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

Assigned: Thursday, August 30, 2012
Checkpoint 1 due: September 13, 2012
Checkpoint 2 due: September 20, 2012
Final version due: September 27, 2012

1 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 [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 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, 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].

That's it. We're going for it! You will have a fully functional web server :-) But be prepared, this is a single person project and it has a lot of depth—your skills will be exercised perhaps to their limit! 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 course staff will be more than welcome to help you debug, design, and provide hints in working on this project.

2 Logistics

- This is a solo project. You must design and implement and submit your own code.
- Submissions will be via AFS into a handin directory. Stay tuned for instructions. (The use of a versioning system, such as git, is suggested during development).
- Checkpoint 1: Code a select()-based echo server handling multiple clients at once,
- Checkpoint 2: 1) implement an HTTP 1.1 parser and persistent connections with HEAD, GET, and POST working
- Checkpoint 3/Final Submission: 1) implement HTTPS handshaking and persistent connections via TLS, 2) implement CGI server-side

3 The Liso 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 [1].

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 in some other manner. 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). 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. 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 [5], install Wireshark [6], 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.

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, 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 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 [12] (link with -lefence) and Valgrind [13]—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[10] for details. When we run make we should end up with the Liso web server compiled lisod.

4.2 Command Line Arguments

Liso will always have 6 arguments—functional or not:

```
      usage: ./lisod <HTTP port> <HTTPS port> <log file> <lock file> <www folder> <CGI
folder or script name> <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 folder or script name – folder containing CGI programs—each file should be executable; if a file it should be a script where you redirect all /cgi/* URIs to
- private key file – private key file path
- certificate file – certificate file path

4.3 Running

This is how we will start your implementation of Liso:

```
      ./lisod 8080 4443 /tmp/lisod.log /tmp/liso.lock /tmp/www /tmp/cgi/flaskr.py
/tmp/priv.key /tmp/cert.crt
```

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. At first, 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), daemonize a process.

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 appreciate that you know and acknowledge the pitfalls of your server, than miss them). 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. 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 correct 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.

6 Handin

Handing in code for checkpoints and the final submission deadline will be done via AFS. We'll give you more details as the deadlines approach.

Your repository should contain the minimum following files:

- Makefile – Make sure all the variables and paths are set correctly such that your program compiles within the campus RHEL environment for any user– not just you, and not just your own machine. 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 design of your current version of lisod.
- tests.txt – File containing documentation of your test cases and any known issues you have.
- replay.test – File containing input to send to the server testing the implementation
- replay.out – File containing expected server output matching input given in replay.test
- vulnerabilities.txt – File containing documentation of at least one vulnerability you identify at each stage.

7 Grading

- Server core networking: 30 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 30 points here. It is better to have partial functionality working solidly rather 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: 15 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: 10 points
  - Server robustness: 5 points
  - Test cases: 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. In fact, each student must submit one test case for replaying against a protocol. Each server will be tested against the whole battery of tests (our tests + your fellow students' tests).

However, 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. We will notify you of errors though.

We will also look at your own documented test cases to evaluate how well you tested your work.

- Style: 5 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.

  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.

- Checkpoints: 5 points each and 5 points from each feature due

  Late policy DOES apply to the checkpoints. However, considering the fact that you only have 2 late days for the entire semester, we strongly encourage you to plan ahead and not to use late days for checkpoints.

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—at your own peril.

• 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 your time to ask questions. 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 reread these sections several times.

• Next, take a second pass over the RFCs. You will want to read 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 to all clients (including the sender of the message). 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. A correct implementation of the simple server gives you approximately 30 points for the core networking part.

• 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 cleanup open file descriptors and sockets nicely even when fatally exiting.

• Be liberal in what you accept, and conservative in what you send [11]. 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.

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] GNU Flex: http://flex.sourceforge.net/manual/
[8] GNU Bison: http://www.gnu.org/s/bison/manual/bison.html
[9] Beej’s Guide: http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html
[10] GNU Make Manual: http://www.gnu.org/software/make/manual/make.html
[11] RFC 1122 http://www.ietf.org/rfc/rfc1122.txt, page 11
[12] ElectricFence: http://perens.com/FreeSoftware/ElectricFence/
[13] Valgrind: http://valgrind.org/
[14] CSAPP: http://csapp.cs.cmu.edu
[15] http://www.cis.temple.edu/~ingargio/old/cis307s96/readings/docs/sockets.html
[16] http://docs.freebsd.org/44doc/psd/20.ipctut/paper.pdf
[17]
http://www.developerweb.net/forum/forumdisplay.phps=f47b63594e6b831233c4b8ebaf10a614&f=70
[18] http://www.gnu.org/software/libc/manual/
[19] http://www.opengroup.org/onlinepubs/007908799/
[20] http://groups.google.com