Asgn 2 Design Doc

Daniel Xiong (dxiong5@ucsc.edu)
Daniel Chen (dchen61@ucsc.edu)

1 Design

`httpserver.cpp` is meant to be a simple HTTP server that handles only GET and PUT commands. The server will create, store, read, and write files in the server directory. The server will also follow the HTTP protocol, which means it will parse HTTP headers. The server is capable of multi-threading; the user can use the command line flag `-N` followed by a number that specifies the number of threads the server will use. The addition of multi-threading means each thread can process a request, which increases throughput of the server. However, multi-threading also means we must design the server in a way that is thread-safe. In addition, the user can use the flag `-r` to activate redundancy. Redundancy means that the server will store three copies of each file in folders named `copy1`, `copy2`, and `copy3`.

First, the server will parse command line arguments using `getopt()`. The address is the only required command line argument. The port, redundancy flag, and threads flag are all optional, with the port number defaulting to 80 and the number of threads defaulting to 4. Then, the server will initialize the socket using the `socket`, `bind`, and `listen` syscall functions.

Since it is possible that multiple threads may try to read/write to the same file, we will need mutex locks for each file to ensure mutual exclusion. We use these mutexes in the following way: if a thread wants to have access to a file, that thread must lock the file's mutex to have access to the critical region, and then unlock the mutex when the thread leaves the critical region. In this case, the critical region would be reading or writing to the file. The server uses a hash-table, or `unordered_map` in C++, to store mappings of file names to their respective mutex. This hash-table, named `file_mutex_map`, is initialized with the current existing files in the server directory, including the files in the `copy1`, `copy2`, and `copy3` folders if the redundancy flag is specified.

Next, the server creates the main dispatcher thread and the worker threads. Since the number of threads can vary, we use a C++`vector` of size `N` which is initialized with `pthread_t` instances. When we create each thread, we pass them the `dispatcher()` and `worker()` functions for the dispatcher thread and worker threads, respectively.

There is a shared data struct that is passed to each thread, and it is what facilitates the communication between threads. The shared data struct contains: a conditional variable to activate worker threads, the file descriptor that the dispatcher uses to get incoming connections, a C++`queue` to store connections, a mutex lock for that connections queue, the `file_mutex_map` hash-table, and a boolean variable for whether redundancy is active or not.

The dispatcher function is very simple and runs in an infinite loop; the thread uses the `accept()` syscall and gets the file descriptor for the accepted socket. Then, the mutex for the connections queue is locked so that the file descriptor can be pushed into the connections queue safely. The mutex is then unlocked and the dispatcher signals a conditional variable to activate any worker threads waiting for a signal. We need these mutexes because we want only one thread to be able to access the queue at a single moment; access to the queue, whether it be pushing or popping, can be seen as a critical region.

The worker function also runs in an infinite loop. Each worker thread first initializes a 16KiB buffer to use for handling requests. Next, the thread locks the connections queue, and if the queue isn't empty, a connection file descriptor is popped off the queue and the queue mutex is unlocked and the thread continues to handle requests using that connection. Otherwise, if the queue is empty, the thread sleeps and waits for a signal to the worker conditional variable. The signal is sent from the dispatcher thread, and is sent after the queue has at least one connection in it.

For each connection a worker thread gets, it will get requests using the `recv()` syscall until the connection is closed. Once the server receives a request, the server will parse the HTTP header and extract three things: the command type (PUT/GET), the file name, and the content length. The server then does a check to see if the file name is valid (contains only alphanumeric characters and has length 10).

If the HTTP header specified a PUT request, then `handle_put()` or `handle_put_redundancy()` is called, depending on whether redundancy is active or not.

If there is no redundancy, then the function adds the incoming file name and a new mutex to the `file_mutex_map` hash-table and then locks the mutex. We have to ensure there is mutual exclusion on the file because we don't want two threads to write to a file at the same time. Then, the file is opened, and the function reads data sent from the client into the buffer and writes it to the file. Lastly, the file mutex is unlocked so other threads can access the file.

On the other hand, if there is redundancy, then the function adds each of the three copies of the file to the `file_mutex_map` hash-table and then locks each of the mutexes for the three files. Then, the files are opened, and the function reads data sent from the client into the buffer and writes it to all three copies. Lastly, the file mutexes are unlocked so other threads can access the files.

If the HTTP header specified a GET request, then `handle_get()` or `handle_get_redundancy()` is called and they also do different things based on whether redundancy is active or not.

If there is no redundancy, then the function will open the specified file for reading. It will first get the Content-Length of the file and send it to the client, so the client knows how many bytes to expect. Then, the function reads from the file into the buffer and sends the data to the client.

On the other hand, if there is redundancy, then the function opens each copy of the file for reading. Before the function actually starts reading, it must first check if there are at least two files that are the same. For this check, we compare both the size of the files and the contents of the files. If both the size and content are exactly the same, then the files are the same. The function sends the Content-Length to the client and then reads from the file into the buffer and sends the data to the client.

2 Data Structures

2.1 `header`

`header` is a struct with members `char* command` (PUT/GET), `char* resource_name` (File name), and `int content_length` (Size of data). We create objects of type `header` to represent the HTTP request header.

2.2 `shared_data`

`shared_data` contains the members `pthread_cond_t worker_cond`, `int listen_fd`, `queue<int> connections_queue`, `pthread_mutex_t connections_queue_mutex`, `unordered_map<string, pthread_mutex_t> file_mutex_map`, and `bool redundancy`. This shared data is used across the main, dispatcher thread, and worker threads.

3 Functions

3.1 struct header parseHeader(char buf[])

We use the `parseHeader` function to parse through a HTTP request header string (`char buf[]`) and extract the `command` (PUT/GET), `resource_name` (File name), and `content_length` (Size of data). If no content length is provided, `content_length` would be equal to -1. For the output, an object of type `header` is returned.

3.2 int send_response(int comm_fd, int response_num, int content_len, char* resource_name)

We use this function whenever we want to send a response. This function takes in as input the file descriptor from accepting a connection (`int comm_fd`), the response number we want to send (`int response_num`), the content length from the header (`content_len`), and the filename from the header (`char* resource_name`). The response numbers we can send are: `200, 201, 400, 403, 404, and 500`. We compare the response number to one of the HTTP status codes and respond appropriately. The function will return 0 upon completion.

3.3 unsigned long getaddr(char* name)

We use this function to convert the hostname/ip address string given as a command line argument (input) into an address that is readable by the `sockaddr` object (output).

3.4 int compare_two_files(const char* file1, const char* file2)

This function opens the two files provided as arguments, reads them character by character, and checks if the two files are identical. It returns true (1) if they are identical and false (0) otherwise.

3.5 int get_which_file(const char* file1, const char* file2, const char* file3)

This function determines which of the three files are identical to each other by calling the previous function `compare_two_files(file1, file2)`. It calls the function three times with file1 and file2, file1 and file3, and file2 and file3 passed as arguments. If one of the function calls returns `true`, then the function returns the file number of the first argument that was passed in to that function call. The function does not check if all three files are identical because it would result in the same file number being returned. If all three function calls return false, it returns -1 to indicate a 500 error response should be sent.

3.6 int handle_put(int comm_fd, char buf[], char* resource_name, int content_length, struct shared_data* shared)

This function is mostly the same as from the previous assignment and handles the PUT requests when '-r' is not provided. It takes in as input the file descriptor for the communication channel, the communication channel buffer, the resource name from the header, the content length from the header, and the shared data. The function first removes the '/' from the resource name, inserts the file into the mutex map if it hasn't been encountered before, and locks the file. Then, it opens the file with the name of 'resource name' with `O_RDWR`, `O_CREAT`, and `O_TRUNC` flags and checks for any errors with opening it. If the content length was provided by the header, it would keep receiving and writing to the opened file until the content length becomes less than or equal to 0. If the content length wasn't provided, it would just keep receiving and writing to the opened file until it reaches EOF. For both cases, once the reading and writing is done, the function closes the file and sends a 201 response. The function then unlocks the file and returns 0 for success.

3.7 int handle_put_redundancy(int comm_fd, char buf[], char* resource_name, int content_length, struct shared_data* shared)

This function gets called to handle PUT requests when '-r' is provided. Like the previous function, it takes in as input the file descriptor for the communication channel, the communication channel buffer, the resource name from the header, the content length from the header, and the shared data. The function first removes '/' from the resource name and creates three strings by adding 'copy1/', 'copy2/', and 'copy3/' to the beginning of `resource_name`. Then, it inserts the three files to the mutex map if they haven't been encountered before and locks them. Next, it opens the three files with `O_RDWR`, `O_CREAT`, and `O_TRUNC` flags and checks for any errors with opening them for at least two of them. If only one file has problems opening, the function can still continue with the other two copies. If the content length was provided by the header, it keeps receiving and writing to the opened files until content length is less than or equal to 0. If the content length wasn't provided, the function receives and writes to the opened files until it reaches EOF. Once the receiving and writing is done for both cases, it closes the opened files, sends a 201 response, unlocks the files, and returns 0 for success.

3.8 int handle_get(int comm_fd, char buf[], char* resource_name, int content_length, struct shared_data* shared)

This function gets called to handle GET requests when '-r' is not provided. Like the function for PUTs, the inputs are the file descriptor for the communication channel, the communication channel buffer, the resource name from the header, the content length from the header, and the shared data. First, we remove the '/' from the resource name and check if the file hasn't been encountered before by looking in the mutex map. It it hasn't, a 404 response is sent and ends. Otherwise, it locks the file and opens the file with the name given by `resource_name` with the `O_RDONLY` flag. Then, we check if there were any problems with opening the file and send based on the `errno` if there was. Next, we create a buffer of size 16384 and keep reading to figure out what the content length is of the requested file. Once we have the content length, we send a 200 response with that content length and reopen the file for sending. We keep sending until we reach the amount of the content length that was counted. Finally, the function closes the opened file, unlocks it from the mutex, and returns 0 for success.

3.9 int handle_get_redundancy(int comm_fd, char buf[], char* resource_name, int content_length, struct shared_data* shared)

This function is for GET requests when '-r' is provided. Like the previous function, the inputs are the file descriptor for the communication channel, the communication channel buffer, the resource name from the header, the content length from the header, and the shared data. It first removes the '/' from the resource name and creates three strings by adding 'copy1/', 'copy2/', and 'copy3/' to the beginning of the resource name. It checks if the they have been encountered before by checking the mutex map and sends a 404 response if they haven't. If they are in the mutex map, then the three files get locked. Next, the function opens the three files and checks if at least two of them have problems opening. A response is sent based on the `errno` if there was a problem. Then, it calls the function `get_which_file()` to determine which file should be used for sending. Once it knows which file to use, it counts the content length of it by reading till the very end, closes all the files that were opened, and sends a 200 response with the content length. After that, it reopens the same file that was should be used and keeps sending based on the content length. Finally, the function closes the opened file, unlocks the three previously locked files, and returns 0 for success.

3.10 void* dispatcher(void* data)

The `dispatcher()` function is what the dispatcher thread uses. The job of this function is to push incoming connections into the connections queue and to wake up worker threads to process connections.

3.11 void* worker(void* data)

The `worker()` function is what the worker threads use. The job of this function is to pop off connections from the connections queue or sleep until there are connections in the queue. This function handles requests from a connection until the connection is closed. Each request header is parsed and calls the appropriate PUT/GET function. Once a worker thread is created, it does not end.

4 Testing

The base code we used to start this assignment was our code from assignment 1, so we knew that the functionalities of the server used in assignment 1 were already working.

To test the multi-threading feature of our server, we used multiple simultaneous `curl` requests, and printed a thread's identifier using `pthread_self()` to the console whenever a thread started working. When we initialized the server with 4 threads and sent 4 `curl` requests, 4 different thread identifiers were printed to the console.

To test the redundancy feature of our server, we first did a simple `curl` request with the redundancy flag. Then, we used `diff` to compare the three copies of the file in the `copy1`, `copy2`, and `copy3` folders. For PUT requests, we tested when two copies don't exist, no read/write permission for one of the files, and no read/write permission for two of the files. For GET requests, we tested no read/write permission for one then two copies, deleting one then two copies of a file, and changing the contents for one or two copies. The server should act appropriately based on the identical content in at least two of the files. We also made all three copies have different contents to see if a 500 error was returned. As per what the TA said in Piazza question @336, we can assume the copy folders will all exist and be readable.

5 Assignment Questions

1. *If we do not hold a global lock when creating a new file, what kind of synchronization problem can occur? Describe a scenario of the problem.*

If there is no global lock when creating a new file during a PUT request, then a race condition may occur. A race condition would occur when two threads are trying to create and write to the same file. The operations that both threads do on the new file are not atomic, which means the operations of each thread may become interwoven with each other.

2. *As you increase the number of threads, do you keep getting better performance/scalability indefinitely? Explain why or why not.*

As you increase the number of threads, you do not keep getting better performance and scalability indefinitely. There will be a point where adding more threads will actually do more harm than good. This is because each thread consumes resources and managing all of them does too, resulting in a large increase in overhead. In addition, more threads add to the level of complexity.