Design Document: Multithreaded HTTP server with logging

Joshua Cheung
CruzId: johcheun

1 Goals

The goal of this program is to create a multi-threaded HTTP server with GET and PUT command capabilities. We will be creating a user input specified or default amount of threads for to process the request. 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. Both these functions will be done in addition to handling errors, which will be specified later on.

2 Design

We will be using the getopt function to parse our command line for -N flag for threads, and -l flag to specify a log file. We will have a global variable that keeps track of how many threads are being used. We will create a thread if the variable is less than the amount requested, and increment the variable. After the thread is finished, we will then decrement the variable to make sure that we do not create threads and thus result in a race condition.

Threading:

We will be creating an array of threads of size N, specified by user or default value of 4. To create threads we will be using the pthread library, which requires a function with a void argument for the thread to call. We will also be using a semaphore to make sure that the threads that are not in use will be sleeping. To know which thread to use when we receive a request, we will have an index variable that increments every time we create a thread, and decrements when the thread is done doing its work. Our void function will be receiving the header and then calling GET and PUT respectively.

```
Global variables:
Initialize count = 0
semaphore t

Main:

sem_wait(t)
Pthread create [count]
count++;
```

```
Void function
    Parse header buffer
    If GET
        Call getRequest Function
    If PUT
        Call putRequest Function
    lock
    Count--
    unlock
    sem_post(t)
```

<u>Logging</u>
All the threads will be able to access the file descriptor regarding the name of the log file. We will get the log file by what comes after the -l flag. We will then open the file as soon as we detect a -l flag.

In our PUT function, we will convert the char buffer containing the file contents to a string, and then call pwrite to our log file using the size of the string. We will make the file descriptor of the log file global, so that it can be written to from all throughout the program, and will close the file descriptor at the end of the main function.

1) Declare global file descriptor for log file
2) Declare global offset int
3) Open file as soon as we detect -l flag
4) Delete file if already present, create file again
5) In put or get, form log string
6) Pwrite log string to file with offset int
7) Add length of log string to offset int

<u>Requests:</u>
First, we need to create a server that will be able to read the headers that come in from the client, and read the requests accordingly. To do this, we must first establish a connection between the server and the client. We will be using the curl command as a client placement to test our command. We will have the option when we run our program to have an address and/or a port argument, but if they are not specified then we will be using default local host and port 80.

There are several steps needed in order to create a connection to the server
1) Connect to the server using a socket()
2) Bind() the socket to the specified port number, for ex. Localhost

3) Use setsockopt() to be able to reuse address and port of server
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 to be able to receive multiple client requests without the server closing.

After we have created a connection between the server and client, we need to account for the two commands, GET and PUT

<u>Algorithm 1:</u>
For the GET functionality

1) Read the header returned by the client into a buffer
2) Parse the header using scanf() and sscanf() to fetch whether or not it is GET request, and the name of the file we are requesting
3) Next we will be using the read() function to read the file into a buffer, and then use the write() function to write the buffer to a new file
4) Next, we will send the length of the file and a success message back to the client
5) We will decrement the threadIndex to show that the thread is done doing the work
6) Release semaphore

Pseudocode

```
Parse file, and server from header
Open file from server
While all contents are not read
    Read contents of file to buffer by repeatedly allocating 32 kiB
Send buffer back to client
Send success code back to client
Decrement threadIndex
Release semaphore
```

<u>Algorithm 2:</u>
For the PUT functionality

1) Read the header returned by the client into a buffer
2) I will be parsing the header using scanf() and sscanf() to fetch whether or not it is a PUT request, the name of the file we will copy data from, and the content length of the file

3) Next, we will read data from the client again to get the contents of the file, and read those contents into a buffer
    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
4) Then we will write the buffer with the contents of the file into a new file in our server
5) We will decrement the threadIndex to show that the thread is done doing the work
6) Release semaphore

Pseudocode

```
    Parse contentLength, and name of textfile
    If textfile already exists
        Send code 201 File created
        Remove the texfile
    Create textfile
    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
    Decrement threadIndex
    Release semaphore
```

3 Error Handling

There are multiple errors we have to account for:

1) If the file we are requesting is not found
    a) If we call open() and find that the return value is less than 0, we use error code 404
2) Incorrect headers
    a) This involves if the file name specified is not equal to 27 characters, then we return error code 400 for bad request. We can do this by parsing our header and checking the length of the text file
    b) Also, if for a PUT request we are missing a content length, we would throw an error
3) Cannot write
    a) If we call write() and find that the return value is less than 0, we use
4) Cannot get
    a) Similarly to step 1, if the file does not exists, then we would use error code 404
5) Cannot Bind

    a) If the bind value returned is less than 0, we perror to stdout because we cannot make a connection to the client

6) Invalid IP
    a) If I had time to implement this, I would of had searched on how to determine if an IP is invalid or valid

Testing

To test the GET portion code, I would be using curl -v hostname:port/fileName, and to test the PUT portion of the code, I would be using curl -v -T fileName hostname:port. 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.

Testing multithreading:

Two requests:

```
Curl -v localhost:8080 --request-target someFile & localhost:8080 --request-target someFile &
Curl -v localhost:8080 --request-target someFile & localhost:8080 --request-target someFile
```

Four requests:

```
Curl -v localhost:8080 --request-target someFile & localhost:8080 --request-target someFile &
Curl -v localhost:8080 --request-target someFile & localhost:8080 --request-target someFile &
Curl -v localhost:8080 --request-target someFile & localhost:8080 --request-target someFile &
Curl -v localhost:8080 --request-target someFile & localhost:8080 --request-target someFile &
```

ECT.