YETI on the Cloud

Manuel Oriol
Department of Computer Science
University of York
York, United Kingdom
Email: manuel@york.ac.uk

Faheem Ullah
Department of Computer Science
ETH Zurich
Zurich, Switzerland
Email: faheem@inf.ethz.ch

***Abstract*—The York Extensible Testing Infrastructure (YETI) is an automated random testing tool that allows to test programs written in various programming languages. While YETI is one of the fastest random testing tools with over a million method calls per minute on fast code, testing large programs or slow code – such as libraries using intensively the memory – might benefit from parallel executions of testing sessions. This paper presents the cloud-enabled version of YETI. It relies on the Hadoop package and its map/reduce implementation to distribute tasks over potentially many computers. This would allow to distribute the cloud version of YETI over Amazon's Elastic Compute Cloud (EC2).**

***Keywords*-Software Testing; Distributed Systems**

I. Introduction

The York Extensible Testing Infrastructure (YETI) is a language agnostic random testing tool. Its reference implementation tests Java programs and is able to perform over one million method calls per minute on fast library code. YETI found bugs in the `java.lang`, iText[1] and many more programs. YETI, however, suffers from two issues:

1) **Performances.** On slow code, the performances can drop as low as 10^3 method calls per minute.[2] While this might seem to be reasonable performances, experience shows that a plateau in the number of faults found for a given class is generally only reached after 10^5 method calls. For large projects (100+ classes) this might mean that testing code on a single computer might not reach a stable point overnight. It also has the advantage of using multiple seeds for the pseudo-random number generator, which previously showed better performances than using only one seed over long running sessions [1].
2) **Security.** The Java binding of YETI relies on the Java security model to forbid undesirable side-effects. This is a reasonable assumption in most cases. It is however sometimes undesirable because some programs need to access files, sockets, or drivers directly. It is also not possible to make such assumptions in case YETI is testing programs written in C.

[1]http://itextpdf.com/

[2]Observed while testing the iText library in September 2009

Distributing YETI over the cloud solves these two issues: (1) testing classes in parallel allows to distribute the testing of 100 classes on 100 cores, resulting in a 10mn test rather than a 1000-minute one; (2) testing classes in a dedicated virtual machine, devoid of sensitive information, means that security restrictions can be relaxed.

This article presents a prototype of the cloud implementation of YETI. The prototype relies on the map/reduce implementation of Hadoop to distribute testing sessions on remote machines and recombine them at the end of a desired testing time.

Section II present the cloud implementation of YETI. Section III presents the future developments of YETI on the cloud. Section IV describe related work. We eventually conclude in Section V.

II. Architecture and implementation

A. YETI standalone application

YETI is an automated random testing tool for Java. A YETI testing session consists in a sequence of calls made to methods at random using arguments generated at random. The oracle for the tests are either contracts –when available – or runtime exceptions/failures.

When a failure is detected the logs of the testing session are minimised to produce test cases that reproduce the failure. Such unit tests can then be stored and be executed later by a unit test framework.

YETI is usually launched on the command-line. A typical call to YETI is:

```
java yeti.Yeti -Java -yetiPath=.
          -time=10mn -randomPlus
          -testModules=String:StringBuilder
```

The options used on this command-line have the following meaning: `-Java` indicates that the tested program is in Java, `-yetiPath=.` indicate that classes in the current directory (and its subdirectories) will be preloaded, `-time=10mn` indicates that the testing session will last 10 minutes, `-randomPlus` indicates that the strategy random+ [2] will be used, and `-testModules=String:StringBuilder` indicates that both `String` and `StringBuilder` will be tested.

While testing, traces of faults found are output to the terminal. For example:

```
Exception 5
java.lang.NullPointerException
 at java.lang.String.replace(String.java:2207)
```

At the end of the testing sessions, YETI outputs generated test cases reproducing the faults found during the testing session as well.

Note that it is possible to avoid the overhead of keeping the traces in the system (and calculating the minimal test cases) by specifying `-nologs` to throw away all logs except exception traces, or `-rawlogs` to output the logs to the terminal. This comes at the cost of not being able to generate test cases reproducing the failures, but still provides the exception traces. In an exploratory phase of the testing, this is generally the way to use YETI.

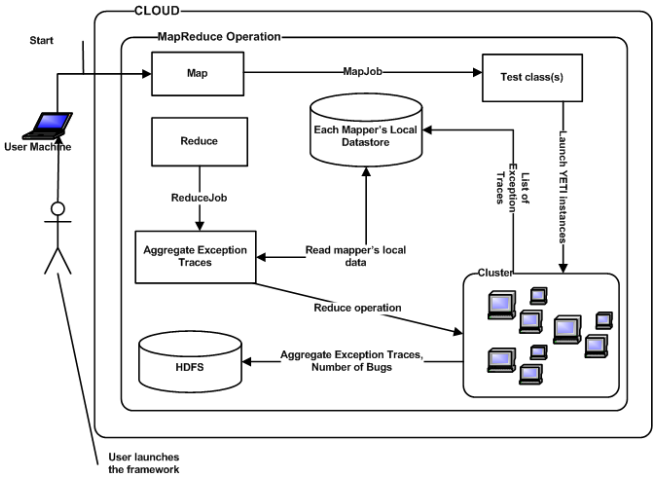

Figure 1. YETI cloud architecture.

B. Architecture of the cloud implementation

As figure 1 shows, the main architecture of YETI on the cloud relies on Hadoop, Apache's implementation of Google's map/reduce framework.

The main idea is that we use a very simple approach where testing jobs are described by calls to YETI stored in normal text files by the user, that will each launch YETI in standalone mode. Commands from within each file are then mapped to different machines in the cloud.

After setting up the map/reduce cluster. These text files and YETI (bundled as a jar file) are then uploaded to the distributed file system (DFS) on the map/reduce cluster master, which then launches individual testing machines for each text file executing the commands contained within the file in the order they appear.

At the end of the testing session, all exception traces are written back to the DFS during reduce step, which can then be downloaded to the user machine and made available to the software tester for evaluation.

C. Evaluation

We only ran preliminary evaluations.

We evaluated our approach using the Amazon Elastic Computing Cloud (EC2) [3]. We performed 5 jobs of 20-minutes each (a total testing time of 100 minutes) in less than 21 minutes, outputting comparable results with YETI's standalone execution. Test cases all tested classes from `java.lang`. The number of faults found in each tested class in the distributed mode were the same as those found during standalone execution of YETI.

By using the cloud we clearly improved the performances of YETI and reduce the testing time by employing more machines and distributing the jobs. Distributing YETI over the cloud requires uploading the files containing typical YETI calls for launching YETI in standalone mode(multiple YETI calls can be placed in a file resulting in the same machine testing several programs or the same program with potentially specific options/strategies for each machine), and YETI itself (bundled as a Jar along with the classes to be tested). Setting up a Hadoop cluster on EC2 and uploading all the required files to the MapReduce Master then takes under a few minutes. However, setting up the EC2 cluster for the first time might take some extra time (up to 10 minutes in our tests), but subsequent testing sessions require less than a few minutes to start. The execution can be performed on a locally set up Hadoop cluster as well. While we did not run experiments with more open security models, YETI jobs were run on one-shot Ubuntu virtual machines, which solves the potential security issues for random testing.

III. Future Work

While this first experiment with YETI on the cloud was quite conclusive, there are many areas that we would like to improve in the future. In the next paragraphs, we explain some of these areas and what in our opinion we would need to do.

Define automated mappings for jobs: It is still unclear what the best mapping for testing sessions would be for large amount of classes. So far, experiments have shown that testing classes independently is not the best way to uncover faults. In particular, testing two classes that are used together (such as `String` and `StringBuilder`) uncovers faults that are not uncovered when tested independently. Grey-box testing might help in finding out which code is dependent on each other and would lead to better understanding of this phenomenon to make the best mappings for testing sessions.

[3] http://aws.amazon.com/ec2/

Define adequacy criteria for distributed random testing sessions: In all tests that were performed it seems that the testing sessions achieve a plateau if parameters of the testing session are unchanged. Being able to detect such plateaus would lead to stopping the testing sessions when YETI does not have high chances to discover further faults. It could then lead to a better way of mapping jobs on a low number of nodes.

Testing programs working over the cloud: One of the issues with programs working on the cloud is that testing such programs requires to run tools that are themselves running on the cloud. YETI could be one of these testing tools.

Real-time feedback on the distributed testing sessions: One of the most valuable aspects of YETI is its graphical user interface [3] which shows information about the testing session in real-time. The map/reduce middleware does not however allows for sending results in real-time back to the master. This implies that a YETI on the cloud would benefit from a more flexible middleware such as Multicast Objects [4].

IV. Related Work

Distributing software testing over multiple computers has received very little attention in the research community. Starkloff [5] presents the general advantages of parallelizing software testing systems and how to achieve parallelism by distributing execution.

Lastovetsky and Alexey [6] present a case for testing of a distributed programming system where serial execution of a regression test suite would on the average take at least 90 hours per bug detected, whereas Parallelizing the execution on a number of local UNIX computers resulted in a speed up of up to 7.7 on two 4-processor workstations, which meant the test suite could be run multiple times in a single day.

Further work resulted in tools like Joshua [7], and GridUnit [8]–[10]. Joshua uses Jini for distributing the execution of a regression test suite across a number of CPUs and writes the results back to a centralized repository, whereas GridUnit employs a computational Grid to speed up the testing process. A single master distributes test cases for execution across machines in a Grid and then waits for the results. Both Joshua and GridUnit extend the JUnit testing framework. Using the same principles, Yao *et al.* [11] present the implementation of a grid-based unit testing framework and present some preliminary results based on simulation. The framework differs from Joshua and GridUnit in its support for NUnit and dbUnit apart from JUnit, extending its support for C# and database-driven projects.

Some of the limitations of such approaches, however, include the requirement that a test case be manually created beforehand, and their execution on either a local LAN or a Grid. A LAN has limited number of machines whereas, in a Grid, nodes might have different computational capacities varying from supercomputers to a low-end personal computer [11]. This would seriously impact the performance of some test cases and work would have to be assigned according to the workers capacity.

The only approach that is somewhat similiar to our implementatin for distributing execution is by Parveen *et al.* [12], where a prototype distributed execution framework for JUnit test cases called HadoopUnit is presented. They also use the MapReduce primitives for distributing test cases. Before execution can begin, an Ant task is run to produce input for the map function from uploaded files and store them in a file to be later read by the Mapper. The Reducer then collects the test case results from each Mapper and outputs them to a file on the DFS. A 150-node cluster leads to a speed-up of 30x in execution time. HadoopUnit, however, only supports execution of JUnit test cases and requires them be created manually beforehand. YETI creates test cases on the fly, executes them against the target code and reports bugs. Since the test cases are created automatically, each node has the same processing power and new nodes can be added dynamically, if more processing power is required.

V. Conclusions

In this article, we introduced a cloud version of the York Extensible Testing Infrastructure (YETI). YETI is a langage agnostic random testing tool for which the reference implementation tests Java programs.

The cloud implementation of YETI allows to solve performances issue when testing large programs. It also allows to perform testing on clean Virtual Machines, solving potential security issues. We performed limited experiments to validate our approach.

Acknowledgment

We thank Julian Friedman for his valuable input.

References

[1] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer, "On the predictability of random tests for object-oriented software," in *International Conference On Software Testing, Verification And Validation (ICST 2008)*, July 2008.

[2] I. Ciupa, M. Oriol, B. Meyer, and A. Pretschner, "Finding faults: Manual testing vs. random+ testing vs. user reports," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2008.

[3] M. Oriol and S. Tassis, "Testing .net code with yeti," in *ICECCS*. IEEE Computer Society, 2010.

[4] P. Eugster and M. Oriol, *Implementing Multicast Objects with Interaction Types*, 2008, http://www.cs.purdue.edu/homes/peugster/DOP/MObjects.pdf.

[5] E. Starkloff, "Designing a parallel, distributed test system," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 16, no. 6, pp. 3 –6, jun 2001.

[6] A. Lastovetsky, "Parallel testing of distributed software," *Inf. Softw. Technol.*, vol. 47, no. 10, pp. 657–662, 2005.

[7] G. M. Kapfhammer, "Automatically and transparently distributing the execution of regression test suites," in *In Proceedings of the 18th International Conference on Testing Computer Software*, 2000.

[8] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado, "Gridunit: software testing on the grid," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 779–782.

[9] A. Duarte, G. Wagner, F. Brasileiro, and W. Cirne, "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, pp. 61–68.

[10] R. N. Duarte, W. Cirne, F. Brasileiro, P. Duarte, and D. L. Machado, "Using the computational grid to speed up software testing," in *Proceedings of 19th Brazilian Symposium on Software Engineering*, 2005.

[11] Y. Li, T. Dong, X. Zhang, Y. duan Song, and X. Yuan, "Large-scale software unit testing on the grid," may 2006, pp. 596 – 599.

[12] T. Parveen, S. Tilley, N. Daley, and P. Morales, "Towards a distributed execution framework for junit test cases," sept. 2009, pp. 425 –428.