15-441/641: Computer Networks
Project 1: A Web Server Called Liso

TAs: Kenneth Yang <canyang@andrew.cmu.edu>
Viswesh Narayanan <visweshn@andrew.cmu.edu>

Assigned: August 30, 2017
Checkpoint 1 due: September 11, 2017
Checkpoint 2 due: September 22, 2017
Final version due: October 6, 2017

1 Introduction

The purpose of this project is to give you experience in designing and 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 [1]. Your web server will also implement HyperText Transport Protocol Secure (HTTPS) via Transport Layer Security (TLS) as described in RFC 2818 [2]. The final part of the project will implement the Common Gateway Interface (CGI) as described in RFC 3875 [3]. This set of features forms the core of the Liso web server's capabilities. Through this project, you will have the opportunity to synthesize the provided specifications into a carefully designed implementation. New this year is the submission of a short design document that describes your design at a high level.

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, HTTPS, and CGI will give you a deep understanding of the core web technologies underlying and fueling much of the Internet's growth. In addition, your Liso web server will be fully functional and capable of running interactive web applications via its CGI interface. At the end of the project the final test will be running a simple web blog written using the Python Flask microframework [4].

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.

2 Logistics

- Source materials for this project may be found in hosted here: `http://www.cs.cmu.edu/~prs/15-441-F17/assignments.html`
- 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 `<andrewID>.tar`. Untarring this file should give us a directory named `15-441-project-1` which should contain the git repository as well as the code. You will submit this tarball using Autolab (`https://autolab.andrew.cmu.edu`). If you still can't login with your andrew ID by the end of September 4, let us know ASAP.

  Checkpoints are designed to ensure that you keep tabs on your progress and are a great guideline to help you complete your project on time. Please note that the Checkpoint 1 is straightforward part of the total work. It helps you familiarize yourself with the basics of socket programming (specifically how to use `select()` system call) and will require some amount of RFC interpretation for implementing a basic HTTP 1.1 parser. The rest of the checkpoints are progressively harder and will build on the previous checkpoints.
- **Checkpoint 1:** (1) Create a git repo named 15-441-project-1 (`mkdir 15-441-project-1; cd 15-441-project-1; git init`), (2) code a `select()`-based echo server handling multiple clients at once (building on the supplied starter code). Specifically, your server must have the capability to parse HTTP 1.1 requests and classify them as good and bad based on the provided RFC [1]. For all good requests, you will simply echo back the original request. For all bad requests, you will return an HTTP response with 400 as error code. (3) Finally, hand-in your submission by the deadline and include all needed files as outlined in §6.
  To aid you in programming, and testing it, we have prepared this starter package `https://www.cs.cmu.edu/~prs/15-441-F17/project1/checkpoint1.tar.gz` for you. It contains the starter codes for an echo server and HTTP parser. This code needs to be modified to use select() as well as adding support for multiple clients at once. Also, the parsing rules need to be enhanced to support the RFC specification [1]. Additional information will be available at `http://www.cs.cmu.edu/~prs/15-441-F17/pj1cp1.html`
- **Checkpoint 2:** (1) Upgrade the your server to respond properly to any HTTP 1.1 request and implement persistent connections with HEAD, GET, and POST working. At this point as we don't 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 `http://www.cs.cmu.edu/~prs/15-441-F17/pj1cp2.html`
- **Final Submission:** (1) implement HTTPS handshaking and persistent connections via TLS, (2) implement CGI server-side, and (3) upload your submission by the deadline. Additional information will be available at `http://www.cs.cmu.edu/~prs/15-441-F17/pj1c3.html`

3 The Liso Server

Your server will implement HEAD, GET (both needed for HTTP 1.1 general purpose server compliance), and POST. This should comply with the specification in the RFC [1]. After

this we will move on into implementing TLS, and finally CGI.

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 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. For example, it must not overflow any buffers when a malicious client sends a message that is "too long." These are 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 [5], install Wireshark [6], 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.

4 Implementation Details and Usage

Your server must be written in the C programming language. You are not allowed to use any custom socket classes or libraries, only the standard socket library and the provided library functions. You **may not** use the csapp wrapper library from 15-213/15-513 or libpthread for threading. We disallow csapp.c for two reasons: first, to ensure that you understand the

raw standard BSD sockets API, and, second, because csapp.c's wrapper functions are not suitable for robust servers. Temporary system call failures (e.g., EINTR) in functions such as `select()` could cause the server to abort and utility functions like `rio_readlineb` are not designed for nonblocking code.

That said, we encourage the use of *anything* for testing. Use Wireshark [6], use telnet, use real web browsers, use Python to script tests—for testing, the sky is the limit.

4.1 Compiling

You are responsible for making sure your code compiles and runs correctly on the Andrew x86 machines running Linux (i.e., `linux.andrew.cmu.edu` / `unix.andrew.cmu.edu`). We recommend using `gcc` to compile your program and `gdb` to debug it. You should use the `-Wall` and `-Werror` flags when compiling to generate full warnings and to help debug. Other tools available on the Andrew unix machines that are suggested are ElectricFence [10] (link with `-lefence`) and Valgrind [11]—use this with full leak checking to ensure you have no memory leaks. For this project, **you will also be responsible for turning in a GNU Make compatible Makefile**. See the GNU make manual[8] for details. When we run `make` we should end up with the Liso web server binary `lisod`.

4.2 Command Line Arguments

Liso will always have 8 arguments—functional or not(It may not be for the initial checkpoints):

usage: *./lisod <HTTP port> <HTTPS port> <log file> <lock file> <www folder> <CGI script path> <private key file> <certificate file>*

HTTP port – the port for the HTTP (or echo) server to listen on

HTTPS port – the port for the HTTPS server to listen on

log file – file to send log messages to (debug, info, error)

lock file – file to lock on when becoming a daemon process

www folder – folder containing a tree to serve as the root of a website

CGI script name (or folder) – for this project, this is a file that should be a script where you redirect all /cgi/* URIs. In the real world, this would likely be a directory of executable programs.

private key file – private key file path

certificate file – certificate file path

4.3 Running

The Liso server will be passed the ports to run on, what log file to use, what lock file to use when daemonizing, folders to serve static data from as well as CGI applications, and TLS private/public key pairs.

Not all of these options need to be functional at each stage of development. Only a port is needed for the first checkpoint when implementing an echo server using `select()`.

4.4 Framework Code

We will provide you with framework code that will, for example, help in forking a process for proper CGI handling and setting up the environment, parse commandline arguments (and sanity check them) and daemonize a process.

DISCLAIMER: We reserve the right to change the support code as the project progresses to fix bugs and to introduce new features that will help you debug your code. You are responsible for checking Piazza to stay up-to-date on these changes. We will assume that all students in the class will read and be aware of any information posted to Piazza.

4.5 HTTP Packet Parsing

Historical evidence suggests that most students spend considerable amount of time writing correct parsers. While parsing packets using C's string manipulation functions may well be an essential skill to have, it might get insanely tedious. We want you to spend time on other more important programming aspects such as socket programming, handling race conditions and memory leaks. For this reason, we require you to use Lex and Yacc for parsing packets. We will also provide you with a basic HTTP parser written in lex and yacc. More about parsing using Lex and Yacc will be covered in recitations. So, stay tuned!

5 Testing

Code quality is of particular importance for server robustness in the presence of client errors and malicious attacks. Thus, a large part of this assignment (and programming in general) is knowing how to test and debug your work. There are many ways to do this; be creative. We would like to know how you tested your server and how you convinced yourself it actually works. To this end, you should submit your test code along with brief documentation describing what you did to test that your server works. The test cases should include both generic ones that check the server functionality and those that test particular corner cases. If your server fails on some tests and you do not have time to fix it, this should also be documented (we would rather know that you are aware of the limitations of your server than think you missed a serious flaw). Several paragraphs (or even a bulleted list of things done and why) should suffice for the test case documentation.

We will be providing test scripts for each checkpoint and also the final finished server. Note however that grading will be based on additional tests that will not be provided to you. This handout lists what functions and properties of your project will be tested.

Here are some simple starting points for scripting your own external tests:

netcat

You may use netcat to send arbitrary files to your server and receive responses. Use regular bash redirection (< and >) along with ncat to achieve this.

Read `man ncat` for more information.

expect

Quoting from the expect man page,

Expect is a program that "talks" to other interactive programs according to a script. Following the script, Expect knows what can be expected from a program and what the cor-

rect response should be. An interpreted language provides branching and high-level control structures to direct the dialogue.

Python socket

This is a very simple and easy to use Python module for creating and interacting with sockets. We have used this in the first checkpoint testing script provided to you for testing your implementation of an echo server. This will be used for creating future testing programs which we will release leading the schedule for deadlines.

You can read about this module here: `http://docs.python.org/library/socket.html`.

In addition for testing HTTP, there is a urllib2 library in Python. You can also use the requests library to create requests for your server.

6 Hand-In

Handing in code for checkpoints and the final submission deadline will be done through Autolab (`https://autolab.andrew.cmu.edu`). You are supposed to upload the tarball file for each checkpoint into the corresponding assessment on our course page of Autolab website.

6.1 Work with git

You are supposed to create your git repo on your local machine or on a **private** git repo hosted online as part of Checkpoint 1. Every checkpoint will be a git tag in this repo. To create a tag, run

`git tag -a checkpoint-<num> -m <message> [<commit hash>]`

with appropriate checkpoint number and custom message filled in. (Put whatever you like for the message — git won't let you omit it.) The optional commit hash can be used to specify a particular commit for the tag; if you omit it, the current commit is used. If you choose to clone your repository onto your local machine for development, be sure to use `git push --tags` to sync your work back to git server; the standard `git push` doesn't send tags.

6.2 Upload your code

To submit your code, make a tarball file of you repo after you tag it. Then login to autolab website, choose `15-441: Computer Networks (f17) -> project1cp<N>` and then upload your tarball. The grader should be finished less than a minute but may take longer depending on system load. When it is done, your score will be shown. Only the latest score will be used.

Untarring the tarball should give us a directory named `15-441-project-1` which contains a valid git repo with tags. Your repo should contain the minimum following files:

- **Makefile** – Make sure all the variables and paths are set correctly such that your program compiles in the handin directory—not just a local machine or account. The Makefile should, by default, always build an executable named lisod.
- **All of your source code** – (files ending with .c, .h, etc. only, no .o files and no executables)

- **readme.txt** – File containing a brief description of your source tree organization.
- **design.pdf** – File containing a detailed description of your design of your current version of lisod. Refer the course website for more details.
- **tests.txt** – File containing documentation of your test cases and any known issues you have.
- **vulnerabilities.txt** – File containing documentation of at least one vulnerability you identify at each stage.

Late submissions will be handled according to the policy given in the course syllabus.

7 Grading

We will be providing some of the scripts but will also be running **additional tests**. Half of the points in each category (HTTP 1.1, HTTPS via TLS etc) will be based on the provided test scripts and the other half will be based on additional scripts that will be run.

- **Server core networking:** 20 points

  The grade in this section is intended to reflect your ability to write the "core" networking code. This is the stuff that deals with setting up connections, reading/writing from/to them (see the resources section below). Even if your server does not implement HTTP 1.1 etc., your project submission can get up to 20 points here. It is better to have partial functionality working solidly than lots of code that doesn't actually do anything correctly.

  Have a working `select()`-based foundation, and recieve full credit here.

- **HTTP 1.1:** 20 points

  The grade in this section reflects how well you read, interpreted, and implemented HTTP 1.1. We will test all the requests specified in Section 3: HEAD, GET and POST. All requests sent to your server for this part of the testing will be valid. So a server that completely and correctly implements the specified commands, even if it does not check for invalid messages, will receive 20 points here.

  We will extensively test correct behavior for HEAD, GET, POST, and persistent connection handling. Feel free to check things via web browsers at this point.

- **HTTPS via TLS:** 10 Points

  The grade in this section reflects how well you read, interpreted, and implemented the TLS protocol for HTTP.

  Point a web browser at your server: https://xxx.xxx.xxx.xxx and verify correct connection. Obviously you will not have Certificate Authority (CA) signed certificates, but this stage should work with any web browser provided you acknowledge the security warnings and ignore them. In addition, the standard requests HEAD, GET, POST, and persistent connections should all work as before for the HTTP 1.1 implementation.

- **CGI:** 15 points

  The grade in this section reflects how well you read, interpreted, and implemented the CGI interface. This will be tested via a Python WSGI application which you can also run to verify your implementation is correct. It's a blog. Make sure you can perform all the operations it supports (such as displaying blog entries and adding new entries).

- **Robustness and Performance:** 15 points
  - Server robustness: 10 points
  - Performance: 5 points

  Since code quality is of a high priority in server programming, we will test your program in a variety of ways using a series of test cases. For example, we will send your server very long messages to test if there is a buffer overflow. We will make sure that your server does something reasonable when given an unknown request or a request with invalid headers. We will verify that your server correctly handles clients that leave abruptly without sending the proper "close" header line in HTTP 1.1. We will test that your server correctly handles concurrent requests from multiple clients without blocking inappropriately. The only exception is that your server may block while doing DNS lookups, reads from the file system, or during the execution of CGI programs.

  We will have tools that replay HTTP 1.1, TLS, and CGI interactions with your application. Note that there are many corner cases that the RFC does not specify. You will find that this is very common in "real world" programming since it is difficult to foresee all the problems that might arise. Therefore, we will not require your server pass all of the test cases in order to get a full credit on any part of the assignment. Autolab system will show you about the errors, though.

  We will also test to ensure that your code meets a minimum performance threshold. We will run a load generator against your server and see the throughput in Requests Per Second(RPS) that your server is able to sustain. You will get full credit if you are above the minimum threshold (TBD) and an exponentially decreasing grade if you are below the threshold. We do not want you to implement any form of caching in your web-server.

- **Design, Style and Your Tests: 20 points**
  - Design: 5 points
  - Style: 5 points
  - Your Tests: 10 points

  Poor design, documentation, or code structure will probably reduce your grade by making it hard for you to produce a working program and hard for the grader to understand it; egregious failures in these areas will cause your grade to be lowered even if your implementation performs adequately.

  We expect the file **design.pdf** to reflect the overall design of your web server. Specifically, we will be looking for how well you have modularized the components, the design

trade-offs you made along with a strong reasoning behind those choices.

Document code using Doxygen-style comments.

In some of our structured code examples, we showcase an underlying logging facility that logs to a configured file. Use something similar to this to keep traces of your server and debug.

We expect you to come up with test scripts and there are 10 points for test scripts/-manual testing(That you will document in tests.txt. We expect to see a list of tests in test.txt with a brief description of what each test is testing). Quantity of test scripts is not a criteria. We expect your test scripts to test different parts of your system thoroughly.

8 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 don't 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 provided Python script in Checkpoint 1.
- Next, enhance the starter code we have provided for HTTP parsing. Apply all the RFC knowledge you have gathered from previous steps and try to convert them into rules. After you combine this parser along with the echo server, you should be ready 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. You might start by implementing the routines that read and parse commands. Then implement commands one by one, testing each with `telnet`.
- Thoroughly test your server. Use the provided scripts to test basic functionality. For further testing, use `telnet`, a web browser, or replay scripts. Learn Python from our scripts and as we go to make repeatable "regression tests"—every time you implement a new feature you use regression tests to see if anything broke.
- Make sure to check the return code of all system calls and handle errors appropriately. Temporary failures (e.g., EINTR) should not cause your server to abort or exit in failure. Fatal errors can be dealt with via a perror() call and exiting—but try to clean up open file descriptors and sockets nicely even when fatally exiting.
- Be liberal in what you accept and conservative in what you send [9]. Following this guiding principle of Internet design will help ensure your server works with many different and unexpected client behaviors.
- Code quality is important. Make your code modular and extensible where possible. You should probably invest an equal amount of time in testing and debugging as you do writing. Also, debug incrementally. Write in small pieces and make sure they work before going on to the next piece. Your code should be readable and commented. Not only should your code be modular, extensible, readable, etc, most importantly, it should be your own!
- You may want to consider turning warnings into errors to avoid bad programming style. Do this by passing `-Werror` to `gcc` during compilation.
- If you have a question about a project handout or a technical issue, there is an excellent chance that other students have the same question. Please read Piazza to see if there has been traffic and consider posting your questions there.

9 Resources

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

- Beej's Guide [7]
- Class Textbook – Sockets, etc
- Class Piazza – Announcements, clarifications, etc
- Class Website – Announcements, errata, etc
- Computer Systems: A Programmer's Perspective (CS 15-213 text book)[12]
- BSD Sockets: A Quick And Dirty Primer[13]
- An Introductory 4.4 BSD Interprocess Communication Tutorial[14]
- Unix Socket FAQ[15]
- Sockets section of the GNU C Library manual
  - Installed locally: info libc
  - Available online: GNU C Library manual[16]
- man pages
  - Installed locally (e.g. man socket)
  - Available online: the Single Unix Specification[17]
- Other Google Groups / Stackoverflow - Answers to almost anything[18]

References

[1] RFC 2616: `http://www.ietf.org/rfc/rfc2616.txt`

[2] RFC 2818: `http://www.ietf.org/rfc/rfc2818.txt`

[3] RFC 3875: `http://www.ietf.org/rfc/rfc3875`

[4] Flask: `http://flask.pocoo.org/`

[5] Apache: `http://httpd.apache.org/`

[6] Wireshark: `http://www.wireshark.org/`

[7] Beej's Guide: `http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html`

[8] GNU Make Manual: `http://www.gnu.org/software/make/manual/make.html`

[9] RFC 1122 `http://www.ietf.org/rfc/rfc1122.txt`, page 11

[10] ElectricFence: `http://perens.com/FreeSoftware/ElectricFence/`

[11] Valgrind: `http://valgrind.org/`

[12] CSAPP: http://csapp.cs.cmu.edu

[13] http://www.cis.temple.edu/~ingargio/old/cis307s96/readings/docs/sockets.html

[14] http://docs.freebsd.org/44doc/psd/20.ipctut/paper.pdf

[15] http://www.developerweb.net/forum/forumdisplay.php?s=f47b63594e6b831233c4b8ebaf10a614&f=70

[16] http://www.gnu.org/software/libc/manual/

[17] http://www.opengroup.org/onlinepubs/007908799/

[18] http://groups.google.com