Design Document: httpserver

Cagan Bakirci

CruzID: cbakirci

1. Goals

   This program's goal is to modify the previously implemented multi-threaded HTTP server into a new version that will include the patch feature. The server responds to simple GET, PUT, and PATCH commands. These commands will read a specified file from the server and write a file to a server respectively. The program will not use any of the C library FILE * functions. The threads will not allocate more than 16 KiB of memory each. The server must be able to handle multiple requests simultaneously, each in its own thread.

2. Design

   There are multiple parts of the design. These multiple parts have been explained in this section.

   - First of all, the initial step of this project will be to re-structure the existing multi-threaded HTTP server. The previous version of the program was not too complicated, so almost every functionality was implemented within the thread function. The previous version of the program only used a few functions outside of the main. However, this version of the program is too complicated to handle all functionality within the main function. So, the initial step of this project will be breaking up the code into feasible components and making sure the code still works as expected.
   - Second step will be to the implementation of the algorithm that check the arguments presented to the server. The server must still be given at least 2 arguments, but the number of arguments presented to the server can be as high as 9 in this case. The server must parse through these arguments, extract the information, and assign the right values to the right variables. If the number of arguments is less than 2, the algorithm exits. The server must also assure that the argument -a is presented. This is shown in Algorithm 1.

```
portval <- 80
found <- false

for I = 1 to argc-1
        if argv[i] <- "-a"
                found <- true
                hashFileName <- argv[i + 1]
        else if argv[i] <- "-N"
                noThread <- argv[i + 1]
        else if argv[i] <- "-l"
                logCreate <- True
                filename <- argv[i + 1]
        else
                hostname <- argv[i]
                portVal <- argv[i + 1]
if (!found)
        //Error: -a must be present
        Exit()
```

Algorithm 1.

- The second part of the design will be concerned with everything the old algorithm does with the addition of patching. The patching is done in couple of stages. The first stage is to add the patch command and the ability to recognize the patch command. The updated pseudocode is given below:

```
while (there is no reason to wake up)
        //Sleep
recieve
Parse the socked into three parts <- first, second, third
bool flag <- validName(second, len)
if (flag)
        //Throw an error message
        Send the header Message
        Write to log if -l is passes as an argument
if (first == “GET”) // This section will handle get requests
                //Handle get thread
else if (first == “PUT”) // This section will handle put requests
                //Handle put thread
else if (first == “PATCH”)
                //Handle patch thread
else
                //Set the error code
                Send the header Message
                Write to log if -l is passes as an argument
```

Algorithm 2

- The third part of the design is to write the algorithm that will handle the patch thread. The server will already be checking whether the arguments contain "-l" command. If the argument contains this command, the server will open a file with the named of the argument immediately after -l. The log file will be kept track by a Boolean. If the -l is present in the code, this flag will tell where the logWrite function should be executed. The implementation of the logWrite function will be the last step of the project.

```
Recv <- acpVal into buff
Parse the buff for exName & alias
Open <- exName
        If (open)
                //Error
        Else
                insertHash(alias, exName)
```

Algorithm 3.

- The fourth part of the project is designing the algorithm is the helper function instertHash that is used above. The pseudocode is given in the Algorithm 4.

```
For all i in alias
        Key += alias[i] * i
Loc = (key % 8000) * 128
hasFile <- write exName to Loc
```

Algorithm 4.

- The fifth part of the project is designing the algorithm is the helper function getHash function to calculate the correct location to extract the necessary information. The pseudocode is given in the Algorithm 5.

```
For all i in alias
        Key += alias[i] * i
Loc = (key % 8000) * 128
hasFile <- read exName to a variable
```

Algorithm 5

- The final part of the algorithm is to update the algorithm to handle the patch command when needed. The updated versions of the articles will be taken from the alias files and the values will be updated. The second value extracted from the article should be passed as second to the algorithm. To do this, we will want to make sure that the name was found in the hash file. The algorithm that handles this code is given below:
-

```
If (name not valid && hashNameArray is empty)
        second <- hashNameArray
if (name not valid && not patch command && notinHash)
        //Invalid name
```

-

- *Algorithm 6*