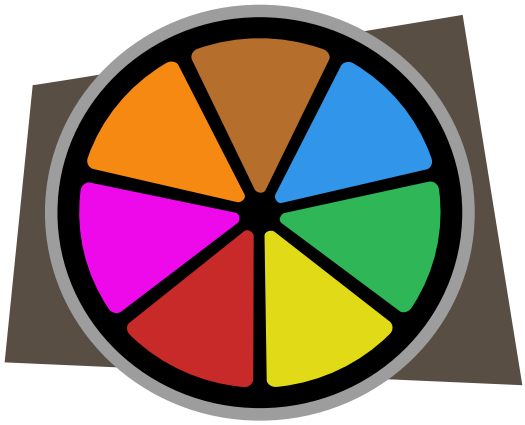
Trivial Torrent Quality Assurance

Xarxes, Grau d'Enginyeria Informàtica



Document version: 1.0
Date: May. 2020

Universitat Autònoma de Barcelona

Departament d'Enginyeria de la Informació i de les Comunicacions

Document Information

Document identifier: trivial_torrent_quality_assurance.md
Title: Trivial Torrent Quality Assurance
Number of pages: 5
Date: May. 2020
Author(s): Ian Blanes

Change Control

Revision	Date	Changes
1.0	May. 2020	(initial version)

Contents

1 Quality Assurance

This document gives some indications and ideas on the quality assurance (QA) actions you can take in order to make sure you deliver a high quality assignment.

The document provides a non-exhaustive list of QA ideas, and it is not mandatory to apply all (or any) of them. Use the QA strategies that you think will work best on your project (this will depend on your coding style, what you have tested so far, etc).

Remember: you need to deliver high quality code, not just code that works most of the time. Projects delivered without quality assurance are FAILED projects.

For this task you will need to write a convincing two page report on how great your QA has been. There is minimal template document that you have to employ for your report.

Notes:

- We may, or may not, perform some automatic testing of your source files, so be careful with what you deliver. You do not want it crashing on the first test. Remember: if it is not tested, it does not work.
- Given how large the solution space is, finding two similar quality reports will be *very* suspicious.

1.1 Clean code

Having clean and legible source code files before starting any QA activity is a good investment. You may want to pick a coding style and stick to it.

Remember to *properly* comment your code. Add meaningful comments, and do not add obvious comments (assume people know how to read code).

```
// We will proceed to iterate over all poll responses. <- OK
for () {
}
```

```
i++ // and here we increase i by 1. <- Not OK.
```

1.1.1 Some coding style we may like

```
int function (const int parameter) {
    if (parameter) {
        call_function(parameter);
    }
    return parameter;
}
```

1.1.2 Some coding style we will likely not like

```
int f1(){
    if (test) {
    return (f(3));
    }
    if(Totally_inconsisent_space_usage
        && variableCaseConvention)
    {
    } else
    {
// are we really using "{" after a new line here?
// BTW, why is this comment not properly indented?
    }
}
```

1.1.3 We do not like too many levels of nested control structures

```
int f() {
    if (bind(...) == 0) {
        if (listen() == 0) {
            if (accept() == 0) {
                //...
            }
        } else {
            return -1;
        }
    } else {
        return -1;
    }
}
```

1.1.4 Instead, we like to handle errors and forget about them

```
int f() {
    if (bind(...) < 0) {
        perror("...");
        reurn -1;
    }

    if (listen() < 0) {
        perror("...");
        reurn -1;
    }

    if (s1 = accept() < 0) {
        //...
    }

    return 0;
}
```

1.1.5 Here we handle an error far from where it occurs

```
for (int i = 0; i < MAX; i++) {
    if (function_ok) {
        // some very large piece of code
    } else {
        // handle error and continue
        // This is too far to remember what was this about :-(.
    }
}
```

1.1.6 Here we handle an error near to where it occurs

```
for (int i = 0; i < MAX; i++) {
    if (function_ko) {
        // handle error and continue
        // Nice :-).
        continue;
    }

    // some very large piece of code
}
```

1.2 Code review

A good starting point it to carefully review your code. Here is a checklist of things you may want to check:

- Are *all* function calls, except for those that cannot possibly fail, followed by an error check (including malloc, close, etc)?
- Is `errno` reset to zero if necessary?
- Does your code compile without warnings?
- Before any type cast, are proper verifications made so that no overflow is possible?
- Is your application failing silently? If there is an error, it should say so.
- (add more items here)

1.3 Static analisis

Perform some static analysis on your code with `clang --analyze`. Be careful with static analyzers, as they signal a lot of false positives (things that are not errors).

1.4 Memory issues and resource leaks

Does your code run without issue under `valgrind`? Are there any file descriptor leaks (`--track-fds`)?

1.5 Code hardening

Compile with `-fsanitize=undefined -fno-sanitize-recover=all` to detect undefined behaviors at run time. Fill your code with assertions (`assert`) to make sure everything is as it is expected.

1.6 Test more things than the network stack

Make sure your application works properly when dealing with rare filenames, relative paths, invalid command lines, etc. Examples:

- `ttorrent -L 8080 should_not_work.ttorrent`
- `ttorrent -L -1 should_not_work_either.ttorrent`
- `ttorrent -L -1 should_not_work_either.ttorrent.extra_extension`
- `ttorrent -L -1 should_work.extra_extension.ttorrent`
- `ttorrent client.ttorrent extra_invalid_argument`
- `ttorrent this_file_does_not_exist.ttorrent`
- `ttorrent some_very_long_file_name.ttorrent`
- `ttorrent ./normal_file_in_the_current_directory.ttorrent`
- (your other tests here)

1.7 Fault injection

Use `lib-fiu` to inject random faults (`fiu-run -x`) and make sure your application does not crash (it should exit, but not crash).

1.8 Malicious client or server

Create a copy of your client (or server) that tries to hang your server (or client). For example:

- by spliting your messages in small parts and waiting (`sleep`) between `send`s,
- by waiting before `close`,
- by send invalid block numbers (integer overflows?), or
- by sending too much (or too little) information.

1.9 Corner cases

Test for corner cases. Create a `.ttorrent` file with zero blocks (an empty downloaded file), or one with 10k block. Test 2000 clients in parallel (add `sleep`s to make sure they are actually enabled). Test with multiple servers in different machines. Test over the internet (be careful).

1.10 Fuzz your code

Use a *fuzzer* to make sure your code works properly. For example, use the *American Fuzzy Lop* to test your server (`apt install afl`). You will need a fuzzing harness for your project (one will be provided). This is a difficult task, but it will yield an amazing number of bugs.

1.11 Test on other platforms

You may want to compile your code with multiple compilers and watch for their warnings.

You may want to ensure that your code works on platforms other than `x86_64`.

For example, you may want to build `i386` binaries (you will likely need to `dpkg --add-architecture i386` and `apt install libssl-dev:i386`).

You may want to install Debian an emulated big endian virtual machine. Here are some hints on how to call QEMU once the virtual machine is installed (don't expect this command line to work directly).

```
qemu-system-mips -hda your_image.qcow -M malta -bood d \
    -kernel the_kernel_needs_to_be_outside_of_your_image.vmlinux-4.9...-malta \
    -initrd thesamething.img-...-malta -m 2048 -nographic -append "root=/dev/sda1 nokaslr"
```

1.12 Remember to proof read your QA report

Do some QA on your QA report.