Quality Assurance in Complex Distributed Systems

by

M.A. de Vos

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday January 1, 2013 at 10:00 AM.

Student number:	1234567	
Project duration:	March 1, 2012 – January 1, 2013	
Thesis committee:	Prof. dr. ir. J. Doe,	TU Delft, supervisor
	Dr. E. L. Brown,	TU Delft
	Ir. A. Aaronson,	Acme Corporation

This thesis is confidential and cannot be made public until December 31, 2013.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

Preface...

M.A. de Vos
Delft, January 2013

Contents

1 Introduction **1**

2 The History of Tribler **3**

3 Problem Description **5**

3.1 Testing Infrastructure 5

3.1.1 System and Deployment Testing 5

3.1.2 Stability of the Tests 5

3.1.3 Execution Time 6

3.1.4 Code Coverage 6

3.2 Separation of the GUI and the Core 6

3.3 Documentation 6

4 The Architecture of Tribler **7**

5 Improving Quality Assurance **9**

6 Documentation **11**

7 Evaluations **13**

Bibliography **15**

1 Introduction

Hallo

2

The History of Tribler

Here will be the history of Tribler

3

Problem Description

Tribler is the result of 10 ongoing years of research in the area of peer-to-peer systems. With the first commit dating back to september 20, 2006 and 31 search results showing up in the TU Delft Institutional Repository browser, the code base of Tribler has evolved tremendously. 44 contributors have worked so far on Tribler, creating over 11.000 commits. Managing a software project on such a large scale requires adequate tools, a proper testing infrastructure, clear documentation, a logical project structure and a clear code style, where every team member adheres to.

Instead of an easy to understand code base with a single style and consistent, logical naming, Tribler is more like a collection of glued together thesis projects. In this Chapter, the problems with the current source code, infrastructure and project structure will be addressed and explained.

3.1. Testing Infrastructure

To test the code base, a Jenkins server for Continuous Integration (CI) is used. When a Pull Request (PR) in the Tribler repository on GitHub is opened, the tests are executed on a Linux server. In this Section, various problems with the testing infrastructure will be elaborated.

3.1.1. System and Deployment Testing

When starting this thesis, the available test suite more resembles system and integration tests. Only a small fraction of the tests are actual unit tests. Most of the tests are starting up a complete Tribler session while often, only a very small subset of the features of Tribler will be tested in a particular run. For instance, in most tests, Dispersy is started while in some situations, no Dispersy-related code is tested. Each of these enabled features might lead to unwanted side-effects and failures during the test execution.

Each unit test must be very small and should only focus on the code that actually is being tested. Mocking techniques should be used as much as possible to have more control of the testing environment.

Deployment testing should be in place to test whether Tribler actually runs correctly on all supported platforms (Windows, OS X and Linux), using the real Tribler network. Some basic operations might be executed such as remote searching, downloading torrents and streaming a video. Screenshots can be taken to (manually) verify the correctness of the Graphical User Interface (GUI).

3.1.2. Stability of the Tests

The test suite is very unstable and failing randomly across various runs. The code base of Tribler is infected with race conditions that are happening a fraction of the time, due to sloppy multi-threaded programming. Another factor that effects the stability of the tests is the dependency on other peers in the network. Instead of isolated tests in a controlled environment, the Tribler sessions that are started during the tests, are actually connecting to other users in the network, resulting in uncertain situations.

The unit tests should not be dependent on the outside network. Tests that rely on network-dependent code should be tested with mocked objects where right (artificial) data is returned, possibly after some small delay.

3.1.3. Execution Time

The total execution time of these tests is around seven minutes. This can be explained by the fact that often a Tribler session is booted up for each test. Starting and stopping Tribler takes a significant amount of time and resources and for some tests, it it not even necessary to start up the whole Tribler session. Together with the fact that some tests are randomly failing, it can take several tries to get all tests passing, resulting in wasted time and unnecessary waiting for developers. Moreover, some tests are very fragile and tend to timeout, leading to a failure after around twenty minutes.

There are two important factors for solving this problem. First, the tests should be very stable, failing only when a developer has written code that makes one or more tests failing. Second, mocking must be used to have more control over the data flow. By mocking the Tribler session, it might not be necessary anymore to start up the whole interface.

3.1.4. Code Coverage

The code coverage of Tribler 6.5 Release Candidate 7 is around 65% of the lines of source code. As can be seen in Figure X, no effort has been made to increase the code coverage over time.

The code coverage of conditional statements is around 45%. This lower percentage is due to the fact that the tests are only executed on a Linux environment. Tribler contains many platform-conditional statements which are currently not covered. To solve this problem, the tests should also be executed on Windows and OS X in parallel.

Another issue is that many exceptions remain uncovered during the tests. By making small unit tests and by the usage of mocking, these exceptions can be covered, thus increasing the code coverage.

3.2. Separation of the GUI and the Core

The GUI is interleaved in the core of Tribler. This means that the GUI does not exist without the Tribler core and vice versa. The disadvantage of this approach is the lack of portability of the core. With a clear separation of the GUI and core function, developers can create their own GUIs and power them using *libtribler* (the core of Tribler without the GUI). *libtribler* could also be used stand-alone, in the form of a command line application. This approach is very similar to the separation of the GUI and kernel in Linux.

To make it easy for developers to work with *libtribler*, an API should be written, for instance, using JSON-RPC. The endpoints of this API should contain the most commonly used operations within Tribler such as searching for torrents, managing downloads and changing settings.

3.3. Documentation

The project in its current state contains very little and poor documentation. Most of the documentation is available is outdated or not relevant anymore. A solid documentation base is important for volunteers and new team members wanting to start working on Tribler. Guides to get new developers up and running should be written to make Tribler a more attractive platform to work on.

4

The Architecture of Tribler

Here will be the architecture of Tribler

5

Improving Quality Assurance

Hallo

6

Documentation

Hallo

7 Evaluations

Hallo

8

Conclusions

Hallo

Bibliography