# 15-441/641: Networking and the Internet
# Project 3: HTTP

TAs: Zili Meng, Hugo Sadok, Nirav Atre

**Last Update**: November 14, 2022
**Checkpoint 1 due**: 6:00 pm on November 22nd, 2022
**Checkpoint 2 due**: 6:00 pm on December 7th, 2022

## 1 Introduction

In this project, you will explore the application layer from the perspective of a networked system we interact with all the time: a *web server*. Recall that the goal of a web server is to deliver *web pages* (or, more generally, *web resources*) to clients (e.g., web browsers) across the world. To realize this goal, both the server and client must speak a common "language": the Hypertext Transfer Protocol (HTTP).[1] You will use the Berkeley Sockets API to write a web server using a subset of HTTP 1.1. 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!

In reality, our goals are the following: for you to practice building non-trivial computer systems in a low-level language (C), complying with real industry standards (as all developers of Internet services must!), understanding the benefits of various performance optimizations, and knowing how to balance trade-offs in different scenarios. This project is divided into two checkpoints. In the first checkpoint, you will implement the HTTP server, which will allow you to load the served web content in a browser. In the second checkpoint, you will implement the HTTP client to evaluate the relative impact of two performance optimizations (e.g., pipelining and parallel connections).

## 2 Checkpoint 1: HTTP Server Functionality

In this checkpoint, you will implement an HTTP server capable of parsing and serving HTTP requests from multiple concurrent clients. This section provides an overview of the complete requirements for the *Liso* web server (i.e., what you will turn in for this checkpoint). The starting point for your server is framework code that we provide on GitHub: https://github.com/computer-networks/cmu-http.[2]

### 2.1 Background

For this project, you will use `socket()`, the most commonly used API in network-related development. In P1 and P2, we handled the connections between machines for you. In P3, you are required to set up connections yourself using a **TCP** socket. There are several good tutorials on setting up a TCP socket,[3] but be careful with the various parameters in the `socket()` (and related) system calls. You can learn about these parameters here.

You will also use the Unix directory API to deal with local files and folders. For example, in the starter code, we demonstrate the usage of `opendir()` and `closedir()`, which are commonly used for directory processing in C. You can learn more about this API here.

### 2.2 Starting the Web Server

The Liso server takes a single command line argument: the absolute path to a directory containing the web resources to be served. You may not modify Liso to, e.g., take a different number of arguments or reorder the arguments.
***usage:*** *./server <www folder>*

[1]Or its secure variant, the Hypertext Transfer Protocol *Secure* (HTTPS)... we're not an imaginative lot.
[2]Sorry, here is legitimately the 100% valid, course staff-approved hyperlink.
[3]TCP Server-Client implementation in C - GeeksforGeeks

## 2.3 Supporting HTTP 1.1

The HTTP 1.1 standard is defined in RFC 2616.[4] You will find that the full specification is long and complicated; for the purpose of this project, we only care about three methods:

- **GET**: requests the specified resource, and the response should contain the bytes associated with the resource.
- **HEAD**: requests the specified resource, but, unlike GET, the response should not include the body (i.e., no bytes from the requested resource)
- **POST**: submit data to be processed to an identified resource; the data is contained in the body of the request, and side-effects are expected. The response to POST requests is echo-replying the body in POST requests if the request is correctly formatted.

**HTTP Parsing**: Parsing HTTP messages is a cumbersome task involving tokenization[5] and evaluating grammar.[6] To help you get started, the starter code implements the HTTP parser for you. More specifically, we provide the following three functions that you may find helpful:

```
/**
 * @brief      Serialize an HTTP request from the Request struct to a buffer.
 *
 * @param      buffer  The buffer (output)
 * @param      size    The size of the buffer (output)
 * @param      request The request (input)
 * @return     the error code
 */
test_error_code_t serialize_http_request(char *buffer, size_t *size, Request *request);

/**
 * @brief      Parse an HTTP request from a buffer to a Request struct.
 *
 * @param      buffer  The buffer (input)
 * @param      size    The size of the buffer (input)
 * @param      request The request (output)
 * @return     the error code
 */
test_error_code_t parse_http_request(char *buffer, size_t size, Request *request);

/**
 * @brief      Serialize an HTTP response from the Request struct to a buffer.
 *
 * @param      msg                   The message (output)
 * @param      len                   The length of the message (output)
 * @param      prepopulated_headers  The prepopulated headers (input)
 * @param      content_type          The content type (input)
 * @param      content_length        The content length (input)
 * @param      last_modified         The last modified time (input)
 * @param      body_len              The HTTP body length (input)
 * @param      body                  The HTTP body (input)
 * @return     the error code
 */
test_error_code_t serialize_http_response(char **msg, size_t *len,
```

[4]RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1
[5]https://en.wikipedia.org/wiki/Lexical_analysis
[6]https://en.wikipedia.org/wiki/Formal_grammar

```
const char *prepopulated_headers, char *content_type, char *content_length,
char *last_modified, size_t body_len, char *body);
```

Note that we do not provide an implementation for `parse_http_response()`. You will implement this yourself in Checkpoint 2.

**Error Messages**: Your server should support the following HTTP response codes, some of which you may have already encountered while browsing the web:

- `HTTP 200`. Success message indicating that the HTTP request was successful.
- `HTTP 404`. If the requested resource is not available on the server (e.g., requesting a non-existent file), the server should reply with `404 File Not Found`.
- `HTTP 503`. When the server has more than 100 active connections, the server should respond to the requests from new connections with `503 Service Unavailable` to protect the server from being overloaded.
- `HTTP 400`. All other requests should be responded with `400 Bad Request`.

Among three error codes, `HTTP 503` has the highest priority – if the server has already been overloaded, the server should return 503 even if the file does not exit. **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 may not be robust to malformed requests, so in some scenarios 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. Your server must support multiple requests in the same connection.

## 2.4 Supporting Many Clients

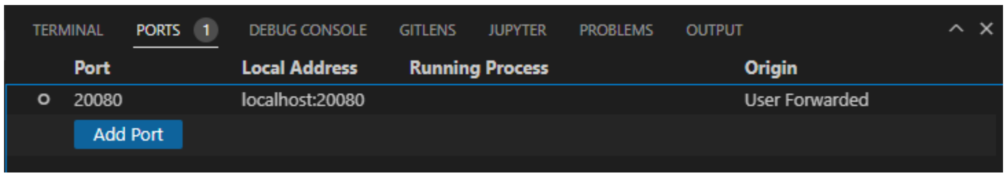
Your server should be able to support multiple clients concurrently. While the server is waiting for a client to send the next command, it should be able to handle inputs from other clients. Clients may "stall" (send half of a request and then stall before sending the rest) or cause errors; these problems 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 this project, you **must implement your server using `poll()` to support concurrent connections**. Threads or `select()` are **NOT** permitted at all for the project. The `poll()` feature allows programs to multiplex input and output through a series of file descriptors. More detailed instructions on how to use `poll()` can be found at How to Use Poll System Call in C.
In summary, your server should:

1. Use `poll()` to handle concurrent requests;
2. Respond with HTTP 404 (File Not Found) when a client requests a non-existent resource;
3. When the server has more than 100 active connections, respond to new connection requests with HTTP 503 (Service Unavailable) until previous ones finish.

Next, we describe two tools to help you debug your web server's functionality and performance.

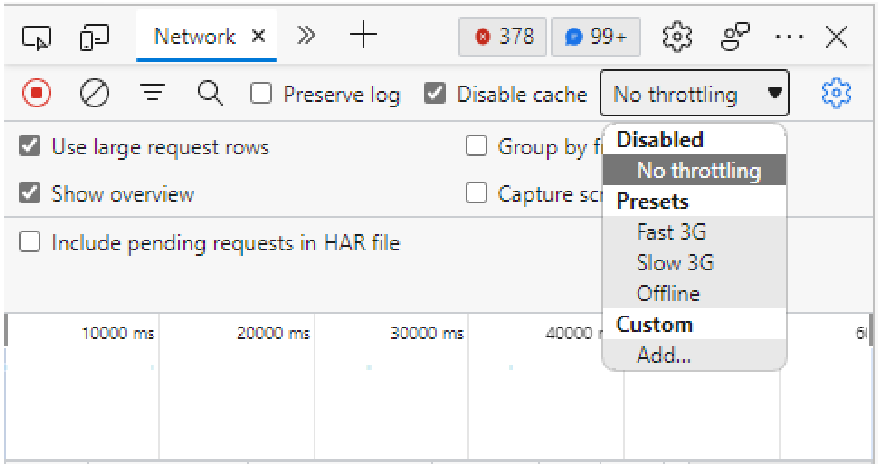
**1. Port Forwarding or Binding**: The easiest way to validate your server functionality is to load the webpage in a browser! This is straightforward to do if you're running the server locally: simply navigate to `localhost:20080` in a browser of your choice. If you deploy the HTTP server on a remote machine (e.g., an AWS instance), you can either view the webpage by navigating to `${RemoteIP}:20080` (where ${RemoteIP} is the *public* IP address of the remote machine), or, alternatively, forward port 20080 on the remote machine to your localhost. For instance, if you're using Visual Studio Code, you can enable port forwarding in the 'Ports' tab:

Here 20080 is defined in `include/ports.h` and is the port that we are going to use for the HTTP server.



```
TERMINAL   PORTS 1   DEBUG CONSOLE   GITLENS   JUPYTER   PROBLEMS   OUTPUT
     Port       Local Address      Running Process            Origin
  ○  20080      localhost:20080                               User Forwarded
     Add Port
```

**2. DevTools**: DevTools is a powerful tool to profile HTTP performance and emulate network conditions in Chrome and other Chromium-based browsers (e.g., Edge). You can learn more about DevTools at Open Chrome DevTools - Chrome Developers. In most cases, pressing F12 in the browser window will open the DevTools console. In the 'Network' tab (depicted below), you can view a timeline of how different elements in an HTML page are loaded. You will also be able to emulate different network conditions – click on 'No throttling' to open the corresponding dropdown menu, then set a custom delay and bandwidth.



We provide four webpages for you to test your server implementation in `cp1`:

- `test_visual`: A simple webpage with some hypertext and an image.
- `test_single`: A webpage containing a single, large file of size ∼1MB.
- `test_multiple`: A webpage containing a number of small files, with a total size of ∼1MB.
- `test_dependency`: A webpage containing a number of files which depend on each other, with a total size of ∼1MB.

## 2.5 Implementation Component

You should use the provided script to create an archive with your submission. For the autograded portion of checkpoint 1, the grade breakdown is approximately as follows:

- **Handling normal requests (50%).** These testcases will exercise your server with a set of properly-formed requests. Your implementation should respond with HTTP 200 and the expected data.
- **Handling malformed requests (20%).** These testcases exercise your server with malformed requests. Your implementation should never crash, returning HTTP 400 for these requests.
- **Handling non-existent files (10%).** These testcases exercise your server with requests for one or more non-existent files. For such requests, your implementation should return HTTP 404.

- **Overload protection (20%).** We will use `Apache Bench`[7] and `siege`[8] to create concurrent connections to your server. Your implementation should handle up to 100 maximum concurrent connections, returning HTTP 503 to additional connection requests.

The starter code includes four example webpages, but the autograder tests will *not* be public. Please write your own test-cases for the supported features (e.g., handling malformed requests and non-existent files).

### 2.6 Experimental Component

For the experimental component, you will measure the *page load time* (i.e., how long it takes to finish loading a web page) under different RTTs (10ms, 20ms, 40ms, 80ms, 160ms) and link bandwidths (100kbps, 1Mbps, 10Mbps). You should emulate these network conditions using the DevTools in Chrome. You will evaluate two sets of network configurations: (a) when the bandwidth is fixed to 1Mbps and the RTT varies from 10ms to 160ms, and (b) when the RTT is fixed to 40ms and the bandwidth varies from 100kbps to 10Mbps. For each network condition, you will measure the page load time for `test_single`, `test_multiple`, and `test_dependency`.

**Hypothesis**: Without performing any experiments, answer the following questions.

1. When the RTT is 40ms and bandwidth is 1Mbps, predict the relative ordering (in increasing order of page loading time) of these three websites.
2. When the RTT changes from 10ms to 160ms, and the bandwidth is fixed to 1Mbps, how do you expect the page load time to change? Do you expect a linear relationship between the RTT and page loading time? Why or why not?
3. When the bandwidth changes from 100kbps to 10Mbps, and the RTT is fixed to 40ms, how do you expect the page load time to change? Do you expect a linear relationship between the bandwidth and page loading time? Why or why not?

**Experiments**: Create the following plots:

- Plot 1. A bar plot showing page load time (in ms) for the three different webpages.
- Plot 2. Line plot where the x-axis represents the manually added delay, and the y-axis represents the page load time. There should be three lines, representing the three webpages.
- Plot 3. Line plot where the x-axis represents the manually throttled bandwidth, and the y-axis represents the page load time. There should be three lines, representing the three webpages.

**Inferences**: Based on the results of your experiments, answer the following questions:

- For each of your hypotheses, are your experimental results aligned with your predictions? What do you observe in the relationship between page size and page load time? Why?

## 3 Checkpoint 2: HTTP Optimizations

In this checkpoint, you will implement an HTTP client, replacing the browser that you used in the Checkpoint 1. You will implement two optimizations employed by HTTP: *multiple connections* and *pipelining*. You will also evaluate the performance impact of physical separation between the client and server.

### 3.1 Starting the Client

The Liso client also takes a single command line argument: the IP address of the HTTP server. You may not modify Liso to, e.g., take a different number of arguments or reorder the arguments.

***usage:*** *./client <server-ip>*

[7] ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4.
[8] Siege is an http load tester and benchmarking utility.

## 3.2 Pipelining and Parallel Connections

HTTP pipelining and parallel connections are the two HTTP optimizations you will implement.

- **Pipelining**.[9] HTTP pipelining is a feature of HTTP/1.1 which allows multiple HTTP requests to be sent over a single TCP connection without waiting for the corresponding responses. Similar to the sliding window you implemented for CMU-TCP, HTTP also allows more than one *inflight* requests.
- **Parallel Connections**.[10] With parallel connections, one client can simultaneously create multiple connections to the server, improving the page load time in some scenarios.

**Understanding dependencies**: When using the browser as a client, the client first requests the `index.html` file. The client then parses `index.html` to determine what resource(s) to request next. Faithfully implementing HTML dependency parsing is cumbersome; instead, we provide a `dependency.csv` file, describing all the files in the current webpage and their dependencies. As long as you request `dependency.csv`, you will know the elements that the client needs to request. The first column of the `dependency.csv` is the name (URI) of the element, and the second column is its dependency (empty if the client can directly request the file). In your client implementation, you must make a GET request for `dependency.csv`, parse the dependency graph, then issue HTTP requests adhering to the dependency relationships between resources.

## 3.3 Autograder

For the autograded portion of checkpoint 2, the grade breakdown is approximately as follows:

- **Functionality (40%).** The client should be able to load the given webpage and save all files to a local `./www/` folder, following the dependency graph.
- **Pipelining (30%).** When loading the given web page, your client must issue as many concurrent *inflight* requests as possible.
- **Parallel connection (30%).** The client should spawn multiple connections to the same HTTP server in order to send parallel HTTP requests.

## 3.4 Experiments

For the experimental portion, you will first investigate the interaction between resource dependencies and parallel connections. Next, you will evaluate the impact of proximity between a server and its clients on the overall page load time.

**Hypothesis**: Without performing any experiments, answer the following questions.

1. Given a different number of parallel connections (1, 2, 4, and 8), what is the impact on page load time for each of the three webpages?
2. Imagine you deployed your HTTP server in 4 different AWS regions (one in North America, one in Europe, one in East Asia, and one in Australia). How would the geographic distance between your client (running locally on your machine) and the server affect the page load time for different webpages?

**Experiments**: To investigate the behavior over the real Internet, you will perform the following experiments and plot the following:

1. Sweep the number of parallel connections from 1 to 8, and deploy both the HTTP server and client in a single AWS instance. Measure how the page load time changes as a function of the number of parallel connections for each webpage. Create a line plot where the x-axis represents the number of parallel connections, and the y-axis represents page load time. There should be three lines, representing the three webpages.

[9] HTTP pipelining - Wikipedia.
[10] 4.4. Parallel Connections - HTTP: The Definitive Guide [Book].

2. Deploy an HTTP server in the four aforementioned AWS regions, and the client on your machine. Measure the page load time for each webpage. Create a line-plot where the x-axis represents physical distance, and the y-axis represents page load time. There should be three lines, representing the three webpages.

**Inferences**: Based on the results of your experiments, answer the following questions:

1. Do the results corroborate your hypothesis? If the page load time *does* improve with number of parallel connections, explain whether the relationship is linear, superlinear, or sublinear, and why. If not, explain why not.
2. Do the results corroborate your hypothesis? Why or why not?

## 4 Logistics

**Starter code.** We reserve the right to change the starter code as the project progresses to fix bugs and to introduce new features that will help you debug your code (we commit to making the changes as transparent as possible so as to not break your working implementation). You are responsible for checking Ed to stay up-to-date on these changes. We will assume that all students in the class will read and be aware of any information posted to Ed.

**Reusing code.** You are permitted to use any code you have previously written (e.g., from 15-213/513, a hackathon, etc.), but you may not use code from anyone or anywhere else. You are welcome to clone the project repository to stay up-to-date with changes (e.g., bug-fixes), but please do not push your solution code to a public fork. We take academic integrity violations *very* seriously, and you will be held just as liable for sharing code with someone as you will be for using it.

**Testing.** The starter code includes a suite of public test-cases to help you get started, but the autograder may use private test-cases to grade your solution (you will see the final score at the time of submission, though). Please come up with your own tests; it'll save both you and your partner a lot of debugging pain in the long term!

## 5 Resources

This section describes some recommended tools to develop the HTTP server and client.

### 5.1 How do I debug?

So far, you have built two fairly complex networked systems involving several moving parts and APIs you haven't seen before. Naturally, we have higher expectations from you! For this project, we expect that:

- You will first try to debug your code on your own. 'My code doesn't work' is not a valid request for help.
- When you go to office hours, please provide a minimal working example (MWE).[11] The MWE should be as small and as simple as possible, and also be able to demonstrate the problem. Constructing a MWE is a good way to help you to localize the issue.
- 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).

### 5.2 Tcpdump and Wireshark

`tcpdump` and Wireshark is a pair of tools that every network engineer or researcher should be excel at. You can capture everything going through the network with `tcpdump`, and analyze yourself with Wireshark. In P2, we provide a wrapped script for you to capture the packets and analyze. This is not the case for P3. In P3, you need to learn how to use `tcpdump` and Wireshark yourself. We will also go through them in the recitation.

[11] How to create a Minimal, Reproducible Example - Help Center - Stack Overflow.

## 6 Acknowledgement

We acknowledge the efforts of TAs for previous iterations of this course in designing this project. We also thank Joseph Koshakow (15-641, Fall 2021) who kindly allowed us to reuse part of his HTTP parser.