Distributed Systems ☁ (https://www.cs.columbia.edu/~du/ds)

Homework (https://www.cs.columbia.edu/~du/ds/homework.html)

Tips (https://www.cs.columbia.edu/~du/ds/tips.html)

Policies (https://www.cs.columbia.edu/~du/ds/policies.html)

Schedule (https://www.cs.columbia.edu/~du/ds/schedule.html)

Discussion (https://piazza.com/columbia/fall2013/w49952/home)

Homework1 C++ Warmup

Due: 2013/09/11 23:59:59
1344 Days passed

NOTE: This first homework is graded Pass/Fail only

✎ Written Problems

⚙ Programming Problems

What do you need to submit in the end?

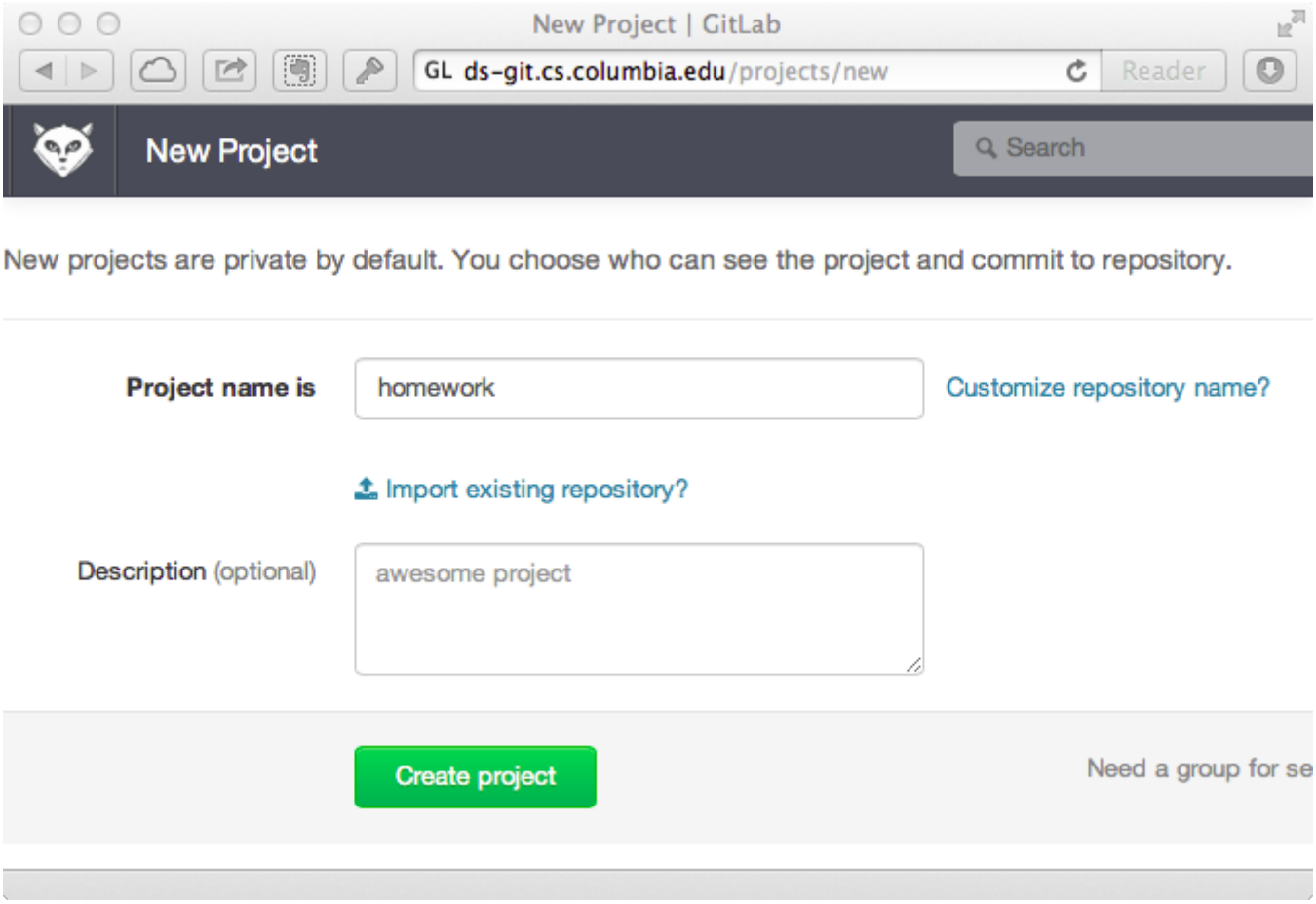
At the end of the homework, you need to submit the following files

1. A text file named `WRITTEN` that is for the written problems
2. `README` file that explains what you did for the programming homework, including the test cases
3. All your source code for programming problems
4. A `makefile` that generates `tcp_client` and `tcp_server` from your source code

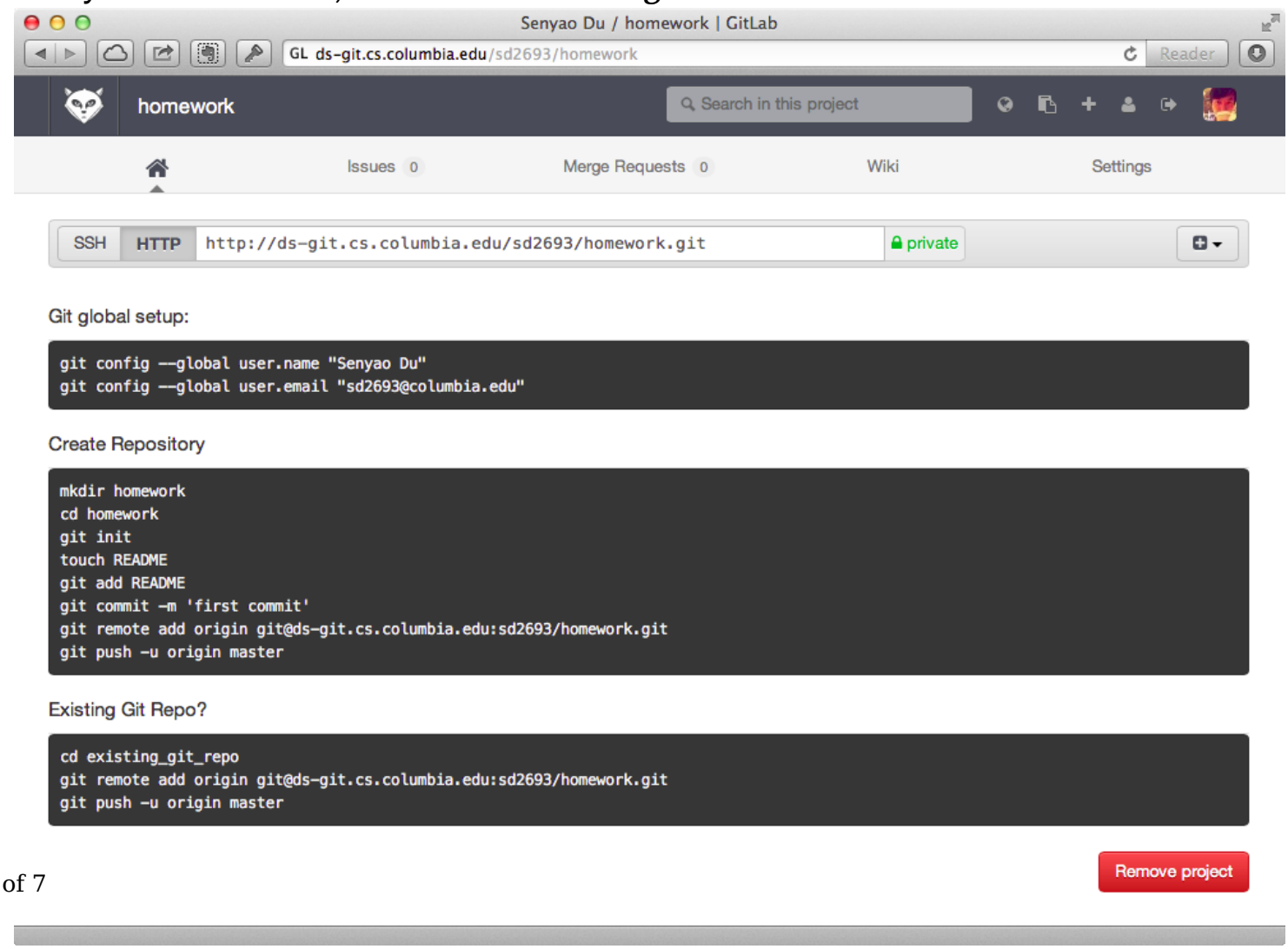
How do to create repo and submit via Git?

This is the first assignment, and I'll work you through the entire setup for the assignments to come

1. Login onto `ds-git.cs.columbia.edu` using your CS account, and visit New Project (http://ds-git.cs.columbia.edu/projects/new) page to create a project named `homework`



2. Follow the instructions onscreen for a new Git repo. Use the *HTTP* address if you don't have SSH Keys.
And you if have those, check out the settings tab. You know what to do.



3. You should have a folder named *homework* now.

```
% cd homework
% git checkout -b homework1
```

   after these two commands, you are good to go, code away for this project.
4. After you have done with the project

```
% git commit -am "Finished homework 1"
% git push
```

   Remember to push your code changes. Send the files to the server so that I can see them.
5. For all the general version control portion of Git, please take a look at the general information page (https://www.cs.columbia.edu/~du/ds/homework-instructions.html)

Written Problems↩

1. Contrast TCP and UDP. Under what circumstances would you choose one over the other?
2. What's the difference between caching and data replication?
3. Why do we need an application-level cache to optimize programs, i.e. why are the benefits of application-level cache over hardware or os-level cache?

Programming Problems↩

Notice

This is the first assignment of the yfs series homework. It requires basic Unix network programming knowledge. If you are not able to finish this homework on your own, you should seriously reconsider taking this course.

All the programming assignments need be written in C++.

Introduction

In this homework, you will implement both a TCP client for retrieving files, and a TCP server for serving the requested files. The client, after establishing a connection with the server, sends over the name of the file it requests. Then, the server searches for the file in its memory cache followed by a predefined directory. If the file is found in either of the places, the server then transmits the content of file back to the client via the same connection, and caches the file content, if it is not inside the memory cache.

The size limitation of the memory cache is 64MB. If the file size is more than 64MB on disk, you do not cache it in memory. And if the cache size will exceed 64MB after putting the file into memory, you have to purge enough cached content to make sure the entire cache size is restricted to 64MB anytime.

Include a `README` file along with your code. You can start your assignment by writing down in that file the following information:

- An overview which include the structure of your program
- The details of cache implementation
- Different test cases you use to test your program: e.g. requesting an invalid file, requesting from a dead server

Homework Overview

The flow of the program should look like the following description:

1. The server side creates a socket, binds it, and listens to a predefined port. After that, it blocks until a client connects.

   ```
   % tcp_server port_to_listen_on file_directory
   e.g.
   % ./tcp_server 9089 /home/dist/homework1
   ```

   The above argument will make the server listen on port `9089` and look up file in `/home/dist/homework1`.
2. The client connects to the server and sends an request for a file. The client side takes 4 parameters - the server name, server port, the file to request, and the local directory to save the file in. It looks like this:

   ```
   % tcp_client server_host server_port file_name directory
   e.g.
   % ./tcp_client 59.78.58.28 9089 homework1.html .
   ```

   The above argument will make the client connect server `59.78.58.28:9089` and request file `homework1.html` and save it in the current directory.
3. Once a client is connected, the server will parse the request and figures out which file the client is asking for. It prints a line `"Client xxx.xxx.xxx.xxx is requesting file X"`, where `xxx.xxx.xxx.xxx` is the client IP address and X is the file name being requested.
   - If the file is in the cache, it returns the file directly and outputs `"Cache hit. File X sent to the client."` If not, it will find the file in the predefined directory and save it in the cache, and then return it to the client and output `"Cache miss. File X sent to the client"`, where X is the file name.
   - If the file does not exist at all, it will return an error message. It prints a line

`"File X does not exist"`, where X is the file name. Note that the server only looks up the file in the given directory, not other directories.

A typical output on the server side looks like this:

```
% ./tcp_server 9089 /home/dist/homework1
Client 59.78.55.65 is requesting file homework1.html
Cache miss. homework1.html sent to the client
Client 59.78.55.66 is requesting file homework1.html
Cache hit. homework1.html sent to the client
Client 59.78.55.66 is requesting file homework1.html
Cache miss. homework1.html sent to the client
Client 59.78.55.65 is requesting file homework1.html
Cache hit. homework1.html sent to the client
```

4. The client side for response sending the request to the server. When the response comes, the client either saves it as a local file, and prints a line `"File X saved"`, where X is the local filename, or it will output an error message if the file does not exist on the server. In such case, it prints a line `"File X does not exist in the server".`

   A typical output on the client side looks like this:

```
% /tcp_client 59.78.58.28 9089 homework0.html .
homework0.html saved
```

You do NOT need to implement a multi-threaded server - a single thread version will be fine.

You are required to handle error gracefully. You will need to check the return value of each function, such as bind and receive. Should there be any error, you will print out the corresponding error message.

Hints

- You may find std::map (http://www.cplusplus.com/reference/stl/map/) useful when implementing the cache. Note that all files being requested are in the same directory, meaning that they all have different file names, so the key of map can be the file name, and the value would be the content of the file. Be extra careful with the std::map::operator[] (http://www.cplusplus.com/reference/stl/map/operator[]/), as this will insert a new element to the map if the key does not exist in the current map. You may simply use std::map::find (http://www.cplusplus.com/reference/stl/map/find/) instead.
- You will need to keep track of the memory usage of the cache as we require that the cache never exceed 64 MB. Thus, you will need to implement an appropriate cache replacement policy. The LRU (Least Recently Used) algorithm will be efficient, but a simple random replacement algorithm is also OK.

- If you need some refreshing with UNIX socket programming, you may find this guide (ftp://gaia.cs.umass.edu/cs653/sock.ps) useful. Section 1 through 4 will be enough for this homework.
- Do not use strncpy (http://www.cplusplus.com/reference/clibrary/cstring/strncpy/) when transferring data from disk to cache. Use memcpy (http://www.cplusplus.com/reference/clibrary/cstring/memcpy/) instead. The strncpy (http://www.cplusplus.com/reference/clibrary/cstring/strncpy/) behaves incorrectly when dealing with binary files.

Grading

NOTE: This first homework is graded Pass/Fail only

We will compile and test your code in Ubuntu X86_64 LTS 12.04.2 environment, namely clic machines. Failure to compile in such environment will lead to a 0! Read the instructions (https://www.cs.columbia.edu/~du/ds/homework-instructions.html) for more information how to test your program both locally as well as in the clic environment

Your program will be graded on the following components:

1. **Correctness**: If the program behaves correctly, e.g. does it transfer the file in its entirety? does it cache the content properly?
2. **Coding Style**: Is your code readable with proper variable naming? Are there any messy codes around? Are the braces consistent?
3. **Documentation**: Are there sufficient comments in the code to make it understandable? Does the README contains succinct documentation describing the your solution?
4. **Error Handling**: Does the program output informative message when an error is occurred? Note: If any segfault is encountered, you'll get half the marks that you could have gotten. And if you output "No" for all the error messages, I'll just count those and deduct the sum.

Make sure you read the Policies (https://www.cs.columbia.edu/~du/ds/policies.html) before you begin. You must write all of your code yourself. We use MOSS (http://theory.stanford.edu/~aiken/moss/) for Software Plagiarism. You can try it out yourself

homework 7 (https://www.cs.columbia.edu/~du/ds/categories.html#homework-ref)

homework 7 (https://www.cs.columbia.edu/~du/ds/tags.html#homework-ref)

homework1 1 (https://www.cs.columbia.edu/~du/ds/tags.html#homework1-ref)

← Previous | Archive (https://www.cs.columbia.edu/~du/ds/archive.html)
Next → (https://www.cs.columbia.edu/~du/ds/homework/2013/09/12/Homework2)

Generated by Jekyll Bootstrap (http://jekyllbootstrap.com) and Twitter Bootstrap (http://twitter.github.com/bootstrap/)