Jessica Pan

jeypan@ucsc.edu

Faisal Nawab

Fall 2019 – Asgn3: HTTP Server with caching and logging

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.
5. ./httpserver -c localhost 8080

    - This was used to test if the server would create the cache and for the server and only let four files into the cache. As a result, the cache was made properly and only stores at max 4 file/pages.

6. ./httpserver -l logfile.txt localhost 8080
    - This tests for the creation of the log file in the server's directory and also if the program would log the header (and data) of the requests made to the server by a client.
7. ./httpserver -cl logfile.txt localhost 8080
    - This tests if the server would perform both caching and logging. The server would also log differently with the caching flag turned on. As a result the server does log correctly with '[was in cache]' and '[was not in cache]' with the GET/PUT headers (and data).

***With the server already running and listening for a connection (./httpserver -cl logfile.txt 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.

Caching & Logging of requests

For this section of testing, I performed a series of different scenarios to test my program. First, I ran a series of PUT requests with the server, this series contained about 8 pages. All these files consisted 6 different names, where a set of 2 files had the same names but they didn’t have the same contents. Below are a list of the different put requests I used in this scenario:

1. `curl -v -T file1.txt http://localhost:8080 –request-target ABCDEFabcdef012345XYZxy--`
2. `curl -v -T file2.txt http://localhost:8080 –request-target ABCDEFabcdef012345XYZxyz-`
3. `curl -v -T file3.txt http://localhost:8080 –request-target ABCDEFabcdef012345XYZxx--`
4. `curl -v -T file4.txt http://localhost:8080 –request-target ABCDEFabcdef012345XYZx---`
5. `curl -v -T file1.txt http://localhost:8080 –request-target ABCDEFabcdef012345XYZx---`
6. `curl -v -T file6.txt http://localhost:8080 –request-target ABCDEFabcdef012345XYZx--z`
7. `curl -v -T abcdefghijklmnopqrstuvwxyzy http://localhost:8080 –request-target ABCDEFabcdef012345XYZxyz-`
8. `curl -v -T file5.txt http://localhost:8080 –request-target ABCDEFabcdef012345XYZx-zz`

In this scenario, my server has an empty cache at the start.

1. ‘`ABCDEFabcdef012345XYZxy--`’ is pushed into the Cache with the contents in file1.txt
2. Check cache if it already contains a file of this name ‘`ABCDEFabcdef012345XYZxyz-`’ and cache contains less than 4 files. ‘`ABCDEFabcdef012345XYZxyz-`’ is pushed into the Cache with the contents in file2.txt
3. Check cache if it already contains a file of this name ‘`ABCDEFabcdef012345XYZxx--`’ and cache contains less than 4 files. ‘`ABCDEFabcdef012345XYZxx--`’ is pushed into the Cache with the contents in file3.txt
4. Check cache if it already contains a file of this name ‘`ABCDEFabcdef012345XYZx---`’ and cache contains less than 4 files. ‘`ABCDEFabcdef012345XYZx---`’ is pushed into the Cache with the contents in file4.txt
5. Check cache if it already contains a file of this name ‘`ABCDEFabcdef012345XYZx---`’ and cache contains less than 4 files. The cache does contain a file of that name, thus the data in the cache for ‘`ABCDEFabcdef012345XYZx---`’ is overwritten with the new contents of file1.txt
6. Check cache if it already contains a file of this name ‘`ABCDEFabcdef012345XYZx--z`’ and cache contains less than 4 files. Cache contains more than 4 files, each file with a different name than ‘`ABCDEFabcdef012345XYZx--z`’. The cache writes the oldest file in the cache to

the disk and pops it off the queue, in this case, ‘`ABCDEFabcdef012345XYZxy--`’ is popped off the cache. ‘`ABCDEFabcdef012345XYZx--z`’ is now pushed into the cache with the contents in file6.txt

7. Check cache if it already contains a file of this name ‘`ABCDEFabcdef012345XYZxyz-`’ and cache contains less than 4 files. The cache does contain a file of that name, thus the data in the cache for ‘`ABCDEFabcdef012345XYZxyz-`’ is overwritten with the new contents of ‘`abcdefghijklmnopqrstuvwxyzy`’
8. Check cache if it already contains a file of this name ‘`ABCDEFabcdef012345XYZx-zz`’and cache contains less than 4 files. Cache contains more than 4 files, each file with a different name than ‘`ABCDEFabcdef012345XYZx-zz`’. The cache writes the oldest file in the cache to the disk and pops it off the queue, in this case, ‘`ABCDEFabcdef012345XYZxyz-`’ is popped off the cache. ‘`ABCDEFabcdef012345XYZx-zz`’ is now pushed into the cache with the contents in file5.txt

Logfile of this looks like:

```
1. PUT ABCDEFabcdef012345XYZxy-- length 32 [was not in cache]
   (data in hex)
2. PUT ABCDEFabcdef012345XYZxyz- length 29 [was not in cache]
   (data in hex)
3. PUT ABCDEFabcdef012345XYZxx-- length 15 [was in cache]
   (data in hex)
4. PUT ABCDEFabcdef012345XYZx--- length 38 [was not in cache]
   (data in hex)
5. PUT ABCDEFabcdef012345XYZx--- length 20 [was in cache]
   (data in hex)
6. PUT ABCDEFabcdef012345XYZx--z length 10 [was not in cache]
   (data in hex)
7. PUT ABCDEFabcdef012345XYZxyz- length 3 [was in cache] (data
   in hex)
8. PUT ABCDEFabcdef012345XYZx-zz length 89 [was not in cache]
   (data in hex)
```

I performed the same scenario test with GET which was also performed the same way with the same file names as PUT. It’s just when the logging is being performed, there wouldn’t be any data. These two scenarios resulted in the correct outputs and they were also logged correctly. After these two scenarios, I performed a mix of GET and PUT requests. Below is a list the different scenarios I performed for this part:

1. PUT → GET
   a. PUT -T `ABCDEFabcdef012345XYZxy--` `http://localhost:8080`
   b. GET `http://localhost:8080/ABCDEFabcdef012345XYZxy--`

    c. This case, GET would receive the contents of the file that was just put into the cache, not the contents of the file that's on the disk.

2. GET → PUT → GET
    a. GET `http://localhost:8080/ABCDEFabcdef012345XYZxy--`
       `**Gets contents of file from disk and placed into cache`
    b. PUT -T `ABCDEFabcdef012345XYZxy-- http://localhost:8080`
       `**Puts different contents into the file at cache`
    c. GET `http://localhost:8080/ABCDEFabcdef012345XYZxy--`
       `**Gets contents of file from cache which contains the contents from the previous request.`
    d. The contents received in the last GET request will not have the same contents as the first GET request as the PUT request could've overwritten the contents in the file.

Asgn3 Question

When the httpserver has caching, its performance would improve by a lot, especially with big files. Below I've attached a table of the different times for GET and PUT requests with and without caching.

	Without Caching	With Caching
GET	0.069	0.058
GET	0.077	0.073
GET	0.078	0.072
PUT	2.118	1.167

These times were provided from my program after running the same tests for each request at the same time, with and without caching. The caching provided a faster performance of the server this means it's better to have caching in a server than to not have a cache. This increase in performance means that each request is faster in performance. This would mean a decrease in latency, in other words, the amount of time it takes to read/write a byte of data is decreased. A decrease in latency means there's a possibility of a greater throughput, meaning there could be more requests performed within a period of time. Overall, a lower latency and an increase in throughput would mean a faster httpserver.