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

TAs: Mingran Yang (mingrany@andrew.cmu.edu)
Alex Bainbridge (abainbri@andrew.cmu.edu)

August 30, 2019

1 Introduction

In this class you wil learn about Computer Networks, from bits on a wire or radio ('the bottom') all the way up to the design of applications that run over networks ('the top)'. In this project, you will start from the top with an application we are all familiar with: a web server.

You will use the Berkeley Sockets API to write a web server using a subset of the HyperText Transport Protocol (HTTP) 1.1 —RFC 2616 [2]. Your web server will also implement HyperText Transport Protocol Secure (HTTPS) via Transport Layer Security (TLS) as described in RFC 2818 [3]. Students enrolled under the 15-641 course number will implement the Common Gateway Interface (CGI) as described in RFC 3875 [4]. Reading an RFC is quite different from reading a news article, mystery novel, or even technical paper. Appendix B.2 has some tips read and use and RFC efficiently.

RFCs are one of many ways the Internet declares 'standards:' agreed upon algorithms, wire formats, and protocols for interoperability between different implementations of systems on a network. Without standards, software from one company would not be able to talk to software from another company and we would not have things like e-mail, the Web, or even the ability to route traffic between different companies or countries. Because your web server is compatible with these standards, you will, by the end of this project, be able to browse the content on your web server using standard browser like Chrome or Firefox.

With this project, you will gain experience in:

- ... building non-trivial computer systems in a low-level language (C).
- ... complying with real industry standards as all developers of Internet services do.
- ... reasoning about the many tasks that application developers rely on the Internet to do for them.

To guide your development process, we have divided this project into three key checkpoints.

CP	**Goal**	**% P1 Grade**		**Deadline**	
		441	641	441	641
1	Determine whether an HTTP 1.1 request is valid or invalid.	25%	15%	Sep 10, 2019	Sep 6, 2019
2	Respond correctly to HTTP 1.1 HEAD and GET requests.	75%	50%	Sep 27, 2019	Sep 20, 2019
3	Support HTTPS and CGI, run as a daemon, and complete all previous tasks.	-	35%	-	Sep 27, 2019

At each checkpoint (CP), we will assess your server to verify that it implements all of the goals for the current checkpoint and we will rerun tests from the previous checkpoint(s) as well. This means that if you fix a bug you had in an earlier checkpoint, you will get partial credit in a later checkpoint. Of course, if you

break a feature that was originally implemented correctly, you will lose some points, so you should rerun your tests from earlier checkpoints before you submit a later checkpoint.

There are a few hard and fast requirements for your project. Your code must use the select() function. The use of threads is not allowed, expect for the CGI implementation within CP3. Your code must be implemented in C. If your server does not meet these requirements you will receive a 0 for the project.

2 The Liso Server

In this section, we give an overview of the the complete requirements for the Liso server: what you will turn in at the final checkpoint. In the next three sections (§3–5), we break the project into a development strategy with intermediate checkpoints. The starting point for your server is framework code that we will provide on the course website and in Autolab.

2.1 Supporting HTTP 1.1

You will find in RFC 2616 that there are many HTTP methods: commands that a client sends to a server. Liso will only support three methods:

- **GET** – requests a specified resource; it should not have any 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

Error Messages: 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).

Robustness: As a public server, your implementation should be robust to client errors. Even if the client sends malformed inputs or breaks off the connection mid-request, your server should never crash. For example, your server must not overflow any buffers when a malicious client sends a message that is "too long." The starter code we will provide you is not robust to malformed requests (e.g, it cannot handle requests which do not have proper [CR][LF] line endings) and so you will have to extend it.

Multiple Requests: During a given connection, a client may send multiple HEAD/GET/POST requests. The client may even send multiple requests back-to-back, without even waiting for a response. This is called 'HTTP pipelining' [22]. Your server must support multiple requests in the same connection, and it must support HTTP pipelining.

2.2 Supporting Many Clients

Your server should be able to support multiple clients concurrently. You should set the maximum number of connections that can be supported by your server to 1024 (which corresponds to the typical number of available file descriptors in most operating systems). We will not test your server with more than 1024 clients simultaneously.

While the server is waiting for a client to send the next command, it should be able to handle inputs from other clients. Clients may 'stall' (send half of a request and then stall before sending the rest) or cause errors; these problems should not harm other concurrent users. For example, if a client only sends half of a request and stalls, your server should move on to serving another client. 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.

2.3 HTTPS and CGI

Students enrolled in 15-641 will extend your basic Liso server with support for HTTPS (which encrypts connections to your server) and CGI (which allows your server to support interactive programs) in Checkpoint 3.

HTTPS Support: Your server will use the OpenSSL library for HTTPS support. Specially, you will wrap communication calls with with SSL wrapping functions that will encrypt data sent over the channel and authenticate the server based on a certificate. In order to test HTTPS, you will need to get a certificate that your HTTPS-enabled server can use to authenticate itself to clients.
CGI Support: The Common Gateway Interface (CGI) provides a standard interface that allows a web server to call other processes or servers. Web servers use this typically to generate dynamic content, i.e., the content that is generated is based on input provided by the client (using POST) or other client-specific information. Your server will support CGI requests and will provide a built in 'ASCII art generator' as a demo application. When a client submits a POST to the ASCII art generator, the server will generate a page that spells out client-specified phrase in ASCII art.

More details on HTTPS and CGI are in §5.

2.4 Command Line Arguments

Liso will always have 8 arguments. You may not modify Liso to, e.g., to take a different number of arguments or to reorder the arguments. If you do not need one of the arguments in the earlier checkpoints, you may simply ignore it in your code.

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

2.5 Tools, Skills, and Grading

The projects in this course are significant larger and more complex than the 15-213/513 projects. They are also more open-ended, in the sense that we only specify *what* the system must do, but not *how* you do it. For example, a web server needs to perform several functions, e.g., managing sessions, using the file system, generating responses including error messages, etc. You are responsible for the design: what modules you have, the data structures they use, and how they interaction.

You will also have to strengthen your programming skills. A big part of that is making good use of of a variety of tools, such as `gdb`, `Valgrind`, and others (see Section 6. We will review these tools in recitations sessions early on in the semester. One important skill is debugging. You should first try to debug code yourself (using the above tools) but if you have problems, you use the office hours of not only the lead TA but also the other TAs to get help. Note that the TAs are not allowed to debug your code for you. Specifically, they will only look at your code for a limited time (up to 10 minutes; leaving them time to help other students) and they will not modify your code (they cannot touch the keyboard).

You will also have to develop your own test suite to test and help in debugging your code. While we will give you some tests for each checkpoint, these tests will only test some of the features of your implementation and they are only a subset of the tests we will use for grading. You are responsible for writing additional tests so you cover all the features and requirements listed in this handout.

3 Checkpoint 1: Parse and Identify Valid HTTP Requests

In this checkpoint, you will parse requests from a client like Opera, Firefox, or Chrome. Instead of responding to the requests, your Liso server will simply try to identify whether to request is a valid HEAD, GET, or POST request. If it is a valid HEAD, GET, or POST request, the server will 'echo' the request back to the client. Otherwise, it will return an HTTP response with 400 as the error code.

Your server MUST:

- Correctly identify whether an HTTP HEAD, GET, or POST request is correctly formed
- Handle multiple clients sending HTTP requests at the same time.
- Use `lex` and `yacc` to parse the requests.
- Use `select()` to accept incoming data and to handle a maximum of 1024 sessions in parallel. Handling pipelined requests is only required in CP2, so may assume that for each session, clients will only send a new request after they have received the response to their previous request from the server.
- Never crash for any error.

We will review `lex`, `yacc`, and `select` in the recitations.

3.1 Getting Started

1. Unpack the starter code and create a git repo (`tar -zxvf Project1_starter.tar.gz; cd 15-441-project-1; git init`).
2. Create a `select()`-based echo server handling multiple clients at once, building on the starter code.
3. Parse HTTP 1.1 requests and classify them as "good" or "bad" based on the provided RFC [2]. 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 the error code.
4. Test using our provided cp1_checker.py test script (read that script and understand it too.) Try to 'break' your server and make it crash – and then patch the bugs you find that makes it crash.
5. Finally, hand-in your submission by the deadline and include all needed files as outlined in §6.4.

3.2 Tips & FAQ

- To test 'Valid' requests, we will specifically be using requests as formatted by Opera, Firefox, Chrome, or Apache Bench. In industry folks test web services on a tens or even hundreds of combinations of browsers and environments... we'll stick to just these four for this project.
- The RFC is long. Reading an RFC is quite different from reading a news article, mystery novel, or even a technical paper. Appendix B.2 has some tips read and use and RFC efficiently.
- When reading the RFC, there are many tricky points in determining whether or not a request is 'Valid'. *It is not cheating to discuss what the RFC says.* Clarification about what the RFC requires is absolutely okay discussion!

- Note that Apache Bench expects a Content-Length: field on every reply – even error messages. Every valid request we send you will include a Content-Length field.
- If the request is validly formed but not a HEAD, GET, or POST you may either reply with a 400 or a 501 error code for this checkpoint (in CP2 you must reply with a 501).
- Can we assume that we will only receive one request per connection?
  Yes, we will not require multi-request support or pipelining until CP2.

3.3 Grading

The breakdown of grading for Checkpoint 1 is below.

Task	Weight	Subcriteria
Format	20%	*Assigned by human grader:* • Correct turnin – Makefile, compilation, properly tagged repo, TA's don't have to edit or search for files (10%) • Code style (5%) • Code commenting (5%)
Basics	15%	*Assigned in AutoLab:* • Accepts requests • Provides a response • Does not crash on receiving valid or invalid requests • Accepts/Replies to requests from multiple clients at the same time
Valid Request Handling	25%	*Assigned in AutoLab:* • Echoes to valid requests from Opera, Chrome, Firefox, or Apache Bench.
Invalid Request Handling	40%	*Assigned in AutoLab:* • Provides error code 400 to requests formatted improperly according to RFC 2616. • Does not cause problems or slow down concurrent clients just because one client breaks or hangs.

Checkpoint 1 is worth 15% of P1, and P1 is worth 15% of your final grade. Hence, in completing Checkpoint 1 with a perfect score you would earn 2.25% of the points available for the final grade.

4 Checkpoint 2

In this checkpoint, your web server will reply to client requests as a real HTTP web server for static content. You will be able to browse to your server using a real web browser!

Your server MUST:

- Respond to properly formatted HTTP HEAD and GET requests. You do not need to respond to POST requests yet (continue to echo reply to POST requests if the request is correctly formatted; send an error 400 if the request is malformed).
- Support five HTTP 1.1 error codes: 400, 404, 408, 501, and 505. 404 is for files not found; 408 is for connection timeouts; 501 is for unsupported methods; 505 is for bad version numbers. Everything else can be handled with a 400.
- Handle concurrent connections using select().
- Handle pipelined requests.

4.1 Getting Started

1. Begin with your repository for Checkpoint 2, using the result of Checkpoint 1 as a starting point.

2. You are highly encouraged to create a simplified logging module for your project that writes out formatted logs to the log file specified on the command-line; this will help you debug and trace requests that come to your system. *Nonetheless we will not require this in grading.* See *here* for some examples of how Apache handles logging.

3. Enhance your server to respond properly to any HTTP 1.1 request and implement persistent connections with HEAD and GET as defined in RFC 2616. At this point as we don't have CGI and so it doesn't make sense to support POST requests – continue to echo reply to correctly formatted POSTs.

4. Your server will need to handle lots of concurrent and pipelined requests: make sure you test with many simultaneous connections. Your server should respond to pipelined requests in the order that they were received, as specified in the RFC.

5. The server should also handle errors in a practical way. **It should never completely crash** (make it as robust as possible). In testing, try to crash your server by sending malformed and strange requests and fix your server to prevent these crashes.

6. Double check that your server sends the appropriate error messages to malformed requests.

7. Submission is the same as Checkpoint 1. Tag and upload your repo in a tarball to the corresponding lab on Autolab.

4.2 Tips and FAQ

1. Do we have to include a Last-Modified field in our responses?
   Yes

2. Do we have to support Chunking?
   No

3. Do we have to support "Conditional" GETs?
   No.

4. Can I use a hash table library written by another person?
   No, for this project implement your own if you really want it. You won't have to track every header, only the important ones for basic compliance.

5. Should we expect HTTP/1.1 requests to fit in one buffer?
   No, do not assume that they always fit inside one buffer. Be prepared to parse across buffer boundaries.

6. Can I assume that the request always has the Content-length field?
   Yes, assume requests to your server have Content-length if applicable. If it is missing, return a 400 response.

7. Since the server will serve static files, are we allowed to use http://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types
   Yes, You are allowed to use that or a simplified version of it.No need to support all well-known MIME-types, just the most common ones: text/html, text/css, image/png, image/jpeg, image/gif and maybe a few others up to your discretion.

8. For last-modified field, is there a C library that we can use to read file metadata such as this? Or is our web server supposed to handle that manually. (i.e. stamping each and every file in the www root with a last-modified stamp and manually updating that stamp every time a file is changed....).
   stat() is a system call to check for metadata on a file.

9. Are we allowed to reject requests with headers beyond a maximum size?
   Yes, you may reject any header line larger than 8192 bytes. Note, that this is different than a Content Length of greater than 8192. Additionally, you must find and parse the next request properly for pipelining purposes if you do reject the request.

10. Should we also handle requests made up with /n instead of /r/n?
    Some web servers do this and it is nice for telnet testing. However, Liso does not have this as a requirement – you do not have to do this.

4.3 Grading

The breakdown of grading for Checkpoint 2 is below.

Task	Weight	Subcriteria
Format	10%	*Assigned by human grader:* • Correct turnin – Makefile, compilation, properly tagged repo, TAs don't have to edit or search for files (5%) • Code style (2.5%) • Code commenting (2.5%)
Error Handling	10%	*Assigned in AutoLab:* • Responds to wrong version with 505 • Responds to unsupported method with 501 • Responds to other problems / invalid requests with 400 • Responds to time out with 408.
Request Handling	50%	*Assigned in AutoLab:* • Replies to correctly formatted HEAD requests. • Replies to correctly formatted GET requests. • Echos in reply to correctly formatted POST requests. • Accepts and responds to pipelined requests.
System Design	30%	*Assigned in AutoLab:* • Never crashes. • Uses select() (no multithreading). • Handles hundreds of concurrent connections when tested with Apache Bench.

Checkpoint 2 is worth 25% of P1, which is worth 15% of your final grade. Hence, in completing Checkpoint 2 with a perfect score you would earn 3.75% of the points available in the final grade.

5 Checkpoint 3

In this checkpoint, students enrolled in 15-641 will add support for CGI requests and for HTTPS, which uses encryption to keep connection content private from Internet eavesdroppers. In addition, you will 'daemonize' your server to allow it to run as a persistent background task, and you will double-check all of the requirements from Checkpoints 1 and 2 for a final grade.

Although this is completely optional, students enrolled in 15-441 may also try to implement CP3 if they wish to do so. Such courageous and slightly masochistic students will be rewarded with t-shirts, snacks and their own personal pride.

Your server MUST:

- Support the CGI standard as specified in RFC 3875.
  Note that a new error code will be added for that purpose: 500.
- Support HTTPS via TLS as specified in RFC 2818.
- Run as a daemonized, background process.
- **Remember: Continue to support regular HTTP requests as in CP2.**

5.1 Getting Started

We suggest you work on this checkpoint in two phases. In one phase, implement CGI. In another phase, add support for HTTPS. You can do these steps in either order – just don't try to do both at the same time! The remainder of this section explains the steps involved in completing CP3. A high-level overview of CGI and HTTPS will be presented in the web lecture in the beginning of the semester (before you get to CP3).

5.1.1 Implementing CGI

The Common Gateway Interface (CGI) allows a web browser to generate dynamic content for a client, e.g., content that is based on input the client provided. When the web server receives a request using a CGI URI, it will start a new process in which it starts the CGI program and provides it with the client's requests. CGI program then generates the response and returns it to the web server, which returns it to the client. We provide instructions for implementing CPI below, but we recommend that you 'skim' the CGI RFC [4] and the URI RFC [1] to get the big picture before diving in to your implementation.

1. Create support for the CGI variables listed in §A. Not all variables may be used in the ASCII Art sample, but we will have additional checks to make sure that you do set these variables in case other CGI scripts are used. Using a python CGI script that echos the environment variables may be useful for your debugging.
2. Implement your CGI module. Any URI starting with "/cgi/" will be handled by a single command-line specified executable via a CGI interface coded by you. We have also provided a CGI runner in C.
   (a) CGI URI's may accept GETs, POSTs, and HEADs; your job is not to decide this, just pass along information to the program being called
   (b) You need to pipe stdin, pipe stdout, fork(), setup environment variables per the CGI specification, and execve() the executable (it should be executable) Note: Watch the piped fd's in the parent process using your select() loop. Just add them to the appropriate select() sets and treat them like sockets, except you have to pipe them further to specific sockets.
   (c) Pass any message body (especially for POSTs) via stdin to the CGI executable
   (d) Receive any response over stdout until the process dies (monitor process status), or there is nothing more to read or a broken pipe is encountered
   (e) If the CGI program fails in any way, return a 500 response to the client, otherwise send all bytes from the stdout of the spawned process to the requesting client.
   (f) The CGI application will produce headers and message body as it sees fit, you do not need to modify or inspect these bytes at all.

5.1.2 Implementing HTTPS

HTTPS uses the Transport Layer Security (TLS) protocol to secure HTTP sessions. TLS is a session layer protocol so it sits between the HTTP protocol you are implementing and the TCP transport protocol. TLS ensure the *secrecy* and *integrity* of the data exchanged over the TCP connection, so third parties cannot read or modify the data. TLS also allows the client to *authenticate* web server, the client knows it is communicating with the right server. Authentication is based on a *certificate*, a data structure links the server's URL to its public key and is signed by a Certificate Authority (CA). To support HTTPS, your web server needs to use the OpenSSL library instead of the socket API, and the person responsible for the web server (that means you) needs to obtain a URL and a signed certificate.

Please follow the following instructions to implement HTTPS:

1. Create a DNS hostname for yourself with a free account at No-IP [13] (or use a domain name you already have...)
2. In order to test your HTTPS implementation, we need to monitor your (encrypted!) HTTPS traffic. To allow this, add the 15-441 Carnegie Mellon University Root CA to your browser (import certificate, usually somewhere in preferences)
   (a) Now we can man-in-the-middle your HTTPS :-) Being course staff has perks!
   (b) But just trust us till this part is over...or make your own **CA**.
   (c) Really though, this is the part of the course where you need to Reflect on Trusting Trust.
3. Obtain your own private key and public certificate from the 15-441 CMU CA. You will need this when you run your HTTPS-enabled web server.
4. Implement SSL support - we have provided you a sample C server in the Autolab Handout.
   (a) Use the OpenSSL library. [7]
   (b) Create a second server listening socket in addition to the first one. Use the passed in SSL port from the commandline arguments.
   (c) Add this socket to the select() loop just like your normal HTTP server socket.
   (d) Whenever you accept connections, wrap them with the SSL wrapping functions.
   (e) Use the special read() and write() SSL functions to read and write to these special connected clients
   (f) If you setup your browser, you may now verify that connections to your webserver use TLSv1.0; inspect the ciphers, message authentication hash scheme, and key exchange methods used by your server.
5. Implement daemonization - we have provided you a sample of daemonizing code.
6. Submission is the same as Checkpoint 1. Tag and upload your repo in a tarball to the corresponding lab on Autolab.

5.2 Grading

The breakdown of grading for Checkpoint 3 is below.

Task	Weight	Subcriteria
Format	10%	*Assigned by human grader:* • Correct turnin – Makefile, compilation, properly tagged repo, TA's don't have to edit or search for files (5%) • Code style (2.5%) • Code commenting (2.5%)
CGI	40%	*Assigned in AutoLab:* • Supports all CGI variables • Runs provided ASCII Art CGI script • Accepts/Replies to CGI requests from multiple clients at the same time over HTTP.
HTTPS	30%	*Assigned in AutoLab:* • Presents a certificate and initiates a TLS connection. • GET, HEAD, and POST requests all work through the HTTPS connection. • Multiple clients can connect at the same time over HTTPS.
Basic HTTP	20%	*Assigned in AutoLab:* • All criteria from CP2.

Checkpoint 3 is worth 60% of P1, which is worth 15% of your final grade. Hence, in completing Checkpoint 3 with a perfect score you would earn 9% of the points available in this class.

6 Implementation and Submission

In this section we talk about your code and requirements for development. Recall that in all checkpoints, part of your grade is based on meeting the formatting requirements. So, please read this section carefully!

6.1 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. You may download this code from the course website or from Autolab.

6.2 Coding and Compilation

Your server must be written in the C programming language. You may use code from the 15-213/15-513 course, so long as you cite the borrowed code in a reference. Nonetheless, we warn you that the 15-213/513 code will require substantial modification (e.g. it can cause your server to crash/abort in certain conditions; the code is blocking and can impede your ability to handle concurrent connections). You are not allowed to use any other third party socket classes or libraries, only the standard socket library and the provided library functions in the starter code

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 [11] (link with `-lefence`) and Valgrind [14]—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[9] for details. When we run `make` we should end up with the Liso web server binary `lisod`.

For each checkpoint we will provide tests for some of the features you have to implement. **However, we will limit the number of autolab runs for each checkpoing to twelve**. Passing these tests will allow you to make sure your implementation is on the right track. It also means that you have gained some of the points associated with the assignment. However, you will not have enough autolab runs that you can use it for debugging (which does not work anyway). You still need to write your own tests.

6.3 Work with git

All of your project files and submissions **must** be stored in a git repository.

You are supposed to create your git repo on your local machine or on a **private** shared repo hosted online as part of Checkpoint 1. **If you use github do not make your project code public or you will find cheaters copying your code and yourself in a very uncomfortable academic misconduct meeting.**

Every checkpoint will be a git tag in your 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. Be sure to use `git push --tags` to sync your work back to git server; the standard `git push` doesn't synchronize tags.

6.4 Hand-In

To submit your code, make a tarball file of you repo after you tag it. 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`).

Then login to autolab website, choose `\15-441: Computer Networks (f19)" -> \project1cp<N>"` and then upload your tarball. The grader should be finished in 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 – what does each file do? This file should also contain your name(s).

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

6.5 Reusing Code

The code you submit must be your own and we will use tools to compare your code with previous (both from this year and previous years), code available on the Internet, etc.

There are a few exceptions. You can use the following code:

- any starter code that we provide (of course!).
- any code that was provided to you in the 15-441/641 course.

A Required CGI Variables

We will test for the following CGI variables only.

1. CONTENT_LENGTH – taken directly from request
2. CONTENT_TYPE – taken directly from request
3. GATEWAY_INTERFACE – "CGI/1.1"
4. PATH_INFO – $< path >$ component of URI
5. QUERY_STRING – parsed from URI as everything after "?"
6. REMOTE_ADDR – taken when accept() call is made
7. REQUEST_METHOD – taken directly from request
8. REQUEST_URI – taken directly from request
9. SCRIPT_NAME – hard-coded/configured application name (virtual path)
10. SERVER_PORT – as configured from command line (HTTP or HTTPS port depending)
11. SERVER_PROTOCOL – "HTTP/1.1"
12. SERVER_SOFTWARE – "Liso/1.0"
13. HTTP_ACCEPT – taken directly from request
14. HTTP_REFERER – taken directly from request
15. HTTP_ACCEPT_ENCODING – taken directly from request
16. HTTP_ACCEPT_LANGUAGE – taken directly from request
17. HTTP_ACCEPT_CHARSET – taken directly from request
18. HTTP_HOST – taken directly from request
19. HTTP_COOKIE – taken directly from request
20. HTTP_USER_AGENT – taken directly from request
21. HTTP_CONNECTION – taken directly from request
22. HTTP_HOST – taken directly from request

B Tips

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

B.1 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.

B.2 How to read an RFC

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.

Next, take a second pass over the RFCs, focusing on the sections that that describe functionality you need to implement. 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.

Before you start coding, you then go back and read with an eye toward implementation. Mark the parts which contain details that you will need to write your server code. You may want to add bookmarks to the sections that you will need to reference during the implementation. 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.)?

B.3 Testing

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 [10]. Following this guiding principle of Internet design will help ensure your server works with many different and unexpected client behaviors.

Turn warnings into errors. You may want to consider turning warnings into errors to avoid bad programming style. Do this by passing `-Werror` to `gcc` during compilation.

C Resources

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

- Beej's Guide [8]
- Computer Systems: A Programmer's Perspective (CS 15-213 text book)[15]
- BSD Sockets: A Quick And Dirty Primer[16]
- An Introductory 4.4 BSD Interprocess Communication Tutorial[17]
- Unix Socket FAQ[18]
- Sockets section of the GNU C Library manual[19]
- man pages
  - Installed locally (e.g. man socket)
  - Available online: the Single Unix Specification[20]

References

[1] RFC 2396: http://www.ietf.org/rfc/rfc2396.txt

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

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

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

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

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

[7] Open SSL: https://www.openssl.org/docs/manmaster/man3/

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

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

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

[11] ElectricFence: http://perens.com/FreeSoftware/ElectricFence/

[12] Common Log Format: https://httpd.apache.org/docs/1.3/logs.html#common

[13] No-IP: https://www.noip.com/free

[14] Valgrind: http://valgrind.org/

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

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

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

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

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

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

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

[22] https://en.m.wikipedia.org/wiki/HTTP_pipelining