Asgn 3 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. There are three additional features: backup, recovery, and list. With the backup command `/b`, the server will backup the current files in the server directory into a folder named `/backup-[timestamp]`. The user can then recover the most recent backup using the command `/r` or specify a timestamp to recover using `/r/[timestamp]`. Lastly, the user can use the command `/l` to return a list of available timestamps. The core design of this server is mostly the same as the one from assignment 1.

First, `httpserver.cpp` will parse the command line arguments and get the specified hostname/ip address and the optional port number. Then, it will initialize the socket using the `socket`, `bind`, and `listen` syscall functions.

There may be multiple connection requests sent by a client, so the server handles them one by one, in the outer `while` loop, using the `accept` syscall function. For each connection, the server will receive (`recv`) requests and/or data from the client until no more bytes are sent. Once no more bytes are sent, the server will move onto the next connection in the queue.

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. If the request is a PUT request, the server will first open a file for writing, then receive and write data sent from the client until the content length is reached or until there are no more bytes sent. If the request is a GET request, the server will check if the given resource name starts with either `/b` for backup, `/r` for recover, `/l` for list, or none of them for a regular GET request.

If it is a backup request, the server will get the Unix timestamp for the current time and create a backup folder with that timestamp. Then, it iterates through all files in the server's directory and for each file, it creates a file with the same name in the backup folder and copies the contents over. A 201 response is sent to the client.

If it is a recover request, the server will first check if the recover command was also given a specific timestamp to recover. If there was no given timestamp, the server iterates through all backup folder names and finds the most recent timestamp. The server then iterates through all files in the backup folder and creates files with the same name in the server directory and copies the contents. If there was a given timestamp, the server iterates through all backup

folder names to find the specified timestamp. If it doesn't exist, the server issues a 404 response. If it does exist, the server iterates through all files in the backup folder and creates files with the same name in the server directory and copies the contents. At the end, a 201 response is sent to the client.

If it is a list request, the server will iterate through all the backup folders and send the timestamps of each backup folder to the client. A 200 response is sent when it is done.

If it is a regular GET request, the server will open the specified file for reading, then reads and sends the data to the client, along with a 200 response.

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.

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 int handle_put(int comm_fd, char buf[], char* resource_name, int content_length)

This function is the same as from the previous assignment and handles PUT requests. It takes in as input the file descriptor for the communication channel, the communication channel buffer, the resource name from the header, and the content length from the header. The function first removes the '/' from the resource name. 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 returns 0 for success.

3.4 int handle_get(int comm_fd, char buf[], char* resource_name, int content_length)

This function is the same as from the previous assignment and gets called to handle GET requests. 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, and the content length from the header. First, it removes the '/' from the resource name. 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, the function checks if there were any problems with opening the file and sends a response based on the `errno` if there was. Next, it creates a buffer of size 16384 and keeps reading to figure out what the content length is of the requested file. Once it has the content length, it sends a 200 response with that content length and reopen the file for sending. The function keeps sending until it reaches the amount of the content length that was counted. Finally, the function closes the opened file and returns 0 for success.

3.5 int handle_backup(int comm_fd, char buf[], char* resource_name, int content_length)

This function handles the case where a GET request has the `/b` backup command and takes the same inputs as `handle_get()`. The function gets the current Unix timestamp and creates a backup folder with that timestamp. Then, it opens the server's directory and iterates through all the files. For each valid file, it opens the file for reading and creates a new file with the same name in the backup folder. The content of the original file gets copied over to the backup file by reading the original file and writing to the backup file until the reading reaches EOF. Files with no read permission are just skipped over and the rest of the files in the directory are added to the backup. For every valid file, once the copying is done, the function closes the original file and backup file. The server's directory that was opened in the beginning gets closed, value 0 is returned on successful completion, and a 201 response is sent to the client.

3.6 int handle_recovery(int comm_fd, char buf[], char* resource_name, int content_length)

This function handles the case where a GET request has the `/r` recover command and takes the same inputs as `handle_get()`. The function will first check if the recover command was also given a specific timestamp to recover. If there was no given timestamp, the function iterates through all backup folder names and finds the most recent timestamp. It then

iterates through all files in the backup folder and creates files with the same name in the server directory and copies the contents. If there was a given timestamp, it iterates through all backup folder names to find the specified timestamp. If the specified timestamp doesn't exist, the server issues a 404 response. If it does exist, the function iterates through all files in the backup folder and creates files with the same name in the server directory and copies the contents. At the end, a 201 response is sent to the client and the value 0 is returned.

3.7 int handle_list(int comm_fd, char buf[], char* resource_name, int content_length)

This function handles the case where a GET request has the `/l` list command and takes the same inputs as `handle_get()`. The function first opens the current directory, goes through each file in the directory (skipping directories), and looks for files beginning with "backup-". For every file that begins with "backup-", it extracts the timestamp to get rid of "backup-" and adds the "\n" character at the end. Then, it adds the number of characters for that string to a variable for content length. Once the function iterates through all the files in the directory, the function closes the opened directory. If the previously opened directory contains no backups, it sends a 200 response with content length: 0 and returns 0 for successful completion. Else, the function sends a 200 response with the content length being the value of the variable used for counting the number of characters. Then, it reopens the current directory and goes through each file. If the file contains "backup-", the function, again, extracts the timestamp and adds the "\n" character at the end. This time, it sends the timestamp to the client. Once all the files in the directory have been iterated through, the function closes the opened directory and returns 0 upon successful completion. Our implementation of this function will list folders with no permissions, which the TA on Piazza post @487 is fine.

3.8 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).

4 Testing

To test the backup functionality, we put some files into the server directory and then issued a backup GET request. We then used `diff` to check if the backed up files had the same content as the ones in the server directory. We also checked if forbidden files got backups. An edge case that we tested was the case where there was a folder in the server directory with a valid name (alphanumeric name length of 10). This folder should not be given a backup because only files should be backed up.

There are two cases of recovery commands we tested: the case where no timestamp is specified, and the case where there was one specified. To test the first case, we created multiple backup folders each with a different timestamp string. Each of the backup folders had files with unique contents. We did a GET request with the `/r` option and used `diff`

to make sure the recovered file is identical to the one in the backup folder. To test the second case, we use the same backup folders for the first test and specify a backup with the `/r/[timestamp]` command and use `diff` to make sure the recovered file is identical to the specified backup folder.

There are a few edge cases that we also accounted for, such as an invalid timestamp supplied by the client. We tested this case by giving an invalid timestamp in the GET request, such as `/r/12345abc`, with an expected response code of 400. Another edge case is where the commands supplied are incorrectly capitalized, such as `/B`, `/R`, and `/L`. The server responds to these requests correctly with a 400 response.

To test the list feature, we used the same backup folders as the previous tests and gave the list command, and compared the returned timestamps to the ones in the server directory. An edge case we tested was if there was a backup folder with an invalid timestamp, such as `backup-12345abc`.

5 Assignment Question

How would this backup/recovery functionality be useful in real-world scenarios?

Backup and recovery is useful for when files get corrupted or deleted (or some other unexpected and unwanted event) and you need to return to a previous 'safer' state in your system. If you schedule your system to create backups often, if you run into a scenario where a necessary file is corrupted or deleted, you can recover the most recent backup and go back to that 'safe' state of the system.