Automated test infrastructure BS2015

Koen Koning
koenkoning@gmail.com

Taddeüs Kroes
taddeuskroes@gmail.com

Introduction

This document describes an automated testing/grading system, developed for the 2015 Besturingssystemen course at the UvA. This system generates a (partial) grade based on the source files submitted by the students without intervention from the TAs. The primary goals are to make grading easier on the teaching staff, and to make grades more consistent. However, the system is set up in such a way that students themselves can also use it. This provides transparency about the grading process and gives the students detailed feedback even whilst still working on the assignment. The system is intended to be used by the TAs to generate final grades as well. Therefore, security and reliability were big factors in its design. Since most of the system is accessible to students, we need to prevent them from exploiting, bringing down or cheating the system.

Course-wide, there are two primary components: a per-assignment test suite and a server that compiles and runs the relevant test suite when a student submits their code for an assignment. The server exists for several reasons: it provides a pre-configured environment, so students (or the test suite) do not need to set up an environment themselves. Furthermore, the secure environment is used by TAs when grading to avoid the risk for malicious code running on their own machines. Finally, it allows for further development and refinement of a test suite while the students are already working on the corresponding assignment. An example of this is the live leaderboard of assignment 2 (see below).

Usage of the server is easy: running `make check` on the command line will automatically create an archive, submit it to the test server and wait for a response.

Note: this test infrastructure does not completely eliminate manual grading. Code should quickly be checked for quality and to see if people try to cheat the system. Furthermore, the reports should of course be graded manually.

This documents outlines the testing infrastructure: all ofits components and the associated scripts in the git repository, including limitations and future work. We will also discuss the assignment-specific test suites, some server setup instructions (e.g. nginx and docker) and scripts that can be used for the grading process by TAs.

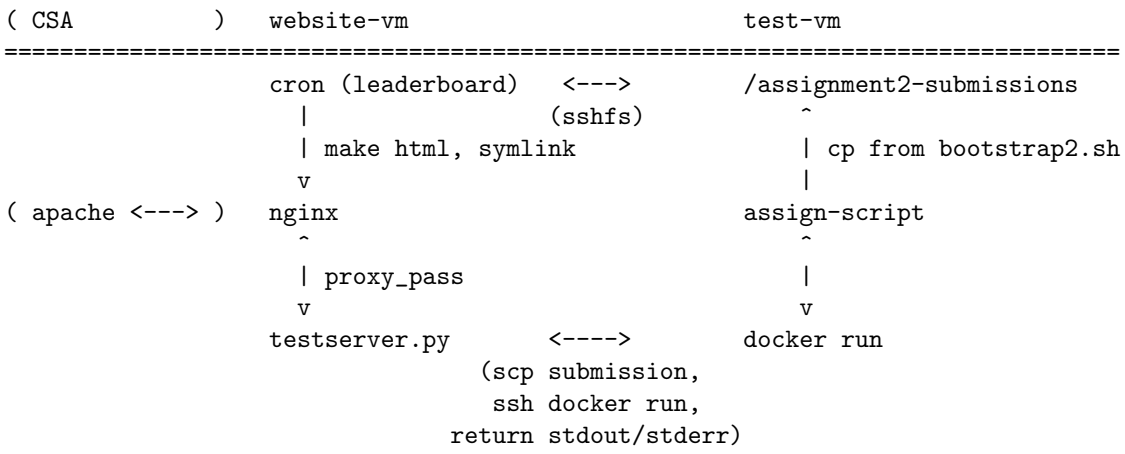
Interaction between web server and test server

First of all, we assume a setup with two virtual machines, isolated but able to communicate to each other using SSH. We refer to these as the "webserver machine" and "eval/test machine" respectively.

The course website runs on the first VM, where nginx serves static files from the /srv/www directory. In the 2015 edition of the course there was an additional layer in between, since the VMs were not directly accessible from the internet: an Apache instance on the CSA server served as a reverse proxy, redirecting all requests to `https://csa.science.uva.nl/bs2015/*` to the nginx server in the VM.

The nginx server serves the course website as static files, and also the periodically generated leaderboard for assignment 2. Furthermore, it proxies all requests to `/upload*/` to a Python-based test server which initiates the corresponding testsuite. This scheme makes the test server operate over http(s), so students do not have issues with firewalls etc..

The following diagram illustrates the scheme outlined above:



Test server

The test server implements a minimal HTTP web server accepting file uploads of tar.gz files. It accepts multiple clients in parallel, and once an upload is done the connection is moved to a queue for testing. There are a number of threads running that read from this queue, the number probably equal to the number of cores in the test machine. It can be found in the git repository at `testing/testserver.py`.

In order to ensure the safety and stability of the system, we use two layers of isolation: the actual tests run in a separate VM (the test machine), inside a docker container. The container limits resource usage and provides a clean (UnionFS) file system so students do not interfere with each other. The separate VM is an additional security measure: there is no risk of the website and test server going down, and it prevents an additional barrier if students manage to break out of the container.

Because an nginx proxy to remote host + file uploads didn't mix well, the test server runs on the website-vm. It then (if enabled) transfers the file and starts the docker with tests on the test-vm using ssh (scp for the transfer).

The test server has a per-assignment list of files to mount inside the container, and the initial script to execute. It will return the `stdout` and `stderr` of the latter in the response to the user.
Note: the mounted files are on the machine executing the docker command, so in the case of the split-setup via ssh, these files refer to files on the test-vm.

The test script running inside the docker container does several steps: set up the environment to run under the user "user" instead of root (since users and file permissions are semi-shared outside the container). It will then extract the student archive and run the assignment-specific test script. Extraction occurs in a slightly smart way, as we want to enforce certain framework files (e.g. so they cannot disable prints or compiler warnings). It extracts certain assignment-specific 'base files' (files the students are not supposed to edit), and then parses the makefile for additional sources and header files added by the student. The extract script disallows these to overwrite any framework files, and generates a makefile from the framework with the additional sources definitions copied from the user-provided one.

Assignment 1 - Shell

The test suite for the first assignment is all contained in a single file (`testing/testcases1.py`). For most cases the output of the submitted shell is compared against the output of bash, as this should provide the correct output and doesn't depend as much on the specific setup of the machine. Ideally, every command would run under `valgrind` to check for errors, but this slows down the entire test suite by a factor 3. Therefore, a single command is executed under `valgrind` which should touch most parts of the student code. The large amounts of forks happening here can allocate a lot of memory, as every fork also forks `valgrind`.

While this test script works, it is quite large and written in Python which is not well suited for testing shell commands. Ideally it should be rewritten from scratch in bash, split out over multiple files (like in assignment 2). Also, a check that should be added is to check the open file descriptors of the shell process, to make sure students close the pipe after both child processes are forked.

Assignment 2 - Memory allocator

The test suite for assignment 2 is much more modular and written in bash. `testing/bootstrap2.sh` is the script executed in the docker instance and takes care of setting up the environment inside docker, extracts the relevant files from the framework and user (`testing/extract.sh` and then executes the assignment-specific scripts. These files can be found under `opdrachten/myalloc/check/`, and are indeed part of the framework (and thus extracted from `framework2.tar.gz` in the docker instance. Note that the makefile for assignment 2 also offers the `make localcheck` command which runs the test suite locally on the student's machine.

This assignment also has a live leaderboard for performance and energy usage, which is the last step of the `testing/bootstrap2.sh` script. Getting the results from the docker container to the website-vm's website adds even more complexity to this already convoluted system. . . :-)

Docker mounts a host directory inside the container, accessible by root only. After a successful run of the test script it will copy the binary, authors and leaderboard_name files into this dir. This (test-vm) directory is also mounted by the website-vm via sshfs. On the website-vm, a cronjob runs every 5 minutes and runs `make` inside the leaderboard directory of assignment 2. The makefile generates statistics for all updated binaries (in a docker instance, but on the website vm, since we can trust the binary not to deadlock the machine by now, probably), and then runs a Python script to generate the HTML page. This HTML page is symlinked into the nginx www root so that it can be served on the website.

Assignment 3 - Scheduler

There is currently no grading script for assignment 3.

Since the current framework of this assignment already does error checking, the grading script could pick up on these messages. Optionally, a leaderboard could perhaps be made similar to assignment 2. Correctness of e.g. the n-try implementation would be more difficult to automatically test. A "smart cpu-scheduler" would also be very difficult to automatically generate points for. The script could perhaps put a threshold on the expected performance figures.

Assignment 4 - Locks

There is currently no grading script for assignment 4.

In the future a grading script for this assignment could look at the errors already generated by the framework. To know a file actually implements the API it says it does could perhaps be done using `nm` to see what symbol were (and weren't) used by the code. Similarly, `ltrace` could be used.

An additional problem for this assignment would be people writing OS-specific locking strategies, e.g. a few people had OSX implementations.

Note that for this assignment the manual grading of the code is quite easy as long as the TAs are familiar with the APIs: there is very little code per implementation and the APIs can often only be used in one way. An automatic grading script, on the other hand, would be much more difficult to develop. The framework should error when the implementation is not correct, so students can also get immediate feedback without an automated testing infrastructure.

Scripts

The `scripts` directory in the git repository contains some scripts to ease the grading process. All scripts contain usage instructions (usually in comments). `bb_extractor.py` is an old, ugly, hacked together script to deal with Blackboards bulk download feature. `run_make_check.sh` runs the `make check` command for every folder in the current directory, dumping the output to `make_check_out`. `parse_authors.py` is a helper script used by `gen_tsv.sh` to extract names and number from the AUTHORS files submitted by students, which can be a real mess sometimes. `gen_tsv.sh` generates a tsv file with all groups which can be imported into google sheets for further grading. It can pull the grade from `make_check_out` (generated via `run_make_check.sh`), but this is currently commented out. Finally, `excel2bb.py` takes the filled in grading sheet from excel/google sheets and merges it with the "Work offline" thing from Blackboard, so these grades can be automatically imported into Blackboard.

Server setup

Hopefully the setup is still intact from last year, in which case the test server can easily be started with:

```
$ cd /home/koenk/bs2015/testing
$ python3 testserver.py  # Preferably in a screen/tmux
```

This command should be run on the webserver vm, which should be able to ssh into the test vm without passphrase. See the top of `testserver.py` for some settings including the location of the test vm. The test vm should have the entire testing tree at ∼`/bs2015/testing`, since these are the scripts that will actually run/will be mounted in docker.

Assignment 2 (myalloc) leaderboards

As described earlier, assignment 2's leaderboard requires some extra setup. It will dump all submitted binaries in `/home/koenk/bs2015/testing/submitted`. Thus on the webserver vm:

```
$ cd /home/koenk/bs2015/opdrachten/myalloc/leaderboard
$ mkdir submitted
$ sshfs 172.20.0.151:bs2015/testing/submitted submitted/
$ cd /srv/www
$ ln -s /home/koenk/bs2015/opdrachten/myalloc/leaderboard/index.html \
            myalloc_leaderboard.html
$ sudo crontab -e
```

And add(/uncomment):

```
*/10 * * * * sudo -n -u koenk make -C /home/koenk/bs2015/opdrachten/myalloc/leaderboard
               >/root/cron_out 2>/root/cron_err
```

(on a single line)

The `myalloc_leaderboard.html` would sometimes get stomped over by Raphaels rsync on the `/srv/www` directory, so better place it outside of there and add a nginx location for it for next year...

Docker

```
$ docker pull ubuntu
$ docker run -i -t ubuntu /bin/bash
    # apt-get update
    # apt-get install build-essential clang-3.4 flex libreadline-dev \
                        valgrind python-pexpect bc
    # groupadd user
    # useradd -u 1234 -s /bin/bash -d /home/user -m -g user -p <random-password> user
    # exit
$ docker ps -l  # Note container ID
$ docker commit <container-id> uva_cs/os-base
```

Optionally test with

```
$ docker run -i -t uva_cs/os-base /bin/bash
```

Nginx

In `/etc/nginx/sites-enabled/os`:

```
    server {
        listen 80 default_server;
        listen [::]:80 default_server ipv6only=on;
        server_name localhost;

        root /srv/www;
        index index.html index.htm;


        location / {
            try_files $uri $uri/ =404;
        }

        # So rsync on /srv/www doesn't delete these
        location /reports/ {
            alias /srv/www-reports/;
            autoindex on;
            auth_basic            "Restricted Area";
            auth_basic_user_file  reports-htpasswd;
            types {
                text/plain log;
            }
        }
```

```
    # Regex and proxy_pass aren't friends?
    location = /upload1/ {
        proxy_pass http://localhost:12346/1;
    }
    location = /upload2/ {
        proxy_pass http://localhost:12346/2;
    }
    location = /upload3/ {
        proxy_pass http://localhost:12345/3;
    }
    location = /upload4/ {
        proxy_pass http://localhost:12345/4;
    }
}
```