Testing Report

April 14 2018

SEng 468 : Spring 2018

Group G12

Heather Cape V00795197
Geoff Lorne V00802043
Tanner Zinck V00801442

Table of Contents

1.0 Introduction

Testing in any large scale software system is incredibly important, but sometimes not given the appropriate level of consideration. Admittedly, due to the amount of code rework that happened in this project, some automatic testing of basic command requirements could have saved headache in the long run. The testing that was done was mostly manual, with some tools developed to help make bugs easier to find. There is a lot of room for improvement. Now that the project has been stood up, third party test tools could be integrated to help detect problems.

There is also a section on stress testing that can be read in the Fault Tolerance document.

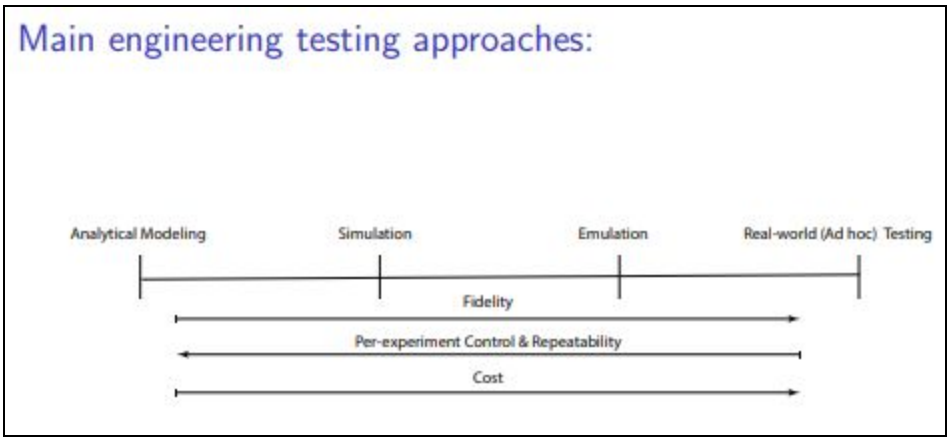

Figure 1: Testing approaches[1]

The main type of testing used for this system based on those learned in class was emulation. We emulated real users through the workload generator and were able to run controlled tests over and over. We took steps to keep our workload generator as realistic as possible, for example for a while we loaded user files alphabetically, but because our transaction servers were divided based on username as well, this could lead to some biased behaviour. We changed our setup so that the order of the user files were randomized with each run.

[1] Distributed Software Systems, Stephen W. Neville, 2017-18, Slide 31

2.0 Manual Testing and Tool Development

2.1 Mock Quote Server

In order to test our system without being physically in the lab, one of the first components we implemented was a fake legacy quote server. This is a simple TCP server written in Go, which operates in a thread safe manner to server quotes in the same format as the legacy quote server. This allowed us to be more efficient with our development time, and come into the lab with a mostly working system. Having a functioning mock quote server allowed us to do end to end testing without being in the lab, and then spend our time in the lab working on performance, and emergent behaviours and bugs.
Unfortunately, there was a point where our services were running faster than the mock quote server could handle. At this point when testing user files larger than 45, most testing that involved the entire system was done in the lab.

2.2 Testing With Varied Environments

Since our development environments (our laptops) differ greatly from the production environment, it was important to ensure that the differences did not cause any problems regarding execution of commands in the system. One simple solution was to do all testing inside docker on our development machines. This works, but requires the Docker images to be rebuilt just to test the smallest of changes. To allow testing directly on our development machines, we have ensured that during the loading of configuration values in each service, there is a check to see whether it is running in Docker or not, and load the system configuration accordingly to ensure a working system. This dynamic configuration loading provided transparency in the testing process, and ensured no time was wasted in configuration.

2.3 Endpoint Test Plans

In order to ensure testing consistency throughout the development of the trading system, we devised and implemented a universal set of endpoint tests which we ran from the Insomnia REST API client. Insomnia gave us a very simple way to test whether each endpoint was responding as expected when refactoring code, or adding features.

Acceptance testing and integration testing were also performed manually using the Insomnia test suite, with each test being ran manually as needed during the addition or change to any component of the system.

Eventually when the webview was more fleshed out the end-to-end system could be tested from there. It would have been good to have the website be built alongside the rest of the project as it revealed some bugs.

2.4 Problem Solving Solutions

Several external tools were written to help us debug problems throughout the project.
First was a file of somewhat complex PSQL commands used to debug system problems with the audit database. These helped us locate missing transactions or figuring out TPS discrepancies. For example, our workload file bash script had an error where it incorrectly parsed all Display Summary commands; the command helped us quickly solve that problem. There was also one case were one single transaction was missing from the log, but we were able to query the database and quickly confirm that it was a malformed buy command that was not logged.
We wrote a small external service to compare the input and output values of the cryptokeys coming from the legacy server to those that were in our logfile. The cryptokey validation was failing due do decoding errors, so we thought that some type conversion could have changed the value of the keys. This test tool helped us prove that our database was outputting an identical key value to that which the legacy server outputted.

As demonstrated in class, the utilization of a service is very important in determining its performance. We were able to leverage the tools with docker compose to check the values of memory and CPU usage for each service. For example, one time our system was hanging after presumably finishing all transactions; viewing the utilization narrowed down that the audit server was under heavy load because it was writing multiple versions of the log.xml file instead of just one.

A python script was written to test the connection times to the legacy quote server, collect the data, and parse it to be easily graphed. This helped to visualize the distributions of quote responses, and pick optimal time out times for the TCP connections to the legacy server.[2]

3.0 Continuous Integration

3.1 TravisCI

Travis Continuous Integration was implemented and hooked into our Github workflow to check whether any commit pushed to a remote branch will successfully compile. This caught errors a couple times when we forgot to push all changed files, and saved valuable time that would have been spent on debugging once we got to testing in the lab. Travis built our code with a specific version of golang that matched the Golang version in our Docker container. This prevented any

[2] More on this in the Architecture document

version mismatch errors as we updated our golang version midway through the course. TravisCI has the ability to run Go unit tests as part of its build process. Our next step in improving test robustness and code coverage would be to write unit tests for our componentes which would be run every time a Travis build is triggered.

4.0 Future Testing

Moving forward with development of the trading system, we would like to implement automated regression testing, acceptance testing, as well as unit testing. Now that our codebase is much more stable, this is feasible and worthwhile.

4.1 Regression Testing

An automated regression test suite would be run anytime changes are made to the system to ensure that no previous business errors or software bugs can creep back into the system. These tests would be implemented as low level, and in as simple of a manner as possible. For example, if we found a bug had crept into a utility function, implementing specific unit tests for that function to ensure the bug cannot come back would be our regression test.

4.2 Acceptance Testing

Our acceptance test suite would consist of a set of commands to be run as end to end tests, ensuring that the system is adhering to all business constraints. These would be implemented as HTTP requests to simulate an actual user using the system, and detect any possible errors in the flow of processing any command. A sampling of all commands that the system can perform would be used to have the highest chance of detecting any present errors.

4.3 Unit Testing

Go provides the ‘testing’ package which allows developers to run unit tests from the ‘/test’ directory with the command ‘go test’. This is a very simple setup with low startup time, and will allow us to setup and maintain unit tests for all code with minimal developer effort. Unit tests will be created for all utiliting functions, as well as HTTP endpoints. When necessary, HTTP request mocking will be used to ensure that our Go router functions are working correctly. Unit tests for functions that directly handle money will have tests which focus on potential errors in currency operations, such as string to integer conversion errors, malformed strings, and negative values. Rigorously testing all functions that handle money is key to ensuring that our system does not make small off by one errors once in a while, since these can quickly add up in a system performing millions of transactions and cause major business problems. A suite of Selenium tests can also be written and used to test the front end automatically as well.

4.4 Integration With Travis CI

Writing all unit tests using the ‘testing’ package also has the advantage of very simple integration with Travis, our continuous integration tool. Travis can easily be setup to run the test test suite contained in ‘/test’ by adding a single line to our Travis config file. This would ensure that all unit tests are ran whenever a commit is pushed to a remote branch. By cutting down on time required to ensure new commits pass unit tests, code review times will be kept to a minimum, and since the developer effort to implement this is so low, using the builtin testing package is the clear choice for the trading system given our current development situation.

5.0 Conclusion

Relative to some of the other goals we had for the development of the trading system, testing was on the less important end, with favor given to chasing performance, and formal testing efforts kept to a minimum. Since we were able to achieve a very high level of performance, and produce a system that satisfies the business requirements, this strategy worked out well for us.