Project 1 should be done in groups of two or individually.

You should do the Wireshark labs in Homework 1 before attempting this project.

This assignment was originally developed by Prof. Peter Dinda, but I have modified it in several ways.

Overview

In this part of the project, your group will build a simple web client and a web server to which it can connect. The goal is to slowly introduce you to Unix socket programming, and the HTTP protocol, which is the most important network application protocol on the Internet. You should develop the client and server in Python.

Notice that I am not asking you to build a web **page** or **application**, but a web **server**. Website developers normally deploy their pages/apps on an existing web server framework such as Apache, Node.js, Django, Tomcat, Microsoft IIS, etc. You'll be building such a web server from scratch, and in the process you will learn about HTTP and low-level network programming.

Coding

Your Python code must use the low-level socket API for networking and you must use the version of Python that is installed on the test machine (version 3.6). (More recent versions are probably also OK, but be sure to test on moore.)

You must test your code on moore.wot.eecs.northwestern.edu before submitting.

HTTP and HTML

The combination of HTTP, the Hypertext Transport Protocol, and HTML, the Hypertext Markup Language, forms the basis for the World Wide Web. HTTP provides a standard way for a client to request *typed* content from a server, and for a server to return such data to the client. "Typed content" simply means a bunch of bytes annotated with information that tells us how we should interpret them (a MIME type). For example, the MIME type "text/html" tells us that the bytes are HTML text and the MIME type "image/jpeg" tells us that the bytes represent an image in JPEG format. You will implement a greatly simplified version of HTTP 1.0.

HTML content provides a standard way to encode structured text that can contain pointers to other typed content. A web browser parses an HTML page, fetches all the content it refers to, and then renders the page and the additional embedded content appropriately.

HTTP Example

In this project, you will only implement HTTP, and only a tiny subset of HTTP 1.0 at that. HTTP was originally a very simple, but very inefficient protocol. As a result of fixing its efficiency problems, modern HTTP is considerably more complicated. It’s current specification, RFC 2616, is over a hundred pages long! Fortunately, for the purposes of this project, we can ignore most of the specification and implement a tiny subset. In fact, it may be easier to refer to the older, simpler version of the HTTP spec for this assignment, RFC 1945.

The HTTP protocol works on top of TCP, a reliable, stream-oriented transport protocol, and it uses human-readable messages. Because of these two facts, we can use the telnet program to investigate how HTTP works. We’ll use telnet in the role of the client and insecure.stevetarzia.com in the role of the server. This is essentially the same as fetching http://insecure.stevetarzia.com/basic.html using your favorite web browser.

> *Note*: Most webservers nowadays require TLS encryption (HTTP**S**) on the connection. That's a good thing, but it would add a lot of complexity to this assignment. We are testing with "insecure.stevetarzia.com" because that is a webserver that I configured specially to **not** require encryption.

```
[spt175@moore ~]$ telnet insecure.stevetarzia.com 80
Trying 54.245.121.172...
Connected to stevetarzia.com.
Escape character is '^]'.
GET /basic.html HTTP/1.0
Host: insecure.stevetarzia.com

HTTP/1.1 200 OK
Date: Wed, 08 Jan 2020 13:10:56 GMT
Server: Apache/2.2.34 (Amazon)
Last-Modified: Tue, 07 Jan 2020 23:59:54 GMT
ETag: "2770e-65-59b959285347f"
Accept-Ranges: bytes
Content-Length: 101
Connection: close
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>
<p>My first paragraph.</p>

</body>
</html>
Connection closed by foreign host.
[spt175@moore ~]$
```

Above, I just ran the command "telnet insecure.stevetarzia.com 80" then I typed "GET /basic.html HTTP/1.0" pressed return, typed "Host: insecure.stevetarzia.com" and pressed "return" **twice** to send a two-line HTTP request to the server. The telnet command just creates a TCP connection to the specified server and port (port 80 is the standard HTTP port), and then it lets the client and server send text back and forth to each other. I specified the address with a DNS name, "www.mccormick.northwestern.edu", so the telnet program actually has to first do a DNS lookup to determine that this corresponds to IP address 129.105.1.129.

Our request above said “please give me the data at location called insecure.stevetarzia.com/basic.html using the 1.0 version of the HTTP protocol.” HTTP requests can be several lines long, so the blank line at the end indicates that the request is finished. The TCP connection we're using provides a *stream* of data, so the server will probably receive the request in several pieces. Once it sees the blank line, it knows that it has the full request and it can proceed in handling it.

The response always begins with a line that states the version of the protocol that the server speaks (“HTTP/1.1” in this case), a response code (“200”), and a textual description of what that error code means (“OK”). If we had specified an invalid url, like "GET /blargh HTTP/1.0" then the server would has responded with a "404 Not Found" response code. (Please try it out!) After the status line, the server provides a bunch of information about the content it is about the send as well as information about itself and what kinds of services it can provide. The most critical lines here are “Content-Length: 101”, which tells us that the content will consist of 101 bytes, and “Content-Type: text/html”, which tells us how to interpret the content we shall receive. A blank line marks the end of the response header and the beginning of the actual content (the body). After the content has been sent, this server has chosen to close the connection (although it's not required).

> *Note*: The "Content-Length: 101" response header is important because there is no clear way for the client to know when the received data is finished. This is especially true if the server is planning to leave the connection open so the client can make another HTTP request. The HTTP 1.0 protocol does allow the server to omit the Content-Length header, but only if it closes the connection when the transfer is completed.

Part 1: a simple curl clone

This part is worth 30% of the total grade, if completed correctly.

You will implement a simple command-line HTTP client, implemented using the BSD socket interface. Throughout all the parts of this assignment, you must use Python's "socket" package and no other (higher-level) network library. For example, you **cannot** use "urllib" even though it's part of the standard Python library.

Like a simplified version of the unix "curl" command, your program should take just one parameter, which is an http web address to fetch. If successful, if should print the body of the response (the html code) to stdout.

For example, you might invoke it as follows:

```
$ python3 http_client.py http://somewebsite.com/path/page.htm
```

And it would print out the HTML for that page.

Additionally, you must follow these rules:

- Only the **body** of the response (excluding the header) should be printed to **stdout**. If you want to print any other messages, they should be printed to **stderr**.
- All requests are assumed to use the HTTP "GET" method (you do not have to support POST or anything else).
- Your client must include a "Host: ..." header. This is necessary because some web servers handle multiple domains.
- Your program should return a Unix exit code (using **sys.exit**) to indicate whether the request is successful or not. It should return a Unix exit code of 0 on success (meaning that you eventually get a "200 OK" response with valid HTML) and non-zero on failure.
- Your client should understand and follow 301 and 302 redirects. When you get a redirect, your client should make another request to fetch the corrected url. Your client should also print a message to **stderr** with the format: "Redirected to: http://other.com/blah" *(The specific url should be shown)*.

  - An example of a url with a 301 permanent redirect is:
    http://airbedandbreakfast.com/ (which redirects to https://www.airbnb.com/belong-anywhere)
  - An example of a url with a 302 temporary redirect is:
    http://maps.google.com/ (which redirects to http://maps.google.com/maps, which redirects to https://www.google.com:443/maps)
  - Another example is:
    http://insecure.stevetarzia.com/redirect which redirects to http://insecure.stevetarzia.com/basic.html Unlike the google and airbnb links above, the final destination is not encrypted with HTTPS, so your program should be able to print the HTML without any fancy cryptography code.
  - You must handle a chain of multiple redirects (like the above), but give up after 10 redirects and return a non-zero exit code. For example, fetching the following page should not loop infinitely, but return a non-zero exit code:
    http://insecure.stevetarzia.com/redirect-hell
- If you try to visit a https page (note the added "s" for "secure" which means that it requiring encryption) or if you are redirected to an https page, just print an error message to **stderr** and return a non-zero exit code.
- If you get an HTTP response code >= 400, then you should return a non-zero exit code, but also print the response body to **stdout**, if any.
  - For example, the following page returns a 404 response:
    http://cs.northwestern.edu/340
- Check the response's content-type header. Print the body to stdout only if the content-type begins with "text/html". Otherwise, exit with a non-zero exit code.
- Return a non-zero exit code if the input url does not start with "http://"
- Allow request urls to include a port number.
  - For example: http://portquiz.net:8080/
- You should not require a slash at the end of top-level urls. For example, both of the following urls should work: http://insecure.stevetarzia.com and http://insecure.stevetarzia.com/
- You should be able to handle large pages, such as: http://insecure.stevetarzia.com/libc.html
- Your client should run quickly. In other words, it should not use timeouts to determine when the response it fully transferred.
- Your client should work even if the Content-Length header is missing. In this case, you should just read body data until the server closes the connection. This behavior is part of the HTTP/1.0 spec, so some web servers will do this. For example, your client should work with http://google.com.

Part 2: a simple web server

This part is worth 30% of the total grade, if completed correctly.

As above, you must implement this using low-level BSD sockets in Python.

Write an HTTP server that handles one connection at a time and that serves any files in the current directory if their name ends with .html or .htm. To prepare for this assignment, download the sample html file rfc2616.html and save it to your working directory. The command-line interface for your server will be

```
$ python3 http_server1.py [port]
```

Where "port" is a port number >= 1024. You will then be able to use http_client, telnet, curl, or any web browser, to fetch files from your server. However, please be aware that there is a firewall on moore (the test machine) that blocks network access from the outside world. If, for example, you run your server on moore.wot.eecs.northwestern.edu on port 10002, then you can test your server by running the following command on another SSH session on moore:

```
$ curl http://moore.wot.eecs.northwestern.edu:10002/rfc2616.html
```

It is important to note that you will not be able to use port 80. Port numbers less than 1024 are reserved, and you need special permissions to bind to them. The firewall will block access to moore's high ports if you are not on the Northwestern wired campus network.

Your server should have the following structure:

1. Create a TCP socket on which to listen for new connections. (What packet family and type should you use?)
2. Bind that socket to the port provided on the command line. We'll call this socket the "accept socket." Please listen on all IP addresses on the machine by choosing "" as the listening address.
3. Listen on the accept socket. (What will happen if you use a small backlog versus a larger backlog? What if you set the backlog to zero?)
4. **Do the following repeatedly:**

   a. Accept a new connection on the accept socket. (When does accept return? Is your process consuming cycles while it is in accept?) Accept will return a new socket for the connection. We'll call this new socket the "connection socket." (What is the 5-tuple describing the connection?)

   b. Read the HTTP request from the connection socket and parse it. (How do you know how many bytes to read?)

   c. Check to see if the requested file requested exists (and ends with ".htm" or ".html").

   d. If the file exists, construct the appropriate HTTP response (What's the
   right response code?), write the HTTP header to the connection socket, and then open the file and write its contents to the connection socket (thus writing the HTTP body).

   e. If the file doesn't exist, construct a HTTP error response (404 Not Found) and write it to the connection socket. If the file does exist, but does not end with ".htm" or "html", then write a "403 Forbidden" error response.

   f. Close the connection socket.

TIP: If you are running your server code locally (on your laptop), then you can use the special hostname "localhost" to refer to your own machine. For example, direct your browser to "http://localhost:8000/index.html".

To test your server, you should make sure that a client can correctly fetch the HTML content. Run these commands in your working directory:

```
$ curl http://[hostname]:[port]/rfc2616.html > copy.html # saves the output to copy.html
$ diff copy.html rfc2616.html # this should print nothing if the two files have no difference
```

Also, try to load a page with a standard browser, like Chrome. If you have done everything correctly, Chrome will render your version of the page exactly as we see it on the IETF's website: https://tools.ietf.org/html/rfc2616.

Part 3: multi-connection web server

This part is worth 30% of the total grade, if completed correctly

The server you wrote for part 2 can handle only one connection at a time. To illustrate the problem, try the following. Open a telnet connection to your http_server1 and type nothing. Now make a request to your server using a browser. What happens? If the connection request is refused, try increasing the backlog you specified for listen in http_server1 and then try again. After http_server1 accepts a connection, it blocks (stalls) while reading the request and thus is unable to accept another connection. Connection requests that arrive during this time are either queued, if the listen queue (whose size you specified using listen) is not full, or refused, if it is.

Consider what happens if the current connection is very slow, for example if it is running over a weak cellular signal. Your server is spending most of its time idle waiting for this slow connection while other connection requests are being queued or refused. Reading the request is only one place where http_server1 can block. It can also block on waiting for a new connection, on reading data from a file, and on writing that data to the socket.

Write an HTTP server, http_server2, that avoids two of these situations: (1) waiting for a connection to be established, and (2) waiting on the read after a connection has been established. You can make the following assumptions:

- If you can read one byte from the socket without blocking, you can read the whole request without blocking.

- Reads on the file will never block

- Writes will never block

It is important to note that if you have no connections with new data available and no new connections just established, then you should block.

To support multiple connections at a time in http_server2, you will need to do two things:

- Explicitly maintain the state of each open connection

- Block on multiple sockets, file descriptors, events, etc.

It is up to you to decide what the contents of the state of a connection are and how you will maintain them. However, Unix, as well as most other operating systems, provides a mechanism for waiting on multiple events. The Unix mechanism is the ***select*** system call. If you have not taken an Operating Systems course, then the term "system call" may be unfamiliar. System calls are special functions that allows a process to ask the OS to do something on its behalf. For the purposes of this assignment, just think of "select" as a library function, like "printf".

"*Select*" allows us to wait for one or more file descriptors (a socket is a kind of file descriptor) to become available for reading (so that at least one byte can be read without blocking), writing (so that at least one byte can be written without blocking), or to have an exceptional condition happen (so that the error can be handled). In addition, *select* can also wait for a certain amount of time to pass. You can read about *select* here: https://pymotw.com/3/select/.

Your server should have the following structure:

1. Create a TCP socket on which to listen for new connections
2. Bind that socket to the port provided on the command line.
3. Listen on that socket, which we shall call the "accept socket."
4. Initialize the list of open connections to empty
5. Do the following **repeatedly:**

a. Make a list of the sockets we are waiting to read from the list of open connections. We shall call this the "read list." In our case, it's simply the list of all open connections.

Note: A more sophisticated solution (beyond the scope of this assignment), would also do the reading of files from disk in a non-blocking way using *select*. In that case, the read list would also include a list of files that we were in the process of reading data from. In this assignment we assume that file data can be read immediately, without an impact on performance, but that is not really true in practice.

b. Add the accept socket to the read list. Having a new connection arrive on this socket makes it available for reading, it's just that we use a strange kind of read, the *accept* call, to do the read.

c. Call *select* with the read list. Your program will now block until one of the sockets on the read list is ready to be read.

d. For each socket on the read list that *select* has marked readable, do the following:

i. If it is the accept socket, accept the new connection and add it to the list of open connections with the appropriate state

ii. If it some other socket, performs steps 4.b through 4.f from the description of http_server1 in Part 2. After closing the socket, delete it from the list of open connections.

Test your server using telnet and curl (or a web browser) as described above, to see whether it really handles two simultaneous connections.

Your server should also be robust. If a request is empty or does not start with "GET", your server should just ignore it.

Part 4: dynamic web server

This part is worth 10% of the total grade, if completed correctly

Implement an "http_server3.py" which implements a JSON-based API for simple multiplication.

The request url will encode a math problem to solve by using what are called "query parameters" in the url. Here's an example request:

GET /product?a=12&b=60&another=0.5

The response should have "Content-Type: application/json" and the response body should be a json object like:

```
{
    "operation": "product",
    "operands": [12, 60, 0.5],
    "result": 360
}
```

Additional notes:

- You should use the built-in json python library to generate the response.
- Notice that in this particular application the names of the query parameters are unimportant and are ignored (above, the parameters were named "a", "b", and "another").
- We will not be testing multiple simultaneous connections, so you can base the code on Part 2 if you wish.
- If a user requests a url other than "/product" you should return a *404 Not Found* status code.
- If there are no parameters to the "/product" request or if a parameter is not a number (eg., "GET /product" or "GET /product?a=blah") then you should return a *400 Bad Request* status code.
- Please treat the parameters as floating point numbers. The result can also be "inf" (infinity) or "-inf" if floating point overflow occurs. Notice that JSON requires us to specify "inf" as a string (with double quotes) whereas regular numbers should **not** have quotes.

Submission

- You should work in pairs. List the participants (names and netids) in a README.txt file.
- Please make just one submission for the pair.
- Your code must compile and run on moore.wot.eecs.northwestern.edu.
- Name your source files "http_client.py", "http_server1.py", etc.
- You may not use any outside libraries to in your code (do not "pip install" anything).
- Your submission should be a .tgz file (a gzipped tarball), including a README.txt, and the python sources. To create the archive, use a command like the following
    - tar -czvf project1_NETID1_NETID2.tgz README.txt *.py
    - Notice that I'm including both partners' netids in the filename.
- Remember that you should leave a comment explaining any bits of code that you copied from the Internet (just in case two groups copy the same code from the Internet, we won't think that you copied each other directly).
- After you make your tarball, copy it to a temporary folder, extract it, compile it, and test it again to make sure you included everything.

Sharing files with your partner using git

I recommend that you create a git repository for your code and use Bitbucket to share the repository with your partner. Each of you will be able to work on your own copy of the code, and periodically push updates to the central repository. I recommend Bitbucket instead of Github because private repositories are free on Bitbucket (and it would be a violation of our cheating policy to post your code on a public repository).