Project 1: Socket Programming

Introduction to Data Communication Networks (2023 Fall)

Instructor: Prof. Kyugnhan Lee

TAs: Jongseok Park, Dongsu Kwak

Deadline: November 2nd, 2023

Project Overview

In this project, we will be implementing two common networked applications using socket programming. The first will be an **HTTP server**, and the other will be a **BitTorrent-like P2P file sharing** (abbr. torrent) application.

In this document, we will first give you a general introduction to both applications. We will then give details on what behavior you should expect from your implementation, and what you will have to implement to complete the applications. Make sure to read them all carefully.

We included a **TA implementation** of both the HTTP server and the torrent application so you could compare them to your implementation and check if everything is working properly. We also included a generous number of hints in the skeleton code.

Details on grading and submission will be presented in the "Grading" and "Submission" sections separately.

Application 1: HTTP server

A. Introduction to HTTP server

The Hypertext Transfer Protocol, or HTTP, is used to transfer web page elements such as HTML files or CSS files across the Internet. HTTP clients are usually web browsers, such as Chrome, Firefox, or Safari. HTTP is the protocol that these clients (web browsers) communicate with web servers, such as Apache or NginX to retrieve web pages from the internet. We will use HTTP 1.0 for our project.

HTTP communication is started by the client connecting to the web server using TCP. The port number for HTTP server is usually 80, but in our project, we will be using port **62123**, as port 80 is usually blocked by ISPs / OS in many environments. You can specify which IP and port you are connecting to by entering the following URL in your web browser:

<div align="center">http://<IP>:<PORT>/</div>
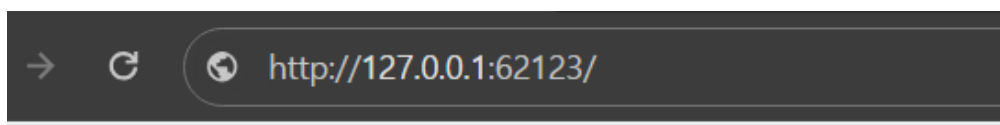
You can run the included TA binary and check out the expected result of your HTTP

server, using the following steps. We assume that you have already extracted the project files inside your development environment and have VS code with a terminal open in the development directory.

1. Grant execution permission on the included binaries.

chmod +x ./http_server*

2. Run the appropriate binary depending on your system.

./http_server_<YOUR_SYSTEM> 62123

```
cakeng@optiplexmicro:~/DCN_2023/project_1_http_server$ ./http_server_linux 62123
Initializing HTTP server...
```

3. Connect to the server by entering http://127.0.0.1:62123/ as the URL on your favorite web browser. You should be greeted with the following web page.
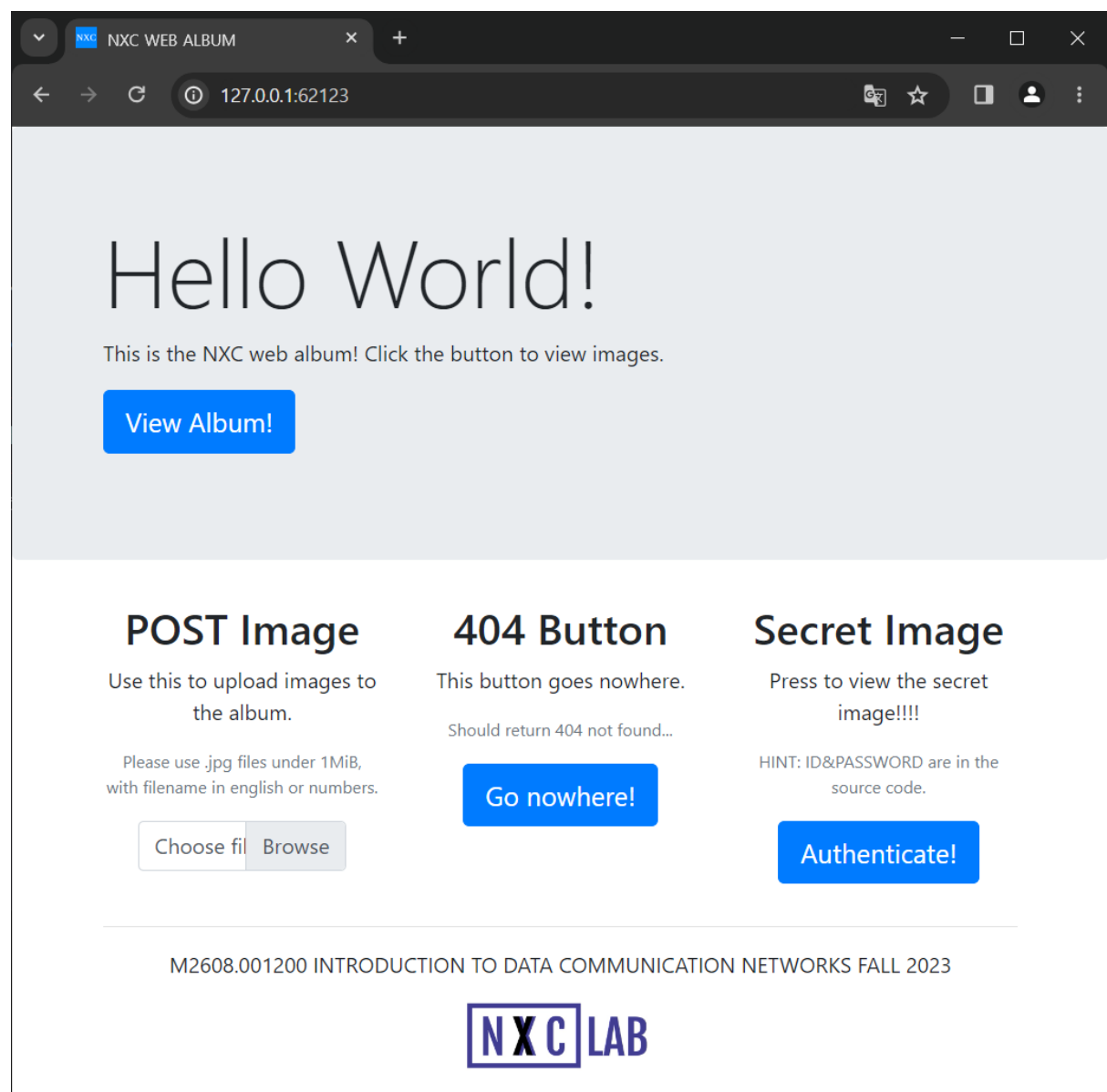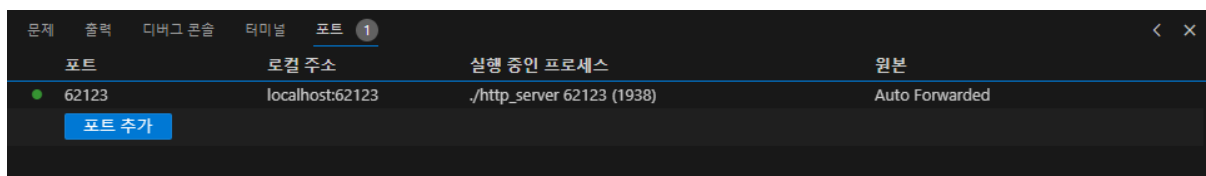
ATTENTION! – WINDOWS USERS!

To connect to your web server inside WSL from a browser in Windows, you must **forward your port** using VS code. VS code may auto-forward the port when you run the server application, but if it does not, just follow these steps.

1. Press on the "Port" tab next to the "Terminal" tab in your VS code. Refer to Project 0 on getting to the "Terminal" tab open.

2. Click on the blue "Forward a Port" button.

3. Type 62123 in the textbox and press "Enter".

You have successfully forwarded your port if your "Port" tab lists are as below.

문제 출력 디버그 콘솔 터미널 **포트** ①			‹ ✕
포트	로컬 주소	실행 중인 프로세스	원본
● 62123	localhost:62123	./http_server 62123 (1938)	Auto Forwarded
포트 추가			

(You will learn what port forwarding is further down the class when you learn about network address translation or NAT.)

B. HTTP Server Program Behavior

Use your web browser to check the behavior of the HTTP server.

1. When you click on the "View Album!" button, it should take you to a **web album** with 12 images.

2. If you click on the "Browse" button on the "POST Image" section, you should be able to upload a **.jpg image under 1 MB with a filename in English or numbers** to the web album. Try it yourself and check the web album to see if it works.

3. The "Go nowhere!" button will simply return a "404 Not Found" error.

4. The "Authenticate!" button should create a pop-up asking to enter login credentials. Enter **DCN** for the username and **FALL2023** for the password. The page should show a secret image.

If any of the above does not work, please contact the TAs using the Q&A board.

## C. Implementation Objectives

Your server will 1. create a listening socket and accept incoming connection from web browsers, 2. receive and parse the HTTP request, 3. take appropriate action based on the request, 4. create an HTTP response based on the actions it took, and 5. return the HTTP response back to the web browser. Steps 2 ~ 5 will be repeated indefinitely. **You only have to modify the "http_engine.c" file for your implementation.**

1. Creating a Listening Socket and Accepting Connections

The function server_engine() in http_engine.c is called by the main function when starting your server. Initialize a TCP socket and bind it to the server_port argument. Inside an infinite loop, accept incoming connections and serve the connections using the server_routine () function.

2. Receiving and Parsing HTTP Requests

HTTP requests take the following form. Refer to the class materials for details.
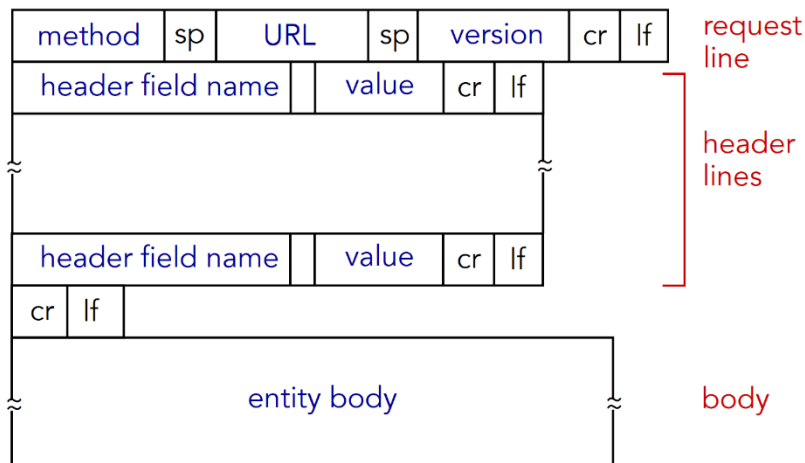
"sp" refers to the space character (' '), "cr" refers to the carriage return character ('\r'), and "lf" returns to the line feed character ('\n').

Inside the server_routine function, a header_buffer character array and an infinite loop is presented. Receive the HTTP header inside the loop, until 1. an end of header delimiter is received (i.e., \r\n\r\n) 2. An error occurs or the client disconnects, or 3. the header message from the client is too long.

After receiving the header successfully, you will have to parse the message to know what the client is requesting. That is, you must extract the method, URL, and header field name & header field value tuples from the received header string.

We recommend you implement the parse_http_header () function, which converts the header string into a http_t struct. http_t struct is a C struct that we created to help you easily manage and manipulate various HTTP elements, using the various helper functions provided in the http_util.c. However, implementing parse_http_header () is **NOT** necessary, and you can parse the header in any way you like.

3. Take appropriate action

The requests from the web browser will be either a GET or a POST method. GET methods are used to retrieve web page elements, such as HTML files or CSS files from the web server. POST methods are used to upload data from the client to the HTTP server, such as user images.

On a GET method, you must return the file requested by the URL as the HTTP response. The body of the GET method HTTP request will be empty.

On a POST method, you must retrieve the file from the HTTP request's body, save it as a file in the server, and update the HTML file to correctly display the newly uploaded file in the web album.

The cases that you must support for the expected server functionalities are documented in the http_engine.c as comments. Please refer to them for details.

(Uploading using GET will not be covered by this project.)

4. Creating HTTP responses

Depending on the client request, and the actions you took, there will be different HTTP responses that should be sent back to your web browser. Again, using the included http_t struct and the helper functions will make things much easier for you. You simply have to initialize a response with the init_http_with_arg () function with the appropriate response code and append the right header fields and body to the response.

An example of using the http_t struct and its functions to create an HTTP response is included in the http_engine.c file, on sending a "431 Header too large" response.

5. Returning the HTTP response

When your HTTP response is ready, you must send the response back to your client. The response message must be formatted according to the HTTP protocol standards, following a similar structure to an HTTP request message, but with a status line instead of a request line.

status line
(protocol
status code
status phrase)

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

header
lines

data, e.g.,
requested
HTML file

Fortunately, **we have implemented** the formatting and sending of the HTTP response for you. It will automatically format an http_t struct into a correct HTTP response

message and send it back to your web browser. Checking out the write_http_to_buffer () function may help you better understand the formatting of an HTTP response message and may also help you implement the parse_http_header () function.

D. Important Notes

To build your server, simply enter "**make**" and hit enter. Make sure to "make" after saving changes to your source code to run the updated binary. Use ctrl+c to exit the server program.

Also, **do not change** the function return values, function name, and the function arguments of any of the defined functions.

You can modify other files and functions for testing purposes, **but you will only submit the "http_engine.c" file.** Therefore, the changes you make on other files will be lost in your submissions.

You are **allowed to define** more functions, global variables, enums, structs, etc. inside "http_engine.c" if you need to. You are also allowed to use additional libraries if it is included in the C / POSIX standard**.** External libraries that require installation will **NOT** be supported in your grading environment.

Please read the comments in the "http_engine.c" closely, as they contain guides and useful hints on implementing your project. Also, please make good use of the included http_t struct and its helper functions, as it will make the management and manipulation of HTTP elements much easier. We **strongly recommend reading the descriptions and definitions of http_t struct and its functions in http_functions.h and http_util.c** before starting the project.

Also, the "**Web Developer Tools**" of your web browser can help you debug your server. Its "Network" tab allows you to see the HTTP request message and the HTTP response messages exchanged between your server and the web browser.
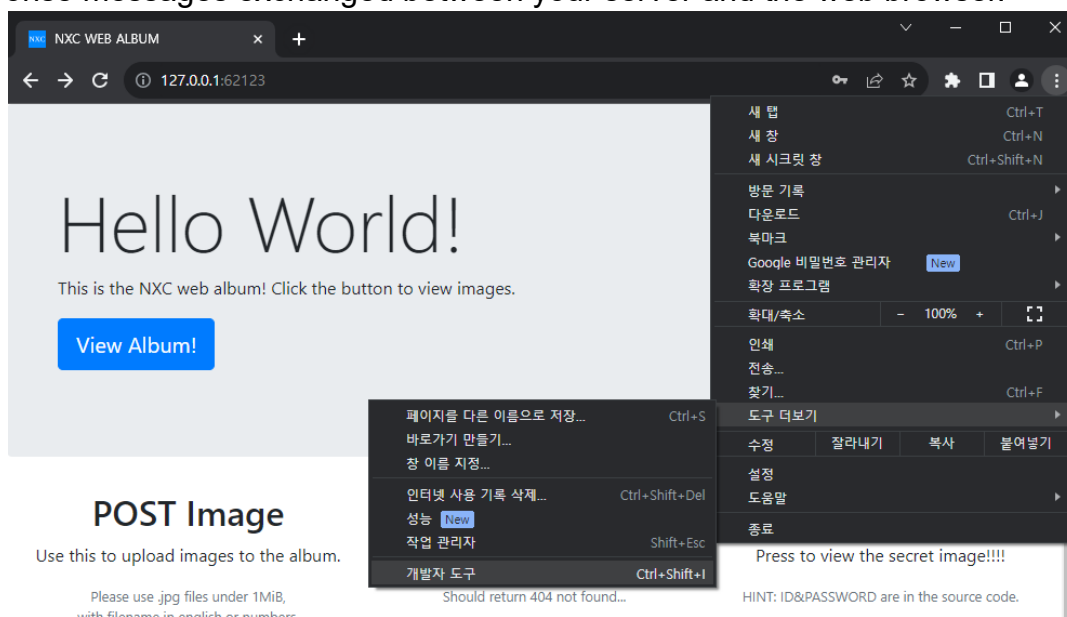
We will not grade for memory management. However, **understanding C pointers and memory structures will help you greatly in this project**, as this project requires a tremendous amount of dynamic memory allocations, of C string parsing, and reading/writing from/to sockets and files. We strongly recommend getting a general understanding of C pointers and memory structure before starting the project.

Front-end web page elements, such as HTML, CSS, or JavaScript, are also an important part of the web. However, the web front-end is not included in the scope of this project. We provide all front-end elements for this project, and you do not have to modify any of these elements.

Here are some documents that may help you in this project.

HTTP Overview: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

HTTP GET: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET

HTTP POST: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

HTTP Response Codes: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

HTTP Headers: https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

C strings: https://www.geeksforgeeks.org/strings-in-c/

C string functions: https://www.geeksforgeeks.org/c-library-string-h/

C pointers: https://www.geeksforgeeks.org/c-pointers/
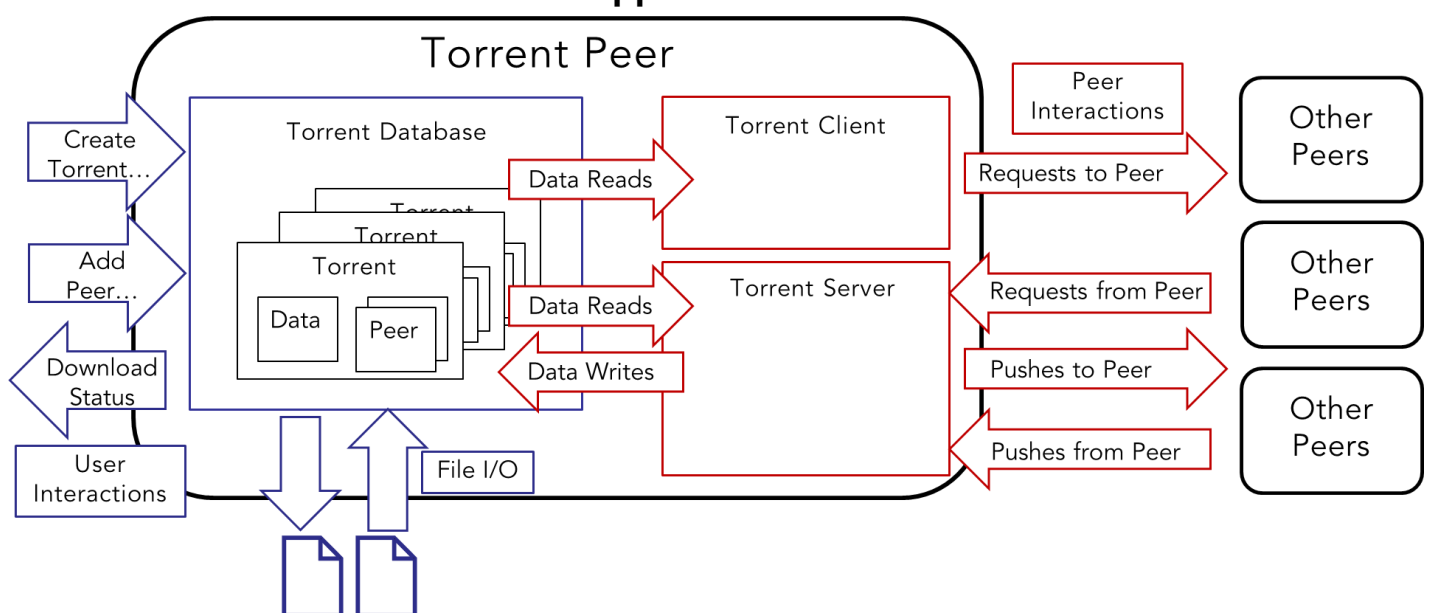
Application 2: BitTorrent-like P2P file sharing

A. Introduction to Torrent Application

BitTorrent is a great example of a Peer-to-Peer networking application. Compared to

the traditional server-client-based file exchange, it allows faster file sharing and better scalability of the system through distributed networking. For our torrent application, we will use the above program structure. You will implement the red elements.

Each torrent application will have a list of torrents (Torrent Database), a client function, and a server function.

Each torrent will split its file into 32-kilobyte blocks. The client function will iterate through all torrents in the database and perform the required actions to download the torrent blocks, such as requesting block data from a peer that is known to have that block.

The server function will receive messages from remote peers, and take appropriate actions based on the commands of the connections, like that of the HTTP server. There are two different types of messages that the server must serve: a request message and a push message.

A request message is sent from a client function of a remote peer and will request certain torrent elements. There are **four different torrent elements** that a remote peer can request: **1. torrent information** (such as filename, file size, etc.), **2. list of known peers for that torrent**, **3. current download status of the peer**, **4. block data**. When a request message is received, the server function will call an appropriate handler for that message, which will return the requested element using the corresponding push message.

A push message is sent from a server function of a remote peer and will have the torrent element that the client function has requested. There can be four different torrent elements that a remote peer can push, each corresponding to the four elements of the request messages. When a push message is received, the server function will also call an appropriate handler for that message, which will update its torrent database to save the pushed element.

Our application will manage torrents based on a **hash value**. When a torrent is created from a file, it will be given a unique hash value. Users who wish to download that file can add that torrent using the hash value. When a peer is added to a torrent, it will automatically send request messages for torrent elements that need to be updated.

Therefore, message exchanges in our torrent application will have the following order.

1. The torrent client finds that a torrent needs a certain torrent element to be updated from a remote peer and sends a request message for that element to the remote peer.

2. The server client of the remote peer receives the request message, handles the request message, and responds with a matching push message that contains the requested torrent element.

3. The server client of the requesting peer receives the push message from the remote peer, handles the push message, and updates the torrent database to save the pushed torrent element.

You can run the included TA binary and check out the expected result of your

application, using the following steps. We assume that you have already extracted the project files inside your development environment and have VS code with a terminal open in the development directory.

1. Grant execution permission on the included binaries.

<div align="center">chmod +x ./torrent*</div>

2. Run the appropriate binary depending on your system.

<div align="center">./torrent_<YOUR_SYSTEM> 62123</div>

```
cakeng@RZN3600-WIN:~/project_1/project_2_torrent_server$ ./torrent_linux 62123
  Initializing torrent engine...
  ENTER COMMAND ("help" for help): █
```

Use help to see the supported commands. For example, entering "create music_torrent.mp3 music.mp3" will create a torrent from the included music file.

```
ENTER COMMAND ("help" for help): create music_torrent.mp3 music.mp3
TORRENT 0x353f137d (music_torrent.mp3) CREATED.
```

Entering "status" will show the torrents in the database and their download status.

```
ENTER COMMAND ("help" for help): status
PRINTING ENGINE STATUS:
        ENGINE RUNNING - ENGINE HASH 0xeee9cf5d, PORT: 62123, NUM_TORRENTS: 1
        TORRENT 0:
                NAME: music_torrent.mp3, STATUS: COMPLETED
                HASH: 0x353f137d, SIZE: 3.51 MiB, BLOCKS: 113/113, NUM PEERS: 0, SPEED: 0.00 KiB/s
```

Entering "info <IDX>" will show the detailed information of the torrent with index <IDX>.

```
ENTER COMMAND ("help" for help): info 0
PRINTING TORRENT STATUS:
        NAME: music_torrent.mp3, STATUS: COMPLETED
                HASH: 0x353f137d, SIZE: 3.51 MiB, BLOCKS: 113/113, NUM PEERS: 0, SPEED: 0.00 KiB/s
                BLOCK STATUS:
                        0    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                        16   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                        32   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                                            ...
                        80   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                        96   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                        112  0
```

To add a torrent based on hash, open another terminal using the "+' button on the VS code and run another torrent peer with a different port number.

```
문제    출력    디버그 콘솔    터미널    포트 ②                              [+]    ./torrent_linux - project_1_torrent_server  ⊟

ENTER COMMAND ("help" for help): info 0
PRINTING TORRENT STATUS:
        NAME: music_torrent.mp3, STATUS: COMPLETED
                HASH: 0x353f137d, SIZE: 3.51 MiB, BLOCKS: 113/113, NUM PEERS: 0, SPEED: 3616.00 KiB/s
                BLOCK STATUS:
                        0    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                        16   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                        32   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                                            ...
                        80   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                        96   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                        112  0

ENTER COMMAND ("help" for help): █
cakeng@RZN3600-WIN:~/project_1/project_1_torrent_server$ ./torrent_linux 62124
  Initializing torrent engine...
```

Enter "add <HASH>" to add torrent based on hash. Running the "status" command will return that it has no information on the torrent.

```
ENTER COMMAND ("help" for help): add 0x353f137d

ENTER COMMAND ("help" for help): info 0
PRINTING TORRENT STATUS:
        NAME: , STATUS: NO INFO
                HASH: 0x353f137d, SIZE: 0 B, BLOCKS: 0/0, NUM PEERS: 0, SPEED: 0.00 KiB/s
```

Add peer using "add_peer <IDX> <IP> <PORT>". Adding the original peer will make the application automatically retrieve the torrent info and start downloading.

```
ENTER COMMAND ("help" for help): add_peer 0 127.0.0.1 62123
COMMAND: add_peer 0 127.0.0.1 62123
PEER 127.0.0.1:62123 ADDED to 0x353f137d.

ENTER COMMAND ("help" for help): info 0
PRINTING TORRENT STATUS:
        NAME: music_torrent.mp3, STATUS: DOWNLOADING
                HASH: 0x353f137d, SIZE: 3.51 MiB, BLOCKS: 8/113, NUM PEERS: 1, SPEED: 256.00 KiB/s
                BLOCK STATUS:
                    0   X  X  X  X  X  X  O  X  X  X  X  X  X  X  X  X
                    16  X  X  X  X  X  O  X  X  X  X  X  X  X  X  X  X
                    32  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
                                            ...
                    80  X  X  X  X  X  O  X  X  X  X  X  O  O  X  X  X
                    96  O  X  X  X  X  X  X  X  X  X  X  X  X  X  X  O
                    112 O
        PEER STATUS:
                PEER 0:
                    IP: 127.0.0.1, PORT: 62123
                    PEER BLOCK STATUS:
                    0   O  O  O  O  O  O  O  O  O  O  O  O  O  O  O  O
                    16  O  O  O  O  O  O  O  O  O  O  O  O  O  O  O  O
                    32  O  O  O  O  O  O  O  O  O  O  O  O  O  O  O  O
                                            ...
                    80  O  O  O  O  O  O  O  O  O  O  O  O  O  O  O  O
                    96  O  O  O  O  O  O  O  O  O  O  O  O  O  O  O  O
                    112 O
```

You can use "watch <IDX>" to watch the automatically refreshed printing of torrent info. You can also use "watch" without an index to watch the automatically refreshed printing of the database status. To exit the watch mode, press enter.

B. Torrent Application Behavior

We have included scripts to automatically test the behavior of our application. Open two terminals, and enter "cat ./script_1.txt - | ./torrent_linux 62123" on one terminal, and "cat ./script_2.txt - | ./torrent_linux 62124" on the other. The two peers should automatically enter watch mode, with 4 torrents each, and exchange the file with one another.

After all files are downloaded by the torrents, enter the "torrents" directory. Under the hash name directory of each engine, check if any of the downloaded files are broken.
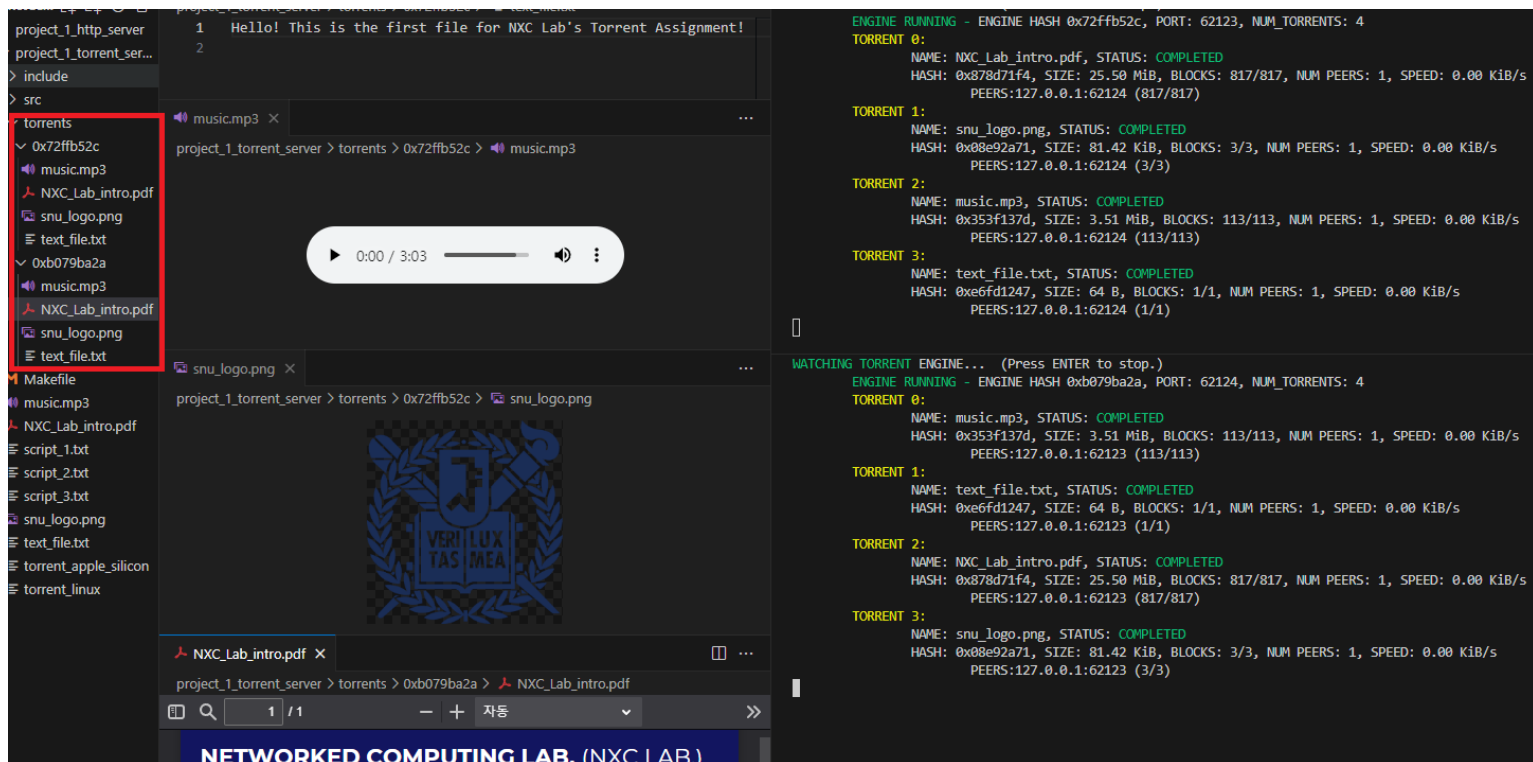
NOTE: The PDF file may look broken if viewed inside VS code. If so, please check it again using an external PDF reader.

If any of the above does not work, please contact the TAs using the Q&A board.

C. Implementation Objective

1. Functions

You will have to implement three socket functions (listen, accept, connect), the client function, the server function, the four request functions, the four post functions, and the eight handler functions for handling each of the request and post messages.

a. The listen function will open a listen port that the server function will use to receive incoming connections from a remote peer. **The accept function** will be used by the server function to accept the incoming connections, and **the connect function** will be used by the four request functions and the four post functions to connect to the server function of the remote peer. The accept and the connect function must support a **timeout** in case the remote peers are unresponsive.

b. The client function will iterate through the torrents in the database and send request messages to remote peers, using the four request functions each corresponding to the four torrent elements. For every torrent in the database, it will iterate through all peers of the torrent and send four different request messages for torrent elements.

1. If the torrent is missing **torrent info**, it will request torrent info with a certain time interval.

2. If the torrent info exists, it will request the known **peer list** of the peer with a

certain time interval.

3. If the torrent info exists, it will request the **block status** of the peer with a certain time interval.

4. If the torrent info exists, it will randomly select a **block data** that is missing from itself but exists on the peer, and request it to the peer with a certain time interval.

c. The server function will use the accept socket functions to receive messages from remote peers. It will parse the common arguments of the messages, to check what kind of message it is (message command), which peer it is from (peer's engine hash), which port the peer is listening on (peer's listening port), and which torrent the message is targeting (torrent hash). It will then hand the remaining portions of the message by to the eight handler functions each corresponding to the eight types of the messages.

d. The four handler functions for request messages will receive the torrent element data request from the four different request messages from a remote peer. It will check if the message is valid and respond by sending a post message to the requesting peer, using the four post functions each corresponding to the four torrent elements.

e. The four handler functions for post messages will receive the torrent element data contained in the four different post messages from a remote peer. It will check if the message is valid and update the torrent database with the received torrent element data.

f. The four request functions and the four post functions will use the connect socket functions to connect to the server function of the remote peer and send either a request or a post message with the appropriate message protocol.

Request, push, and the two handler functions for the **torrent info** torrent element are **already implemented for you** to serve as an example of how to handle and format the messages. **You only have to modify the "torrent_engine.c" file for your implementation.**

2. Message Protocol

Our torrent application will use a total of 8 different messages.

a. Torrent info request message.

This message will be sent by the client function and will request torrent info from a remote peer. The message will have the following protocol.

```
REQUEST_TORRENT_INFO [MY_ENGINE_HASH] [MY_LISTEN_PORT] [TORRENT_HASH]
```
"REQUEST_TORRENT_INFO" is a fixed string, and the rest of the arguments enclosed in [ ] are variables. [MY_ENGINE_HASH] refers to the requesting torrent's engine hash, [MY_LISTEN_PORT] refers to the requesting torrent's listening port, and [TORRENT_HASH] refers to the hash of the torrent whose info is being requested.

The message will be contained in a fixed-length string of MSG_LEN bytes, and the unused portion of the message after the [TORRENT_HASH] argument will be padded with 0 ('\0', not char '0').

Please refer to the implementation example in torrent_engine.c for more details.

b. Torrent info push message.

This message will be sent by the handler function of the request message and will push torrent info to a remote peer. The message will have the following protocol.

```
PUSH_TORRENT_INFO [MY_ENGINE_HASH] [MY_LISTEN_PORT] [TORRENT_HASH] [TORRENT_NAME] [FILE_SIZE] [BLOCK_HASH]
```

"PUSH_TORRENT_INFO" is a fixed string, and the rest of the arguments enclosed in [ ] are variables. [MY_ENGINE_HASH] refers to the pushing torrent's engine hash, [MY_LISTEN_PORT] refers to the pushing torrent's listening port, and [TORRENT_HASH] refers to the hash of the torrent whose info is being pushed. [TORRENT_NAME] is the name of the given torrent. [FILE_SIZE] is the file size of the given torrent. [BLOCK_HASH] is the block hash array of the given torrent.

The message up through [FILE_SIZE] will be contained in a fixed-length string of MSG_LEN bytes, and the unused portion of the message after the [FILE_SIZE] argument will be padded with 0 ('\0', not char '0').

After MSG_LEN bytes, **[BLOCK_HASH] will be appended to the message as a binary**. That is, all data contained in the block_hash memory location of the given torrent will be appended to the message as a binary dump, without translation to a string format.

Please refer to the implementation example in torrent_engine.c for more details.

c. Torrent peer list request message.

This message will be sent by the client function and will request a list of known peers from a remote peer. The message will have the following protocol.

```
REQUEST_TORRENT_PEER_LIST [MY_ENGINE_HASH] [MY_LISTEN_PORT] [TORRENT_HASH]
```

"REQUEST_TORRENT_PEER_LIST" is a fixed string, and the rest of the arguments enclosed in [ ] are variables. [MY_ENGINE_HASH] refers to the requesting torrent's engine hash, [MY_LISTEN_PORT] refers to the requesting torrent's listening port, and [TORRENT_HASH] refers to the hash of the torrent whose peer list is being requested.

The message will be contained in a fixed-length string of MSG_LEN bytes, and the unused portion of the message after the [TORRENT_HASH] argument will be padded with 0 ('\0', not char '0').

d. Torrent peer list push message.

This message will be sent by the handler function of the request message and will

push a list of known peers to a remote peer. The message will have the following

```
PUSH_TORRENT_PEER_LIST [MY_ENGINE_HASH] [MY_LISTEN_PORT] [TORRENT_HASH] [NUM_PEERS] [PEER_0_IP]:[PEER_0_PORT] [PEER_1_IP]:[PEER_1_PORT] ...
```

protocol.

"PUSH_TORRENT_PEER_LIST" is a fixed string, and the rest of the arguments enclosed in [ ] are variables. [MY_ENGINE_HASH] refers to the pushing torrent's engine hash, [MY_LISTEN_PORT] refers to the pushing torrent's listening port, and [TORRENT_HASH] refers to the hash of the torrent whose peer list is being pushed. [NUM_PEERS] is the number of peers in the list being pushed, and [PEER_N_IP]:[PEER_N_PORT] is the IP&port tuple of n-th peer in the list. (ex. 127.0.0.1:62123 127.0.0.1:62124 …)

The message up through [NUM_PEERS] will be contained in a fixed-length string of MSG_LEN bytes, and the unused portion of the message after the [NUM_PEERS] argument will be padded with 0 ('\0', not char '0').

After MSG_LEN bytes, **a list of [PEER_N_IP]:[PEER_N_PORT] will be appended to the message, also as a string**. That is, all IP&port tuples of the peers being sent will be formatted as strings with ':' between IP&port, and ' ' between the tuples. **Fixed length of [NUM_PEERS] * PEER_LIST_MAX_BYTE_PER_PEER bytes** will be allocated for the string, and any unused portion of the string will be padded with 0 ('\0', not char '0').

e. Torrent block status request message.

This message will be sent by the client function and will request torrent block status from a remote peer. The message will have the following protocol.

```
REQUEST_TORRENT_BLOCK_STATUS [MY_ENGINE_HASH] [MY_LISTEN_PORT] [TORRENT_HASH]
```

"REQUEST_TORRENT_BLOCK_STATUS" is a fixed string, and the rest of the arguments enclosed in [ ] are variables. [MY_ENGINE_HASH] refers to the requesting torrent's engine hash, [MY_LISTEN_PORT] refers to the requesting torrent's listening port, and [TORRENT_HASH] refers to the hash of the torrent whose block status is being requested.

The message will be contained in a fixed-length string of MSG_LEN bytes, and the unused portion of the message after the [TORRENT_HASH] argument will be padded with 0 ('\0', not char '0').

f. Torrent block status push message.

This message will be sent by the handler function of the request message and will push its torrent block status to a remote peer. The message will have the following protocol.

```
PUSH_TORRENT_BLOCK_STATUS [MY_ENGINE_HASH] [MY_LISTEN_PORT] [TORRENT_HASH] [BLOCK_STATUS]
```

"PUSH_TORRENT_BLOCK_STATUS" is a fixed string, and the rest of the arguments enclosed in [ ] are variables. [MY_ENGINE_HASH] refers to the pushing torrent's

engine hash, [MY_LISTEN_PORT] refers to the pushing torrent's listening port, [TORRENT_HASH] refers to the hash of the torrent whose block status is being pushed, and [BLOCK_STATUS] is the block status array of the given torrent.

The message up through [TORRENT_HASH] will be contained in a fixed-length string of MSG_LEN bytes, and the unused portion of the message after [TORRENT_HASH] argument will be padded with 0 ('\0', not char '0').

After MSG_LEN bytes, **[BLOCK_STATUS] will be appended to the message as a binary**. That is, all data contained in the block_status memory location of the given torrent will be appended to the message as a binary dump, without translation to a string format.

g. Torrent block data request message.

This message will be sent by the client function and will request torrent block data from a remote peer. The message will have the following protocol.

```
REQUEST_TORRENT_BLOCK [MY_ENGINE_HASH] [MY_LISTEN_PORT] [TORRENT_HASH] [BLOCK_INDEX]
```

"PUSH_TORRENT_BLOCK" is a fixed string, and the rest of the arguments enclosed in [ ] are variables. [MY_ENGINE_HASH] refers to the requesting torrent's engine hash, [MY_LISTEN_PORT] refers to the requesting torrent's listening port, [TORRENT_HASH] refers to the hash of the torrent whose block data is being requested, and [BLOCK_INDEX] refers to the index of the block whose data is being requested.

The message will be contained in a fixed-length string of MSG_LEN bytes, and the unused portion of the message after the [BLOCK_INDEX] argument will be padded with 0 ('\0', not char '0').

h. Torrent block data push message.

This message will be sent by the server function and will push torrent block data to a remote peer. The message will have the following protocol.

```
PUSH_TORRENT_BLOCK [MY_ENGINE_HASH] [MY_LISTEN_PORT] [TORRENT_HASH] [BLOCK_INDEX] [BLOCK_DATA]
```

"PUSH_TORRENT_BLOCK" is a fixed string, and the rest of the arguments enclosed in [ ] are variables. [MY_ENGINE_HASH] refers to the pushing torrent's engine hash, [MY_LISTEN_PORT] refers to the pushing torrent's listening port, [TORRENT_HASH] refers to the hash of the torrent whose block data is being pushed, and [BLOCK_INDEX] refers to the index of the block whose data is being pushed.

The message up through [BLOCK_INDEX] will be contained in a fixed-length string of MSG_LEN bytes, and the unused portion of the message after [BLOCK_INDEX] argument will be padded with 0 ('\0', not char '0').

After MSG_LEN bytes, **[BLOCK_DATA] will be appended to the message as a binary**. That is, all data contained in the memory location of the given block data will

be appended to the message as a binary dump, without translation to string data. (You can use get_block_ptr () to get the pointer to the block data of the given index.)

D. Important Notes

When calling the request functions inside the client function, **make sure to use the versions with _thread in the name**. These are wrapper functions that launch your request functions in a separate thread. The program will work without client request multi-threading, but will be much slower, and is not likely to scale with multiple peers.

To build your application, simply enter "**make**" and hit enter. Make sure to "make" after saving changes to your source code to run the updated binary. Use ctrl+c to exit the program.

You can modify other files and functions for testing purposes, **but you will only submit the "torrent_engine.c" file.** Therefore, the changes you make on other files will be lost in your submissions.

Also, **do not change** the function return values, function name, and the function arguments of any of the defined functions.

You are **allowed to define** more functions, global variables, enums, structs, etc. inside "torrent_engine.c" if you need to. You are also allowed to use additional libraries if it is included in the C / POSIX standard**.** External libraries that require installation will **NOT** be supported in your grading environment.

Please read the comments in the "torrent_engine.c" closely, as they contain guides and useful hints on implementing your project. Also, please make good use of the included structs and the associated functions, as it will make the management and manipulation of torrent data much easier. We **strongly recommend reading the descriptions and definitions of various structs and the associated functions in torrent.h and torrent.c** before starting the project.

Again, we will not grade for memory management. However, **understanding C pointers and memory structures will help you greatly in this project**, similar to the HTTP server. The string function strtok () will be very useful for this project.

Debug information printing can be enabled by entering "I" as the command in the UI. It will print various useful information about the application's operation, which will help debug your implementation greatly. You can also add additional debug information prints by using the INFO_PRT () macro as you would use a printf () function. You can enter "I" again to stop the debug information printing.

Check its operation by using it on the TA binary. Below is an example output.

```
[1.736s] REQUESTING 0x08e92a71 BLOCK 1 FROM 127.0.0.1:62123.
[1.736s] REQUESTING 0x878d71f4 BLOCK 44 FROM 127.0.0.1:62123.
[1.736s] RECEIVED "PUSH_TORRENT_BLOCK 0x72ffb52c 62123 0x878d71f4 44" from peer 127.0.0.1:46660.
[1.736s] RECEIVED 0x878d71f4 BLOCK 44 FROM 127.0.0.1:62123.
[1.736s] RECEIVED "PUSH_TORRENT_BLOCK 0x72ffb52c 62123 0x08e92a71 1" from peer 127.0.0.1:46670.
[1.736s] RECEIVED 0x08e92a71 BLOCK 1 FROM 127.0.0.1:62123.
[1.918s] RECEIVED "REQUEST_TORRENT_BLOCK_STATUS 0x72ffb52c 62123 0x08e92a71" from peer 127.0.0.1:46674.
[1.918s] RECEIVED 0x08e92a71 BLOCK STATUS REQUEST FROM 127.0.0.1:62123.
[1.918s] PUSHING 0x08e92a71 BLOCK STATUS TO 127.0.0.1:62123.
[1.918s] RECEIVED "REQUEST_TORRENT_BLOCK_STATUS 0x72ffb52c 62123 0x353f137d" from peer 127.0.0.1:46676.
[1.918s] RECEIVED 0x353f137d BLOCK STATUS REQUEST FROM 127.0.0.1:62123.
[1.918s] PUSHING 0x353f137d BLOCK STATUS TO 127.0.0.1:62123.
[1.919s] RECEIVED "REQUEST_TORRENT_BLOCK_STATUS 0x72ffb52c 62123 0xe6fd1247" from peer 127.0.0.1:46684.
[1.919s] RECEIVED 0xe6fd1247 BLOCK STATUS REQUEST FROM 127.0.0.1:62123.
[1.919s] PUSHING 0xe6fd1247 BLOCK STATUS TO 127.0.0.1:62123.
[1.951s] ENGINE REV COMPLETE.
ENGINE RUNNING - ENGINE HASH 0x3207a874, PORT: 62124, NUM_TORRENTS: 4
TORRENT 0:
        NAME: music.mp3, STATUS: COMPLETED
        HASH: 0x353f137d, SIZE: 3.51 MiB, BLOCKS: 113/113, NUM PEERS: 1, SPEED: 3616.00 KiB/s
                PEERS:127.0.0.1:62123 (113/113)
TORRENT 1:
        NAME: text_file.txt, STATUS: COMPLETED
        HASH: 0xe6fd1247, SIZE: 64 B, BLOCKS: 1/1, NUM PEERS: 1, SPEED: 32.00 KiB/s
                PEERS:127.0.0.1:62123 (1/1)
TORRENT 2:
        NAME: NXC_Lab_intro.pdf, STATUS: DOWNLOADING
        HASH: 0x878d71f4, SIZE: 25.50 MiB, BLOCKS: 1/817, NUM PEERS: 1, SPEED: 0.00 KiB/s
                PEERS:127.0.0.1:62123 (817/817)
TORRENT 3:
        NAME: snu_logo.png, STATUS: DOWNLOADING
        HASH: 0x08e92a71, SIZE: 81.42 KiB, BLOCKS: 1/3, NUM PEERS: 1, SPEED: 0.00 KiB/s
                PEERS:127.0.0.1:62123 (3/3)
```

Grading

HTTP Server

The HTTP server will be graded on functionality. That is, we will connect to your server implementation using a web browser and check the functionality of your server. These are the functionalities we will be checking to grade your submission.

(10 points) Connection establishment (Port opened and connection accepted)

(10 points) Successful HTTP response message sent back. (of ANY kind)

(20 points) Successful handling of a GET request.
   (Status 200, the main index.html page pops up.)

(10 points) Successful handling of a GET request for a non-existent page.
   (Status 404)

(10 points) Successful handling of all GET requests except authorization
   (All HTTP elements - HTML, CSS, JS, images, etc. - are properly loaded.)

(20 points) Successful handling of a GET request which requires authorization

(Status 401, requesting WWW-Authenticate, and handling BASE-64 codes)

(20 points) Successful handling of a POST message of a .jpg image.
(Receiving image, saving as a file, and updating the HTML file to show the image)

A total of 100 points are allocated for the HTTP server.

Torrent Application

The torrent application will be graded using test cases. We will replace the TA-implemented function with your function in the TA implementation binary and run the test cases. This will be done individually for each of the functions you must implement, and if the execution does not behave correctly, points allocated for the function will be deducted. These are the total points allocated for each function.

(10 points) torrent_client ()

(10 points) torrent_server ()

(4 points) listen_socket ()

(8 points) accept_socket ()

(8 points) connect_socket ()

(5 points) request_torrent_peer_list ()

(5 points) request_torrent_block_status ()

(5 points) request_torrent_block ()

(5 points) push_torrent_peer_list ()

(5 points) push_torrent_block_status ()

(5 points) push_torrent_block ()

(5 points) handle_request_torrent_peer_list ()

(5 points) handle_request_torrent_block_status ()

(5 points) handle_request_torrent_block ()

(5 points) handle_push_torrent_peer_list ()

(5 points) handle_push_torrent_block_status ()

(5 points) handle_push_torrent_block ()

A total of 100 points are allocated for the Torrent application.

Keep in mind that only a single function will be changed into your implementation for each test. Therefore, **your implementation should be able to communicate with the TA implementation** to get credit for your implementations.

Submission

Take your "http_engine.c" and "torrent_engine.c" files and compress them into a single zip file, with the name "<YOUR_NAME>_<YOUR_STUDENT_ID>_Proj1.zip". **DO NOT include any other files than "http_engine.c" and "torrent_engine.c" in the .zip file.** Upload the .zip file to the ETL.

We hope you have fun!

2023. 10. 1.

Jongseok Park.