Jessica Pan

jeypan@ucsc.edu

Faisal Nawab

Fall 2019 – Asgn2: Multithreading HTTP server with logging

WRITEUP

The testing performed on my program was the same unit testing and whole-system testing I performed in the previous assignment 1. With the previous assignment completed, I added onto the code I had for the server. As I added the code to the server, little by little, I performed small tests on the program so that I don't build up technical debt in my program. In other words, after I implement a small portion of the code, I perform a test to check for bugs and if there are bugs, I'd fix them before moving onto implementing other parts of the program. For example, when the arguments taken to run the server was changed to accept flags for the number of threads and logging, I implemented that part first and tested whether the program would work with those arguments properly first before implementing the threads for the program. Once I finished the program, I performed whole-unit testing where I'd run a series of requests on a shell script and see if the server would perform multithreading and logging on the requests. After running the requests, I'd check the log file for its proper contents. Below, I've listed the different whole-system tests I've performed on my program and the reasons of the test (some of these tests are also from Asgn1):

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 -N 3 localhost 8080
    - This tests if the program would create 3 threads to process the requests made by a client. As a result my program does create 3 threads within a thread pool.
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 -N 3 -l logfile.txt localhost 8080
    - This tests if the program would create 3 threads and a log file so the header (and data) of the client would be logged.

***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.

Multithreading requests

Below are the multiple requests (shell script: `shell.sh`) performed on the HTTP server to see if it's able to perform multithreading and complete these requests without breaking the server:

(1) `curl -v http://localhost:8080`
(2) `curl -v http://localhost:8080/invalidfilename`

(3) `curl -v http://localhost:8080/invallllliddfileeeeeenameeeeeeee`

(4) `curl -v http://localhost:8080/fjeillllnotlllfoundllllfile`

(5) `curl -v http://localhost:8080/badsyntax////////////////`

(6) `curl -v http://localhost:8080/abcdefghijklmnopqrstuvwx---`

(7) `curl -v http://localhost:8080/abcdefghijklmnopqrstuvwxyzz`

(8) `curl -v -T file1.txt http://localhost:8080 -request-target ABCDEFabcdef012345XYZxyz-.- > cmd1.output &`

(9) `curl -v -T test.txt http://localhost:8080 -request-target ABCDEFabcdef012345XYZxyzz > cmd2.output &`

(10) `curl -v -T abcdefghijklmnopqrstuvwxyzy http://localhost:8080 -request-target ABCDEFabcdef012345XYZx--- > cmd3.output &`

(11) `curl -data @abcdefghijklmnopqrstuvwxyzy http://localhost:8080 -v`

Commands 1-7: These requests are all GET requests that would fail with a 400 status code, except 6 & 7. The 6th command will result in a successful GET response and the 7th command will result in a failed GET request of status code 403.

- What's logged for these requests are the headers of the requests along with the response code of the requests.

Commands 8-10: These requests are all PUT requests and only the 8th and 9th command will fail. The 8th will fail with a status code of 400 and the 9th will fail with a status code of 403. The 10th command will succeed and with the status code of 201.

- What's logged for these PUT requests are the headers and its data. However, only the 8th and 9th command will be logged into the log file with only the header and the status code of the request. The 10th command is a successful PUT request so the header of the request and the data of the file the client is trying to put into the server is logged into the log file (data of the file will be logged in hex, not ASCII).

Commands 11: This command is a POST request which doesn't work with the httpserver so the request would fail and the header would be logged into the file.

**The requests are logged into a log file only if a log file was specified when the user starts to run the program.

Results of these commands (this is an image of what was logged into the log file for these requests):

To run the server: `./httpserver -N 3 -l logfile.txt localhost 8080`

In a separate terminal, I ran the shell script (shell.sh): `./shell.sh`

```
jessica@jessica-VirtualBox:~/CSE130/Asgn2$ cat logfile.txt
FAIL: GET / HTTP/1.1 --- response 400
========
FAIL: GET invalidfilename HTTP/1.1 --- response 400
========
FAIL: GET invallllliddfileeeeenameeeee HTTP/1.1 --- response 400
========
FAIL: GET fjeillllnotlllfoundllllfile HTTP/1.1 --- response 404
========
FAIL: GET badsyntax HTTP/1.1 --- response 400
========
GET abcdefghijklmnopqrstuvwx--- length 0
========
FAIL: GET abcdefghijklmnopqrstuvwxyzz HTTP/1.1 --- response 403
========
FAIL: POST / HTTP/1.1 --- response 500
========
FAIL: PUT abcdefghijklmnopqrstuvwxyzz HTTP/1.1 --- response 403
========
PUT ABCDEFabcdef012345XYZxyz-.- length 27
00000000 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50 51 52 53 54
00000020 55 56 57 58 59 5a 0a
========
PUT abcdefghijklmnopqrstuvwx--- length 63
00000000 74 65 73 74 74 74 74 0a 68 69 69 69 69 69 69 69 0a 74 68 65
00000020 72 65 0a 0a 41 48 48 48 48 48 48 0a 41 47 47 48 41 47 48 48
00000040 48 47 48 47 48 47 48 48 47 48 48 0a 61 73 64 66 67 68 6a 6b
00000060 6c 3b 0a
========
jessica@jessica-VirtualBox:~/CSE130/Asgn2$
```

Asgn2 Questions:

After performing the tasks required to answer the assignment questions, the difference in performance were fairly similar. When the shell script (in the multithreading requests section) was ran on the assignment 1 server, the performance was around the same as the multithreading server. The reason these two servers had fairly similar performance times were that asgn1's server didn't have multithreading and this assignment's server requires time for logging the data of each request into the log file.

The bottleneck of my system would most likely be the use of all the threads in the system. If the threads in the system were all occupied by a request then the following requests would build up and wait for the next available thread. In this case, the requests that are occupying a thread would be preventing following requests from being processed and the availability of logging the requests would be greatly increased. I say this because the logging of a request could be done contiguously, meaning more than one thread could be writing to the log file at the same time. That means there's no restriction to the number of requests or threads being written to the log file. What's holding up the line in logging the requests is the occupation of all the threads. With no threads available, we'd have to wait for one of the threads to finish it's request, log its data and allow the next request in.

With the various parts of the system, there's concurrency within the worker threads and logging. The concurrency in the worker threads is that multiple threads can be processed at the same time. Multiple threads could be parsing its header, changing the offset for logging and writing to the client's standard output at the same time. The concurrency with logging is that each thread reserves a certain number of bytes in the log file which allows multiple threads to write to the log file at the same time and not write over each other. However, in our system, there's no concurrency with dispatching of threads because the system only assigns the requests to a thread one at a time. To increase concurrency in our system, we can improve the concurrency of the dispatchers by having more dispatching threads so more requests could be assigned to a thread in the thread pool. There isn't much we can increase in concurrency with logging as there seems to be no limitation on how many threads can write to the file. As for increasing concurrency of worker threads, you could probably create more threads than before, allowing for more threads to be working in the system.