Design Doc for Asgn3 Back-Up and Recovery in HTTP Server

Part 1: Struct HttpObject

The HttpObject struct is used to store and maintain the data from the server and the client across all processes and functions. This struct will also store the string value of the backup and recovery files and folders onto a vector. This will be stored in the httpserver.h file:

```
typedef struct HttpObject
{
    int      clientSocket;                         // socket the client is on
    int      statusCode;                           // 200, 201, 400, 403, 404, 500
    char     buffer[BUFFER_SIZE];                  // Header will not be larger than 4 KiB
    char     method[METHOD_SIZE];                  // PUT or GET
    char     fileName[FILENAME_SIZE];              // what is the file we are worried about
    char     httpVersion[HTTPVERSION_SIZE];        // HTTP/1.1 should be the only version
    char     statusText[STATUS_SIZE];              // "OK", "Bad Request", "Not Found", etc..
    char     response[RESPONSE_SIZE];              // Contains the full response send to
client
    bool     hasSentResponse;                      // checks whether the response has been
sent
    long int contentLength;                        // size of the client's file

    // Backup() variables
    std::vector<std::string> vecBackupFiles;       // stores backup file names
    std::vector<std::string> vecBackupFolder;      // stores backup folder names

    // Recovery() and RecoverySpecial() variables
    std::vector<std::string> vecBackups;             // used in
Recovery_GetMostRecentBackup()
    std::vector<std::string> vecRecoveryFiles;       // holds files in recovery folder
    std::vector<std::string> vecRecoverySpecialFiles; // holds files in specific recovery
folder
    std::string              recoveryFolder;         // most recent recovery folder
    std::string              recoverySpecialFolder;  // specific recovery folder

} HttpObject;
```

The data that is in this struct will be reused throughout this assignment to minimize repeating data, we used this struct to organize information.

Part 2: Implement Server Socket in Main

The data that needs to be added to create a successful server and client socket are these:

```
const char* port = (const char*)”80”;
char* address = argv[1]
int enable = 1
```

```
    int serverSocket
    socklen_t clientAddressLength
    struct sockaddr_in serverAddress
    struct sockaddr clientAddress
```

The port will store a specific port number depending on what the user uses as his localhost. The enable variable will help the setsockopt function. The address variable will hold the value of whatever the user wants to call the http server with. The server_sockd will hold the socket function value. We will use the sockaddr_in to hold the values for the server address and the sockaddr to hold the client address that needs to be accessed. There will also need to be a socklen_t size to hold the correct address length for the client socket to be used.

The code will then check to see if the user prompts the correct inputs:

```
    if(argc > 2){port = argv[2];}
    else if (argc == 1 || argc > 3) {print "Usage: httpserver
[][];"}
```

A memset is used to set the server_addr to the size of the server_addr along with other calls from the struct sockaddr_in :

```
    memset(&serverAddress, 0, sizeof(serverAddress))
    serverAddress.sin_family = AF_INET
    serverAddress.sin_port = hton(atoi(port))
    serverAddress.sin_addr.s_addr = H_GetAddress(address)
```

Prior to connecting to the client server there will be checks to make sure certain system calls are used correctly, if not exit the code:

```
    if((server_sockd = socket(...)) <= 0){
        print (error in socket)
        exit code
    }
    if(setsockopt(...) < 0){
        print (error in setsockopt)
        exit code
```

```
}

if(bind(...) < 0){
     print (error in bind)
     exit code
}

if(listen(...) < 0){
     print (error in listen)
     exit code
}
```

Once all checks pass, the next thing is to connect to the client. A while loop will be used to run through the functions that are declared in Part 3. The HttpObject struct will be called within the for loop:

```
while(true){

     HttpObject message;
     message.hasSentResponse = false;

     AcceptConnection(&message, ...)
     ReadHTTPRequest(&message)
     ProcessHTTPRequest(&message)
     ConstructHTTPResponse(&message)
     SendHTTPResponse(&message)

     memset(&serverAddress, 0, sizeof(serverAddress))
     close(message.clientSocket)
}

return 0
```

Part 3: Functions for HTTPServer & Implementation of BackUp and Recovery

Prior to creating the process and execution of the HTTP Server, the identification by the “name” of the address is required for an IPv4 address which is represented in sockaddr_in. We will resolve this in this function:

```
unsigned long H_GetAddress(char* name)
```

In this function, provided by Daniel Alves’s sample code, we will use the addrinfo struct to help contain and access the desired address. The getaddrinfo and freeaddrinfo functions will be used as well:

```
if(getaddrinfo(...) != 0 || info == NULL){
     print error message
     write(...)
     exit(1)
}
```

After checking if there is an address info error, we store the address information in res:

```
unsigned long res
res = ((struct sockaddr_in*) info->ai_addr)->sin_addr.s_addr
freeaddrinfo(info)
return res
```

When the server sockets are all set up the next thing is to connect the client and server together. The entire process and execution of HTTP Server consists of these five high level steps:

1. Accept Connection – creates a connection between the client and the server
2. Read http message – parses through the clients request and checks if it is valid, also checks to see if the GET call is backup, recovery, or list.
3. Process request – stores the information in a struct
4. Construct response – constructs a response message based on the validity of the request
5. Send message – sends the response back to the client

In order to have these information pass accordingly between the client and the server, these 5 functions are organized to do that:

Part 3a: AcceptConnection

```
void AcceptConnection( HttpObject* pMessage, struct sockaddr* clientAddress,
socklen_t* clientAddressLength, int serverSocket);
```

Create a client socket and check for errors.
The system call accept(...) is needed to accept the connection between the client socket and server, which is implemented in this function.

```
        pMessage->clientSocket = accept(serverSocket, clientAddress, clientAddressLength)
```

We check if there is an error with the client_socket when it's value is less than 0.

```
        if(pMessage->client_socket < 0){
                contentlength = 0
                print error message
                return
        }
```

Part 3b: Read HTTP Message with Backup and Recovery Checks

This function will check and store the method name, file name, and http version. If the request is a PUT then it also stores content length.

```
void ReadHTTPRequest( HttpObject* pMessage);
```

The system call recv(...) is used to check if the message is received into the socket. We also make sure that the bytes read in this function is ssize_t.

```
        ssize_t bytesRead = 0

        if((bytesRead = recv(...)) < 0){
                print error (in recv(...))
        }else if (bytesRead == 0){
                print error (client is disconnected)
        }
```

To read the client that was requested, we read the information with the tool sscanf(...) and store them.

```
        sscanf(pMessage->buffer, "%s %s %s", pMessage->method, tempBuffer,
pMessage->httpVersion)
```

With this we parse the information, but we need to check the buffer to see what is being read.

```
if(buffer[0] == '/'){
     move information to the right location
}

if(check if tempBuffer == 10 ascii characters){
     copy filename and the buffer
}else{
     Print error with error code
}
```

Now we check to see the types of GET method calls. If the method has b along with the call, it will run the backup function. If the method has a r along with the call, it will run the recovery function. If the method had r/ along with the call, it will run the specific recovery function. If the method has a l along with the call it will call the list function. These function calls will be further described in Part 5.

```
if(method == GET && b){
     Backup(pMessage)
}

else if(method == GET && r){
     Recovery(pMessage)
}

else if(method == GET && r/){
     SpecialRecovery(pMessage)
}

else if(method == GET && l){
     List(pMessage)
}

else{


     if(Check if the filename has 10 ascii)
          Return filename

     else{
          Send error because filename is not 10
     }

     if(method == PUT){
          Store content length
     }
```

```
        pMessage->buffer[bytesRead] = 0
    }
```

Part 3c: Process Request

This function rescans the stored content to make sure all information is stored correctly and is valid.

```
void ProcessHTTPRequest(HttpObject* pMessage);
```

The information that needs to be checked is if the method name is valid, if the filename is valid, and if the HTTP version is valid.

```
    if(method name is the method name){
        sscanf(error, …, messageStatus)
        print error code
        Send a response to the server
    }

    if(filename is the filename){
        sscanf(error, …, messageStatus)
        print error code
        Send a response to the server
    }

    if(httpversion is the httpversion){
        if( (compare string == httpversion)  !=  0){
            sscanf(error, …, messageStatus)
            print error code
            Send a response to the server
        }
    }
```

Lastly, we look for the GET request:

```
    if(method == GET) {
        if(file exist but empty){
            print success code
            sscanf(good, …, statusMessage)
        }else{
            print error code
```

```
                sscanf(Missing, …, statusMessage)
            }

            sprintf(reponse, Contentlength, httpversion, statusCode,
            statusMessage,  content length)
        }
```

Part 3d: Construct Response

This function will check that if no errors were thrown, a valid status code and status text are set according to the request.

```
void ConstructHTTPResponse( HttpObject* pMessage);
```

The PUT and GET are checked first:

```
        if(method == PUT){
            sscanf(Created, …, statusMessage)
        }else if (method == GET){
            sscanf(OK , ..., statusMessage)
        }else{
            sscanf(Error, …, statusMessage)
            print error code
            write out the response for the error
        }
```

Now we check to see if the file exists and if it does grant access permission to read or write accordingly:

```
        if(fileExists){
            sscanf(OK, …, statusMessage)
            print success code
        }

        if(method == GET){
            ret = access(fileName, file_flag)
            if(file not in directory){
                print error code
                sscanf(Error, …, statusMessage)
                send response to the server
            }else if(file access in wrong area){
                print error code
                sscanf(Error, …, statusMessage)
                send response to the server
            }
        ret = access(fileName, read_flag)

        if(read is not ok){
             print error code
             sscanf(Error, …, statusMessage)
             send response to the server
        }

        ret = access(fileName, write_flag)

        if(write is not ok){
             print error code
             sscanf(Error, …, statusMessage)
             send response to the server
        }
   }
```

Part 3e: Send Response

This function handles the client requests (PUT & GET) and sends the final response as the final step.

```
void SendHTTPResponse(HttpObject* pMessage);
```

To check if there are any errors or problems, we first check the string method is PUT to handle the request:

```
   if(method == PUT){
        open file
        if(file doesn't exist){
             print error
             close file
        }

        if(file is empty){
             print success code
             sscanf(create, …, statusMessage)
             close file
        }
        ...
   }
```

Now we read the buffer and write out what is given from the file. We check to see what is in the file or exit if there is an error:

```
bytesRead = read(...)

if(bytesRead is wrong){
     print error
}

ssize_t bytesWritten = write(...)
ssize_t totalBytes = bytesWritten

while(totalBytes < contentlength){
     if(bytesWritten is wrong){
          print error
     }else{
          bytesRead = read(...)
          bytesWritten = write(...)
          totalBytes = totalBytes + bytesWritten
     }
}

close(file);
```

We check if the request is GET and we do the same thing as PUT but we open the file and just read it:

```
if(method == GET){
     if(file exist){
          print success code
     }

     open file(READ)
     if(file has errors){
          close file
          print error code
     }else{
          ssize_t bytesRead = read(...)
          if(bytesRead has error){
               print error code
          }
          ssize_t totalBytes = 0
          ssize_t bytesWritten = 0
          while(totalBytes < contentlength){
               bytesWritten = write(...)
               bytesRead = read(...)
               totalBytes = totalBytes + bytesWritten
```

```
                }
            }
            close(file)
        }
```

Part 4: Helper Functions for HTTPServer

These helper functions will be used to help organize the 5 main functions needed to run the client and server:

Handles functionality when clients request is GET:

```
void GET( HttpObject* pMessage);
```

This helper function is implemented in `void SendHTTPResponse( HttpObject* pMessage)` to organize the read functions when GET is called in method.

Handles functionality when clients request is PUT:

```
void PUT( HttpObject* pMessage);
```

This helper function is implemented in `void SendHTTPResponse( HttpObject* pMessage)` to organize the read and write functions when PUT is called in method.

Checks the Read and Write access of a file and prints out specific errors.

```
        void H_CheckReadWriteAccess(HttpObject* pMessage)
```

This helper function also calls `H_CheckFilesExists` and `H_CheckMethod` to correctly see if a file has permission or not.

Checks the validity of the client's file name:

```
int H_CheckFileName(char* name);
```

This helper function will loop through the given filename and check to see if it is a viable filename. This is achieved by checking each character in the filename to make sure each character is a legal character.

Checks the validity of the client's file method name. A valid method name will only be PUT or GET:

```
int H_CheckMethod( HttpObject* pMessage);
```

This helper function will compare the string in the method to see if the string value is a PUT or GET. If it is PUT return 1 or if it is GET return 2.

Checks the validity of the clients http version. It should only ever be “HTTP/1.1” or else an error is thrown:

```
void H_CheckHTTPVersion( HttpObject* pMessage);
```

This helper function is implemented in `void ProcessHTTPRequest( HttpObject* pMessage)`. This will compare strings if HTTP/1.1 is part of the version. If it is not it will print out an error code with a bad request.

Calls either PUT() or GET():

```
void H_CallMethod( HttpObject* pMessage);
```

This helper function is called in `void SendHTTPResponse( HttpObject* pMessage)`. This just checks if the string is PUT or GET and calls the respected function. If there is an error, send an error code and a bad request.

Parses through the clients request and stores the content length:

```
void H_StoreContentLength( HttpObject* pMessage);
```

This helper function is called in `void ReadHTTPRequest( HttpObject* pMessage)`. This function uses the strtok() function to help store the content length.

This function is used if a file descriptor throws an error. The HTTP status code error is set automatically based on the errno:

```
void H_SetStatusOnError( HttpObject* pMessage);
```

This helper function uses a switch case to organize the desired HTTP status code errors.

Different from H_StoreContentLength(),This function H_ReadContentLength() analyzes the file pointed to by stat() and returns the size by using .st_size. This is stored in the content length for future reference:

```
void H_ReadContentLength( HttpObject* pMessage);
```

Simply checks if the file exists. Makes use of stdbool.h:

```
bool H_CheckFileExists( HttpObject* pMessage);
```

Sets the response that will eventually be sent to the client. This is called frequently throughout the program:

```
void H_SetResponse( HttpObject* pMessage);
```

Sends the response to the client:

```
void H_SendResponse( HttpObject* pMessage);
```

This helper function uses the send() and write() function to send the desired response in the server. The send() value will be stored in ssize_t while the write() function will be stored in size_t.

Part 5: Implementation of Backup, Recovery, and List

This program will focus on creating a backup folder that saves all the files that were within the main folder with a timestamp at the tail of the backup folder name. If files were deleted from the main folder, calling recovery will bring back the files from the most recent backup and return the files back into the main folder. The list function will list out the amount of backups that were backed up and print them onto the server. All of these functions will be saved on a vector with string variables. The variables that were used from the httpserver struct are:

```
    std::vector<std::string> vecBackupFiles;
    std::vector<std::string> vecBackupFolder;


    std::vector<std::string> vecBackups;
    std::vector<std::string> vecRecoveryFiles;
    std::vector<std::string> vecRecoverySpecialFiles;
    std::string              recoveryFolder;
    std::string              recoverySpecialFolder;
```

Part 5a: Backup Functions

We separated three distinct functions that are needed to recreate the backup command.

```
     void Backup(HttpObject* pMessage)
     void Backup_ScanExistingFiles(HttpObject* pMessage)
     void Backup_CopyFiles(HttpObject* pMessage)
```

The main backup function is Backup(...) and the other functions will be called within this function.

```
void Backup(HttpObject* pMessage){
     Backup_ScanExistingFiles(pMessage)

     ...

}
```

After checking if the files exist, we will create a path variable that will save the backup name with the timestamp.

```
std::string path = "./backup-" + H_GetTimeStamp()
```

Now we create the a backup directory and copy all the files that are in the main folder into this directory:

```
if(!mkdir(path.c_str(), ...){
     pMessage->vecBackupFolder.push_back(path)

     Backup_CopyFiles(pMessage)

     vecBackupFiles.clear()
     vecBackupFolder.clear()

     print response code

}
else{
     print error code
}
```

We need to clear the vector with files and folders to prepare for another backup call if the user wants to make another one. That way the information won't be stacked on top of each other saving the wrong information.

The first backup function that is called in Backup(...) is Backup_ScanExistingFiles. This function will check through a directory and see if all the files are in the main folder and if it has permissions as well. We will be using the DIR and the dirent struct to access the directory.

```
void Backup_ScanExistingFiles(HttpObject* pMessage){
     DIR* directory
     struct dirent* entry
     if((directory = opendir(".")) != NULL){
          while((entry = readdir(directory)) != NULL){
               if(H_IsFile(entry->d_name)){
                    if(files that don't end with "." or ".."){
```

```
                              vecBackFiles.push_back(entry->d_name)
                         }
                    }
               }
          }
     }
     else{
          print error can’t open directory

     }
     closedir(directory)
```

Next we copy all the files in the server directory into the new ./backup-[] folder. We will use a for loop to go through each file and copy them over into the directory. Within the for loop we will check if the file is zero bytes, open the file in the server folder, read from the file, create the new backup file, and then close the files.

```
     void Backup_CopyFiles(HttpObject* pMessage){
          for(i = 0; i < pMessage->vecBackupFiles.size(); i++){
               stat(pMessage->vecBackupFiles.[i].c_str(), &statbuf)

               if(CreateZeroByteFile){continue}
               if((fdServerFile = open(...)) != -1){
                    if(bytesRead = read(...)) != ERROR){
                         fdBackupFile = open(...)

                         if((bytesWritten = write(...)) != ERROR){
                              totalBytesWritten = bytesWritten

                              while(totalBytesWritten < statbuf.st_size){
                                   bytesRead = read(...)
                                   bytesWritten = write(...)
                                   totalBytesWritten += bytesWritten

                                   if(Read or write has error){
                                        close(fdBackupFile)
                                        close(fdServerFile)
                                        continue
                                   }
                              }

                              close(fdBackupFile)
                              close(fdServerFile)
                         }else{
```

```
                                        close(fdBackupFile)
                                        close(fdServerFile)
                                        continue
                                   }

                              }else{
                                   print error
                                   close(fdServerFile)
                                   continue
                              }

                         }else{
                              close(fdServerFile)
                              continue
                         }
                    }
               }
```

Part 5b: Recovery Functions

We separated five distinct functions that are needed to recreate the backup command:

```
     void Recovery(HttpObject* pMessage)
     void Recovery_ScanBackupDirectories(HttpObject* pMessage)
     void Recovery_GetMostRecentBackup(HttpObject* pMessage)
     void Recovery_ScanExistingFiles(HttpObject* pMessage)
     void Recovery_CopyFiles(HttpObject* pMessage)
```

The main recovery function is the Recovery(...). The other functions are embedded within this main function.

```
     void Recovery(HttpObject* pMessage){
          Recovery_ScanBackupDirectories(pMessage)
          ...
     }
```

After we scan all the backup directories, we get the most recent backup data. We then check to see if there is a directory that has been backed up. Afterwards, we check to see if the files within the backup directory have no errors, and if there aren't we recover the copied files into the main folder.

```
if(Recovery_GetMostRecentBackup(pMessage)){
     Recovery_ScanExistingFiles(pMessage)
     Recovery_CopyFiles(pMessage)

     print success status code
}else{
     if not successful
     print error code
}
```

The first recovery function that is called in Recovery(...) is Recovery_ScanBackupDirectories(...). This function checks to see if the backup directory name is a valid directory that can be accessed.

```
void Recovery_ScanBackupDirectories(HttpObject* pMessage){
     DIR* directory
     struct dirent* entry

     if((directory = opendir(“.”)) != NULL){
          while((entry = readdir(directory)) != NULL){
               if(H_IsFolder(entry->d_name)){
                    if(files don’t have “.” or “..”){
                         string temp = entry->d_name
                         if(temp.substr(0,7) == “backup-”){
                              vecBackups.push_back(entry->d_name)
                         }
                    }
               }
          }

     }else{
          print error cannot access directory

     }
     closedir(directory)
}
```

The next function is a boolean that looks at the most recent backup before accessing the backup. A for loop will be used to check all the backups, if there are multiple, and push the most recent one on the vector. There is also a check to see if there are no backups to backup, then return an error code.

```
bool Recovery_GetMostRecentBackup(HttpObject* pMessage){
     string mostRecentBackup = “”
     vector<string> tempBackups
```

```
          for(i = 0; i < pMessage->vecBackups.size(); i++){
                string backupNumber = pMessage->vecBackups[i].substr(...)
                tempBackups.push_back(backupNumber)
          }

          if(!tempBackups.empty()){
                auto maxElement = max_element(begin(tempBackups), end(tempBackups))
                pMessage->recoveryFolder = "./backup-" + *maxElement
                return true
          }
          return false
    }
```

After we access the directory and check to see if they can be accessed, we check the files within the backup to see if they can be recovered.

```
    void Recovery_ScanExistingFiles(HttpObject* pMessage){
          DIR* directory
          struct dirent* entry

          if((directory = opendir(pMessage->recoveryFolder.c_str())) ! = NULL){
                while((entry = readdir(directory)) != NULL){
                      if(H_isFile(entry->d_name)){
                            if(file don't have "." and ".."){
                                  vecRecoveryFiles.push_back(entryd->d_name)
                            }
                      }
                }
          }
          else{
                print error, couldn't access directory
          }
          closedir(directory)
    }
```

When no error arises, we then copy all the files within the backup folder and bring it into the main directory. We loop through each file and copy each filename one by one into the main directory.

```
    void Recovery_CopyFiles(HttpObject* pMessage){
          for(i = 0; i < pMessage->vecRecoveryFiles.size(); i++){
                stat(recoveryFilePath.c_str(), &statbuf)
                if(CreateZeroByteFile(...)){continue}
                if((fdRecoveryFile = open(...)) != -1){
                     if((bytesRead = read(...)) != ERROR){
                          fdBackupFile = open(...)
                          if((bytesWritten = write(...) != ERROR){
                               totalBytesWritten = bytesWritten
                               while(totalBytesWritten < statbuf.st_size(){
                                    bytesRead = read(...)
                                    bytesWritten = write(...)
                                    totalBytesWritten += bytesWritten

                                    if(Read or write have error){
                                         close(fdBackupFile)
                                         close(fdRecoveryFile)
                                         continue
                                    }
                               }

                               close(fdBackupFile)
                               close(fdRecoveryFile)

                          }else{

                               close(fdBackupFile)
                               close(fdRecoveryFile)
                               continue
                          }

                     }else{
                          print error can't read from file
                          close(fdRecoveryFile)
                          continue
                     }

                }else{

                     close(fdRecoveryFile)
                     continue
                }
           }
      }
```

Part 5c: Recover Specific Backups

We separated three distinct functions that are needed to recreate the backup command.

```
      void RecoverySpecial(HttpObject* pMessage)
      void RecoverySpecial_CopyFiles(HttpObject* pMessage)
      void RecoverySpecial_ScanExistingFiles(HttpObject* pMessage)
```

The RecoverySpecial(...) function will get all the backup folder names and store them in the vector vecBackups. It will then search for a specific backup folder and then put the file names in the folder into a vector. After all checks are looked at, the function will then copy all the contents from the backup into the server folder.

```
      void RecoverySpecial(HttpObject* pMessage){
            Recovery_ScanBackupDirectories(pMessage)
            if(find(vecBackups.begin(), vecBackups.end(),
endOfVECTOR){
                  RecoverySpecial_ScanExistingFiles(pMessage)
                  RecoverySpecial_CopyFiles(pMessage)

                  print success status code
            }else{

                  print error code
            }
      }
```

The function `RecoverSpecial_CopyFiles(HttpObject* pMessage)` has the same functionality as `Recovery_CopyFiles(HttpObject* pMessage)` but the vector that holds each file is saved in `vecRecoverySpecialFiles`.

The function `RecoverySpecial_ScanExistingFiles(HttpObject* pMessage)` has the same functionality as `Recovery_ScanExistingFiles(HttpObject* pMessage)` but the vector that holds each file is saved in `vecRecoverySpecialFiles.`

Part 5d: List Backup Functions

There are two list functions that are called:

```
      void List(HttpObject* pMessage)
      std::vector<std::string> List_GetExistingDirectories()
```

Before we go into what the main list function does, we need to see how the list accesses the directories. The function will open the directory to see if it has a valid folder and file to access, then it will push the directory name onto the vector called vecDirectories.

```
      std::vector<std::string> List_GetExistingDirectories(){
        DIR* directory
        struct dirent* entry
        std:vector<std::string> vecDirectories
        if((directory = opendir(“.”)) != NULL){
             while((entry = readdir(directory)) != NULL){
                  if(H_IsFolder(entry->d_name)){
                       if(files don’t have “.” and “..”){
                            string temp = entry->d_name
                            if(temp.substr(0,7) == “backup-”){
                                 vecDirectories.pushBack(entry->d_name)
                            }
                       }
                  }
             }
        }else{
             print error, directory could not be opened
        }

        closedir(directory)
        return vecDirectories
   }
```

The main list function will call the vecDirectories to access them and check to see which http responses are needed to give them the proper response.

```
   void List(HttpObject* pMessage){
        vecDirectories = List_GetExistingDirectories()
        string directoryList = “”
        pMessage->contentLength = 0

        if(!vecDirectories.empty()){
             for(i = 0; i < vecDirectories.size(); i++){
                  output the directories onto the client side
             }

             print success status code
             send(pMessage->clientSocket, directoryList.c_str(),
             strlen(directoryList.c_str()), 0)
        }
        else if(vecDirectories.empty()){
             print no backups found
             print error code
        }
        else{
             print unknown error
```

```
            print server error code
        }
    }
```

Part 5e: Helper functions for Backup, Recovery, and List

Here are some of the helper functions used within these implementations:

This function returns the time stamp for the backups that are created.

```
    std::string H_GetTimeStamp(){
        std::time_t t = std::time(0)
        std::string time = std::to_string(t)
        return time
    }
```

This function will create a zero byte file and will create a new backup file if it hasn't been made yet.

```
    bool H_CreateZeroByteFile(HttpObject* pMessage){
        int fdBackupFile = 0
        struct stat statbuf
        stat(path, &statbuf)

        if(statbuf.st_size == 0){
            fdBackupFile = open(...)
            close(fdBackupFile)
            return true
        }
        return false
    }
```

This function checks to see if a file is in the directory.

```
    bool H_IsFile(const char* filename){
        DIR* directory
        if((directory = opendir(filename)) == NULL){
            closedir(directory)
            return true
        }
        return false
```

```
}
```

This function checks to see if the folder is in the directory.

```
bool H_IsFolder(const char* filename){
      DIR* directory
      if((directory = opendir(filename)) == NULL){
            closedir(directory)
            return false
      }
      return true
}
```

TESTING
Your design document is also where you'll describe the testing you did on your program and answer any short questions the assignment might ask. The testing can be unit testing (testing of individual functions or smaller pieces of the program) or whole-system testing, which involves running your code in particular scenarios. In particular, we want you to describe testing that you did for:

- **Creating multiple backups and recovery to the most recent one or an earlier one.**

We created individual functions in order to handle the GET/b, GET/r, GET/r/, and GET/l requests. This made the code modular and helped to debug each scenario. After testing each one of these functions individually, we then tested them all together by leaving the server running after each request, then alternated each request by going back and forth between a backup request and a recovery request. During the testing phase, we created piazza post #479 with the following test cases and closely followed piazza post #485 to make sure our server responded with the appropriate status codes and text.

Backup

1. GET/b with only the binary in that folder
2. GET/b with many files in that folder (+30)
3. GET/b with half of the files that have permission, half files with no permission
4. GET/b with all files having no permission
5. GET/b with no files (don't think this is possible but it's an idea worth mentioning)

Recovery

6. GET/r on valid recovery folder
7. GET/r where no recovery folder exists
8. GET/r on recovery folder with no permissions
9. GET/r on valid recovery folder but files inside have no permission
10. GET/r on an empty recovery folder (do we erase all the items in the server folder then?)
11. GET/r on a valid recovery folder where the files already exist in the server folder. (overwrite the server folder files)
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

Specific Recovery Folder

13. GET/r/backup-number on valid recovery folder
14. GET/r/backup-number on recovery folder that doesn't exist
15. GET/r/backup-number on recovery folder with no permissions
16. GET/r/ on valid recovery folder but files inside have no permission
17. GET/r/backup-number where the recovery folder is empty (do we erase all the items in the server folder then?)
18. GET/r/ with no backup folder specified
19. GET/r/backup-number on a valid recovery folder where the files already exist in the server folder. (overwrite the server folder files)
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

List

21. GET/l on a single backup folder
22. GET/l on many backup folders (+30)
23. GET/l where no backup exists
24. GET/l on a many backup folders that have no permission

QUESTION
For this assignment, please answer the following question in the design document:

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

This functionality is useful in the real-world because backups make sure that your information will not be permanently lost.

Example 1: Suppose a client finishes writing to file A and decides to make a copy of it on an external hard-drive. For some reason the power to their computer goes out and the next time they try to open file A it is corrupted and can no longer be read or written to. In order to recover the file, all they would have to do is get file A from the external hard drive and copy it to the same location, overwriting the previous file A.

Example 2: Suppose hard-drive A is full of information and is backed up onto hard-drive B. Some time later hard-drive A is lost, stolen, or corrupted. Hard-drive B can be used to recover that information by copying the data back onto hard-drive A or by physically replacing hard-drive A with hard-drive B. This gives the client peace of mind that there is a second option in case something goes wrong.

Because backups provide a sense of security and a certain type of insurance, they are absolutely essential for individuals and businesses who want to protect themselves against any potential loss of data. Like a bank is to cash, backups and recoveries are to computers.