Design Document: Multi Threaded HTTP Server with Logging

Joshua Cheung
CruzId: johcheun

1 Goals

The goal of this program is to create a simple multi-threaded HTTP server with GET. PUT, and HEAD command capabilities. These commands will allow us to write and read data into or from a server. For GET, we must be able to grab the specified file and send the contents and its length back to the client. For PUT, we must be able to read the contents of the file sent in by the client, and write it to a new file created in our server. Our HEAD request will simply return the header response back to the client.

2 Design

First, we need to create a connection between the server and the client that will be able to read the headers that come in from the client, and read the requests accordingly. We will be using the curl command as a client placement to test our command.

Handling Command Line Arguments:

Our program must be able to handle several scenarios:

- N flag for number of threads
- l flag for log_file name
- Port number

To handle this we will be using the getopt() function which checks for specific flag characters. This will be used for the log and number of threads specification. Then, we will use optind to get the argument without a flag, which is the port number. As said in the document specification, the log and threads flags are optional whereas the port number is a mandatory argument.

Making Server and Connecting to the Client:

There are several steps needed in order to create a connection to the server:

1) Connect to the server using a socket()
2) Use setsockopt() to be able to reuse address and port of server
3) Bind() the socket to the specified port number, for ex. Localhost
4) Listen() to wait for the client to connect to the server
5) Use the Accept() and connect() function to establish a connection between client and server. We will be infinitely running this part using a while loop to be able to receive multiple client requests without the server closing.

After we have created a connection between the server and client, we will parse the header received from the client to check whether it is a GET, PUT, or HEAD by using sscanf() which separates strings by white space, allowing us to grab specific parts of the header. After determining the request type, we will pass the socket and header as a char array into our respective request functions. Our response headers will consist of the server type, success/fail code, and if succeeded we will send the content length of the file.

Implementing Multithreading:

To implement multithreading we will first initialize a default amount of threads 4 as specified in the document specification. We will then implement a semaphore queue structure that has the specified amount of threads.

Mutexes:
We will use mutexes by using the pthread library.
Pthread: the purpose of having these mutexes is so that we do not have race conditions when multiple threads try to access on variable. We will need to lock/unlock:

- Variable keeping track of number of threads in use
- Variable calculating offset when writing to log file

Handling multiple requests:

- Every time we receive a request, we will call sem_wait which decrements the threads available in the semaphore, and if there are no threads left then the semaphore will wait until threads become available
- After completing a request we call sem_post which increments the number of threads available in the semaphore
- After a request we call pthread join on the thread that was processing the thread to get rid of it

Implementing Logging:

My logic for this part was that whenever I send something back to the client, in the same space I do my logging. I have a global offset variable that keeps track of where to write to in our log file, which we lock every time we add to the offset. To write to our logfile with a given offset we use the pwrite() function, which takes in the file descriptor we are writing to, the content of what to write, the length of what we are writing, and the offset of where to write to in the logfile.

Hex Characters:
If we have a successful request without errors, we need to convert every character in the requested file to hex, and then write it to our logfile. We implement this by using sprintf with a buffer argument, an “x” argument and the character we want to convert, where x converts the

char to a hex value, where we then write it to our log file, looping through until we have converted all chars of the respective file.

Function 1: GET REQUEST

getRequest(socket, header)

1) Read the header returned by the client into a buffer.
2) Parse the header using sscanf to get the server type and file name we are getting.
    a) We will need to loop through the filename to get rid of the '/' character that precedes it
3) Check for bad request and forbidden errors when opening the file with a file descriptor. The specification for these errors will be discussed in another section below.
4) Send the appropriate header response
5) Similar to our dog assignment, read() through the file using our buffer size at a time, and then while our read() is greater than 0 (while there is something left to read) send the contents of the file back to the client. In each iteration, we will be sending:
    a) Arg1: socket
    b) Arg2: contents of our buffer
    c) Arg3: Length of what we have read

    Afterwards we will clear what we have read into our buffer to prepare for the next iteration.
6) After we have finished sending, we will free our file buffer and close our descriptor.

Pseudocode

```
Parse file, and server from header
Open file from server
If errors
        Send header error response to client
else
        While all contents are not read
                Read contents of file to buffer by repeatedly allocating 32 kiB
        Send buffer back to client
        Send header and success code back to client
```

Function 2: PUT REQUEST

putRequest(socket, header)

1) Read the header returned by the client into a buffer.

2) Parse the header using sscanf to get the server type and file name we are getting.
    a) We will need to loop through the filename to get rid of the '/' character that precedes it.
3) Check if the file we are putting already exists. If so, delete the file and recreate it, then check for permission errors and bad request errors.
4) Send the appropriate header response.
5) Next, we will read data from the client again to get the contents of the file, and read those contents into a buffer using recv().
    a) If the content length of the file exceeds our buffer size, we will need to repeatedly create a new buffer until all the contents of the file are read.
6) Then we will write the buffer with the contents of the file into a new file in our server.

Pseudocode

```
Parse contentLength, and name of textfile
If textfile already exists
        Send code 201 File created
        Remove the texfile
Create textfile
If errors
        Send respective error response header
Else
        Send success response header
        Recv the socket from client to get contents into buffer
                Allocate size of the buffer to be the content length
        Write buffer to the file created
```

Function 3: HEAD REQUEST

headRequest(socket, header)

7) Read the header returned by the client into a buffer.
8) Parse the header using sscanf to get the server type and file name we are getting.
    a) We will need to loop through the filename to get rid of the '/' character that precedes it.
9) Check for bad request and forbidden errors when opening the file with a file descriptor. The specification for these errors will be discussed in another section below.
10) Send the appropriate header response.

My head request code was essentially a copy of my get request without actually reading the contents of the file.

Pseudocode

Parse file, and server from header
Open file from server
If errors
    Send header error response to client
Else
    Send success response header to client

3 Response Codes and Error handling

There are multiple errors we have to account for:

1) 200 OK\r\n → if there are no errors and we have sent everything correctly.
2) 201 Created\r\n → during a put request, if the file did not previously exists in the server.
3) 400 Bad Request\r\n → If the filename is greater than 27 chars, if there exists and invalid character within the file name specified in the asgn spec, or if the request type is neither a GET, PUT, or HEAD. If we ask for a healthcheck when there exists no log file
4) 403 Forbidden → If the permissions for opening the requesting file is denied.
5) 404 Not Found → If the file requested is not found.
6) 500 Internal Server Error → If there is a server error.

Testing

To test the code I would add the -v flag to display the response header

- To test the GET portion code: curl -v hostname:port/fileName.
- To test the PUT portion of the code: curl -v -T fileName hostname:port.
- To test the HEAD portion of the code: curl -v -I hostname:port/fileName

I would be running it on binary files, large files, and small files. I would also be using incorrect input to see if my program handles all the errors correctly.
To test multithreading, I would add a '&' in between curl commands to run these concurrently.