Project 1

Prof: Deborah Estrin, Vitaly Shmatikov,
TA: Eugene Bagdasaryan <ebagdasa@cs.cornell.edu>

Due Date: March 1st 11:59 PM

1 Note

For all private/public questions please use Piazza group: `https://piazza.com/cornell/spring2017/cs5450/home`

2 Introduction

The purpose of this project is to give you experience in developing concurrent network applications. You will use the Berkeley Sockets API to write a web server using a subset of the HyperText Transport Protocol (HTTP) 1.1 Request for Comment-RFC 2616. This set of features forms the core of the Liso web servers capabilities and will be the focus of the first part of the course. Why are we doing this? Because web applications are becoming increasingly important today. Many startups and businesses are based entirely on web applications. Understanding HTTP, how web servers work. Be prepared: this is a single person project and it has a lot of depth-your skills will be exercised (perhaps to their limits). So start early and feel more than welcome to ask questions. This is a hard project, we have an implementation in mind, so both TAs and the instructors will be more than welcome to help you debug, design, and provide hints in working on this project.

3 Logistics

- Source materials for this project may be found in hosted here:
  https://pages.github.coecis.cornell.edu/cs5450/website/
- This is a solo project. You must implement and submit your own code.
- All of your project files and submissions must be stored in a git repository.
- You will submit your code as a tarball named $< NetID >$.tar. Untarring this file should give us a directory named *project-1* which should contain the git repository as well as the code. You will submit this tarball using CMS. Checkpoint is designed to ensure that you keep tabs on your progress and are a great guideline to help you complete your project on time.

- Checkpoint: (1) Create a git repo named project-1 (mkdir project-1; cd project-1; git init), (2) code a select()-based echo server handling multiple clients at once (building on the supplied echo server), and (3) hand-in your submission by the deadline and include all needed files:
  - Makefile - make sure nothing is hard coded specific to your user; should build a 'lisod' file which runs the echo server
  - All of your source code - all .c and .h files
  - readme.txt - file containing a brief description of your current implementation of lisod
  - tests.txt - file containing a brief description of your testing methods for lisod
  - vulnerabilities.txt - identify at least one vulnerability in your current implementation

  To aid you in programming an echo server, and testing it, we have prepared this starter package for you. This code needs to be modified to use select() as well as adding support for multiple clients at once. Additional information will be available at `https://pages.github.coecis.cornell.edu/cs5450/website/assignments/lab1.html`

- Final Submission: (1) Implement an HTTP 1.1 parser and persistent connections with HEAD, GET, and POST working. At this point as we dont have CGI we will not check responses to your POST requests but rather your ability to handle the request Body.(2) upload your submission by the deadline. Additional information will be available at `https://pages.github.coecis.cornell.edu/cs5450/website/assignments/lab1.html`

4 HTTP Server

Your server will implement HEAD, GET (both needed for HTTP 1.1 general purpose server compliance), and POST, which we are adding. This should comply with the specification in the RFC.

HTTP 1.1

- GET requests a specified resource; it should not have any other significance other than retrieval

- HEAD asks for an identical response as GET, without the actual body-no bytes from the requested resource

- POST submit data to be processed to an identified resource; the data is in the body of this request; side-effects expected

For all other commands, your server must return "501 Method Unimplemented." If you are unable to implement one of the above commands (perhaps you ran out of time), your server must return the error response "501 Method Unimplemented," rather than failing silently (or not so silently). While you develop, you may want to just return this error response always until features are implemented - no matter what you will have a valid HTTP 1.1 server! Your server should be able to support multiple clients concurrently. The only limit to the number of concurrent clients should be the number of available file descriptors in the operating system (the min of ulimit -n and FD SETSIZE-both typically 1024). We will not be testing beyond 1024 concurrent connections. While the server is waiting for a client to send the next command, it should be able to handle inputs from other clients. Also, your server should not hang up if a client sends only a partial request. In general, concurrency can be achieved using either select() or multiple threads. However, in this project, you must implement your server using select() to support concurrent connections. Threads are NOT permitted at all for the project. See the resources section below for help on these topics. As a public server, your implementation should be robust to client errors. For example, your server should be able to handle malformed requests which do not have proper [CR][LF] line endings. The provided lex and yacc parser is not robust to these malformed requests and you are expected to add the necessary rules if you decide to go-ahead and use the provided parser. It must not overflow any buffers when the client sends a message that is "too long." In general, your server should not be vulnerable to a malicious client. This is something we will test for. You are implementing a real, standards-compliant web server. Therefore, comparing protocol exchanges to existing web servers is both valid and encouraged. Install Apache, install Wireshark, and sniff the protocol exchanges and compare to your own-even use captured web browser requests to replay from files you save as input to your implementation of Liso. Come up with other create ways to test as well as there is a portion of the grade reserved for testing.

5 Getting Started

This section gives suggestions for how to approach the project. Naturally, other approaches are possible, and you are free to use them.

- Start early! The hardest part of getting started tends to be getting started. Remember the 90-90 rule: the first 90% of the job takes 90% of the time; the remaining 10% takes the other 90% of the time. Starting early gives you time to ask questions. For clarifications on this assignment, post to Piazza and read project updates on the course web page. Talk to your classmates. While you need to write your own original program, we expect conversation with other people facing the same challenges to be very useful. Come to office hours. The course staff is here to help you.

- Read the RFCs selectively. RFCs are written in a style that you may find unfamiliar. However, it is wise for you to become familiar with it, as it is similar to the styles of many standards organizations. We dont expect you to read every page of the RFC, especially since you are only implementing a small subset of the full protocol, but you may well need to re-read critical sections a few times for the meaning to sink in.

- Begin by taking a cursory first pass over the RFCs. Do not focus on the details; just try to get a sense of how they work at a high level. Understand the role of the server.

Understand what error conditions are possible, and how they are used. You may want to print the RFCs, and mark them up to indicate which parts are important for this project, and which parts are not needed. You may need to re-read these sections several times.

- Next, take a second pass over the RFCs. You will want to read all of them together. Again, do not focus on the details; just try to understand the requests and responses at a high level. As before, you may want to mark up a printed copy to indicate which parts of the RFCs are important for the project, and which parts are not needed.
- Now, go back and read with an eye toward implementation. Mark the parts which contain details that you will need to write your server. Start thinking about the data structures (input and output buffers, etc.) your server will need to maintain. What information needs to be stored about each client while servicing requests (maybe an HTTP 1.1 finite state machine per client, etc.)?
- Get started with a simple server that accepts connections from multiple clients. It should take any message sent by any client, and "echo" that message back to its sender. This server will not be compatible with HTTP clients, but the code you write for it will be useful for your final server. Writing this simpler server will let you focus on the socket programming aspects of a server without worrying about the details of the protocols. Test this simple server with the simple provided Python script for Checkpoint 1.
- At this point, you are ready to write a standalone HTTP 1.1 server. But do not try to write the whole server at once. Decompose the problem so that each piece is manageable and testable. For each request, identify the different cases that your server needs to handle. Find common tasks among different commands and group them into procedures to avoid writing the same code twice.

6 Resources

For information on network programming, the following may be helpful:

- Beejs Guide `http://beej.us/guide/bgnet/output/html/multipage/index.html`
- Class Piazza `https://piazza.com/cornell/spring2017/cs5450/home`
- Class Website `https://github.coecis.cornell.edu/cs5450/website`
- BSD Sockets: A Quick And Dirty Primer `https://cis.temple.edu/~giorgio/old/cis307s96/readings/docs/sockets.html`
- man pages
  - Installed locally (e.g. man socket)
  - Available online: the Single Unix Specification
- Other Google Groups / Stackoverflow - Answers to almost anything