Jessica Pan

jeypan@ucsc.edu

Faisal Nawab

Fall 2019 – Asgn1: HTTP Server

WRITEUP

The testing I performed on my program was unit testing and whole-system testing. As I build my program, I tested every small part that I'd finish programming. This way it allowed me to know that each small part of the program was functioning properly before I did a whole-system testing. After each portion of my program, I'd test for bugs and fix them, this process continued throughout the whole program. For example, I first set up the socket connection between the server and client without any IP address or port number specified. Once the connection worked and functioned properly, I made corrections to this code to take in specifically 2 arguments, one being the IP address the server will bind to and the optional port number. After implementing this, I tested if it worked and then moved onto the next implementation. Once I finished the implementation of the whole program, I performed whole-system testing on my program. Below I've listed the different whole system tests I've performed on my program and the reason of the test:

HTTP server

1. ./httpserver localhost
   - This was to test if the server would automatically bind to standard port 80 when a port number isn't specified. As a result, the server did connect to port 80, however, when performing this test, virtual box didn't allow the client to make the connection.
2. ./httpserver 8080
   - This was to test if the server would run with just a port number or whether the server would interpret this as an IP address. The server would not run, in this case.
3. ./httpserver localhost:8080
   - This was testing whether the server would understand the arguments used to run the server. As a result, the server wouldn't run, as the localhost and port number are connected by a colon. Also, how I implemented the reading of the arguments is that the IP address and port number would be spaced out by a ‘ ’.
4. ./httpserver localhost 8080
   - This is to test whether the server would run and wait for a connection with a client. As a result, if the client specifies the same IP address and port number, then the connection is established.

***With the server already running and listening for a connection (./httpserver localhost 8080) I will use curl(1) to make a connection and make/send GET/PUT requests to the server as a web client.

GET requests

1. curl 127.0.0.2:8080
   - Tests: I ran this to test to see if the client would connect to the server since the IP address the client is addressing is not the same IP address as the server. As a result, the client can't send requests to a server with a different IP address from what they had specified in their command with curl. And my HTTP server does reject this client that may have been trying to connect to the server.
2. curl 127.0.0.1:8888
   - Tests: I ran this to test if a client would be able to connect to the HTTP server when the client specifies the same IP address and a different port number. As a result, the client is unable to connect to the server, as the request sent from the client can only be received at the same IP address and the same port. This means the port in the server was unable to receive the request of the client, even though the client had reached the server.
3. curl 127.0.0.1:8080
   - Tests: I ran this to test if my server would be able to convert the "localhost" alias to 127.0.0.1 and make a connection with the client. Since the command used to run the server specified its IP address as "localhost", my server was able to convert the IP address and make a connection to the client that's trying to send a request.
4. curl localhost:8080
   - Tests: I ran this line to test if the server would collect the proper the data from the GET request header. After collecting the correct data, I'd check if the server produced the proper GET response to the client. My program provided a 400 status code for the response and the Content-Length equaling to 0.
5. curl localhost:8080/[*filename*]
   - Tests:
     - Running this with a valid file name (fitting the assignment criteria). Assuming the file I'm getting from the server exists, the server should return the data to the client by writing it to the client's standard output and the server would send a 200 OK status code, along with a content length header to the client. The client wouldn't see the response but they'd see the data that they requested.
     - Running this with an invalid file name.

        - < 27 or > 27 ASCII characters or file name not containing characters from A-Z or a-z or “-” or “_”. The server would check the file name after receiving the request and then a response of status code 400 would be sent back to the client, along with the content length equaling to 0. After this response the client would disconnect from the server.
        - 27 ASCII characters, but file doesn’t exist on server. When a client makes a GET request of a file with 27 ASCII characters fitting the file name criteria but the file doesn’t exist on the server, the server will send the client a 404 status code response and a content length header equaling to 0. After this response the client will disconnect from the server.
    - Running this with a valid file name but client has no access to the file. When the client sends a GET request to the server and the file name specified fits the file name criteria but the client doesn’t have access to it (from the server), the server will respond with a 403-status code with the content length header of 0. Once this response is sent, the client would disconnect from the server.

PUT requests

1. curl --upload-file [filename] localhost:8080
    - Tests:
        - I ran this to see if the server was able to store the file that’s being sent by the client in a PUT request. Once the filename is valid and the file was able to be read by the server, the server would return a 201-status code and the content length header of the file’s content length back to the client. This file would then be created and stored into the directory the server is on.
        - If the file name is valid but it requires read permissions, then the server would return a status code of 403 and a content length header of 6, that is when the client tries to put a file, that requires permission, into the server. However, the curl command would prevent the client from sending this request to the server as it will say that it can’t have access to the to send it to the server.
2. curl --upload-file [invalid filename] localhost:8080
    - Tests: If the filename is invalid then the curl command would prevent the client from sending the request. And the curl command would write out to the client that the file doesn’t exist. But ideally, if curl was able to

send the request, the server would send a status code of 400 and a content length of 0 back to the client. Also, if the file isn't accessible for the client when the client sends the PUT request, then the server would send a response back to the client with a status code of 403 and a content length of 0. Then the client would disconnect from the server.

3. curl -H 'Content-Length:' --upload-file [filename] localhost:8080
    - Tests: I ran this test to see if the server would still respond to the PUT request without a content length header in the request's header. Without the content length of the PUT request, the server would continuously read the file specified by the client until the client closes the connection (ctrl + C). As a result, the server will respond to the client with a 201-status code and a content length of how many bytes were read in the file.

NON GET/PUT requests

1. curl --head localhost:8080
    - Test: I tested the server with this to see if it would return a response of status code 500 Internal Server Code, along with a content-length header of 0. As a result, my server does do that because it only performs GET and PUT requests.

To answer the question in the assignment, it asks what happens to the implementation of my server if, during a PUT request with a content length, the connection was closed, ending communication early? My answer to this is that the server would have had some packets sent from the client but not all packets/data would have been received if the connection was cut. The server would only have parts of the data. For the server to obtain all the data, the client would have to send the data packets to the server again. This concern wasn't present during the implementation of the dog program because the program only ran locally on my machine. There was no connection set up between a client and a server because, in a sense, the dog program was itself the client and the server. There was no network established between to separate programs/machines. That's why this complexity being added by an extension of requirements does not occur in the dog program. But this complexity does occur in this HTTP server program.