Theo Novak
CSE 150
6 March 2022

Questions

1. How did you decide which type of socket to use? Why?

   First, we needed to establish that we were using the TCP protocol because HTTP works using the TCP transport protocol. This determines the second argument of the socket() function, SOCK_STREAM, which denotes the socket type as TCP. The other field of importance denotes the version of IP addresses we should make our socket compatible with, IPv4 or IPv6. The current iteration of the internet uses IPv4 with 32-bit addresses, and has not yet been upgraded to IPv6 addressing. Therefore, the first argument of the socket() function is AF_INET for IPv4, rather than AF_INET6 for IPv6. Additionally, the addresses provided in the Project.pdf are 32-bit addresses (IPv4).

2. How did you choose the destination ports?

   According to documentation provided by IBM, the format of a url is:
   scheme://host:port/path?query
   When I processed the requested urls, I first validated and subsequently removed the *scheme://* portion of the url. Once that step is complete, the port number, if provided, will be between a colon and the first forward slash character. An additional criteria is that the colon character is required to appear before the first forward slash, otherwise it is not a valid port. This can be seen in the example url, where the 80 is a part of the path rather than specifying a port:
   http://www.google.com/somedoc1.html:80
   If no port is specified in the url, the default destination port is designated as port 80. This is the port that is most commonly used for TCP connections involving web servers.

3. What error handling cases did you implement?

   There were many error cases caught and treated in the MyCurl file. First, the command line was validated to make sure only one or two additional arguments were provided to the MyCurl file, otherwise it responds with usage messages. If the requested url contains *https*, the program responds by saying HTTPS connections aren't supported. If the url doesn't contain *http://*, the program terminates because the HTTP scheme must be provided with the requested url. When establishing the socket connection or receiving data from the web server, timeout exceptions are caught to avoid unnecessary idle time. When connecting the socket, catches for errno 8 (mac) and errno -2 (mininet) are implemented, which responds with "Could not resolve host". Also, errno 61 (mac) and errno (111) are used to respond with "Failed to connect to <hostname> port <port>: Connection refused". Once the connection is established and we are actively receiving from the web server, there is an error case that applies to some sites with port 443 that catches “Empty reply from server” to prevent unnecessary idling. After receiving the HTTP header, we analyze it to determine if the content is chunk-encoded, in which we terminate with the message “Chunk encoding is not supported”. Other error cases in this section catch more port 443 connections that respond with errno 54 (mac) and errno 104 (mininet), which responds with “Recv failure: Connection reset by peer”. For a small percentage of sites, I encountered an error where the content length field in the HTTP header was not an integer and I had to catch an exception and print “content length is undefined”. The most common error that I had to catch on several occasions, which is stated earlier, was timeout exceptions.

4. How does your program terminate? What happens to the TCP connection?

   The program in every scenario, success or error, ends with the closing of the socket and a call to sys.exit(). In most of the tests, the TCP connection is able to reach its concluding [FIN, ACK] that denotes the end of the connection. An exception to this is during our exception for chunk-encoded HTTP responses, in which we close the socket and exit after the first call to socket.recv(). We are able to do this because the HTTP header is contained in this first HTTP response, from which we can extract the “Transfer-Encoding” field. By terminating before receiving all of the response data, the TCP connection does not reach its [FIN, ACK]. This could be altered to terminate after the entire response is retrieved, but this would be unnecessary computation considering the execution will result in a chunk-encoding error anyway. In addition, because *curl* doesn’t have issues with chunk-encoded sites, there isn't a *curl* Wireshark capture to mimic with our MyCurl program.

5. For the unsuccessful URLs, why were they unsuccessful?

   There were many different cases for unsuccessful URLs, beyond the obvious cases where a URL is malformed, the program arguments aren’t filled according to the restrictions, or the connection scheme is not HTTP. In some later tests of http://www.neverssl.com, though this was only for a period of about half of a day, I experienced failures because of an undefined content length, which is required to know when to stop reading the HTML content. Some URLs, like http://www.yahoo.com, fail with a status code 301, which denotes a redirect in which the site has been moved permanently. As stated in question 3, chunk-encoded URLs like http://www.google.com failed despite their 200 status code because the program isn’t built to handle chunk-encoding. Sites with port 443 failed because of that port’s nature as an encrypted HTTPS port using SSL, which were not accompanied by a status code. Another unsuccessful case was using http://wwwexample.com, which responds with status code 403, meaning that the connection to the site is forbidden. Other sites are unsuccessful because the page we are trying to access can’t be found (404), such as http://www.example.com/foo.html, which doesn’t have a *foo.html* page. Occasionally connections are unsuccessful with status code 400, meaning that a bad request was issued to the web server. This happened with http://www.neverssl.com:443.

6. What happens if you try to access a site using HTTPS?

   In my code, because I verify and then trim the scheme (such as HTTP or HTTPS) off of the requested url, the scheme does not make it into the creation of the socket connection. Instead, in the validate_arguments() function, if the requested url begins with the HTTPS scheme, an error is thrown informing the user that HTTPS connections are not supported, and an unsuccessful connection entry is appended to the log file. This entry only contains the unsuccessful field, the requested url, and the custom server response line, with the rest of the row attributes omitted.