Design Document: Adding Aliases to HTTP Server

Julia Sales
Cruz ID: jesales

---

1) Goals

The goal for Assignment 3 is to add aliases to our http server. An alias is another name that can refer to the target file in the HTTP server's key-value store. In order to do this we need to find a way to support the PATCH command and use a hash table. We need to use city hash

*Another goal is to make sure the previous assignment works and Netcat.

---

2) Design

For this assignment, it will take three parts. The first part we need to do is to add -a to getopt. We then need to handle the PATCH request in worker. Inside handling the PATCH request, we then want to put our Aliases into the hash table. For this we will be using cityhash. the hash table.

* I redid my design for asgn 2 to get multithreading, netcat and logging working.

2.1 Completed Multithreading, Netcat, and Logging

I redid my assignment so my logic and code changed from the assignment before. I created 7 functions.

Main

```
local : nthreads ← 4 (default threads)
local : option ← for getopt(3)
local : font, back ← both initialized to 0, meant for queuing
local : log, d_rc
Global : LOG_FILE, SERVER_NAME_STRING, PORT_NUMBER
Global mutex locks : queue_lock, worker_lock
Global conditional lock : worker_signal

1)  main(argc, argv)
    Parsing the header using getopt(3)
2)      | while (option = getopt(argc, argv, "N:l:") is not -1 ← we use : to indicate
    required arg after flag
3)          | switch option
4)              | case "N"
5)                  | nthreads = number that is entered in args after "-N"
6)                  | if nthreads < 4 then error and exit failure
7)                  | break
8)              | case "l"
```

```
9)                | LOG_FILE = log file name that is entered in args after "-l"
10)               | create log file using open
11)               | if log file cannot be created, then error: "Cannot open", exit
12)               | break
13)           | default
14)               | exit

15)  if server name or port number is NULL
16)      | error: Missing server name and/or port address
17)      | exit
18)  SERVER_NAME_STRING ← whatever the args has, ex: "localhost"
19)  PORT_NUMBER ← whatever the args has, ex: "8080"

20)  log ← when we open the file, we also create it and then need to close after
creation
21)  front = back = 0
22)  Using pthread_mutex_init: queue_lock  ← NULL, worker_lock ← NULL
23)  Using pthread_cond_init: worker_signal ← NULL
24)  Using pthread_t: dispatch_thread, worker_id[nthreads] ← pool of workers

25)  struct dispatch_params dp;
26)  dp.bindaddr ← SERVER_NAME_STRING
27)  dp.port ← PORT_NUMBER
28)  Create dispatcher thread using pthread_create

29)  starting the worker threads
30)  for 1 to nthreads
```

Create Socket

Created a function that creates the socket and binds to the server. This was originally in my main, but then decided to make it a function to improve modularity. All code in this function was provided in section for ASGN2.

```
1)  function create_socket(bindaddr, port)
2)     create sockaddr_in
3)     | struct hostent *hent = gethostbyname(bindaddr)
4)     | struct sockaddr_in addr
5)     | memcpy(&addr.sin_addr.s_addr, hent→h_addr, hent→h_length)
6)     | addr.sin_port = htons(port)
7)     | addr.sin_family = AF_INET
8)     | sock = socket(AF_INET, SOCK_STREAM, 0)
9)     | if sock = 0 then error: "In socket" and exit

10)    bind socket to server
11)    | enable ← 1
12)    | setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(enable))
13)    | if bind of the sock is < 0, then error: "In bind" and exit

14)    | return sock
```

GET Function

This function performs the GET request and prints out the appropriate status codes. This was also in my main in ASGN1 and moved it to a function for ASGN2.

```
local : resp_buff[80] ← will store the 200 OK status code
local : file_buff[BUFFSIZE]
Global: BUFFSIZE = 16000
local : file_read ← will use for read

1)  function get_function(con_l, r_target)
2)      | struct stat st;

3)      | fd ← open the r_target file and have read write permissions
4)      | if fd cannot be opened ← if fd = -1
5)          | write to cl : file not found status code
6)      | else if fd can be opened
7)          | file_read ← read the fd and put contents into file_buff, with BUFFSIZE
8)          | stat ← of r_target and st
9)          | write to con_cl, the client, the 200 OK status code
10)         | while file_read >= BUFFSIZE ← this is a large file
11)             | write the content from the resp_buff to the client, con_l
12)             | then read from the fd again
13)         | write the last portion of content or if it's not a large file, then just write out
    the contents
14)     | else
15)         | write to client: 500 Internal server error status code
16)     | close the fd
```

Logging Function

The logging function will take in the file descriptor, the lines, and the logging file. It will take the request and the target file as well as content length and write it to the logging file. After it write it to the logging file, it will covert the contents of the target files into hex and write it 5 bytes at a time per line. It will also log status codes for FAIL.

```
local : log_enable ← this is our flag for enabling logging
local : first_buff[100] ← this will take in the request, target and content_length
local : prev_offset ← gets previous offset

1)  function logging (request, target, type, content_length, buffer, status code,
    log_file)
2)      | check if logging is enabled ← set to 0 if there is no logging, if there is
    logging it is set to 1
3)      | if logging is enabled
4)          | mutex.lock()
5)          | mutex.wait()
6)          | set prev_offset to 0 ← the original offset
7)          | if status code is 200 OK or 201 Created
8)              | parse and get the request, target, and content_length
```

```
9)          | then read the buffer and converting it into hex
10)         | for i=0, i < strlen(buffer), i++ ← this is converting byte by byte
11)             | need to have padded zeros and check for every 20 bytes ← use
    i%20
12)             | then we convert the chars to hex using "%s2x" ← a string every 2
    bytes into hex
13)         | write hexed content and header to the log file
14)         | print out the ======== to indicate end of logging that one request
15)       | if status code is 400 Bad Request, 403 Forbidden, 404 File not Found, or
    500 Internal server error, then it is a FAIL
16)         | get header and status code number and put it to a fail buffer ← fail
    message
17)         | write the fail message to the log file
18)         | print out the ======== to indicate end of logging that one request
19)       | mutex.unlock()
20)       | mutex.signal()
```

Queue Insert Function

This queue function will insert sockets into a stack. It will take in a value.

```
parameter : value ← int
Global : MAX ← 20
Global : queue[MAX] ← the queue/stack
Global : front, back ← both of these defined in main

1)  function queue_insert(value)
2)      | if back = MAX, then this indicates our queue is full
3)      | else if back < MAX insert the socket at the back of the queue
```

Queue Pop Function

This queue function will take the socket out of the queue from the front. This function takes in no parameters.

```
local : pop_value ← int
Global : front, back ← both of these defined in main

1)  function queue_pop()
2)      | if font = back, the the queue is empty
3)      | else assign queue[front] to pop_value
4)      | shift all elements over
5)      | for 0 to back, shift queue by 1
6)      | decrement back
7)      | return the pop_value
```

Dispatcher

This is the function that the dispatcher thread uses. It listens to the socket and accepts connections. When connections arrive, they get placed into the queue.

```
parameters : d_params ← void value, will be used in a struct
local : sock ← int, this will create the socket
local : cl ← the client socket

1)  function dispatcher(d_params)
2)      | struct dispatch_params d_par = d_params
3)      | sock ← use create_socket and have d_par point to the server name in
    bindaddr and the port number in port
4)      | listen ← take in sock and 10

5)      | while(1) ← infinite
6)          | cl ← accept with sock and NULL values
7)          | ↑ this will establish the connection
8)          | we then lock the queue with pthread_mutex_lock so we can insert the
    socket.
9)          | call queue_insert to insert the socket cl
10)         | we then want to release the queue lock using pthread_mutex_unlock
11)         | and then send a signal to the server using pthread_cond_signal
```

Worker

This is the function that the worker threads will use. The worker will wait until the dispatcher signals that the queue is ready. Then one of the workers will take the socket out of the queue and start processing the socket and parse the header and perform the GET and PUT requests. This function takes in no parameters. For the netcat portion of the code, it goes line by line.

```
local : myid, valread
local : write_line_mode ← this is a flag for put, this will indicate if the client is
sending in a curl or netcat command

1)  function worker()
2)      | lock the worker lock so the worker can do work using pthread_mutex_lock
3)      | have the make the worker thread wait until it receives a signal
4)      | after we get a signal and a lock, we need to lock the queue so we can pop
    from the queue
5)      | unlock worker, so other threads can pick up items from the queue
6)      | process connection
7)      | read from client and put it in a buffer
8)      | if valread is -1 then it is an error: “In read”, exit
```

9) | while valread > 0 ← keep reading until end of file stream
10) | copy over the buffer to a process_buffer and then process line by line
11) | while the next line is not null
12) | if write_line_mode = 1 ← this means that this is a netcat command
13) | check if putbytes_left > 0 ← checks if there are still bytes left for file
14) | if file_d > 0 ← write if file_d is good
15) | else just move on to the next request
16) | the program still needs to go through the rest of the buffer. It will continue to count the number of bytes, but will not write, just pass the content
17) | else
18) | there is no more data to write
19) | reset write_line_mode back to 0
20) | close the file

GET REQUEST

21) | if write_line_mode is 0 and the request is a GET
22) | parse the header to get the request, target, and HTTP type
23) | check if there's a slash at the beginning of the target file and remove it
24) | if there's a slash in the middle of the target name and not at the beginning, return a 403 forbidden error
25) | if the target name is not 27 characters long, return a 404 bad request error
26) | else call the get_function

PUT REQUEST

27) | if write_line_mode is 0 and the request is a PUT
28) | parse the header to get the request, target, and HTTP type
29) | check if there's a slash at the beginning of the target file and remove it
30) | if there's a slash in the middle of the target name and not at the beginning, return a 403 forbidden error
31) | if the target name is not 27 characters long, return a 404 bad request error
32) | if there is no target name then return a 404 bad request error
33) | else
34) | create a file using open
35) | if file cannot be created, open returns a -1, then return a 500 internal internal server error
36) | we want to keep track of how many bytes we write by setting putbytes_left to content_len
37) | look for a put continue or blank with next line ← this will tell us if it is a curl or nectat command
38) | if there is a put continue then we know this is a curl command and will read from a file.
39) | if it is a large file keep track of putbytes_left
40) | if putbytes_left > 0 then clear the buffer and read from the file again and put all the contents into the newly cleared buffer
41) | if large file, keep reading and writing from buffer and file
42) | close file
43) | write the created message
44) | else it is netcat
45) | set write_line_mode to 1
46) | close(client)

2.2 Implemented Aliases

2.2(a) -a in command line

First we want to edit getopt to include the -a argument. If there is no -a in the argument we pass to the command line, the our program fails. This will create the mapping file if one does not exist.

```
local : magic_num ← this will be 1, will check if mapping file is valid

1)    | while (option = getopt(argc, argv, "N:l:a") is not -1 ← we use : to indicate
      required arg after flag
2)          | switch option
3)              | case "N"
4)                  | nthreads = number that is entered in args after "-N"
5)                  | if nthreads < 4 then error and exit failure
6)                  | break
7)              | case "l"
9)                  | LOG_FILE = log file name that is entered in args after "-l"
10)                 | create log file using open
11)                 | if log file cannot be created, then error: "Cannot open", exit
12)                 | break
13)             | case "a"
14)                 | ALIAS = the new name
15)                 | if mapping file does not exist create the mapping file ← will hold all
      key and value pairs
16)                 | if mapping file cannot be created, then error: "file cannot be
17)                 | if mapping does exist, just open it and check for magic number
18)                 | break
19)             | default
20)                 | if ALIAS is still NULL
21)                     | calls an error: "Not found"
22)                     | exit
23)                 | exit
```

2.2(b) PATCH Request

Another request, like GET or PUT, however it will pair the new name to the existing name in the hash table. Cityhash functions are used here.

```
1)  If request is PATCH
2)      | read the and parse the header line by line
3)      | the first header is PATCH <new name> HTTP/1.1
4)      | then check if there is an Expect 100 message or blank line
5)      | if Expect 100 message then we know this is a curl command
6)          | send 100 status code
```

```
7)        | read from the socket again to get the Alias name
8)        | if existing name does not exist on server, return 404 Not Found error
9)        | else store all aliases in hash table ← insert() from city hash
10)   | else, if blank line, then we know it's a netcat command
11)       | check for "ALIAS"
12)       | insert the key value pair into the hash table

// inserting into the hash table will be from cityhash which was by google
```

2.2(c) Implementing city hash

Cityhash is an open source hash function that I have downloaded from Google's github: https://github.com/google/cityhash/

We first import cityhash into our program. We need the city.h file in our directory in order for city hash to work.

```
include "city.h" ← this will be in main with all the other libraries
include "city.cc" ← this will also be in main
```

We use the CityHash64 function which is defined in city.h and city.cc. We use CityHash64 because the total length of the two strings (key and value) will be no longer than 128 bytes.

CityHash64 accepts a char and the length of the char and returns a uint64 type. It returns a uint64 type and creates a hash table. In this case we would call CityHash64 twice, once for the existing name and once for the new name.

Example:

```
1) uint64 new_hash, existing_hash;
2) existing_hash = CityHash64(*existing_hash, 27)
3) new_hash = CityHash64(*new_hash, 27)

// this is how cityhash is ran
```