Important Dates

Assigned on **11/3/2020 0:01AM**
Due on **11/17/2020 23:59PM**

Instructions and Notes

Each student must complete and submit the project deliverables individually. This project is partly based on an assignment by Kurose and Ross, but with significant differences.

Project Overview

In this assignment, you will learn the basics of socket programming for TCP connections in Python: how to create a socket, bind it to a specific address and port, as well as send and receive a HTTP packet. You will also learn some basics of HTTP header format, and notion of multithreaded programming.

You will develop a simple web server using Python's networking primitives to serve multiple requests concurrently. For each request, your web server should accept and parse the HTTP request, get the requested file from the server's file system, create an HTTP response message consisting of the requested file preceded by header lines, and then send the response directly to the client. If the requested file is not present in the server, the server should send an HTTP "404 Not Found" message back to the client.

Note: this assignment must be completed using Python 3.

Project specifications

1. The server must serve HTTP GET requests from clients. Requests are assumed to be for static files (i.e., the server only serve files, and lacks the capability of executing any server-side code)

2. The server should serve files from its startup directory. If an HTTP request contains a path to the parent folder (e.g. **GET /../../important_file.txt HTTP/1.1**), the request should be rejected with an error (more info below)

3. The server should support the GET method, and HTTP versions 1.0 and 1.1. Paths can be assumed to start with a '/' (e.g., **GET /index.html** rather than **GET http://index.html**). Note that paths are relative to the current folder; i.e., if you execute your server from the **/tmp** directory, **GET /index.html HTTP/1.1** will refer to **/tmp/index.html**.

4. The server must accept but ignore all HTTP header fields with the exception of a custom field called 'X-additional-wait', which receives a numeric parameter. We will use this to introduce artificial delays within requests to evaluate concurrency. If such header field is present, your server should wait for the number of seconds passed as parameter. For example, when serving a request of the form:
   GET /index.html HTTP/1.1
   X-additional-wait: 5
   The server should accept the client connection, wait 5 seconds, and then serve the request.

5. The server must be able to serve multiple concurrent requests. E.g., if a new request arrives while a previous one is being served, the new request shall not fail or have to wait for the previous one to complete. Instead, it should be served immediately. You are expected to learn the basics of Python's **threading** module and use it to implement this feature.

6. The server should support the following command-line options. Such options may be passed in any order.
   a. --port : port on which the HTTP server should listen for connections. Upon startup, the server must listen on this port. If this option is not specified, the server should listen on port 8080. The server must always bind to the 127.0.0.1 IP address ("localhost")
   b. --maxrq <# REQUESTS>: maximum number of concurrent requests. If a new request arrives while # REQUESTS requests are already pending, the server must immediately disconnect the client. Default: 10 requests.

   c. --timeout : maximum number of seconds to wait for a client. If a client connects but more than SECONDS seconds elapse before the client submits a GET request, the server must disconnect the client. Default: 10 seconds.

Error management

Requests containing invalid paths (requests containing "../" in the path, or requests for files that do not exist) must receive an HTTP 404 error. For all other errors (e.g. invalid HTTP method, non-ASCII characters, etc.) it is acceptable to immediately terminate the connection.

Logging

The server must log to stderr the following instances:

- New connection from client: **"Information: received new connection from <IP ADDRESS>, port <PORT>"**
- Timeout while waiting for data: **"Error: socket recv timed out"**
- End of data while waiting for more input: **"Error: unexpected end of input"**
- Invalid character encoding (non-ASCII characters in input): **"Error: invalid input character"**
- Incorrectly formatted request line (method other than GET, unknown HTTP version, etc.): **"Error: invalid request line"**
- Incorrectly formatted headers: **"Error: invalid headers"**
- Too many concurrent requests: **"Error: too many requests"**
- Malformed or non-existing file path: **"Error: invalid path"**
- Successfully served a file: **"Success: served file <FILE NAME>"**

To be clear, these messages **must not be returned to the client**; instead, they **must be printed to console** while the server is executing. It is acceptable to provide additional information in the output, but the strings above **must appear in the output** when the corresponding event occurs.

Important: logging must be thread-safe, i.e., if two threads concurrently attempt to print log messages, their messages must appear sequentially and must not interleave.

Constraints

You can use the functionality provided by the core Python language (those which do not necessitate to import any module in order to function.) Additionally, you are required to implement network I/O using the Python **socket** module, and support for multiple concurrent requests using the **threading** module. You may use the **argparse** Python module to perform command-line option parsing. Finally, you may use functionality provided by the **time, sys** and **os** modules. Use of any other functionality (e.g. high-level HTTP parsing libraries) will disqualify your submission. You must also use the following HTTP response statuses:

errmsg = 'HTTP/1.1 404 NOT FOUND\r\n\r\n'
response10 = 'HTTP/1.0 200 OK\r\n\r\n'
response11 = 'HTTP/1.1 200 OK\r\n\r\n'

Additional material

You can find additional information on HTTP, Python, etc. in the book. You will also be able to find plenty of online resources on these topics.

Testing your work

In order to test whether your server works, you can use the **curl** or **wget** utilities, e.g., **wget http://127.0.0.1:8080/mytestfile.txt**. wget also allows you to pass custom HTTP headers (necessary to test the X-additional-wait option) via the **--header** option. You also use **telnet**, which allows you to establish a TCP socket and send/receive characters to/from it (if you use telnet, make sure your telnet client is configured to send "\r\n" when you press Enter, otherwise things won't work).

Deliverables and Grading

Upload your work to the appropriate Canvas assignment as a Python file called **httpserver.py**. Include the entire server logic in a single Python file.

Grading rubric:

7. **1 point:** Basic connection establishment. The server must accept valid TCP socket connections. The code must be Python and parse correctly (i.e., no syntax errors when we attempt to run it).

8. **2.5 points:** Basic HTTP service. The server must parse a correctly formed request and serve the requested file if it exists. Prerequisite: Task 1.
9. **2.5 points:** Support for multiple clients via threads. The code must properly address all concurrency issues that could arise. Prerequisite: Task 2.
10. **1.5 point:** Error checking (with security checks) and logging must be fully implemented and correct. Prerequisite: Task 2.

11. **0.5 points:** Correct support for command line arguments and dynamic listen port. Prerequisite: Task 2.
12. **0.5 points:** Proper support for time-outs. Prerequisite: Task 2.
13. **0.5 points:** Proper support for maximum users. Prerequisite: Task 3.
14. **1 point:** If we load a simple static website into your server and we use the Firefox browser to connect to it, the website is displayed correctly.