Design Document: httpserver.c

1.0 Introduction

1.1 Goals and objectives

The goal of the program is to build a HTTP server with the two additional features: backups and recovery on the base of the assignment 1 that interacts with the client through curl commands. The server should respond to GET and PUT requests, which means the client can ask the server to send a file named by 10-character ASCII names from client to server or get a file from the server to client. Given the context of the specified request, the program should respond with response headers and appropriate status codes.

Note that the function of backups is to store a backup of all the files in the server as well as the function of recovery is to recover to an earlier backup.

1.2 Statement of scope

The major inputs of the HTTP server are a hostname or an IP address and an optional HTTP port number. The major processing functionality is to handle PUT as well as GET requests.

Besides finishing the requests and outputting appropriate status codes as the assignment 1, the HTTP server has three additional functionalities. It creates a new backup folder when receiving a GET b request. It restores the most recent files when receiving a GET r request, and if a timestamp is provided, it restores the specific one. Lastly, it returns a list of timestamps of all available backups when receiving a GET l request.

2.0 Data design

2.1 Global data structure

Status codes: ERROR -1, FILE_END 0, CREATED 201, OK 200, CLIENT_ERROR 400, SERVER_ERROR 500. BUFFER_SIZE 32768.
int port: for the port number. struct sockaddr_in server_address, struct sockaddr client_address, socklen_t addrlen, int server_socket, socklen_t client_addrlength, int client_socket

char *tokens: for getting the file names, char *get: for the GET request, char *put: for the PUT request, char* putLength: for the content length, char *buffer: for storing the file contents.

2.2 Temporary data structure

char *resourceName: for storing the file names.
int file: for storing the files when using open.
int length: for storing the file length.
char lenStr[12]: for storing the content length.
char *nextBuffer: for storing the files when using read.
DIR *dirp: a DIR pointer defined in the <dirent.h> header, represents a directory stream.
struct dirent *dp: a dirent structure defined in the <dirent.h> header describes a directory entry.
struct stat statbuf: a stat structure containing the fields of a file.
char folderName[256]: for storing the folder name.
char targetFilePath[258]: for storing the path of the target file.
char timestamp[256]: for storing the timestamps.

3.0 System Structure

void http_response(int client_socket, int status_code)

This function constructs the corresponding status codes according to different situations by using if-else statements to test which one is a match. It takes a client socket and the status code as inputs. The status codes are 200 (OK), 201 (Created), 400 (Bad Request), 403 (Forbidden), 404 (Not Found), and 500 (Internal Server Error). If the one of the status codes is matched, the corresponding messages and the content lengths will be outputted.

void handleGet(int socket, char *resourceName)

This function takes a client socket and a literal string of the resource name as inputs.

If the file name is not alphabet or digits, it calls http_response() to give a status code of 400 (Bad Request). If the length of the file name is not 10, it also calls http_response() to give a

status code of 400 (Bad Request). If the file cannot be opened, it calls http_response to give a status code of 403 (Forbidden).

It then creates a struct stat and gets the file length by using fstat() taking a file descriptor of open and the struct stat as well as using the attribute st_size of struct stat. If the file descriptor returned by opening the file is negative and errno is EACCES, it responds with a 403 (Forbidden). If errno is ENOENT, it gives a status code of 404 (Not Found). Otherwise, it gives 500 (Internal Server Error).

If the file is opened successfully, it gives 200 (OK) and uses lseek() to reposition the read file offset and takes a file descriptor, offset and directive as inputs. In this case, the request is GET, so it uses read that takes a file descriptor, a character array, and the number of bytes to read and returns the number of bytes that were read. If the return value is greater than 0, which means it is successful to read the file, it will then use write that takes a file descriptor, a pointer to a buffer, and the number of bytes to write and outputs the number of bytes that were written.

void handlePut(int socket, char *resourceName, char *length)

This function takes a client socket and a string literal of the resource name and the content length as inputs.

If the file name is not alphabet or digits, it calls http_response() to give a status code of 400 (Bad Request). If the length of the file name is not 10, it also calls http_response() to give a status code of 400 (Bad Request). If the file cannot be opened, it calls http_response() to give a status code of 403 (Forbidden).

It then uses atoi() to convert the string literal of the content length to an int and repeats the same process to check if the file can be opened. If the file does not exist and the given the content length is greater than 0, it only stores up to the content length using the same read and write as the handleGet() function. If the content length is not provided, it reads all the contents from the socket and writes to the created file. Finally, it calls http_response() using the 201 (Created) as the input status code.

If the file already exists, we use open to overwrite the file by passing the resource name and O_RDWR | O_TRUNC as arguments. We repeat the same process of checking if the content length is provided or not and handle the request as we did when the file does not exist. Finally, it calls http_response() using the 200 (OK) as the input status code.

int copyFile(const char *fromFilePath, const char *toFilePath)

This function inputs a string literal of the start of the file path and the end of the file path. It first uses open() that takes the start of the file path and O_RDONLY as inputs to open the file. If the file cannot be opened, it calls http_response() to give a status code of 403 (Forbidden). Depending on the return value of opening the toFilePath, we call open() with O_CREAT, O_RDWR and 0666 or O_RDWR and O_TRUNC. If the return value is -1, we pass the first arguments. Otherwise we pass the second ones.

We then use the same read and write as we use handleGet() to read and write the contents of files. Finally, we close the start files and end files with close().

void clearCurrentDirectory()

This function declares a DIR pointer that requires the <dirent.h> header file, a dirent structure defined in the <dirent.h> header describes a directory entry and a stat structure containing the fields of a file.

We use opendir() and pass a string argument that is a name or path to a directory. In this case, we pass one dot as the input. Then we check if opendir() returns a DIR pointer. If the DIR pointer is null, we call http_response() to give a 404(Not Found).

While we repeatedly call readdir() and pass the DIR pointer to check the return value is not null, we use lstat() that takes a name of entry and a stat structure to get the information of files. We use the st_mode field and S_ISDIR to check whether the file type is a directory and call isValidFile() to check the name is valid. If not, we use remove() that inputs the name of the file to delete the file. Finally, we use closedir() that inputs a DIR pointer to close the directory.

void handleBackup(int socket, time_t timestamp)

This function takes a socket file descriptor and a timestamp value of type time_t as inputs. Note we should include <time.h> header. It then first creates two character arrays to hold the folder name and the path of the target file. It also declares a DIR pointer that requires the dirent.h header file, a dirent structure defined in the <dirent.h> header describes a directory entry and a stat structure containing the fields of a file. We use mkdir() and pass the name of the folder and the default file creation permissions 0666 to make a directory.

We use opendir() and pass a string argument that is a name or path to a directory. In this case, we pass one dot as the input. Then we check if opendir() returns a DIR pointer. If the

DIR pointer is null, we call http_response() to give a 404(Not Found). While we repeatedly call readdir() and pass the DIR pointer to check the return value is not null, we use lstat() that takes a name of entry and a stat structure to get the information of files.

We use the st_mode field and S_ISDIR to check whether the file type is a directory and call isValidFile() to check the name is valid. If not, we call sprintf() and pass the path of target file, the folder name and the name of entry as well as call copyFile() to copy the files. Once readdir() returns null, we use closedir() that inputs a DIR pointer to close the directory and call http_response() to give a 200(OK).

void handleRecovery(int socket, const char *timestamp)

This function takes a socket file descriptor and a string literal of timestamp as inputs. It first creates three character arrays to hold the name of the folder, the start of the file path and the end of the file path. It also declares a DIR pointer that requires the <dirent.h> header file, a dirent structure defined in the <dirent.h> header describes a directory entry and a stat structure containing the fields of a file.

We use opendir() and pass a string argument that is a name or path to a directory. In this case, we pass the name of the folder as the input. Then we check if opendir() returns a DIR pointer. If the DIR pointer is null, we call http_response() to give a 404(Not Found). We use access() and pass the folder name and 0 to check if the folder is forbidden. If so, we give a 403(Forbidden).

While we repeatedly call readdir() and pass the DIR pointer to check the return value is not null, we use lstat() that takes a name of entry and a stat structure to get the information of files. We use the st_mode field and S_ISDIR to check whether the file type is a directory and call isValidFile() to check the name is valid. If not, we use increment the int countFile variable by 1.

Once readdir() returns null, we use closedir() that inputs a DIR pointer to close the directory. If the countFile variable is greater than 0, we then can use opendir() and call clearCurrentDirectory(). We repeat the same process of the above while loop. If not, we call copyFile() and pass the start of the file path and the end of the file path. Finally, we use closedir() again and call http_response() to give a 200(OK).

void handleRecoveryRecent(int socket)

This function takes a socket file descriptor as an input. It first creates a character array to hold the timestamps. It also declares a DIR pointer that requires the <dirent.h> header file, a dirent structure defined in the <dirent.h> header describes a directory entry and a stat structure containing the fields of a file.

We use opendir() and pass a string argument that is a name or path to a directory. In this case, we pass one dot as the input. Then we check if opendir() returns a DIR pointer. If the DIR pointer is null, we call http_response() to give a 404(Not Found).

While we repeatedly call readdir() and pass the DIR pointer to check the return value is not null, we use lstat() that takes a name of entry and a stat structure to get the information of files. We use the st_mode field and S_ISDIR to check whether the file type is a directory and strncmp() to check whether the name of entry starts with “backup-”. If so, we use strncmp() to find the most recent backups as well as call handleRecovery() that inputs the socket and the timestamp to copy files from the recent folder to current one with the given timestamp.

void handleList(int socket)

This function takes a socket file descriptor as an input. It first creates a character array and a buff to hold the timestamps. It also declares a DIR pointer that requires the <dirent.h> header file, a dirent structure defined in the <dirent.h> header describes a directory entry and a stat structure containing the fields of a file.

We use opendir() and pass a string argument that is a name or path to a directory. In this case, we pass one dot as the input. Then we check if opendir() returns a DIR pointer. If the DIR pointer is null, we call http_response() to give a 404(Not Found).

While we repeatedly call readdir() and pass the DIR pointer to check the return value is not null, we use lstat() that takes a name of entry and a stat structure to get the information of files. We use the st_mode field and S_ISDIR to check whether the file type is a directory.

Then we close the directory with closedir() inputing a DIR pointer. Finally, we repeat the same process of giving a 200(OK) and the content length as we do in handleGet().

int isValidFile(char *resName)

This function takes a string literal of the resource name as an input and is used to check whether the file name is acceptable by using strlen() inputting the file name to check if the length of the name is exactly 10 and using strspn() inputting the file name and the valid numbers as well as alphabet to check if the name is alphanumeric.

int main(int argc, char** argv)

This function first gets the http server address and the port number. If no port number is provided, the default is 80. It then uses sockets to build the connection between the server and client. First, we use socket system call to create a socket and it takes AF_INET(IP), SOCK_STREAM as well as a 0 as parameters, and it will return a file descriptor for the socket. Then we use bind() system call to identify what address the socket reserves for use, and this function takes a struct container that describes the address, the length of the address as well as the socket as parameters.

The next system call listen() will let the socket listen on the previous address and take the socket and a backlog size as parameters. Accept system call is for creating a new socket for the first connection request of the queue and takes the listening file descriptor and a data structure as parameters. The output is a listening file descriptor for that connection. Finally, we build a connection. Note that we close the socket using the socket as an input of close() system call. If any system call fails, we should print an appropriate message as standard error.

We set up the above system calls in a while loop until the connection is closed. Inside the loop, it uses accept to accept connections and then creates a buffer to for reading the data using read which takes a socket file descriptor, a buffer and the size of the buffer. In order to get the content length of the request, we use strstr() which takes a buffer and the string of the content length and outputs a pointer to the first occurrence of the string and add 15 to get the exact content length.

We then use strtok() to parse the requests and use strncmp() to compare the first three characters of the token to check whether the request is a GET or PUT. If it is neither of them, we call http_response() function to send a sever error. When the request is PUT, we call handlePut() to handle the request as we did on the assignment 1.

When the request is GET, we check the first character after the dash. If it is a “b”, we call handleBackup() to do the backups. If it is a “r”, we call handleRecoveryRecent() to do the recovery. If it is a “r/”, we call handleRecovery() to do the specified recovery. If it is a “l”, we call handleList() to return a list of timestamps. If it is not one of them, we just call

handleGet() to handle the request as we did on the assignment 1. Lastly, we use close that inputs a socket file descriptor to close the socket connection.

4.0 Restrictions, limitations, and constraints

The performance of the HTTP server is similar to the average of the curl server. The program is a simple version of the HTTP server, which only contains GET and PUT commands. Compared with the complete written file, the number of error tests is limited.

5.0 Testing (From Piazza post 479 & 485)

5.1 Backup:

1. GET/b with only the binary in that folder 200/201

2. GET/b with many files in that folder (+30) 200/201

3. GET/b with half of the files that have permission and half files with no permission 200/201

4. GET/b with all files having no permission 200/201

5. GET/b with no files 200/201

5.2 Recovery:

6. GET/r on valid recovery folder 200/201

7. GET/r where no recovery folder exists 404

8. GET/r on recovery folder with no permissions 403

9. GET/r on valid recovery folder but files inside have no permission 200/201

10. GET/r on an empty recovery folder 200/201

11. GET/r on a valid recovery folder where the files already exist in the server folder. (overwrite the server folder files) 200/201

12. GET/r where the files in the recovery folder have full permissions but the same files already exist in the server folder and have no permission 200

5.3 Specific Recovery Folder:

13. GET/r/backup-number on valid recovery folder 200/201

14. GET/r/backup-number on recovery folder that doesn't exist 404

15. GET/r/backup-number on recovery folder with no permissions 403

16. GET/r/ on valid recovery folder but files inside have no permission 200/201

17. GET/r/backup-number where the recovery folder is empty 200/201

18. GET/r/ with no backup folder specified 400

19. GET/r/backup-number on a valid recovery folder where the files already exist in the server folder. (overwrite the server folder files) 200/201

20. GET/r/ where the files in the recovery folder have full permissions but the same files already exist in the server folder and have no permission 200/201

5.4 List:

21. GET/l on a single backup folder 200/201

22. GET/l on many backup folders (+30) 200/201

23. GET/l where no backup exists 200/201

24. GET/l on many backup folders that have no permission 200/201

6.0 Assignment Questions

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

The backup functionality is useful as it provides the convenience of storing the duplicate HTTP server data. For example, we can imagine the possible scenario in the real world that a photographer wants to send a copy of his photos in the HTTP server to his clients. As a result, he can take advantage of the backup functionality to create several backups

in order to send them to different clients. On the other hand, the recovery functionality is also useful as it offers an opportunity of restoring the most recent backup. For instance, we can imagine that the photographer accidently lost his photos. The recovery functionality can help him to find the most recent backup. Moreover, if he has a timestamp, he can retrieve his deleted or lost photos with the time he wants.