analysis And Presentation of Test Run Results

The results of the passed tests in seen in Figure 4.1 clearly show that the physical machine performs best from the point of view of passing tests which is the key criteria for any test solution. In every test scenario it returned a pass rate of 100%. It even was able to run 66 tests concurrently without any issue.

Tests Passed Chart

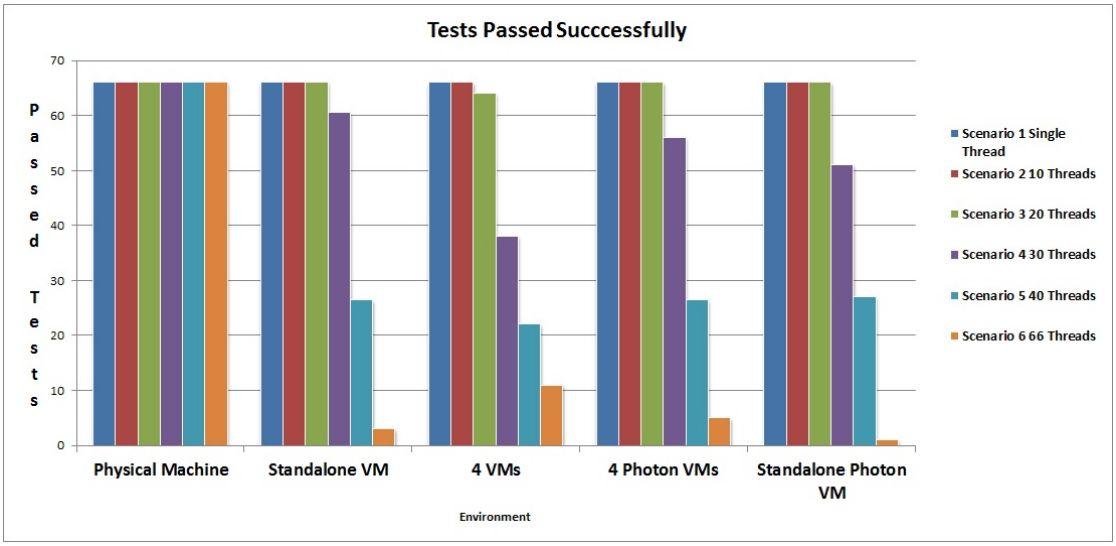


FIGURE 4.1: Tests Passed

Scenarios 1 and 2 had 100% pass rates on all 5 environments. All but the *"4 VMs, Scenario 3, 20 Threads"* environment had 100% pass rates on Scenario 3, but that still had a very high pass rate of 97%. All environments apart from the physical suffered degradation as each testing scenario added more concurrency, along similar patterns. The "Standalone VM", "Standalone Photon VM", and "4 Photon VMs" showed very similar trends in their pass rates across the 6 test Scenarios. Looking at Scenario 4 *"Standalone VM, Scenario 4, 30 Threads"* still performed well, with a pass rate of 91% passing 60 of 66 tests, which was 36% better than the *"4 VMs, Scenario 4, 30 Threads"*, 7% better than the *"4 Photon VMs, Scenario 4, 30 Threads"* , and 15% better than the *"Standalone Photon VM, Scenario 4, 30 Threads"* . Once the test suites passed the 40 concurrent Tests mark at Scenario 5, all environments bar the physical machine drop below a 40% pass rate and the value of running such automation is almost lost. The *"Standalone VM, Scenario 5, 40 Threads"*, *"Standalone Photon VM, Scenario 5, 40 Threads"*, and *"4 Photon VMs, Scenario 5, 40 Threads"* environments all perform joint

best with 39% pass rate with the *"4 VMs, Scenario 5, 40 Threads"* again performing the worst at 33%.

Another metric that is of interest to the study is the speed of test execution when tests are passed. There are two things to note about this metric. The first is that it was necessary to write custom wait code that will wait for certain events to happen, such as waiting for a page to load, before the test declares the page to have timed out and the test failed. This timeout is set to 10 minutes as some instances of successful calls taking up to 8 minutes to complete have been observed. Typically this type of slow performance occurs as increases in CPU and Memory are seen on the machine. The second thing to note is that a test can fail out early and the test time recorded is short but worthless as it has failed, therefore making it inadmissible for these results.

For these reasons, it makes sense to only count fully successful test suite runs (or very near successful test suite runs) when measuring the time it takes to complete a test run. The following chart shows the completion times for the first three scenarios.

Minutes to Complete Test Suite Chart

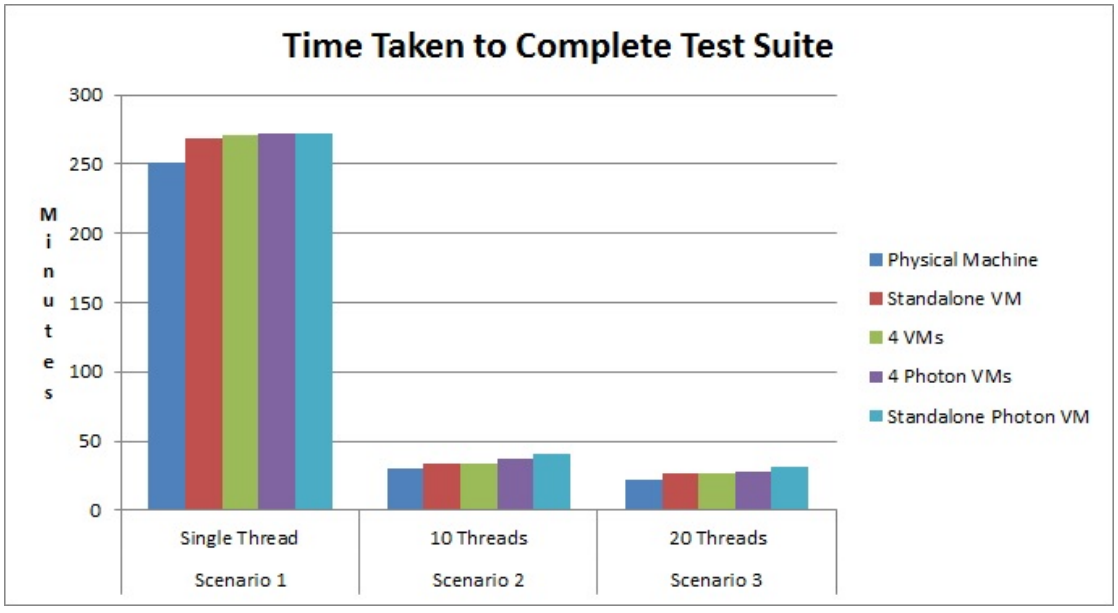


FIGURE 4.2: Minutes To Complete Test Suite

Scenario 1 shows a single threaded execution of the 66 Test Case suite. The *"Physical Machine, Scenario 1, Single Thread"* again performs best finishing in 250 minutes, the other four environments all finish within 4 minutes of each other from 268 to 272 minutes.

Once the test runs move up to 10 threads as in Scenario 2 it can be seen that the test time is cut down considerably, by up to a factor of 8 depending on the test run. The *"Physical Machine, Scenario 2, 10 Threads"* still performs best with a 30 minute completion time. The RHEL environments are next best performing with *"Standalone VM, Scenario 2, 10 Threads"* and *"4 VMs, Scenario 2, 10 Threads"* finishing in 33 minutes and 34 minutes respectively. The Photon environments are slowest with *"4 Photon VMs, Scenario 2, 10 Threads"* finishing in 37 minutes and *"Standalone Photon VM, Scenario 2, 10 Threads"* finishing in 40 minutes. These times show the Physical Machine to be 10% and 13% faster than the two virtual machine environments, and 23% and 33% faster than the containerized environments. At 20 threads, the completion times are reduced further but only by about 10 minutes. The *"Physical Machine, Scenario 3, 20 Threads"* finishes in 21 minutes, with the RHEL envrionments both finishing in 26 minutes, making the Physical Machine environment 23% faster. The *"4 Photon VMs, Scenario 3, 20 Threads"* and *"Standalone Photon VM, Scenario 3, 20 Threads"* finishing in 28 minutes and 31 minutes respectively, making the Physical machine faster by 33% and 47%.

A further point to note here which is not included in the chart is that the Physical Machine was able to reduce its execution time down to a 14 minute execution time for a *"Physical Machine, Scenario 6, 66 Threads"* test run where every test in the suite was run in parallel. It is the only environment to continually decrease its execution time as more parallelization is introduced. The other environments lost all value as timeouts, browser crashes and out of memory issues start to occur.

The Physical RHEL Machine that was briefly borrowed ran Scenario 4 and performed extremely poorly. It took 361 minutes to complete, and only passed 16 tests.

4.1.3 Interpretation of Results

The first set of tests to start to fail is the *"4 VMs, Scenario 3, 20 Threads"* environment. It is exclusively timeouts that cause this. The error received is a selenium timeout exception when waiting for a web element to disappear. It is interesting that the Storage Group is actually created, but that the browser is in a hanging state.

```
org.openqa.selenium.TimeoutException: Timed out after 1200 seconds waiting for
    ↪ element to no longer be visible: By.xpath: //label[text()='Task in
    ↪ process...']
```

One place to look for a reason for this is that the *"4 VMs, Scenario 3, 20 Threads"* environment is the only environment that has inter virtual machine communication as there is a selenium hub on one VM which is responsible for managing the 4 nodes. The standalone systems all have one hub and one node on the same machine, while the docker-selenium setup on the photon VMs sets up a hub within a container per VM and only communicates with other node containers on that VM. However the average Data receive rate (36 KBps) and Data transmit rate (8 KBps) of these VMs is very low, so it is probable that this is not the major reason.

4.1.3.1 CPU

It is noticeable that the tests which failed in this *"4 VMs, Scenario 3, 20 Threads"* test run were in the first batch of tests. There is a "big bang" type start to the suite run and it can be seen from looking at the CPU performance for the ESX as provided by vSphere. This is a common trend across all of the test runs. After the initial burst of activity, tests tend to finish, then start at different times and the CPU activity flattens out a little. The following figure shows an example of this bursting during the test run. CPU usage goes over 100% for several minutes at the start of the test run and tests that were running in this phase were the tests that timed out.

CPU Real Time Chart

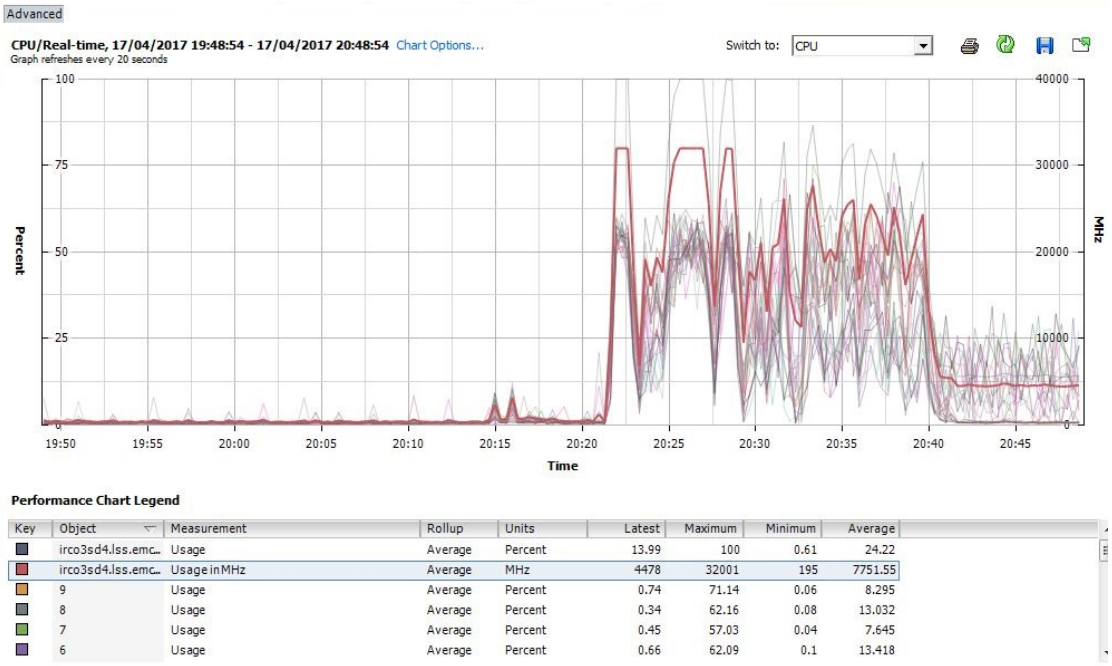
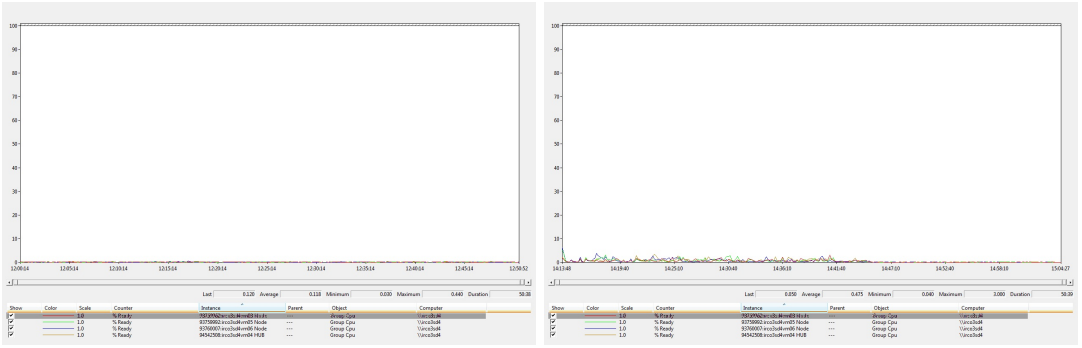


FIGURE 4.3: CPU Bursting

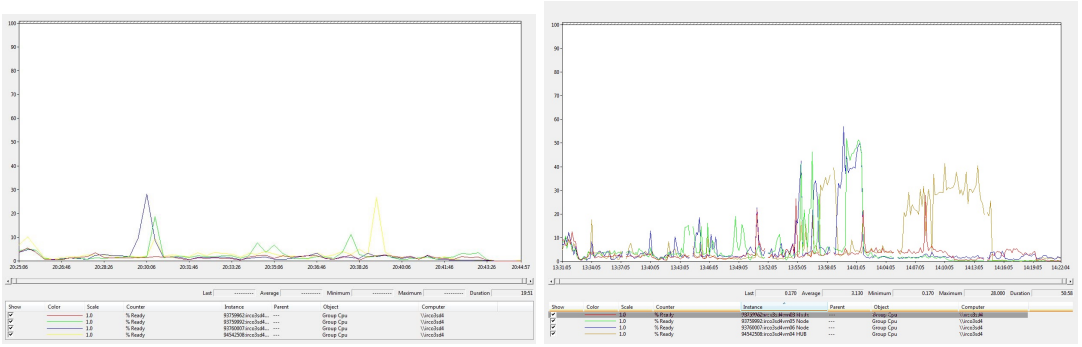
Two other test runs for this environment did not hit the same level of bursting, they came close to 100% CPU usage and even hit it briefly. In these tests timeouts did not occur and the pass rate was 100%. This is a trend that will continue on into the other Scenarios. It should also be noted that the amount of memory being used on the host never goes over 50% for any of the environments up to *Scenario 3*.

A related concept when examining the CPU usage is the metric known as %RDY which is available from *esxtop*. This defines how long the virtual machine is waiting to execute CPU cycles but could not get access to the physical CPU. A general rule of thumb is that this should be less than 5% [70]. While this data is recorded at 10 second intervals, it is better seen plotted over the course of the test run to see how there can be spikes in the test run. For the following example, a sample from the first four Scenarios for the *"4 VMs* environment will be examined to illustrate how this metric may impact performance. In the single threaded executions Figure 4.4 shows the %RDY on a single threaded and 10 thread execution and it can be seen that the percentage remains extremely low right throughout the test run, and this coincides with the 100% pass rates. Figure 4.5 shows a 20 Thread execution and in the first intances of spiking are seen, which coincide with a small drop in the pass rate. For the 30 Thread execution, the %RDY is consistently



(a) Single Threaded Execution (b) 10 Thread Execution

FIGURE 4.4: CPU %RDY on 4 VM Environment



(a) 20 Thread Execution (b) 30 Thread Execution

FIGURE 4.5: CPU %RDY on 4 VM Environment

going over the 5% mark for all VMs and this also happens to be the Scenario where performance drops off considerably.

4.1.3.2 Memory

Once the test Scenarios get to 30 and 40 concurrent thread runs it can be seen that memory usage also increases on the hosts. As all four environments are run on vSphere it gives us a good "apples to apples" way of observing the memory and CPU usage on its performance chart feature. The trend in terms of passing and failing is very similar for all hosts as is the trend in CPU usage and Memory usage. The chart as seen in Figure 4.6 shows a *"4 VMs, Scenario 5, 40 Threads"* test run. It can be seen that memory usage is very high at the beginning of this test run and critical vSphere alerts for memory usage were received.

FIGURE 4.6: Memory Usuage for 40 Concurrent Tests

The observations for both the RHEL environments are that "Unexpected modal dialog" errors are received when there are memory problems which causes tests to fail. There are numerous different types of these errors which mainly relate to loading js files. Here are two common examples which regularly come up.

```
Unexpected modal dialog (text: A script on this page may be busy, or it may
    ↪ have stopped responding. You can stop the script now, open the script
    ↪ in the debugger, or let the script continue.
```

```
Script: https://10.73.28.70:8443/univmsystem.src.js line 1853 > eval:10): A
    ↪ script on this page may be busy, or it may have stopped responding. You
    ↪ can stop the script now, open the script in the debugger, or let the
    ↪ script continue.
```

```
Unexpected modal dialog (text: A script on this page may be busy, or it
    ↪ Script: https://10.60.141.19:8443/univmax/app/lib/system.src.js:2975)
    ↪ The alert could not be closed. The browser may be in a wildly
    ↪ inconsistent state, and the alert may still be open. This is not good.
    ↪ If you can reliably reproduce this, please report a new issue at
    ↪ https://github.com/SeleniumHQ/selenium/issues with reproduction steps.
```

Once these tests fail memory usage falls back but there is still very high CPU usage throughout the suite run as can be seen from Figure 4.7. The timeout errors return for the second half of the test run as described before and this ties in with previously described behaviour. The effect of this is a high failure rate, and a very long test execution time as the timeouts kick in later in the suite.

FIGURE 4.7: CPU Usuage for 40 Concurrent Tests

On the Photon VMs the pattern of Memory and CPU Usage is very similar, as are the pass rates, but there is a difference in the way that the tests fail. Figure 4.8 is showing the memory usage on a test run of the 4 Photon VMs starting at approximately 23.43.

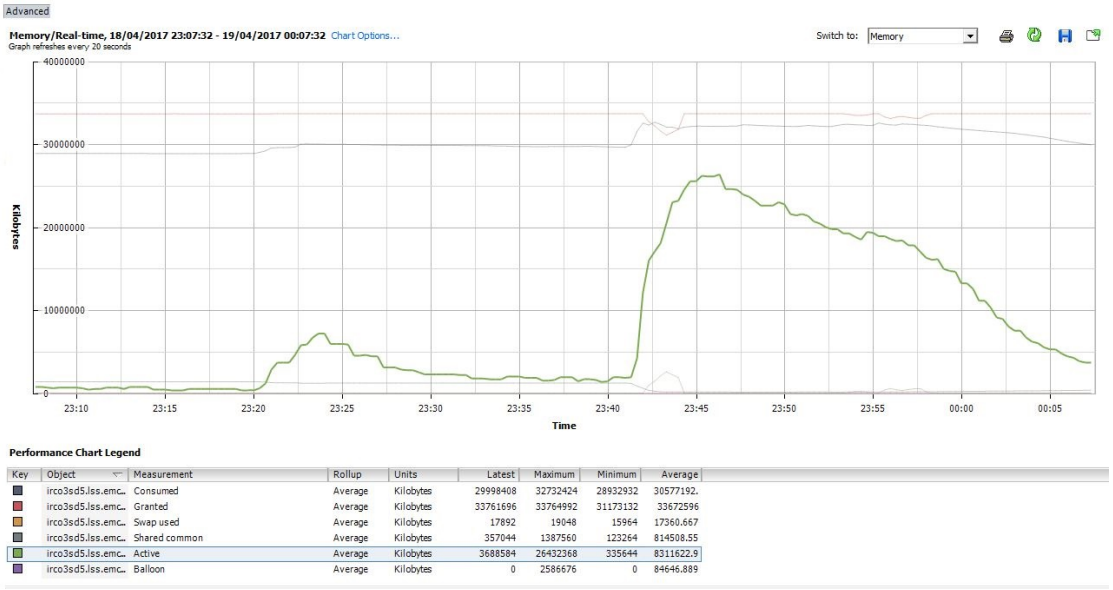


FIGURE 4.8: MEM Usuage for 40 Concurrent Tests (Docker Containers)

Looking at the Photon machines console, it can be seen that the each containers memory usage gets higher and higher until all available memory is used up. Errors like the following are visible on the VM itself as the Linux kernel attempts to free up memory by killing processes

```
Out of memory: Kill process 9518 (firefox) score 69 or sacrifice child
KIlled process 9518 (firefox) total-vm-1290040kb, anon-rss:650404kB,
    ↪ file-rss:0kB
```

When this happens browser crashes are seen in the tests and the tests fail immediately. The selenium exception below is what is seen in the error logs.

```
org.openqa.selenium.WebDriverException: Error communicating with the remote
    ↪ browser. It may have died.
```

The tests fail but they fail quickly and as such the execution time overall is shorter. Initially it was not noticed that the RHEL images had a 4GB swap file which the Photon VMs did not have and updating these brought the pass rate more into line with the RHEL environments. It may be possible to optimize the memory management even more using the docker cgroups but the bottom line for this is that there is a limit on the amount of memory available and that is the ultimate limiting factor. Another

interesting observation is that timeout errors have not been seen for the container tests, just browser crashes and modal script errors.

It must also be noted that even though the CPU chipset on the Physical Machine has lower benchmark score than the ESXi host machines, its performance is considerably better.

The final part of this section is a comment on the Physical RHEL machine which performed so poorly. The immediate standout point from this test run is that it had 8 cores less than the other machines, and could not cope with the 40 test run even though it was a Physical Machine. There were browser timeouts in over 70% of the tests.

4.2 Observations on Operational Flexibility

The first thing to mention here is that this is the user experience of one user as was found from conducting this particular study and is not indicative of any survey of opinions. From a provisioning point of view it was found that the Photon/Docker solution provided many benefits over the RHEL system. First among these are a much smaller image size, and a much shorter boot time. Table 4.3 shows the boot times of the three Operating systems.

Table 4.3: Boot Times

Operating System	Boot time in Seconds
RHEL	23
SLES	32
Photon	9

The RHEL OS VMs even minus the GUI packages takes 23 seconds to boot, and the SUSE Physical machine takes 32 seconds. The Photon OS takes only 9 seconds. This makes the Photon OS 255% faster than the RHEL, and 355% faster than the SLES on boot up. If a follow on study whereby some level of data center orchestration/scripting were brought into the equation were to be undertaken then this would be an important consideration. In a cloud service, the ability to rapidly provision and deprovision virtual machines is of great benefit.

Another thing to consider here is the convenience of using the Docker images for configuration and for managing software versioning. Take the example whereby the user needs to get Chrome driver for selenium.

It is a very simple process to add a few lines to the docker-compose file, or similarly a user could also just pull in the image from the docker hub repository as shown in the command below. Notice here that a legacy version of Selenium, 2.45, is chosen. Docker makes it really simple to pull in different versions of software as needed and in an open source project like this it can be very useful to bring in the latest drivers as bug fixes come in from this very active development community.

```
root@irco3sd5vm3 [ ~/dockerComposeFiles/selstartup ]# docker run -d -P -p
    ↪ 5901:5900 --link selenium-hub:hub selenium/node-chrome:2.45.0
Unable to find image 'selenium/node-chrome:2.45.0' locally
2.45.0: Pulling from selenium/node-chrome
04c460fac791: Pull complete
0a0916b29f3e: Pull complete
b25f4e7a7766: Pull complete
a3ed95caeb02: Pull complete
3591a8ba8b08: Pull complete
2a951d2b6c90: Pull complete
8bdf71ec1ca9: Pull complete
b28cfe472e0e: Pull complete
daaafa40501c: Pull complete
f33915be70ef: Pull complete
4ef9a1197f73: Pull complete
b41fee477a24: Pull complete
3f0bc9aaf21f: Pull complete
ccd42975ca3b: Pull complete
7698e7e04700: Pull complete
3e715743c49b: Pull complete
6234f5b8155d: Pull complete
Digest: sha256:41532e0f2765259e282fa397b214a8333c6359331a2254df58a4c04e4994d172
Status: Downloaded newer image for selenium/node-chrome:2.45.0
root@irco3sd5vm3 [ ~/dockerComposeFiles/selstartup ]# docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED
    ↪            SIZE
```

```
selenium/node-firefox-debug   2.53.0              45a43e79687e        6 months
    ↪ ago         657.3 MB
selenium/hub                  2.53.0              bb88d60869b6        6 months
    ↪ ago         317.9 MB
selenium/node-chrome          2.45.0              78d0952fd0c7        22
    ↪ months ago       694.3 MB
```

Another factor to consider is that a lot of the configuration that would normally be a manual process is taken care of in the container images. The selenium/node-firefox-debug image for example comes with the Xvfb and a Virtual Network Computing (VNC) server installed within the container. If the end user wishes to visually observe the browser as the tests run, it is then a straightforward process for the user to simply find the port to open a Virtual Network Computing (VNC) connection to the container. The Figure below shows the command to find the Container ID and Port information for each container. This enables the user to find the port than the VNC server is exposed to. In the example below, container 0da284170af5 is connectable via port 32785. The VNC client would connect through irco3sd5vm3:32785

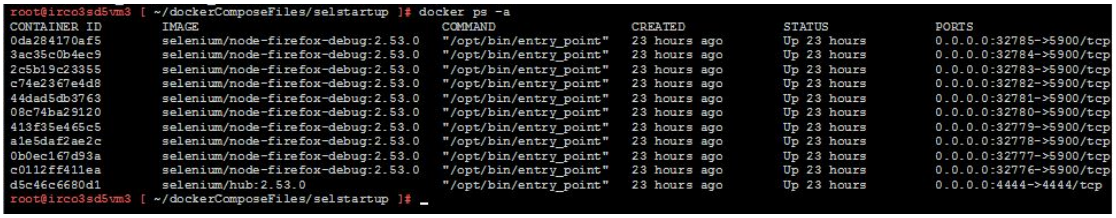


FIGURE 4.9: Docker PS to Get Port information

On the RHEL and SUSE environments, all of this is a much more manual process. It is up to the user to manage all libraries and dependencies. The Selenium Jars needed for the Grid need to be kept up to date. Browser drivers need to be downloaded to the machine. The user needs to get the Xvfb rpms and set up the linux DISPLAY to run on a port. If running the tests headlessly VNC needs to be setup. This can be time consuming configuration and daunting for those unfamiliar with Linux. There are also problems around support of different softwares on these operating systems. It was only after acquiring and installing the RHEL 6 VMs that it was discovered that Chrome was not supported. A lot of these problems do not arise with containers as all the component pieces are set up to work together within the container. Even when newer software such

as Selenium 3, or the latest browser versions are released it can be more easily managed, even on older operating systems as long as they can support containers.

One major drawback of the container solution is the lack of support for Internet Explorer at this time.

Another benefit of the containerized solution should be that since each Selenium processes is now self-contained (every node is run within its own container and is isolated from the others) then there should never be problems with focus event handling as when running multiple tests/browsers in parallel on the same VM. There has been an ongoing issue with browsers not triggering focus/blur events, especially when running tests in parallel. Initially it was thought that dealing with this issue would be one of the cornerstones of the study. The expectation was to encounter some focus problems when kicking off heavy loads of concurrent tests on the RHEL and SUSE environments but in fact this did not materialize. It would appear that Selenium code itself is handling this efficiently, which according to some blogs and StackOverflow it did not do before [71] [72].

Chapter 5

Conclusions and Future Work

5.1 Discussion

The purposed of this study was to answer the research questions as they apply to each of the three different environments:

a) A physical machine
b) A virtual machine
c) A container optimized virtual machine

The questions posed were:

a) How will the performance of a Test Automation suite be affected by being run on each environment?
b) What other factors should be considered, outside of performance, when deciding which environment best suits a Test Automation framework?

In order to further enhance the study, the virtual machine and container optimized virtual machine environments were sub-divided into two distinct environments, where one environment had a single VM that had all the host resources, and a second environment that split resources between four VMs.

The results clearly show that the *"Physical Machine"* performs best in terms of pass rates and execution time. It is easily the most performant system. It should also be noted that the physical machine has an inferior CPU yet still performs better. It can be claimed that this performance should be expected as it does not have any communication abstraction layers such as hypervisor, or container engine to bypass like the other systems, but

nonetheless the difference in performance is stark. It must be noted that the physical machine runs SLES 11, while the VMs run RHEL 6, and the container optimized VMs run Photon OS 1.0.

The performance of the virtual machine environments and the containerized environments is much closer in terms of passed tests. The *"4 VMs* environment is the worst performing of these four, with possible reasons given in the analysis section. The pass rate differences between the other three environments is quite small. All scored 100% on Scenario 3, and with only 0.5% difference between all three on Scenario 5.

In terms of test execution times, there was a noticeable trend in that the RHEL VMs finished their test suites faster than the Photon VMs on each of the three Scenarios in the sample. The RHEL VMs were 1% faster on Scenario 1, 15% faster on Scenario 2, and 13% faster on scenario 3.

As regards an opinion on operational flexibility and ease of use, the benefits of virtualization and containerization in general as described in Chapter 2, come into play here. There is no capability to take point in time snapshots of the state of Physical Machine. Also, the Physical Machine is only capable of supporting one operating system so browser/operating support matrix coverage is immediately limited to Linux in this instance. The virtual machine and container optimized virtual machines are both run on the ESXi hypervisor getting all of the functionality that comes with vSphere. This includes snapshot creation and management, as well as its Distributed Power Management (DPM), Distributed Resource Scheduler (DRS) and High availability (HA) functionality [73] . Using these features gives the Test team much control and flexibility over the environment, and leaves it much less likely that a machine going down will affect the test run. Having a short boot time is also a plus in the case of a machine going down and needing to be restarted as discussed in Chapter 4.

The Selenium Suite of software, and the configuration needed is moderately complex. On the physical machine and the virtual machine this setup is done manually as described in Appendix B.2 . It is up to the user to ensure that all software components are compatible with one another and work correctly. On the container optimized virtual machine it is possible to pull down a Docker image from DockerHub with all the required parts and simply start up the container. Software versioning management is also simplified considerably. If the user needs to run a certain number of tests, it is very easy to spin up

that number Selenium Node containers, use them, then destroy them and to have those host resources returned to the host using Dockers scaling and container management functionality. Also, Docker containers can be connected without the need to expose container ports externally, so Node containers do not need public IPs, only the Hub needs to be on the external network. This saves on IP address space which may be a concern for some businesses. It is not so easy to provision and deprovision full Selenium Node VMs in this way. The big drawback of the Docker Selenium solution is the lack of support for IE, which is still a big player in the Enterprise market.

5.2 Conclusion

In this study the performance of a Test Automation suite was measured across the different environments and the operational flexibility of each of these was considered throughout the study's implementation. The conclusion is that in terms of pure "horse-power", the physical machine will run more tests, quicker and in a more reliable fashion than either the virtual or containerized solutions. This has been well established across numerous test runs. If a test team is interested in purely running as many browser instances as possible simultaneously then this is the solution to choose.

The execution speed on the virtual machine environments is demonstrably faster than the containerized environments once the concurrency starts to rise, and the pass rates on these test runs are almost identical. So it is fair to say that the virtual machine environments are performing better at a point when the automation is useful and would be the next best choice for a test team. After that, when the level of parallel tests rise, tests begin to fail and in this instance the environment with four virtual machine performs notably worse than the others. Based on these results, it appears to be the worst performing once the test load becomes heavy.

Another conclusion to be made from this study is that CPU appears to be the crucial factor when trying to ensure that the framework runs reliably at high levels of concurrency. It could be seen when running a physical machine with 16 cores that the framework could not cope, in contrast to the 24 core physical machine which was able to run everything thrown at it without any issue.

Also, with CPU performance in mind, the study recommends to stagger the execution of the test run, especially at the very start, so that lots of browser processes do not start simultaneously causing excessive CPU load and subsequent failed tests. By doing this it may be possible keep the CPU load at an acceptable level and pass rates may improve.

As regards Operational Flexibility and Ease of Use, it has been the experience of this study that the Containerized solution is the most flexible for setting up a Selenium Grid quickly with minimum configuration. The Hub and Node configuration and scaling can be managed simply using Docker Compose. If only Chrome and Firefox support are required for the tests in question, and heavy levels of concurrent testing are not needed, then this is the recommended solution to use. The boot time is also easily the fastest, so if teams are part of an effort to build their own private Cloud or even just to bring more automation and orchestration to their Data Center, this could be a key advantage.

The virtualized solution is the next best, in terms of this operational flexibility. Like the containerized solution, it has all of the vSphere functionality, but the configuration of the Selenium Grid is a manual process, as is keeping all of the software up to date. It does however support Internet Explorer, since it is possible to provision Windows VMs, which depending on the team, could be a deciding factor.

5.3 Future Work

There is scope for future research on this project. Some possibilities are enumerated below:

1. It was not possible to source completely identical host machines for this study. The Physical machine running SLES 11 is a Dell PowerEdge R430 with a different chipset to the virtualized environment. It would be interesting to see how the results would be effected if these environments were identical.

2. It would also be interesting to see the performances by adding more resources to the physical machines. For example, would having 64GB memory and 48 cores give a two fold increase in speed and reliability? It was shown that the physical environment could easily run 66 threads without a problem so it is possible that a ”beefed up” physical server could run huge numbers of browser sessions.

3. Another possible area of investigation is to come up with a way of staggering the start up of tests based on the load of the CPU in order to avoid the spikes of activity that seem to cause most of the problems. The test runs are characterized by these peaks and troughs of CPU activity, so if it was possible to smooth this out, there is potential to have a much more efficient and reliable test run.

4. A final area of investigation would be to include Chrome in the study. This was not possible to do this time due to the unavailability of the operating systems needed.

Bibliography

[1] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, *Guide to the software engineering body of knowledge-SWEBOK*. IEEE Press, 2001.

[2] C. Mike, "The forgotten layer of the test automation pyramid. e-document." [Online]. Available: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[3] "Functional testing." [Online]. Available: http://softwaretestingfundamentals.com/functional-testing/

[4] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[5] S. Ng, T. Murnane, K. Reed, D. Grant, and T. Chen, "A preliminary survey on software testing practices in australia," in *Software Engineering Conference, 2004. Proceedings. 2004 Australian*. IEEE, 2004, pp. 116–125.

[6] S. Brian E F, "Qa and testing budgets, getting a grip on test spending," 2017. [Online]. Available: https://www.sogeti.com/globalassets/global/downloads/testing/wqr-2016-2017/wqr_2016-17_final_secure.pdf

[7] Wikipedia, "Devops." [Online]. Available: https://en.wikipedia.org/wiki/DevOps

[8] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 561–570.

[9] "What is docker," 2017. [Online]. Available: https://www.docker.com/what-docker

[10] L. Crispin and J. Gregory, *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.

[11] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison," in *21st Twente Student Conference on IT*, 2014, pp. 1–7.

[12] B. Agrawal, "Docker containers performance in vmware vsphere - vmware vroom! blog," 2014. [Online]. Available: https://blogs.vmware.com/performance/2014/10/docker-containers-performance-vmware-vsphere.html

[13] SeleniumHQ, "What is selenium?" [Online]. Available: http://www.seleniumhq.org/

[14] Atlassian, "Top software development trends in 2016." [Online]. Available: https://www.atlassian.com/blog/software-teams/software-development-trends-2016

[15] E. Dustin, J. Rashka, and J. Paul, *Automated software testing: introduction, management, and performance.* Addison-Wesley Professional, 1999.

[16] K. Beck and M. Fowler, *Planning extreme programming.* Addison-Wesley Professional, 2001.

[17] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "The agile manifesto," 2001.

[18] E. F. Collins *et al.*, "Software test automation practices in agile development environment: An industry experience report," in *Proceedings of the 7th International Workshop on Automation of Software Test.* IEEE Press, 2012, pp. 57–63.

[19] D. Burns and S. Stewart, "Webdriver," W3C, W3C Working Draft, Mar. 2017, https://www.w3.org/TR/2017/WD-webdriver-20170308/.

[20] J. Marini, *Document object model.* McGraw-Hill, Inc., 2002.

[21] "Browser automation." [Online]. Available: http://www.seleniumhq.org/projects/webdriver/

[22] Angular, "angular/protractor," Mar 2017. [Online]. Available: https://github.com/angular/protractor

[23] "Julie ralph, protractor timeout," October 2013. [Online]. Available: https://github.com/angular/protractor/issues/169

[24] "Why is thread.sleep so harmful," Sept 2015. [Online]. Available: http://stackoverflow.com/questions/8815895/why-is-thread-sleep-so-harmful

[25] "How to avoid thread.sleep in test automation," August 2016. [Online]. Available: http://stackoverflow.com/questions/8815895/why-is-thread-sleep-so-harmful

[26] "Html5 test automation." [Online]. Available: https://smartbear.com/product/testcomplete/web-module/html5-testing/

[27] "Abandoning the narrative of containers vs. hypervisors." [Online]. Available: http://searchservervirtualization.techtarget.com/tip/Abandoning-the-narrative-of-containers-vs-hypervisors

[28] M. Plauth, L. Feinbube, and A. Polze, "A performance evaluation of lightweight approaches to virtualization," *CLOUD COMPUTING 2017*, p. 14, 2017.

[29] V. JURENKA, "Virtualizace pomocí platformy docker," Ph.D. dissertation, Masarykova univerzita, Fakulta informatiky, 2015.

[30] "It all starts with coreos container linux," 2017. [Online]. Available: https://coreos.com/why

[31] A. Aspernäs and M. Nensén, "Container hosts as virtual machines: A performance study," 2016.

[32] "Docker overview," 2017. [Online]. Available: https://docs.docker.com/engine/docker-overview/

[33] ClusterHQ, "Container market adoption survey : 2016," Tech. Rep., 2016. [Online]. Available: https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf

[34] "Introducing infrakit, an open source toolkit for creating and managing declarative, self-healing infrastructure," October 2016. [Online]. Available: https://blog.docker.com/2016/10/introducing-infrakit-an-open-source-toolkit-for-declarative-infrastructure/

[35] "rkt fetch," 2017. [Online]. Available: https://coreos.com/rkt/docs/latest/subcommands/fetch.html/

[36] "Manage docker resources with cgroups," 2017. [Online]. Available: https://www.cloudsigma.com/manage-docker-resources-with-cgroups/

[37] "Get off my lawn, separating docker workloads using cgroups," 2017. [Online]. Available: https://sthbrx.github.io/blog/2016/07/27/get-off-my-lawn-separating-docker-workloads-using-cgroups/

[38] "Docker basics webinar qa understanding union filesystems," 2015. [Online]. Available: https://blog.docker.com/2015/10/docker-basics-webinar-qa/

[39] "Welcome to docker hub," 2017. [Online]. Available: https://hub.docker.com/

[40] "Docker images for selenium standalone server," 2017. [Online]. Available: https://github.com/SeleniumHQ/docker-selenium

[41] VMWARE, "Developing Container Applications with vSphere Integrated Containers Engine," Tech. Rep., 2017. [Online]. Available: https://vmware.github.io/vic-product/assets/files/pdf/0.8/vic_app_dev.pdf

[42] "Vmware photon: Minimal linux container host." [Online]. Available: https://github.com/vmware/photon

[43] R. H. E. Linux, "Red hat enterprise linux 6," 2010.

[44] "Virtualization with linux containers (lxc) suse linux enterprise server 11 sp4," 2015. [Online]. Available: https://www.suse.com/documentation/sles11/singlehtml/lxc_quickstart/lxc_quickstart.html

[45] S. Gnanasundaram and A. Shrivastava, *Information Storage and Management: Storing, Managing, and Protecting Digital Information in Classic, Virtualized, and Cloud Environments.* John Wiley & Sons, 2012.

[46] S. Milanovic and Z. Petrovic, "Building the enterprise-wide storage area network," in *EUROCON'2001, Trends in Communications, International Conference on.*, vol. 1. IEEE, 2001, pp. 136–139.

[47] "Host bus adapter (hba) architecture explained," 2008. [Online]. Available: http://searchstorage.techtarget.com/feature/HBAs-explained

[48] VMWARE, "San conceptual and design basics," Tech. Rep., 2006. [Online]. Available: https://www.vmware.com/pdf/esx_san_cfg_technote.pdf

[49] EMC, "Storage Provisioning: Transitioning to Symmetrix VMax AutoProvisioning Groups - A detailed review," Tech. Rep., 2009.

[Online]. Available: https://www.emc.com/collateral/hardware/white-papers/h6547-storage-provisioning-v-max-auto-provisioning-groups-wp.pdf

[50] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 171–172.

[51] M. Bures and M. Filipsky, "Smartdriver: Extension of selenium webdriver to create more efficient automated tests," in *IT Convergence and Security (ICITCS), 2016 6th International Conference on*. IEEE, 2016, pp. 1–4.

[52] M. Kuutila, M. Mäntylä, and P. Raulamo-Jurvanen, "Benchmarking web-testing-selenium versus watir and the choice of programming language and browser," *arXiv preprint arXiv:1611.00578*, 2016.

[53] A. Holmes and M. Kellogg, "Automating functional tests using selenium," in *Agile Conference, 2006*. IEEE, 2006, pp. 6–pp.

[54] G. SeleniumHQ, "Remotewebdriver." [Online]. Available: https://github.com/SeleniumHQ/selenium/wiki/RemoteWebDriver

[55] Oracle, "Threadlocal." [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html

[56] "Martin fowler, pageobject," September 2013. [Online]. Available: https://martinfowler.com/bliki/PageObject.html

[57] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Improving test suites maintainability with the page object pattern: An industrial case study," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 108–113.

[58] SeleniumHQ, "The pagefactory." [Online]. Available: https://github.com/SeleniumHQ/selenium/wiki/PageFactory

[59] B. Cdric, "Testng," 2015. [Online]. Available: http://testng.org/doc/

[60] T. A. S. Foundation, "Introduction to the standard directory layout," 2017. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

[61] G. Kevin, "Dockerwork," 2017. [Online]. Available: https://github.com/gearyk

[62] Bugzilla, "Bug 1103196 support for untrusted self-signed tls certs," 2017. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1103196

[63] L. Henrichsen, M. T. Smith, and D. S. Baker, "Byu department of linguistics," *Taming the Research Beast*, 1997.

[64] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[65] Linux, "Linux man pages - top." [Online]. Available: http://man7.org/linux/man-pages/man1/top.1.html

[66] E. Duncan, "Esxtop." [Online]. Available: http://www.yellow-bricks.com/esxtop/

[67] Wikipedia, "Headless browser." [Online]. Available: https://en.wikipedia.org/wiki/Headless_browser

[68] "Intel xeon x5650 @ 2.67ghz," April 2017. [Online]. Available: https://cpubenchmark.net/cpu.php?cpu=Intel+Xeon+X5650+%40+2.67GHz&id=1304&cpuCount=2

[69] "Intel xeon x5650 @ 2.67ghz," April 2017. [Online]. Available: https://cpubenchmark.net/cpu.php?cpu=Intel+Xeon+X5650+%40+2.67GHz&id=1304&cpuCount=2

[70] vfrank, "Esxtop vfrank." [Online]. Available: http://www.vfrank.org/esxtop/

[71] H. Koch, "How to solve selenium focus issues." [Online]. Available: https://makandracards.com/makandra/12661-how-to-solve-selenium-focus-issues

[72] Github, "Focus and blur events not fired when browser has no focus." [Online]. Available: https://github.com/seleniumhq/selenium-google-code-issue-archive/issues/7346

[73] VMWARE, "Introduction to vmware drs and vmware ha clusters." [Online]. Available: https://pubs.vmware.com/vsphere-50/index.jsp?topic=%2Fcom.vmware.wssdk.pg.doc_50%2FPG_Ch13_Resources.15.6.html

Appendix A

Code Snippets

A.1 WebDriver Manager Class

This is the main class in the application. It is responsible for starting the browser sessions, deciding which hub to use, which instance of Univmax to run the tests on and managing the multithreading. It is also responsible for closing the browser session, and takes a screenshot if the test fails. If these parameters are not specified, the default values as set by @Optional parameters will be used.

```
package com.cit.ie.base;

import java.io.File;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;

import org.apache.commons.io.FileUtils;
import org.apache.http.HttpHost;
import org.apache.http.HttpResponse;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.message.BasicHttpEntityEnclosingRequest;
import org.apache.http.util.EntityUtils;
import org.json.JSONException;
```

```
import org.json.JSONObject;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.UnexpectedAlertBehaviour;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.firefox.FirefoxProfile;
import org.openqa.selenium.remote.CapabilityType;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.HttpCommandExecutor;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Optional;
import org.testng.annotations.Parameters;



/**
* @author gearyk2
* @description Create an instance of WebDriverManager for each thread
*/
@SuppressWarnings("deprecation")
public class WebDriverManager {

   protected ThreadLocal<RemoteWebDriver> threadDriver = null
   public static String appURL;

   @Parameters({"browser","hubIP","applicationURL"})
   @BeforeClass()
   public void launchbrowser(@Optional("firefox")String
    ↪ browser,@Optional("10.60.141.26")String
    ↪ hubIP,@Optional("https://10.60.141.19:8443/univmax/") String
    ↪ applicationURL) throws MalformedURLException, InterruptedException {

      appURL=applicationURL;
      System.out.println("Setting URL to : "+appURL);
      if (browser.equalsIgnoreCase("chrome")) {
```

```
        System.out.println(" Executing on CHROME");
        DesiredCapabilities cap = DesiredCapabilities.chrome();
        cap.setBrowserName("chrome");
        String Node = "http://"+hubIP+":4444/wd/hub";
        threadDriver = new ThreadLocal<RemoteWebDriver>();
        threadDriver.set(new RemoteWebDriver(new URL(Node), cap));
        long id = Thread.currentThread().getId();
        System.out.println("TestRunning. Thread id is: " + id);
    }

    if (browser.equalsIgnoreCase("firefox")) {
        System.out.println(" Executing on FIREFOX");
        DesiredCapabilities cap = DesiredCapabilities.firefox();
        cap.setBrowserName("firefox");
        cap.setCapability("marionette", false);
        cap.setCapability(FirefoxDriver.PROFILE, new FirefoxProfile());
        cap.setCapability(CapabilityType.ACCEPT_SSL_CERTS, true);

↪ cap.setCapability(CapabilityType.UNEXPECTED_ALERT_BEHAVIOUR,UnexpectedAlertBehaviour.AC
        String Node = "http://"+hubIP+":4444/wd/hub";
        threadDriver = new ThreadLocal<RemoteWebDriver>();
        threadDriver.set(new RemoteWebDriver(new URL(Node), cap));
        long id = Thread.currentThread().getId();
        System.out.println("TestRunning. Thread id is: " + id);
    }

}

public String getApplicationURL(){
    return appURL;
}

public WebDriver getDriver()
{
if(threadDriver==null){
    return driver.get();
}
else{
```

```
    System.out.println("returning thread driver");
    return threadDriver.get();
    }
}

@AfterClass
public void closeBrowser() throws IOException {
    if(!getDriver().toString().contains("null"))
    {
        System.out.println("Driver has not been closed. Take screenshot and
↪ close");
        File scrFile =
↪ ((TakesScreenshot)getDriver()).getScreenshotAs(OutputType.FILE);
        String SessionID=threadDriver.get().getSessionId().toString();
        FileUtils.copyFile(scrFile, new
↪ File("c:\\screenshot\\screenshot_"+SessionID+".png"));

        System.out.println("Browser Still Open: ");
        getDriver().quit();
        long id = Thread.currentThread().getId();
        System.out.println("Webdriver quit for thread: " + id);
    }
    else
    {
        //driver.get().quit();
    }
}

@SuppressWarnings("resource")
public void findRemote(RemoteWebDriver driver) throws
↪ IOException,JSONException {
    HttpCommandExecutor ce =
↪ (HttpCommandExecutor)driver.getCommandExecutor();
    ce.getAddressOfRemoteServer();
    ce.getAddressOfRemoteServer().getHost();
    String HubIP=ce.getAddressOfRemoteServer().getHost();
    int HubPort=ce.getAddressOfRemoteServer().getPort();
```

```
      HttpHost host = new HttpHost(ce.getAddressOfRemoteServer().getHost(),
    ↪ HubPort);
      DefaultHttpClient client = new DefaultHttpClient();
      URL testSessionApi = new URL("http://" + HubIP + ":" + HubPort +
    ↪ "/grid/api/testsession?session=" +  driver.getSessionId());
      BasicHttpEntityEnclosingRequest r = new
    ↪ BasicHttpEntityEnclosingRequest("POST",
    ↪ testSessionApi.toExternalForm());
      HttpResponse response  = client.execute(host,r);
      System.out.println(response);
      System.out.println(response.getEntity());
      JSONObject object = new
    ↪ JSONObject(EntityUtils.toString(response.getEntity()));
      @SuppressWarnings("unused")
      String proxyID =(String) object.get("proxyId");
      System.out.println("in find remote - leaving");
   }

}
```

A.2 Sample Page Object

This the Page Object class for the Home Dashboard. It shows the WebElement locator Strings, the WebElement declarations using PageFactory, and the services provided by this PageObject which are the void methods.

```
package com.cit.ie.pageobjects;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
import org.openqa.selenium.support.ui.WebDriverWait;

import com.cit.ie.base.HelperMethods;
```

```
/**
* @author gearyk2
* @description Page Object Class for HomeDashboard
*/
public class HomeDashboardPO extends HelperMethods {

   public HomeDashboardPO(WebDriver wdriver) {
      super(wdriver);
      driver = wdriver;
      wait = new WebDriverWait(driver, timeOut);
      PageFactory.initElements(driver, this);
   }

   //LOCATOR STRINGS
   public final String DELL_EMC_LOGO_XPATH = ".//*[@id='dell-emc-logo']";
   public final String U4V_LOGO_XPATH = "//span[text()[contains(.,'Unisphere
    ↪ for VMAX')]]";
   public final String
    ↪ STORAGE_SIDEMENU_XPATH=".//a[@ng-if='item.items']/div[text()='STORAGE']";
   public final String STORAGE_GROUPS_SIDEMENU_XPATH="//nav[@class='ng-scope
    ↪ emc-framework-menu-secondary-sub
    ↪ sub-menu-display']/menu/ul/li[contains(@class,'emc-menu-subitem-storage_groups')]/a/div
   public final String USER_PROFILE_XPATH="//button[@aria-label='User
    ↪ Profile']";
   public final String LOGOUT_XPATH="//button[@aria-label='Logout']";
   public final String LOGOUT_CONFIRM_XPATH="//button/span[text()='OK']";

   //WEB ELEMENTS
   @FindBy(xpath=USER_PROFILE_XPATH)
   public WebElement userProfileButton;
   @FindBy(xpath=LOGOUT_XPATH)
   public WebElement logoutButton;
   @FindBy(xpath=LOGOUT_CONFIRM_XPATH)
   public WebElement logoutConfitmButton;
   @FindBy(xpath=DELL_EMC_LOGO_XPATH)
   public WebElement dellEmcLogo;
```

```
    @FindBy(xpath=U4V_LOGO_XPATH)
    public WebElement u4vLogo;
    @FindBy(xpath=STORAGE_SIDEMENU_XPATH)
    public WebElement storageSideMenuItem;
    @FindBy(xpath=STORAGE_GROUPS_SIDEMENU_XPATH)
    public WebElement storageGroupsSideMenuItem;

    public void waitForHomeDashboardPageObjects() throws InterruptedException{
        waitForElementVisiblity(U4V_LOGO_XPATH);
    }

    //HOMEDASHBOARD SERVICES

    //Navigate to this view
    public void navigateToSideMenu() throws InterruptedException{
        jsClickElement(storageSideMenuItem);
    }

    //Navigate to this view
    public void navigateToStorageGroups() throws InterruptedException{
        navigateToSideMenu();
        jsClickElement(storageGroupsSideMenuItem);
    }

    public void doLogout() throws InterruptedException{
        jsClickElement(userProfileButton);
        jsClickElement(logoutButton);
        jsClickElement(logoutConfirmButton);
    }

    //Quit the webdriver
    public void quitWebDriver() throws InterruptedException{
        doLogout();
        driver.quit();
    }

}
```

A.3 Sample TEST

Create an Empty Storage Group and set the SRP, SLO and Workload values to None.

```
@Test(priority=1)
private void _004_CREATE_STORAGEGROUP_EMPTYSETTOTRUE_SRPNONE_SLONONE_WLNONE()
    ↪ throws JSONException, IOException, InterruptedException {
   HelperMethods.printTimeStart("Test004");
   sgName="00DC04";
   if(threadDriver!=null)
   {
      findRemote(threadDriver.get());
   }
   LoginPagePO lppo=new LoginPagePO(getDriver());
   lppo.gotoStorageGroupsPage(lppo,sgName);
   StorageGroupsPO sgpo=new StorageGroupsPO(getDriver());
   sgpo.waitForStorageGroupsPageObjects();
   sgpo.jsClickElement(sgpo.createStorageGroupButton);
   ProvisionStorageWizardPO pswpo=new ProvisionStorageWizardPO(getDriver());
   pswpo.waitForElementVisiblity(pswpo.WAIT_FOR_PAGELOAD);
   pswpo.jsClickElement(pswpo.storageGroupNameTextField);
   pswpo.storageGroupNameTextField.sendKeys(sgName);
   //SET SRP
   pswpo.setSrpInformation(pswpo,"None");
   pswpo.setSloInformation(pswpo, "None");
   //SET WORKLOAD
   //Leave as Unspecified
   pswpo.jsClickElement(pswpo.selectRunMethodMenu);
   pswpo.jsClickElement(pswpo.createSgRunNow);
   sgpo.waitForElementToDisappear(pswpo.TASK_IN_PROCESS_XPATH);
   sgpo.quitWebDriver();
   pswpo.verifyAndCleanup(sgName);
   HelperMethods.printTimeFinish("TEST004");

}
```

A.4 REST Client

This the RESTClient Code. This class allows static calls to the Unisphere RESTAPI. It is used for verification and cleanup of test data.

```
@SuppressWarnings("deprecation")
public class RESTClient extends Constants
{
   public static String responseFormat = "json";
   public static String username, password = null;
   public static int responseStatus;
   public static Status responseStatusText;
   public static String responseOutput;
   public static URI responseLocation;
   public static String previousCall;
   public static String url;
   public static String param;
   public static String parsed, value = null;

   /**
   * @name GETString
   * @description REST GET call
   * @param restCall
   * @return responseOutput
   */
   public static String GETString(String restCall) {
      GET(restCall);
      return responseOutput;
   }

   /**
   * @name GET
   * @description REST GET call
   * @param restCall
   * @return responseStatus
   */
   public static int GET(String restCall) {
```

```
    url = restCall;
    previousCall = GET;
    if(restCall.contains(" ")) {
       restCall = restCall.replaceAll(" ", "%20");
    }
    ssl();
    if(username == null && password == null){
       username = "smc";
       password = "smc";
    }
    String authString = username+":"+password;
    String authStringEnc =
↪ Base64.getEncoder().encodeToString(authString.getBytes());
    Client client = Client.create();
    WebResource wr = client.resource(restCall);
    ClientResponse response = null;
    if(isJson()) {
       response = wr.accept("application/json").header("Authorization",
↪ "Basic " + authStringEnc)
       .get(ClientResponse.class);
    } else if(isXml()){
       response = wr.accept("application/xml").header("Authorization",
↪ "Basic " + authStringEnc)
       .get(ClientResponse.class);
    }

    responseLocation = response.getLocation();
    responseOutput = response.getEntity(String.class);
    responseStatus = response.getStatus();
    responseStatusText = response.getResponseStatus();
    return responseStatus;
}

/**
* @name DELETEString
* @description REST DELETE call
* @param restCall
* @return responseOutput
```

```
*/
public static String DELETEString(String restCall) {
   DELETE(restCall);
   return responseOutput;
}

/**
* @name DELETE
* @description REST DELETE call
* @param restCall
* @return responseStatus
*/
public static int DELETE(String restCall) {
   url = restCall;
   previousCall = DELETE;
   ssl();
   if(username == null && password == null){
      username = "smc";
      password = "smc";
   }
   String authString = username+":"+password;
   String authStringEnc =
↪ Base64.getEncoder().encodeToString(authString.getBytes());
   Client client = Client.create();
   WebResource wr = client.resource(restCall);
   ClientResponse response = null;
   if(isJson()){
      response = wr.type("application/json").header("Authorization", "Basic
↪ " + authStringEnc)
      .delete(ClientResponse.class);
   } else if(isXml()){
      response = wr.type("application/xml").header("Authorization", "Basic
↪ " + authStringEnc)
      .delete(ClientResponse.class);
   }

   responseLocation = response.getLocation();
   responseStatus = response.getStatus();
```

```
        responseStatusText = response.getResponseStatus();
        if (responseStatus != 204) {
            responseOutput = response.getEntity(String.class);
        }

        return responseStatus;
    }

    /**
    * @name POSTString
    * @description REST POST call
    * @param restCall
    * @return responseOutput
    */
    public static String POSTString(String restCall, String params) {
        POST(restCall, params);
        return responseOutput;
    }

    /**
    * @name POST
    * @description REST POST call
    * @param restCall, params
    * @return responseStatus
    */
    public static int POST(String restCall, String params) {
        url = restCall;
        param = params;
        previousCall = POST;
        ssl();
        if(username == null && password == null){
            username = "smc";
            password = "smc";
        }
        String authString = username+":"+password;
        String authStringEnc =
    ↪ Base64.getEncoder().encodeToString(authString.getBytes());
        Client client = Client.create();
```

```
    WebResource wr = client.resource(restCall);

    ClientResponse response = null;
    if(isJson()) {
       response = wr.type("application/json").accept("application/json")
       .header("Authorization", "Basic " +
↪ authStringEnc).post(ClientResponse.class, params);
    } else if(isXml()) {
       response = wr.type("application/xml").accept("application/json")
       .header("Authorization", "Basic " +
↪ authStringEnc).post(ClientResponse.class, params);
    }

    responseLocation = response.getLocation();
    responseOutput = response.getEntity(String.class);
    responseStatus = response.getStatus();
    responseStatusText = response.getResponseStatus();
    return responseStatus;
}

/**
* @name PUTString
* @description REST PUT call
* @param restCall, params
* @return responseOutput
*/
public static String PUTString(String restCall, String params) {
    PUT(restCall, params);
    return responseOutput;
}

/**
* @name PUT
* @description REST PUT call
* @param restCall, params
* @return responseStatus
*/
public static int PUT(String restCall, String params) {
```

```
    url = restCall;
    param = params;
    previousCall = PUT;
    ssl();
    if(username == null && password == null){
       username = "smc";
       password = "smc";
    }
    String authString = username+":"+password;
    String authStringEnc =
↪ Base64.getEncoder().encodeToString(authString.getBytes());
    Client client = Client.create();
    WebResource wr = client.resource(restCall);

    ClientResponse response = null;
    if(isJson()) {
       response = wr.type("application/json").accept("application/json")
       .header("Authorization", "Basic " +
↪ authStringEnc).put(ClientResponse.class, params);
    } else if(isXml()) {
       response = wr.type("application/xml").accept("application/json")
       .header("Authorization", "Basic " +
↪ authStringEnc).put(ClientResponse.class, params);
    }

    responseLocation = response.getLocation();
    responseStatus = response.getStatus();
    responseStatusText = response.getResponseStatus();
    if (responseStatus != 204) {
       responseOutput = response.getEntity(String.class);
    }

    return responseStatus;
}

/**
* @name ssl
* @description Bypass ssl certs
```

```
*/
public static void ssl() {
   SSLContext sc = null;
   try {
      sc = SSLContext.getInstance("TLSv1.2");
   } catch (NoSuchAlgorithmException e1) {
      e1.printStackTrace();
   }
   try {
      sc.init(null, new TrustManager[] { new TrustAllX509TrustManager() },
 ↪ new java.security.SecureRandom());
   } catch (KeyManagementException e) {
      e.printStackTrace();
   }
   HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
   HttpsURLConnection.setDefaultHostnameVerifier( new HostnameVerifier(){
      public boolean verify(String string, SSLSession ssls) {
         return true;
      }
   });
   HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
}



/**
* @name refreshRestDB
* @description A simple post to refresh the REST Database
* @param url
* @example
 ↪ https://10.73.28.231:8443/univmax/restapi/sloprovisioning/symmetrix/000196700348/storag
*/
public static void refreshRestDB(String storageGroupurl) {
   String rand=UUID.randomUUID().toString().replaceAll("-",
 ↪ "").substring(0, 6);
   String json="{\"srpId\": \"None\",\"storageGroupId\":
 ↪ \"Omadeup"+rand+"\",\"create_empty_storage_group\": true}";
   POST(storageGroupurl,json);
   printResponses();
```

```
        //Assert.assertEquals(responseStatus,200);
        DELETE(storageGroupurl+"0madeup"+rand);
        //printResponses();
        //Assert.assertEquals(responseStatus,204);

    }



}
```

A.5 TESTNG.XML

This XML file specifies the details of the test run. Parameters are passed to each test individually.

```
<suite verbose="1" name="FF DOCKER Suite" parallel="tests" thread-count="3">
<test name="TEST0">
<parameter name ="browser" value="firefox"/>
<parameter name ="hubIP" value="10.73.29.41"/>
<parameter name ="applicationURL" value="https://10.73.28.70:8443/univmax"/>
<classes>
<class name ="com.cit.ie.storagegroups.Test000"/>
</classes>
</test>
<test name="TEST1">
<parameter name ="browser" value="chrome"/>
<parameter name ="hubIP" value="10.73.29.41"/>
<parameter name ="applicationURL" value="https://10.73.28.70:8443/univmax"/>
<classes>
<class name ="com.cit.ie.storagegroups.Test000"/>
</classes>
</test>
<test name="TEST2">
```

```
<parameter name ="browser" value="firefox"/>
<parameter name ="hubIP" value="10.73.29.12"/>
<parameter name ="applicationURL" value="https://10.73.28.71:8443/univmax"/>
<classes>
<class name ="com.cit.ie.storagegroups.Test000"/>
</classes>
</test>
<test name="TEST4">
<parameter name ="browser" value="chrome"/>
<parameter name ="hubIP" value="10.73.29.12"/>
<parameter name ="applicationURL" value="https://10.73.28.19:8443/univmax"/>
<classes>
<class name ="com.cit.ie.storagegroups.Test000"/>
</classes>
</test>
```

Appendix B

Configuration

B.1 Configuring Swap Space on Photon VM

After initial installation there is no swap space on this Photon VM.

Before

```
root@irco3sd5vm2 [ ~ ]# free -m
total        used        free      shared  buff/cache   available
Mem:           7983        3215        4309          54         457        4627
Swap:             0           0           0
```

Run the following commands to set up 4GB swapfile

```
dd if=/dev/zero of=/swapfile bs=1G count=4
chown root:root /swapfile
chmod 0600 /swapfile
mkswap /swapfile
swapon -s
```

After

```
root@irco3sd5vm2 [ ~ ]# free -m
total        used        free      shared  buff/cache   available
Mem:           7983        3660        3847          39         475        4180
Swap:          4095        1540        2555
```

B.2 Selenium Grid installation on Linux VM

These are the steps for setting up a Selenium Hub and Node on a Linux VM to run Firefox in headless mode. Xvfb, Firefox ESR 45.06 need to be downloaded as well as the correct Selenium jar files. Xvfb is run on display port 99. There was no license for connecting to yum repositorys so wget must be used to get the rpms directly.

```
wget http://vault.centos.org/6.2/os/x86_64/Packages/
    ↪ xorg-x11-server-Xvfb-1.10.4-6.el6.x86_64.rpm
yum localinstall xorg-x11-server-Xvfb-1.10.4-6.el6.x86_64.rpm
yum remove firefox
cd /usr/local
wget https://ftp.mozilla.org/pub/firefox/releases/45.6.0esr/
    ↪ linux-x86_64/en-GB/firefox-45.6.0esr.tar.bz2
tar xvjf firefox-45.6.0esr.tar.bz2
sudo ln -s /usr/local/firefox/firefox /usr/bin/firefox
vi /etc/hosts  , add the line "127.0.0.1 localhost"
export DISPLAY=:99
/usr/bin/Xvfb :99
java -jar selenium-server-standalone-2.53.0.jar -role hub -maxSession 70 &
java -jar selenium-server-standalone-2.53.0.jar -role node -hub
    ↪ http://10.73.28.229:4444/grid/register
    ↪ -Dwebdriver.firefox.bin=/usr/local/firefox/firefox -browser
    ↪ "browserName=firefox, maxInstances=10, platform=LINUX,
    ↪ seleniumProtocol=WebDriver"
```

B.3 Selenium Grid installation on Photon OS with DockerSelenium

Creating a Selenium Grid with DockerCompose is the easiest way to do this. Create the configuration file with whatever docker images you need. In this example Selenium version 2.53.0 hub and firefox-node of the same version are used. The links section enables communication between the containers. Then start it. If the images are not present on your environment they will be fetched from the DockerHub repository.

Create yaml file with Docker instructions

```
cat > docker-compose.yml
seleniumhub:
image: selenium/hub:2.53.0
environment:
- GRID_MAX_SESSION=10
ports:
- 4444:4444
firefoxnode:
image: selenium/node-firefox-debug:2.53.0
ports:
- 5900
links:
- seleniumhub:hub
```

Start up the containers.

```
root@irco3sd5vm2 [ ~/dockerComposeFiles/selstartup ]# docker-compose up -d
```

B.4 Scaling with DockerCompose

Docker compose makes it very simple to scale up or down your environment. In the example here another 4 firefox nodes are needed. One command will scale this up.

```
root@irco3sd5vm2 [ ~/dockerComposeFiles/selstartup ]# docker-compose scale
    ↪ firefoxnode=5
Creating and starting selstartup_firefoxnode_2 ... done
Creating and starting selstartup_firefoxnode_3 ... done
Creating and starting selstartup_firefoxnode_4 ... done
Creating and starting selstartup_firefoxnode_5 ... done
```

See that there are now 5 node containers running.

```
root@irco3sd5vm2 [ ~/dockerComposeFiles/selstartup ]# docker ps --all --format
    ↪ "table {{.ID}}\t{{.Image}}"
CONTAINER ID        IMAGE
```

```
b4f017f1ff68        selenium/node-firefox-debug:2.53.0
30ceec4ce862        selenium/node-firefox-debug:2.53.0
ca3b8cc333b9        selenium/node-firefox-debug:2.53.0
c29600d1fd80        selenium/node-firefox-debug:2.53.0
051b7e88e665        selenium/node-firefox-debug:2.53.0
4af60e41d4df        selenium/hub:2.53.0
```

Scale back down to one node.

```
root@irco3sd5vm2 [ ~/dockerComposeFiles/selstartup ]# docker-compose scale
    ↪ firefoxnode=1
Stopping and removing selstartup_firefoxnode_2 ... done
Stopping and removing selstartup_firefoxnode_3 ... done
Stopping and removing selstartup_firefoxnode_4 ... done
Stopping and removing selstartup_firefoxnode_5 ... done
```

See that there are now 5 node containers running.

```
root@irco3sd5vm2 [ ~/dockerComposeFiles/selstartup ]# docker ps --all --format
    ↪ "table {{.ID}}\t{{.Image}}"
CONTAINER ID        IMAGE
051b7e88e665        selenium/node-firefox-debug:2.53.0
4af60e41d4df        selenium/hub:2.53.0
```