Distributed Testing of Cloud Applications Using the Jata Test Framework

Hlöðver Tómasson

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
University of Iceland
2011

DISTRIBUTED TESTING OF CLOUD APPLICATIONS USING THE JATA TEST FRAMEWORK

Hlöðver Tómasson

30 ECTS thesis submitted in partial fulfillment of a
Magister Scientiarum degree in Computer Science

Advisors
Helmut Wolfram Neukirchen
Helgi Þorbergsson

Faculty Representative
Hjálmtýr Hafsteinsson

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, May 2011

Distributed Testing of Cloud Applications Using the Jata Test Framework

30 ECTS thesis submitted in partial fulfillment of a M.Sc. degree in Computer Science

Faculty of Industrial Engineering, Mechanical Engineering and
Computer Science
School of Engineering and Natural Sciences
University of Iceland
Hjarðarhagi 2-6
107, Reykjavík, Reykjavik
Iceland

Telephone: 525 4000

Bibliographic information:
Hlöðver Tómasson, 2011, Distributed Testing of Cloud Applications Using the Jata Test Framework, M.Sc. thesis, Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland.

Printing: Háskólaprent, Fálkagata 2, 107 Reykjavík
Reykjavik, Iceland, May 2011

Abstract

Cloud computing is becoming a viable and common choice for software infrastructures. Quality assurance such as testing is an important aspect of any product and technology. With a new technology new challenges are introduced to quality assurance. Thus, software testing research is lacking in following the fast growing trends of the cloud computing industry.

The thesis investigates the challenges of software testing in elastic cloud computing environments and the applicability of the Jata test framework for testing distributed cloud applications. Jata uses concepts from the Testing and Test Control Notation version 3 (TTCN-3) to implement distributed test cases in Java. A cloud application case study has been performed in this thesis. It reveals that there are specific considerations to be made for testing of cloud applications to deal with the elastic nature of clouds. The work furthermore suggests that Jata is a promising framework for distributed testing, whether for cloud or non-cloud deployments.

Ágrip

Tölvuský eru orðin raunhæfur og algengur kostur fyrir hýsingu tölvukerfa. Prófanir eru mikilvægur þáttur gæðatryggingar á tæknivörum og þjónustu. Með nýrri tækni fylgja ætíð nýjar áskoranir fyrir gæðatryggingu, en rannsóknir á sviði hugbúnaðarprófana má segja að skorti samfara hinni örri þróun tölvuskýja.

Í ritgerðinni eru áskoranir hugbúnaðarprófana í tengslum við tölvuský ígrundaðar og notagildi Jata prófanakerfisins athugað í þeim tilgangi. Jata notar hugtök frá TTCN-3 staðlinum (Testing and Test Control Notation version 3) sem grunn til að styðja útfærslu dreifðra hugbúnaðarprófana. Tilraun þar sem Jata er notað til að útbúa tölvuskýprófanir er framkvæmd og lýst í þessari ritgerð. Rannsóknin leiðir í ljós atriði sem þarf að hafa sérstaklega í huga við tölvuskýsprófanir til dæmis til að bregðast við aðstæðum vegna slembiúthlutunar tölvuskýja á IP netvistföngum. Niðurstöðurnar sýna ennfremur að Jata er áhugaverður kostur fyrir hönnun og útfærslu dreifðra prófa, hvort heldur sem er fyrir tölvuský eða hefðbundnar uppsetningar.

Acknowledgements

This M.Sc. project was carried out at the *Faculty of Industrial Engineering, Mechanical Engineering and Computer Science* at the University of Iceland.

Special thanks goes to my supervisor, Associate Professor Dr. Helmut Neukirchen, for professional and enjoyable support. This thesis is inspired by his research and teaching in the field of distributed systems and software testing. I also greatly appreciate Associate Professor Dr. Helgi Þorbergsson's contributions as secondary supervisor as well as Associate Professor Dr. Hjálmtýr Hafsteinsson's efforts as faculty representative.

I would like to thank the authors of Jata; Wu, Yang, and Luo – at the Beihang University, Beijing, China – for providing the source code of the Jata framework used in this thesis and their valuable assistance at the early stages of the research. Thanks goes also to Ragnar Skulason M.Sc. student – at the University of Iceland – for providing the source code and assistance for the distributed cloud application used in this thesis case study.

Furthermore, my gratitude goes to my professors at the University of Iceland and the Lund University, Sweden, for providing the foundations this work is based on. Last, but not least, I would like to thank my family for their support and patience throughout my studies.

Reykjavík, 18. May 2011.

Hlöðver Tómasson

Contents

List of Figures

List of Tables

1. Introduction

Cloud computing is a relatively new concept which has been emerging rapidly as an alternative to traditional data-centers, grids, and private clusters. Cloud computing can be seen as a new way of outsourcing hardware and software services, and it promises that any organization can extend their IT infrastructure in a simple and efficient way. It also promises to help new businesses by requiring less upfront capital expenses for IT infrastructure.

Though cloud computing offers the potential to improve productivity and reduce costs it also poses many new challenges such as security and legal considerations, technical challenges, including software testing challenges. There are various technical challenges that need to be considered when doing research for cloud integration and it is far from trivial to integrate infrastructures to cloud models in a scalable and secure way. The construction of distributed systems within clouds poses numerous challenges on development and testing, because of their diversity and complexity. A multitude of different sub-systems and protocols make up a single distributed system, and thus implementing and maintaining a distributed system is error-prone. Reliability issues such as security, scalability, failure handling, concurrency, and network related issues such as latency and throughput are all non-trivial testing objectives. It is furthermore extremely difficult to test all possible system configurations where the software will be executed i.e. different operating systems, hardware, and software versions.

Software testing is a critical part of the quality assurance in software systems development. It is one of the most challenging and costly process activities. Research and experience show that software testing provides a strong support for the development of high quality software. Security and technical challenges of cloud computing have received a good focus in academic and commercial literature. Software testing of cloud applications has unfortunately not been as well addressed. The goal of this thesis is to improve on this.

1.1. Scope of this Thesis

The main goal of the thesis is investigation of the challenges imposed on software testing for cloud applications. For this work the Jata test framework [60] is used in a case study to implement test cases for a cloud application. Jata has recently been created to support distributed component testing. It is a Java framework which includes a meta-model for defining and structuring test cases. The meta-model uses concepts from the standardized testing language TTCN-3 (*Testing and Test Control Notation version 3*). Therefore an additional research goal was made to evaluate the applicability of Jata for advanced distributed testing like in a cloud environment.

Cloud computing is not a fundamentally new paradigm. It is based on existing distributed technologies such as utility computing and Software-as-a-Service (SaaS). Much of the testing of cloud applications can be done with traditional distributed testing methods regardless of the type of hardware or network infrastructure in use.

This thesis is largely inspired by Grid application testing methods introduced by Rings, Neukirchen and Grabowski in *Testing Grid Application Workflows Using TTCN-3* [50]. In their work TTCN-3 is used for defining and implementing test cases for a distributed application running in a Grid environment. The focus of their work is testing of the distribution management aspects of the system under test, as is also the focus in this thesis. Jata uses TTCN-3 concepts and therefore it is possible to use test structure ideas from the previous grid testing work.

This thesis focuses on black-box functional testing of distributed computing systems using the Jata test framework. A case study was made where a distributed parallel application was tested within the Amazon EC2 cloud environment. A parallel application, which carries out computing concurrently on multiple machines, was chosen in order to use advanced testing techniques. The objectives are to reveal what aspects of cloud computing need to be considered and addressed when doing software testing of cloud applications, and furthermore to evaluate the Jata framework for distributed testing.

1.2. Structure of this Thesis

An outline of the thesis is given in the following: After this introduction, the foundations for this thesis are given in Chapter 2. This includes an overview of cloud computing and the Amazon cloud services which are used in this thesis. Furthermore the chapter includes an overview of software testing, the Testing and Test Control Notation version 3 (TTCN-3), and the Jata test framework used in the case study.

Chapter 3 contains discussion of related work for this thesis. Literature on software testing of cloud application is unfortunately still lacking and this chapter surveys the work that relates to testing and cloud computing, even though the relationship with this thesis topic is not strong. The chapter is divided into three categories: *Testing in the cloud*, *testing of the cloud*, and *migrating testing to the cloud*.

The main contribution, a case study for testing a cloud application, is described in Chapter 4. The chapter describes the design and implementation of a black-box test case for a cloud application using Jata. The chapter begins with an overview of the Turnip application that is under test in the case study. The Turnip application is a cloud extension of the open-source Sunflow image rendering system. The management of the distributed application is the focus of the testing. In later sections of the chapter the test plan, test design and test case specification with Jata are covered in detail.

Subsequently, in Chapter 5, an evaluation is made concerning both software testing of cloud applications and the Jata test framework. The evaluation is based on the results of the case study.

Finally, a summary, and an outlook are given as a conclusion in Chapter 6. This thesis is completed by a list of acronyms and the referenced bibliography. An appendix on the setup of the development environment for the case study is at the end of the thesis.

2. Foundations

In this chapter background information is provided on *Cloud Computing* in Section 2.1. Section 2.2 contains a description of the *Amazon Elastic Compute Cloud (EC2)* which is used as the cloud platform for the case study in this thesis. Finally, in Section 2.3, *Software Testing* is covered.

2.1. Cloud Computing

Cloud Computing is a relatively new concept which has been emerging as an alternative to traditional data-centers, grids, and private clusters. It can help organizations to start businesses with less upfront capital expenses for IT infrastructure. It also gives the ability to scale the IT infrastructure, up or down, more easily and efficiently alongside the organization's growth.

Cloud computing is however not fundamentally a new paradigm. It is based on existing distributed technologies such as utility computing and Software-as-a-Service (SaaS). It can be seen as a new way of outsourcing hardware and software services, and it promises that any organization can extend their IT structure in a simple and efficient way.

2.1.1. Definition

Though it seems impossible to agree on a single definition for cloud computing, the seemingly most acceptable one today is provided by the U.S. National Institute of Standards and Technology (NIST):

> Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential *characteristics*, three *service models*, and four *deployment models.* [41]

The five essential characteristics are *On-demand self-service*, *Broad network access*, *Resource pooling*, *Rapid elasticity*, and *Measured Service*. The cloud service models and deployment models are covered in sections 2.1.2 and 2.1.4.

Another new and widely used definition comes from the UC Berkeley Reliable Adaptive Distributed Systems Laboratory (RAD lab):

> Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data-centers that provide those services. The services themselves have long been referred to as *Software as a Service (SaaS)*. The data-center hardware and software is what we will call a *Cloud*. When a Cloud is made available in a pay-as-you-go manner to the general public, we call it a *Public Cloud*; the service being sold is *Utility Computing*. We use the term *Private Cloud* to refer to internal data-centers of a business or other organization, not made available to the general public. Thus, Cloud Computing is the sum of SaaS and Utility Computing, but does not include Private Clouds. [7]

Many other definitions of cloud computing are available and substantial work has been done to come up with a single definition, e.g. Vaquero et al. [58] and the work of an expert group of 21 individuals [24]. Most of the definitions, as the two above, harmonize well with the version of Vaquero et al. of a minimum definition which groups a set of three common features for the different cloud computing definitions:

> *Scalability, pay-per-use utility model and virtualization.* [58]

The keyword in cloud computing is *virtualization*, which makes it possible for cloud customers to dynamically scale their computing resources, such as storing and processing capacity, to build tailored systems in an on-demand manner, paying only for the resources needed.

2.1.2. Cloud Service Models

Cloud services are mainly divided into three services delivery models [41, 58], where hardware and platform resources are provided as services in an on-demand way:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)

Figure 2.1 depicts the nature of the cloud services by comparing the levels of technology stacks provided by the different service types. The service type definitions represent the separation between the responsibilities of the consumer and the provider of the services. In case of SaaS the complete software stack is provided by the vendor, but in the other two cases it is the responsibility of the consumer to provide the remainder of the stack.

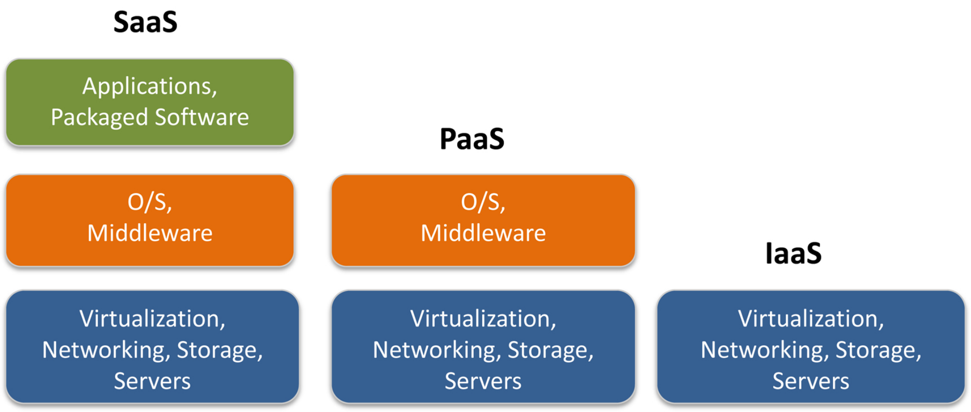

SaaS
Applications, Packaged Software
O/S, Middleware
Virtualization, Networking, Storage, Servers
PaaS
O/S, Middleware
Virtualization, Networking, Storage, Servers
IaaS
Virtualization, Networking, Storage, Servers

Figure 2.1: Cloud service types

Software as a Service

In the SaaS model the whole software stack is provided by the cloud environment; from hardware, to operating systems, to end user applications. This is an alternative to locally executed software applications and the only client software needed is a web browser. An example of SaaS is the on-line versions of typical office applications such as word processors and spreadsheet applications. A well known SaaS provider is Google with its *Google Apps* including *Google Docs* and *Google Mail*. Another major SaaS provider is *Salesforce.com*[1] which provides on-demand customer relationship management software (CRM).

Platform as a Service

The PaaS service type provides a software platform for systems, including operating systems, middleware and software development frameworks for developing applications on an abstract platform. Example of PaaS providers and solutions are *Microsoft Windows Azure* and *Google App Engine*.

[1]Salesforce.com, inc. http://www.salesforce.com/

Infrastructure as a Service

This IaaS model is basically a hardware-renting service where consumers can request computer resources and are billed based on the actual time of usage. The consumer is responsible of maintaining the software on the rented machines, including the operating system. Example IaaS providers are *Amazon*[2], *Eucalyptus*[3], *RackSpace*[4], and *GoGrid*[5].

This thesis focuses on software testing for applications within the IaaS model, and uses Amazon's *Elastic Compute Cloud (EC2)* platform as the cloud infrastructure of the test-bed.

2.1.3. Elastic and Auto-Scaling

Cloud services are elastic by nature where the number of resources are controllable at any point in time. Organizations extend or shrink its infrastructure by launching or terminating virtual machines which are called instances. One of the main objectives of cloud services is the ability to respond quickly to different loads and scale resources either up or down automatically.

The auto-scaling feature is especially valuable for applications that have unpredictable spikes in load on a regular (or irregular) basis, which allows them to be prepared and extend the resources they need on-demand. In contrast to traditional hosting, where a fixed number of resources are provided for a fixed amount of time, and any changes of load can mean problems. In that case the only way for organizations to secure themselves is by adding extra fixed resources to handle the peaks, leading to wasted resources in regular hours. Using cloud technology it is possible to be prepared for unexpected spikes in the load and utilization of resources therefore becomes more efficient.

2.1.4. Cloud Deployment Models

According to NIST's definition [41] the four cloud deployment models are:

- *Private cloud*
- *Community cloud*

[2]Amazon EC2. http://aws.amazon.com/ec2/
[3]Eucalyptus Systems, Inc. http://www.eucalyptus.com/
[4]Rackspace, US Inc. http://www.rackspace.com/
[5]GoGrid. http://www.gogrid.com/

- *Public cloud*
- *Hybrid cloud*

The selection of cloud deployment model depends on the different levels of security and control required.

The *Private cloud* infrastructure is operated solely for a single organization with the purpose of securing services and infrastructure on a private network. This deployment model offer the greatest level of security and control, but it requires the operating organization to purchase and maintain the hardware and software infrastructure, which reduces the cost saving benefits of investing in a cloud infrastructure. Rackspace, Eucalyptus, and *VMware*[6] are example providers of private cloud solutions.

A *Community cloud* infrastructure is shared by several organizations and supports a specific community that has shared concerns. It may be established where organizations have similar requirements and seek to share cloud infrastructure. Example of community cloud is Google's *Gov Cloud.*

Public clouds provide services and infrastructure over the Internet to the general public or a large industry group and is owned by an organization selling cloud services. Major public cloud providers are Google and Amazon. These clouds offer the greatest level of efficiency in shared resources, however they are also more vulnerable than private clouds.

A *Hybrid cloud* infrastructure, as the name suggests, is a composition of private, public, and/or community clouds possibly through multiple providers. Reasoning for hybrid cloud infrastructure is to increase security, better management or failover purposes. For some it may not be feasible to place assets in a public cloud, therefore many opt for the value of combining different cloud deployment models. The drawbacks of a hybrid cloud however is the requirements of managing multiple different security platforms and communication protocols.

2.1.5. Challenges

Cloud computing offers potential to improve productivity and reduce costs but it also poses many new challenges such as security and legal considerations, technical challenges, and software testing challenges.

One of the main concerns for cloud computing are security issues. Some organiza-

[6]VMware, Inc. http://www.vmware.com/

tion's data might be considered too sensitive to be stored in a non-private cloud. Cloud customers must put trust in the cloud provider hands to not, intentionally nor by accident, compromise their data. Even if the nodes of the cloud infrastructures are secure, data encryption also needs to be in place for the various communication channels.

There are numerous technical challenges that need to be considered when doing research for cloud integration and it is far from trivial to integrate infrastructures to cloud models in a scalable and secure way. Network problems, such as performance issues, latency, and data transfer rates can for example be hard to manage.

Security and technical challenges have received good focus in academic and commercial literature. Khajeh-Hosseini, Sommerville and Sriram address research challenges for enterprise cloud computing in the journal paper *Research Challenges for Enterprise Cloud Computing* [37], which focuses on security and regulatory issues. Cloud consultant John Roton describes technical and organizational challenges in his book *Cloud Computing Explained* [48].

Software testing has however not been quite as well addressed. The aim of this thesis is to improve this.

2.1.6. Grid Computing

Grid computing can be thought of as cloud computing's technical ancestor. Both are resource packages (hardware, software, network) to be used on demand. The main difference is that the resources in a grid can be global and provided by many users or organizations, while cloud resources are generally only provided by a single provider. There are also differences in applications to be deployed to the two different infrastructures. Grids are typically used for scientific computing while clouds are for business.

Coulouris et al. describe the 'Grid' as a middleware that is designed to enable the sharing of resources such as files, computers, software, data and sensors on a very large scale. The resources are shared typically by groups of users in different organizations who are collaborating on the solution of problems requiring large numbers of computers to solve them [19].

Example application that uses grid technology is the *World-Wide Telescopce* [26], a data-intensive application shared by the astronomy community. Another example, *SETI@home*, was created for an experiment in public-resource computing. The SETI (Search for Extra-Terrestrial Intelligence) application uses personal computers at volunteers' homes to detect intelligent life outside the earth [6].

2.1.7. High Performance Computing (HPC) in the Cloud

With the introduction of cloud technology recent case studies show that developers are seeking ways to utilize cloud platforms for scientific computing or what is known as High Performance Computing (HPC). One example is an image processing application, developed by NASA's Jet Propulsion Laboratory (JPL), which uses Amazon AWS for parallel processing of satellite images[7].

However care must be taken in designing and implementing HPC applications for the cloud. Iosup et al. analyze performance of scientific computing applications within different cloud platforms. Their results indicate that the current clouds need an order of magnitude in performance improvement to be useful to the scientific community. They point out that the current commercial clouds have been built to support web and small database workloads, which are very different from typical scientific computing workloads. Nevertheless they find that while current cloud computing services are insufficient for scientific computing at large, they may still be a good solution for the scientists who need resources instantly and temporarily [35].

The NEON project (Northern Europe Cloud Computing), a cross-Nordic (Sweden, Norway, Denmark, Finland and Iceland) - project, had the purpose of evaluating the usefulness of private versus public cloud services for HPC in the Nordic eScience community. The project's aim was to review the promises and summarize the overall offering cloud computing could give to their community. The project's findings are that private cloud solutions are now not mature enough for a transparent user experience, which is however expected to be gained by the mid of 2012. It is further stated that public clouds, especially Amazon AWS, are in a more mature stage than private ones but still lack features that are necessary to include for cloud resources in a transparent manner in national infrastructures. Public clouds are already competitive in the low end for non-HPC jobs (low memory, low number of cores) on price and public clouds are recommended for non-HPC and some HPC users. About 20% of the jobs running on the current Nordic eScience supercomputer infrastructure are potentially suitable for cloud-like technology [20].

Amazon is aware of the resource demand of scientifical computing, and they have launched special services for HPC[8]. This new service is claimed to be suited for complex computational workloads that are sensitive to network performance or in need of substantial processing power.

[7]http://aws.amazon.com/solutions/case-studies/nasa-jpl/

[8]http://aws.amazon.com/about-aws/whats-new/2011/04/07/announcing-amazon-ec2-spot-integration-with-hpc-instances/

2.1.8. Distributed Systems

Cloud computing and distributed systems are closely related. Cloud technology adds new organizational and technological ways of creating infrastructure for distributed systems.

Resource sharing of computing power, printers, files, web pages, and databases is the motivating factor for creating distributed systems. These systems are everywhere and are largely connected together via the Internet. The systems come in various flavors and are heterogeneous in nature. They can roughly be divided into three types: distributed information systems, distributed computing systems, and distributed pervasive/ubiquitous systems [11].

The construction of distributed systems poses many challenges because of their complexity and diversity. A multitude of different sub-systems and protocols make up a single distributed system, and thus implementing and maintaining a distributed system is very challenging; attributes such as security, scalability, failure handling, concurrency and transparency must be considered [19].

Distributed systems pose numerous challenges on software testing. Because of the diversity it is extremely difficult to test all possible system configurations where the software will be executed i.e. different operating systems, hardware, and software versions. Clustering and load-balancing architectures make matters even more complicated. Reliability issues such as fail-over and redundancy and network related issues such as latency and through-put are further non-trivial testing objectives. These challenges fall under non-functional testing.

This thesis focuses on functional testing of distributed computing systems, mainly on parallel computation which is a form of computation in which many calculations are carried out concurrently [4]. Parallel computing is used in different areas as diverse as mathematics, medicine, molecular biology, astrophysics, and image rendering.

2.2. Amazon Cloud Services

This thesis deals with software testing in an elastic cloud environment. The research involves launching a distributed application within a cloud environment as a case study for software testing. The cloud platform selected for the research was *Amazon Elastic Compute Cloud* (EC2) which is part of *Amazon Web Services* (AWS).

2.2.1. Amazon Web Services (AWS)

Amazon is arguably a pioneer in offering cloud services and AWS has become the de facto standard for cloud infrastructure services. Other IaaS services either complement AWS or are considered competitors to them.

The story behind Amazon's rise in the IT sector was that they wanted to find ways to make better use of underutilized peak computing power of their on-line retail. High proportion of their sales are processed in the weeks before Christmas. The company's peak day in on-line retail in the year 2010 was November 29, when more than 13.7 million items where ordered worldwide across all product categories [31], and it takes a tremendous amount of computing power to handle such a load.

Amazon launched AWS in 2002 to turn their IT cost weakness into an opportunity. The initial plan was selling idle capacity to organizations who needed secure and reliable computing infrastructure, at other times than around the Christmas sales.

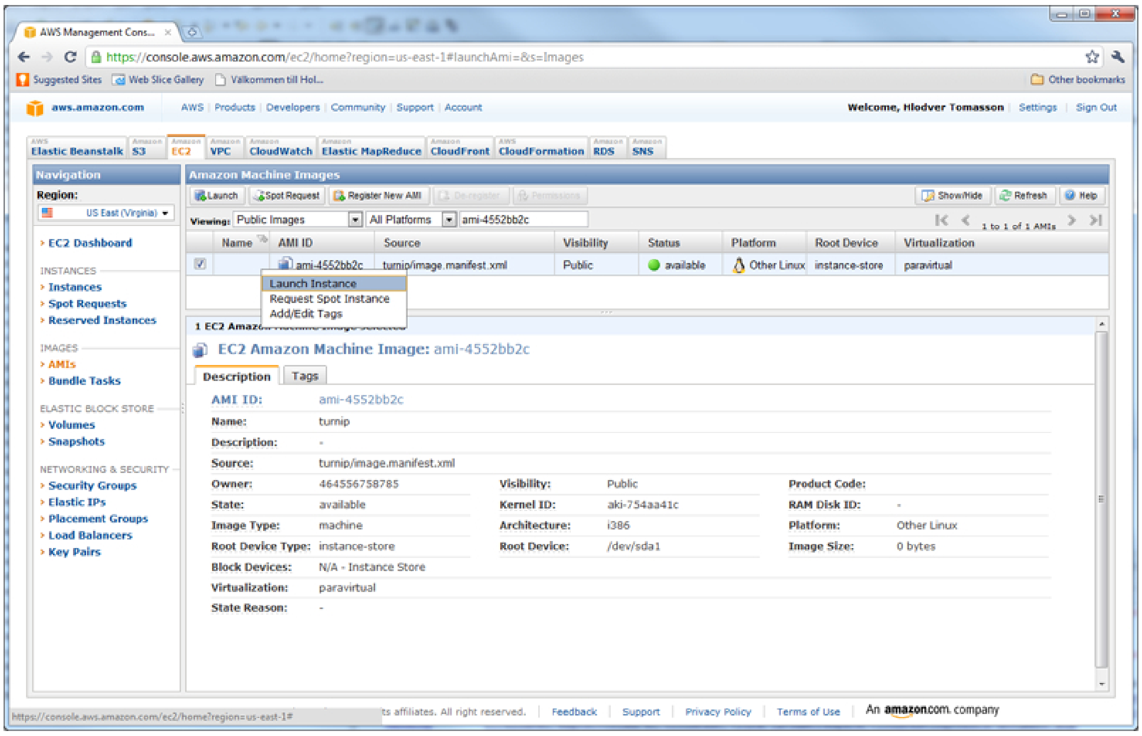
Today Amazon offers a large number of cloud services, with largest attention on Amazon EC2, described in section 2.2.2 and accommodating storage services described in section 2.2.3.

2.2.2. Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (Amazon EC2) is the name of the service that provides the re-sizable compute capacity, or what is referred to as a cloud. Included in Amazon EC2 is a set of web services that allows users to launch and manage Linux/UNIX and Windows server instances in Amazon's data centers, and load them with custom application environment.

It is possible to commission from one up to thousands of server instances simultaneously. Public information on on how many instances EC2 customer can reserve at a time is not available, but the default starting limit is twenty reserved instances that a customer can purchase each month. The limit can be raised by contacting Amazon[9].

Amazon provides web user interface for managing cloud resources. Figure 2.2 is a snapshot of the AWS Management console EC2 view. The console provides management for AWS's compute, storage, and other cloud features. For EC2 management the console has options to start and stop EC2 instances and configure networking and security features.



Name	AMI ID	Source	Visibility	Status	Platform	Root Device	Virtualization
	ami-4552bb2c	turnip/image.manifest.xml	Public	available	Other Linux	instance-store	paravirtual

EC2 Amazon Machine Image: ami-4552bb2c

AMI ID:	ami-4552bb2c				
Name:	turnip				
Description:	-				
Source:	turnip/image.manifest.xml				
Owner:	464556758785	Visibility:	Public	Product Code:	
State:	available	Kernel ID:	aki-754aa41c	RAM Disk ID:	-
Image Type:	machine	Architecture:	i386	Platform:	Other Linux
Root Device Type:	instance-store	Root Device:	/dev/sda1	Image Size:	0 bytes
Block Devices:	N/A - Instance Store				
Virtualization:	paravirtual				
State Reason:	-				

https://console.aws.amazon.com/ec2/home?region=us-east-1#

Figure 2.2: The AWS management console

Amazon furthermore provides a software development kit (SDK) for cloud service management called *Amazon AWS SDK*. Currently the SDK's API has been implemented for Java, Microsoft .NET, PHP, Python, and Ruby. Eclipse users can download the *AWS Toolkit* plug-in which makes Java application deployment to instances possible from within the Eclipse IDE, plus additional instance management features [5]. The AWS SDK for Java is used for this thesis' case study, to manage cloud instances for the test bed.

[9]http://aws.amazon.com/contact-us/reserved-instances-limit-request/

Amazon Machine Image (AMI)

A core feature of cloud computing is virtualization. Virtual machines in Amazon EC2 are called *Amazon Machine Images* (AMI). An AMI consists of a read-only file-system image which includes an operating system and typically some application software (e.g. web application or a database).

Amazon provides prepackaged AMIs with Windows and Linux operating systems installed. Various 3rd party vendor also provide prepackaged images, like Ubuntu which provides official Ubuntu Linux images on Amazon EC2. Users can also create their own images from scratch or extend publicly available images and re-package them with new instance IDs.

An AMI can be *public*, *paid*, or *shared*. A public AMI can be used by anyone: A paid image requires subscription fees, and a shared AMI is private and can only be used by Amazon EC2 users specified by the owner of the image.

Amazon EC2 Hardware Specifications and Pricing

The hardware specification for Amazon instances is selected at startup of each instance and can be selected from predefined a set of instance types defined by Amazon (e.g. *Small*, *Large*, or *Extra Large*). The instance type controls what resources the AMI uses; RAM, CPU and disk size. Table 2.1 lists the variations for EC2 instances and the different prices in USD per hour based on computing power.

The prices shown in the table are for Linux/UNIX usage within the European region. Windows operating system costs are roughly around 10-25% higher for each instance type. The case study in this thesis uses the standard 'Small' instance type. The cost for this thesis' research has summed up to be 112.59 USD for 1,186 instance hours.

Amazon provides further pricing options for reserved instances on annual basis or for spot-priced instances. Pricing details can be find on Amazon's EC2 website[10].

2.2.3. Amazon Data Storage

Because of EC2's elastic nature permanent data persistence is not possible for AMIs. Amazon provides various different cloud services to address data persistence issues in the cloud:

[10]Amazon EC2 Pricing: http://aws.amazon.com/ec2/pricing/

Type	**RAM (GB)**	**CPUs (EC2)**	**Instance Storage**	**Platform**	**I/O Perform.**	**Price/ hour**
Micro Instances:						
Micro	0.613	2	0 GB	32/64-bit	Low	$0.025
Standard Instances:						
Small	1.7	1	160 GB	32-bit	Moderate	$0.095
Large	7.5	4	850 GB	64-bit	High	$0.38
Extra Large	15	8	1,690 GB	64-bit	High	$0.76
High-Memory Instances:						
Extra Large	17,1	6.5	420 GB	64-bit	Moderate	$0.57
Double Extra Large	34,2	13	850 GB	64-bit	High	$1.14
Quadruple Extra Large	68,4	26	1,690 GB	64-bit	High	$2.28
High-CPU Instances:						
Medium	1.7	5	350 GB	32-bit	Moderate	$0.19
Extra Large	7	20	1,690 GB	64-bit	High	$0.76

Table 2.1: Amazon on-demand instance types (May 7, 2011)

- Amazon Simple Storage Service (S3) provides a web services interface for storing and retrieve data.
- Amazon Elastic Block Storage (EBS) provides block level storage volumes for use with Amazon EC2 instances.
- Amazon SimpleDB is a non-relational data store for data queries via web services.

The difference between S3 and SimpleDB is that SimpleDB is meant to be for smaller amounts of data. SimpleDB was created for performance optimization and in order to minimize costs across AWS services for large objects and files which are stored in S3. Meta-data associated with those files is stored in the SimpleDB. It is possible to run applications in Amazon EC2 and store data objects in Amazon S3.

Amazon SimpleDB can then be used to query the object meta-data from within the application in Amazon EC2 and return pointers to the objects stored in Amazon S3. This allows for quicker search and access to objects, while minimizing overall storage costs.[11]

2.2.4. Eucalyptus

Eucalyptus (*Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems*) is an open-source private cloud infrastructure, which implements the Amazon specification for EC2. Eucalyptus conforms to both the syntax and the semantic definition of the Amazon API and tool suite, with few exceptions. Thus it is possible to use the Amazon command-line tools directly for Eucalyptus[12].

2.3. Software Testing

This section starts by introducing the fundamentals of software testing, before introducing software testing techniques for distributed systems and cloud computing. Common challenges for this area are discussed and what testing methods should be considered for testing within cloud environments.

2.3.1. Fundamentals of Software Testing

Software testing is a critical part of the software development process. It is one of the most challenging and costly process activities, and in its fullest definition it provides strong support for the development of high quality software [15]. Testing objectives can be seen as twofold:

- Evaluating quality and
- Defect detection

As other quality assurance activities, testing helps ensuring that software is built on time, within budget, and is of the quality expected, defined by quality attributes such as reliability, usability, performance, and the ability to meet users' requirements.

[11] Amazon S3 vs. SimpleDB. http://aws.amazon.com/simpledb/#sdb-vs-s3
[12] Eucalyptus FAQ. http://open.eucalyptus.com/wiki/FAQ

2.3.2. Verification and Validation

The testing process includes the *Verification and Validation* (V&V) processes which are a set of activities whose goal is to foster software quality during the development life-cycle [33]. The processes are defined in *IEEE Standard Glossary of Software Engineering Terminology* [47] as the following, with Boehm's [12] informal definitions of the terms inside brackets.

Verification is the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase.

(Am I building the product right?)

Validation is the process of evaluating software at the end of the software development process to ensure compliance with software requirements.

(Am I building the right product?)

Verification and validation are closely related to *static*- and *dynamic* testing where verification is usually associated with activities such as inspections, reviews and walkthroughs (static testing). Validation on the other hand is usually associated with running executable test cases on the code (dynamic testing), which this thesis focuses on.

2.3.3. Types of Testing

Software testing is a complex activity. Figure 2.3 shows the many dimensions of software testing which can cover the different stages of a software development project. Different test case design strategies can be used at different testing levels, and there are many quality attributes of the software to consider when testing.

With black-box testing strategy the software under test is considered to be an opaque box, with no knowledge of its inner structure. Testers have only knowledge of what the software is supposed to do from the software specifications. Black-box approaches can be can be used for all levels of testing from unit to acceptance, and are especially useful for revealing requirements and specification defects. Because black-box testing only consider software behavior and functionality, it is often called functional, or specification-based testing.

White-box testing approaches, on the other hand, focus on the inner structure of the software under test. White-box approaches are mainly used for unit testing and up to the integration testing level. They are especially useful for revealing design and code logic defects in elements such as loops, branching and data flow [14].

It is important that there are several degrees of freedom of what, where, and when should be tested in a software system, and the key to successful testing is the selection, and the execution, of appropriate methods in each case. This thesis focuses on black-box functional integration testing, for validation of a software application running in a cloud environment.

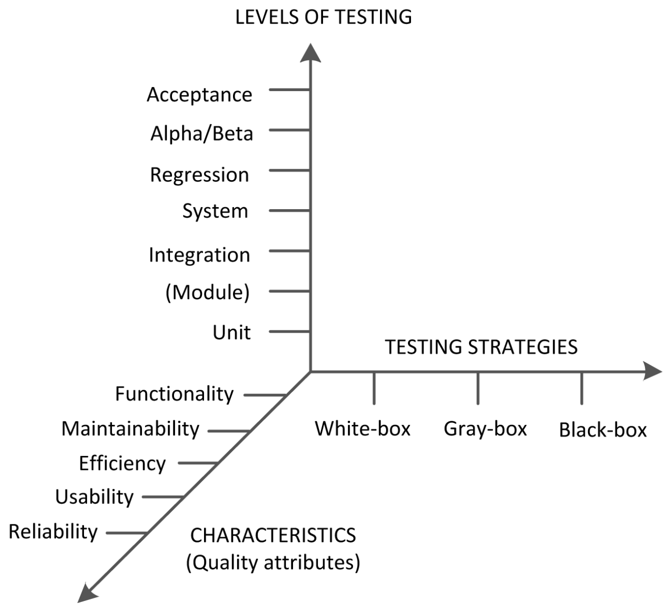


Figure 2.3: Types of testing [53]

2.3.4. Testing Distributed Systems

As described in Section 2.1.8, cloud applications can take on the form of various types of distributed systems. Testing of distributed systems focuses on the interaction between components of the distributed systems, as opposed to non-distributed testing which do not usually involve any interaction with other systems or distributed components. In distributed testing, the test system itself is often distributed as well, i.e. test components can be distributed on the different nodes of the system.

Distributed testing should not be confused with remote testing which is a method to distribute test cases over many machines. No communication is between the different machines in case of remote testing, even though many machines are used at once. Those tests do not involve any interaction between the different processors or the test components of the test cases.

2.3.5. Testing Cloud Applications

When discussing software testing and cloud computing it is beneficial to distinguish between the different testing objectives. The Software Testing in the Cloud (STITC)[13], a special interest group, categorized the different objectives of cloud testing in the following three key areas at the *2nd International Workshop on Software Testing in the Cloud*, held in Paris, France in April 2010. The STITC workshop was co-located with the 3rd IEEE International Conference on Software Testing, Verification, and Validation (ICST 2010).[14]

1. Testing *in* the cloud: Leveraging the resources provided by a cloud computing infrastructure to facilitate the concurrent execution of test cases in a virtualized environment.

2. Testing *of* the cloud: Testing applications that are hosted and deployed in a cloud environment.

3. Migrating testing *to* the cloud: Moving the testing process, test assets, and test infrastructure from their current state to facilitate either testing *in* the cloud or testing *of* the cloud.

Testing *in* the cloud is about utilizing the cloud for testing, such as for configuration testing and load testing. Software testing *of* the cloud, the focus of this thesis, is the least researched area. Very few papers have been published that focus directly on testing applications in a cloud environment. Migrating testing *to* the cloud is a hybrid category for the two other categories. Related work for these three categories is discussed in Section 3.

Testing *of* the Cloud

This thesis focuses on testing *of* the cloud. Much of the testing of cloud applications can be done with traditional distributed testing methods regardless of the type of hardware or network infrastructure in use.

On the system testing level the whole system is tested via its user interface and the transparency of the clouds should not impose any special requirements of conforming to functional requirements of the software, so testing is not affected by the fact the application is running within cloud environment. Same system testing approaches can therefore be used as for non-cloud systems. The same goes for the unit testing

[13]Software Testing in the Cloud (STITC). http://www.stitc.org/
[14]ICST 2010. http://vps.it-sudparis.eu/icst2010/

level where classes are tested in isolation and should not interact with the cloud environment. Unit testing can even be run outside the cloud in organization's local test environment.

Within traditional, fully-controlled, and predictable test-beds IP address allocation is usually not an issue, because the testers know the addresses of each machine used in the testing. However, the elastic nature of clouds as described in 2.1.3, introduces new challenges for software testing. This paper addresses issues regarding elastic behavior of clouds, particularly IP address allocation of virtual machine instances.

The integration testing level, which lies between the unit testing and system testing levels, is the focus of this thesis. The focus is on the communication on two (or more) different software components which run on two (or more) cloud instances. To be able to test the communication, test components running on each of the involved cloud instances are needed to perform actions on the components under test. The problems for this type of testing have been solved with test technologies such as TTCN-3 and researched for grid application workflows [50].

This thesis case study uses the *Jata* test framework [60] which uses concepts of TTCN-3 for setting up the communication points for the functional black-box integration testing of a cloud application. TTCN-3 and Jata concepts are described in sections 2.4 and 2.5 respectively.

For testing within cloud environment there are also various considerations to be made regarding non-functional requirements. Security testing and privacy-aware testing probably being at the top of many organizations' list. Substantial amount of work has been done to identify cloud computing security risks and privacy issues [27, 56]. Another consideration is performance testing because virtualization and resource time-sharing may introduce unexpected performance penalties when applications are deployed in the cloud.

Testing of non-functional requirements for cloud applications, such as the challenges for non-functional distributed system testing discussed in Section 2.1.8, are not within the scope of this thesis.

2.3.6. Cloud Testing Tools

One area which targets software testing using the cloud, and is widely used, is the easy access to large clusters of test machines for various purposes, e.g. load-, stress- and performance testing. Several commercial testing tools exist that utilize cloud computing for testing. They can for instance allow automation of functional and configuration testing such as cross browser testing of websites with different browsers and different operating systems. An example tool for this purpose is *Cloud Testing*[15].

PushToTest[16] can be used within a *Continuous Integration* [21] environment, where the build server can start the cloud testing tool automatically to deploy the system under test and the test code to available test nodes in the cloud. The cloud testing framework runs the test cases, collects results and tears down the cloud instances afterwards.

SOASTA CloudTest[17] supports performance and stress testing by leveraging the compute power offered by cloud computing.

Test Lab as a Hybrid Cloud Example

One example of cloud service usage is cost reduction model for hardware and software costs for software company's development- and test labs. The idea is to use a hybrid cloud where the public cloud part is used to satisfy any spikes in load for the operation of the system's infrastructure. Figure 2.4 shows how the contributed cost can be set to lower levels than the maximum need for the spike loads. In this example the cloud infrastructure is a hybrid cloud where the companies test lab is a private cloud but is extended to the public cloud when more resources are needed for cases such as integration-, load-, stress-, and performance testing.

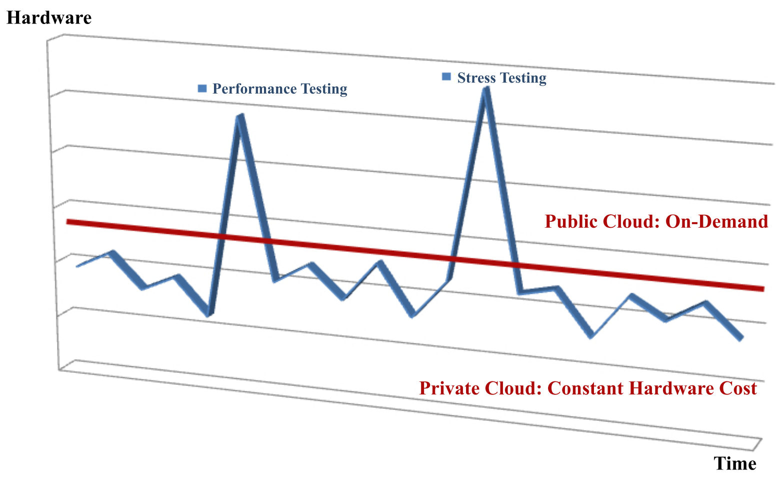


Figure 2.4: Hybrid cloud example for a software test lab

[15]Cloud Testing. http://www.cloudtesting.com
[16]PushToTest. http://www.pushtotest.com
[17]SOASTA CloudTest. http://www.soasta.com/cloudtest/

The same argument holds for this cloud infrastructure as a cost reduction model for on-line retail system, where spikes in load are at maximum around Christmas sales. Amazon, the large on-line retailer, went the other way around and decided to make use of their peak hardware investments to create services and profits of their regular basis underutilized computing resources. Amazon created AWS, a large set of web services, which includes what is now known as one of the largest cloud services, Amazon EC2, as described in Section 2.2.

2.3.7. Test Documentation

For structuring purposes the case study performed in this thesis uses a *Software Test Plan* approach with added test design and test case details for describing a black-box application testing of a cloud application. The test procedure is furthermore covered.

A software test plan is a document describing the testing scope, approach, resources, and activities of testing. The test plan holds the basis for formal testing. It identifies amongst others test items, the features to be tested, the test environment, and the test design techniques.

The format and content of a software testing documentation can vary depending on the processes, standards, and testing techniques used. The structure used in Sections 4.2 to 4.5 is based on concepts from *IEEE Standard for Software and System Test Documentation* (IEEE Std 829-2008) [32], in particular concepts from the *Level Test* chapters, listed in table 2.2.

IEEE Std 829 - Level Test Chapters	
LTP	*Level Test Plan(s)*
LTD	*Level Test Design*
LTC	*Level Test Case*
LTPr	*Level Test Procedure*

Table 2.2: Level test chapters used from IEEE Std 829-2008

The sections used from the IEEE level test chapters are listed with comments in table 2.3.

IEEE-829 Section	Concept	Description
LTP 1	*Introduction*	Provides an overview of the test plan. Specifies goals, objectives, and any constraints.
LTP 2.1	*Test items*	List of software/products and their versions.
LTP 2.3	*Features to be tested*	Features of the software/product to be tested.
LTP 2.4	*Features not to be tested*	Features of the software/product which will not be tested, and the reasons.
LTP 2.5	*Approach*	The overall approach to testing. Specifies the testing levels, the testing types, and the testing methods.
LTP 2.6	*Item pass/fail criteria*	Specifies the criteria that is used to determine whether each test item has passed or failed testing.
LTP 3.2 & LTC 2.5	*Test environment*	Properties of the test environment: hardware, software, communications etc.

Table 2.3: IEEE Std 829-2008 concepts used for the case study documentation

2.4. The Test Language TTCN-3

The *Testing and Test Control Notation version 3* (TTCN-3) is a test specification and test implementation language that supports black-box functional testing of distributed systems. It also has support for automatic execution of the specified test cases by adding an adaption layer. TTCN-3 can be used for different levels of testing such as system-, integration-, and unit testing. It is not only applicable for specifying and implementing functional tests, but also for scalability, robustness, or stress tests of huge systems [25, 43].

TTCN-3 originates from the telecommunication domain and it is published as an international standard by the ETSI (European Telecommunication Standard Institute) [1]. As a standardized language TTCN-3 is supported by various tools and used by industry and standardization bodies. Powerful tools are available for specifying, deploying, and executing test cases. TTCN-3 is applied in various other domains than telecommunication (e.g. data communication, automotive, railway, and finance). However only one work [50] is known where the language is used in the field of parallel systems [43].

2.4.1. Conformance Testing Methodology and Framework

TTNC-3 is based on concepts of the of the international ISO/IEC standard 9646 OSI Conformance Testing Methodology and Framework (CTMF) [36]. CTMF consists of seven parts for defining a comprehensive procedure for the conformance testing of Open Systems Interconnection (OSI) protocol implementations. The different parts contents are: 1) General concepts, 2) test suite specification and test system architectures, 3) test notation, 4) test realization, and 5-7) means of testing and organizational aspects.

CTMF has been successfully applied for testing distributed telecommunication systems such as ISDN- and GSM-based systems. CTMF concepts were adapted by Rings, Neukirchen and Grabowski in *Testing Grid Application Workflows Using TTCN-3* [50], for functional testing of distributed Grid applications. Part 2 of the CTMF standard, *test suite specification and test system architectures*, was followed, while other parts were too OSI or conformance testing specific to be used for grid testing.

This thesis uses CTMF's *multi-party test architecture*, in accordance to [50], to test the parallel application management functionality of a cloud application. The conceptual test architecture is shown in figure 2.5.

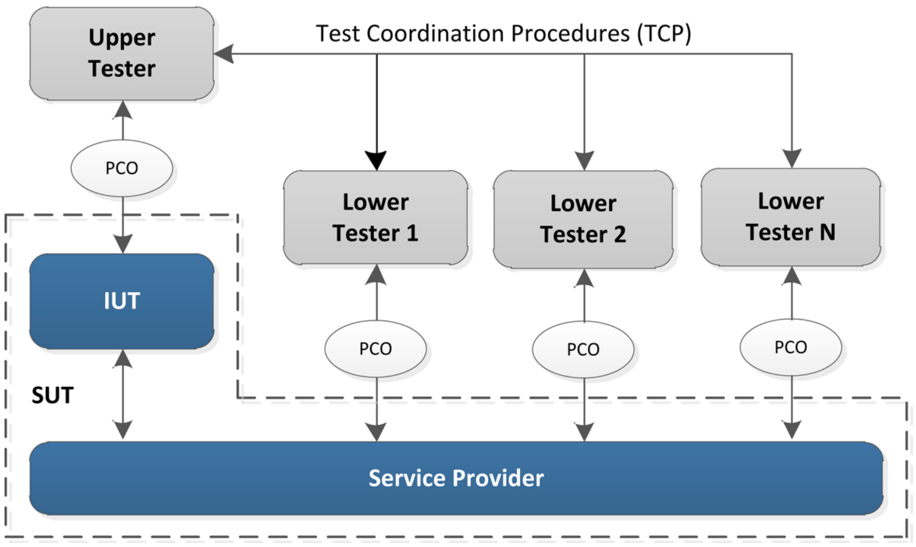


Figure 2.5: CTMF multi-party test architecture [50]

The system under test (SUT) is made up of the service provider (e.g. cloud or grid) and the implementation under test (IUT). For cloud and grid testing it is assumed that the service provider has already been adequately tested, and the focus is on the communication aspects of the application's parallel functionality.

The SUT is controlled by an Upper Tester function (UT) and one or more Lower Tester functions (LT). The UT plays the role of a user of the service provided by the SUT. IUT and the LTs play the role of peer entities, i.e. the LT and the IUT realize together the service provided to the UT.

The Points of Control and Observation (PCO) are the interfaces used by UTs and LTs to communicate with the SUT. UTs and LTs communicate by means of Test Coordination Procedures (TCP). Further test architectures for distributed systems using CTMF concepts are described in details by Walter, Schieferdecker and Grabowski [59].

2.4.2. TTCN-3 Concepts

This section contains description of TTCN-3 test architecture and language elements. The concepts covered are only subset of the TTCN-3 test language, i.e. those concepts that are relevant for this thesis.

The TTCN-3 language is partly comparable to general programming languages, such as C++, C#, and Java which include modules, data types, variables, functions, parameters, loops, and conditional statements. However TTCN-3 is based on concepts which are independent from general programming languages, such as test cases, test systems, test verdicts, and test components.

The building-blocks of TTCN-3 are *modules*, which can be parsed and compiled as separate entities. A module can include one or more test cases, and describes the execution sequence of them.

A *test case* is a complete and independent specification of the actions required to achieve a specific test purpose. Test cases define the executable behaviors for stimulating the SUT and analyzing the results. In a typical black-box test case, a stimulus is sent to a SUT and the response observed to decide whether the SUT has passed or failed the test.

Test Architecture

TTCN-3 architecture supports distributed testing via *Test Components* (TC). Not only may the SUT be distributed or parallel, but also the test itself may consist of several test components that execute test behavior in parallel. A TTCN-3 test system consists of one *Main Test Component* (MTC) and zero or more *Parallel Test Components* (PTCs). The MTC's behavior is specified in the body of the test case definition. Test components can dynamically create, start, and stop other PTCs.

Each test component runs concurrently and can execute test behavior in parallel with other test components. Test case execution ends when the MTC terminates.

Test cases are executed by a *Test System*. A test system configuration consists of a set of inter-connected test components with well-defined communication *ports* and an explicit test system interface which defines the borders of the test system [1]. Figure 2.6 shows an overview of a TTCN-3 test system with test components communicating together via ports and to the SUT through layers of abstract and real test system interfaces.

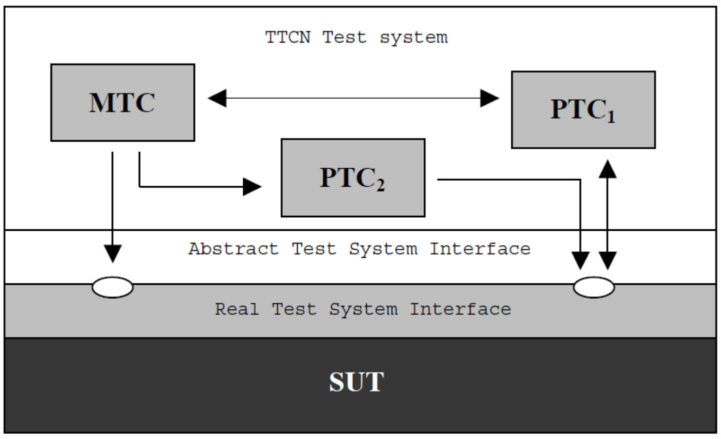


Figure 2.6: Conceptual view of a typical TTCN-3 test configuration [1]

The definition of an *abstract test system interface* (as seen in Figure 2.6) is identical to a test component definition, i.e. it contains communication ports through which the test case is connected to the SUT.

The abstract test system interface is described using high-level TTCN-3 code with abstract port definitions, whereas the *real test system interface* is realized by a low-level adaptation layer that implements the ports. The real test system interface, i.e. the physical connections, can be implemented with a programming language such as Java. The specification and implementation of the real test system interface is outside the scope of TTCN-3 [1].

When designing test systems with TTCN-3 it is desirable to make the real test system interface ports as 'thin' as possible, by implement all test case logic within the abstract TTCN-3 layer. That way it is possible to re-use the port implementation for other test cases.

Communication Ports

For communication between test components or between TCs and the test system interface (SUT), *ports* are used. Each test component has a set of ports which control in- and out-directions. If two ports are connected, the in-direction of one port is linked to the out-direction of the other, and vice versa. The out-direction is directly linked to the communication partner, i.e., outgoing information is not buffered.

Each port is modeled as an infinite FIFO queue which stores the incoming messages or procedure calls until they are processed by the component owning that port. Port connections are created and destroyed dynamically at runtime and there are no restrictions on the number of ports a component may have.

Communication between ports can be message-based and procedure-based communication. Message-based communication (e.g. low-level network messages) is based on an asynchronous message exchange and the principle of procedure-based communication is to call procedures in remote entities (e.g. high-level procedure or function calls). Figure 2.7 shows an example of test case port communication setup for three test components and a SUT.

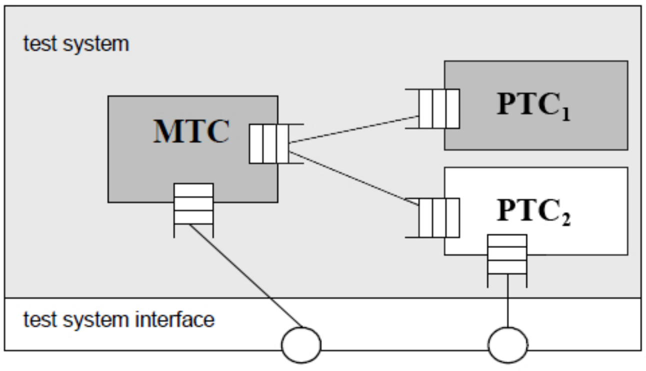


Figure 2.7: Example TTCN-3 port communication setup

Alt Statements

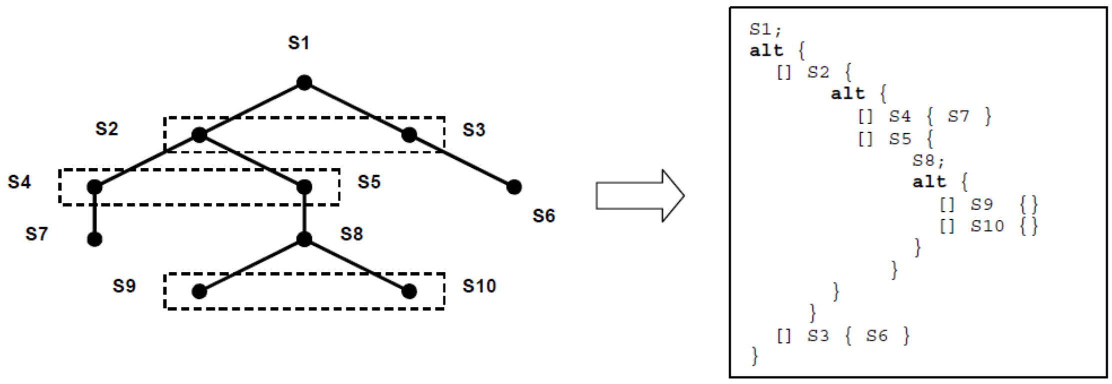
The *alt* statement is used where several alternatives can occur at a certain point in time, for instance if two different messages can be received as a response. Alt statements express the sets of possible alternatives that form the tree of possible execution paths by defining the branching of test behavior for receiving and handling of events and responses, i.e. messages, timer events, and termination of PTCs [1, 25].

Code example in listing 2.1 shows a typical structure of a TTCN-3 test case using an *alt* statement. The test stimulates the SUT by sending a message (line 11) and a timer is started (line 13) to control the timeout for a response. For analyzing the outcome multiple alternatives are defined (line 15-35). Expected valid response makes the test pass. Invalid response (anticipated or unexpected) makes the test fail and the timer functionality makes sure that the test case fails if the response is not not received in time.

```
 1 systemType { timer replyTimer;
 2
 3   // The expected response
 4   var charstring v_expectedResponse := "42";
 5
 6   // Map ports between MTC and SUT (system component)
 7   map(mtc:pt_mtc, system:port_system);
 8
 9   // Send the stimulus via MTC's port
10   pt_mtc.send(a_MeaningOfLifeRequestMessage)
11
12   replyTimer.start(100.0);
13
14   alt {
15     // Handle the case for an expected answer.
16     [] pt_mtc.(receive(a_ExpectedResponse(v_expectedResponse)){
17          replyTimer.stop;
18          setverdict(pass);
19     }
20     // Handle the case for an invalid (but anticipated) response.
21     [] pt_mtc.receive(a_InvalidButAnticipatedResponse){
22          replyTimer.stop;
23          setverdict(fail);
24     }
25     // Handle the case for any other (invalid) response.
26     [] pt_mtc.receive(){
27          replyTimer.stop;
28          setverdict(fail);
29     }
30     // Handle timeout.
31     [] replyTimer.timeout {
32          setverdict(fail);
33     }
34   }
35   unmap(mtc:pt_mtc, system:pt_system);
36   stop;
37 }
```

Listing 2.1: TTCN-3 typical test case structure

It is possible to define powerful and complex forms of behavior with TTCN-3 where sequences of statements are expressed as sets of possible alternatives to form a tree of execution paths, as illustrated in figure 2.8.



```
S1;
alt {
  [] S2 {
       alt {
         [] S4 { S7 }
         [] S5 {
              S8;
              alt {
                [] S9  {}
                [] S10 {}
              }
            }
       }
     }
  [] S3 { S6 }
}
```

Figure 2.8: Illustration of alternative behavior in TTCN-3 [1]

Functions and Altsteps

Behavior defined in the body of test cases can be structured into *functions* and *altsteps*. TTCN-3 functions are similar to functions in typical programming languages, i.e., they can be used to structure computation or to calculate a single value. Behavior of test components is generally defined by a function.

Altsteps are a combination of a function and an *alt* statement and are used to specify default behavior or to structure the alternatives of an alt statement. Altsteps are scope units similar to functions and may invoke functions and altsteps.

Test Verdict

The result of a test case execution is a *test verdict*. TTCN-3 provides a special test verdict mechanism for the interpretation of test runs. This mechanism is implemented by a set of predefined verdicts, local- and global test verdicts and operations for reading and setting local test verdicts.

The verdict can have five different values: *pass*, *fail*, *inconc*, *none*, and *error* which are used for the judgement of complete and partial test runs. A pass verdict denotes that the SUT behaves according to the test purpose, a fail indicates that the SUT violates its specification, an inconc (inconclusive) describes a situation where neither a pass nor a fail can be assigned and the verdict error indicates an error (i.e. communication error) in any of the test devices which can make the test run impossible to continue in a normal way. The verdict none is the initial value for local and global test verdicts, i.e., no other verdict has been assigned yet [25].

Each test component maintains a local verdict for tracking its own individual verdict. Additionally, the TTCN-3 run-time environment maintains a global test case verdict that is updated when each test component (i.e. the MTC and each and every PTC) terminates execution. The global test verdict, which is not accessible to test components, is updated according to overwriting rules when a test component terminates. The final global test verdict is returned to the module control part when the test case terminates.

Figure 2.9 illustrated the relationship between test case, test components and verdicts [1]. Test components can communicate with each other to co-ordinate their actions or to come to a common test verdict.

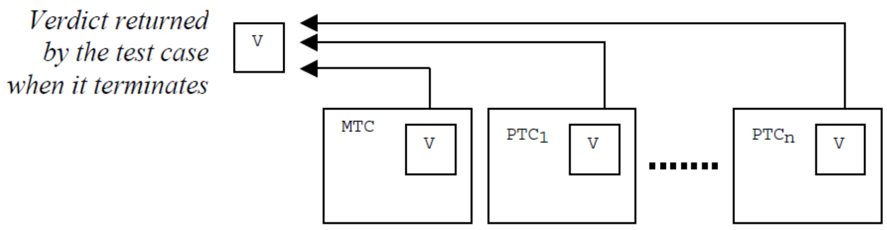


Figure 2.9: Illustration of the relationship between verdicts in TTCN-3 [1]

The concepts of TTCN-3 allow creation of powerful distributed test architectures, for example instantiations of the CTMF test methods [50]. This overview touches only on the most common concepts and the ones which relate to distributed testing such as testing for cloud applications, and the concepts Jata borrows from TTCN-3.

2.5. The Jata Test Framework

This thesis uses *Jata* [60], a Java test framework, in a case-study for testing a cloud application, described in Section 4.2.

Jata was introduced in *Jata: A Language for Distributed Component Testing*, by Wu, Yang and Luo [60]. It is a Java library which includes a meta-model for defining and structuring test cases. The meta-model uses concepts from the well known standardized testing language *TTCN-3*, described in Section 2.4, to support distributed component testing.

Jata's Java source code, binaries, and examples can be downloaded from: `http://code.google.com/p/jata4test/`

Jata was created to address issues in test automation for distributed testing. Its approach is combining the strengths of two well known test frameworks: the de-facto standard unit testing framework JUnit [10] and the TTCN-3 testing language. Both are incredibly powerful, each in their own way, in specifying both simple and complex test cases.

2.5.1. Jata vs. JUnit

JUnit is arguably the most popular test framework for Java and has become de-facto standard for automated unit testing. It does however not focus on support for distributed testing. JUnit runs in its own thread space and has no means of supporting distributed communication back to its testing system context. In distributed testing, distributed test components need a way to communicate with the test system, as supported by TTCN-3.

Jata is not based on JUnit nor does it use any concepts from JUnit. What is common between them are that both are frameworks for creating and running repeatable tests written in Java. JUnit focuses on locally executed unit testing, preferably in isolation, while Jata focuses on distributed unit testing. In locally executed unit testing there is no need for communication between different units and *test doubles* [42], such as stubs or *mock objects* [22], are used to eliminate dependencies between components.

The fundamental differences between the two is that JUnit does not, out of the box, support testing of distributed and parallel objects. JUnit test cases also terminate the test execution as soon as an exception is caught while distributed tests need to finish all threads before analyzing the results, even though an error has been discovered. Being a Java library it is possible to use JUnit from Jata test cases as any other Java libraries. JUnit assertions provide optionally powerful and useful features to use within Jata.

2.5.2. Jata's Meta-Model for Distributed Testing

For distributed testing a test technology that provides a systematic and structured way of defining test cases is needed. With multi-threading test evaluation architecture Jata promises to support complex test scenarios in a flexible manner [60]. Jata's approach to distributed testing is providing rich constructs, using TTCN-3 concepts, to design test cases in a systematic and structured way.

Jata's meta-model is shown in figure 2.10. Testing concepts such as *test components*, *alternatives*, *communication ports*, and *codecs* are borrowed from TTCN-3. Jata can be seen as a minimal implementation of TTCN-3 testing concepts for Java.

Because of the similarity to TTCN-3 this section does not go into detail on Jata's architecture, the corresponding concepts are described in Section 2.4, which includes a high-level overview of TTCN-3 architecture. In the following section, 2.5.3, the commonalities and differences between the two languages are described.

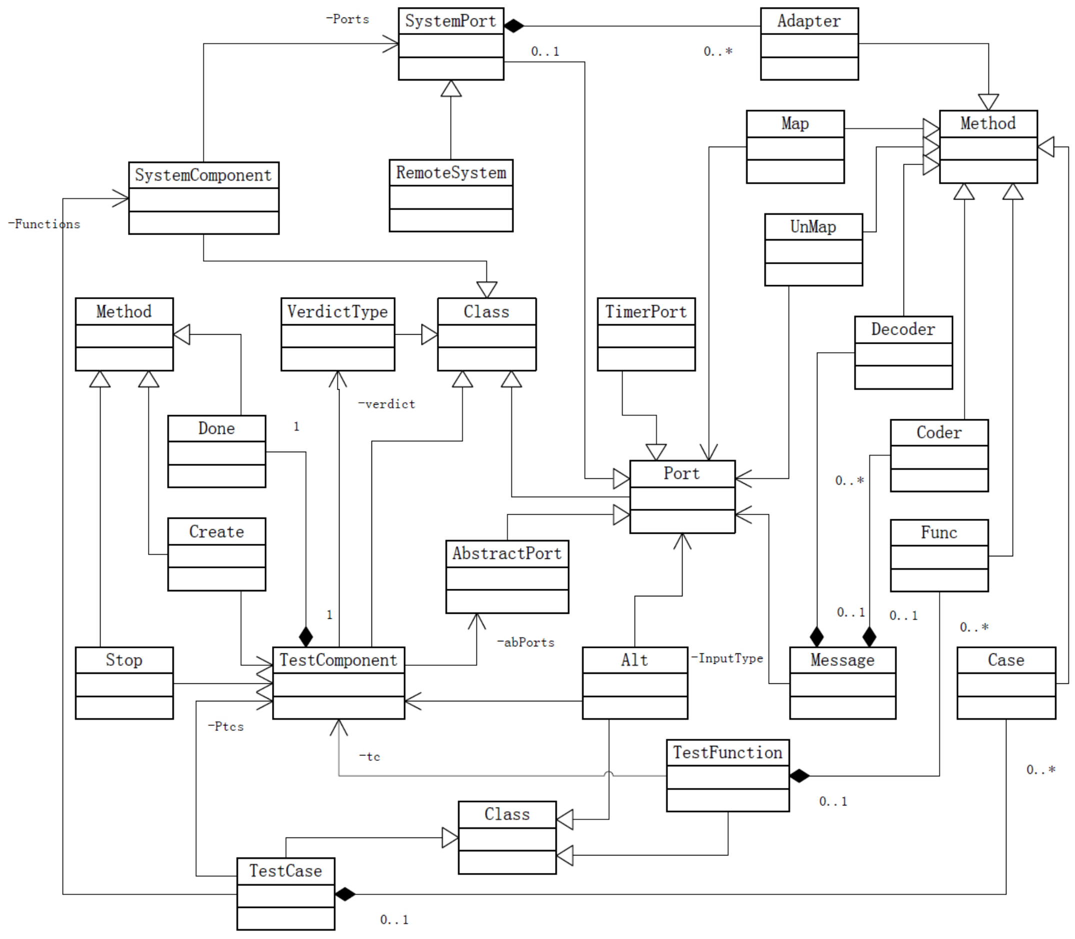


Figure 2.10: Jata's meta-model [60]

Usage of Jata is described in details in the case study in Section 4.2, where challenges of testing cloud applications are investigated. Jata's usability and applicability for testing applications in cloud environments is covered in the evaluation chapter of the thesis (Chapter 5).

Why Jata?

For testing distributed system within a cloud environment, TTCN-3 would be a natural option to look at. There are couple of issues however regarding usage of TTCN-3 for research as this one. Access to appropriate tools being one and another being TTCN-3's relatively high learning curve.

Availability of tools for TTCN-3, because of licensing matters, is an issue for cloud applications. Most TTCN-3 tools have a licensing scheme based on MAC network addresses, which is unfortunate for cloud research. The elastic nature of clouds makes it impossible to know beforehand the MAC address of the virtual machine assigned each time, making cloud research with this type of licensing model difficult. Licensing issues are major obsticles for the cloud and need to be solved on case-by-case basis between the software vendor and the cloud provider.

The Jata language was chosen for its simplicity and promising techniques for distributed systems testing based on combination of sound techniques from TTCN-3 and the flexibility of it being a Java library. Licensing issues played some role in the selection, but it should be noted that a TTCN-3 tool sales representative was not contacted for possibilities of research grant options. Early indications of Jata applicability returned promising results for cloud testing, and it was therefore selected for the research.

2.5.3. Jata and TTCN-3 Comparison

TTCN-3 is a widely recognized standard from ETSI for a testing specification language. TTCN-3 tools have been developed and widely used in the industry for various testing purposes [25] while Jata's only usage is the authors introduction of the language [60].

Jata does not implement the TTCN-3 standard, rather it selects the concepts needed to implement a light-weight test framework for testing distributed systems. TTCN-3 can be considered a heavyweight approach which requires commercial tool support and relatively high learning curve. Jata only uses subset of the TTCN-3 features, and could be described as a trimmed down version of TTCN-3's core concepts. Table 2.4 lists the TTCN-3 concepts that map to Jata's meta-model, shown in figure 2.10.

As seen in the list, Jata adopts the core concepts of TTCN-3 for specifying test cases. Following sections describe how Jata uses the these TTCN-3 concepts for test case definitions.

TTCN-3 Concept
Test Case
Test Function
Main Test Component (MTC)
Parallel Test Component (PTC)
Port
Map/Unmap ports
Alt statements
Adapters
Codecs (Coders/Decoders)
Verdicts

Table 2.4: Adopted TTCN-3 concepts in Jata

Test Case and Test Function

To define a test case in Jata the *TestCase* class is inherited. The actual test case definition goes into the overriden *Case* method. The *Case* method has two input parameters: the MTC and the SystemComponent. There are four things a test case generally needs, independent of the type of the test [42]:

1. Setup (configure test components and prepare the test data)
2. Exercise (stimulate the SUT)
3. Verify (evaluate the response)
4. Tear-down (close connections and take down systems if needed)

The basics of how these things are structured and implemented in a Jata test case is borrowed from TTCN-3.

The TestFunction meta-class is used to define test behaviors, as functions are used in TTCN-3. TestFunction has an overridable method Func for the actual definition. The Func operation takes a test component (PTC) as a parameter which forces test engineers to create the PTC outside the function's scope.

Test Components

Jata uses the concepts of test components as is done in TTCN-3. There are are two types of test components in Jata: TestComponent and SystemComponent. The former represents MTC and PTCs which are used for sending stimulation messages and receiving response messages from the SUT. As in TTCN-3 the MTC is created automatically by the test system at the beginning of each test case execution and the PTCs are dynamically created and stopped during the execution.

SystemComponent represents the abstract and real test system interfaces in TTCN-3 and is used by test components for communication with the SUT for sending and receiving messages. There is exactly one SystemComponent for each Jata test system, but it can contain multiple system ports.

There is one major difference in the distribution, i.e. deployment, of parallel test components between Jata and TTCN-3. In Jata it is not possible to distribute the PTCs. MTC and PTCs must run inside the same Java virtual machine. Instead *remote ports* are distributed on parallel nodes of the SUT (see following section on communication ports).

Communication Ports

As in TTCN-3 test components contain communication ports. There are three types of ports defined in Jata: *AbstractPort* (AP), *SystemPort* (SP) and *TimerPort* (TP). AbstractPort owns an infinite FIFO queue in which the incoming messages are stored. SystemPort is composed of an AbstractPort and an *Adapter*. Timer-ports are used to control timeouts and are special in the way that they do no connect to other ports. An AP can map with either AP or SP, while a SP can only map with an AP. Any message passing through a port is checked against the port specification, and validation exception is thrown (for the test system to handle) if the check fails.

For distributed testing *SystemPorts* can be distributed to remote machines, other than the SystemComponent runs on. Test system communication with the SUT is controlled through the SystemComponent's ports (local or remote). The remote implementation of *SystemPort* is one of the main feature of Jata for distributed testing support and is used in this thesis case-study for testing of parallel components.

Jata uses Java RMI is to connect remote ports with test system ports. Automated port mapping in Jata is only supported within local execution context (see line 13 in listing 2.2 below). Mapping remote ports with test system ports requires manual configuration of the mapping (e.g. with RMI client and server configuration).

Alt statements

The *alt* (alternative) concept is the same as in TTCN-3. The response from a SUT for a given stimulation message needs to be evaluated for correctness and also with respect to response time. In TTCN-3 and Jata, *alt* statements are used to check the possible responses. As in TTCN-3 a single Alt can have multiple branches and all branches are evaluated sequentially according to the occurrence order. To see what branch was hit in Jata, the result index maintained by the Alt class is checked.

The powerful expression of the TTCN-3 test language is clearly demonstrated in the definition of *alt* statements (lines 15-35 in listing 2.1). The TTCN-3 *alt* statement resembles a single switch-case sentence which is arguably easy to define and read. Jata relies on Java to express the same behavior using objects, methods and conditions. The Alt class is used for the possible branches and the AltResult class for the response (lines 18-27 in listing 2.2). The proc() method of the Alt class (line 25) has to be explicitly called to start the behavior. A conditional statement (lines 30-40) is then needed to structure the handling of the response. Those familiar with TTCN-3 might grasp this structure easily, but for those who are not, this might seem an awkward procedure.

Codecs and Adapters

Adapters are used in Jata for connecting the test system with the SUT. *Codecs* (Coders/Decoders) are used to map between test system data types and the SUT data types. When a test component stimulates the SUT, an encoding of the message is done within the Adapter method of a system port. Decoding functionality for messages from the SUT is implemented within the message object itself.

As described in Section 2.4.2, TTCN-3 has two layers for the SUT adaption: An *abstract test system interface* and a *real time test system interface* as seen in figure 2.6. Jata does not include this strict layering, the seperation is honored by the means of the abstract (AP) and system ports (SP), where test components contain abstract ports which are mapped to system ports encapsulated by the SystemComponent, i.e. system adapter. The system ports contain the physical interface implementation.

Physical adapters and codecs concepts are not TTCN-3 language elements because TTCN-3 focuses on abstract test suite definitions. TTCN-3 test engineers develop adapter and codecs implementation in a general programming language such as Java. In Jata these mappings are possibly simpler to maintain because the test system, adapter layer and the codecs are all written in the same programming language.

Verdicts

Jata uses the same verdict mechanism as TTCN-3. A test component maintains the test result as a VerdictType. The verdict types are the same as in TTCN-3 and share the same values: *pass*, *fail*, *inconc*, *none*, and *error*.

Code Example

Listing 2.2 shows a typical test case written in Jata. A main entry method (line 43) is needed to start a test case. The test case itself is the Case method (line 7) which is overridden from the Jata TestCase class.

Expected testing response is defined first (line 10). The port mapping is done with the help of the PipeCenter class (line 13). In this case the MTC has a single abstract port which is mapped to a system port for a single request-response stimulus. This example does not use a remote port and the system adapter, along with its port, runs in the same runtime context as the MTC. The stimulus is sent via the MTC port which is mapped, as previously described, to the system port (line 16). An alt statement is defined (lines 18-24) and then started (line 25). The result is received (line 27) and handled based on defined branches of the alt statement (lines 30-40). If the expected response was received the MTC is indicated so, by a call to its Pass() method (line 31). Jata does not provide test runners as JUnit, other than textual logging, so the test verdict is written out at the end of the test (line 46).

```
 1  public class MyTestCase extends TestCase<AbsComp, SysComp> {
 2    public MyTestCase() throws JataTestCaseException {
 3      super(AbsComp.class, SysComp.class);
 4    }
 5
 6    @Override
 7    protected void Case(AbsComp mtc, SysComp sys)
 8                                throws JataException {
 9      // The expected response
10      String expectedResponse = "42";
11
12      // Map ports between MTC and SUT (system component)
13      PipeCenter.Map(mtc.abstractPort, sys.systemPort);
14
15      // Send the stimulus via MTC's port
16      mtc.abstractPort.send(new QueryMsg("Meaning of life?"));
17
18      Alt alt = new Alt();
19      Timer t = new Timer();
20      t.start();
21
```

```
    alt.addBranch(mtc.abstractPort,
                  new ResponseMsg(expectedResponse));
    alt.addBranch(t, new SetTimeMessage(100));
    alt.proc();

    AltResult result = alt.getResult();

    // Handle the case for an expected answer.
    if (result.index == 0){
      mtc.Pass();
    }
    // Handle timeout
    else if (result.index == 1) {
      mtc.Fail();
    }
    // Handle any other message
    else {
      mtc.Fail();
    }
  }

  public static void main(String[] argv) throws JataException {
    MyTestCase myTestCase = new MyTestCase();
    myTestCase.start();
    System.out.println("MyTestCase result:" +
                       myTestCase.getVerdict());
  }
}
```

Listing 2.2: Jata typical test case structure

Compared to the TTCN-3 code example in listing 2.1 the two structures are very similar and shows that Jata tries to mimic TTCN-3, as much as it can, in test case structuring.

3. Related Work

This chapter describes the related work where software testing meets cloud computing. It is worth noting that because of the recent introduction of cloud computing, software testing researches are generally lacking for the field.

To get an overview of the importance of the research topic, and the availability of literature, table 3.1 shows the number of articles found by searching for for the keywords "Cloud-Computing" and "Software-Testing" in the literature search engines *Google scholar* and *LibHub*[1].

Year	LibHub	Google scholar
2010	22	97
2009	4	26
2008	3	9
2007	0	1

Table 3.1: Literature search results

No evaluation is made here on the relevance of the results returned, i.e. how relevant each result is to both cloud computing and software testing. Of the 97 results returned by Google scholar only very few relate to testing of cloud applications or within the cloud. It can in fact be stated that research is lacking in the area of software testing for clouds, especially for testing *of* the cloud. The numbers indicate however that in the last years these concepts together have been receiving more attention from academic researchers. This goes hand in hand with Google's trend results on search trends for Cloud-Computing's rising popularity as seen in figure 3.1.

The upper section of the diagram shows search volume index for the two terms and the lower for news reference volume. The blue line shows search trends for software testing while the red one shows clearly the rising trend of cloud computing. Software testing does not show up on the news reference volume while the news chart-line for cloud computing bears signs of hype in the commercial community. The flags on

[1]SemperTool's LibHub is Lund University search tool to discover and access the institution's subscribed to and recommended information resources.

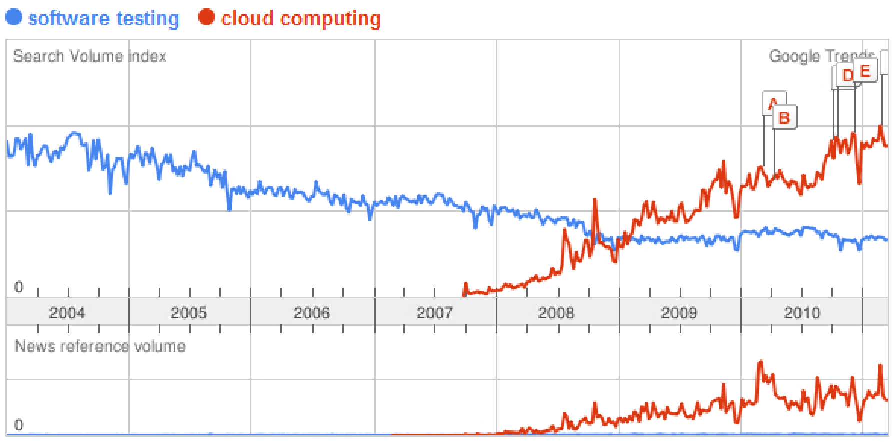


Figure 3.1: Google trends diagram [57]

the cloud computing search chart-line, marked A-E, are links between certain news items and changes in the graph, which is a special feature for trend analysis.

For an overview of related work in the field of software testing for cloud computing, the work is grouped into the three categories of STITC (see Section 2.3.5). The scope of this thesis is testing *of* the cloud, with related work discussed in Section 3.2.

3.1. Testing *in* the Cloud

Most of the current related work for software testing and cloud computing focuses on testing in the cloud, e.g. for configuration testing and load testing.

YETI on the Cloud, by Oriol and Ullah [44] discusses and proposes a testing tool for distributing software testing over multiple computers in a cloud, for performance gain in running the tests. YETI (York Extensible Testing Infrastructure)[2] is an automated random testing tool for Java applications. A YETI testing session consists of a sequence of calls made to methods at random using random generated arguments ("monkey crash test"). Oriol's and Ullah's paper presents cloud-enabled version of YETI which relies on on map/reduce implementation of Apache Hadoop[3] to distribute the testing sessions on remote machines and recombine them at the end of the test.

In *Cloud-based Performance Testing of Network Management Systems*, by Ganon

[2]YETI. http://yeti.origo.ethz.ch/
[3]Hadoop. http://hadoop.apache.org/

and Zilbershtein [23], a method for performance testing of network management systems (NMS) is presented by a case study where a NMS was tested using commercial cloud computing services (Amazon EC2). The case study involved instantiation of large numbers of virtual network elements in a cloud infrastructure, to simulate large scale networks for NMS testing purposes. After the network of virtual elements is up, it is used for performance testing of the NMS. They used cloud infrastructure instead of running a simulator or other infrastructure stating that the cloud infrastructure provides less expensive and more scalable implementation.

Cloud9 [13, 16, 18] is a platform for automatic testing which parallelizes *symbolic execution*, an effective, but still poorly scalable test automation technique to cloud infrastructures. Currently supported platforms are Amazon EC2 and Eucalyptus. Symbolic execution tries to eliminate the need for humans in selecting test inputs, but the method faces many challenges, such as it is requires an "expensive" code branch search algorithm. Symbolic execution algorithms require high memory consumption and are CPU-intensive, both of which are roughly exponential in program size. The aim of Cloud9 is to overcome these challenges and offer symbolic testing techniques as an easily accessible web service. By using large clusters of cloud instances, the plan is to make this automated test case technique viable and practical for software testing. Cloud9's service cost is proportional to the size of the SUT. The pricing model is based on charges according to test goal specifications that users provide.

D-Cloud [8, 28, 29] is an example of work that can fall either in category of testing *in* the cloud, or *of* the cloud. It enables automation of system configuration and the ability to run test cases simultaneously, which places the framework in this category of testing *in* the cloud. The framework has also the ability to emulate hardware faults which can put it in the category testing *of* cloud applications, i.e. how applications respond to and handle hardware failures. Eucalyptus is D-Cloud's supported cloud management framework.

3.2. Testing *of* the Cloud

The focus of this thesis is testing *of* the cloud. Little work for testing of cloud application has been published. Chan, Mei and Zhang [17] discuss testing of cloud applications and propose testing criteria based on modeling of cloud applications. One criterion is defined to test whether cloud application performs correctly after horizontal scaling of the cloud, and addresses the problem that the number of possible scaling ways are potentially infinite, and thus, infeasible to test every configuration. Horizontal scaling, also known as scaling out, is adding more machines, generally inexpensive commodity hardware, to the environment. Cloud computing strengths lie to a large degree in the concept of horizontal scaling. Vertical scaling, also known as scaling up, in contrast means adding more hardware resources to the same machine, generally by adding more processors and memory.

Various studies have been made for testing distributed systems. In *Distributed Unit Testing* [54], by Runtao et al., the authors introduce the *DisUnit* framework to address challenges of distributed testing. Issues such as synchronization, complex configurations, and test result management. Synchronization is one important aspect in distributed testing and a time-out feature was not included in previous version of JUnit, which DisUnit solves. The DisUnit framework is used to control testing by automatically setting up remote machines, running test-cases and shutting down the servers. DisUnit also removes the configuration information from the test code, which permits the test to run in different environments and configurations without having to edit the code, making DisUnit valuable for configuration testing.

DisUnit's testing architecture is based on what is described as *in-container testing* [40]. With in-container testing the test cases run within the same context as the run-time container for the SUT, for instance a web server or application server. Jakarta Cactus[4] is an example of in-container test framework for Java Servlets, Enterprise Java Beans and Java Tag libraries. By running the test cases within the same runtime context as the components under test, the test cases have access to semantic features like configuration files and available services which are valuable for the test. DisUnit currently supports in-container testing for Java HORB[5] and CORBA applications. Detailed information on DisUnit's architecture and communication between the distributed testing components is however missing making it difficult to argue about the solution for distributed testing. The downloadable version of the framework as promised by the paper is also missing.

Cloud computing has evolved from grid computing technologies. Some research has been done on the testing of grid computing applications. Rings, Neukirchen and Grabowski [50] use TTCN-3 for testing of a grid application, in particular testing work-flow of grid applications.

This thesis is largely inspired by the grid testing methods introduced by Rings, Neukirchen and Grabowski [49, 50]. Instead of using TTCN-3 in a grid environment this thesis investigates cloud testing challenges with the support of the Jata test framework [60]. Jata uses TTCN-3 the concepts and therefore it is possible to use testing architecture ideas from the previous grid work. The focus of the previous work testing was on testing of the distribution management aspects of the SUT as is also done in this thesis.

[4]Jakarta Cactus. http://jakarta.apache.org/cactus/
[5]HORB. DisUnit author's middleware for distributed Java ORB applications

Jata combines the test case structure of JUnit and testing concepts of TTCN-3. Its initial version and research ideas were published recently and it has only been used by the authors in a case study for web services functional testing [60]. Usage of Jata for testing parallel functionality of distributed systems or systems within cloud environment, as is done in the case study of this thesis, has not been addressed before. TTCN-3 and Jata are described in foundation sections 2.4 and 2.5 respectively.

3.3. Migrating Testing *to* the cloud

This category contains ideas of testing as a service provided by the cloud vendor, or other third party provider. The concept involves moving the testing process, test assets, and test infrastructure from on-premises *to* the cloud.

In the cloud world it is common to name services in the style of *X-as-a-Service*. Various work has been done in the field of *Software-Testing-as-a-Service* (STaaS). The service was defined by Leo van der Aalst as:

> Software Testing as a Service (STaaS) is a model of software testing used to test an application as a service provided to customers across the Internet. By eliminating the need to test the application on the customer's own computer with testers on site, STaaS alleviates the customer's burden of installing and maintaining test environments, sourcing and (test) support. Using STaaS can also reduce the costs of testing, through less costly, on-demand pricing. [2]

With STaaS the cloud can decrease costs for testing activities. Section 2.3.6 described an example of test-lab for cost reduction related to resources needed for different testing activities. STaaS takes that model even further in the way that not only hardware is scaled on-demand but also the testing software and even test engineers. The idea can be seen as test service outsourcing where test machines, test software and, possibly test engineers are outsourced.

When to Migrate Software Testing to the Cloud?, by Parveen and Tilley [45], was presented at the *2010 Software Testing in the Cloud* (STITC) conference. The paper discusses what variables affect decisions in migrating testing to the cloud. According to the authors the decision should be made by viewing the matter from two perspectives: the *characteristics* of an application under test, and the *types of testing* performed on the application. Characteristics to consider are test case independence, the operational environment (test bed) and the ability of an application's interface to be tested against, i.e. does it have an API or only graphical user interface. They conclude that the types of testing to be appropriate for testing in the cloud are unit testing, high volume automated testing (HVTA), and performance testing.

Riungu et al. did a qualitative study [51] to identify research issues for software testing in cloud computing, where interviews were conducted with managers from different organizations. The motivation for the work was the fact that little had been written about the topic before and academic research in the area is scarce. The research aim was to explore and understand the conditions that influence software testing as an on-line service in order to point out research issues for software testing in the cloud. The discussion with the managers took into account the pros and cons of using the cloud for various test services. They categorized their issue findings according to *application issues*, *management issues*, and *legal- and financial issues.* The authors claim that by addressing the issues listed under the categories, researchers should be able to offer reliable recommendation for practitioners in the industry.

Riungu et al. made an earlier study [52] were they acknowledged that cloud computing was going to be a central platform for on-line testing and service delivery and also listed various research issues.

Cloud9, D-Cloud, and YETI on the cloud, which were covered in Section 3.1, can be considered as an examples of STaaS services that have already available in the cloud. Examples of commercial services are *Sauce onDemand*[6] and *UTest*[7]. Sauce OnDemand is based on Selenium[8] web testing system, that enables web applications to be tested by multiple browsers running in the cloud. Through UTest, test engineers are available on-demand which provides software testing solutions to its customers through access to professional testers community. Large available communities like UTest have been described as *crowd-sourcing* [30].

Another testing service in the cloud is *Test-Support-as-a-Service* (TSaaS), and is described by King and Ganti in *Migrating Autonomic Self-Testing to the Cloud* [38]. Autonomic self-testing (AST) and TSaaS make use of the processing capabilities provided by the cloud to improve self-testing processes. TSaas provides cloud customers automated test operations (setup, assertion, and tear-down) for remotely hosted cloud services. The cloud vendors provide the self-test harness for monitoring and validating dynamic adaptations and updates to their hosted services, with the help of AST, where test managers are deployed throughout the cloud to validate dynamic adaptations and updates of cloud instances.

[6]Sauce OnDemand. http://saucelabs.com/ondemand
[7]UTest. http://www.utest.com/
[8]Selenium. http://seleniumhq.org/

4. Testing Cloud Applications

In this chapter a case study for integration testing a distributed system within cloud enviroment is described.

Section 4.1 contains an overview of the Sunflow rendering system that is used as the system under test for the case study, including a description on how Sunflow was adapted to run within a cloud environment. Section 4.2 introduces the testing for the distributed rendering application in the cloud. In section 4.3 the test case design and in section 4.4 the Jata test case specification for the case study are described in detail. Finally in section 4.5, the procedure of setting up and running the test bed is described.

4.1. Cloud Application Under Test

This section describes the cloud application that is used as the system under test (SUT) for the case study in this thesis. *Sunflow*, multi processing image rendering application, was selected as the application under test for its parallel functionality.

> *Sunflow is an open source rendering system for photo-realistic image synthesis. It is written in Java and built around a flexible ray tracing core and an extensible object-oriented design.*[1]

The reason for using parallelization and concurrent computing application is to explore to the full, with advanced testing techniques, the challenges proposed in testing distributed systems within a cloud environment.

Image rendering is an extremely time consuming computing task, therefore rendering systems like Sunflow have been created to split up the task into smaller subtasks for parallel computing. For a typical rendering job, one master node is needed for the application management and several worker nodes for solving individual rendering tasks.

[1] Sunflow. http://sunflow.sourceforge.net/

The control part or the distribution management of the application is therefore a good candidate for testing within a cloud environment. One cloud instance is needed for the master node and an arbitrary number of other instances for the workers, where the focus of the testing is on the communication of the distributed components involved.

4.1.1. Image Rendering

Image rendering is the process of generating an image from a model which is defined in a *scene file*. A scene file contains information necessary for the rendering such as geometry, viewpoint, texture, lighting and shading. Figure 4.1 shows a 3D view of a model (scene) before any rendering has been done, whereas figure 4.2 shows the rendered image result.

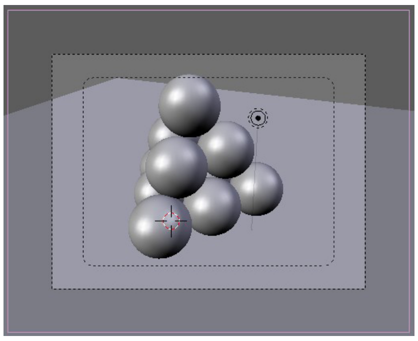


Figure 4.1: 3D view af a scene [39]

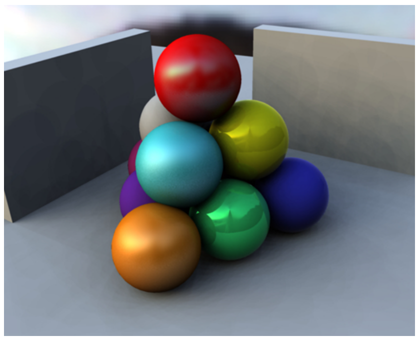


Figure 4.2: Rendering results [39]

An image rendering job is split into *buckets* of smaller tasks. Each bucket represent a part of the image. The working threads for rendering a bucket is called a *worker*. In the cloud computing environment each worker runs on a dedicated cloud instance.

4.1.2. Cloud Support for Sunflow

Sunflow's work distribution algorithm distributes the workload for parallel processing to multiple CPU cores on a single computer. The system has however been adapted to run on both grid and cloud platforms. Ragnar Skulason, fellow student at the *Faculty of Industrial Engineering, Mechanical Engineering and Computer Science* at the University of Iceland[2], adapted Sunflow to run on Amazon EC2, for his Masters thesis research *Architectural Operations in Cloud Computing* [55].

[2]University of Iceland. http://www.hi.is/en/introduction

Skulason's research is about usage of an *Architectural Scripting Language* (ASL) for clouds. Architectural scripting is a way to model the dynamic aspect of runtime and deployment-time software architecture. The notion of architectural scripting and the exploration of its theoretical and practical utility was introduced by Ingstrup and Hansen in *Modeling Architectural Change: Architectural scripting and its applications to reconfiguration* [34].

Skulason presents *Cloud-ASL*, an external domain specific language (DSL) which enables architectural operations and architectural scripting in cloud computing environments to create and initialize cloud instances. To model and test the Cloud-ASL, a working architectural software prototype was implemented and named *Turnip*. The Turnip software is an extension of Sunflow to enable its distributed ray-tracing features within a cloud computing environment.

The objective with the Sunflow extension was to create a case study application to use with Cloud-ASL, mainly for launching and destroying worker cloud instances [55]. In Turnip, Sunflow is adapted to run the different rendering workers on separate computers in a cloud. The distribution algorithm is based on Helios[3], which supports Sunflow distributed rendering computations on JGrid [46], a grid service implemented with Jini[4] technology.

Turnip is used in this thesis case study as the SUT with focus on testing the application management features. For deployment and startup of the SUT within Amazon EC2 the Cloud-ASL is utilized, but it is not part of the test focus in this thesis case study. Figure 4.3 shows an overview of the machines and communication protocols involved in the testing of the application. The following section 4.1.3 describes the software architecture and communication protocols for Turnip.

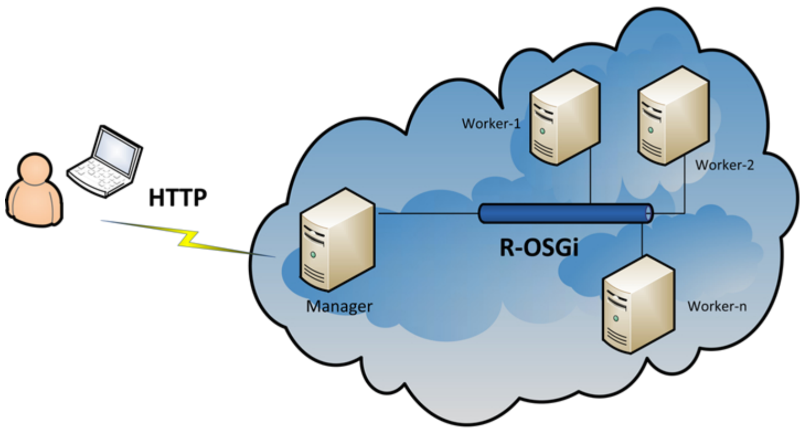


Figure 4.3: Turnip rendering application in the cloud

[3]Helios. http://sfgrid.geneome.net/
[4]Jini. http://www.jini.org

4.1.3. Turnip Rendering Application for the Cloud

The Turnip application includes a web management console. Figure 4.4 shows the console when an image rendering is in process. The scene file behind this particular rendering uses 300 buckets to complete the job. In the partially rendered image roughly 100 buckets have been rendered resulting in 30% of the final image.

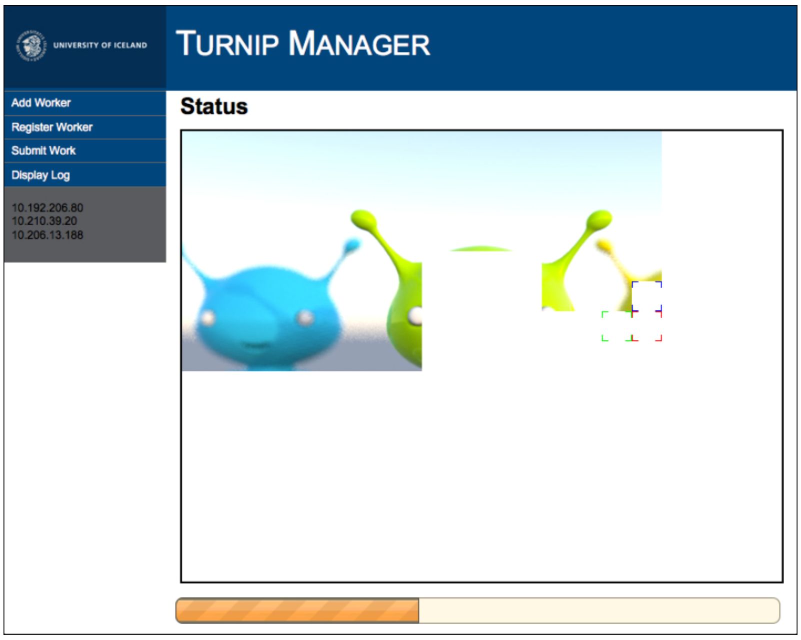


Figure 4.4: Turnip web management console

The left hand side of the console contains the actions menu. The Turnip console provides only the basic functionality needed to render images. Available actions are 'add worker', 'register worker', 'submit work', and 'display log'. In this prototype version of Turnip, the console provides no means of selecting what scene file should be rendered. The file needs to be uploaded to the server manually. Below the actions menu is the list of workers, IP addresses, in use.

A typical scenario for the full life-cycle of a rendering job is following:

1. Start Turnip web application on a cloud machine
2. Add 1..n workers
3. Submit work (start rendering)

The actual rendering job (i.e. the image result) is not the focus of the testing. Rather it is the application management behind a rendering job that is the focus of the black-box functional testing in this thesis. The following section describes the application management features and the scenario which is tested.

Application Management

Figure 4.5 shows an architectural overview of the Turnip application in a cloud environment. The application management functionality, which controls distribution of work, is the point of interest for software testing in this case study.

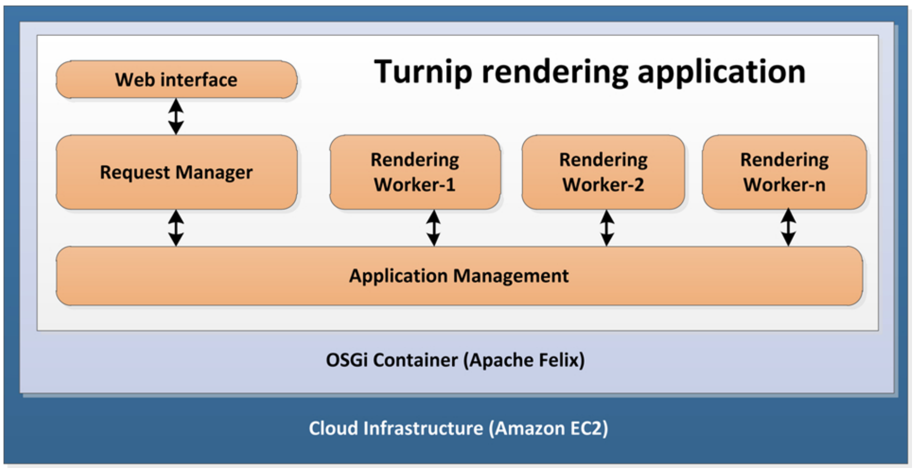


Figure 4.5: Turnip application architecture

The application management part of the software is responsible for adding and removing workers, and distributing rendering tasks (buckets) to the workers. Adding a new worker involves launching a new cloud instance. After the number of workers have been selected, a rendering task can be submitted. The software component within Turnip which encapsulates the application management is called *Request Manager* (see figure 4.5).

The testing focus in this case study is on the application management behavior for rendering jobs. The scenario is depicted in the sequence diagram in figure 4.6. The rendering scenario includes two workers in the cloud and only two buckets, for the sake of simplicity. This particular scenario forms the basis for the application testing described in 4.2.

For a rendering job the request manager is responsible for interpreting the scene file, assigning tasks to workers, and collecting and composing the results from the workers. After a work request is submitted the request manager uses the Sunflow engine to divide the request into buckets for rendering. For each available worker the request manager initializes it for rendering (interactions 2 and 3). After a worker

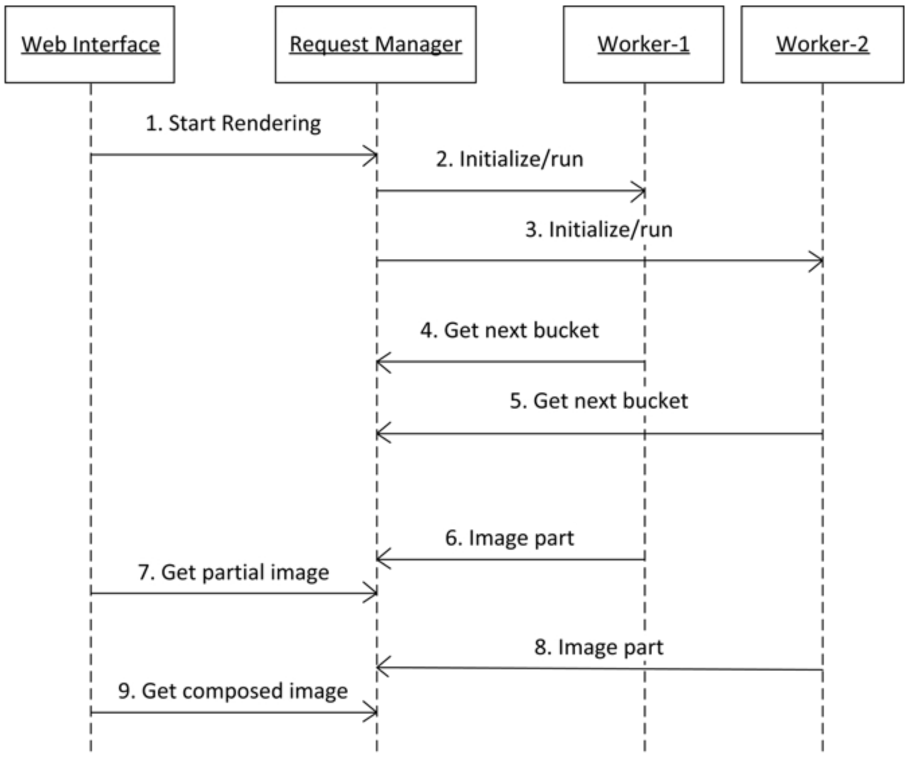


Figure 4.6: Rendering job scenario

is initialized it asks for the next bucket for rendering (interactions 4 and 5). The worker is returned task information about the bucket from the request manager, which the worker uses for partial rendering of the image. After a worker finishes a task it sends an image part result back to the request manager (interactions 6 and 8), which is merged into the partial result image. The continuously changing image is repeatedly written to a file directory on the server, where it is available for the web interface (interaction 7).

The result, partial and final, is presented in the Turnip web management interface, by periodically polling the working image, using AJAX (Asynchronous JavaScript and XML) features. Once all tasks are completed, the whole image is available in the web interface and that fact is indicated with a finished progress bar status (interaction 9).

Software Architecture

The application software architecture influences the testing strategies described in section 4.2. Figure 4.7 contains an UML deployment diagram for the Turnip application.

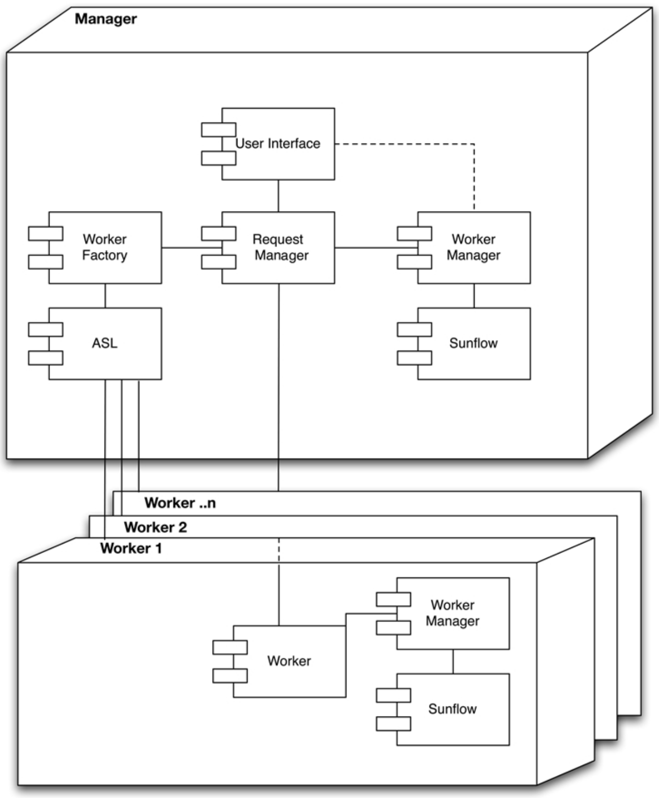


Figure 4.7: Deployment diagram for Turnip [55]

The Sunflow application and the cloud support extension of Turnip is written in Java. The *Request Manager* encapsulates the core functionality of the extension. It is responsible for communication with workers, i.e. assigning tasks and publishing the partially rendered image by saving a new copy of it in a web directory where it is available for the user interface. The web user interface is implemented as a Java Servlet and the web page functionality is provided through AJAX. The user interface provides actions for rendering jobs and shows the progress of rendering tasks by loading the current status of the rendered image provided by the request manager.

The Cloud-ASL is responsible for configuration and automatic deployment of application components by means of the *Open Services Gateway Initiative* (OSGi)[5] framework. The OSGi framework provides a runtime for component based life-cycle management in Java, where services are packaged as *bundles* which provides the ability to dynamically install and uninstall deployment units to an OSGi runtime server. The bundles also provide means to publish and search for packaged services [3].

[5]OSGi Alliance. http://www.osgi.org/

The manager and workers are deployed as OSGi services and the communication between manager and workers is via remote OSGi (R-OSGi). The workers are created by the Cloud-ASL, which uses SSH and Telnet to access the OSGi remote shell on the cloud instances. The *Worker Factory* and *Worker Manager* components encapsulate the OSGi communication for creating and launching workers within the cloud. The manager bundle and the worker bundles include their own copy of the Sunflow Java library.

4.1.4. Amazon Machine Image

The AMI created by Skulason [55] used for this case study is public with ID **ami-4552bb2c** in Amazon's instance store. The same image is used both for the manager and worker machines. The runtime software installed on the AMI is listed in table 4.1.

Runtime	Version	Description
Operating System	32 bit Ubuntu 9.10	Commercially sponsored Debian-derived Linux distribution.
Java runtime	OpenJDK 1.6	Open source implementation of Java SE 6
OSGi runtime	Apache Felix 3.2.0	OSGi R4 Service Platform for OSGi bundles
HTTP container	Jetty (Eclipse/Codehaus)	Included in the Felix distribution

Table 4.1: Run-time software for the case study application

The AMI includes the necessary runtimes for the Turnip rendering application, with Ubuntu Linux as the operating system and Java SE 6 installed. *Apache Felix* is used as the OSGi service platform, which includes the *Jetty*[6] web application server.

Note that the Turnip application binaries are not part of the image, because they are installed with Cloud-ASL features after the cloud instance has been launched.

[6]Jetty. http://www.eclipse.org/jetty/

4.2. Test Plan

The structure of this section is based on concepts from the *Test Plan* chapter of the *IEEE Standard for Software and System Test Documentation* (IEEE Std 829-2008) [32]. The standard is introduced in section 2.3.7 on test documentation. This section and the following describe the test plan, the test design, and the test case specification for a distributed image rendering application running in the Amazon EC2 cloud environment. Jata was described in section 2.5 and the application features were described in section 4.1.

Following the test plan in this section the test design is covered in section 4.3 whereas the test case specification is covered in 4.4. A brief description of the test procedure is in section 4.5. Note that the IEEE format is only used as basis for the structure of this thesis text. The following sections might omit details that are not relevant to the research purpose of this thesis, but might be needed in a conventional test documentation. The thesis also adds information to support the thesis work, such as source code listings, as is normally not done in a conventional software test documentation.

4.2.1. Introduction

The aim of this case study is to point out the challenges and obstacles in testing applications running in elastic cloud environment. Another objective is to evaluate Jata's applicability for the task. The focus is on software testing for cloud application. The Turnip application [55], a cloud extension of Sunflow rendering system adapted for Amazon EC2, is used as the system under test. The application is a parallel computing application that provides means for advanced software testing techniques.

As opposed to 'real world' software testing the objective is not to reveal defects in the application itself or the cloud service platform it runs on. Rather the objectives are to reveal issues for software testing within cloud environment, and to evaluate Jata for testing advanced cloud application functionality.

Figure 4.8 shows a high level overview of the components within the distributed system under test. The main program, i.e. the application management, is the focus of the test. It distributes rendering tasks to workers for rendering small parts of the final image which is the output from the application. A common and straightforward way of testing is by unit testing and system testing. Those levels are depicted, however they are not the focus of this thesis.

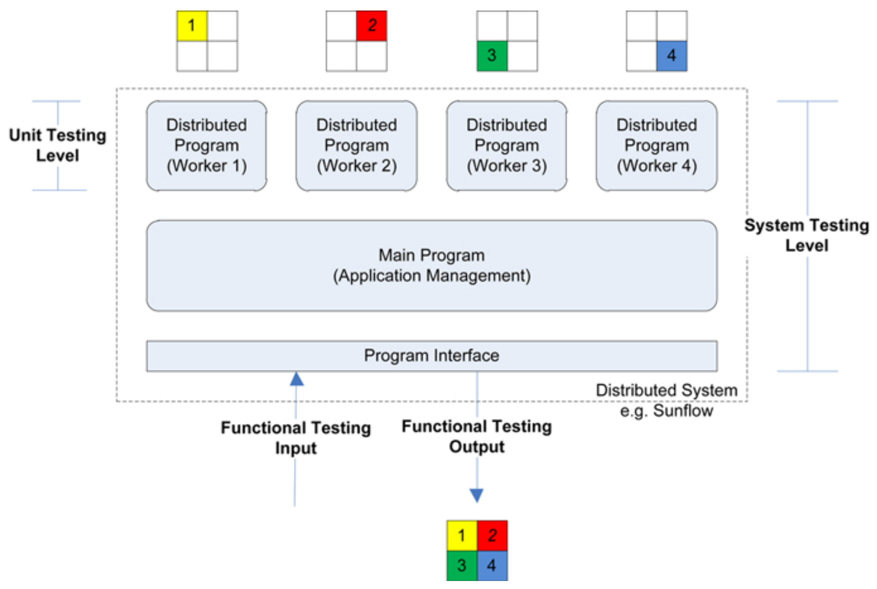

1
2
3
4
Unit Testing Level
Distributed Program (Worker 1)
Distributed Program (Worker 2)
Distributed Program (Worker 3)
Distributed Program (Worker 4)
System Testing Level
Main Program (Application Management)
Program Interface
Distributed System e.g. Sunflow
Functional Testing Input
Functional Testing Output
1 2 3 4

Figure 4.8: Application components

With system testing the whole application is used and the actual output is tested. System testing would verify that the correct image output is created by composing all partial images of the rendering tasks. In unit testing each worker, i.e. the partial image rendering algorithm, is tested in isolation.

The objective in this case study is testing the distribution algorithm with integration techniques. Figure 4.9 shows how test components are added to the test bed for integration and communication testing. Following sections describes the integration testing method.

4.2.2. Test items

The application management, i.e. the *Request Manager* in figure 4.7, is the item to be tested. The request manager, which provides the core functionality of the Turnip application, takes orders from the user interface to add workers or to start a rendering job. It uses the Sunflow engine to split jobs into tasks and delegate to workers. Finally it composes the final image out of all the image parts received back from the workers.

Figure 4.9 shows how integration testing is enabled by installing test components at each worker. A main test component (MTC) controls the stimulation and verification of the application behavior. Actual rendering does not take place, rather dummy image parts are returned from the parallel test components.

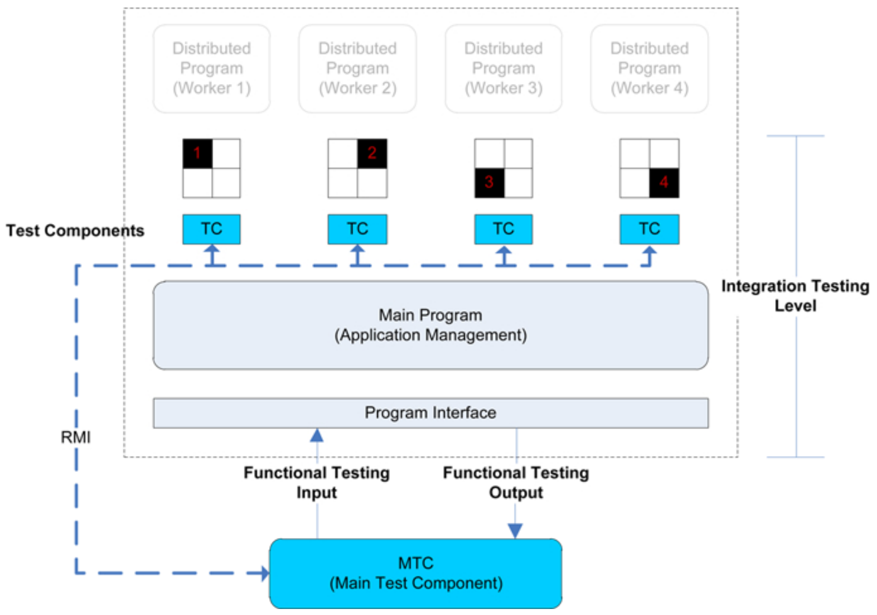

Distributed Program (Worker 1)
Distributed Program (Worker 2)
Distributed Program (Worker 3)
Distributed Program (Worker 4)
1
2
3
4
Test Components
TC
TC
TC
TC
Integration Testing Level
Main Program (Application Management)
Program Interface
RMI
Functional Testing Input
Functional Testing Output
MTC (Main Test Component)

Figure 4.9: Integration testing of application components

4.2.3. Features to be tested

The testing focuses on the application management, i.e. the communication with workers and the distribution of rendering tasks in order to verify that testing can be performed within a cloud computing environment. The integration between the web interface layer and the request manager is also under test. The testing is done by simulating an HTTP request to the web layer which needs to interpret the request correctly in order to start the application management. The rendering scenario which forms the basis of the testing is shown in figure 4.6.

Application output testing is done in the way that a “dummy” non-rendered image is assembled and verified. The dummy image does not represent the input scene file information. The application management uses the scene file however in deciding the number of buckets (tasks) needed for the job. The input scene file used requires 300 rendering tasks to be handled resulting in 300 image parts. The disk file size of the composed dummy image is verified in a simple manner to make sure all parallel test components have been issued the precise number of rendering tasks and returned dummy image parts accordingly.

4.2.4. Features not to be tested

The testing assumes that Amazon EC2 is adequately tested and that failures do not relate to the cloud service platform, but only to the cloud application, i.e. the distribution management aspects of the application. Testing the cloud middleware is therefore not within the scope of this thesis. The Cloud-ASL features of Turnip are also not within the scope of testing.

As stated earlier real output generation is not tested, which would be done with a system testing approach. Besides real output generation verification the performance of rendering might be of special interest from a user perspective.

It would be possible to utilize Jata for output verification and performance testing. Verifying the output is made in the case study for a composed dummy image. A Jata test case for output verification would be simple in the sense that it does not require parallel testing components.

4.2.5. Approach

The testing approach is a black-box functional integration testing, which is implemented with the Jata test framework. The Jata test architecture uses multi-party testing concepts from CTMF and TTCN-3.

4.2.6. Item pass/fail criteria

The pass/fail criteria is based on the scene file input for the rendering task. For the application management features the testing verifies that all available nodes are utilized for task submissions. Furthermore, based on the scene file, if 300 buckets should be rendered, the test fails if not exactly 300 buckets are rendered. For output evaluation, the file size of the resulting image is verified for correctness.

4.2.7. Test environment

The application under test was described in section 4.1. This section describes the software and hardware used in the case study test bed.

Eucalyptus was considered as a runtime cloud provider for this thesis case study.

However when starting the work, access to an Eucalyptus private cloud was not at hand. What favored the Amazon EC2 was the availability of distributed application source code and knowledge in our research group at the university [55]. Other reasons include EC2's easy accessibility, inexpensiveness, and good software API and documentation support.

Software Specification

The runtime software used for the SUT is installed on Amazon Machine Image id *ami-4552bb2c*, and was described in section 4.1.

Cloud-ASL features are used to launch the AMI within the Amazon EC2 cloud and install the Turnip application on the machine. That node is referred to as the application's 'manager' node.

The Turnip version has not been labeled but it is the latest version from the work of Skulason [55]. The Amazon EC2 used is the current (May 2011) on-line cloud service version. The Jata framework used for testing and is considered for evaluation is version 0.1, downloaded from `http://code.google.com/p/jata4test/`.

Hardware Specification

The hardware used for the case study is specified by the cloud instance type configuration. The case study used the Amazon EC2 standard 'Small' instance type with the attributes listed in table 4.2.

Type	RAM (GB)	CPUs (EC2)	Instance Storage	Platform	I/O Perform.	Price/ hour
Small	1.7	1	160 GB	32-bit	Moderate	$0.095

Table 4.2: Amazon 'Small' instance type (May 7, 2011)

Figure 4.10 shows the test bed for the case study. The test bed consists of four cloud instances and one test engineer machine for control and monitoring purposes. The four cloud machines roles are following:

- Instance 1: The test system (Jata)
- Instance 2: The Turnip master node
- Instance 3: Turnip rendering worker 1
- Instance 4: Turnip rendering worker 2

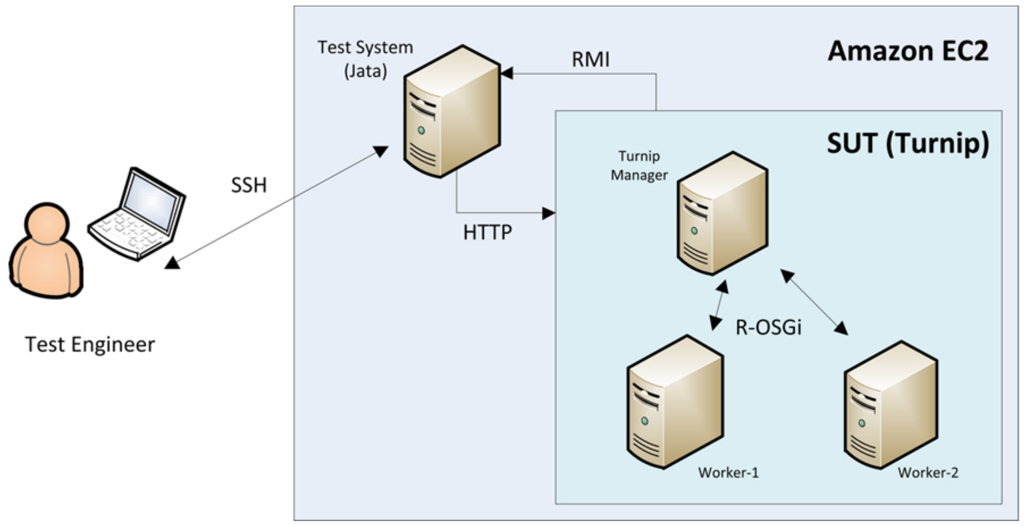


Figure 4.10: The test bed

The test bed figure includes communication protocols used in the testing. The test engineer uses SSH (Secure Shell) to communicate with the Jata test system machine, which runs in the cloud for simpler firewall configuration reasons. The Jata test components use Java RMI (Remote Method Invocation) for exchanging messages. RMI needs special security configuration for the cloud if participants are needed to run outside the cloud. Another argument in favor of including the test system machine within the cloud is that it improves performance of test case execution, because of the geographical location of involved instances.

The test system stimulates the SUT via HTTP through the application web interface. The SUT services are packaged and deployed as OSGi bundles and internal application communication uses the Remote-OSGi (R-OSGi) protocol.

4.3. Test Design

The following sections are structured using concepts from the *Test Design Specification* chapter of the IEEE Standard for Software and System Test Documentation [32].

4.3.1. Conceptual Test Architecture

The multi-party test architecture from the Conformance Testing Methodology and Framework (CTMF) is used for testing the parallel application management functionality. CTMF concepts were described in section 2.4.1, with conceptual view of the multi-party test architecture shown in figure 2.5.

The architecture of the SUT is shown in figure 4.11. Users access the system through a web interface where they can add workers and start rendering jobs. Each new worker started runs on a new cloud instance.

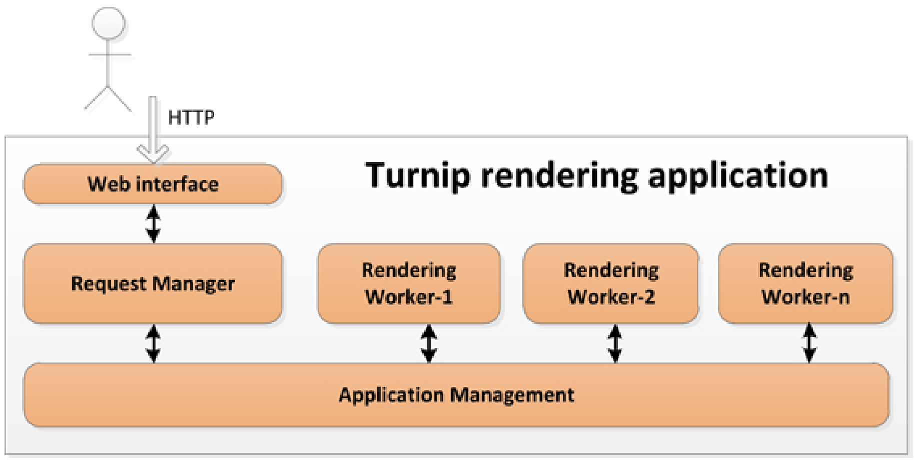


Figure 4.11: System architecture

The CTMF test architecture for the system architecture is presented in figure 4.12. Parallel test components that include communication ports are implemented for each rendering worker for the points of control and observation (PCO). High level architecture of the test component introduction was shown in figure 4.9.

According to rules of Jata and TTCN-3 the architecture includes a single main test component (MTC) which controls the test case behavior and verification. The architecture includes two parallel test components (PTCs), i.e. one for each worker. The MTC has three communication ports. One HTTP port for SUT stimulation and two input ports for messages from the PTCs. The SUT adapter is a Jata SystemComponent (adapter) and includes system ports for sending or receiving messages from the SUT. Remote ports run on each worker instance (test double) which communicate with the PTCs through the system adapter. The worker test double and remote port functionality is described in section 4.3.3.

For simplicity the test case contains only two workers, and two corresponding PTCs. The PTCs run on the same cloud instance, within the same Java virtual machine as the MTC, because of Jata's abstract port implementation restrictions. Jata abstract ports are implemented as POJOs (Plain Old Java Objects) without means of remote communication. An arbitrary number of worker test doubles (and PTCs) can be added to a test case.

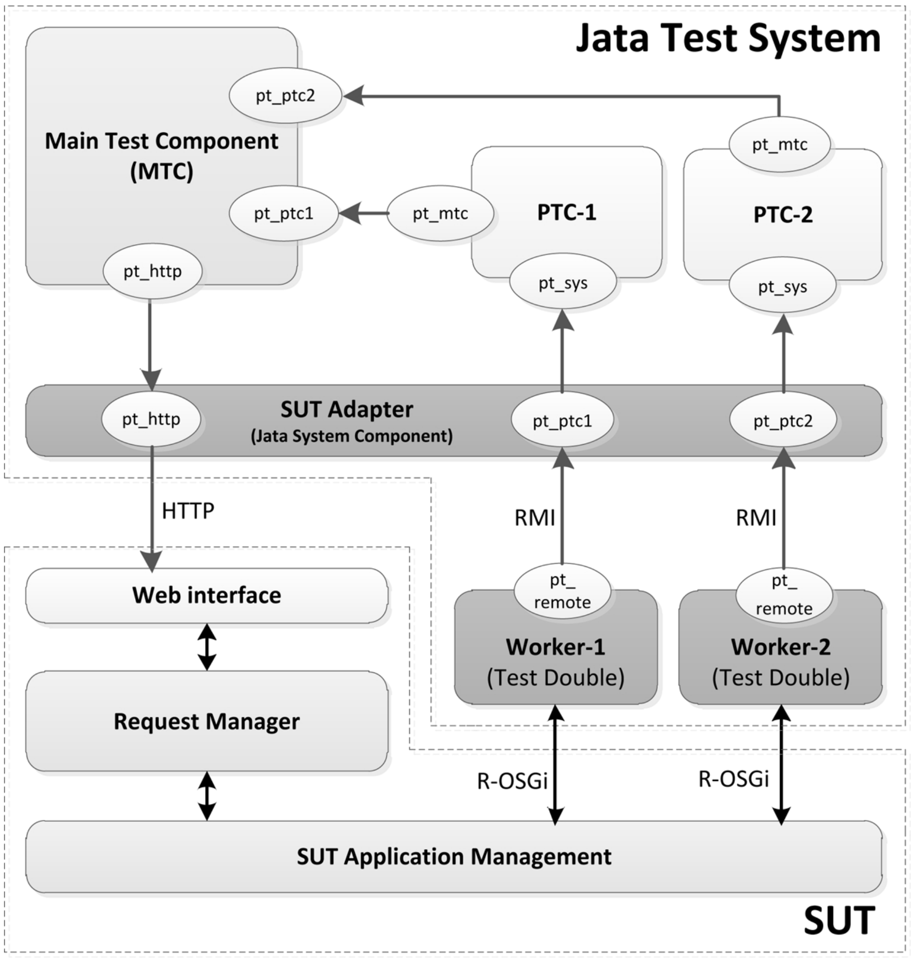


Figure 4.12: Detailed test architecture

A detailed description of the test components and their behavior is in the following section on the features to be tested.

4.3.2. Features to be Tested - Conceptual Test Scenario

The behaviour of the application management under test was depicted in the sequence diagram in figure 4.6. A conceptual view of the testing procedure is visualized in the sequence diagram in figure 4.13.

The interaction sequence is identical between the actual application rendering scenario and the test architecture scenario. The difference is that test components have been introduced to simulate both user behavior and bucket rendering in the architecture to control the interactions. Instead of a user (web-browser) an MTC is implemented to control stimulation and verification. The rendering workers have

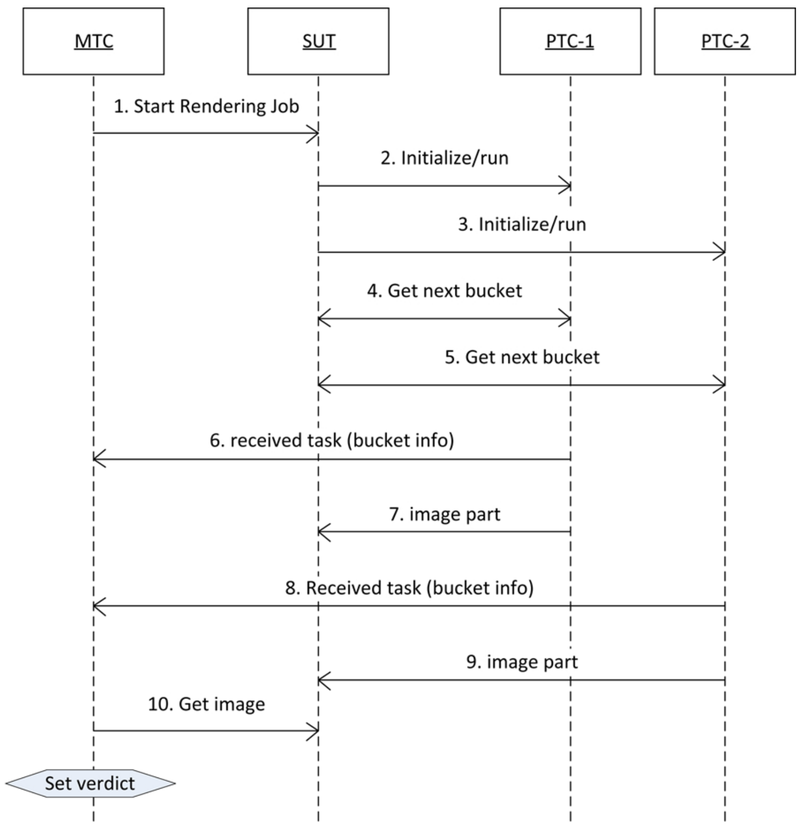


Figure 4.13: Conceptual test sequence diagram

been replaced by PTCs with communication ports to receive and send information intended for the workers. The MTC starts the test case by sending 'rendering job' stimulus (interaction 1). The SUT, application management part, who thinks it is announcing real workers to start rendering is actually calling PTCs (interactions 2 and 3). The PTCs ask for the next bucket to render as would be expected from a worker (interactions 4 and 5). When a PTC receives the bucket information it sends it to the MTC (interactions 6 and 8), which collects all knowledge about rendering tasks for verification. The PTC creates a dummy image to send back to the application management as is expected from rendering tasks (interaction 7 and 9). Finally when the MTC has received all expected bucket information it downloads the composed image result from the SUT (interaction 10). The MTC verifies the number of messages it received from the PTCs and the file size of the composed dummy image.

To implement this behavior the PTCs need to be deployed on the worker instance machines which is unfortunately not supported by Jata. The following section 4.3.3 describes a workaround with the help of a *Test Double*. Section 4.3.4 shows how the conceptual test architecture is implemented with the help of the test double.

4.3.3. Concrete Test Architecture Using Worker Test Double

In principle it is preferred to deploy the PTCs on the worker instances, which is unfortunately not easy to achieve with Jata. To solve this issue a *Test Double* is introduced. Test doubles are a group of testing objects such as: *stubs*, *mock objects*, and *fake objects*, as described by Meszaros [42].

The reason for having a test double implementation for the worker component instead of an actual PTC installed on the worker instance is that in the Jata environment PTCs are not able to run on remote machines like the case is with TTCN-3. Instead remote ports need to be located at the remote machines. The test double encapsulates Jata remote port functionality for workers.

The worker OSGi service was therefore switched out for a test double. The whole OSGi service for the worker is replaced by a Java component that implements the Java interface for the worker service. In the OSGi service descriptor file the test double is configured to be used instead of the real one. That way the test doubles can be started with the same methods as real workers are started, i.e. through the web interface command. Starting test doubles is not done within the test case itself. It can be done in a test setup script or through a test UI utility made for this case study. The test procedure details are covered in section 4.5.

This variation of a test double can be considered a stub that forwards rendering information received by the worker to the test system. The stub is implemented to support the remote port functionality in Jata, for forwarding information about buckets to a corresponding PTC. The test double is also responsible for sending fake image parts back to the request manager, without the need of Sunflow. Instead of using the Sunflow engine for rendering tasks, the test double sends information about buckets directly to the PTC through a remote port. With test double architecture it is easy to factor out the usage of Sunflow. Figure 4.14 depicts the functionality and behavior of the test double worker in a partial view of the test system. Note that in the picture the Jata SystemComponent (adapter) is omitted for simplicity and the communication between the test double remote port and the PTC is shown directly.

It would be preferred testing technique that the PTC would create the image part based on bucket information and send it to the SUT, instead of making it the worker's test double responsibility. There are however some difficulties with that approach. The request manager expects a remote OSGi call back with the image part, and the OSGi platform does not allow remote OSGi communication from non-OSGi application. Remote OSGi method invocations can only be made from within a OSGi container. The Jata test system runs as a standalone Java application and can not communicate via OSGi to the SUT. Therefore communication from a PTC

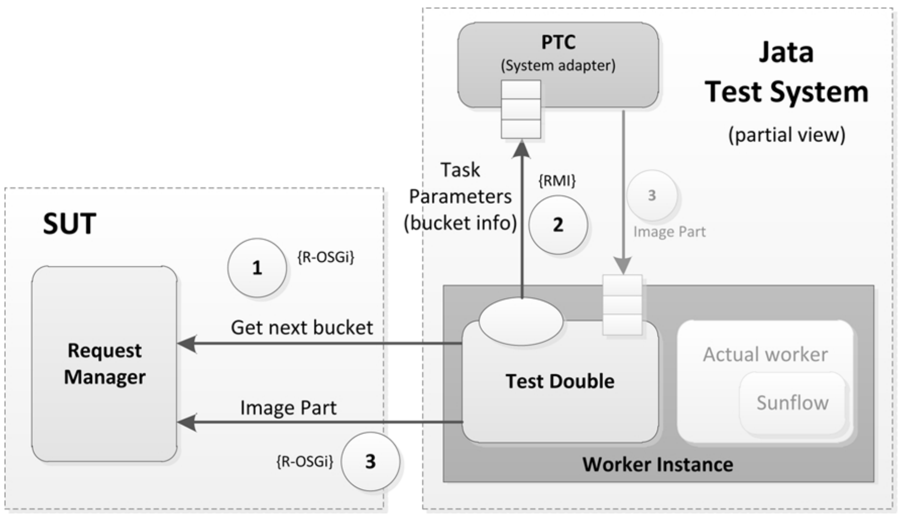


Figure 4.14: Worker test double implementation

to the request manager is not possible.

What could be done is to add a 'call-through' port to the test double that accepts messages from the PTC and forwards them with R-OSGi calls to the request manager. In figure 4.14 this is shown with the grayed-out communication between the PTC and the worker. That however introduces new challenges since using RMI for the communication, which is the preferred Jata way of communication, is non-trivial within OSGi containers. There are class loading issues to deal with when registering RMI server components (There are however no issues using client RMI code from within OSGi, which is what the test double remote port does). Therefore another way of communication to OSGi components might be more feasible for this test case structure, like through a web service interface.

What was decided to do for this case study was to follow the Agile *simplest thing that possibly works* [9] design strategy. As a result, it is not the PTC that creates and sends the image part for the SUT via the remote port, but the remote port itself without an involvement of the PTC. It should however be emphasized that testing functionality, for test case specification readability and manageability, should preferably be within test components (MTC and PTCs).

4.3.4. Concrete Test Scenario

This section shows how the conceptual test architecture, described in Section 4.3.2, behaves using the test doubles for the workers. The test behavior for the application management test case is visualized in a sequence diagram in figure 4.15. The test system, i.e. the MTC, stimulates the SUT by submitting 'start rendering job' to the web interface of the SUT (interaction 1). The stimulus is sent through an abstract port of the MTC to an HTTP port (pt_http) of the system adapter.

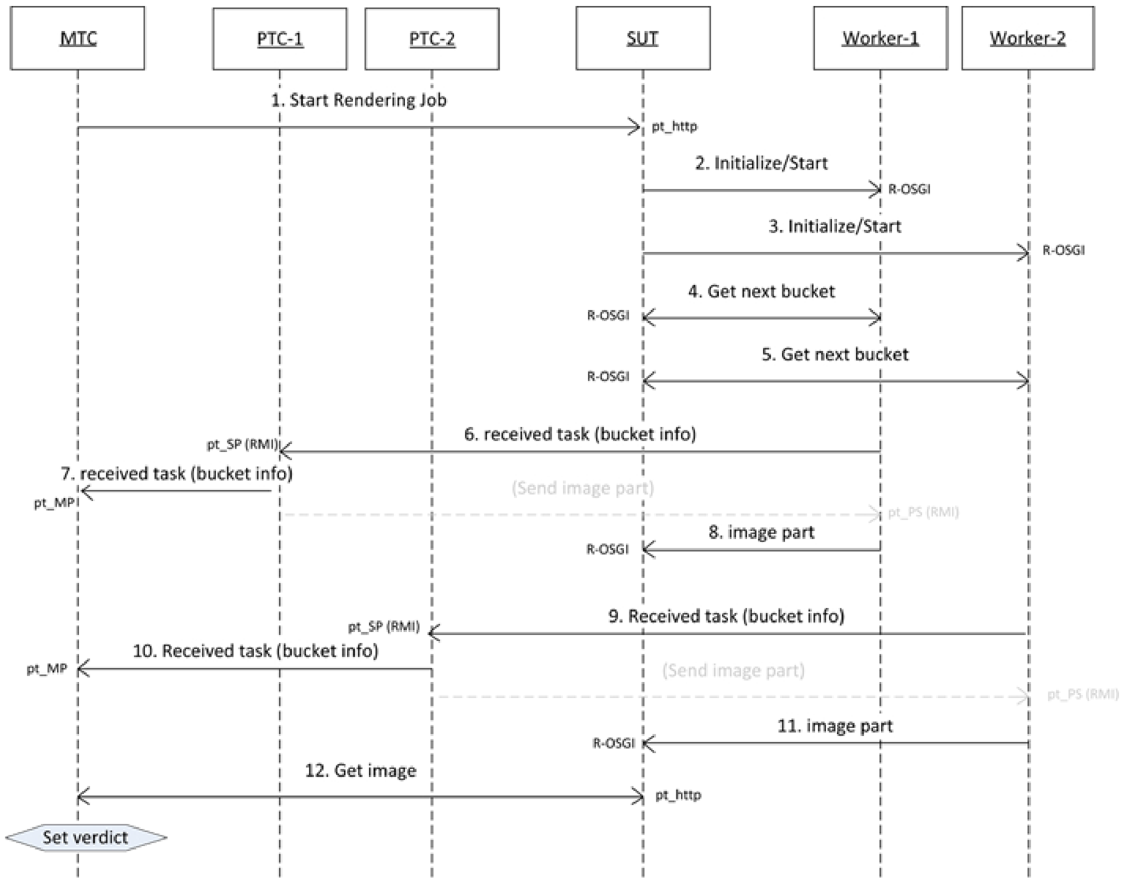


Figure 4.15: Concrete test sequence diagram

The input scene file for the rendering process is preloaded to the application manager's node, so no input parameters are needed in the stimulation message. The scene file is used to determine the job's tasks for the rendering workers.

After loading the scene file, the request manager sends 'run' (initialize) message to each available worker node (interactions 2 and 3), announcing that they should take part in a rendering job. After receiving a initialization message, the workers themselves are responsible for making calls to the request manager asking for the next bucket in line to render (interactions 4 and 5).

In the Jata test case implementation the worker OSGi service has been switched out for a *test double* [42]. The test double can be described as a stub that forwards rendering information from the request manager to the test system. It is also responsible for sending fake image parts back to the request manager instead of using

Sunflow. The test double architecture is further described in 4.3.3.

The Worker-1 test double sends the bucket information, through its remote port, to PTC-1 (interaction 6). The pt_PTC port on the system adapter receives the message and forwards it to the PTC abstract port (see ports structure in figure 4.12).

The PTC listens for any messages to arrive and forwards them all to the MTC for final composition verification (interaction 7). The (pt_MTC) abstract ports of the PTC and the MTC are used for that message delivery. If needed the PTC can inspect the incoming message content and do verification. In this test case however, the MTC handles the verification for all the messages at the end of the test case.

The test system is responsible for creating an image part, to send back to the SUT, which is the expected result from a worker after being assigned a bucket for rendering (interactions 8 and 11). For application management testing the image part can simply be a fake. Because the application management is being tested, not the rendering process itself, there is no need for real rendering with Sunflow. The worker's rendering implementation should be tested with other methods, such as unit testing. The worker test double therefore sends the same all-black image part back for all the rendering tasks. Ideally the PTC should create the image part based on bucket information and send it to the SUT, instead of making it the worker's test double responsibility. The grayed-out 'send image part' interactions (following interactions 7 and 10) indicate this preferred option. Reasoning for the test double implementation is discussed in section 4.3.3.

After a worker has finished rendering a bucket, it asks the request manager for the next bucket to render. Interactions from 10 to 13 are identical to the steps described above, only for the second worker and corresponding PTC-2. The same interaction sequence is used for all buckets until the rendering job finishes.

When all of the expected image part results have been received by the request manager, the whole image has been 'rendered' and clients can download the image. The MTC downloads the image through an HTTP port (interaction 14). The test verdict is based on the number of task submissions and the final image result. The image result, which is composed of 300 black image parts, is shown in figure 4.16. The snapshot is actually taken just before the final image is ready. The red colored square in the image is a Sunflow feature to indicate what part of the images is being rendered, which is the final one in this case. The actual image rendering result, for the scene file, was shown in figure 4.4.

The test behavior for the application management testing is realized as a Jata test case and described in Section 4.4.

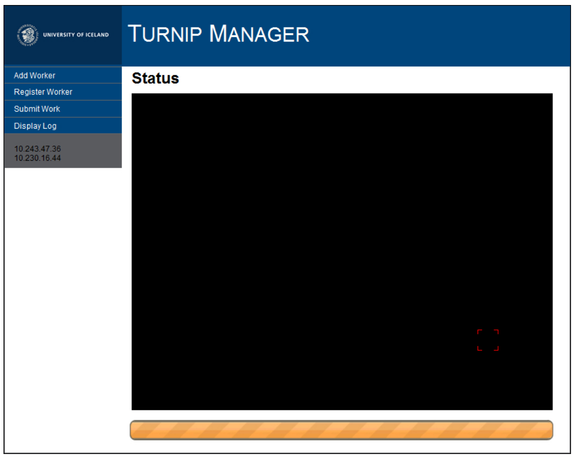


Figure 4.16: Dummy image result

4.4. Test Case Specifications

The following sections describe the Jata implementation for the case study testing technique. The sections are structured using concepts from the *Test Case Specifications* chapter of the IEEE Standard for Software and System Test Documentation [32].

4.4.1. Introduction

The system under test is the Turnip application deployed within OSGi container on an Amazon EC2 instance. The request manager component runs on the application master node. It encapsulates the application management functionality that is to be tested. The request manager delegates rendering tasks to other available nodes and when all tasks have finished correctly it composes a final image result.

The prerequisites for the test case, are that the SUT has been initialized and two worker instances (test doubles) added. The worker instances with test double implementation are started through the Turnip web interface. The scene file also needs to have been uploaded to the master node. A description of how the test environment is set up is in the *Test Procedure* section (4.5).

The test case stimulus results in the determination of 300 rendering tasks and a composition of a resulting image. The number of tasks and the resulting fake image are verified.

Objective

The testing focus is on the work of the distribution management of the application, i.e. that distributing tasks and merging their partial rendering outputs works as expected. The test checks if the management determines the tasks correctly and distributes all of them to the available cloud nodes correctly. The communication between distributed components in the cloud is under inspection.

Inputs

No parameters are needed in the request stimulation at the start of the test case. The determination of rendering tasks is based on the preloaded scene file, which is uploaded to the master node by a test bootstrapping utility when the SUT is initialized. The utility, made for this case study, is described in section 4.5. The number of workers is also decided and launched with the test utility.

4.4.2. Test Case Structure

The Java code for the test case is shown in listing 4.1.

```java
public class tc_TestTheApplicationManagement extends TestCase<Mtc,Sys>{
  private String sutIpAddress;
  private Map<Integer, MsgReceivedJob> receivedJobMessages = new
      HashMap<Integer, MsgReceivedJob>();
  private static int EXPECTED_TASK_MESSAGES_COUNT = 300;
  private static int EXPECTED_FILE_SIZE = 3247; // size of 300 black
      image parts

  public TAM_TestCase(String[] args) throws Exception {
    super(Mtc.class, Sys.class);
  }

  @Override
  protected void Case(Mtc mtc, Sys sys) throws JataException {
    Ptc ptc1 = createPtc(Ptc.class);
    Ptc ptc2 = createPtc(Ptc.class);

    PipeCenter.Map(ptc1.pt_PS, sys.pt_PTC1); // PTC-1 <-> System
    PipeCenter.Map(mtc.pt_MP1, ptc1.pt_PM);  // PTC-1 <-> MTC
    PipeCenter.Map(ptc2.pt_PS, sys.pt_PTC2); // PTC-2 <-> System
    PipeCenter.Map(mtc.pt_MP1, ptc2.pt_PM);  // PTC-2 <-> MTC
    PipeCenter.Map(mtc.pt_system, sys.pt_http); // MTC - System

    try {
      // Send stimulus
      mtc.pt_systemHttp.send(new MsgStartRendering(sutIpAddress));

      // Create and start PTC functions
      startPTC(ptc1);
      startPTC(ptc2);

      // Wait for all tasks parameters or timeout
      getReceivedJobMessages();

      // Verify
      verifyJobMessages();
      verifyResultsFileSize();
    }
    catch (JataException e) { // can represent a time-out
      mtc.Fail(); // VERDICT=FAIL
      System.out.println(e.getMessage());
      return;
    }

    mtc.Pass(); // VERDICT=PASS
  }

  private void getReceivedJobMessages() throws JataException {
    Timer jataTimer = new Timer();

    int timeOutSeconds = 20; // for all messages to arrive
    SetTimeMessage timeMsg = new SetTimeMessage(timeOutSeconds * 1000);
    jataTimer.start();

    while(true) // wait until all messages have arrived (or a timeout)
    {
      jataTimer.lock();
      Message m = jataTimer.waitHandle(timeMsg); // returns TimeOutException if time exceeds
      if(m instanceof TimeOutMessage){
        throw new JataTimerException(null, "Time out while waiting for received: " + timeOutSeconds + " seconds");
      }

      if(receivedJobMessages.size()>=EXPECTED_TASK_MESSAGES_COUNT){
        break;
      }

      getReceivedJobMessage(mtc.pt_PTC1);
      getReceivedJobMessage(mtc.pt_PTC2);
    }
  }

  /**
   * Receive incoming task (parameter) messages.
   */
  private void getReceivedJobMessage(IAltBranch port) throws
```

```
        JataException {
    Alt alt = new Alt();
    Timer t = new Timer();
    t.start();

    alt.addBranch(port, new MsgReceivedJob());
    alt.addBranch(t, new SetTimeMessage(2000));
    alt.proc();

    if (alt.getResult().index ==0){
      MsgReceivedJob msg = (MsgReceivedJob) alt.getResult().Result;

      int indexId = msg.getParameters().getBucketId()/2;
      receivedJobMessages.put(indexId, msg);
    }
    else{
      // Ignore: The test case polls the message-queue
      // regularly, so it can normally be empty at times.
    }
  }

  private void verifyJobMessages() {
    if(receivedJobMessages.size() != EXPECTED_TASK_MESSAGES_COUNT){
      throw new JataException("FAILURE: Number of received task
          messages should be: " + EXPECTED_TASK_MESSAGES_COUNT);
    }
    // else continue.. = mtc.Pass
  }

  private void verifyResultsFileSize() throws JataException{
    mtc.pt_systemHttp.send(new MsgGetImage(sutIpAddress));

    Alt alt = new Alt();
    Timer t = new Timer();
    t.start();

    alt.addBranch(mtc.pt_systemHttp, new MsgGetImageResponse());
    alt.addBranch(t, new SetTimeMessage(3000));
    alt.proc();

    AltResult r = alt.getResult();

    if (r.index == 0){
      MsgGetImageResponse msg = (MsgGetImageResponse) alt.getResult().
          Result;

      File file = (File)msg.Coder();

      if (EXPECTED_FILE_SIZE != file.length()) {
        throw new JataAssertException("Failure. Expected result file
            size: " + EXPECTED_FILE_SIZE + ". Actual size: " + file.
            length());
```

```
121         }
122         System.out.println("Result file size correct.");
123       }
124       else{
125         throw new JataAssertException("Failure in receiving image");
126       }
127     }
128 }
```

Listing 4.1: Jata test case implementation

The private variables of the test case class are used in stimulation and expectations. The structure of the test case Java method complies to the general test case structure of Jata and TTCN-3. First the PTCs are created (lines 13 and 14), and then the communication ports are mapped (lines 16-20), i.e. connected with each other according to the test architecture. In Jata only local executed ports can be mapped with the PipeCenter methods. Therefore the ports being mapped in the test case can only be between test components (MTC and PTCs) or the system adapter (SystemComponent). Remote ports cannot be mapped with Jata library methods and need to be manually configured (with RMI). After the configuration of test components and ports the stimulus is sent (line 24). Then the PTC test functions are created and started (lines 27-28).

The getReceivedJobMessages method (line 46) waits for all rendering messages to arrive before continuing the test case. The method configures the timeout for receiving all rendering job messages to be 20 seconds (line 49) before starting a loop (line 53) to collect the received job messages that the PTCs have received from the test doubles (lines 66 and 66). The getReceivedJobMessage method (line 73) is called periodically for both PTCs until another timeout occurs or all 300 messages have arrived. If the job message from a PTC is the last one according to the expected number of messages, the loop can break (line 61), and the test case continues to the final verification.

The getReceivedJobMessage method (line 73) uses an alt statement to receive messages through a PTC port of the MTC (line 78). When a PTC receives a message from the test double it forwards it to the MTC. The PTC test function behavior is described in listing 4.12. The alt statement contains a timeout definition (line 79) so that the method does not get blocked if the PTC has not sent a message within a certain time, or has finished sending all the messages. The 'outer' timeout of 20 seconds (lines 49-51) is the timeout for all the messages to arrive. The Jata Timer class is used for the timeout handle but Java Date/Time methods could also have been used. The jataTimer.start method (line 51) starts the timer. The jataTimer.lock method (line 55) stops (or locks) the time measurement, from the timer start to the point of the lock. It is therefore possible to call the lock method in every round of the loop until the defined timeout is received. The elapsed time is checked within the jataTimer.waitHandle method. The timer returns a message of TimeOutException in case of a timeout (line 56).

When a message about a task submission is received at a MTC port, it is added to the map collection of receivedJobMessages, a member variable of the MTC (line 86). The bucket ID is contained in the message (line 85), so it is possible to make sure that the same bucket information is not received twice or that all 300 bucket IDs are received. (The division by 2 in line 85 is a way to deal with a 'bug' in the SUT, which creates only even number bucket IDs).

The verify method calls (lines 34 and 35) make two different assertions for the test case. The verifyJobMessages method (line 94) verifies that indeed 300 tasks were used to render the image. When the test case has received expected number of task messages (line 95), defined with the EXPECTED_TASK_MESSAGES_COUNT constant, the test case can continue. If all messages do not arrive in the defined time, the test case ends with a failure. The verifyResultsFileSize (line 101) verifies the disk file size of the completed fake image. It uses the HTTP port to fetch the image as a Java File object for verification of its size. The method uses an alt statement as the structure. The total size of 300 black image parts is exactly 3247 bytes (line 119), defined with the EXPECTED_FILE_SIZE constant. The HTTP port returns a MsgGetImageResponse message result which contains the image coded as a file object (line 117).

Test Components and the System Adapter

Following listings (4.2, 4.3, 4.4) show the structure of the test components (MTC and PTCs) and the system adapter of the test case. The relationship between the components and communication ports is depicted in figure 4.12.

```java
public class Mtc extends jata.Component.TestComponent {
  public PortMP pt_PTC1; // port to PTC-1
  public PortMP pt_PTC2; // port to PTC-2
  public PortSystem pt_system; // mapped to system HTTP port

  public Mtc() throws Exception{
    super();
    pt_PTC1 = new PortMP();
    pt_PTC2 = new PortMP();
    pt_system = new PortSystem();
  }
}
```

Listing 4.2: Main Test Component

The MTC is shown in listing 4.2. It has an abstract output port (pt_system) which is mapped to the system HTTP port for sending stimulation message to the SUT via HTTP. It has also input ports (pt_PTC1 and pt_PTC1) for the bucket information messages received from the PTCs.

```java
public class Ptc extends TestComponent {
  public PortPS pt_system; // port to system (adapter)
  public PortPM pt_mtc; // port to mtc

  public Ptc() throws Exception{
    super();
    pt_system = new PortPS();
    pt_mtc = new PortPM();
  }
}
```

Listing 4.3: Parallel Test Component

Listing 4.3 shows the PTC definition which includes the port definitions to the MTC (pt_mtc) and the system adapter (pt_system).

```java
public class Sys extends SystemComponent{
  public PortSP pt_PTC1;
  public PortSP pt_PTC2;
  public PortSystemHttp pt_http;

  public Sys() throws Exception{
    pt_PTC1 = new PortSP(Config.getRmiConfig("PTC-1"));
    pt_PTC2 = new PortSP(Config.getRmiConfig("PTC-2"));
    pt_http = new PortSystemHttp();
  }
}
```

Listing 4.4: System Component/Adapter

The SystemComponent, in listing 4.4, includes definitions for three ports. One (outgoing) system port for HTTP stimulation of the SUT (pt_http), and two (incoming) RMI system ports mapped to each of the defined PTC (pt_PTC1 and pt_PTC2). The RMI system ports support RMI functionality to receive messages from the SUT. The Config class makes sure that each system port is initialized with a distinct RMI port number.

Messages

The Message object plays a vital role in a Jata test architecture. The communication ports use message objects as a means of data transfer. The Jata message includes methods for data encoding and decoding. Listing 4.5 shows the message for 'start rendering' stimulation. The message includes the IP address of the SUT (line 2) so that the HTTP system port can construct the correct HTTP request (the HTTP port is described in listing 4.9). The message only contains a coding method (line 19) and not decoding because it is only used to stimulate the SUT and not to receive a response.

```
 1 public class MsgStartRendering extends Message{
 2   private String sutIpAddress;
 3
 4   public MsgStartRendering(){
 5     this((Message)null);
 6   }
 7
 8   public MsgStartRendering(String sutIpAddress){
 9     this((Message)null);
10     this.sutIpAddress = sutIpAddress;
11   }
12
13   public MsgStartRendering(Message parent) {
14     super(parent);
15     codeToType = "java.lang.String";
16   }
17
18   @Override
19   protected Object Coder() throws JataException {
20     return sutIpAddress;
21   }
22 }
```

Listing 4.5: Message 'Start rendering'

Listing 4.6 shows the response message created by the HTTP port after the start rendering message has been sent to the port and handled. This particular message could be analyzed in the test case method but is not. Instead the decoding method (line 13) throws an assert exception if the HTML string from the stimulation is incorrect (line 19). In principle the test case method should include an alt statement and perform this verification instead of the message, but it was not done for the sake of simplicity.

```
1 public class MsgSutResponse extends Message{
2
3   public MsgSutResponse(Message parent) {
```

```java
    super(parent);
    decodeFromType = "java.lang.String";
  }

  public MsgSutResponse() {
    this(null);
  }

  @Override
  protected boolean Decoder(Object stream) throws Exception {
    String content = (String)stream;

    // "Request submited!" string is the result from the Turnip ajaxServlet after starting a rendering job.
    if (!"Request submited!".equalsIgnoreCase(content)){
      throw new JataAssertException("Unexptected stimulation result");
    }
  }
}
```

Listing 4.6: Message 'Stimulation response'

The MsgReceivedJob, shown in listing 4.7, is used for the information about rendering jobs. It includes a JobParameters object (line 2), which holds the actual information. The definition of the JobParameter class is included in the listing (line 39). The JobParameter class is simply a wrapper for the bucket information.

```java
 1 public class MsgReceivedJob extends Message implements Serializable{
 2   private JobParameters jobParameters = null;
 3
 4   public MsgReceivedJob(Message parent) {
 5     super(parent);
 6     codeToType = "is.hi.jata.example.parametertestcase.turnip.JobParameters";
 7     decodeFromType = "is.hi.jata.example.parametertestcase.turnip.JobParameters";
 8   }
 9
10   public MsgReceivedJob(){
11     this((Message)null);
12   }
13
14   public MsgReceivedJob(JobParameters jobParameters) {
15     this((Message)null);
16     this.jobParameters = jobParameters;
17   }
18
19   public JobParameters getParameters(){
20     return jobParameters;
21   }
22
```

```
  @Override
  protected Object Coder() throws JataException {
    return jobParameters; // no coding required. simply using the POJO
  }

  @Override
  protected boolean Decoder(Object stream) throws Exception {
    if (stream instanceof JobParameters){
      this.jobParameters = (JobParameters) stream;
      return true;
    }
    else
      return false;
  }
}

public class JobParameters implements Serializable{

  private int workerId;
  private int bucketId;
  private int frame;

  ... getters and setters omitted from the listing
}
```

Listing 4.7: Message 'Received job'

The last message described here is the MsgGetImageResponse, which the HTTP system port sends as a response when the MTC fetches the image result. The message encapsulates a Java File object which contains the composed image.

```
public class MsgGetImageResponse extends Message implements
    Serializable {

  private File file = null;

  public MsgGetImageResponse(Message Father) {
    super(Father);
    codeToType = "java.io.File";
    decodeFromType = "java.io.File";
  }

  public MsgGetImageResponse() {
    this(null);
  }

  @Override
  protected Object Coder() throws JataException {
    return file;
  }

```

```java
  @Override
  protected boolean Decoder(Object stream) throws Exception {
    if (stream instanceof File)
    {
      this.file = (File) stream;
      return true;
    }
    else
      return false;
  }
}
```

Listing 4.8: Message 'Get image response'

Communication Ports

In this section the source code for the 'main' ports of the test case is shown. Not all port source code is listed because of similarities or simplicity. Listing 4.9 shows the code for the HTTP system port used in stimulation of the SUT. In figure 4.12 this port is named pt_http within the SUT adapter (Jata SystemComponent).

```java
public class PortSystemHttp extends SystemPort {

  public PortSystemHttp() throws JataException{
    super();
    addInputType(Message.getICD(MsgStartRendering.class));
    addOutputType(Message.getICD(MsgSutResponse.class));
    addInputType(Message.getICD(MsgGetImage.class));
    addOutputType(Message.getICD(MsgGetImageResponse.class));
    this.PortMode = SystemPort.AdapterMode;
  }

  @Override
  protected AdapterPackage Adapter(AdapterPackage pkg)
                                              throws Exception {
    if (pkg.MessageType.equals(MsgStartRendering.class.getName())) {
      String url = (String) pkg.stream;
      return new AdapterPackage(submitWork(url));
    } else if (pkg.MessageType.equals(MsgGetImage.class.getName())) {
      String url = (String) pkg.stream;
      return new AdapterPackage(getImage(url));
    }
    return null;
  }

  private String submitWork(String sutIpAddress) throws Exception{

    final WebClient webClient = new WebClient();
28
29      String ajaxMethod = "makeRequest";
30      String ajaxUrl = "http://" + sutIpAddress + ":9000/turnip/ajax/" +
           method;
31
32      final HtmlPage page = webClient.getPage(ajaxUrl);
33      final String pageAsText = page.asText();
34      assertTrue(pageAsText.contains("work submitted"));
35      webClient.closeAllWindows();
36
37      return pageAsText;
38    }
39
40    private File getImage(String sutIpAddress) throws Exception {
41      String fileUrl = "http://" + sutIpAddress + ":9000/images/
           sunflowImage.png";
42
43      HttpClient client = new DefaultHttpClient();
44      HttpGet httpget = new HttpGet(fileUrl);
45      HttpContext context = new BasicHttpContext();
46      HttpResponse getResponse = client.execute(httpget, context);
47
48      String responsestatus = getResponse.getStatusLine().toString();
49      if (!STATUS_OK.equals(responsestatus)) {
50        throw new Exception("Result file statusline failed.");
51      }
52
53      URL url = new URL(fileUrl);
54      BufferedImage image = ImageIO.read(url);
55
56      String workingDir = System.getProperty("user.dir");
57
58      File file = new File(workingDir + "/result.png");
59      ImageIO.write(image, "png", file);
60
61      return file;
62    }
63  }
```

Listing 4.9: HTTP system port

All ports define their *input* and *output* message types in the constructor (lines 5 to 8). The input/output message types are used from the owner component perspective, i.e. input type message is a stimulation message. The message is sent 'in' to the port, which then uses it for outgoing calls to the SUT. Output type message is a message which goes 'out' from the port, i.e. response message after the port is called via the 'send' method. For instance if an MTC port and a system port are mapped together the MTC port sends an 'input' type message to the system port, which returns the call with an 'output' type message. The system HTTP port has an input message type defined as 'start rendering' (line 5) with a corresponding output type as 'SUT response' (line 6). For the final image downloading, the port contains a 'get image' input type (line 7) and a corresponding 'get image response' output message (line 8).

System ports override the Adapter method (line 11) which is called by the Jata framework when messages are sent to the ports. The Adapter method is overridden to create the appropriate stimulation request for the SUT. There are two ways of sending messages with the port and the message type is used to distinquish between the options (line 15 and 18). The IP address of the SUT web server is used to create the correct HTML request for either request.

For the 'start rendering' stimulation in method submitWork (line 25), the port uses *HtmlUnit*[7], a Java HTML test library, for communication with the SUT web application. The Turnip web application is implemented with AJAX so that requests, such as starting a rendering job, can be made directly with URL requests. Using HtmlUnit also gives the opportunity of making assertions on the returned HTML content (e.g. line 27). The URL for submitting rendering job in Turnip looks like:

`http://APPLICATION-HOST:9000/turnip/ajax/makeRequest`.

For the 'get image' feature of the port, defined in method getImage (line 40), the port uses Apache *HttpClient*[8], because HtmlUnit does not support image downloading. The communication is set up with HttpClient (lines 43 to 46) and the ImageIO class used to download the image (line 54). The image is written to the test system file directory also with the help of ImageIO (line 59). The reason why it is written to a disk is that then it can be encapsulated as a Java File object for file size verification (line 61). The request URL for the image is:

`http://APPLICATION-HOST:9000/images/sunflowImage.png`.

Abstract Ports

Abstract ports are used to link test components together or to link test components with the system adapter. Their definition is generally simple and does not need any extra coding as the system ports. Listing 4.10 shows an abstract port that does not implement any behavior, only input and output message definitions. The PortPS is owned by an PTC and maps to a system port. PTCs receive job task submission messages with information about tasks (buckets) through this port. The defined input type message is 'received job' (line 4). Other abstract ports are just as simple

[7]HtmlUnit. http://htmlunit.sourceforge.net/
[8]Apache HttpClient. http://hc.apache.org/httpcomponents-client-ga/index.html/

and do not need to be listed here. The seven test components' ports seen in the test architecture diagram in figure 4.12, are all abstract ports (PortPS is used to implement the ones identified as pt_sys within the PTCs in the architecture diagram).

```java
public class PortPS extends AbstractPort{
  public PortPS() throws JataException {
    super();
    addInputType(Message.getICD(MsgReceivedJob.class));
  }
}
```

Listing 4.10: PTC's abstract port

Remote Port (Within the Test Double Worker)

The remote port is owned by the test double implementation of the worker. Listing 4.11 shows the code for the whole test double. The test double architecture and behavior was described in Section 4.3.3. The test double extends the actual SUT worker component (line 1) and is deployed on the worker instance in the cloud. The test double only overrides the run method (line 11) of its parent, which is called by the *Request Manager*.

The run method starts by setting up the communication port to the test system adapter, in a call to the configJata method (line 12). As stated earlier remote ports need a manual configuration between the ports involved. Information about the system adapter port is uploaded in a configuration file to the worker cloud instance when it is started. The initialization procedure, described in section 4.5, takes care of uploading the configuration information.

After configuration of the test double a new thread is started for handling rendering tasks (lines 14 to 19). The render method (line 34) asks for next bucket to render (line 37) and sends information about the task to the test system (line 38), through the remote port. The PortWorkerToPtc (declared in line 6) is an RMI port for sending messages via RMI to a Jata system adapter port (line 47). The receiving system adapter port is mapped to an abstract port of a PTC (see figure 4.12).

```java
1 public class WorkerComponentTestDouble extends WorkerComponent {
2   /**
3   * Jata remote port definition
4   */
5   private PortWorkerToPtc portWorkerToPtc = null;
6
7   /**
8   * Starts the rendering (called by the request manager)
```

```
  */
  @Override
  public void run()  {
    configJata();

    renderThread  = new Thread(new Runnable() {
      public void run() {
        render();
      }
    });
    renderThread.start();
  }

  private void configJata() throws Exception {
    FakeRenderWorkerConfiguration config = new FakeRenderWorkerConfiguration();
    config.workerId = Config.getWorkerId();
    config.ptcRmiConfiguration = Config.getPtcConfig();

    try {
      portWorkerToPtc = new PortWorkerToPtc(config.ptcRmiConfiguration);
    } catch (Exception e) {
      System.out.println("WARNING: Could not create PortToMtc");
    }
  }

  private void render() {
    while (!doBreak()) {
      try {
        currentBucket = requestManager.getNextBucket(currentFrame, workerId);
        sendParametersToJata();
      } catch (EmptyStackException e) {
        break; // stop rendering
      }
    }
  }

  private void sendParametersToJata() {
    try {
      portWorkerToPtc.pipe2Me(new MsgReceivedJob(new JobParameters(workerId, currentBucket.id, currentFrame)));
    } catch (JataPortException e) {
      System.out.println("Render worker-" + workerId + " failed to send message to test system: " + e.getMessage());
    }
  }
}
```

Listing 4.11: Worker's Rendering Test Double

PTC test function behavior

This section describes the behavior of the PTC test function. The function source code is in listing 4.12.

```java
public class ParameterPtcTestFunc extends TestFunction<Ptc>{

  @Override
  public void Func(Ptc ptc) throws JataException{
    getReceivedJobMessages(ptc);
    ptc.Pass();
  }

  private void getReceivedJobMessages(Ptc ptc) throws JataException{
    Timer jataTimer = new Timer();

    int timeOutSeconds = 120;
    SetTimeMessage timeMsg = new SetTimeMessage(timeOutSeconds * 1000);
    jataTimer.start();

    while(true){
      jataTimer.lock();
      Message m = jataTimer.waitHandle(timeMsg);
      if(m instanceof TimeOutMessage){
        // Time out exceptions are caught in the MTC, which has another
            loop for retrieving these messages.
        break;
      }
      receiveMessage(ptc);
    }
  }

  /**
   * Wait for incoming task (parameter) messages from
   * the test-double worker remote-port.
   */
  private void receiveMessage(Ptc ptc) throws JataException{
    Alt alt = new Alt();
    Timer t = new Timer();
    t.start();

    alt.addBranch(ptc.pt_system, new MsgReceivedJob());
    alt.addBranch(t, new SetTimeMessage(2000));
    alt.proc();

    if (alt.getResult().index == 0){
      MsgReceivedJob msg = (MsgReceivedJob) alt.getResult().Result;

      // "Forward" PTC's incoming parameter message to the MTC
      ptc.pt_MTC.send(msg);

46          // Ideally send a result image-part to the SUT.
47          // ptc.portPS_out.send(new MsgImagePart()); -> Not 'possible'
48        }
49        else{
50          // Ignore: The test function polls the message-queue
51          // regularly, so it can normally be empty at times.
52        }
53    }
54  }
```

Listing 4.12: PTC test function

The purpose of this PTC test function is to retrieve the messages that the parallel test component receives from the SUT, and forward them to the MTC for analysis and verification.

The PTC function is started in the main test case method (listing 4.1 lines 26 and 27). In this test case the MTC is responsible for collecting all bucket information messages for verification, and with this implementation, the MTC has all the messages and knowledge needed for validation of all the messages.

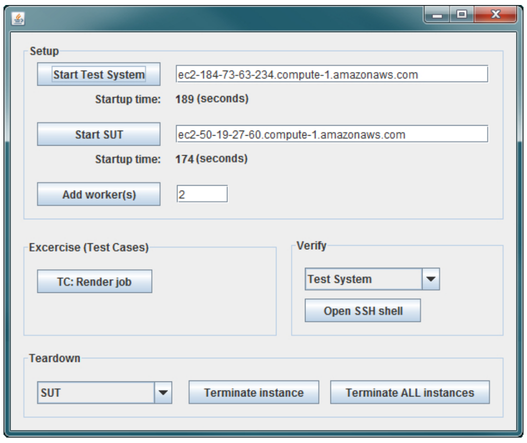
Just like the MTC (listing 4.1 line 46) the PTC function class defines a getReceivedJobMessages method (line 9). The reason is that the worker test double remote port sends the initial message to a system adapter port which forwards it to the PTC. When the PTC receives such a message it instantly forwards it to the MTC through its abstract port which is mapped to a MTC port.

In the getReceivedJobMessages method a timer is started (lines 12 to 14) before entering a loop which receives all incoming messages within the defined timeout. The timeout is made large enough for all rendering tasks to finish. The MTC is responsible for defining and controlling the timeout of the whole test case. In the loop a call to the receiveMessage method (line 23) is made to receive each message from the system port. The receiveMessage method includes an alt statement which is made in the usual way (lines 36 to 38). For each MsgReceivedJob message received (line 40) the message is forwarded to the MTC (line 44). Line 47 shows where image parts should in principle be created and sent to the SUT if it were possible. The test double workaround for the dummy image creation and sending is described in Section 4.3.3.

4.5. Test Procedure

This section is structured using concepts from the *Test Procedure* chapter of the IEEE Standard for Software and System Test Documentation [32].

There are a few issues to be considered to be made for the execution of the test case, such as security and runtime configuration because of the elastic nature of the cloud. Therefore a simple test harness utility was implemented to ease the setup and execution of test cases. The test harness utility only helps with setting up the test bed and start the test case. It is not possible to view the results from the utility. Jata only includes a command line test runner (similar to the JUnit text UI runner), so results must be verified through the shell of the cloud instance that runs the test system. Figures 4.17 and 4.18 show screen shots of the test engineer's environment consisting of the special test harness utility and command line shell for reading the output.



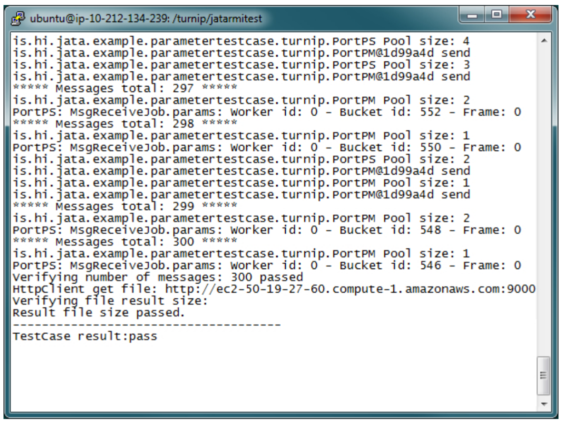
Setup
Start Test System: ec2-184-73-63-234.compute-1.amazonaws.com
Startup time: 189 (seconds)
Start SUT: ec2-50-19-27-60.compute-1.amazonaws.com
Startup time: 174 (seconds)
Add worker(s): 2

Excercise (Test Cases)
TC: Render job

Verify
Test System
Open SSH shell

Teardown
SUT
Terminate instance
Terminate ALL instances

Figure 4.17: Test harness utility



```
ubuntu@ip-10-212-134-239: /turnip/jatarmitest
is.hi.jata.example.parametertestcase.turnip.PortPS Pool size: 4
is.hi.jata.example.parametertestcase.turnip.PortPM@1d99a4d send
is.hi.jata.example.parametertestcase.turnip.PortPS Pool size: 3
is.hi.jata.example.parametertestcase.turnip.PortPM@1d99a4d send
***** Messages total: 297 *****
is.hi.jata.example.parametertestcase.turnip.PortPM Pool size: 2
PortPS: MsgReceiveJob.params: Worker id: 0 - Bucket id: 552 - Frame: 0
***** Messages total: 298 *****
is.hi.jata.example.parametertestcase.turnip.PortPM Pool size: 1
PortPS: MsgReceiveJob.params: Worker id: 0 - Bucket id: 550 - Frame: 0
is.hi.jata.example.parametertestcase.turnip.PortPS Pool size: 2
is.hi.jata.example.parametertestcase.turnip.PortPM@1d99a4d send
is.hi.jata.example.parametertestcase.turnip.PortPM Pool size: 1
is.hi.jata.example.parametertestcase.turnip.PortPM@1d99a4d send
***** Messages total: 299 *****
is.hi.jata.example.parametertestcase.turnip.PortPM Pool size: 2
PortPS: MsgReceiveJob.params: Worker id: 0 - Bucket id: 548 - Frame: 0
***** Messages total: 300 *****
is.hi.jata.example.parametertestcase.turnip.PortPM Pool size: 1
PortPS: MsgReceiveJob.params: Worker id: 0 - Bucket id: 546 - Frame: 0
Verifying number of messages: 300 passed
HttpClient get file: http://ec2-50-19-27-60.compute-1.amazonaws.com:9000
Verifying file result size:
Result file size passed.
--------------------------------------
TestCase result:pass
```

Figure 4.18: Test case result view (log)

The first thing that is done is starting the test system machine, which contains the Jata test system. The instance is started with the help of Java code that uses the Amazon AWS SDK for launching the instance and *JShc*[9] (*Java Secure Channel*) for uploading the required Jata binaries instance. JSch is a Java implementation of the SSH (*Secure Shell*) network protocol for data exchange.

An important thing to mention is that for Amazon EC2 security, i.e. launching and managing instances within the cloud, the Amazon EC2 account credentials need to be available to the startup process. This is done by including a authentication file with the account information where the test utility can locate it.

The next step is starting the master Turnip machine. The test utility uses the Cloud-ASL provided with the Turnip application for the startup and initialization of the application. The SUT is initialized from the utility. The scene file used for the rendering needs to be uploaded to the SUT master node. JSch is used for the uploading.

From within the utility the SUT workers (cloud instances) to be used in the test

[9]Java Secure Channel (JSch). www.jcraft.com/jsch

case are added. The worker machines themselves are simply added through the web interface (AJAX) of the Turnip application master node – previously started. The worker OSGi service has been pre-configured to use the test double, described in 4.3.3 instead of a real rendering implementation. An important step needs to be made by the test harness utility at this stage. The step involves uploading a configuration file about the test system machine for the remote port of the test double to use in their RMI configuration. The remote ports communicate with the Jata system adapter of the test system and need to know the IP address and the RMI port to connect to. Because of the elastic nature of clouds this information cannot be provided at design-time and needs to be done at run-time.

After the four cloud instances are ready (the test system, Turnip master node and two worker instances), the test case can be started. The test harness utility uses SSH for starting the Jata test case on the test system. The output is redirected to an output file on the test system and also to the *System.out* console of the test harness utility Java program – because of SSH features. The test results can therefore either be viewed on the test machine or in the utility program's *System.out* console.

Ordered Description

Table 4.3 lists the test procedure for the most important activities in an ordered description, as introduced in the IEEE test procedure documentation. The table includes columns for the activity type and the actions/steps needed for the activity. The right-most column 'Utility' indicates whether the activity can be controlled from within the custom made test harness utility mentioned above.

Test Case Timing

Table 4.4 shows the mean time it takes to execute each activity in the testing. The launching and setup of systems takes approximately six minutes, where each cloud instance takes about three minutes to launch and become initialized for the test. The workers are started in parallel, which takes around two minutes. The test case itself runs in approximately 24 seconds, and 32 seconds go into tearing down the systems, adding up to a total of approximately nine minutes for executing the whole test. The test bed startup should be optimized in real case scenarios by adding parallel support for instances startup. In this case the workers need to be started after the master node but the test system machine could have been started in parallel with the SUT master. Individual machines should have support for restart and reinitialize. If for instance a tester needs to change a test case, the only thing needed is uploading the new test case binaries to the test system and start the test case. There would be no need to restart instances of the SUT. The same applies if code or configuraiton fixes need to by applied to nodes of the SUT.

Type	Actions/Steps	Utility
Log:	Java *System.out* statements in the system console of the test system (and the test harness utility via SSH).	Yes
Setup:	Amazon EC2 authentication file needs to be in place. The test system needs to be started. Then the SUT is started and initialized and the scene file is automatically uploaded to the SUT master node if started from the utility. Two workers need to be added.	Yes
Start:	The Jata test case is started from the test utility.	Yes
Proceed:	No actions needed during the execution of the test case	N/A
Measurement:	Test measurements are made within the Jata test case.	N/A
Shut down:	It is not possible to temporarily suspend testing, when unscheduled events dictate.	N/A
Restart:	The Jata test case can be restarted without initializing the SUT again.	Yes
Stop:	It is not possible to stop the test case run	N/A
Wrap-up:	Cloud instances need to be terminated after the test run	Yes
Contingencies:	No actions are available to deal with anomalies that may occur during execution.	N/A

Table 4.3: Ordered activities description

Activity	**Exec. Time (seconds)**
Test system startup	*Total 167 s*
- EC2 instance startup	52 s
- Java jars upload	115 s
Start Turnip master	*Total 180 s*
- EC2 instance startup	52 s
- Upload scene file	21 s
- Start Felix runtime	35 s
- Install OSGi bundles	54 s
- Start OSGi bundles	18 s
Add two workers (in parallel)	*Total 132 s*
- EC2 instance startup	52 s
- Start Felix runtime	35 s
- Install OSGi bundles	36 s
- Start OSGi bundles	9 s
Run Jata test case	*Total 24 s*
Terminate all instances	*Total 32 s*
Total test execution time:	**535 s**

Table 4.4: Test activities execution times

5. Evaluation

This chapter covers an evaluation of the performed case study. The objectives are twofold: Investigating challenges in software testing for applications running in cloud environment and evaluation of the applicability of Jata for distributed testing. In section 5.1 the evaluation of software testing of cloud application is made whereas section 5.2 covers an evaluation of the Jata test framework.

5.1. Software Testing of Cloud Applications

The main result revealed by the case study is that software testing for cloud applications is possible with traditional testing methods. However testing within a cloud environment includes certain issues such as:

- Run-time configuration required
- MAC address license issues
- Firewall issues

The elastic nature of the cloud demands that the test harness (i.e. a test setup controller) has knowledge of all the machines, i.e. their IP addresses, participating in the test bed. For configuring distributed test components such as PTCs this 'knowledge' needs to be automated by a test harness. In traditional test beds the IP addresses of participants can be configured at design-time in a configuration file or a database. Design-time configuration of distributed cloud test components is however not possible and remote port mapping configuration therefore needs to be provided at run-time. This puts an extra burden on the test harness but can be solved in various ways. The case study in this thesis used a test harness utility for creating the cloud instances to be used and uploaded configuration files about the test case to the test instances. The configuration file included RMI port information for the remote port of the test double running on the instance.

Another concern related to the elastic behavior of instances is licensing issues. Many test system providers, such as TTCN-3 tool providers, use MAC addresses licensing schemes for their products. Licensing based on MAC addresses is problematic for the cloud and needs to be addressed by the software vendors.

Even if cloud vendors provide good firewall configuration support it is recommended, for simplicity, that all participating machines in a test bed be located within the cloud. In this case study, running the test system locally and the SUT in the cloud using RMI as communication method between them was not possible in a simple manner due to firewall configuration issues between the sites. Another argument in favor of including all participants within the cloud is improved performance of test case execution, because of the geographical location of involved instances. Mixing private and public cloud infrastructures in a test bed can however be a good option for security testing.

5.2. Jata

Jata's main benefits can be summarized in the following list:

- Light-weight
- Free of charge
- Common vocabulary (by using TTCN-3 concepts)
- Systematic way of defining test cases (by using TTCN-3 concepts)
- Java library

Jata is a light-weight Java library. Much like JUnit the only thing needed to get up and running for writing Jata test cases is including the Jata jar file in the Java class-path. The framework is targeted at testing where the test cases and the communication with the test system interface is written in Java. Jata benefits from using Java since Java is a mature programming language, extremely powerful and yet simple. Another benefit is that the test cases can be compiled and built together with the components under test, assuming they are also developed with Java, which makes any changes between the test system and the SUT easier to handle. Conversion between types (Codecs) is much simpler, or not even needed, if both the SUT and the test system are written in the same language. Furthermore Java developers do not need to learn a new language or syntax for the testing specifications.

What is good about Jata is that it borrows structuring from TTCN-3, essentially from CTMF, in designing test cases. By this Jata forces test engineers to think abstractly and express the software architecture by organizing the code into test components (MTC and PTCs), ports, and alternatives. By using Jata a groundwork is therefore laid for a common vocabulary as well as a systematic way of creating test specifications. TTCN-3 is widely known and used by many professionals in various fields. Documentation and information for TTCN-3 concepts are easily available but the TTCN-3 language and concepts can have some learning curve. It is however necessary for Jata users to know at least the basic concepts of TTCN-3.

Jata is a newly introduced technology so it does not come without limitations. Most notably it needs support for distribution of test components. Using Java lowers the limitations of the framework, because theoretically the only limitations in implementing test cases are the limitation of Java. Workarounds, as the test double solution described in Section 4.3.3, can always be made. However additional encapsulation of concepts such as support for definition of distributed test components and mapping of remote ports would improve the library base and possibly encourage test engineers to use the framework.

Being a new framework little documentation is available. The framework is relatively simple, but not so that documentation is not needed. Currently the only available documentation is the introductory paper by the authors [60]. There are code examples provided with the framework download package[1]. The code examples are simple and do not deal with distributed test components, which also could use improvements. Actually Jata does not support distribution of the test components themselves. However, distributed ports may be used. The case study in this thesis uses therefore a test double on the remote port instance.

Jata also needs to be improved to generate associated test reports, instead of relying solely on a textual test runner. In the first run this might be achieved by using a log utility such as Apache log4j[2] to write the test output to a file on the test system. Result maintainability is an important aspect of testing activities. By writing the output to a file the different test runs can be compared. In later stages it might be feasible to write test outputs to XML files, as JUnit is capable of, for build servers to archive and use for graphical displaying of the test results.

A minor issue with Jata, though worth noting, is that it fails in a few places on adhering to the common Java naming convention, which can be confusing. All method names should start with a lowercase letter. Most notably are the Verdict class methods ('Pass', 'Fail', 'Error' etc.), the 'UnMap' method of the IPipe interface, 'Result' method of the AltResult class, and the 'Adapter' method of the SystemPort class.

[1]Jata download: http://code.google.com/p/jata4test/
[2]Apache Logging Services. http://logging.apache.org/

Jata improvements can be summarized as follows:

- Add support for test components distribution
- Add support for remote port mapping
- Improve documentation
- Add test reporting features
- Refactor method names

While Jata can be seen as some sort of replacement for TTCN-3, some TTCN-3 concepts can only be used in an awkward way. For example, the convenient syntax of a TTCN-3 *alt* statement cannot be used in Jata (Java). From that point of view, Jata test cases look rather like the code that is typically generated by TTCN-3 compilers when translating TTCN-3 into some target language, e.g. Java.

6. Conclusion

This chapter summarizes the results and provides an outlook. The results are reviewed and discussed in section 6.1, whereas the outlook is discussed in section 6.2.

6.1. Summary

In this thesis a test case was implemented with the Jata test framework for testing of a cloud application. The cloud system under test and its management were analyzed in order to implement an advanced test case for a parallel computing application. The work revealed some challenges in testing of cloud applications. Software testing of cloud application needs to deal with run-time configuration of test-beds based on the elastic nature of clouds. Other issues deal with firewall configuration and tool licensing. A test harness utility was created in the thesis to support run-time configurations of distributed test items.

The case study also showed that Jata is a promising test framework for distributed testing. For the purpose of this thesis, Jata can be considered applicable for implementation of black-box functional test cases for cloud applications. The software testing performed with Jata contributes to assuring the quality of applications within cloud computing environment. It could be considered a weakness that Jata has only been used by the authors in their introductory paper [60], however every child needs to take its first steps somewhere. This thesis is the second step in keeping it on its feet.

The importance of software testing can never be stressed enough. Recently Amazon AWS experienced a major outage in one of its region service[1]. Amazon EC2 uses multiple geographic regions to make their services more reliable. Amazon EC2 is currently available in five regions: US East (Northern Virginia), US West (Northern California), EU (Ireland), Asia Pacific (Singapore), and a second Asia Pacific (Tokyo). Each region contains multiple availability zones claimed to be designed to be insulated from failures in other availability zones[2]. The outage was caused

[1]http://news.cnet.com/8301-30685_3-20056029-264.html
[2]http://aws.amazon.com/ec2/

by a network event which affected multiple availability zones in the US-East region, breaking the promise on the failure handling for AWS availability zones. The outage brought down multiple public websites using Amazon's service. The outage failure at Amazon is a reminder that both cloud customers and cloud vendors need to be prepared for an outage. Distributed testing techniques help to explore and prepare for the various types of failures that can occur within cloud infrastructures.

6.2. Outlook

This work on software testing cloud applications can be used to support further research in the area. Improvements in the Jata framework can be, and should be, made. Support for distributed parallel testing components is needed in Jata as opposed to running all test components in the local context of the test system. Remote port mapping between distributed test components needs improvements with e.g. library configuration methods for mapping ports. Further suggestions on Jata improvements were made in the evaluation chapter (chapter 5). Jata is a light-weight alternative to TTCN-3 tools. Its strengths lie in its simplicity and Jata should continue focusing on being a light-weight option for distributed testing.

Besides Jata improvements various new research fields, resulting from the establishment of Jata, can be created. An interesting research topic is to create a *TTCN-3 to Jata* converter, where abstract test cases could be defined with TTCN-3 and Java code using Jata objects would be generated automatically.

Furthermore, the test case defined and implemented in this thesis can be used to run the same tests on Eucalyptus which uses the same API as Amazon EC2. For other cloud platforms the application under test used in this case study needs however to be adapted. The test case in the thesis focused on the functionality of the application management. Additional areas to cover could include testing quality attributes of the cloud middleware itself, with focus on performance, reliability, and security. Stress testing of components such as the application management could furthermore be researched using Jata.

Finally, testing can be done where private and public cloud infrastructures are mixed. In this case, the test components need be distributed in a way that some test components run within a certain cloud infrastructure and others out of it. The focus should be on test components communication. Remote ports in Jata can be used for such a test architecture.

Acronyms

AJAX	Asynchronous JavaScript and XML
AMI	Amazon Machine Instance
AP	Abstract Port (Jata)
API	Application Programming Interface
ASL	Architectural Scripting Language
AWS	Amazon Web Services
CTMF	Conformance Testing Methodology and Framework
EC2	Elastic Compute Cloud (Amazon)
ETSI	European Telecommunications Standards Institute
FIFO	First In First Out
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
IUT	Implementation Under Test
LT	Lower Tester
MTC	Main Test Component
OS	Operating System
OSGi	Open Services Gateway Initiative
OSI	Open Systems Interconnection
PaaS	Platform as a Service

PCO	Point of Control and Observation
POJO	Plain Old Java Object
PTC	Parallel Test Component
RMI	Remote Method Invocation
R-OSGi	Remote OSGi
SaaS	Software as a Service
SDK	Software Development Kit
SP	System Port (Jata)
SSH	Secure Shell
SUT	System Under Test
STITC	Software Testing in the Cloud
TC	Test Component
TCP	Test Coordination Procedures
TP	Timer Port (Jata)
TTCN-3	Testing and Test Control Notation version 3
UML	Unified Modeling Language
UT	Upper Tester

Bibliography

[1] ETSI European Standard (ES) 201 873-1 V4.2.1 (2010). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), 2010.

[2] L.v.d. Aalst. Software Testing as a Service (STaaS). In *PNSQC 2008. 26th Annual Pacific Northwest Software Quality Conference*. PNSQC, 2008.

[3] OSGi Alliance. *OSGi Service Platform, Release 3*. IOS Press, Inc., 2003.

[4] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings Pub Co, 1994.

[5] Amazon. Amazon Elastic Compute Cloud. Getting Started Guide. `http://docs.amazonwebservices.com/AWSEC2/latest/GettingStartedGuide/`, 2011.

[6] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[7] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, et al. Above the Clouds: A Berkeley View of Cloud Computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[8] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato. D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 631–636. IEEE Computer Society, 2010.

[9] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[10] K. Beck and E. Gamma. JUnit Test Infected: Programmers Love Writing Tests. *Java Report, July 1998, Volume 3, Number 7*, 1998.

[11] A. Belapurkar, A. Chakrabarti, H. Ponnapalli, N. Varadarajan, S. Padmanabhuni, and S. Sundarrajan. *Distributed Systems Security.* Wiley, 2009.

[12] B.W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *Software, IEEE*, 1(1):75–88, 1984.

[13] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys 2011, ACM SIGOPS European Conference on Computer Systems*, 2011.

[14] I. Burnstein. *Practical Software Testing: a Process-Oriented Approach.* Springer Verlag, 2003.

[15] I. Burnstein, T. Suwanassart, and R. Carlson. Developing a Testing Maturity Model for software test process evaluation and improvement. In *Test Conference, 1996. Proceedings., International*, pages 581–589, Oct. 1996.

[16] G. Candea, S. Bucur, and C. Zamfir. Automated Software Testing as a Service. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 155–160. ACM, 2010.

[17] WK Chan, L. Mei, and Z. Zhang. Modeling and Testing of Cloud Applications. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pages 111–118. IEEE, 2009.

[18] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A Software Testing Service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[19] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley Longman, 2005.

[20] Å. Edlund, M. Koopmans, Z.A. Shah, I. Livenson, F. Orellana, J. Kommeri, M. Tuisku, P. Lehtovuori, K.M. Hansen, H. Neukirchen, and E. Hvannberg. Practical Cloud Evaluation from a Nordic eScience User Perspective. In *VTDC11. The 5th International Workshop on Virtualization Technologies in Distributed Computing held in conjunction with the 20th International ACM Symposium on High-Performance Parallel and Distributed Computing San Jose, California, June 8-11, 2011 (to appear in ACM digital library)*, 2011.

[21] M. Fowler and M. Foemmel. Continuous Integration. Available on-line: `http://martinfowler.com/articles/originalContinuousIntegration.html`, 2000.

[22] S. Freeman and N. Pryce. *Growing Object-Oriented Software, Guided by Tests.* Addison-Wesley Professional, 2009.

[23] Z. Ganon and IE Zilbershtein. Cloud-based Performance Testing of Network Management Systems. In *Computer Aided Modeling and Design of Communication Links and Networks, 2009. CAMAD'09. IEEE 14th International Workshop on*, pages 1–6. IEEE, 2009.

[24] J. Geelan. Twenty One Experts Define Cloud Computing. *Virtualization, August*, 2008.

[25] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). *Computer Networks*, 42(3):375–403, 2003.

[26] J. Gray and A.S. Szalay. The World Wide Telescope: An Archetype for Online Science. *Arxiv preprint cs/0403018*, 2004.

[27] L. Gu and S.C. Cheung. Constructing and testing privacy-aware services in a cloud computing environment: challenges and opportunities. In *Proceedings of the First Asia-Pacific Symposium on Internetware*, page 2. ACM, 2009.

[28] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato. Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems. In *Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 428–433. IEEE, 2010.

[29] T. Hanawa, H. Koizumi, T. Banzai, M. Sato, S. Miura, T. Ishii, and H. Takamizawa. Customizing Virtual Machine with Fault Injector by Integrating with SpecC Device Model for a Software Testing Environment D-Cloud. In *2010 Pacific Rim International Symposium on Dependable Computing*, pages 47–54. IEEE, 2010.

[30] J. Howe. The Rise of Crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

[31] Steiner I. Amazon Peak Shopping Day up 44%, Kindle Becomes Its Best-selling Product. Available on-line: `http://www.auctionbytes.com/cab/cab/abn/y10/m12/i28/s01`, 2010. Date of publication: December 28, 2010. Date retrieved: April 4, 2011.

[32] IEEE. Standard for Software and System Test Documentation. *IEEE Std 829-2008*, 2008.

[33] IEEE Standard for Software Verification and Validation. IEEE Std 1012-2004. Inst. of Electrical and Electronical Engineers, 2004.

[34] M. Ingstrup and K.M. Hansen. Modeling Architectural Change: Architectural scripting and its applications to reconfiguration. In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 337–340. IEEE, 2009.

[35] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[36] ISO/IEC. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. International ISO/IEC multipart standard no. 9646. *ISO/IEC*, 1994-1997.

[37] A. Khajeh-Hosseini, I. Sommerville, and I. Sriram. Research Challenges for Enterprise Cloud Computing. *Arxiv preprint arXiv:1001.3257*, 2010.

[38] T.M. King and A.S. Ganti. Migrating Autonomic Self-Testing to the Cloud. In *Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 438–443. IEEE, 2010.

[39] C. Kulla. Sunflow Rendering System Tutorial. `http://www.foro3d.com/f214/sunflow-rendering-system-tutorial-76462.html`, 2009. Date of publication: Juny 2, 2009. Date retrieved: April 10, 2011.

[40] V. Massol and T. Husted. *JUnit in Action.* Manning Publications Co. Greenwich, CT, USA, 2003.

[41] P. Mell and T. Grance. The NIST Definition of Cloud Computing. *National Institute of Standards and Technology*, 2009.

[42] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code.* Prentice Hall PTR Upper Saddle River, NJ, USA, 2006.

[43] H. Neukirchen. Testing Distributed and Parallel Systems with TTCN-3. [Extended Abstract]. In *Para 2010: State of the Art in Scientific and Parallel Computing*, 2010.

[44] M. Oriol and F. Ullah. YETI on the Cloud. In *Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 434–437. IEEE, 2010.

[45] T. Parveen and S. Tilley. When to Migrate Software Testing to the Cloud? In *2010 International Conference on Software Testing, Verification, and Validation.* IEEE, 2010.

[46] S. Pota and Z. Juhasz. The benefits of Java and Jini in the JGrid system. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, page 255. IEEE, 2006.

[47] J. Radatz. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610121990*, 121990, 1990.

[48] J. Rhoton. *Cloud Computing Explained.* Recursive Limited, 2010.

[49] T. Rings. Testing Grid Applications Using TTCN-3. Master's thesis, Georg-August-University, Göttingen, 2007.

[50] T. Rings, H. Neukirchen, and J. Grabowski. Testing Grid Application Workflows Using TTCN-3. In *2008 International Conference on Software Testing, Verification, and Validation*, pages 210–219. IEEE, 2008.

[51] L.M. Riungu, O. Taipale, and K. Smolander. Research Issues for Software Testing in the Cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 557–564. IEEE, 2010.

[52] L.M. Riungu, O. Taipale, and K. Smolander. Software Testing as an Online Service: Observations from Practice. In *Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 418–423. IEEE, 2010.

[53] P. Runeson. Software Testing. Lecture at Lund University, Jan., 2011.

[54] Q. Runtao, H. Satoshi, T. Ohkawa, T. Kubota, and R. Nicolescu. Distributed Unit Testing. University paper. Available on-line: `http://citr.auckland.ac.nz/techreports/2006/CITR-TR-191.pdf`, University of Auckland, New Zealand, [year missing].

[55] R. Skulason. Architectural Operations in Cloud Computing. Master's thesis, University of Iceland, Reykjavik, Iceland, 2011.

[56] H. Takabi, J.B.D. Joshi, and G. Ahn. Security and Privacy Challenges in Cloud Computing Environments. *Security & Privacy, IEEE*, 8(6):24–31, 2010.

[57] Google Trends. Google Trends Diagram (searchwords: 'cloud computing, software testing'). `http://www.google.com/trends?q=Cloud+computing%2C+Software+Testing`, mars 2011.

[58] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[59] T. Walter, I. Schieferdecker, and J. Grabowski. Test Architectures for Distributed Systems-State of the Art and Beyond. In *Testing of communicating systems: proceedings of the IFIP TC6 11th International Workshop on Testing of Communicating Systems (IWTCS'98)*, page 149. Citeseer, 1998.

[60] J. Wu, L. Yang, and X. Luo. Jata: A Language for Distributed Component Testing. In *2008 15th Asia-Pacific Software Engineering Conference*, pages 145–152. IEEE, 2008.

A. Appendix

This appendix contains instructions on how to set up the Java development environment for the case study. Section A.1 contains description of the steps needed in setting up the environment. Section A.2 lists the tools used for the development. Section A.3 describes where the source code for the test case can be found. Finally some considerations need to be made if the Jata core library needs to be rebuilt, which is covered in A.4.

A.1. Development Environment Setup

The steps involved in the process of setting up the development environment are:

1. Download source code from SVN (Section A.1.1)
2. Install the AWS Toolkit for Eclipse (Section A.1.2)
3. Import Jata source code to Eclipse (Section A.1.3)
4. Import Turnip source code to Eclipse (Section A.1.3)
5. Provide Amazon EC2 security credentials (Section A.1.4)
6. Configure Amazon EC2 firewall (Section A.1.5)
7. Build and package the Turnip application (Section A.1.6)
8. Upload Turnip files to an HTTP server (Section A.1.7)
9. Run Jata test harness utility (Section A.1.8)

A.1.1. Download source code from SVN

The source code for both the Turnip application and the Jata case study is hosted at XP-Dev.com[1]. The version control system used is Apache Subversion[2]. The URL of the SVN is:

```
http://svn2.xp-dev.com/svn/cloudtest/trunk
```

[1]XP-Dev.com - http://www.xp-dev.com/
[2]Subversion. http://subversion.apache.org/

The trunk directory contains two directories:

- `/TurnipEclipseWorkspace`
- `/JataEclipseWorkspace`

The *TurnipEclipseWorkspace* contains the Java source code for the Turnip application used as the system under test. The application is described in Section 4.1. The *JataEclipseWorkspace* contains the source code for Jata (version 0.1) and the implemented case study test case, described in Section 4.2.

A.1.2. Install the AWS Toolkit for Eclipse

Eclipse is used for the Java development. The *AWS Toolkit for Eclipse*[3] provided by Amazon is an Eclipse plug-in which needs to be installed. The AWS Toolkit includes the AWS SDK for Java which is used to create and terminate EC2 instances.

A.1.3. Import to Eclipse

The first thing to do within Eclipse is to create a new solution workspace. After that all the Java projects under both the Jata workspace directory and the Turnip workspace directory are imported into the solution workspace. The Java projects for import are listed in table A.1.

Jata	The Jata core framework
JataEC2	Contains the case study test harness utility to initialize the test bed and run the test case
JataExamples	Contains the case study implementation of the test case.
turnip_asl	The Turnip ASL (Asynchronous Scripting Language) core
turnip_library	Interfaces and core code for the Turnip application. Includes the Sunflow source code
turnip_requestmanager	Contains the Request Manager OSGi bundle
turnip_web	Contains the web interface layer for Turnip
turnip_worker	Worker OSGi bundle (deployed on worker instances)
turnip_workerfactory	OSGi bundle responsible of creating worker instances

Table A.1: Eclipse Java projects for the case study

[3]AWS Toolkit for Eclipse. http://aws.amazon.com/eclipse/

A.1.4. Provide Amazon EC2 credentials

In order to run the test case within Amazon EC2 the Amazon AWS account credentials need to be available to the startup process. This is done by including a properties files under the Turnip ASL Java project called *AwsCredentials.properties*.

- `/turnip_asl/src/is/hi/turnip/cloud/aws/AwsCredentials.properties`

The file contains the two following credential properties shown in listing A.1.

```
secretKey=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
accessKey=xxxxxxxxxxxxxxxxxxxx
```

Listing A.1: Amazon AWS credential properties file

A 40 character long secret key is needed and a 20 character access-key. The information can be found within the AWS Management Console[4], under the *Security Credentials* menu.

A.1.5. Configure Amazon EC2 firewall

Various TCP/IP ports need to be open within the cloud environment, both for the Turnip application and the Jata test system. The ports for Turnip are HTTP, telnet, and Remote-OSGi ports. The telnet port is used by the Cloud-ASL for communicating with the Felix OSGi runtime. The HTTP port and the R-OSGi ports are used by the Turnip application. The ports for the Jata test system are Java RMI ports used for the Jata remote ports and system ports communication. The TCP/IP ports are listed in table A.2. The cloud firewall is configured within the Amazon AWS management console.

Port	Description
6666	Felix remote shell port (Telnet)
9000	Turnip web application port (HTTP port)
9279	Remote-OSGi port
1291	Jata RMI remote port (PTC-1)
1292	Jata RMI remote port (PTC-2)

Table A.2: Amazon EC2 Ports

[4] `http://aws.amazon.com/console/`

A.1.6. Build and Package Turnip

The Turnip Java projects needs to be built and packaged as OSGi bundles. The Java projects are built with ANT. All Turnip projects include an ANT build file which builds the Java project code and packages them as an OSGi bundle. A master build file is provided under the `turnip_library` project. Project's individual ANT build files are under each project's `/resource` directory. The master build file is:

- `/turnip_library/resources/build-main.xml`

A.1.7. Upload Turnip Files to an HTTP Server

After building the Turnip bundles they need to be uploaded to an HTTP server, along with other dependencies. The Cloud-ASL uses HTTP download features for installing the bundles to the OSGi runtime server of the cloud instances when new instances are started.

Currently no configuration file is used in Turnip so the HTTP server location is hard-wired into the ASL startup classes (is.hi.turnip.asl.groovy.ASLGroovy) for the master and (is.hi.turnip.workerfactory.impl.WorkerFactoryComponent) for the workers. The corresponding Java source files are:

- `/turnip_asl/src/is/hi/turnip/asl/groovy/ASLGroovy.java`
- `/turnip_workerfactory/src/is/hi/turnip/workerfactory/`
  `impl/WorkerFactoryComponent.java`

Listing A.2 shows the Java code that uses ASL startup procedure to launch the Turnip master node in the cloud (line 7 needs to be edited). Listing A.3 shows code for the startup of Turnip workers (line 5 is the one that needs editing) As seen in the listings not only the Turnip OSGi bundles need to be present but also OSGi packaged versions of dependent libraries. The dependent jars can be found under the `turnip_lib` project's library folder:

- `/turnip_library/lib/`

The jar files that are needed to be available on a HTTP server are the following:

1. jata-bar.jar
2. turnip_asl.jar
3. turnip_library.jar

4. turnip_requestmanager.jar
5. turnip_web.jar
6. turnip_workerfactory.jar
7. turnip_worker.jar
8. aws-java-sdk-1.0.005.jar
9. com.springsource.org.apache.commons.httpclient-3.1.0.jar
10. org.apache.servicemix.bundles.jsch-0.1.42_2.jar
11. jta26_bar.jar
12. groovy-all-1.7.3.jar
13. com.springsource.org.apache.commons.logging-1.1.1.jar
14. com.springsource.org.apache.commons.codec-1.3.0.jar
15. remote-1.0.0.RC4.jar
16. jslp-osgi-1.0.0.RC5.jar

```java
public Device start(){
  DeviceImpl device = create_instance_device("m1.small"); // creates
      ec2 instance and starts Felix on it

  ASL asl = new ASLImpl();
  asl.initialize_device(device);

  String path = "http://www.myserver.com/turnip/";

  install_component(device, path + "turnip_asl.jar");
  install_component(device, path + "turnip_library.jar");
  install_component(device, path + "turnip_requestmanager.jar");
  install_component(device, path + "turnip_web.jar");
  install_component(device, path + "turnip_workerfactory.jar");
  install_component(device, path + "aws-java-sdk-1.0.005.jar");
  install_component(device, path + "com.springsource.org.apache.commons
      .httpclient-3.1.0.jar");
  install_component(device, path + "org.apache.servicemix.bundles.jsch
      -0.1.42_2.jar");
  install_component(device, path + "jta26_bar.jar");
  install_component(device, path + "groovy-all-1.7.3.jar");
  install_component(device, path + "com.springsource.org.apache.commons
      .logging-1.1.1.jar");
  install_component(device, path + "com.springsource.org.apache.commons
      .codec-1.3.0.jar");

  start_components();

  return device;
}
```

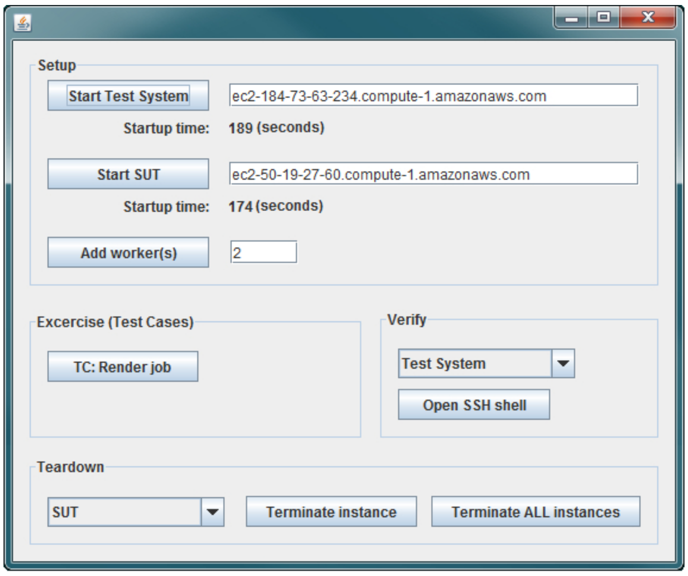
Listing A.2: Java code for starting up the Turnip main node

```java
public void run(){
  Device device = asl.create_instance_device("m1.small");
    asl.initialize_worker(device);

    String lib_path = "http://www.myserver.com/turnip/";

    asl.install_component(device, lib_path + "jata-bar.jar");
    asl.install_component(device, lib_path + "turnip_library.jar");
    asl.install_component(device, lib_path + "remote-1.0.0.RC4.jar");
    asl.install_component(device, lib_path + "jslp-osgi-1.0.0.RC5.jar")
        ;
    asl.install_component(device, lib_path + "turnip_worker.jar");

    ...
 }
```

Listing A.3: Java code for starting up Turnip worker instances

A.1.8. Execute Jata Test Harness

To execute the test case, the test harness utility is used (figure A.1). The test procedure is described in section 4.5.



ec2-184-73-63-234.compute-1.amazonaws.com
Startup time: 189 (seconds)
ec2-50-19-27-60.compute-1.amazonaws.com
Startup time: 174 (seconds)
2

Figure A.1: Test harness utility

The Java main entry class for the utility is:

- `/JataEC2/src/is/hi/amazon/turnip/TestHarnessUtility.java`

A.2. Software Tools Used

Table A.3 lists the tools used in the development of the Jata test cases in this thesis.

Purpose	Tool
Version control	Subversion hosted at XP-Dev.com
IDE	Eclipse Java EE IDE for Web Developers
Test framework	Jata version 0.1
OSGi BND tool	Peter Kriens BND tool
Amazon EC2 Firewall	The AWS management console
Test harness utility	Custom made tool for the case study

Table A.3: Software used for the case study

A.3. Jata Test Cases source

The `JataExamples` project contains the source code for the Jata test cases. The main Java package is:

- `is.hi.jata.example.parametertestcase.turnip`

This package contains code for the MTC, PTCs, the system adapter and the ports used in the test case. The test case class is:

- `is.hi.jata.example.parametertestcase.turnip.ParameterTestCase`

A.4. Jata Core Changes (if needed)

If changes are done to the core Jata library, it needs to be re-bundled as an OSGi bundle.

The Jata source is automatically built with Eclipse. What needs to be done however is to export the `Jata` project from Eclipse as a Java jar file. Furthermore the resulting jata.jar file needs to be bound as an OSGi bundle. The worker instances (test double implementation) use the Jata core library for the remote ports so the Jata jar is needed at the workers' OSGi runtime (Felix).

Peter Kriens BND tool[5] can be run on the Jata jar file to bundle it. The bnd.jar file is under directory:

- `/turnip_asl/resources/lib`

The Java command line looks like:

```
java -jar bnd.jar wrap jata.jar
```

The output is an OSGi bundled version of the Jata library. Note! The filename is '`jata.bar`' which needs to be renamed to '`jata-bar.jar`'. This file needs then to be uploaded to the HTTP server for the Cloud-ASL to access it (as described in Section A.1.7).

[5]BND Tool. `http://www.aqute.biz/Code/Bnd`