HW 5 - The HTTP/1.1 Server

CS 361 - Fall 2024 - Maratos

Description

HyperText Transfer Protocol (HTTP) is one of the most commonly used protocols to exchange data in the application layer. All transactions take the form of a request followed by a response and in class we model this as an exchange between a client and server process. In this assignment you will implement a server process that performs transactions in the HTTP/1.1 protocol. You will learn how to process requests and send various responses, all common tasks for the server process. This document will describe the various types of transactions that your server will fulfill and the test cases associated with this functionality.

This document first presents the format for describing HTTP protocol sequences that is used throughout the rest of this document in Section 1. Then this document presents each test group in the autograder and some occasional hints in Section 2. You do not necessarily need to read the entire document before starting the assignment. We recommend reading Section 1 first and then read the relevant test group section as you work on the assignment.

1 HTTP Requests and Responses

Description	Symbol
Line changes (`\r\n`) in HTTP	$\bigtriangledown$
Space characters	$\bullet$
End of message	$\otimes$

In the HTTP protocol, **line changes** are signified by the pair of characters `\r\n` and to make it clear when you need to write this in your message we will use the symbol $\bigtriangledown$ to show that you need to insert a line change. This is commonly known as Carriage Return Line Feed (CRLF). **Spaces** are part of the HTTP protocol but it can be difficult to indicate their presence (or how many there are) we will use the symbol $\bullet$ to show when a single space character is needed.

Messages in the HTTP protocol *do not* end in the `NULL` character so do not use them when constructing your messages. We will use the symbol $\otimes$ to indicate the **end of the message**. It will informally mean *no more data* and when constructing the message you should add no further characters once you encounter it.

1.1 HTTP Request Format

Requests are HTTP messages sent by clients. They have the following format.

```
1 <METHOD>•<URL>•HTTP/1.1▽
2 <HEADERS>▽
3 ▽
4 <BODY>⊗
```

The various components of the request are described below.

- Line 1 is the *request line* and it specifies the type of request being made from the client.
- HTTP requests can have zero or more *request headers*. Line 2 is an abbreviated representation, we use to make this document easier to read. This line should be interpreted as follows.

abbreviated version	what it actually means
<HEADERS>▽	<Header>:•<Value>▽ <Header>:•<Value>▽ ... <Header>:•<Value>▽

- Line 3 is a line only containing a single CRLF. This is is part of the HTTP protocol and specifies that the header has ended. The remaining bytes will be the body of the request.
- Line 4 is a shorthand for the entire body of the message followed by the ⊗ symbol which indicates the end of the request.

1.2 HTTP Responses

Responses are HTTP messages sent by the server in response to a client request. They have the following form.

```
HTTP/1.1•<CODE>▽
<HEADERS>▽
▽
<BODY>⊗
```

The field <CODE> communicates whether or not the request was fulfilled or not. If the server can fulfill the request then the message will typically have the following form.

```
HTTP/1.1•200•OK▽
<HEADERS>▽
▽
<BODY>⊗
```

2 Test Cases

The autograder will act as a client to your server. Each individual test will establish a connection, send a request to your server, then validate that the response with the specification described in this document[1]. It can be challenging to precisely communicate what byte sequences the server/client will send to each other when some of those correspond to ASCII characters that are invisible to `STDOUT`. Read Section 1, if you have not already, which describes how this document presents the request/response byte sequences.

2.1 Establishing A Connection (Test 01)

You server should be able to establish a connection with a client. This test should be easy to complete if you already have done Lab 07 as you can reuse the `open_listenfd` function implemented there. No client in this test group will actually send a request after the connection is established. For each accepted request, it is sufficient to immediately close the connection to pass these tests. It is important to note that if your server only accepts a single connection request before terminating then it will still fail because the autograder will always check if a connection can be established before running any tests, including Test Group 01.

Remember that you can use the function `accept` to accept connection requests. We suggest that you design your server so that it will indefinitely run the following procedure at first.

1. accept the new connection request
2. receive the request from the client
3. handle the request
4. close the connection

The last test group will have you handle clients concurrently so you will have to rework this algorithm when you get to that stage.
Important Note: when configuring the listening socket, add the code segment below to ensure that the socket can be re-binded to the address `127.0.0.1:<PORT_NUM>` without causing an error. Assuming the listening socket is called `listenfd`. This is important for the autograder to function properly.

```
int optval = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
```

Important Note: If you are working on the server then you will be sharing the ports with other students. What this means is that you need to select an open port or else your server will not be able to bind it's socket to the socket address. When every user was created I ensured that users obtained an id that also corresponded to a valid port number. You can use this port in your tests by running the following command line which writes your id to a file called `port.txt`.

```
$ printf "$(id -u)" >port.txt
```

Ensure that this file is in the same directory as the `runtests.sh` script.

[1] Some of the later tests are more complicated and may repeat some of these steps.

2.2 Ping Requests (Test 02)

Your server will respond to `GET` requests for the URL `/ping` by sending an HTTP response with the body containing the message `pong`. See below for the details of the transaction.

Request

```
GET•/ping•HTTP/1.1▽
<HEADERS>▽
▽
⊗
```

Response

```
HTTP/1.1•200•OK▽
Content-Length:•4▽
▽
pong⊗
```

We provide the code below to assist you with sending responses. The purpose of `send_data` is to ensure that the entire message is sent. In most cases, you should not call `send_data` directly but instead call `send_response`. You should call the function `send_response` after you have correctly prepared the HTTP message `header` and `body`, which are declared in the starter code as global character buffers. Furthermore, make sure to set `HSIZE` to be the number of header bytes to send and `BSIZE` to be the number of body bytes to send.

```
static void send_data(int clientfd, char buf[], int size)
{
    ssize_t amt, total = 0;
    do {
        amt = send(clientfd, buf + total, size - total, 0);
        total += amt;
    } while (total < size);
}

static void send_response(int clientfd)
{
    send_data(clientfd, header, HSIZE);
    send_data(clientfd, body, BSIZE);
}
```

Here is an overview of how you should handle requests in general, using the global variables provided in the starter code.

1. read the client's request into `char request[]`
2. determine the type of request
3. Prepare the header and body of the response by doing the following
    - prepare the header into `char header[]` and set `HSIZE` to be the number of header bytes to send
    - prepare the header into `char body[]` and set `BSIZE` to be the number of body bytes to send

4. call `send_response` to send the response
5. `close` the connection

2.3 Echo Request (Test 03)

Your server will respond to `GET` requests for the URL `/echo`. The request will contain a sequence of header values that are CRLF delimited. When the client makes this request, the server response should be to echo the headers as the body of the message. It may help to inspect the expected output for this test group if you are unsure what to do when there is more than one header. Below is an example of this behavior with a single header.

Request

```
GET•/echo•HTTP/1.1▽
Header:•Value▽
▽
⊗
```

Response

```
HTTP/1.1•200•OK▽
Content-Length:•13▽
▽
Header:•Value⊗
```

2.4 Error Handling (Test 04)

If the request cannot be fulfilled then the server should respond with a `<CODE>` that indicates what caused the failure. Below we list two codes that are relevant to this test group.

Bad Request

```
HTTP/1.1•400•Bad•Request▽
▽
⊗
```

Request Entity Too Large

```
HTTP/1.1•413•Request•Entity•Too•Large▽
▽
⊗
```

In general, the code `400` should be returned when the request made to the server is invalid. The code `413` should be sent when the request is too large for the server to handle. Below is a list of scenarios where an error should be sent.

- If the request is not valid HTTP, for example there is no double `CRLF` to signify the end of the headers then respond with error `400`.
- For echo requests, the body of the response should be at most 1024 bytes. If the response to send exceeds that then send error `413`.

2.5 Read and Write (Test 05)

The server should have functionality for saving and retrieving data given by the client. The client will access the URL's `/write` and `/read` to use this feature. Below is an example request for saving data given by the client.

Request

```
POST•/write•HTTP/1.1▽
<HEADERS>▽
Content-Length:•24▽
<HEADERS>▽
▽
This•is•the•data•to•save⊗
```

Response

```
HTTP/1.1•200•OK▽
Content-Length:•24▽
▽
This•is•the•data•to•save⊗
```

Your server will echo the body of the request back to the client and save that data for later requests. Your server should be using the header `Content-Length` to determine how many bytes to save. Accessing the URL `/write` should be done with the `POST` method as opposed to `GET` (which was for all previous requests). Finally note that `Content-Length` could be embedded anywhere within the header section. There may or may not be header lines before or after the `Content-Length` header.

Once the data has been `POST`ed, the client can retrieve it in subsequent requests. See the example below which shows the client saving the data in the first request and then making a second request to retrieve it by accessing the URL `/read`.

Request

```
GET•/read•HTTP/1.1▽
<HEADERS>▽
▽
⊗
```

Response

```
HTTP/1.1•200•OK▽
Content-Length:•7▽
▽
<empty>⊗
```

```
POST•/write•HTTP/1.1▽
<HEADERS>▽
Content-Length:•24▽
<HEADERS>▽
▽
This•is•the•data•to•save⊗
⊗
```

```
HTTP/1.1•200•OK▽
Content-Length:•24▽
▽
This•is•the•data•to•save⊗
```

```
GET•/read•HTTP/1.1▽
<HEADERS>▽
▽
⊗
```

```
HTTP/1.1•200•OK▽
Content-Length:•24▽
▽
This•is•the•data•to•save⊗
```

You can assume that `Content-Length` exists in every `write` request, and it's value is a number. However, the body sent by the client may have different length than specified in the header. If the body is equal to or longer than specified, only write the number of bytes specified in the header and ignore the rest. If the body is shorter than specified, write the bytes in the body.

Note that the body may contain any kind of character, so `memcpy` might be a better idea than functions that only process null-terminated strings.

2.6 Error and Limit Handling in POST (Test 06)

There are certain limits on what the client can `POST` and these tests ensure your server follows these restrictions. Remember that your server should only consider the number of bytes mentioned in the `Content-Length` header for `POST`ing, not simply all of the body bytes in the request. The client is not allowed to `POST` more than `1024` bytes of data and if they do then send error `413`.

Request Entity Too Large

```
HTTP/1.1•413•Request•Entity•Too•Large▽
▽
⊗
```

2.7 Common GET Requests (Test 07)

The server should have the ability to send regular files to the client upon request. We will refer to this as the common GET request and occurs when the client makes a request `GET /<PATH>` and `PATH` does not correspond to `read`. Below is an example of the client requesting a file called `index.html` which contains 469 bytes.

```
GET•/index.html•HTTP/1.1▽
<HEADERS>▽
▽
⊗
```

```
HTTP/1.1•200•OK▽
Content-Length:•469▽
▽
<FILE-CONTENTS>
```

2.8 Error Handling (Test 08)

There is one final test group that mostly checks Common GET but does have some additional miscellaneous checks, which we describe below.

- If the pathname is not valid or the path leads to a non-regular file (for example it leads to a directory), then the server should respond with `404 Not Found`. Note that `open(path)` will succeed even if `path` is a directory. You will find the function `fstat` useful for determining the size of the file and if the file path points to a regular file.
- The validity of the `METHOD URL` pair in the request line should be checked. For example, if the client's request is `POST /read` that should be considered an invalid request and handled with `400 Bad Request`. The format for those errors are below.

Bad Request

```
HTTP/1.1•400•Bad•Request▽
▽
⊗
```

Resource Not Found

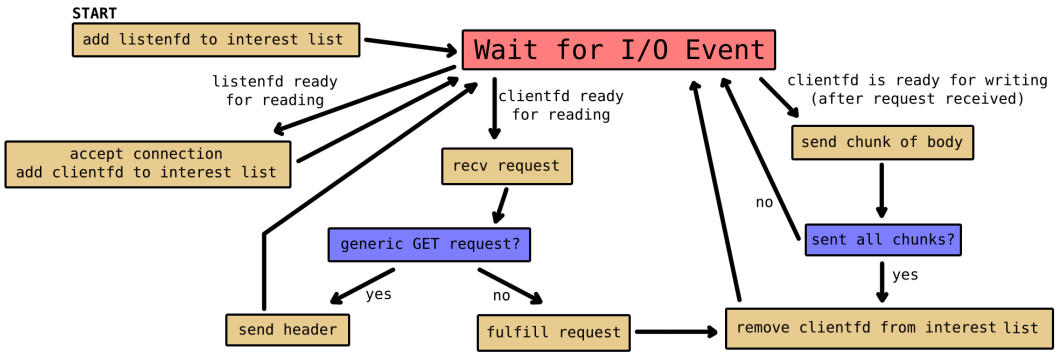
```
HTTP/1.1•404•Not•Found▽
▽
⊗
```

2.9 Concurrently Handling Common GET Requests (Tests 09 and 10)

You must use Epoll, which we will discuss in class. Select for I/O Multiplexing, fork, and threads are forbidden. If you submit work using anything other than Epoll I/O Multiplexing for handling multiple clients I will not accept your work. Test groups 09 and 10 focus on your server's ability to handle multiple clients concurrently. They are set up so that multiple clients will request to download a large file at the same time and if you only handle the requests in sequence then these tests will fail. To pass these tests your server must be able to manage multiple downloads simultaneously.

For example let's say there are two clients, where client A will request a 100 MB file and client B is requesting a 1 MB file. If client A makes the request followed by client B and you are handling the download requests in sequence then client B could have to wait a long time for client A's download to finish before receiving it's small 1 MB file.

From a design perspective, you will employ I/O multiplexing which means your server will schedule logical flows based on I/O events. We present a high level diagram below, of an I/O event driven server, to aid you in this process.



You should start by adding the listening socket to the interest list before accepting any connections. The listening socket should be registered for reading. If the listening socket is available for reading then that means there is a connection request and you can call `accept` without blocking the server.

You should first register client sockets for reading after accepting the connection. When it becomes available for reading then that means the client has sent their request and you can call `recv` without blocking. Make sure to check the return values of `epoll_ctl` and `epoll_wait` for errors!

Hint: Send chunk sizes of `1024` bytes. Hint: Normally, if the client makes a common GET request then your server will open the file and it's contents are the body of the response. This

corresponds to Section 2.7. If you are implementing the server using the diagram above then you will need to design a data structure that maps client socket file descriptor to the regular file descriptor your server is actively sending to the client to successfully handle the third branch (clientfd is ready for writing).

3 Due Date and Resubmission Policy

- The deadline for this homework is 11/20 11:59 PM. There is no late policy.
- Your submission is the last commit you made in your repository. We will use the timestamp of this commit to verify the assignment was completed on time.
- Assuming you have attended **Lab 07** and **Lab 08** you can resubmit your work up to one week after the deadline, 11/27 11:59 PM. You must also submit a written document that describes the changes you made since the deadline. The written document is how you can demonstrate your understanding of what was missing in your original submission.

4 Bonus Points

- You can earn up to 10% bonus points from endorsed posts you make on Piazza.
- You can earn 10% bonus points for completing Lab 07 and 08

5 Hardcoding and Academic Integrity

Just as a reminder, the syllabus policy on academic integrity will be enforced. Also, if we discover that you are subverting the goal of the assignment and gaming the autograder to simply earn points is a zero on the assignment. If your code is trying to detect that a certain input is given (because you know it is associated with a specific test) and simply output an expected result (without doing the reasonable work required by the assignment) then that is gaming the autograder. If you have further questions on what would be considered "gaming" then post them on Piazza as soon as you have them.