Architecture

Web Client

This is the browser which is responsible for a few things.

- The front end code (CSS and HTML) to be displayed
- Every few seconds (3 is a good number) if the program text has been changed, then we trigger a save api call to the web server. This will save the program and avoid program loss (switching tab, hitting the back button, . . .).

Local Client

This is a C library that we will provide to communicate with the webserver. The idea is that we want people to be able to develop their code locally if they choose to. Submission for grading is still done on our server.

Components

enforced limitations

- The sanbox is not enforced in the local client, but it is enforced when we perform the grading on the server

WebServer

The webserver is responsible for interacting between the user and the worker node. Aside from the local client, it is the only public interface we expose to working on the labs.

The webserver does not require special hardware (does not require a gpu). It can be hosted on a standard AWS instance and the database server can be hosted on the same machine.

enforced limitations

- a 3 second limiter is enforced between submissions

API

- **GET** `/login` : gets the login page
- **POST** `/login` (user,pass): checks if the user and a SHA hash of the password exists in the users database
- **GET** `/signup` : gets the signup page
- **POST** `/signup` (user,pass) : creates a new user in the database with a SHA hash of the password
- **POST** `/logout` : logs out the user (clears the cookie)
- **GET** `/oauth_login` : implements oauth authentication (this is used for coursera)
- `/oauth_login/callback` : callback for oauth (can be passed parameters like the origin page)

- **GET** `/grades` : shows the grade page (a table of the summary of the grades)
- **GET** `/grade/{mp_id}` : shows a detailed view of the grade
- **GET** `/mp/{mp_id}` : shows the MP interface
- **POST** `/mp/{mp_id}/submit` : triggered when one clicks the submit button. The programs is passed in the body of the **POST**
- **POST** `/mp/{mp_id}/save` : saves the program into the database. The programs is passed in the body of the **POST**
- **GET** `/mp/{mp_id}/program` : gets the latest saved program for the database. If not found, then the template program is read from the file on the server
- **GET** `/mp/{mp_id}/description` : gets the description of the MP from disk and renders it in markdown format
- **GET** `/mp/{mp_id}/attempts` : returns a JSON object with a summary of the attempts. The summary contains information such as when the attempt was made, the id of the attempt (not shown in the table), whether the attempt passed, and which dataset was the attempt made againast.
- **GET** `/mp/{mp_id}/attempt/{attempt_id}` : gets detailed information about the attempt. This includes the same information of the attempt summary + program/compiler output + program text
- **POST** `/mp/{mp_id}/submit_grade` : this will be a button that the user presses to submit the grade to the coursera server
- **POST** `/mp/{mp_id}/grade` : this will cause the server to test the program against all the datasets and provide the user with the grade they would get if they submit the program. the **POST** field must contain an API key that is secret and unique to each user (could be something as simple as hash(user_name + "secret")[-10:])
- **POST** `/mp/{mp_id}/run_results` : this is called by the worker. The possible behaviors are: the program failed to compile, the program compiled but failed to run, or the program compiled and ran. This data is stored into the database as an attempt by the user.
- **GET** `/workers` : list all the worker nodes
- **POST** `/worker/register` : registers the worker with the server . This inserts the worker into the queue.
- **GET** `/worker/{id}` : list all information about a worker node (including log information)
- **GET** `/images` : public images are posted here
- **GET** `/whoami` : gives the user id of a user. A user may need to visit this page to also get information to configure that local client (API key, user id, . . .)
- **GET** `/log` : log output from the server
- **GET** `/404` : not found
- **GET** `/500` : error

Worker

Unlike the server, the worker must be hosted on a machine with a GPU. The sandboxing might also enforce a certain OS and a certain revision of the kernel (the previous implementation was hosted on an ubuntu machine since centos did not have the kernel features needed for the sandboxing).

enforced limitations

The server enforces the following limitations:

- no compilation can take more than 10 seconds

- no execution of a program can take more than 3 seconds
- before compilation, the program will be textually checked for black listed keywords (exit, abort, include, . . .)
- on run, a program will use seccomp and a white list of syscalls. the program terminates if a syscall not in the ones approved is used

API

- **POST** `/mp/{mp_id}/compile` : compiles and runs | if dataset id is set to -1, then all the datasets are run against the program |
- **GET** `/mp/{mp_id}/data` : checks if the worker has the configuration and data required for the MP
- **POST** `/mp/{mp_id}/data` : sends the data and configuration for the MP to the worker node
- **GET** /is_gpu_alive : runs a simple GPU kernel and checks if it terminates without errors. If not, then the gpu is not alive. It is the job of the webserver to deal with this (send an email, send another command to the worker, remove the worker from the queue, . . .)
- **GET** `/status` : gets the status of the server. This includes information such as the current load, disk information, . . .
- **POST** `/stop` : terminates the server . It is the job of the webserver to remove the worker from its queue.
- **GET** `/log` : log output from the worker

MP Structure

Program Template

Configuration

Dataset

For each MP

Delivery

Previous Infrastructure

The previous infrastructure (wb1) is very similar to the one described above. There are a few omissions however:

- we did not have an instantaneous way to grade MPs. Instead we opted to do it off-line
- we did not have a way to check if a GPU worker is down. This caused the server to go down for some time without notice (also cause misdiagnosis for many users about their bugs and distrust in the stability of the server — more aparent towards the end of the course)
- we did not have a local client that people (who do not want to depend on our webserver) can develop with

Implementation

The previous implementation (wb1) used node.js

New Infrastructure

Why Go?

Why not Haskell?

Why not Python?

Why not Ruby?

Why not Java?

Why not JavaScript?