International Journal of Advance and Applied Research
www.ijaar.co.in
ISSN – 2347-7075 Impact Factor – 7.328
Peer Reviewed Bi-Monthly
Vol.11 No.1 September – October 2023

Cloud-Based Testing Frameworks For Distributed Systems

Sumathi Rajkumar
Assistant Professor,
Department of Computer Science,
Thakur Ramnarayan College of Arts and Commerce, Mumbai, Maharashtra, India
Corresponding Author - Sumathi Rajkumar

DOI - 10.5281/zenodo.10947428

Abstract:

The purpose of this work is to introduce the framework known as Cloud Testing, which is a solution that allows for the execution of a test suite to be parallelized across a distributed cloud foundation. When compared to more conventional approaches, the use of a cloud as a runtime environment for automated software testing offers a solution that is both more efficient and effective in terms of the investigation of variety and heterogeneity for testing coverage. With the help of this study, we want to assess our solution in terms of the performance benefits that were accomplished through the use of the framework. This evaluation will demonstrate that it is feasible to enhance the software testing process while incurring very little configuration overhead and minimal expenses.

Keywords: Software Testing, Cloud Computing

Introduction:

If you want your software testing process to be successful, it has to be carried out swiftly and automatically. There are solutions that have been around for a long time [2] that are designed to automate the process of software testing. Additionally, there are other solutions that are primarily geared at accelerating the process by spreading the execution of a test suit among a group of processors [4] [3]. In the same vein, there are also efforts being made to investigate the properties of distributed computing platforms, such as grids, as well as their broad parallelism and great variability of settings, with the goal of minimising the impact of the development environment on the outcomes of tests [5]. On the other hand, cloud computing has just lately been used by new research as a platform for testing software on a big scale. "6" and "7"

When compared to more conventional approaches, the use of cloud computing platforms for the purpose of conducting software testing may result in considerable improvements in terms of both the efficiency and efficacy of testing capabilities [15]. This assertion is supported by a number of aspects, including the reduction of costs associated with deployment, maintenance, and licencing environments; the flexibility to acquire and install customised test environments on demand; and the capacity to scale in a fast and cost-effective manner [14].

On the other hand, the process of development and testing in the cloud often requires a large amount of work in the setup, distribution, and execution of tests [6]. We offer a framework that we term CloudTesting in order to make it easier to investigate different cloud computing platforms and settings for the purpose of software testing. An abstraction solution that may facilitate its adoption is the fact that the tool does not require any source code modification in order to execute software tests in the cloud. This is one of the ways in which our solution enables parallel execution of automated software tests in heterogeneous environments, thereby reducing the amount of time spent during the testing process.

In order to assess our solution, we carried out a series of experiments using the resources that were made available to us by Amazon EC2. These experiments were compared to the execution of the identical tests that were carried out locally. utilising the cloud infrastructure results in considerable improvements in execution time, with very little setup overhead and extra expense, according to the quantitative study that was done utilising the cloud infrastructure.

The Cloud Testing Frame Work:

The number of test cases that are often included in big software projects is typically much higher than average [5]. Because these tests often need a significant amount of time to execute, the usage of agile development processes that primarily depend on automated testing, such as Extreme Programming [8], is made more difficult at times. A huge parallelization of the execution of the tests is the only method to reduce the amount of time spent on the testing process [9]. This is because each test requires a certain amount of time to run, and the amount of time spent testing might vary depending on the size and complexity of the programme.

The Cloud Testing Framework makes this process more efficient by encapsulating all of the complexity required in the parallel execution of test cases utilising on-demand computing resources. This is accomplished without the need for any modifications to be made to the source code of the tests in order to make use of the framework.

When it is chosen to distribute and parallelize the tests, the outcome is a considerable decrease in the amount of time necessary to run a big test set. This, in turn, reduces the amount of time spent discovering and resolving problems, which has a major influence on the overall cost of development. Furthermore, the framework improves the reliability of the test findings by using settings that are diverse and free of contamination while the tests are being performed. This makes it easier to uncover faults that would otherwise remain hidden until the software production stage.

Initially, the framework is mostly concerned with software that is written using Java. By conducting a reflection on the local classes that contain the tests and scheduling the execution of each test on a separate computer in the cloud, it distributes the execution of a collection of unit tests. This is accomplished by distributing the execution of the tests. An implementation of the Round-Robin algorithm [10] is used to ensure that the load is distributed evenly among all of the computers that are available. This ensures that each and every request is spread uniformly over all of the computers that are a part of the test infrastructure.

Fig. 1 displays the architectural components of the CloudTesting framework, which are comprised of the configuration, reflection, distribution, connection, log, and main components. This is a very essential element of the proposed framework.

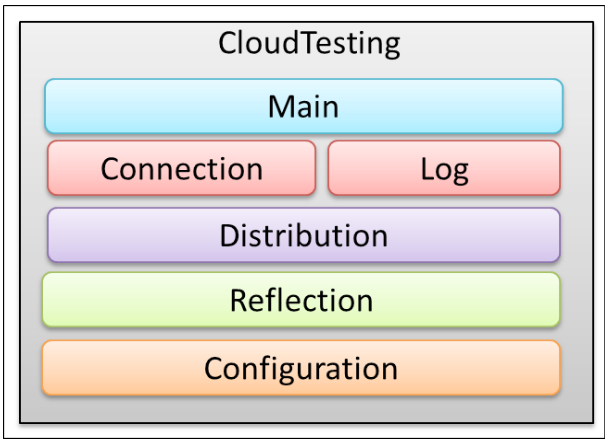

Fig. 1. Cloud Testing components

Additionally, the configuration component provides assistance in the definition of information pertaining to load balancing, hosts, and pathways. For example, it may be used to define the local storage space for test results, the libraries that need to be sent to the cloud in order to ensure that the test is executed correctly, and the permissions for accessing files on the cloud provider. In addition to that, it contains the settings for the load balancer as well as the list of machines that are accessible for test execution at any given specific time. It is the responsibility of the Reflection component to extract the test cases in order to provide the distribution component with information on the examination procedures that should be carried out in the cloud.

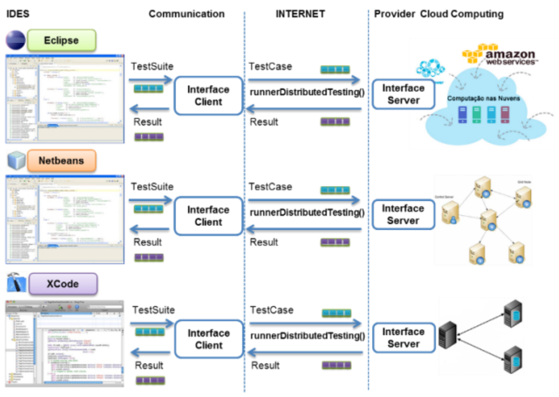

Fig. 2. Cloud Testing intermediating the distributed execution of automated tests on different parallel infrastructures

An example of the distribution component that is being used to facilitate the execution of test suites across a parallel infrastructure can be seen in Figure 2. It is necessary to expand the framework in order to include certain plugins in order to make it compatible with a particular integrated development environment (IDE) and parallel infrastructure. Plugins for the Eclipse integrated development environment (IDE) and the Amazon Web Services architecture are provided by the present implementation.

There is an interface on the client side that is provided by the connection component, which allows for communication with the cloud provider. A service that handles the execution of each test and transmits the results of the tests back to the client in real time is provided by this component, which is located on the cloud site. In the process, the log component is responsible for recording events that occur.

Related Work:

It has been noticed that over the course of the last few years, a number of well-founded studies have been generated that approach methods and means to automate and speed the process of software testing [2, 4], [3]. Despite this, the amount of labour required for software testing increases in proportion to the size and complexity of the computer systems. Since this methodology takes advantage of the characteristics of wide parallelism and extensive heterogeneity of environments, numerous automatic distributed software testing systems or large scale systems have been proposed in recent years. This is because the goal of this methodology is to limit the effects of the development environment on the test results [17]. The following discussion will focus on a few research that are relevant to our solution.

An open-source solution for automatically performing unit tests inside the grid is presented by the GridUnit tool [17], which studies the usage of computational grids as a testing environment and delivers the solution. The solution is an extension of the JUnit framework [2], which enables the execution of a JUnit test suite to be distributed throughout the grid without the need for any modifications to be made to the source code. Grid Unit has five primary qualities in its design. The distribution is both transparent and automated; each JUnit test is considered to be an independent job, and the scheduling of the execution of the task in the grid is carried out without any involvement from a human administrator. For the purpose of avoiding contamination, each test is executed by using the virtualization capabilities provided by the grid. This ensures that the results of the A1 test do not influence the results of the A2 test. In order to test load distribution, the tool has a Grid Scheduler, which is responsible for managing load distribution. In order to ensure the integrity of the test suite, each JUnit test is executed as a separate job, as was indicated before. Grid Unit generates a new instance of the Test class for each new test, then immediately destroys the object after making calls to the set Up (), test Method (), and tear Down () methods; As a way of creating the execution and monitoring of tests in a centralised manner, the GridTestRunner and Grid Test Listener interfaces provide the ability to govern the execution of tests.

In terms of the infrastructure and the abstraction of complications, our approach, which is called Cloud Testing, is very different from GridUnit. GridUnit makes use of computing grids, and Cloud Testing allows for the use of several distributed execution platforms in order to carry out automated testing of an application in a number of runtime settings, including the cloud. Using the cloud has several benefits, including the capacity to automatically resize virtualized hardware resources, the provision of security via virtualization, the elimination of concerns over workflow, the ease of administration, the usability, and the flexibility of the business model used.

A solution is proposed in the D-Cloud study [9] for testing parallel or large-scale distributed systems that need features of highly reliable systems. The method focuses on fault tolerance testing at the hardware level. A controller node is responsible for managing all of the hosted operating systems, and a frontend is responsible for controlling hardware and software configurations as well as test scenarios. The research presents the infrastructure of cloud computing for software testing, which is comprised of multiple nodes of virtual machines that run operating systems hosted with fault injection. D-Cloud's conceptual architecture draws attention to the following characteristics among its components:

- A virtual machine that is equipped with fault injection is known as the FaultVM, and it is based on QEMU, which is the hypervisor software.
- Management of computational resources via the usage of Eucalyptus - The Eucalyptus software is used in order to handle the vast quantity of resources that are utilised in the cloud management process. The tester is relieved of the task of managing the allocation of computer resources as a result of this procedure being carried out automatically;

- Testing and configuration of the system that is automated - the tool is able to automate the process of testing and configuration of the system, including the injection of faults, depending on scenarios that are provided by a tester;
- The preparation of the test scenarios is accomplished by means of a file that is written in XML. By supplying numerous scenario files, it is possible to test various systems simultaneously.

When presented with the concept of Cloud Testing, the D-Cloud work takes a different approach to the execution of automated software testing and approaches the environment in which it is carried out. Essentially, there is a divergence in the path that automated software testing is taking. On the other hand, the second study leads to the execution of a series of unit tests that make use of the JUnit framework. The first project directs its tests for fault tolerance at the hardware level. Additionally, D-Cloud was developed to function just for the infrastructure of cloud computing, neglecting other platforms and modes of execution in the process. Due to the fact that it is a framework, Cloud Testing may be modified to meet certain requirements.

Experimental Results:

One of the capabilities that may be added to the Cloud Testing framework is the ability to collect resources from a variety of execution platforms and to utilise it in a variety of integrated development environments (IDE). To conduct this research, however, we created an instantiation of the framework for the Eclipse integrated development environment (IDE) and the cloud provider Amazon Web Services (AWS).

For the purpose of carrying out the experiments, we developed a collection of 1800 tests, each of which had an average processing time that was previously known when it was carried out on a local computer. Our objective was to draw a comparison with the findings that were acquired via the use of the framework.

The experiments are broken up into two different scenarios: (1) the first scenario involves the test suite being run 45 times on a local system, and (2) the second situation involves the test set being disseminated 45 times using resources made available by the cloud provider.

For the purpose of eliminating outliers, we made use of Chauvenet's criteria [13] when conducting the study. Next, determine the average execution time, the standard deviation, and the greatest and worst execution times. Finally, identify the average execution time. This information allows the calculation of the speedup (SP = T1/Tp where T1 is the execution time of the sequential programme and Tp is the execution time of the same programme running in parallel) and the efficiency of the parallel execution (EF = Sp / Np where Sp is the speedup achieved and Np is the number of cloud machines used to run the tests in parallel). We then proceeded to compute the confidence intervals for 95% and 99% with reference to [12].

Scenario 01:

The tests that were carried out locally adhered to a stringent protocol on the use of the apparatus during the duration of the test. For the purpose of avoiding abnormal outcomes and obtaining true results, we used a system that was only devoted to the testing process. We restarted the system after each test was completed in order to clean the data that was stored in the random access memory (RAM) and the cache of the processor. The computer was equipped with a 32-bit Linux operating system, a 2 GHz Intel Core 2 Duo processor, and 4 gigabytes of random access memory (RAM).

For the purpose of capturing the actual runtime of the test, we used the integrated development environment Eclipse, which is equipped with the JUnit plugin PDE. This plugin maintains all software that is needed for unit testing and includes a default profiler.

Every single unit test was able to obtain an average runtime that was very close to one second in this circumstance. It took an average of thirty minutes to complete a single run of the test set, which consisted of more than one thousand and eight hundred tests. It took a total of 22 minutes and 54 seconds to complete all 45 rounds. With a standard deviation of 1.47% (0:00:27), the average execution time was 0:30:33, while the best and worst execution times were 0:29:58 and 0:31:09 respectively. When using a confidence interval with a 95% level of certainty, the lower limit and higher limit were 00:30:25 to 00:30:41. When using a confidence interval with a 99% level of certainty, the lower limit and upper limit were 00:30:23 respectively. These statistics serve as a foundation for doing an analysis of the speedup in which the CloudTesting framework is used.

Scenario 02:

In order to determine how long it takes for the tests to run in the cloud, we need to take into account a number of aspects, including network latency and volatility. Therefore, in order to ensure theoretically identical bandwidth circumstances for all of the experiments, we decided to establish a similar time scale for the testing. During the time span between 00:00 and 04:00, all of the tests were carried out. This time slice was chosen since it was chosen to mimic the minimal network utilisation that occurred in the laboratory. Because of the extensive quantity of testing that was required, the tests were not carried out on a single day; rather, they were carried out on several days in accordance with the policy that is detailed below.

Using three distinct Amazon instance types—micro, small, and medium—the 45 cycles were repeated three times until they were completed. When carrying out the three different subsets of trials, we used a total of 18 instances of each kind. This was accomplished by using the Eclipse integrated development environment in conjunction with the TPTP plugin profiler in order to record the real runtime of the test. For the download, we used a network that had a nominal capacity of 15 Mbps, and for the upload, we utilised 1 Mbps. A number of configuration settings were applied to the cloud machines before the tests were carried out. These settings included the following: (1) the creation of the log and lib directories for the purpose of storing logs and libraries, respectively; (2) the distribution of the JUnit and CloudTesting libraries to the lib directory; and (3) the activation of the CloudTesting remote service.

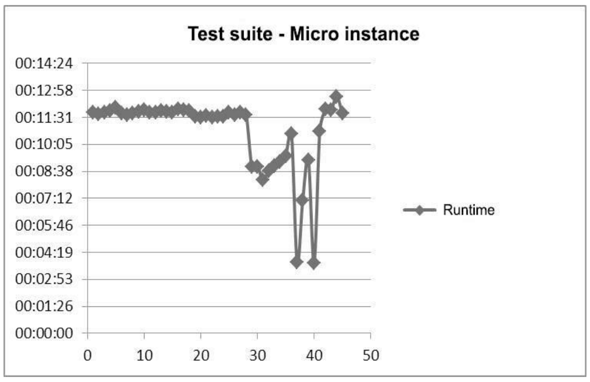

Fig. 3. Execution time for the Scenario 2 – Micro instances

Discussion:

Before deciding if a given speedup is a favourable or bad thing, there are a few things that need to be taken into consideration. Since we employed 18 machines to carry out the tests, one would anticipate that the results would be 18 times quicker than if they had been carried out by a single machine. However, it is essential to keep in mind that in order to parallelize the execution by using cloud computers, we are required to upload the code that will be run remotely using cloud computing. It is also possible that the cost of spreading packets over the network will be high, depending on the scale of the project. The capacity of the computers to process data and the capacity to input and output data in the cloud are two additional factors that are intimately tied to one another. Last but not least, there is also the connection with the virtualized server, which is important to consider since the performance will often be determined by the quantity of resources that are accessible on the actual server.

Despite the fact that the micro instances were not designed to handle big load requests in a short amount of time, we were able to see that they were able to accomplish a significant increase in speed throughout the testing. When it comes to this particular circumstance, the medium examples are more suitable.

In the best case scenario, the trials that were carried out with the micro instances demonstrated a speedup of 8.55 times and a parallel efficiency of 0.48. In the worst case scenario, the speedup for the micro instances was 2.61 times, and the efficiency was 0.14. Additionally, it demonstrated a speedup of 2.89 percent on average, which led to a parallel efficiency of 0.16 percent.

A striking regularity was seen in the tests that were carried out using instances of type tiny. These tests demonstrated a speedup of 5.71 times and an efficiency of 0.32 in the best case scenario, and 5.70 and 0.31 in the worst case scenario. In general, the outcomes are almost identical to what would be expected in the best-case scenario.

The studies that were carried out with the medium instances achieved a speedup of 9.48 times and a parallel efficiency of 0.53 in the best case scenario, while in the worst case scenario, the speedup for the medium instances was 8.72 times and the parallel efficiency was 0.48. We were able to get a speedup of 7.83 percent on average and a parallel efficiency of 0.43 percent. A summary of these findings may be seen in Figure 4.

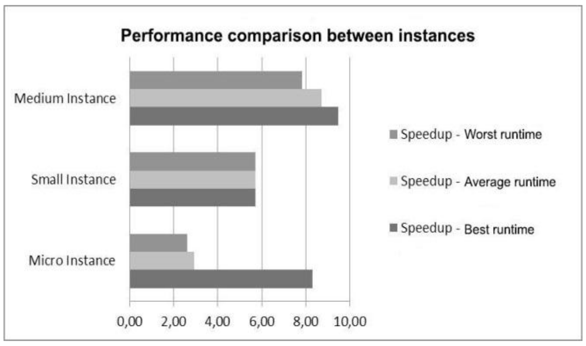

Fig.4. Comparison Speedup between micro, small and medium instances

During several of our tests, we made the interesting discovery that we were able to get greater speedups with the micro instances, which were less expensive, than we were with the tiny instances.

At first, this result came as a surprise; however, as was noted before, micro instances have the ability to temporarily employ up to two ECUs, which means that they have double the computational capability of a small instance. Micro instances, on the other hand, are considered to be much slower than tiny instances on average.

According to what was anticipated, the medium instances that were purchased at a greater price yielded the greatest outcomes. This outcome was anticipated as a consequence of the hardware arrangement, as well as the superior input and output rates in comparison to the other examples that were examined.

Conclusions:

The results of the experiments indicate that there are significant performance gains associated with the distribution of the execution of software tests. These gains are achieved without a significant increase in the costs involved in assembling the infrastructure. As a result, the process of using cloud infrastructures as a platform for automated software testing is made easier.

By performing parallel automated software tests in diverse settings via an abstraction layer for users, the Cloud Testing framework makes it easier to execute automatic tests in dispersed environments. This results in improvements in speed, reliability, and the ease with which configurations may be made.

References:

[1]. Smith, A. and Jones, B. (1999). On the complexity of computing. In *Advances in Computer* Science, pages 555-566. Publishing Press.

[2]. Gamma, E. and Beck, K. (1999). JUnit: A cook's tour. Java Report, 4 (5) :27-38.

[3]. Hughes, D. and Greenwood, G. (2004). A Framework for Testing Distributed Systems, In Proceedings of the 4th IEEE International Conference on Peer-to-Peer computing.

[4]. Kapfhammer, G., M. (2001). Automatically and transparently Distributing the Execution of Regression Test Suites. In Proceedings of the 18th International Conference on Testing Computer Software.

[5]. Duarte, A. *et. al.* (2005). GRIDUNIT: software testing on the grid, Proceedings of the 28th international conference on Software engineering. New York, USA, p. 779-782.

[6]. Hanawa, T. *et al.* (2010). Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems, Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference.

[7]. Oriol, M. and Ullah, F. (2010). YETI on the Cloud. Software Testing, Verification, and Validation Workshops (ICSTW) Third International Conference 2010.

[8]. Wu, X. and Sun, J. (2010). The Study on an Intelligent General-Purpose Automated Software Testing Suite, Intelligent Computation Technology and Automation (ICICTA) International Conference 2010.

[9]. Banzai, and Takayuki Koizumi, Hitoshi (2010). D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology Cluster, Cloud and Grid Computing (CCGrid), 10th IEEE / ACM International Conference.

[10]. Ramabhadran, S. and Pasquale, J. (2003). "Stratified round robin: A low complexity packet scheduler with bandwidth fairness and bounded delay," Proc. of SIGCOMM.

[11]. 'AmazonEC2 (2012). "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/pt/ec2/instance-types/,June.

[12]. Dillard, GM (1997). "Confidence intervals for power Estimates," Signals, Systems & Computers, 1997. Conference Record of the Thirty- First Asilomar Conference.

[13]. Pop, S.; Ciascai, I. and Pitica, D. (2010), "Statistical analysis of experimental data Obtained from

the optical pendulum," *Design and Technology in Electronic Packaging (SIITME), 2010 IEEE 16th International*Symposium.

[14]. Grundy, J. *et al.* (2012), "Guest Editors' Introduction: Software Engineering for the Cloud, "Software, IEEE, vol.29, no.2, pp.26-29.

[15]. Riungu-Kalliosaari, L.; Taipale, O. and Smolander, K. (2012). "Testing in the Cloud: Exploring the Practice," Software, IEEE, vol.29, no.2, pp.46-51.

[16]. Andrade, Nazareno *et al.* (2003). OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pp. 61-86.

[17]. Duarte, A. et al.. Multi-environment Software Testing on the Grid. In: PADTAD ’06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging. New York, NY, USA: ACM, 2006. p. 61–68. ISBN 1-59593-414-6.