1. Introduction

1.1 Objectives

The program will function as a simple single threaded http server with backup and recovery features. It will respond to GET and PUT commands to read and write files with 10 character ASCII names. It will store files in the same directory as the server. The server will have the ability to store backup copies of its contents and be able to recover from an earlier backup. The program will run on Ubuntu 18.04 installation.

1.2 Constraints

All source files must have *.cpp* suffix and be compiled with no warnings using the flags: `-std=gnu++11 -Wall -Wextra -Wpedantic -Wshadow`. Only standard networking system calls may be used, and no FILE * calls can be used to handle reads/writes.

2. Data Design

2.1 Struct

HttpObject: stores information from incoming requests. Contains the following attributes:

- `char method[5]`: access method of the request, spec only mentions PUT and GET, so length of 5 should be more than enough.
- `char filename[11]`: the name of the file, max length is 10 and it seems like names should be exactly 10 characters. Only alphanumerics may be used.
- `char httpversion[9]`: the version of http being used, default is HTTP/1.1.
- `ssize_t request_header_length`: length of the request header, helps to parse the http request.
- `ssize_t content_length`: length of the file.
- `int status_code`: http status code for the response.
- `uint8_t buffer[]`: the buffer used to transfer files, default to 4KiB.
- `char list[]`: a string used to store all available timestamps when requested by client
- `ssize_t total_len`: size of the data read, helps to parse data being received.
- `char opt`: option setting when backup/recover/list operation is received.
- `int exists`: check if “Content-Length” is present in the header.
- `int client_status:` boolean that checks client’s connectivity.

2.2 Functions

- `void backup( struct httpObject* message )`: takes an httpObject pointer as input, creates a backup folder and populates it with accessible files in the server. HTTP status codes are set accordingly. File I/O uses the httpObject’s buffer. If a folder with the current timestamp already exists, thought this should never happen, a 500 status code will be set.

```
sec = current timestamp
if backup with current timestamp exists: 500 Internal Server Error

create folder in the format of "backup-timestamp".
if folder creation fails: 500 Internal Server Error

for each file in current directory:
    if len(filename) != 10 or filename ==
    "httpserver":
        skip

    check file permissions:
        500 stat() fails
        403 if no permissions
    copy file to backup folder:
        open current file with open(filename, O_RDONLY)
        open backup file with open(fiename,
        O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU)
        if read/write fails: 500 Internal Server Error
        copy file

status_code = 201
```

- `void recovery( struct httpObject* message )`: takes an httpObject pointer as input, looks for the requested timestamp for recovery, attempts to restore the files if the backup folder exists. 403 is set when server has no permission to access the backup folder, 500 occurs when server has no permission to access a file inside that backup folder. 500 status code is set because if the file is backed up, then the server had access permission at that time, and something happened to change the file in the backup folder, thus the recovery fails to restore the files and it is an internal server error.

```
check if client requests a particular timestamp or simply the latest backup,
and set path to the correct folder
check if the folder exists and if server has permission to access the folder
    if DNE or no permission:
        message->status_code = 403
        return;
for each file in backup folder:
    if len(filename) == 10:
        if filename == "httpserver:
                skip
            check file status with stat()
            if stat() fails: 500 Internal Server Error

            restore file from backup folder:
                open file in backup folder with open(filename,
                O_RDONLY)
                open file in the directory same as the server
                executable with open(filename,
                O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU)
                if read/write fails: 500 Internal Server Error, this can
                happen when server has no permissions to access
                the file
                copy file to server directory

    message->status_code = 200
```

- `void list( struct httpObject* message )`: takes an httpObject pointer as input, scans through current directory for backup folders, parse the timestamp from each folder and store it in message->list, which has a max size of BUFFER_SIZE (4KiB)

```
    for each file in current directory:
        check with stat(filename, &st):
        if stat() fails: 500 Internal Server Error

        if( st.st_mode & S_IFDIR ): // if the file is a directory
            if the folder name matches the format
            "backup-timestamp":
                parse the timestamp
                append it to message->list
```

- `void read_http_response` *`( int client_socket_fd, HttpObject *message )`*: reads HTTP request from client. Parse the http header, then extract information and store it in the HttpObject message.

```
    creates a buffer to read the http header.
    recv() from client.
    checks for client_status.
    parse for http method, filename, http version.
    if method is GET and filename is a special request, ie r, b, l, and
    set message->opt accordingly
    check for valid method, filename, and http version, sets status code to 400 if any
    of the three is invalid and returns.
```

```
find Content-Length and mark its existence.
100 continue.
```

- `void process_request( int client_socket_fd, HttpObject *message )`: process the request. Checks for client status, checks status code, then checks for the request type. PUT will create and store the file being uploaded to the server. GET will search for the requested file on the server. 404 Status code will be created if the file does not exist, 403 if the client has no access right to the file. 500 Internal Server Error occurs when read/write or access file fails.

```
if client_status is 0:
    do nothing and skips
if status_code >= 400:
    error occurred, skip to next function
if method is GET:
    check for existing files (404 Not Found)

    if message->opt == b:
        backup( message )
    if message->opt == r:
        recover( message )
    if message->opt == l:
        list( message )

    get the file status from stat()
     if failed: (500 Internal Server Error)
    check for access permissions by trying to open the file with  (fd =
    open(filename, S_IRUSR)) == -1  (403 Forbidden if true)
    If able to access file:
    update message->status code, content_length accordingly
if method is PUT:
    create file with filename, call
    open(...,O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU): flags:
    write only, create it if file not exists, or overwrite if file exists
    If failed: (500 Internal Server Error), skips to next function
    if( message->exists ): // Content-Length is provided in the http header
        while( i < content-length)
            check recv() value from client
            if recv() <= 0:
                set client_status to 0
                skips
            write to file
            if write() failed: 500 Internal Server Error

            i += content written
    else:
        while( i = recv() > 0 ): // keeps receiving until client drops
```

```
                write to file
            /* doesn't need to check client status because writing to file only
            stops when client terminates the connection.
    else: // not a valid request
        status_code = 400
```

- `void construct_http_response( HttpObject *message )`: constructs a http response to send to the client. Stores the constructed message in *message->buffer*.

```
    if client_status is 0:
        do nothing and skips
    allocate string buffer msg
    set msg and message->buffer to 0
    if method is PUT or status_cose >= 400:
        message->content_length = 0
    sprintf() response header to msg
    memcpy( message->buffer, msg, strlen( msg ))
    message->request_header_length = strlen( msg )
```

- `void send_message( int client_socket_fd, HttpObject *message )`: sends the http response to the client. Sends the requested file if successfully processed a GET request.

```
    if client_status is 0:
        do nothing and skips
    send http response header to client:
        send( client_sockd, message->buffer,
        message->request_header_length, 0 )
    if method is GET and status_code < 400:
        open file calls open(), flag: read only
        while( bytes = read() > 0 ):
            write to buffer
            send buffer to client

            /* we don't check for failed sends because the http header is
            already sent, there is no way to retract and change the status
            code
            */
```

2.2.1 Helpers

- `char *message_for_code( int status_code )`: returns corresponding message for the given http status code. Codes implemented: 200, 201, 400, 403, 404, 500. Default is 100 Continue.

```
        switch( status_code )
```

- `unsigned long getaddr( char* name )`: returns the numerical representation of the address identified by *name*. Code from example code from section.

- `ssize_t next_line_index( char* buffer, ssize_t start, ssize_t total_len)`: finds the line break in the buffer, and returns the index of the starting position of the next line.

```
for i in range( start, total_len - 1 ):
    if character in buffer[ i ] == '\r' and the next char is '\n'
        break
i += 2 // need to move 2 positions
return i
```

- `int is_al_num( char* name )`: returns 1 if all characters in name are alphanumeric, else 0.

```
int r = 1 // boolean if the string only contains alnum, default to 1,
for char in name:
    if char is not alnum:
        r = 0
return r
```

2.3 Constants

- BUFFER_SIZE: 4 KiB. The http header being received will be no longer than 4 KiB, so it might as well be the buffer size.

3. Structure

Check for the number of arguments, exit if less than 2.
Check for the user given port, default to 80 (80 requires admin, otherwise *"bind() Permission denied" error)*.

Configure socket:

```
int server_sockd = socket(AF_INET, SOCK_STREAM, 0)
    error if server_sockd < 0
hostname/ip(provided by user): sin_addr.s_addr = getaddr( argv[1] )
port (optional): sin_port = htons( port )

// avoid: 'Bind: Address Already in Use' error
```

```
    int enable = 1;
    int ret = setsockopt(server_sockd, SOL_SOCKET,
SO_REUSEADDR, &enable, sizeof(enable));
```

ret = bind(server_sockd, (struct sockaddr *) &server_addr, adddrlen) : bind server address to socket address

ret = listen(server_sockd, 5) : listen for connection

// structs and data needed for comms

```
struct soddaddr client_addr
Socklen_t client_addrlen
struct httpObject message

while( true ): // start server, keeps it running
    // connects to client socket
    client_sockd = accept(server_sockd, &client_addr, &client_addrlen)

    check for successful accept, warn and skips if unsuccessful

    while( message.client_status > 0 ): // while the connection is active, ie the
    client didn't terminate the connection

        read_http_response() // reads http header from client request
        process_request() // process the request depending on header
        construct_http_response() // construct response header
        send_message() // send header, send file if request is GET

    close client socket
```

4. Tests

Tests were done in units and as a whole. HTTP requests were send to server during development to test the outcome of each function.
Tests done for program:

FILES:
- Binary files: client must include --output <filename> in their curl command, otherwise it crashes the server.
- Text file: works
- Big text file (6 MB): works
- Empty file:works
- Plain text with only a space:works

Curl:
- curl http://localhost:8080/ ; // server hangs until client stops, sends 400 Bad request.
- Invalid filename: works
- File that does not exists: works

Backup: backup request is send with the following scenarios in the server
- multiple files in the server
- multiple files but some with no access permission
- no files in the server

Recover: recover request is send with the following scenarios in the server

- multiple backups exists on the server
- no backup exists on the server
- a backup folder with no access permission
- a backup folder which contains file with no access permission
- backup contains file that already exists in the server folder
- backup contains file no longer exists in the server folder
- the request specifies a timestamp to back up
- the request specifies a timestamp but the folder does not exists

List: list request is send with the following scenarios in the server

- only one backup folder exists
- multiple backup folder exist
- no backup exists

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

Answer: In real world, accidents might happen where files are deleted, bad files created, files become corrupted, or the storage becomes too messy and no one knows what is going on. A backup would allow the server to restore to a previous point in time where things perhaps were more manageable. It’s like git, where if our newer implementations created more bugs, we can always revert to an older functional version.