15-441/641: Networking and the Internet
Project 1: A Web Server Called Liso

TAs: Pengyun Zhao (pengyunz@andrew.cmu.edu)

Last update: July 7, 2021

1 Introduction

In this class you will 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 Transfer Protocol (HTTP) 1.1 —RFC 2616 [3]. In Checkpoint 3 (required for 641, optional for 441), your web server will also implement Hypertext Transfer Protocol Secure (HTTPS) via Transport Layer Security (TLS) as described in RFC 2818 [4] and the Common Gateway Interface (CGI) as described in RFC 3875 [5]. Reading an RFC is quite different from reading a news article, mystery novel, or even a technical paper. Appendix B.2 has some tips on how to read an RFC efficiently.

RFCs are one of many ways that 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 will be compatible with these standards, you will, by the end of this project, be able to browse the content on your web server using a 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 checkpoints:

CP	**Goal**	**% P1 Grade**		**Deadline**	
		441	641	441	641
1	Determine whether an HTTP 1.1 request is valid or invalid	25%	15%	5 PM on Sep 16, 2020	5 PM on Sep 9, 2020
2	Respond correctly to HTTP 1.1 HEAD and GET requests	75%	50%	5 PM on Oct 7, 2020	5 PM on Sep 30, 2020
3	Support HTTPS and CGI, run as a daemon, and complete all previous tasks	-	35%	-	5 PM on Oct 7, 2020

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. For example, in CP3, 20% of the points are allocated to basic HTTP functionality, which you should have implemented in CP2. This means that if you fix a bug you had in an earlier checkpoint, it may help your score in a later checkpoint.

Of course, if you break a feature that was originally implemented correctly, then you may 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 in CP3. Your code must be implemented in C. If your server does not meet these requirements, then you will receive a 0 for the project.

2 The Liso Server

In this section, we give an overview of the complete requirements for the Liso web 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 provide on GitHub: `https://github.com/computer-networks/liso-starter-code`

2.1 Supporting HTTP 1.1

You will find in RFC 2616 [3] that there are many HTTP methods: commands that a client sends to a server. Liso will support three methods only:

- **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), then 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 provide you is not robust to malformed requests (e.g, it cannot handle requests which do not have proper [CR][LF] line endings), so you will have to extend it to do so.

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" [23]. 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 your server can support 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 CP3.

HTTPS Support: Your server will use the OpenSSL library for HTTPS support. Specially, you will wrap communication calls 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. We have provided sample CGI scripts and sample code of how to interact with CGI programs in the handout for you to test your CGI implementation.

More details on HTTPS and CGI are in §5.

2.4 Command Line Arguments

Liso will always have 8 command line arguments. You may not modify Liso to, e.g., take a different number of arguments or reorder the arguments. If you do not need one of the arguments in the earlier checkpoints, then 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 interact.

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` [14], `Valgrind` [15], 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 your code yourself (using the above tools), but if you have problems, you can use the TA office hours 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 your keyboard).

You will also need 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. Additionally, you might want to use `Apache Bench` [24] and `Siege` [25] to test your code. We will be utilizing these two tools in some of the later checkpoints. These two command line tools will be automatically installed if you initiate the docker container as described in the *README.md* file in the starter code. Use **$ ab -h** and **$ siege -h** to show the help pages of the two tools.

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 the request is a valid HEAD, GET, or POST request. If it is a valid HEAD, GET, or POST request, then 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 you 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. Clone the starter code from the **GitHub repo** and change the remote repo to your private repo (we assume that you are using a private Git repository for version control):
```
$ git clone https://github.com/computer-networks/liso-starter-code
$ cd liso-starter-code
$ cd 15-441-project-1/
$ git init
$ git remote add origin YOUR_PRIVATE_REPO
```
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 [3]. 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 make it crash.
5. Finally, turn in your submission by the deadline and include all needed files, as outlined in §6.5.

3.2 Tips & FAQ

- To test valid requests, we will specifically be using requests formatted by Opera, Firefox, Chrome, or Apache Bench. In industry, folks test web services on tens or even hundreds of combinations of browsers and environments...we will 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 on how to read an 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 to discuss!
- 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, then 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 HTTP pipelining until CP2.

3.3 Grading

The breakdown of grading for CP1 is below.

Task	Weight	Subcriteria
Format	20%	*Assigned by human grader:* • Use **select()** (10%) • Code style (10%)
Basics	20%	*Assigned in Gradescope:* • Proper make (1%) • Responds to single client (4%) • Responds to multiple clients (5%) • Accepts/replies to requests from concurrent clients at the same time (10%)
Valid Request Handling	25%	*Assigned in Gradescope:* • Echoes to valid requests from Opera, Chrome, Firefox, and Apache Bench. (25%)
Invalid Request Handling	35%	*Assigned in Gradescope:* • Provides error code 400 to requests formatted improperly according to RFC 2616. (25%) • Hidden tests (10%)

4 Checkpoint 2

In this checkpoint, you will extend your web server to reply to client requests for static content. When completed, 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. Use your solution for CP1 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 do not have CGI support and so it does not 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 are 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. Test your server using `Apache Bench`. Make sure to set it to the correct version.
8. Submission is the same as in CP1. Tag and upload your repo in a tarball to the corresponding lab on Gradescope.

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 the "Content-Length" field, if applicable. If it is missing for a POST request, return a 400 response. For GET and HEAD request if the field is missing you can assume the content length is 0.

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 the "Last-Modified" field, is there a C library that we can use to read file metadata?
   `stat()` is a system call to check a file's metadata.

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, if you do reject a request, you must find and parse the next request properly for pipelining purposes.

10. Should we also handle requests with /n instead of /r/n?
    No. 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 CP2 is below.

Task	Weight	Subcriteria
Format	10%	*Assigned by human grader:* • Use **select()** (5%) • Code style (5%)
Error Handling	24%	*Assigned in Gradescope:* • Responds to wrong version with 505 (2%) • Responds to unsupported method with 501 (2%) • Responds to other problems or invalid requests with 400 (2%) • Responds to timeouts with 408 (4%) • Hidden tests (14%)
Request Handling	50%	*Assigned in Gradescope:* • Replies to correctly formatted HEAD requests (12%) • Replies to correctly formatted GET requests (12%) • Echos in reply to correctly formatted POST requests (12%) • Accepts and responds to pipelined requests (14%)
Efficiency	16%	*Assigned in Gradescope:* • Handles hundreds of concurrent connections when tested with `Apache Bench` and `Siege` (10%) • Hidden tests (6%)

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 CP1 and CP2 for a final grade. **In order for us to grade your code, you must implement daemonization.**

Although this is completely optional, students enrolled in 15-441 may also try to implement CP3 if they wish to do so. Such courage will be rewarded with t-shirts and glory.

Your server MUST:

- Support the CGI standard as specified in RFC 3875 [5]. Note that a new error code will be added for that purpose: 500.
- Support HTTPS via TLS as specified in RFC 2818 [4].
- Run as a daemonized, background process.
- **Remember: Continue to support regular HTTP requests as in CP2.**

5.1 Getting Started

We suggest that 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 do not 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 recitations in the beginning of the semester (before you get to CP3).

5.1.1 Implementing Deamonization

Daemonization is the ability for your server to run as a separate process in the background. You will also need to handle two POSIX signals, SIGHUP and SIGTERM. You should be familiar with them if you took 15-213/513.

For SIGHUP, you need to rehash the server. This is used in practice to ensure that servers experience minimal downtime. An administrator can update the certificate for the server, then send a SIGHUP signal that will cause the server to update and use the new certificate.

For SIGTERM, you need to gracefuly shut down the connections, any memory usage, and ultimately the entire server.

5.1.2 Implementing CGI

The Common Gateway Interface (CGI) [5] 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. The CGI program then generates the response and returns it to the web server, which returns it to the client. We provide instructions for implementing CGI below, but we recommend that you skim the CGI RFC [5] and the URI RFC [2] 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 CGI test script we provided, 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 example code of a CGI runner in the handout for CP3.

   (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 data as it sees fit, you do not need to modify or inspect these bytes at all.

5.1.3 Implementing HTTPS

HTTPS [4] 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 ensures 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* the web server, so the client knows that it is communicating with the right server. Authentication is based on a *certificate*, a data structure that 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 [8] 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. In this project, we have simplified the work for you in that it is okay to run the web server locally on localhost (127.0.0.1) with a self-signed certificate.

Please follow these instructions to implement HTTPS:

1. In order to test your HTTPS implementation, you need to get a certificate. We have already provided a generation script (**gen_cert.sh**) in the handout for CP3 for you to generate a self-signed certificate. By simply running **$ bash gen_cert.sh**, you will get two files in the current directory, *localhost.key* and *localhost.crt*. *localhost.crt* is the certificate that can be used for authentication.

2. Implement SSL support - we have provided you a sample C server in the handout.

   (a) Use the OpenSSL library. [8]

   (b) Create a second server listening socket in addition to the first one. Use the passed-in SSL port from the command line 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 TLS; inspect the ciphers, message authentication hash scheme, and key exchange methods used by your server.

3. Implement daemonization - we have provided you a sample of daemonizing code.

4. Submission is the same as in CP1. Tag and upload your repo in a tarball to the corresponding lab on Gradescope.

5.2 Grading

The breakdown of grading for Checkpoint 3 is below.

Task	Weight	Subcriteria
Format	10%	*Assigned by human grader:* • Use **select()** (5%) • Code style (5%)
Daemonization	6%	*Assigned in Gradescope:* • Correctly deamonizes and handles SIGHUP & SIGTERM (6%)
CGI	12%	*Assigned in Gradescope:* • Supports all CGI variables and runs provided CGI script (12%)
HTTPS	30%	*Assigned in Gradescope:* • Responds to valid TLS requests (12%) • Hidden tests (18%)
Basic HTTP	42%	*Assigned in Gradescope:* • All criteria from CP2 (42%)

6 Implementation and Submission

In this section, we discuss your code and the 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 command line arguments (and sanity check them) and daemonize a process. You may download this code from GitHub: `https://github.com/computer-networks/liso-starter-code`

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 code from 15-213/513 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 that your code compiles and runs correctly. 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 aide with debugging. Other tools that we suggest are ElectricFence [11] (link with `-lefence`) and Valgrind [15] (use this with full leak checking to ensure that 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 binary `lisod`.

6.3 Gradescope and Docker

In this project, we will use Gradescope for autograding. Gradescope uses docker containers to run all of the test scripts. In order to maintain consistency between your development environment and the testing environment, we **strongly recommend** that you use the docker container as your development environment. Basic instructions on how to set up the docker container on your machine are provided in the *README.md* file in the starter code. More information about docker can be found in its official documentation (`https://docs.docker.com/`).

For each checkpoint, we will provide tests for some of the features you have to implement. **However, we will limit the number of Gradescope runs for each checkpoint to 3 per day**. So please **start early**! One more day means 3 more submissions! Passing these tests will allow you to make sure that your implementation is on the right track. Passing these tests also means that you have gained some of the points associated with the assignment. However, 3 Gradescope runs per day is insufficient for you to use Gradescope for iterative debugging. Therefore, you still need to write your own tests.

We also have hidden test cases for each checkpoint of each project on Gradescope. These tests will be run everytime you hand in to Gradescope, but the output and the score of those tests will not be visible until after the due date of the checkpoints. The meaning of those tests is to motivate you to make continuous improvements on your projects to make them more resilient and efficient.

6.4 Working with Git

All of your project files and submissions **must** be stored in a Git repository (repo).

As part of CP1, you are supposed to create your git repo on your local machine or use a **private** repo hosted online. **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 of your checkpoint solution>]
```

with the appropriate checkpoint number and custom message filled in. (Put whatever you like for the message — git will not let you omit it.) The optional commit hash can be used to specify a particular commit for the tag; if you omit it, then the HEAD commit is used. Be sure to use `git push --tags` to sync your work back to the Git server; the standard `git push` doesn't synchronize tags.

6.5 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 **handin.tar** (Please use exactly this name, otherwise the autograder will not work). Untarring this file should give us a directory named **15-441-project-1** which should contain the Git repo with your code. You will submit this tarball using Gradescope: `https://www.gradescope.com/` .

Log in to the Gradescope website, choose "15-441: Computer Networks and the Internet" `->` "Project 1 CP*N*", and then upload your tarball. The autograder should finish in less than 20 minutes. When it is done, your score will be shown. If there are hidden tests (as in CP2 and CP3), then the total score will include those hidden tests. 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 following files at a minimum:

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

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

6.6 Reusing Code

The code you submit must be your own and we will use tools to compare your code with your peers' code, submissions from previous semesters, code available on the Internet, etc.

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

- Any starter code that we provide in this course
- Any code that was provided to you in the 15-441/641 textbook: `https://book.systemsapproach.org/`
- Any code that is given to you in 15-213/513 or the textbook for 15-213/513

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 – hardcoded/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

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 90-10 rule: the first 90% of the job takes 10% of the time; the remaining 10% of the job takes the other 90% of the time. Starting early gives you time to ask questions. For clarifications on this assignment, post to Piazza and watch for updates to this document. 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 do not expect you to read every page of the RFCs, 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 their 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 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, go back and reread the RFCs with an eye towards 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.) that 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, replay scripts, `Apache Bench` [24], `Siege` [25], etc.

In order to test your server with a web browser on your machine, do not forget to add a port mapping argument when you launch the container. In order to test your server with a browser on your machine, you need to set up port mapping for the docker container. Simply add a command line argument, such as **-p 8888:15441**, to the command of initiating a container (i.e., **$ docker run ...**). Then a mapping from 127.0.0.1:15441 (in container) to 127.0.0.1:8888 (on your machine) is established. Then you can use the URL 127.0.0.1:8888 in your browser to test your server, which is bound to port 15441 in the container.

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

Turn warnings into errors. You may want to consider turning compiler 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 [9]
- Computer Systems: A Programmer's Perspective (CS 15-213 text book)[16]
- BSD Sockets: A Quick And Dirty Primer[17]
- An Introductory 4.4 BSD Interprocess Communication Tutorial[18]
- Unix Socket FAQ[19]
- Sockets section of the GNU C Library manual[20]
- man pages
  - Installed locally (e.g. **$ man socket**)
  - Available online: The Single Unix Specification[21]

References

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

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

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

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

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

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

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

[8] OpenSSL: `https://www.openssl.org/docs/manmaster/man3/`

[9] Beej's Guide: `https://beej.us/guide/bgnet/html//index.html`

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

[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] GDB: `https://www.gnu.org/software/gdb/`

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

[16] CSAPP: `http://csapp.cs.cmu.edu`

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

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

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

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

[21] `http://www.opengroup.org/onlinepubs/007908799/`

[22] `http://groups.google.com`

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

[24] Apache Bench: `https://linux.die.net/man/1/ab`

[25] Siege: `https://linux.die.net/man/1/siege`