

Assignment 2: HTTP Server

CSE 130: Principles of Computer Systems Design

Due: October 29, 2024 at 11:30 PM

Goals This assignment will provide you with experience building a system that uses client-server/strong modularity. Our learning objectives are: (1) practice implementing a client-server system, (2) practice the advantages of powerful abstractions, (3) practice implementing a large system that solves a large problem, (4) review string parsing, and (5) review memory management.

This assignment is more complex than Assignments 0 and 1. This doc is long out of necessity: we have a lot of things to explain! We encourage you first to read and annotate this document early (in the first few days that the assignment is released), then design your server with a “box-and-arrows” diagram (within the first week that the assignment is released), and only then start writing code.

Assignment Details

You will be building an HTTP server for this assignment. Your server should execute “forever” without crashing (i.e., it should run until a user types **CTRL-C** on the terminal). Your server will create, listen, and accept client connections that arrive on a *port*. Your server’s task is to process the bytes clients send using the HTTP protocol. A key task in this assignment is building a resilient server for malformed (or even malicious) clients: *no matter what a client sends your server, your server should not crash!*

In this section, we explain (1) how to run your server, (2) what your server’s execution should perform, and (3) provide a few examples to help you understand.

Running your server

Your server should take a single, required command line argument, an `int`, named *port*. Note: you may need to run your server with `sudo` if using a port number between 1 and 1023. In other words, your server should be started by specifying the following command from the `asgn2` directory of your repository:

```
./httpserver <port>
```

Your Server’s Execution

This section provides an overview of the tasks that your server should perform. First, your server should create structures needed to listen to incoming connections. Then, it should repeatedly accept a new client connection, process it, and close that connection. Note that this should occur one connection at a time (i.e., it should accept and process its first connection before it begins processing the second connection). We outline the requirements for each of these steps below:

Listening for Connections

First, your server should create a *socket*, bind the socket to a `port`, and make it listen for incoming connections. We will provide you with a struct definition, `Listen_Socket`, for this socket. Additionally, we will provide a function, with `int listener_init(*Listen_Socket, int port)`, that initializes a `Listen_Struct` to listen to a given `port`¹. If the `port` passed to `httpserver` is invalid (not an integer between 1 and 65535) or if `httpserver` cannot bind to the provided `port`, then your `httpserver` should produce the message “Invalid Port\n” to `stderr` and exit with a return code of 1. Check out the return value of `listen_init`; it might help you with this task...

¹Note: We encourage you to use our library functions, but you don’t *have* to do so if you’d rather build everything yourself.

Accepting Connections

Your server should repeatedly accept connections made by clients to the `port`. Linux represents a connection with an external entity (in this case, a client) using the `socket` abstraction. We will provide you with a library function, `int listener_accept(*Listener_Socket sock)`, that blocks execution until a new incoming connection is made to the socket, `sock`, and then returns a new socket for the new connection.

After accepting a connection, your server should process any valid HTTP commands that the client sends on that connection. To do this, your server will need to read bytes sent by the client and send bytes to the client. The `socket`, created when your server accepts the new connection, makes this task possible. You should think of a socket as equivalent to a file descriptor: It is an integer. Your server can read bytes in the order that they were written from the other side of the socket (i.e., the client) by calling `read` with the socket as the `fd` argument. Your server can write bytes in the order that they were written from the other side of the socket (i.e., the server) by calling `write` with the socket as the `fd` argument.

Processing Connections

Your server should process a simplified subset of the HTTP Protocol². In particular, your server should support two types of HTTP operations: `GET` and `PUT`. Below, we describe the HTTP request format that your server should be able to handle, the HTTP response format that your server should send to clients as a response, and then describe the processing for `GET` and `PUT` commands.

Requests

`GET` and `PUT` requests share the same request format (i.e., a valid `GET` request and a valid `PUT` request contain the same fields). Below we show the format, identifying those that are required and those that are Optional. Then, we describe each field in detail. Note, that each field is separated, without any additional whitespace or characters, by the sequence `\r\n`. This sequence is the sequence of the single character ‘`\r`’ followed by the character ‘`\n`’, **NOT** the sequence of four characters ‘`\`’, followed by ‘`r`’, followed by ‘`\`’, followed by ‘`n`’:

<i>request</i>	::=	<i>request-line</i>	(required)
		<i>header-field-list_{opt}</i>	(optional, repeated)
		<i>empty-line</i>	(required)
		<i>message-body</i>	(optional)

You may assume a limit on the size of the first part of a valid request. The characters of the request, starting with the *request-line* and extending up to and including the *empty-line*, will not exceed 2048 characters. That is, the part of the request before any *message-body_{opt}* will not exceed 2048 characters.

Below, we describe each of the fields of a request in detail.

In the grammar rules, **fixed-width font** indicates a regular expression that can be used with `regcomp()`. Within these regular expressions, `\r` and `\n` represent single characters (as they do in C strings, for example `\n` means the newline character). Square brackets are used like `[a-z0-9]` and `[abc123]` to represent a single character within the indicated set and/or range(s). `{1,8}` indicates limited repetition of the preceeding items within the indicated range. Most other letters are literal, except that a period has special meaning in a regular expression, and so to get a period `.` instead use the `\.` character pair.

²The curious reader can find the whole HTTP Protocol at Request For Comments: 2616 (abbreviated RFC 2616). We’ve made several simplifications, though, so follow our instructions rather than the RFC.

- *request-line*. Every valid request includes exactly one *request-line* with the following format, where each field (*method*, *uri*, and *version*) is a case-sensitive sequence of characters.

```

request-line ::= method _ uri _ version \r\n
method      ::= [a-zA-Z]{1,8}
_           ::= single space character, ASCII 32
uri         ::= /[a-zA-Z0-9.-]{1,63}
version     ::= HTTP/[0-9]\.[0-9]          (\. represents a single period)

```

We limit valid requests in the following ways:

- A valid *method* contains at most eight (8) characters from the character range [a-zA-Z]. Your server only needs to implement (i.e., perform the semantics) of GET and PUT. (The distinction between “treats as valid” and “implements” will be clear after the Responses section below).
- A valid *uri* starts with the character ‘/’, includes at least 2 characters and at most 64 characters (including the ‘/’), and except for the leading ‘/’, only includes characters from the character set [a-zA-Z0-9.-] (this character set includes 64 total valid characters). Each *uri* matches the file within the directory in which the `httpserver` is running. For example, the *uri* /foo.txt matches the file foo.txt from the folder in which the `httpserver` is running. **Note: your server does not need to handle uris that include directories.** Your server should only perform the semantics of GET and PUT on requests that include a valid *uri*.
- A valid *version* has the format HTTP/#.#, where each # is a single-digit number. Your `httpserver` should only implement version 1.1, so it should only perform the semantics of GET and PUT requests that include a *version* equal to HTTP/1.1.
- *header-field-list_{opt}*. Valid requests include a list of zero (0) or more *header-fields* after the *request-line*.

```

header-field-listopt ::= zero or more header-field items
header-field      ::= key : _ value \r\n
key               ::= [a-zA-Z0-9.-]{1,128}
value             ::= [ -~]{1,128}          (any printable ASCII characters)

```

- The *key* ends with the first instance of a ‘:’ character. A valid *key* will be at least 1 character, at most 128 characters, and only contain characters from the character set [a-zA-Z0-9.-].
- A valid request’s *value* will contain at most 128 characters and only contain characters from the set of printable ASCII characters (i.e., a valid value will not contain any ASCII “Device Control” characters nor any other binary data).
- Valid requests end each *header-field* with the sequence \r\n. A list of *header-fields* will be followed by *empty-line*. (Essentially, regardless of how many *header-fields* a request contains, the list will terminate with the sequence \r\n\r\n.)
- *empty-line*. A list of *header-fields* will be followed by *empty-line*.

```
empty-line ::= \r\n
```

- So the \r\n that is at end of the final *header-field* will be followed by the \r\n of the *empty-line*, creating a final sequence before any *message-body*: \r\n\r\n. Even if the *header-fields* are omitted, the sequence still will end with \r\n\r\n because the *request-line* that would be immediately before the *empty-line* ends with \r\n.
- *message-body*. Valid PUT requests must include a *message-body*; valid GET requests will not include a *message-body* (or we can say that it has a *message-body* of zero bytes). Valid requests that include a *message-body* will also include a header, with a *key* of **Content-Length**, whose *value* will indicate the number of bytes in the *message-body*.

```
message-body ::= zero or more bytes of any value
```

<i>status-code</i>	<i>status-phrase</i>	<i>message-body</i>	Usage
200	OK	OK\n	When a method is successful
201	Created	Created\n	When a URI's file is created
400	Bad Request	Bad Request\n	When a request is ill-formatted
403	Forbidden	Forbidden\n	When a URI's file is not accessible
404	Not Found	Not Found\n	When the URI's file does not exist
500	Internal Server Error	Internal Server Error\n	When an unexpected issue prevents processing
501	Not Implemented	Not Implemented\n	When a request includes an unimplemented method
505	Version Not Supported	Version Not Supported\n	When a request includes an unsupported version

Table 1: List of HTTP status that your `httpserver` should support.

As a summary, here is an example of a valid PUT request to the URI, `foo.txt`:

```
PUT /foo.txt HTTP/1.1\r\nContent-Length: 12\r\n\r\nHello world!
```

Responses

Your `httpserver` must produce a response for each request, regardless of whether the request is valid or not. Your response must follow the grammar:

```
response ::= status-line
           header-field-list
           empty-line
           message-body
```

- *status-line*. The *status-line* indicates the type of response to the request. It consists of three fields.

```
status-line ::= http-version _ status-code _ status-phrase \r\n
http-version ::= HTTP/1.1
_ ::= single space character, ASCII 32
status-code ::= see Table 1
status-phrase ::= see Table 1
```

`httpserver` must always produce the *http-version* string `HTTP/1.1` regardless of the *http-version* provided in the request. A response with a *status-code* in the 200s indicates a successful response, in the 400s indicates an erroneous response, and in the 500s indicates an internal server error. Table 1 lists the *status-codes* that `httpserver` needs to produce, and their associated *status-phrases* that your server should produce in the response's status line, the message body that your server should produce as a part of the response, and when you should use each code.

- *header-field-list*. The *status-line* should be followed by one or more *header-fields*. A response's *header-field* has the same format as the request header fields:

```
header-field-list ::= one or more header-field items
header-field ::= key : _ value \r\n
key ::= [a-zA-Z0-9.-]{1,128}
value ::= [ -~]{1,128} (any printable ASCII characters)
```

The *key* of the response should be `Content-Length`, and the *value* should be equal to the size of the *message-body*. Your server should end the *header-field* with the sequence `\r\n`. A list of *header-fields* will terminate with an *empty-line*. (Essentially, the list of *header-fields* ends with `\r\n\r\n`).

- *empty-line*. The *header-field* will be followed by *empty-line*.

empty-line ::= \r\n

- The \r\n that is at end of *header-field* will be followed by the \r\n of *empty-line*, creating a final sequence before the *message-body*: \r\n\r\n.
- *message-body*. **httpserver** must produce a *message-body* with each response, whose size, in bytes, is equal to the value identified in the response's **Content-Length** header. Your server should produce the *message-body* indicated in Table 1 for each request, *except for valid GET requests*. We describe the correct *message-body* for valid **GET** requests in the section on **Methods**.

message-body ::= zero or more bytes of any value

With a **GET** request, it is possible that the requested file is empty, and in that case, a zero-length *message-body* will be send (or equivalently the *message-body* is omitted).

Methods

You must implement two HTTP methods, **GET** and **PUT**.

- **GET**: A **GET** request indicates that the client would like to receive the content of the file identified by the **URI**. If a request is valid and specifies a **URI** that is resident in the directory in which **httpserver** is executing, then **httpserver** should produce a response that...
 1. has a *status-code* of 200
 2. has a *message-body* that includes the current state of the file pointed to by *uri*, and
 3. has a **Content-Length** that indicates the number of bytes in the file.

For all other requests (include those that are valid but where the *uri* indicates a non-existent file), your server should produce a *status-code*, *message-body*, and **Content-Length** based upon Table 1.

- **PUT**: A **PUT** request indicates that the client would like to update or replace the content of the file identified by the *uri*.

If a valid **PUT** request's **URI** points to a **file that does not yet exist**, **httpserver** should...

1. create the file
2. set the file's contents equal to the *message-body* in the request
3. produce a response with a *status-code*, *message-body*, and **Content-length** for the *status-code* 201 based upon Table 1.

If a valid **PUT** request's **URI** points to a **file that already exists**, **httpserver** should...

1. replace the file's contents with the *message-body* in the request
2. produce a response with a *status-code*, *message-body*, and **Content-length** for the *status-code* 200 based upon Table 1.

For all other requests, your server should produce a *status-code*, *message-body*, and **Content-Length** based upon Table 1.

Closing Connections

After your server finishes processing a request for a connection, your **httpserver** should close the connection. You can simply call **close** on a socket like you would any other file descriptor. Note: each connection will contain at most one valid request; if it contains extra bytes after a valid request, your server should ignore those bytes (no need to send an additional response!). Also: before closing each connection, your server should read all of the bytes that were sent by the client, regardless of whether the client sent a valid request in their connection.

Notes

In addition to supporting the methods listed above, your project should meet the following functionality and limitations:

- `httpserver` should produce responses with the appropriate status code (see **Usage** in Table 1).
- `httpserver` should not have any memory leaks.
- `httpserver` will need to handle connections that do not send data (this is very important for our testing scripts)!
- `httpserver` should not leak any file descriptors.
- `httpserver` must be reasonably space efficient: it should use less than 10 MB of memory regardless of input.
- `httpserver` must be reasonably time efficient.
- `httpserver` should never crash (e.g., it should never `segfault`).
- `httpserver` must be written using the ‘C’ programming language (*not C++!*).
- `httpserver` cannot use the following functions from the ‘C’ `stdio.h` library: `fwrite`, `fread`, variants of `put` (i.e., `fputc`, `putc`, `putc_unlocked`, `putchar`, `putchar_unlocked`, and `putw`), and `get` (i.e., `fgetc`, `getc`, `getc_unlocked`, `getchar`, `getchar_unlocked`, `getline`, and `getw`).
- `httpserver` cannot use functions, like `system(3)`, that execute external programs.

Examples

In this section, we describe a few example requests and the correct responses for those requests. For each example, assume that your `httpserver` is started in a directory that contains the file, `foo.txt`, with content, “Hello World, I am foo”, and a file, `bar.txt` containing the content, “Hello World, I am bar”.

1. **Ex. 1** The client makes a `GET` request by sending this string (remember that `\n` refers to a single newline character and **not** the two separate characters `'\'` and `'n'`).

```
GET /foo.txt HTTP/1.1\r\n\r\n
```

The server does not change any files. It responds to the client by sending the client the following response:

```
HTTP/1.1 200 OK\r\nContent-Length: 21\r\n\r\nHello World, I am foo
```

2. **Ex. 2** The client makes a `PUT` by sending:

```
PUT /foo.txt HTTP/1.1\r\nContent-Length: 21\r\n\r\nHello foo, I am World
```

The server replaces the contents of `foo.txt` with “Hello foo, I am World” and responds by sending the client the following content

```
HTTP/1.1 200 OK\r\nContent-Length: 3\r\n\r\nOK\r\n
```

3. **Ex. 3** The client makes a `PUT` request by sending:

```
PUT /new.txt HTTP/1.1\r\nContent-Length: 14\r\n\r\nHello\nI am new
```

The server creates a new file in its directory, named `new.txt`, with the contents “Hello\nI am new”. It then responds to the client with the following:

```
HTTP/1.1 201 Created\r\nContent-Length: 8\r\n\r\nCreated\r\n
```

4. **Ex. 4** The client makes a GET request by sending:

```
GET /not.txt HTTP/1.1\r\n\r\n
```

Since `not.txt` does not exist in the server's directory, the server responds with the following:

```
HTTP/1.1 404 Not Found\r\nContent-Length: 10\r\n\r\nNot Found\r\n
```

5. **Ex. 5** The client makes an invalid request by sending:

```
GET /foo.txt HTTP/1.10\r\nhello*world: value\r\n\r\n
```

This request is invalid because (1) `HTTP/1.10` is an invalid `Version` since 10 is not a single digit number; AND (2) The `header-field` is invalid since the key `hello*world` contains the `*` character. So, the server will respond by sending the client the following content:

```
HTTP/1.1 400 Bad Request\r\nContent-Length: 12\r\n\r\nBad Request\r\n
```

Rubric

We will use the following rubric for this assignment:

Category	Point Value
Makefile	10
Clang-Format	5
Files	5
Functionality	80
Total	100

Makefile Your repository includes a Makefile with the rules `all` and `httpserver`, which produce the `httpserver` binary, and the rule `clean`, which removes all `.o` and binary files. Additionally, your Makefile should use clang (i.e., it should set `CC=clang`), and should use the `-Wall`, `-Wextra`, `-Werror`, and `-pedantic` flags (i.e., it should set `CFLAGS=-Wall -Wextra -Werror -pedantic`).

Clang-Format All `.c` and `.h` files in your repository are formatted in accordance with the `.clang-format` file included in your repository.

Files The following files are included in your repository: `httpserver.c`, `Makefile`, and `README.md`. Your repository should not include binary files nor any object files (i.e., `.o` files), except for the file named `asgn2_helper_funcs.a` (see Resources below). To make it easier for you to maintain tests, you can also include binary files in any directory whose name starts with the phrase `test`.

Functionality Your `httpserver` program performs the functionality described in Assignment Details.

How to submit

Submit a 40-character commit ID hash on the Canvas assignment to identify which commit you want us to grade. We will grade the last hash that you submit to the assignment on Canvas and will use the timestamp of your last upload to determine grace days. For example, if you post a commit hash 36 hours after the deadline, we will subtract 2 grace days from your total.

Resources

Here are some resources to help you:

Testing

We provided you with two resources to test your own code:

1. An autograder, which is run each time you push code to GitLab, will show you the points that you will receive for your Makefile, Clang-Format, and Files.
2. A set of test scripts in the resources repository to check your functionality. You can use the tests to see if your functionality is correct by running them on your Ubuntu 22.04 virtual machine. We provided you with a subset of the tests that we will run, but, I bet you can figure the other ones out by adapting what we have given you :-)
3. A set of helper functions in the resources repository that will help you perform basic tasks. Tables 2, 3, and 4 contain descriptions of these functions. We released two things, `asgn2_helper_funcs.h`, a header file that declares each of the functions, and `asgn2_helper_funcs.a`, a Linux archive that contains each of the function definitions in a binary format. You can treat `asgn2_helper_funcs.a` as if it were a `.o` file in your makefile. For example, to build an executable, `httpserver`, using the files `httpserver.o` and `asgn2_helper_funcs.a`, you might execute:

```
clang -o httpserver httpserver.c asgn2_helper_funcs.a
```

Hints

Here are some hints to help you get going:

- You will likely need to look up how some system calls (e.g., `read`) and library functions (e.g., `warn`) work. You can always Google them, but you might also find the man pages useful (e.g., try typing `man 2 read` on a terminal).
- You will have to do a fair amount of string parsing for this assignment. There are many options on how to parse a string; we suggest that you look into using regular expressions. ‘C’ provides a `regex` library: it is a bit clunky, but it is significantly more powerful than using alternatives (such as, e.g., `strtok_r` or `sscanf`).
- There are a few ways to test `httpserver`. Below, we assume that you started `httpserver` on port 1234 by executing the command `./httpserver 1234`. We also assume that you are using your client on the same machine upon which the server is currently executing:
 - You can use an HTTP Client, such as Firefox, Google Chrome, or Safari. These can be fun ways to see your server in action! But, they are extraordinarily robust: our experience is that such web clients will happily accept responses that are formatted incorrectly.
 - RECOMMENDED: You can test with a command-line web browser, such as `curl`. `curl` can produce both `GET` and `PUT` commands. For example, to execute a `GET` of the file `foo.txt` on `httpserver` and place the output into the file `download.txt`, you execute:

```
curl http://localhost:1234/foo.txt -o download.txt
```

Use the following `curl` command to execute a `PUT` request that puts the file, `foo.txt`, into the location `new.txt`:

```
curl http://localhost:1234/new.txt -T foo.txt
```

Note: `curl` has a few idiosyncrasies that you might accidentally rely upon in your assignment. In effect, we’ve seen students in the past struggle on this assignment because of assumptions that they baked into their server that are based upon supporting `curl` as a client. We have two suggestions: (1) In addition to using `curl`, use `cse130.nc` (see below) to have even more control over what bytes are sent to your server and (2) In addition to using the basic `curl` commands

shown above, also try clearing the `Expect` header by adding the following to the command-line `-H "Expect:"`

- **RECOMMENDED:** You will find a file, `cse130_nc`, in the resources repository. This is our port of a standard Linux utility, `nc` or “netcat”. To connect to your server, execute `./cse130_nc localhost 1234`. Then, you can type in the text that you wish to send to your server. You can also automate this approach by piping data to `cse130_nc`. For example, to send the `PUT` command listed above to your server, execute:

```
printf "PUT /foo.txt HTTP/1.1\r\nContent-Length: 12\r\n\r\nHello World!"
| ./cse130_nc localhost 1234
```

- If you try to start your server immediately after killing a previous instance of it, you will likely see the following error:

```
httpserver: bind error: Address already in use
```

In this case, just restart the server with a different port number. The issue is that the operating system must ensure unique ports are used across the entire system; it often waits to gracefully close ports even after the process that was using them terminates.

Getting Started and Design Tips

This is a pretty large project and there are multiple ways to get started. We describe an approach below based upon our experience working with students in the past; but you do not *have* to follow this section (that is why it’s under Hints!).

Design Approach

You should design your server before you start writing any code. Your design should articulate exact function prototypes and struct definitions that you plan to use. You may find it helpful to visualize how your server will function by drawing a picture similar to the “box-and-arrows” pictures that we’ve been drawing in class. By thoroughly designing your system, you will find that you can avoid many potential bugs, thereby saving you hours!

Keep in mind that your design will probably change as you build your server; sometimes, the act of trying to implement something illuminates how bad of an idea it was. If this happens, we encourage you to “go back to the drawing board” and re-design the server. If you find yourself implementing something that you haven’t “designed”, then you’re probably doing it wrong :-)

Step 1: Design your Scaffolding Start thinking about high-level “scaffolding”—essentially, what will your `main` function do? At a high level, `main` will first need to parse command-line arguments and create a listener socket for the port. Then, `main` will need to repeatedly: (1) accept a new connection, (2) perform the logic (i.e., process) that connection, and (3) close that connection (“repeatedly”—sounds like a loop!). You should probably use “top-down” modularization. Make each of these tasks (getting command line arguments, accepting connections, processing connections, and closing connections) a separate function. Don’t worry about *how* those functions work, just treat them as a “black box” that performs the task you expect of them!

Step 2: Design your Connection Processing There are many ways to approach connection processing; we lead you through one approach.

- (a) **Read enough bytes to ensure that you will have read the entire Request-Line from the request.** Since your `httpserver` needs to be efficient, you will need to read multiple bytes at a time and place them into a buffer.

There is one important difference between a socket and a file descriptor that matters for this step. Recall that `read` returns 0 when a program reaches the end of a file. In contrast, a client will not

always tell your server that it is done sending data. So, you might find yourself in a case where there is no new data to **read** from the socket, but the socket nonetheless does not return 0. If you use our helper functions in your assignment, you will eventually observe a timeout during **read**. But, you should not wait for that timeout before processing a correct request.

*Think: How should you change how you read bytes from a connection to tell if you have gotten the full **Request-Line**? (A hint: Does the **read_n_bytes** helper function help you?)*

- (b) **Identify each of the fields within the buffer that you read.** You'll want to next identify each of the fields that are contained in the buffer that you just read. *Think: What fields could be present in the buffer?*

There are many ways to parse the fields. For example, you could first separate the buffer into each high-level field (**Request-Line**, **Header-Fields**, **Message-Body**, (Can each of these be in your buffer?)) and then parse out sub-fields (e.g., **Method** within **Request-Line**). Or, you could parse each of the individual fields one by one from the beginning (i.e., start by parsing **Method** instead of **Request-Line**). Both of these approaches have merits; it is a personal preference what you choose. No matter what you choose, we suggest pulling out the 'C' regular expression library. It is clunky and awkward, but it is the best mechanism for the task at hand.

One important thing to decide upon is where you are going to store each of these fields. We suggest putting them into a struct. No matter where you put the fields, you should ensure that the lifetime of the data (the time that it is allocated, either through **malloc** or by virtue of being on the stack) covers the entire time that the fields are used. In other words: if you use a field in a function, **foo**, then the field must be allocated either (1) dynamically with **malloc**, **calloc**, etc. or (2) on the stack of a function, **bar** that either called **foo**, or calls a function that calls **foo**, or calls a function that calls a function that calls **foo**...

- (c) **Interpret your fields to perform GET, PUT logic or return a response indicating the invalidity in the request.** Finally, think about how to implement the logic in response to a client request. *Think: How can you leverage our helper functions to help you with this task?*

Implementation Approach

Below, we describe steps that can be used to implement your server.

Step 1: Build an Echo Server We suggest that you start building your server by building the Scaffolding. The scaffolding can work by simply repeating the bytes that the client writes to it. This ensures that your code can listen, accept, read from, and close connections.

Step 2: Sending Responses Next, we suggest that you design code that allows your server to send a response. This is important because it will be impossible to test whether your system can parse requests without being sure that your server can produce responses. Our suggestion is to unit test this: use the **cse130_nc** script to validate that your server produces correct responses before you move on to full-scale testing.

Step 3: Request Logic Then, we suggest that you start building code that performs **GET** and **PUT** requests. You might look back at your **memory** assignment and use our helper functions. At this point, you should be able to use any of the tests in Resources that handle "happy-path" scenarios. In other words: your code should work for tests that use requests that are supposed to return a **Status-Code** of 200 or 201.

Step 4: Parsing Requests Finally, we suggest that you start building code parses requests. This is the most complex task in this assignment and the one that trips up the most students. After this step, your server should be able to support all of the tests in Resources.

Signature	Description
<code>int listener_init(*Listener_Socket sock, int port)</code>	Initializes sock to be bound to localhost and listen on port . Note sock must already be allocated (i.e., it should either already be allocated on the stack or heap). Returns 0 if it was successful and -1 if it failed (either because port was in an invalid range, or because it could not bind to port).
<code>int listener_accept(*Listener_Socket sock)</code>	Blocks until a new client connection. Returns a socket for the new client connection.

Table 2: Listener socket helper functions.

Signature	Description
<code>ssize_t read_until(int fd, char buf[], ssize_t n, char *str)</code>	Reads bytes, placing them into buf , from the socket or file descriptor fd , until any of the following are true: (1) it has read exactly n bytes, (2) there are no more bytes to read from fd (i.e., read returned 0), or (3) the string, str is contained in buf . Returns the number of bytes read, or -1 if there was an error. If an error occurred, this function sets errno to be equal to the error that was encountered (see errno.h).
<code>ssize_t read_n_bytes(int fd, char buf[], ssize_t n)</code>	Reads bytes, placing them into buf , from the socket or file descriptor fd , until any of the following are true: (1) it has read exactly n bytes, (2) there are no more bytes to read from fd (i.e., read returned 0), or (3) there is an error reading bytes. Returns the number of bytes read, or -1 if there was an error. If an error occurs, this function sets errno to be equal to the error that was encountered (see errno.h).

Table 3: I/O wrapper helper functions for reading bytes.

Signature	Description
<code>ssize_t write_n_bytes(int fd, char buf[], ssize_t n)</code>	Writes bytes from <code>buf</code> into the socket or file descriptor <code>fd</code> , until any of the following are true: (1) it has written exactly <code>n</code> bytes, or (2) there is an error writing bytes. Returns 0 if successful, or -1 if there was an error. If an error occurs, this function sets <code>errno</code> to be equal to the error that was encountered (see <code>errno.h</code>).
<code>ssize_t pass_n_bytes(int src, int dst, ssize_t n)</code>	Reads bytes from the socket or file descriptor <code>src</code> and places them into the socket or file descriptor <code>dst</code> , until any of the following are true: (1) it has read/written exactly <code>n</code> bytes, (2) there are no more bytes to read from <code>src</code> (i.e., <code>read</code> returned 0), or (3) there is an error reading/writing bytes. Returns the number of bytes written if successful, or -1 if there was an error when reading/writing. If an error occurs, this function sets <code>errno</code> to be equal to the error that was encountered (see <code>errno.h</code>).

Table 4: I/O wrapper helper functions for writing bytes.