Distributed Testing of Cloud Computing Applications Using the TTCN-3-based Jata Test Framework

Hlödver Tómasson
Verisure Innovation AB
Ångbåtsbron 1
211 20 Malmö, Sweden
hlodver.tomasson@verisure.com

Helmut Neukirchen
University of Iceland
Dunhagi 5
107 Reykjavík, Iceland
helmut@hi.is

ABSTRACT

With a new technology, such as cloud computing, new challenges are introduced to software engineering. This paper investigates the challenges of software testing in elastic cloud computing environments as well as the applicability of concepts of the *Conformance Testing Methodology and Framework* (CTMF) and the open-source test framework Jata for distributed testing of cloud computing applications. Jata provides concepts from the standardised test language *Testing and Test Control Notation version 3* (TTCN-3) to implement distributed test cases directly in Java. As the main contribution, a case study of testing a distributed cloud application has been conducted. It reveals that there are specific considerations to be made to deal with the elastic nature of a cloud environment which advises automated run-time configuration of the test cases to adjust to the current environment.

1. INTRODUCTION

Cloud computing has become a viable and common alternative to traditional data centers, hosting services and private IT infrastructures. Cloud computing gives the ability to scale an IT infrastructure up and down by only using and paying for just as many resources as currently needed ("elastic" and "pay-per-use") [2, 10].

While cloud computing is not a fundamentally new paradigm, but based on existing technologies, it is far from trivial and introduces new challenges that make applications running in a cloud computing environment more complex and their development more error-prone. For example, the *Infrastructure as a Service* (IaaS) service model of cloud computing is heavily based on virtualisation technology for CPU, storage and networking. As a result, the underlying environment changes when scaling in an elastic way. Software testing needs to address cloud-specific issues to avoid the risk that it lags behind in following the fast growing trends of the cloud computing industry.

As the main contribution, this paper investigates challenges of distributed testing in an elastic IaaS cloud computing environment as well as the applicability of concepts of the *Conformance Testing Methodology and Framework* (CTMF) [8] and the open-source test framework Jata [17] that provides concepts from the *Testing and Test Control Notation* (TTCN-3) [4]. The approach that we pursue is similar to the approach that one of the authors has applied when investigating issues in testing grid computing applications with TTCN-3 [13]. However, the grid computing environment was static and lacked completely the challenges of the elastic nature of a cloud computing environment.

This paper is structured as follows: subsequent to this introduction, we provide foundations and discuss challenges. Afterwards, in Sect. 3, we describe the cloud application that is put under test in our case study. In Sect. 4, the actual case study of testing the cloud application from Sect. 3 is presented. Related work is discussed in Sect. 5, before we conclude with a summary and outlook.

2. FOUNDATIONS AND CHALLENGES

In the following, we provide an overview on cloud computing, on CTMF, and on the test specification language TTCN-3. Furthermore, an introduction is given on the Jata test framework that can be used to implement distributed tests. We also point out challenges for testing in a cloud environment.

2.1 Cloud Computing

The *National Institute of Standards and Technology* (NIST) defines cloud computing as "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" [10].

2.1.1 *Infrastructure as a Service.*

In the *Infrastructure as a Service* (IaaS) cloud computing service model [10], the computational resources are typically provided by *Virtual Machines* (VMs). A cloud application developer has to take make sure that VM instances are started and terminated on demand in order to scale according to the current load. To this aim, an IaaS cloud environment provides an *Application Programming Interface* (API). It is the responsibility of the cloud application developer to take care that the cloud application is a distributed application that utilises all available VM instances and that the application gets reconfigured if the number of VM instances changes. This may also include cloud firewall settings with respect to the virtual network connections between the VM instances within the cloud and to the outside.

The Amazon *Elastic Compute Cloud* (EC2) [1] (complemented with further services of the *Amazon Web Services* (AWS) family) is a popular IaaS cloud provider and thus its API can be considered as industry standard. An example of an alternative implementation of the EC2 API (and further AWS services) is the open-source/open-core *Eucalyptus* [5] that can be used to install a local, private cloud.

2.1.2 *Challenges.*

Typically, it is out of control of the developer and the application running in a cloud environment which IP address and MAC address is assigned to a newly launched VM instance. This may lead to various problems when running and testing software in a cloud environment.

For example, software (such as libraries to be used by the cloud application or test tools running in the test environment) with a license management which is tied to a unique property of a customer's computer, typically the MAC addresses, cannot be used if the MAC addresses are not fixed.[1]

Another example are changing IP addresses that make the addresses of test nodes unpredictable. Therefore, a run-time configuration of the test cases is required with respect to any IP address that needs to be known as part of testing.[2]

2.2 Conformance Testing Methodology and Framework

The ISO/IEC multipart standard 9464 *OSI Conformance Testing Methodology and Framework* (CTMF) [8] provides proven concepts (yielding benefits comparable to (design) patterns) for conformance testing of *Open Systems Interconnection* (OSI) protocol implementations. Despite the OSI scope, CTMF has been successfully applied for testing other kinds of distributed systems than OSI network protocols. Due to our experience from adopting CTMF for grid application testing, we decided to adopt relevant concepts from CTMF as well for distributed testing of cloud applications. In particular from Part 2 of CTMF, the test suite generation procedures and test architectures are applicable in our context whereas other parts of CTMF are too specific with respect to either OSI protocols or the scope of conformance testing[3].

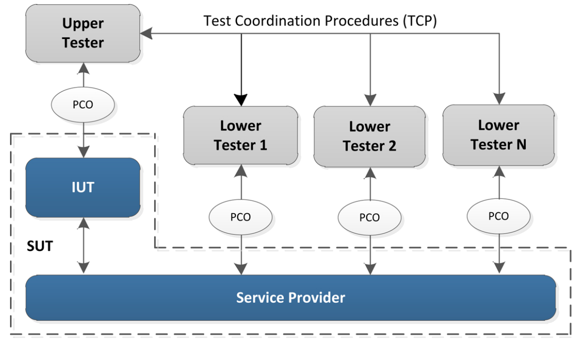

Figure 1: CTMF Multi-party test architecture

CTMF involves the identification of the requirements to be tested for which then a set of *test purposes* has to be created. A test purpose is an informal description of a test case. Next, a test architecture that is suitable for a test purpose has to be selected. Then, for each test purpose, a *test case* is designed based on the chosen test architecture. CTMF suggests to use as test notation the *Tree and Tabular Combined Notation* (TTCN) that is standardised in Part 3 of CTMF. TTCN is a predecessor of TTCN-3 (see Sect. 2.3). Both, TTCN and TTCN-3, are abstract languages to specify abstract test suites containing test cases that are implementation independent. To obtain executable test cases, the abstract test cases need to be implemented either manually or by generating them automatically from the abstract test cases. A *test suite* comprises multiple test cases.

The test architectures provided by CTMF are called *test methods*. Just like TTCN, they abstract from implementation details by introducing the constituents *Implementation Under Test* (IUT), *System Under Test* (SUT)[4], *testers*, *Points of Control and Observation* (PCOs), and the relations between them. The PCOs are used by the testers to stimulate and observe the SUT.

For our purpose, the CTMF *Multi-party test method* is suitable. It is depicted in Fig. 1. The SUT consists of the IUT and an underlying service provider (in our case: the cloud environment) via which the IUT communicates with other peer entities. The IUT is controlled and observed by an *Upper Tester* (UT) and one or more *Lower Tester*s (LTs). The UT takes the role of a higher layer or user of the IUT, the LTs replace the peer entities that together with the IUT provide the service used by the UT. The PCOs are the interfaces used by UT and LTs to communicate with the SUT. These interfaces are given by the SUT. In contrast are the

[1] Just after the case study described in this paper had been conducted, Amazon EC2 added the concept of an *Elastic Network Interface* which is a virtual network interface that exists separate from VM instances and thus keeps its MAC address. It can then be added to a VM instance and thus provides elastic VM instances with a fixed MAC address.

[2] While Amazon EC2 provides *Elastic IP addresses* that stay fixed and can be dynamically mapped to VM instances, the number of such elastic IP addresses is limited, as they are mainly intended to provide a fixed public IP address to access the cloud application from outside, for example, some web interface. Meanwhile, Amazon introduced the *Virtual Private Cloud* (VPC) service that allows to create a complete virtual network with defined IP addresses.

[3] *Conformance testing* is about adherence of implementations to standards, for example to make statements which optional parts of protocol standards are implemented. However, in our case study, we perform functional testing in general which is similar to conformance testing as both apply *black-box testing* approaches: testing via external interfaces without making any assumption about internals.

[4] The IUT is what shall be actually tested; however, some IUT interfaces involved in testing are only accessible via some service provider, for example via a network stack in order to send a network message to the IUT or via a complete cloud environment in which the IUT is running. While such a service provider is not subject of testing, it is unavoidably involved during testing because a further isolation (unit testing) of the IUT is either not possible or not reasonable.

Test Coordination Procedures (TCPs)[5] that are needed to coordinate the testers to reach the common test purpose: the TCPs can be based on interfaces and protocols that are independent from the SUT. Using the CTMF multi-party test method results in testers that are concurrently running and as these are typically running on different nodes, the resulting test is a distributed test.[6]

2.2.1 Challenges.

It is assumed that the underlying service provider has already been adequately tested: only in this case, the actual service provider implementation used in a test environment does not matter. However, for cloud environments, no conformance test suites exist to test them. So even though, for example, Eucalyptus implements the Amazon EC2 API, it may behave differently than Amazon EC2. Therefore, using a local Eucalyptus cloud environment is not an adequate test environment for a cloud application that will later run in the Amazon EC2 as a production environment.

2.3 TTCN-3

The *Testing and Test Control Notation* (TTCN-3) [4, 7] is a test specification language standardised by the *European Telecommunications Standards Institute* (ETSI). TTCN-3 is a successor of TTCN (Part 3 of CTMF), but all OSI-specific concepts have been removed or generalised. As it is a standardised test language, tool-support from multiple vendors is available. TTCN-3 is widely used in the telecommunication domain, but also in other domains such as the automotive industry or for testing implementations of Internet protocols such as the *Session Initiation Protocol* (SIP).

In contrast to other test technologies, such as JUnit, TTCN-3 supports distributed tests. It does so by means of *Test Components* (TCs): in addition to the *Main Test Component* (MTC), further *Parallel Test Components* (PTCs) can be created dynamically. TCs run concurrently and may therefore execute test behavior in parallel to each other. They can be connected to each other or mapped to the SUT interfaces via *ports*. These concepts allow the creation of distributed test architectures, for example instantiations of the CTMF test methods by using the TCs to realise upper and lower testers and the ports take the role of PCOs. (Also the interfaces used for the TCPs can be modelled as ports). A *system component* is used to define the interface to the SUT based on the ports contained in that component.

For the communication between TCs and with the SUT, operations such as **send** and **receive** (TTCN-3 language keywords are printed bold) can be used to transmit messages via ports. The values of these message are specified using so called *templates*: TTCN-3 templates may involve wildcards and thus provide a powerful matching mechanism to check whether expected test data has been received or not.

Further concepts that ease test specification are test verdict handling, timers, and *alternatives* and *defaults*: Alternatives can be used to describe branching test behaviour where a test case reacts differently based on observed events such as received messages or expiration of timers. Defaults provide default reactions for alternative behavior which is typically used to deal with unexpected events. In addition to these test-specific concepts, most of the concepts of general purpose programming languages are available as well. Sample TTCN-3 code will be later explained in Section 4.3.

TTCN-3 test suites are abstract. For obtaining an *Executable Test Suite* (ETS), the TTCN-3 statements of the *Abstract Test Suite* (ATS) are typically translated by a TTCN-3 compiler into statements of an implementation language such as Java or C/C++. In addition, the abstract communication mechanisms need to be adapted to concrete communication mechanisms as well as the abstract messages need to be encoded and decoded into/from some concrete bit-level format. For that purpose, the TTCN-3 standard suggests to use a *System Adapter* (SA) that implements operations such as **send** and **receive** operations to communicate with the SUT and a *Coding/Decoding* (CD) entity.

2.3.1 Challenges.

Some challenges arise when trying to use TTCN-3 for distributed testing in a cloud environment: First, only few of the TTCN-3 tools support distributed testing where different TCs and their ports are located on different network nodes. Second, many commercial TTCN-3 tools use a license enforcement technology that ties a license to a unique property of a customer's computer, typically the MAC address. As described in Sect. 2.1, this leads to problems in an elastic and virtualised environment and therefore, solutions for this problem need to be provided either by the TTCN-3 tool vendor or the cloud provider.

2.4 Jata Framework

Jata [17] is an object-oriented framework for implementing distributed tests in Java. Since it is free and open-source [9], none of the license issues of running TTCN-3 tools to execute test cases in a cloud environment (as discussed in Sects. 2.1 and 2.3) can arise.

Jata supports many concepts from TTCN-3, such as: test components (TCs), ports, alternatives, timers, and verdicts. Therefore, Jata can be used to implement an abstract TTCN-3 test suite, thus turning it into an ETS. Providing the TTCN-3 concepts as classes of a Java framework has the advantage that a Java programmer immediately feels familiar[7] when implementing distributed tests.

Even though test cases can be directly implemented using Jata without a TTCN-3 ATS as intermediate step, the test concepts that are borrowed from TTCN-3 need to be understood, so a Java programmer needs nevertheless some knowledge of TTCN-3. Using Java has the disadvantage that not all TTCN-3 concepts can be mapped well on the Java syntax. In addition, Jata does not support all TTCN-3 concepts: for example, the template pattern matching mechanism needs to be implemented on low-level in Java as well as defaults. Furthermore, it has to be noted that while Jata

[5]Not to be confused with the *Transmission Control Protocol* (TCP) from the Internet protocol family.

[6]Note that the IUT itself is not necessarily distributed even if the corresponding test is distributed.

[7]The Jata framework does not always adhere to Java naming conventions which may be confusing for a Java programmer, though.

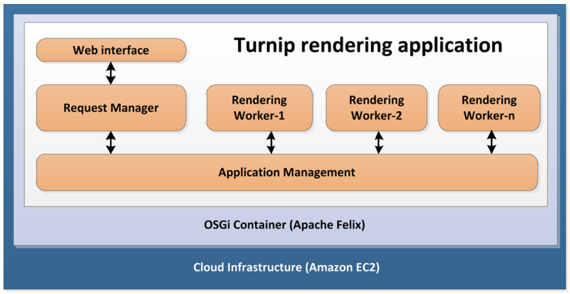

Figure 2: Turnip application architecture

supports distributed testing involving multiple test nodes, only the ports of a TC themselves can be physically distributed – the behaviours of all TCs still run on one single node as multiple concurrent threads within one process. In case of such *remote ports*, the Java RMI middleware technology is used by default for communication between the TCs and the ports that reside on remote nodes.

2.4.1 Challenges.

As Jata is new, there is not much experience and a lack of documentation. Furthermore, Jata does not implement all TTCN-3 features.

3. CLOUD APPLICATION

In order to study distributed testing of cloud computing applications, we used *Sunflow* [15], a photo-realistic image rendering system that supports to split the rendering problem into smaller subtasks (*buckets*) that can be processed in parallel. (Like Jata, Sunflow is implemented in Java, but our black-box testing approach does not require this.) While Sunflow supports well multicore CPUs, it does not support distributed computing in a cloud environment. However, it has been adapted in a preceding project to run in the Amazon EC2 cloud environment by creating worker node instances on demand [14]. This adaptation is called *Turnip* and we use it as a sample cloud application in our case study.

The distributed architecture of Turnip is shown in Fig. 2. It consists of a *Request manager* component that provides a Web-based (HTML/HTTP) user interface to initiate the addition of rendering worker nodes, starting the rendering job, and to monitor and display the rendering progress. Internally, the Request manager does some application management by creating EC2 VM instances, starting Sunflow processes on each instance and collecting their results. The Java code is executed within OSGi [11] containers: using OSGi eases mainly deployment [14], but beyond this, a further OSGi facility, *Remote OSGi* (R-OSGi), is used for the communication between the Request manager and the worker nodes. The resulting distributed application runs within the Amazon EC2 cloud environment: the EC2 API is used to create new VM instances for the worker nodes.

Fig. 3 shows a Turnip scenario with two worker nodes, each of them processing one bucket: first, a work request is submitted via the web interface to the Request manager. In steps 2 and 3, the Request manager initalises the Sunflow workers which then ask the Request manager for the next bucket to render (steps 4 and 5). After a worker finishes a

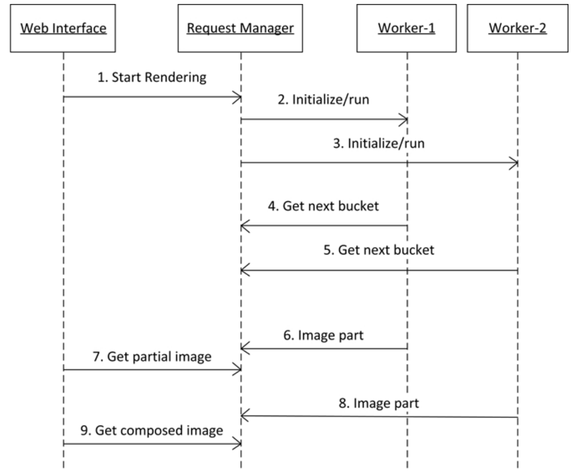

Figure 3: Distributed rendering scenario

task, it sends the resulting image part to the Request manager (steps 6 and 8). The web interface polls the Request manager periodically for intermediate results (Step 7). If rendering is completed, the web interface requests the final composed image (Step 9).

4. CLOUD APPLICATION TESTING CASE STUDY

To investigate challenges and obstacles in distributed testing of applications running in an elastic cloud environment, we used the described Turnip cloud application and followed the applicable procedures of CTMF as outlined in Sect. 2.2: we started with creating a test purpose and selected a test architecture that is appropriate for testing the test purpose. However, to avoid any license issues in the cloud test environment (see Sects. 2.1 and 2.3), we did not use a commercial TTCN-3 tool to generate the ETS, but implemented the test cases using the Jata framework. A further motivation for using Jata was that we wanted to evaluate the applicability and maturity of Jata.

4.1 Test Purpose

As our investigation is about distributed testing, we focused on the testing of requirements for the Request manager which communicates via its Web-based user interface and –as part of its application management– with multiple parties (the workers) and thus requires a distributed test architecture. The considered test purpose is:

> Assess that the rendering job information entered via the Web-based user interface to the Request manager results in the correct sequence of actions of the Request manager with respect to the application management: the proper distribution of tasks to the workers and merging their partial rendering outputs to a final image.

The correct sequence of actions (for a scenario with two workers) is essentially already shown in Fig. 3: the Request manager is the IUT, the initial stimulus via the Web interface is provided by the test system, and the test system replaces the workers by stubs that observe the messages send to the workers and send back responses to the IUT.

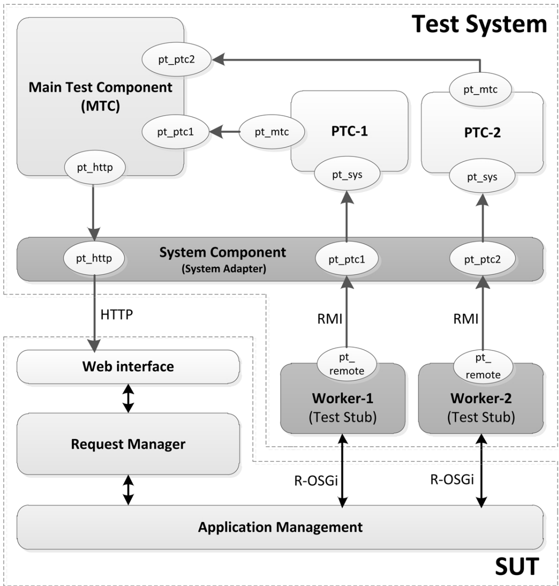

Figure 4: Test architecture (CTMF Multi-party)

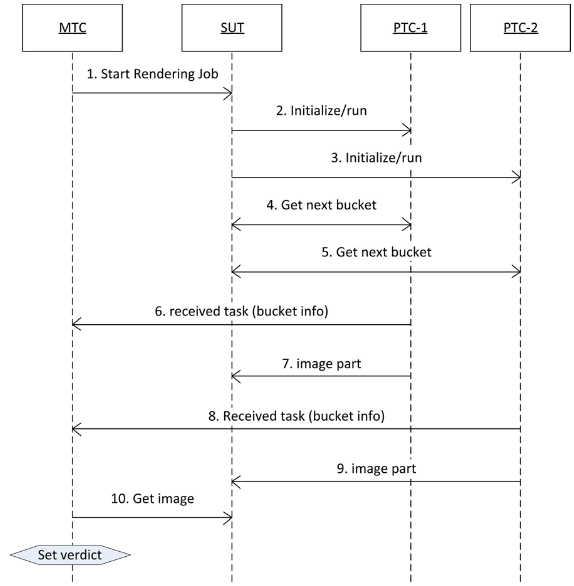

Figure 5: Message sequence of abstract test case

A functional black-box test is performed. The underlying service providers (Amazon EC2, Apache Felix OSGi container) are expected to be correct: we do for example not explicitly test that EC2 actually creates a VM instance when the Request manager uses the EC2 API for that purpose (however, during a test of the Request manager, it would become obvious if an expected VM instance is not running). Furthermore, we test no negative scenarios such as timeouts or wrong input data.

4.2 Test Architecture

For turning the test purpose into a test case, a suitable test architecture needs to be selected in order to be able to assign test behaviour to the different TCs of the test architecture. In our case, the request manager is the IUT, the other constituents of the cloud application need to be replaced by testers: the CTMF *Multi-party test method* is applicable.

The used instantiation of the Multi-party test method is shown in Fig. 4. For simplicity, we use a scenario with two worker nodes. The upper tester role is filled by the MTC, for the lower testers, PTCs are used. They coordinate themselves internally via connected ports and communicate with the SUT via ports that are mapped to the ports of the system component. The MTC controls and observes the SUT via an HTTP connection and to observe the calls made by the SUT via R-OSGi to the workers, they get intercepted by test stubs that forward their observations to the PTCs[8]. The MTC can then determine a test verdict based on its own observations and the ones made by the PTCs.

The underlying service providers are only implicitly depicted in Fig. 4: for example, the R-OSGi-based communication is provided by Apache Felix, all TCP-based communication is provided by the TCP/IP stack of the Linux operating system running inside a VM instance, and all nodes are running within the EC2 cloud environment.

4.3 Abstract Test Case

The test purpose may now be turned into a test case that fits the selected test architecture. To this aim, the test behaviour is accordingly distributed to the involved TCs and necessary *Test Coordination Procedures* (TCPs) are added. The resulting sequence of abstract messages is shown in Fig. 5. It is derived from the scenario in Fig. 3 and contains the additional TCPs in steps 6 and 7 and the setting of a test verdict by the MTC based on the observations.

The MTC starts the test case by sending the 'start rendering job' stimulus (Step 1). The SUT that thinks it is announcing real workers to start rendering, is actually calling PTCs (steps 2 and 3). The PTCs ask for the next bucket to render as would be expected from a real worker (steps 4 and 5). When a PTC receives the bucket information, it sends this information to the MTC (steps 6 and 8), which collects for verification all knowledge about the rendering tasks that are created by the SUT. The PTC creates a dummy image to send back as job result to the SUT as is expected from rendering tasks (steps 7 and 9). Finally, when the MTC has received all expected bucket information it downloads the composed image result from the SUT (Step 10). The MTC verifies the number of messages it received from the PTCs and the file size of the composed dummy image and sets the test verdict accordingly.

In a next step, the graphical test specification in Fig. 5 can be either used as specification to develop a TTCN-3 ATS (and then generate an ETS from it or handcraft an ETS using the TTCN-3 ATS as specification) or –if a platform independent ATS is not required– this step can be omitted and the ETS is immediately implemented manually using Jata. As we decided against a TTCN-3 execution tool and instead wanted to evaluate the applicability of Jata, we did create a TTCN-3 ATS only for comparison purposes.

[8]Using for this a different communication mechanism (Java RMI) avoids probe effects where the communication of the test system could influence the communication of the SUT.

```
1 testcase tc_applicationManagement() runs on mtcType system systemType {
2   var ptcType ptc1 := ptcType.create; // Create PTCs
3   var ptcType ptc2 := ptcType.create;
4   map(ptc1:pt_system, system:pt_PTC1); // PTC−1 <−> SUT
5   connect(self:pt_PTC1, ptc1:pt_mtc); // PTC−1 <−> MTC
6   map(ptc2:pt_system, system:pt_PTC2); // PTC−2 <−> SUT
7   connect(self:pt_PTC2, ptc2:pt_mtc); // PTC−2 <−> MTC
8   map(self:pt_system, system:pt_http); // MTC <−> SUT
9   ptc1.start(ptcFunc()); // Start PTC functions
10  ptc2.start(ptcFunc());
11  pt_system.send(MsgStartRendering(sutIpAddress)); // Stimulus
12  getReceivedJobMessages(); // Wait for all tasks or timeout
13  verifyJobMessages(); // Verify
14  verifyResultsFileSize();
15  setverdict(pass);
16 }
```

Listing 1: Test case implementation using TTCN-3

```
1    timer T;
2    var msgReceivedJobType receivedMsg;
3    T.start(2.0);
4    alt {
5      []  pt_system.receive(msgReceivedJob) −> value receivedMsg {
6          pt_mtc.send(receivedJobsForward(receivedMsg));
7        }
8      []  T.timeout {
9          setverdict(fail);
10       }
11   }
```

Listing 2: PTC test function using TTCN-3

Listing 1 shows the TTCN-3 test case that executes the MTC behaviour and it is therefore declared that it **runs on** (Line 1) a TC of type *mtcType* (that contains the ports used by the MTC) and interfaces to the **system** under test via the ports of a component of type *systemType*. Next, the two identical PTCs are created (lines 2 and 3) and connected to each other or respectively mapped to the ports that represent the SUT (lines 4 to 8 – note that slightly different names than in Fig. 4 are used). After this setup of the test architecture, the two concurrent PTCs are **start**ed by passing as parameter a reference to a TTCN-3 function that contains the PTC behaviour to be executed (lines 9 and 10). Then, the 'start rendering job' stimulus (Step 1 in Fig. 5) is sent (Line 11). Lines 12 to 14 call MTC behaviour that waits for job messages which are forwarded by the PTCs and verifies them subsequently. The called behaviour also sends an HTTP-request to get the final image composed by the SUT (Step 10 in Fig. 5) and verifies that it has the correct size. If any deviations from expected results are detected, the called behaviour sets the test verdict to *fail*. Line 15 sets the verdict to **pass**: due to the TTCN-3 verdict overwriting rules, this will not re-set a previously set **fail** verdict to **pass**.

A TTCN-3 snippet of the PTC behaviour is shown in Listing 2: first, a **timer** and a variable for storing a received message are declared (lines 1 and 2). The timer that is supposed to expire after 2 seconds, is started in Line 3. An alternative (**alt**) is used in lines 4 to 11 to wait either for reception of a job description (Line 5) from the SUT (according to steps 2 and 3 in Fig. 5) and forward (Line 6) the received values to the MTC (according to steps 6 and 8 in Fig. 5) or to or to catch a timeout (Line 8) of the timer in case no job description arrives. In this case, the test verdict is set to **fail** (Line 9). (Steps 4, 5, 7, and 9 from Fig. 5 are not reflected in this snippet.)

4.4 Executable Jata Test Case

For obtaining the full ETS, the complete behaviour conveyed in the graphical ATS specification was manually implemented in Java using the Jata framework. In addition, the adaption to the communication mechanisms of the SUT (including coding/decoding of messages) was realised.

Listing 3 provides an excerpt of the Jata-based implementation of the MTC behaviour: it matches line by line the corresponding TTCN-3 specification in Listing 1 because Jata (and the underlying Java) supports the used TTCN-3 constructs very well. In contrast are TTCN-3 alternatives (as used in Listing 2): in particular the TTCN-3 syntax does not blend well with Java, thus TTCN-3 alternatives look awkward when implementing them using Jata. As shown in Listing 4 that provides the Jata implementation of the TTCN-3 specification in Listing 2, alternatives are represented in Jata as runtime objects (Line 1) to which the alternative branches for receiving a message (Line 4) and for catching a timeout (Line 5) are added at runtime. Waiting for one of the alternative branches to occur is triggered in Line 6. Once one of these alternatives is observed, the corresponding reactions are performed (lines 8 to 15) depending on the which of the branches occurred (Line 7).

For the adaption layer, we use the open-source libraries *HtmlUnit* and Apache *HttpClient* for the HTML/HTTP-based communication. To replace the Sunflow renderers by test stubs, we simply replace the whole OSGi Sunflow worker service implementation by a test stub that implements the same Java interface for the worker service. For this, we only

```
1 protected void Case(Mtc mtc, Sys sys) throws JataException {
2   Ptc ptc1 = createPtc(Ptc.class); //  Create PTCs
3   Ptc ptc2 = createPtc(Ptc.class);
4   PipeCenter.Map(ptc1.pt_PS, sys.pt_PTC1); // PTC−1 <−> SUT
5   PipeCenter.Map(mtc.pt_PTC1, ptc1.pt_PM); // PTC−1 <−> MTC
6   PipeCenter.Map(ptc2.pt_PS, sys.pt_PTC2); // PTC−2 <−> SUT
7   PipeCenter.Map(mtc.pt_PTC2, ptc2.pt_PM); // PTC−2 <−> MTC
8   PipeCenter.Map(mtc.pt_system, sys.pt_http); // MTC <−> SUT
9   startPTC(ptc1); // Start PTC functions
10  startPTC(ptc2);
11  mtc.pt_system.send(new MsgStartRendering(sutIpAddress)); // Stimulus
12  getReceivedJobMessages(); // Wait for all tasks or timeout
13  verifyJobMessages(); // Verify
14  verifyResultsFileSize();
15  mtc.Pass(); // VERDICT=PASS
16 }
```

Listing 3: Test case implementation using Jata

```
1    Alt alt=new Alt();
2    Timer t=new Timer();
3    t.start();
4    alt.addBranch(ptc.pt_system, new MsgReceivedJob());
5    alt.addBranch(t, new SetTimeMessage(2000));
6    alt.proc();  // Process alt statement
7    switch (alt.getResult().index){
8    case 0: // Received expected message
9      // Forward PTC's incoming parameter message to the MTC
10     MsgReceivedJob msg=(MsgReceivedJob)alt.getResult().Result;
11     ptc.pt_MTC.send(msg);
12     break;
13   case 1 : // Time−out
14     ptc.Fail();
15     break;
16   }
```

Listing 4: PTC test function using Jata

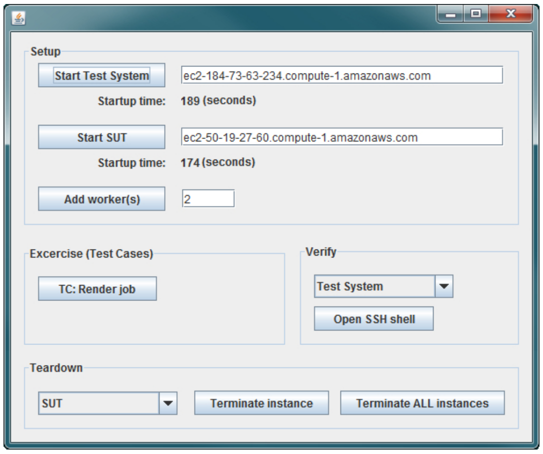

Figure 6: Test execution utility

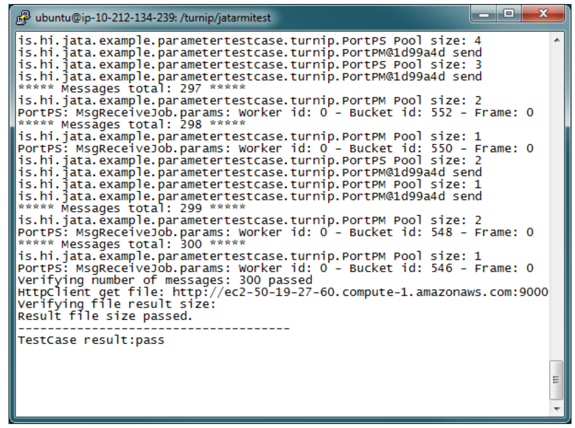

Figure 7: Test log

have to change the OSGi service descriptor file: the SUT is not aware of that change and when it intends to start and call a real Sunflow worker on a VM instance that it created, it does in fact start and call a test stub. This test stub serves as Jata remote port (see Sect. 2.4) which passes the received message via Java RMI through to the PTC. Further details on the Jata test case implementation can be found in the thesis of Tómasson [16].

4.5 Test Execution

The prerequisites for the test execution are that the SUT has been configured and started, the scene file to be rendered has been uploaded, and two worker instances (in fact, the test stubs) have been added: the latter is typically achieved via the Turnip web interface.

While the application under test obviously runs in the cloud environment as well as the test stubs that replace components running in the cloud, there are some parts of the test system that do not necessarily need to run in the cloud environment, but could be executed locally, such as the MTC and the Jata test console. However, to avoid any firewall issues with respect to the Java RMI-based communication between the the test stubs and PTCs (that run on the same node as the MTC), we decided to run everything within the cloud environment. By this means, we avoid to use different EC2 firewall settings for testing and for productive use where any Java RMI messages crossing the cloud boundaries will anyway be blocked for security reasons.

To cope with the elastic nature of the test environment, we created a graphical test execution utility that can be started on a local machine to control (via a *Secure Shell* (SSH) connection) the test campaign running in the cloud environment (Fig. 6): it allows to create an EC2 instance for the part of the test system that runs the MTC (which will then create at run-time further PTCs), to create an EC2 instance that runs the Turnip IUT (request manager including web interface and application management), to configure it (upload scene file, add EC2 instances for the workers containing our test stubs), and to run the Jata test case and view the test log (Fig. 7). (Start of test execution without human intervention can be supported by a script.)

Because the test execution utility created the IUT node, it knows its IP address dynamically assigned by the cloud environment and can pass it automatically to the Jata test case as test case parameter for the target address of the HTTP requests. Also the test stubs need to know the IP address of the node that executes the MTC and PTCs to enable the stubs to forward the intercepted job messages to that node. The test execution utility automates this as well, thus avoiding manual test parametrisation each time the IUT node and MTC node VM instances are (re-)created.

4.6 Discussion

The problems of distributed testing of applications in the Amazon EC2 cloud environment were successfully solved: a) We used the productive cloud environment (Amazon EC2) instead of some artificial (Eucalyptus) to avoid problems due to different cloud service implementations (which however increases testing costs due to pay-per-use). b) Firewall problems were avoided by moving all TCs into the cloud. (However, that also means that the access to the Web-based user interface comes from within the cloud – in practise it would rather come from outside.) c) We avoided license problems by using the open-source Jata framework instead of commercial TTCN-3 tools. (However, technical license problems should be nowadays solvable in cooperation with a tool vendor.) d) To deal with the elastic nature of a cloud environment, we created a test execution tool that supports automated run-time configuration to pass IP addresses that are subject to change into the test cases as run-time parameters without any manual intervention.[9]

Using the Jata Java test framework to obtain an executable test suite instead of using a commercial TTCN-3 tool proved feasible and pleases Java developers who do not need to learn TTCN-3 syntax and can use the full power of Java. However, it has to be said that while some part of the resulting test code resembles one-to-one the comparable TTCN-3 code, other parts are awkward when Jata is used. In summary, it can be concluded that Jata is definitely suitable for smaller

[9]Note that it is nothing new and not unusual to pass in IP addresses of the test environment as parameters into a test case. However, in non-elastic environments, the test environment is set-up once and IP addresses remain fixed between different test runs and manual configuration of the test case parameter is economically reasonable.

projects, but we cannot make a reliable statement about huge, industrial-sized projects where commercial TTCN-3 tools may have their strengths. For example, the scalability of a test system may be limited: even though Jata supports remote ports to be distributed to different nodes, the PTCs to which the remote ports are connected have to run all on the same node as the MTC. This may overload that single test node as well as it produces network load for the communication between remote ports and PTCs.

5. RELATED WORK

To our knowledge, not much work has been published in the field of distributed testing of cloud applications. Gao et al. [6] provide an overview on the broader field of testing and cloud computing. While they point out challenges, these are high-level and not of technical nature as the challenges identified by us in Sect. 2.

Chan et al. [3] propose testing criteria for cloud applications. One criterion relates to testing whether a cloud application performs correctly after scaling and addresses the problem that the number of possible scaling paths are potentially infinite, and thus, it is infeasible to test every configuration.

While Rings et al. [12] mention briefly using TTCN-3 for conformance testing, they focus on interoperability testing of grid and cloud infrastructures. Interoperability testing is different from our CTMF-based testing approach, in particular such tests are typically conducted manually.

6. SUMMARY AND OUTLOOK

We pointed out challenges in testing applications that run in an elastic cloud computing environment and provided a case study to demonstrate how to circumvent these challenges when conducting distributed testing of a cloud application. While we have shown that distributed testing concepts from the *Conformance Testing Methodology and Framework* (CTMF), such as the Multi-party test method, are applicable as well in cloud environments and firewall issues can be solved, still license management and changing IP addresses may be a problem in an elastic cloud environment. An automated run-time configuration that passes IP addresses of dynamically created *Virtual Machine* (VM) instances to a test suite solves the latter problem. Also cloud providers seem to have identified this latter problem and start to provide solutions such as Amazon's VPC.

License issues of commercial TTCN-3 execution environments for distributed testing have been avoided by using the open-source Java test framework Jata. Experience with Jata has shown that while it has some idiosyncrasies it is applicable for implementing and executing distributed tests based on proven CTMF and TTCN-3 concepts.

Now that Amazon has introduced Elastic Network Interfaces, future work seems worthwhile to investigate whether these can be used to address license management problems and investigate the applicability of commercial TTCN-3 tools in a cloud environment.

7. ACKNOWLEDGMENTS

This paper is based on experience from earlier work in testing applications in static grid computing environments [13] by Thomas Rings, Helmut Neukirchen, and Jens Grabowski. The Turnip cloud application used in this case study was provided by Ragnar Skúlason [14] who was supervised by Klaus Marius Hansen and Helmut Neukirchen.

8. REFERENCES

[1] Amazon Web Services LLC. Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`, 2013.

[2] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, University of California, Berkeley, 2009.

[3] W. Chan, L. Mei, and Z. Zhang. Modeling and Testing of Cloud Applications. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*. IEEE, 2009.

[4] ETSI. ETSI Standard (ES) 201 873-1 V4.5.1: The Testing and Test Control Notation version 3; Parts 1-10. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2013.

[5] Eucalyptus Systems, Inc. Eucalyptus. `http://www.eucalyptus.com`, 2013.

[6] J. Gao, X. Bai, and W. T. Tsai. Cloud-Testing- Issues, Challenges, Needs and Practice. *Software Engineering: An International Journal (SEIJ)*, 1:9–23, 2011.

[7] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An introduction to the testing and test control notation (TTCN-3). *Comp. Netw.*, 42(3):375–403, 2003.

[8] ISO/IEC. Information Technology – Open Systems Interconnection – Conformance testing methodology and framework. International ISO/IEC multipart standard No. 9646, 1994-1997.

[9] Jata. `http://code.google.com/p/jata4test/`, 2013.

[10] P. Mell and T. Grance. The NIST Definition of Cloud Computing. National Institute of Standards and Technology: NIST Special Publication 800-145, 2011.

[11] OSGi Alliance. `http://www.osgi.org`, 2013.

[12] T. Rings, J. Grabowski, and S. Schulz. A Testing Framework for Assessing Grid and Cloud Infrastructure Interoperability. *Int. J. On Advances in Systems and Measurements*, 4(1 & 2), 2011.

[13] T. Rings, H. Neukirchen, and J. Grabowski. Testing Grid Application Workflows Using TTCN-3. In *Int. Conf. on Soft. Testing Verification and Validation (ICST)*. IEEE, 2008.

[14] R. Skúlason. Architectural Operations in Cloud Computing. Master's thesis, Faculty of Industrial Engingeering, Mechanical Engineering and Computer Science, University of Iceland, Reykjavík, Iceland, 2011.

[15] Sunflow. `http://sunflow.sourceforge.net`, 2013.

[16] H. Tómasson. Distributed Testing of Cloud Applications Using the Jata Test Framework. Master's thesis, Faculty of Industrial Engingeering, Mechanical Engineering and Computer Science, University of Iceland, Reykjavík, Iceland, 2011.

[17] J. Wu, L. Yang, and X. Luo. Jata: A Language for Distributed Component Testing. In *15th Asia-Pacific Soft. Eng. Conf. (APSEC 2008)*. IEEE, 2008.