Project 3: HTTP Optimizations

Introduction

In this project, you will implement an HTTP client, replacing the browser that you used in Project 2. You will implement two optimizations employed by HTTP: multiple connections and pipelining.

Be prepared: this is a single-person project, and your skills will be exercised. So start early and feel more than welcome to ask questions. However, please note that the TAs are not allowed to debug your code for you during their office hours. They cannot touch your keyboard, and they will only spend a maximum of 10 minutes reading your code. This helps them to assist more students effectively. Therefore, it is recommended that you visit their office hours with specific questions instead of vague statements like "my code doesn't work." For debugging, logging or tools like Wireshark[1] can be helpful. There are many ways to do this; be creative.

Starting the Server

To test your implementation, you will need to use the server you implemented in Project 2. Alternatively, you can also set up a Nginx server[2] in your Linux environment. Follow this instruction to install and configure Nginx in Ubuntu: https://ubuntu.com/tutorials/install-and-configure-nginx.

Note that you'll need to replace the default root folder for Nginx `/var/www/` with your own path to the www folder. You could also copy all the files in your www folder to `/var/www/`.

Starting the Client

The client also takes a single command line argument: the IP address of the HTTP server. You may not modify it to, e.g., take a different number of arguments or reorder the arguments.

usage: ./client <server-ip>

[1] https://www.wireshark.org/
[2] https://www.nginx.com/

Pipelining and Parallel Connections

HTTP pipelining and parallel connections are the two HTTP optimizations you will implement.

- **Pipelining**[3]. HTTP pipelining is a feature of HTTP/1.1 that allows multiple HTTP requests to be sent over a single TCP connection without waiting for the corresponding responses. HTTP allows more than one in-flight request.

- **Parallel Connections**[4]. With parallel connections, one client can simultaneously create multiple connections to the server, improving the page load time in some scenarios.

For 325 students, you only need to implement one of the optimizations; you will get extra credits for implementing both. For 425 students, you will need to implement both.

Understanding dependencies: When using the browser as a client, the client first requests the index.html file. The client then parses `index.html` to determine what resource(s) to request next. Faithfully implementing HTML dependency parsing is cumbersome; instead, we provide a `dependency.csv` file, describing all the files in the current webpage and their dependencies. As long as you request dependency.csv, you will know the elements that the client needs to request. The first column of the `dependency.csv` is the name (URI) of the element, and the second column is its dependency (empty if the client can directly request the file). In your client implementation, you must make a GET request for `dependency.csv`, parse the dependency graph, and then issue HTTP requests adhering to the dependency relationships between resources.

Useful Links

Programming in Python

The 8th edition of the textbook (and what we've discussed in class) uses sockets in Python. A Python socket tutorial is http://docs.python.org/howto/sockets.html

[3] https://en.wikipedia.org/wiki/HTTP_pipelining
[4] https://www.oreilly.com/library/view/http-the-definitive/1565925092/ch04s04.html

It may also help by reading the system implementation of epoll at https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/sys/epoll.h.html. You will better understand the epoll events.

REMINDERS

- Submit your code to Canvas in a zip file
- If your code does not **run**, you will not get credits.
- Document your code (by inserting comments in your code)
- DUE: 11:59 pm, Friday, April 4th.