MP 1: HTTP Client + Server

Due: Friday, Feb 9th, 11:59pm

Please read all sections of this document before you begin coding.

In this assignment, you will implement a simple HTTP client and server. The client should be able to GET correctly from standard web servers, and browsers should be able to GET correctly from your server.

The test setup will be two VMs, one server and one client. Each test will use your client or wget, and your server or thttpd. You don't have to support caching or recursively retrieving embedded objects.

HTTP uses TCP – you can use Beej's client.c and server.c as a base.

HTTP Client

The Basics

Your code should compile to generate an executable named ***http_client***, with the usage specified as follows:

```
./http_client http://hostname[:port]/path/to/file
```

For example:

```
./http_client http://illinois.edu/index.html
./http_client http://localhost:5678/somedir/somefile.html
```

If *:port* is not specified, default to ***port 80*** – the standard port for HTTP.

Your client should send an **HTTP GET request** based on the ***first argument*** it receives. Your client should then write the **message body** of the response received from the server to a file named ***output***. Your client should also be able to handle redirections as described in detail later.

Details on HTTP GET Request

Here's the very simple HTTP GET request generated by *wget*:

```
GET /test.txt HTTP/1.0
User-Agent: Wget/1.15 (linux-gnu)
Accept: */*
Host: localhost:3490
Connection: Keep-Alive
```

- `GET /test.txt` instructs the server to return the file called `test.txt` in the server's top-level web directory.
- `User-Agent` identifies the type of client.
- `Accept` specifies what types of files are desired – the client could say "I only want audio", or "I want text, and I prefer html text", etc. In this case it is saying "anything is fine".
- `Host` is the URL that the client was originally told to get from – exactly what the user typed. This is useful in case a single server has multiple domain names resolving to it (maybe www.cs.illinois.edu and www.math.illinois.edu), and each domain name actually refers to different content. This could be a bare IP address, if that's what the user had typed. The 3490 is the `port` – this server was listening on 3490, so I called "`wget localhost:3490/test.txt`".
- `Connection: Keep-Alive` refers to TCP connection reuse, which will be discussed in class.

Note that the newlines are technically supposed to be CRLF - "**\r\n**". Only the first line is essential for a server to know what file to give back, so your HTTP GETs can be just that first line. HTTP specifies that the end of a request should be a marked by a blank line – so be sure to have two newlines at the end. (This demarcation is necessary because TCP present you a stream of bytes, rather than packets.) You may use either HTTP 1.0 or 1.1. However,

notice that if you are using HTTP/1.0, the Host header is NOT REQUIRED. If you are using HTTP/1.1, the Host header is REQUIRED. But be aware that the Host header may still be necessary for some URLs even in HTTP/1.0.

Handling Redirections

"The HTTP response status code **301 Moved Permanently** is used for permanent URL redirection, meaning current links or records using the URL that the response is received for should be updated. The new URL should be provided in the Location field included with the response." – Wikipedia page on HTTP 301.

In some cases, the server may send back a status code 301, indicating that the content requested has been moved:

```
HTTP/1.1 301 Moved Permanently
Location: <new_url>
```

In this case the client should attempt to retrieve the document in the ***new_url*** given in the Location field.

Here is an example from Wikipedia page:

Client request:

```
GET /index.php HTTP/1.1
Host: www.example.org
```

Server response:

```
HTTP/1.1 301 Moved Permanently
Location: http://www.example.org/index.asp
```

HTTP Server

The Basics

Your code should compile to generate an executable named ***http_server***. It should take one command line argument, the port number, and start listening on the port specified once started.

The server should handle HTTP GET requests by sending back HTTP responses, as described in detail below.

Your server program should treat all filepaths it's asked for as being relative to its current working directory. (Meaning just pass the client's request directly to `fopen`: if the client asks for somedir/somefile.txt, the correct argument to `fopen` is "somedir/somefile.txt").

Warning: running this http_server program essentially makes all files accessible by the http_server process accessible by anyone who can send a request to the host. Discretion is advised (Only run it on a VM)!

Usage examples:

```
sudo ./http_server 80
./http_server 8888
```

(The sudo is there because using any port <1024 requires root access.)

Details of a HTTP response

Here's what Google returns for a simple GET of /index.html:

```
HTTP/1.0 200 OK
Date: Wed, 21 May 2014 17:39:46 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=d985d415c1aaf0cc:FF=0:TM=1400693986:LM=1400693986:S=jBoFRLMuiYWpB6sl;
expires=Fri, 20-May-2016 17:39:46 GMT; path=/; domain=.google.com
Set-Cookie: NID=67=UEN1ApahELM_UhkJDWgHbwLmw1thhjwfucoYpC2E-
UpqH6bwR8Rq9YAqY1ptRu3qCeIjkHLBwY867JmRn4fzFQzJpgId1TLzXhBhLjAKCpGx0DQpVSDFjAPByCQo37
K4; expires=Thu, 20-Nov-2014 17:39:46 GMT; path=/; domain=.google.com; HttpOnly
P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts/bin/answer.py?
hl=en&answer=151657 for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic

<!doctype html><html itemscope="" much more of the document follows...
```

Another example:

```
HTTP/1.1 200 OK
Date: Sun, 10 Oct 2010 23:26:07 GMT
Server: Apache/2.2.8 (Ubuntu) mod_ssl/2.2.8 OpenSSL/0.9.8g
Last-Modified: Sun, 26 Sep 2010 22:04:35 GMT
ETag: "45b6-834-49130cc1182c0"
Accept-Ranges: bytes
Content-Length: 13
Connection: close
Content-Type: text/html

Hello world!
```

Your server's headers can be much simpler (but still correct and complete): just the status line (the first line of the header, contains the status code).

- When correctly returning the requested document, use "**HTTP/1.0 200 OK**".
- When the client requests a non-existent file, use "**HTTP/1.0 404 Not Found**". Note that you can still have document text on a 404, allowing for nicely formatted/more informative message such as "whoops, file not found!"
- For any other errors, use "**HTTP/1.0 400 Bad Request**".

Per the HTTP standard, again an empty line (two CRLF, "\r\n\r\n") marks the end of the header, and the start of message body.

To summarize, your whole response should be: **header + empty line + message body (the file requested if available)**

Reading the Wikipedia page HTTP message body may be helpful.

Handling Concurrent Connections
Your server must support concurrent connections: if one client is downloading a 10MB object, another client that comes looking for a 10KB object shouldn't have to wait for the first to finish.

You can do so by handling each connection in a new thread.

Grade Breakdown

10%: you submitted your assignment correctly
10%: it compiles correctly on the auto-grader
5%: your client can retrieve http://illinois.edu/index.html.
5%: **Use the (SO_REUSEADDR | SO_REUSEPORT) socket option**
15%: wget can retrieve files from your HTTP server
10%: your server can handle invalid client request and request for non-existing files properly
20%: your client can retrieve files from your HTTP server
15%: multiple clients can download files from your server simultaneously (concurrency)
10%: client can handle redirections correctly

Notes

1. You must use C or C++. Your program must have a Makefile; running "make" should build all executables.
2. You must work alone.
3. If you use Git, do not check the .git metadata into the SVN repo: 5% penalty. Do not use a public GitHub repo.
4. You will be held partially responsible for any resultant plagiarism.
5. Your code must be your own. You can discuss very general concepts with others, but if you find yourself looking at a screen/whiteboard full of pseudo code (let alone real code), you are going too far.
    a) Refer to the class slides and official student handbook for academic integrity policy. In summary, the standard for guilt is "more probable than nor probable", and penalties range from warning to recommending suspension/expulsion, based entirely on the instructor's impression of the situation.
    b) The College of Engineering has some guidelines for penalties that we think are reasonable, but we reserve the right to ignore them when appropriate.
    c) If you need to use a library for data structures, you MUST get the approval of the course staff. Additionally, you MUST acknowledge the source in a README. However, algorithms MUST be your own.
6. Your code must run on the test setup, which is just some Ubuntu 14.4 VMs, running on VirtualBox. We will not look at your program on your laptop. Late penalty: 2% of total possible score per hour.
7. The auto-grader will test every student's current version every night, starting at 6PM. We will make an announcement of its starting date.
    a) Do an SVN update the next morning to see your *result.txt*.
    b) The score you are given is hypothetical and has no effect on your final grade.
    c) The late penalty is applied to the last SVN commit you make. If you want to make a late commit that doesn't change your code, like to a README file, please wait until 50 hours after the deadline to make it.

Appendix –VM Setup

Now that you are responsible for both a client and server component, you'll need 2 VMs for thorough testing. Unfortunately, VirtualBox's default setup does not allow its VMs to talk to the host or each other. There is a simple fix, but then that prevents them from talking to the internet. So, be sure you have done all of your apt-get installs before doing the following! To be sure, just run:

```
sudo apt-get install gcc make gdb valgrind libpolarssl5 libpolarssl-dev
iperf tcpdump
```

Make sure the VMs are fully shut down. Go to each of their Settings menus, and go to the Network section. Switch the Adapter Type from NAT to "host-only", and click ok. When you start them, you should be able to ssh to them from the host, and it should be able to ping the other VM.

You can use `ifconfig` to find out the VMs' IP addresses. If they both get the same address, `sudo ifconfig eth0 newipaddr` will change it. (If you make the 2nd VM by cloning the first + choosing reinitialize MAC address, that should give different addresses.)