Exercise 14

HTTP Client

For this exercise, you will be writing a simple program named HTTPclient.c. It will be able to use TCP sockets to make an (unencrypted) HTTP request to get a web page or make a REST call to a server on the Internet. REST is a technique for making a remote procedure call (like RPC) using the same protocol we use to retrieve web pages.

HTTP is a good example of an application-layer protocol. It's implemented inside the process (i.e., not by the operating system). It takes the semantics offered by a transport-layer protocol (e.g., TCP) and adds additional code, control information and policies to create something that's suitable for accomplishing a particular task.

We'll be using unencrypted, HTTP in our implementation. This makes it easy to write our own code to create a simple request, send it to a server and read back the response. In practice, it's much more common to use encrypted HTTPS. Writing our own code for HTTPS would be a much bigger job, far too much for a little programming exercise. This means our client will only be able to communicate with servers that still offer unencrypted HTTP. This exercise provides a test server that uses HTTP, and there are still some servers like that out on the Internet.

Our HTTPclient.c program will expect three command-line arguments.

- The first argument will be the name of a host computer that we'll be sending the request to. While you're developing your solution, you will probably use localhost to connect to a test server running on the same system. Once you have your client working, you should be able to connect to any (unencrypted) HTTP server on the Internet.
- The second argument will be the port number the server is listening on. While you're developing, you will run a test server as the same, randomly-assigned port number you used in homework 5. So, you'll want to give the client this same port number. Once your client is working, you can connect to any (unencrypted) HTTP server on the Internet, so you'll want to use whatever port number (probably 80) that server is running on.
- The third command-line argument will be a path for the URL for our request. It lets us access different resources on the host by providing different paths.

These three fields would normally be packed together in the URL, like the following.

```
http://hostname:port/path
```

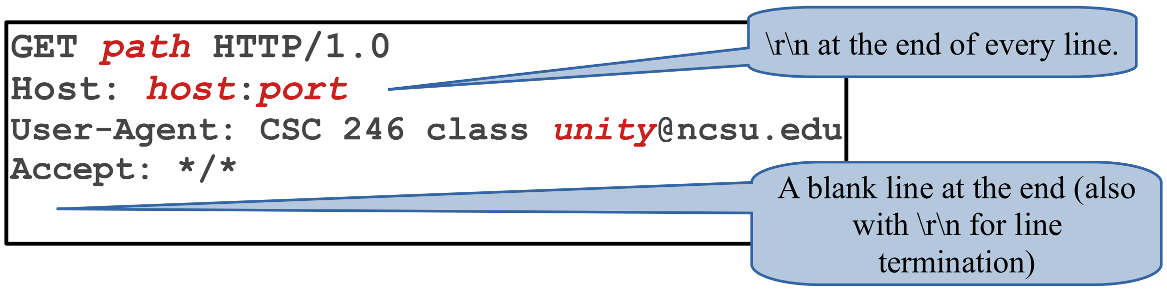
Breaking up these three fields as separate command-line arguments will simplify our client program a little bit. It won't have to worry about parsing these three fields out of a single URL string.

```
$ ./HTTPclient hosname port path
```

Making a HTTP Request

The starter file, HTTPclient.c already has code to establish a TCP connection to the given host and port number. You'll need to write the code to build a HTTP request and send it to the server. An unencrypted HTTP request is represented as text. It can have a header and a body, but we'll just need the header part for our client. The body will be empty, so your client can just send the header to the server, with nothing after it.

For the header, you need to build a string containing the following text. The red, italic text indicates the parts of the header you'll need to modify. The path, host and port are the values of the command-line arguments. You can just copy the text of these arguments into these parts of the header. For the User-Agent line, we'll use a string indicating that we're developing a client for a class, and we'll include contact information for the developer (you) in case something goes wrong. Replace the unity in the header with your own unity ID.



```
GET path HTTP/1.0
Host: host:port
User-Agent: CSC 246 class unity@ncsu.edu
Accept: */*

```

\r\n at the end of every line.
A blank line at the end (also with \r\n for line termination)

The HTTP header is expected to have windows-style line termination at the end of every line. Every line should end with a carriage-return, linefeed sequence (\r\n). You'll also need a blank line at the end of the header (also ending with carriage-return, linefeed). The blank line indicates the end of the header.

You can prepare this header as a string, then send it to server by writing the contents of the string to the socket. The standard library function, snprintf() would be useful in preparing the header as a string, but you can create it however you want.

Handling the HTTP Response

The response to a HTTP request will include a header at the start, followed by a body. Just like in the request, the header will be lines of text ending with carriage-return linefeed, with a blank line marking the end of the header. The body will be anything received after the line termination at the end of the blank line.

Once it has finished sending the request, your client can read the response from the socket connection. Use the read() system call for this. The response may not be available all in one call to read(), so your client should keep reading successive blocks of data from the socket until it gets a return of zero from read() (indicating end-of-file) or a return of -1 (indicating an error).

As your client reads the server's response, it should print it out to the terminal. Print bytes from the header to standard error, and print bytes from the body to standard output. This means your client will need code to detect the blank line at the end of the header. That's when you'll need to switch from printing to standard error to printing to standard output. Remember that lines of the header, including the blank line at the end, are terminated by carriage-return, linefeed (\r\c).

Testing your Client

You should be able to build your client with the following compile command. The gnu99 standard (rather than c99) should let you use the getaddrinfo() function to do name resolution.

```
gcc -Wall -std=gnu99 -g HTTPclient.c -o HTTPclient
```

We've provided a Java test server to help you test your client during development. It's called TestServer.java. You should be able to compile it with javac, like a typical Java program:

```
javac TestServer.java
```

If you develop your client on a university machine, it's possible other students will be using the same system at the same time. So, when you run the test server, we need to make sure you don't try to use the same port number as another student. We'll use the same randomly-assigned port numbers we used for assignment 5. If you click on the "Port Number Allocation" assignment, you can download a feedback file that tells you what port number you should use. Be sure to use that for your test server so you don't interfere with other students who might be developing on the same system.

When you run the test server, it expects the port number as a command-line argument. In the example execution below, I'm running the test server with a port number of 99999. That's larger than any legal, 16-bit port number, so it will fail if you try it. Just replace this number with your assigned port number from the "Port Number Allocation" assignment.

```
java TestServer 99999
```

The test server will continue running until you kill it with ctrl-C. For each request, it sends back a reply containing the path in the request, the address the request came from and a count of the total number of requests it has processed so far.

In another terminal on the same host, you can use a command like the following to run your client program and tell it to connect to your test server. Replace the 99999 with the port number your server is using. Also, be sure your client and server are running on the **same host** or the **localhost** name won't work for connecting to your server.

The following shows the output you should expect, depending on when you actually make the request and the number of previous requests you may have made.

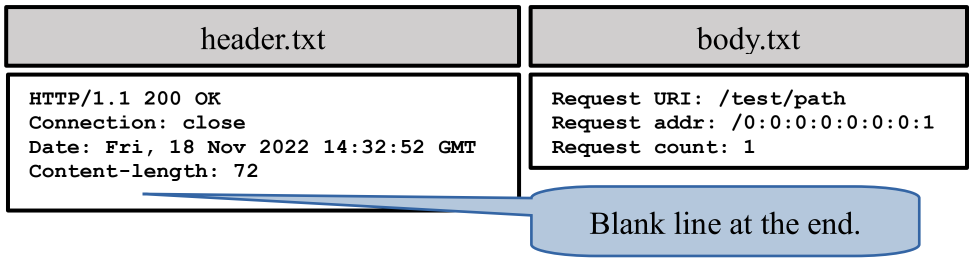
```
$ ./HTTPclient localhost 99999 /test/path
HTTP/1.1 200 OK
Connection: close
Date: Fri, 18 Nov 2022 14:32:52 GMT
Content-length: 72

Request URI: /test/path
Request addr: /0:0:0:0:0:0:0:1
Request count: 1
```

You can run your client as follows, to capture the header and the body of the response to different files. This will help you check to make sure the right part of the server's response is going to standard error vs standard output.

```
./HTTPclient localhost 99999 /test/path 2> header.txt > body.txt
```

If you run your client like this, here's what you should see in each of these files, depending on the time of the request and previous requests processed by your server. Header information went to the header.txt file and the body went to body.txt.


header.txt

```
HTTP/1.1 200 OK
Connection: close
Date: Fri, 18 Nov 2022 14:32:52 GMT
Content-length: 72

```

body.txt

```
Request URI: /test/path
Request addr: /0:0:0:0:0:0:0:1
Request count: 1
```

Blank line at the end.

Fun with your HTTPclient

Once your HTTPclient is working, you should be able to use it to connect to web servers or REST services that still offer unencrypted, HTTP connections. These aren't as common as they used to be, but, as when this assignment is being written (November 2022), you should be able to try out the following.

For each of the following URLs, you will need to break it up into individual fields to make a request with your client. The default port number for HTTP service is 80, so you can just use that for any of these examples.

- Bored API
  This is a REST interface for getting a suggested activity, if you're bored. The following will suggest a random activity.
  http://www.boredapi.com/api/activity
- Numbers API
  This is a REST interface for getting information about numbers. The following will give you a trivia fact about a randomly selected number.
  http://numbersapi.com/random/trivia
- Excuser API
  This is a REST interface for getting a random excuse you can use to get out of something.
  http://excuser.herokuapp.com/v1/excuse

Submitting your Solution

When your code is working, submit your completed client, **HTTPclient.c** to the Gradescope assignment named EX14. To grade the assignment, we'll run your client on a test server like the one given out with the assignment, although it may not be running on the same host at the client.