CUT: Cloud Unit Testing

Demonstration Walk Through

Alessio Gambi, Sebastian Kappler, and Andreas Zeller
Saarland University, Germany

This appendix describes the demonstration of our tool CUT— Cloud Unit Testing. The first section describes the general setup of the tool and the testbed that we use to run the demonstration. The following sections describe the various experiments to showcase CUT's main features. The last section report the list of command line options to operate the tool.

A Environment and Tool Setup

The environment for our experiments consists of two machines: a local machine, which represents the developers workstation and from which unit tests execution triggers, and a remote machine, which hosts Docker containers. In this setup, Docker container simulate virtual machines in the cloud. However, CUT works the same with other clouds, such as OpenStack and Amazon EC2.

The local machine contains the source code of the application and the unit tests as well as it hosts all the dependencies required for building the application and its tests. The remote machine runs the Docker software, the Docker containers, and ActiveMQ, which is required by CUT's components to communicate. CUT's main process runs on the local machine, and the workers (i.e., test runners) run on the remote virtual machines. A custom Docker image hosts the worker's code and automatically starts the worker process at startup.

Since CUT is a command line tool and outputs messages on the console, we capture screenshots of the console on the local machine to illustrate the progress of each experiment. Messages from remote test runners are prepended with the ip address of the virtual machine which run them. Additionally, we show the output of the `docker ps` command on the remote machine to show the running Docker containers that CUT starts.

CUT is designed as a maven plugin, so to enable it is enough to specify the right configuration parameters in the project pom file: in particular, it is enough to change the version numbers of surefire plugin to `2.19.2-parallel` and of JUnit to `4.12-parallel`. All the other options are specified on the command line. Appendix B gives an overview of all the parameters to customize CUT's behavior.

Experiment 1: Parallelizing Tests Execution in the Cloud

This experiment shows the ability of CUT to parallelizing and distributing the execution of unit tests over a set of virtual machine in the cloud. To perform this experiment we use an open source project, tachyon, and run the unit tests of its core module. We show the improvement in the efficiency of test execution by comparing the test execution time for running the tests using only the local machine and using the cloud.

Setup:

- Checkout the `core` module of tachyon at commit `694aefdfeb5`.
- Enable CUT as described in Appendix A.
- Start Docker on the remote host (enabling its Web interface `-H` option) .

Standard Execution:

- Command line:
  `mvn clean test`
  . `-Drat.ignoreErrors=true -Danimal.sniffer.skip=true` # Required by Tachyon
  . `-Dprovider=regular` # We do not use the cloud for this run

- The console output shows the number of passing, failing and skipped tests as well as the test execution time

```
Total time: 19 min, 40 sec

Results :

Tests run: 222, Failures: 0, Errors: 0, Skipped: 0
```

Execution with CUT:

- We configure CUT to parallelize the execution of unit tests on four Docker containers.

- Command line:
  `mvn clean test`
  . `-Drat.ignoreErrors=true -Danimal.sniffer.skip=true` # Required by Tachyon
  . `-Dprovider=cloud -DnumHosts=4` # Use 4 cloud virtual machines
  . `-Djcloudscale.configuration=parallelize.runner.JCSDockerConfig` # Use Docker

- The console output shows the setup operations of CUT: (a) CUT loads the configurations, (b) checks the connectivity with the cloud, and (c) starts the remote hosts. At this point, CUT starts the elaboration of the unit tests: it (d) collects and (e) clusters them, and finally, (f) schedules them for the execution.

```
Loaded config                                                    a)
Ensuring Communication Server is running...                      b)
Configured to use 4 hosts
Starting 4 new static hosts...                                   c)
Starting host number 0
Starting host number 1
Starting host number 2
Starting host number 3
Waiting for hosts to start...
Collecting Tests from: class tachyon.conf.UtilsTest              d)
Collecting Tests from: class tachyon.UnderFileSystemTest
Collecting Tests from: class tachyon.hadoop.UtilsTest
Collecting Tests from: class tachyon.hadoop.fs.TestDFSIO
...
Collecting Tests from: class tachyon.PrefixListTest
Collecting Tests from: class tachyon.util.CommonUtilsTest
Collecting Tests from: class tachyon.util.UnderfsUtilsTest
Creating RunnerWrappers for tests...                             e)
Creating new host for: junit.parallel.runner.RunnerWrapperExecutor
Creating new host for: junit.parallel.runner.RunnerWrapperExecutor
Creating new host for: junit.parallel.runner.RunnerWrapperExecutor
Creating new host for: junit.parallel.runner.RunnerWrapperExecutor
Scheduling RunnerWrapper[listPropertyCommaTest(tachyon.conf.UtilsTest), ] to queue 0     f)
Scheduling RunnerWrapper[listPropertyMixModeTest(tachyon.conf.UtilsTest), ] to queue 1
Scheduling RunnerWrapper[listPropertyHeavyFormatTest(tachyon.conf.UtilsTest), ] to queue 2
Scheduling RunnerWrapper[listPropertiesMissingTest(tachyon.conf.UtilsTest), ] to queue 3
Scheduling RunnerWrapper[listPropertyTest(tachyon.conf.UtilsTest), ] to queue 0
```

- After the setup, the actual execution of unit tests takes place. CUT reports partial test results as the console output shows. Ip addresses identify the virtual machine that executes the tests:

```
--172.17.0.4-- Started to run: rerunOutStream(tachyon.client.TachyonFileUpdateTest)
--172.17.0.2-- Started to run: skipTest(tachyon.client.RemoteBlockInStreamTest)
--172.17.0.3-- Started to run: writeEmptyFileTest(tachyon.client.TachyonFileTest)
--172.17.0.5-- Started to run: readRemoteTest(tachyon.client.TachyonFileTest)
--172.17.0.4-- Started to run: readExceptionTest(tachyon.client.BlockHandlerLocalTest)
--172.17.0.2-- Started to run: isInMemoryTest3(tachyon.client.TachyonFileTest)
--172.17.0.3-- Started to run: readTest1(tachyon.client.LocalBlockInStreamTest)
--172.17.0.5-- Started to run: seekTest(tachyon.client.LocalBlockInStreamTest)
--172.17.0.4-- Started to run: writeTest3(tachyon.client.FileOutStreamTest)
```

- Once the execution is over, CUT prints a summary that shows the number of unit tests that passed and failed as well as the test execution time.

```
#######################################
        Successful Tests:       222
        Failing Tests:          0
        All Tests:              222
#######################################
Finished
Total time: 6 min, 14 sec
```

Results of the experiment:

For the core module of the tachyon project, CUT reduced the execution time from 19 minutes and 40 seconds to 6 minutes and 14 seconds while preserving the results (222 passing tests, and 0 failing tests). Therefore, the results of this experiment show that CUT is able to improve the efficiency of test execution by parallelizing and distribute unit tests execution in the cloud with minimal configuration effort.

Experiment 2: Controlling the Resources Allocations

This experiment shows how CUT is easy to configure. In particular, developers can change the amount of cloud resources that CUT will use by simply changing one input parameter (`numHosts`).

Setup:

- Same setup as the previous experiment
- We change the `numHosts` parameter to be 1, 2, 4 and 8.

Execution:

- Command line:

```
for i in 1 2 4 8; do
. mvn clean test
.  -Drat.ignoreErrors=true -Danimal.sniffer.skip=true # Required by Tachyon
.  -Djcloudscale.configuration=parallel.runner.JCSDockerConfig
.  -Dprovider=cloud -DnumHosts=$i
fi
```

- The four screenshots show that CUT automatically starts the specified amount of hosts:

```
Configured to use 1 hosts
Starting 1 new static hosts...
Starting host number 0
```

```
Configured to use 2 hosts
Starting 2 new static hosts...
Starting host number 0
Starting host number 1
```

```
Configured to use 4 hosts
Starting 4 new static hosts...
Starting host number 0
Starting host number 1
Starting host number 2
Starting host number 3
```

```
Configured to use 8 hosts
Starting 8 new static hosts...
Starting host number 0
Starting host number 1
Starting host number 2
Starting host number 3
Starting host number 4
Starting host number 5
Starting host number 6
Starting host number 7
```

- Each run produced the same amount of passing and failing tests, but the runs which parallelize the execution over more hosts have smaller execution time:

```
#######################################
        Successful Tests:       222
        Failing Tests:          0
        All Tests:              222
#######################################
Finished
Total time: 20 min, 20 sec
```

```
#######################################
        Successful Tests:       222
        Failing Tests:          0
        All Tests:              222
#######################################
Finished
Total time: 10 min, 29 sec
```

```
#######################################
        Successful Tests:       222
        Failing Tests:          0
        All Tests:              222
#######################################
Finished
Total time: 6 min, 14 sec
```

```
#######################################
        Successful Tests:       222
        Failing Tests:          0
        All Tests:              222
#######################################
Finished
Total time: 3 min, 23 sec
```

Results:

This experiment shows how developers can control the allocation of cloud resources for each test execution by simply changing the value of one parameter.

Experiment 3: Controlling Deployment of Tests

This experiment shows how CUT easily can mix the execution of unit tests in local and remote machines. For example, this feature lets developers run unit tests that contain sensitive data in the local machine (i.e., trusted environment), while letting the (possibly untrusted) virtual machines execute the other tests.

Setup:

- Same setup as the previous experiment (but we consider here only the `FileInStreamTest` test case for brevity)
- Use only 1 cloud host

Remote Execution:

- Command line:
  `mvn clean test -Dtests=FileInStreamTest`
  . `-Drat.ignoreErrors=true -Danimal.sniffer.skip=true` # Required by Tachyon
  . `-Djcloudscale.configuration=parallel.runner.JCSDockerConfig`
  . `-Dprovider=cloud -DnumHosts=1`

- The IP address that prepends all the messages in the console output indicates that all tests run on the (sole) Docker host that CUT started:

```
--172.17.0.2-- Started to run: readTest2(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: readTest3(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: readEndOfFileTest(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: seekExceptionTest1(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: seekExceptionTest2(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: seekTest(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: skipTest(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: readTest1(tachyon.client.FileInStreamTest)
```

Mixed Local/Remote Execution:

- To execute some tests locally we provide CUT with a file which contains the name of unit tests that must be executed locally. We use the `local-tests` parameter to indicate such an additional input file. In this example, tests `readTest1`, `readTest2`, and `readTest3` must run on the local machine.
- Command line: `mvn clean test -Dtests=FileInStreamTest`
  . `-Drat.ignoreErrors=true -Danimal.sniffer.skip=true` # Required by Tachyon
  . `-Djcloudscale.configuration=parallel.runner.JCSDockerConfig`
  . `-Dprovider=cloud -DnumHosts=1`
  . `-Dlocal-tests=locals.txt`
- The IP address that prepends all the messages in the console output indicates that all tests run on the (sole) Docker host that CUT started. On the contrary, messages without ip addresses identifies the tests executed locally (as specified in the `locals.txt` file).

```
Started to run: readTest1(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: readEndOfFileTest(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: seekExceptionTest1(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: seekExceptionTest2(tachyon.client.FileInStreamTest)
Started to run: readTest2(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: skipTest(tachyon.client.FileInStreamTest)
Started to run: readTest3(tachyon.client.FileInStreamTest)
--172.17.0.2-- Started to run: seekTest(tachyon.client.FileInStreamTest)
```

Results

This experiment shows that developers can control the deployment of unit tests by listing their names into a configuration file that is passed as input to CUT.

Experiment 4: Test Scheduling

This experiment shows the flexibility of CUT and how developers can you CUT to run complex test scheduling policies. In particular, we show how CUT can schedule unit tests that depends one on another. Test dependencies if not correctly handled during parallel test execution might yield non-deterministic results, alternating failing/passing tests, and false negative/positive [1].

Setup:

- Checkout the crystal project for which dependency information are available from [1, 2]
- Enable CUT as described in Appendix A.
- Start Docker on the remote host (enabling its Web interface -H option).

Basic Parallel Execution:

- We configure CUT to use two Docker containers and its default configuration options for parallelization and test clustering (`MaxParallelStrategy` and `DependencyClusteringStrategy`)
- Command line:
  `mvn clean test`
  . `-Drat.ignoreErrors=true -Danimal.sniffer.skip=true` # Required by Tachyon
  . `-Djcloudscale.configuration=parallel.runner.JCSDockerConfig`
  . `-Dprovider=cloud -DnumHosts=2`
- The console output shows that the test `testSetCompileCommand` runs after `testToString` and `testSetField`. This causes the execution of `testSetCompileCommand` to yield a false negative result: the test passes even though it should fail, as explained in [1].

```
--172.17.0.2-- Started to run: testSetCompileCommand(crystal.model.DataSourceTest)
--172.17.0.2-- Started to run: testIsHidden(crystal.model.DataSourceTest)
--172.17.0.2-- Started to run: testSetKind(crystal.model.DataSourceTest)
```

Dependency-aware Parallel Execution:

- We avoid false results by developing a dependency-aware scheduling policy and by providing information about the dependencies among tests to CUT.
- Command line:
  `mvn clean test`
  . `-Drat.ignoreErrors=true -Danimal.sniffer.skip=true` # Required by Tachyon
  . `-Djcloudscale.configuration=parallel.runner.JCSDockerConfig`
  . `-Dprovider=cloud -DnumHosts=2`
  . `-Dstrategy=parallel.runner.DependencyClusteringStrategy`
  . `-Dpath-to-dependencies=LIST.txt.csv`
- This time tests are executed in a different order (`testSetCompileCommand` is scheduled after `testToString`, `testSetCloneString`, `testSetKind`, `testSetParent`. `testIsHidden` and `testSetEnabled`), which leads to the failure of `testSetCompileCommand`.

```
--172.17.0.3-- Started to run: testSetCompileCommand(crystal.model.DataSourceTest)
--172.17.0.3-- Failure: testSetCompileCommand(crystal.model.DataSourceTest)
--172.17.0.3-- Started to run: testSetRemoteCmd(crystal.model.DataSourceTest)
```

Results:

Crystal has been reported to have 8 dependent tests. This causes false negatives (tests are marked to *pass* even though they are supposed to fail in reality), unless tests are executed in a specific order.

By providing the `DependencyClusteringStrategy` strategy to CUT we shows that it can schedule all tests according to theiir dependencies so that no false positives occur.

B Configuration Parameters

- jcloudscale.configuration
  - The configuration for JCloudScale
  - Options: runner.JCSConfig - parallel.runner.JCSDockerConfig
- provider
  - The Surefire provider to use for test execution
  - Default: cloud
  - Alternative: regular
- testsummary
  - How to summarize the test results
  - Default: none
  - Alternatives: methods - classes - both
- numHosts
  - The number of cloud hosts to use
  - Default: 2
- amqaddress
  - The address of the ActiveMQ instance
- dockerhost
  - The address of the Docker network interface
- dockerimage
  - Name of the docker image to use
- path-to-dependencies
  - The path to a CSV file containing the dependency information
- strategy
  - Implementation of the `ClusteringStrategy` interface
  - Default: parallel.runner.MaxParallelStrategy
  - Alternatives: parallel.runner.SingleWrapperStrategy - parallel.runner.DependencyClusteringStrategy
- distributor
  - Implementation of the `CloudDistributor` interface.
  - Default: parallel.runner.RoundRobinDistributor
- local-tests
  - The path to a file which contains all tests that should be executed locally

References

[1] W. Lam, S. Zhang, and M. D. Ernst. When tests collide: Evaluating and coping with the impact of test dependence. Technical Report UW-CSE-15-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Mar. 2015.

[2] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 385–396, New York, NY, USA, 2014. ACM.