

# Extra Credit Programming Project: Parallel HTTP/1.1 Server

**Goal** Build a robust HTTP/1.1 server in C. Your server will be able to parse real protocols, implement HTTP operations, support multiple concurrent users, and do it safely and robustly. Overall, the goal is for you to complete a major design and implementation to highlight the things that you learned in CSE 130 this quarter!

Do the project in teams of 2 or 3. Each group member must submit the assignment to get credit for it. The group membership should be in a comment at the top of each source file.

You are allowed to use AI to help develop the code.

## Deliverables & Checkins

You will complete the project in two parts. If you want credit, parts 1 and 2 must be submitted before the final exam.

Your team should create a one-page high-level design covering modularization, abstractions, and the parallelization strategy that you plan for your server to use. Put the design in a comment at the top of your main code source file.

- **Part 1:** Produce an HTTP server that can handle one request at a time.
- **Part 2:** Extend the server with concurrency to handle multiple clients, and write a short performance report.

## Getting started

Find teammates. If you need help finding partners, use Ed.

Create a shared repository on UCSC's GitLab ([git.ucsc.edu](https://git.ucsc.edu)). One teammate should create a new project by clicking "New Project" on the home page on UCSC's Gitlab. Then, they should invite each group member as a Maintainer (go to Manage > Members from the left pane in the website). In addition, add the CSE 130 course staff (Scott Brandt, Nihal Talur) as Maintainers.

# Design Details

This is a big enough project that you need to design your server. The design should outline:

- What parallelization strategy will your server use?
- What abstractions will you use to divide your server into manageable modules? What are the functions and structs that you will create?
- How will you test your server locally (i.e., *outside* of the autograder)?

## Part 1

For Part 1, your team will produce an HTTP server that supports a single request at a time—i.e., a sequential HTTP server. Name the binary `httpserver`; it should take the following command-line arguments:

```
./httpserver [-p parallelism] <port>
```

Parameters:

- `<port>` (required): integer in [1,65535]. On failure to parse or bind, your server should print `Invalid Port\n` to `stderr` and exit with status 1.
- `-p parallelism` (optional): default 1. This specifies the amount of parallelism in the server; i.e., the number of processes or threads that your server should use.

**Overview** The high-level workflow of your server is as follows. It will listen on a TCP socket bound to `<port>`. It will repeatedly accept connections from clients, process the request, send the appropriate response, and then close the connection. For simplicity, each TCP connection contains at most **one** HTTP request. Your server should produce a message to standard error for each request it processes; we call this an *Audit Log*. We describe how to parse each request, process their behavior, and send a response in the section below called “Your server’s HTTP/1.1 protocol subset”.

**Submission Instructions:** Submit both Parts 1 and 2 to the CSE 130 autograder. **Note:** each member submits their own source code to receive a grade.

## Your server’s HTTP/1.1 protocol subset

Your server should support the following parts of the HTTP protocol.

### 0.0.1 Requests

Clients will send requests with the following format; you do not need to implement a client for this project. As shown below, both the *requestline* and *headerfieldlist* end with the special sequence `\r\n` (CRLF). At a high level, a request consists of four components:

*request* ::= *request-line header-field-list empty-line message-body?*

**Request line** The request line specifies the highlevel features of the client's command. The syntax is as follows:

```
request-line ::= method uri version \r\n
method      ::= [A-Za-z]{1,8}
uri         ::= /[A-Za-z0-9.-]{1,63}
version     ::= HTTP/[0-9]\.[0-9]
```

Your server should validate that the client's input matches the specifications listed above. For simplicity, your server can assume that the total number of bytes in *request-line+headers* is at most 2048. However, no matter what input request a client sends to your server, the server should not crash!

**Headers** Headers add additional details for some requests, but they're optional in most cases. Their syntax is as follows:

```
header-field ::= key: value \r\n
key          ::= [A-Za-z0-9.-]{1,128}
value        ::= [ -~]{1,128} (printable ASCII)
```

**Message Body** There are no restrictions on the data that is sent in a message body; they can contain arbitrary binary data.

## Responses

Your server should provide HTTP responses to the client, regardless of whether the client sends a valid HTTP Request. The table included in the semantics section below describes the appropriate status code, phrase, and message bodies for your server to send in a variety of scenarios. The response should follow the following format; note that the status line and headerfield list should end with the `\r\n` sequence, but message bodies do not.

```
response ::= status-line header-field-list empty-line message-body
status-line ::= HTTP/1.1 status-code status-phrase \r\n
```

## Semantics

After parsing the client request, your server will perform some actions to process them. How the server process each command depends upon the method and URI that the client sends, as well as the state of the system when the client sends the request.

The server should associate each URI in a request with the file with the same name from the server's current working directory. For example, if a request specifies `/foo.txt`, the associated file would be `foo.txt` in the server's current working directory. The server does not need to support subdirectories, so there is no need to support URIs like `/bar/foo.txt`.

The following table describes the possible responses that your server might send to a client. In all cases, your server should send a **Content-Length:** `<n>` header along with its response, where `<n>` is the number of bytes in the message body.

Code	Phrase	Body	When
200	OK	OK\n or file bytes	Successful GET/PUT (existing file)
201	Created	Created\n	Successful PUT that creates a new file
400	Bad Request	Bad Request\n	Malformed request (syntax, headers, length)
403	Forbidden	Forbidden\n	File not accessible
404	Not Found	Not Found\n	GET for non-existent file
500	Internal Server Error	Internal Server Error\n	Unexpected error
501	Not Implemented	Not Implemented\n	Unimplemented method
505	Version Not Supported	Version Not Supported\n	Version not HTTP/1.1

The specific semantics for client requests that your server should support are:

**GET** GET requests ask the server to get the contents of the file located at the specified URI. If the file associated with the URI exists and is readable, it should respond with a 200 and send the file's contents as the message body; it should be sure to set the **Content-Length** of the response appropriately! If the file does not exist, the server should respond with a 404; if it exists but is not readable, it should respond with a 403.

**PUT** PUT requests ask the server to place the contents from the request's message body into the file located at the specified URI. If the request includes the special **Content-Length** header, then your server should expect to see that many bytes in the message body. For example, if the request includes the header **Content-Length: 10**, then your server should expect to see 10 bytes in the message body. If the request does not include a **Content-Length**, your server should treat it as a PUT with an empty file. If the file associated with the URI exists, the server should overwrite the file's contents and respond with a 200 response with the default message body (see the table above). If the file does not exist, the server should create the file and respond with a 201 response. If the server experiences an error (e.g., **write** returns -1), it should respond with a 500 response.

**Syntactically valid requests** If the client's request is syntactically valid but not a GET or PUT, your server should return 501. If it is syntactically valid but does not include the HTTP version HTTP/1.1, it should send a 505 response.

**Syntactically invalid requests** If the client's request is syntactically invalid, the server should send a 400 response.

## Audit Log

For each request that your server processes, produce a line to standard error with the following format:

```
<METHOD>,<URI>,<Status-Code>,<Request-ID>\n
```

where `<METHOD>` is the request method, `<URI>` is the request URI, `<Status-Code>` is the returned status code, and `<Request-ID>` is the request ID. To identify the request-ID, your server should look for the optional header, `Request-ID`, from the request. If the header is not included in a request, it should treat the request-ID as the value 0.

## Part 2

Part 2 will require parallelizing your server so it can process multiple client requests concurrently. Even though your server supports concurrent client requests, it will need to properly uphold two properties:

1. For all pairs of requests,  $R_1$  and  $R_2$ , if  $R_2$  starts after  $R_1$  finishes, then  $R_2$  must occur after  $R_1$  in the audit log.
2. For all pairs of requests,  $R_1$  and  $R_2$ , if  $R_2$  appears after  $R_1$  in the audit log, then  $R_2$  must observe the effects of  $R_1$ , as described in the semantics above.

Moreover, your server should synchronize only when necessary to ensure these properties—this property is vital to ensure performance of a server.

**What to Submit** Submit Part 2 to the CSE 130 autograder. **Note:** each member submits their own source code to receive a grade.

## Implementation Constraints

- Your server should be written in C.
- Your server cannot use third-party HTTP or parsing libraries; all of your server's dependencies should be included in the standard GNU libc and pthread libraries.
- Your server cannot use `system`, `popen`, or `spawn` shells.
- Your server should not crash on malformed input.
- Your server should not leak file descriptors or memory.

## Examples

Assume a directory with `foo.txt` containing `Hello` (5 bytes).

### GET existing

```
Client: GET /foo.txt HTTP/1.1\r\n\r\n
Server: HTTP/1.1 200 OK\r\nContent-Length: 5\r\n\r\nHello
```

## PUT new

```
Client: PUT /bar.txt HTTP/1.1\r\nContent-Length: 3\r\n\r\nbye
Server: HTTP/1.1 201 Created\r\nContent-Length: 8\r\n\r\nCreated\r\n
```

## Rubric

The following rubric includes all aspects of the final project.

Category	Points
Team and Repo	5
Design	10
Makefile	10
Clang-Format	5
Sequential Functionality (Part 1)	35
Parallel Functionality (Part 2)	35
Total	100

We will grade Team & Repo (Checkin 1) via the Checkin 1 Canvas assignment; Design Doc (Checkin 2) via the Checkin 2 Canvas assignment; and the Makefile, ClangFormat, Sequential Functionality (Part 1), and Parallel Functionality (Part 2) via the autograder.