Assignment 2

Due date: 23:59, November 22, 2023

Contact TAs: ntu.cnta@gmail.com

Updates

All of the updates will be colored in red. Please check them carefully and follow the latest version of this document.

- Oct. 23th: Fixed some typos in Listing 13 in Section 3.3.2. (Credit: William Tsui on NTU Cool Discussion)
- Oct. 25th: Added more descriptions in Section 2.2.1. (Credit: Yi, Chen-Yao on NTU Cool Discussion)
- Nov. 4th: Added missing descriptions in Section 2.7 and Section 3.2.2. The deadline has been extended to 23:59, November 22.
- Nov. 8th: Corrected descriptions about `Content-Length` in Section 2.2.3, Section 2.5.4 and Section 2.5.5.

Contents

Instructions

- In this assignment, you are asked to implement a web-based video uploading and streaming system.
- Please study the specifications in the following sections carefully.
- Make sure your programs can run on the container we provided.
- It's a better way to discuss the assignment requirements or your solutions—away from a computer and without sharing code—but you should not discuss the detailed nature of your solution. In your report, you need to cite the sources (either from textbooks, academic papers, website articles, or ChatGPT) and give credit to whom you discuss with. Also, don' t put your code in a public repository before the end of this course.
- No plagiarism is allowed. A plagiarist will be graded zero.

1 Background

1.1 HTTP

HyperText Transfer Protocol (HTTP)[1][2] is widely used application protocol on the Internet nowadays. HTTP uses a client-server model[3] to exchange information. An HTTP client sends a request to the server, and the server answers with a response that the client requested. HTTP is a stateless protocol, which means there is no link between two requests on the same connection. In this assignment, you will implement an HTTP server/client that handles HTTP requests/responses.

1.1.1 Messages Structure

The format of an HTTP message[4] is:

- A **start-line** describing the requests to be implemented, or its status of whether successful or a failure. It is always a single line.
- A optional set of **headers**[5] separated by lines. Field names in headers are case-insensitive.[6]
- A **blank line** indicating all meta-information for the request has been sent.
- An optional **body** containing data associated with the request or the document associated with a response.

The end-of-line marker in HTTP requests is CRLF[7], which is represented as `\r\n` in C language.

1.1.2 Requests

HTTP requests are the messages that the client sends to the server in order to get the resources. Listing 1 shows a simple example of an HTTP request made by Google Chrome. The start-line containing a method (like `GET`, `PUT` or `POST`), a query string (in this case `/`), and the HTTP protocol version (`HTTP/1.1`). Header lines provide information about the request.

[1]https://datatracker.ietf.org/doc/html/rfc2616
[2]https://www.w3.org/Protocols/HTTP/1.0/spec.html
[3]https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
[4]https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages
[5]https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
[6]https://datatracker.ietf.org/doc/html/rfc2616#section-4.2
[7]https://developer.mozilla.org/en-US/docs/Glossary/CRLF

```
GET / HTTP/1.1\r\n
Host: example.com\r\n
Connection: keep-alive\r\n
Pragma: no-cache\r\n
Cache-Control: no-cache\r\n
Upgrade-Insecure-Requests: 1\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/117.0.0.0 Safari/537.36\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng
    ,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7\r\n
Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7,fr;q=0.6,ms;q=0.5,zh-CN;q=0.4\r\n
\r\n
```

Listing 1: An example of an HTTP request

1.1.3 Responses

HTTP responses are the messages that the server sends to the client with the resources that the client has asked for. Listing 2 shows a simple example of an HTTP response made by `example.com`. The start-line contains the HTTP protocol version (`HTTP/1.1`), a status code (`200`), and a status text (`OK`). Header lines provide information about the response.

```
HTTP/1.1 200 OK\r\n
Accept-Ranges: bytes\r\n
Age: 198919\r\n
Cache-Control: max-age=604800\r\n
Content-Type: text/html; charset=UTF-8\r\n
Date: Sun, 08 Oct 2023 09:44:46 GMT\r\n
Etag: "3147526947"\r\n
Expires: Sun, 15 Oct 2023 09:44:46 GMT\r\n
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT\r\n
Server: ECS (laa/7B7B)\r\n
Vary: Accept-Encoding\r\n
X-Cache: HIT\r\n
Content-Length: 1256\r\n
\r\n
<!doctype html>
...
</html>
```

Listing 2: An example of an HTTP response

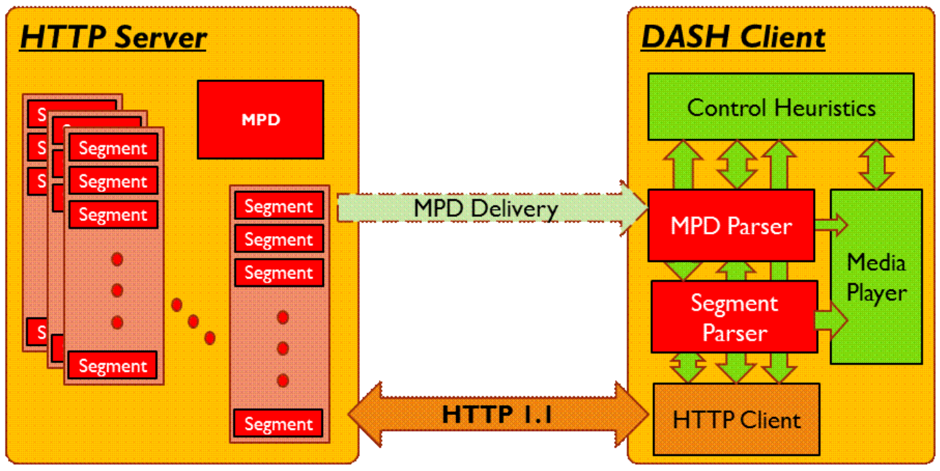

Figure 1: DASH architecture

1.1.4 URL Encoding / Decoding

All the symbols in URL expect `[A-Z][a-z][0-9][-_.~]` should be encoded by using percent-encoding[8]. Percent-encoding a reserved character involves converting the character to its corresponding byte value in ASCII and then representing that value as a pair of hexadecimal digits (if there is a single hex digit, a leading zero should be added).

The digits, preceded by a percent sign % as an escape character, are then used in the URI in place of the reserved character. For example, the path `/path/to/the/file/hello world.txt` should be encoded as `/path/to/the/file/hello%20world.txt`. (The reason why `/` is not percent-encoded is it is the delimiter between path segments in this case.)

In this assignment, You don’ t need to deal with non-ASCII characters.

1.2 MPEG-DASH

DASH is a suite of standards providing a solution for multimedia streaming using existing HTTP infrastructure.[9] Figure 1 shows a simple scenario between a server and a client. In the figure, the multimedia content is stored on an HTTP server and is delivered using HTTP. The content on the server can be grouped into two parts:

- **Media Presentation Description (`.mpd` file)** which describes a manifest of the available content, its various alternatives, their URL addresses, and other characteristics.
- **Video segments (e.g., `.m4s` files)** which contain the actual multimedia bitstreams in the form of chunks, in single or multiple files.

[8]https://en.wikipedia.org/wiki/Percent-encoding
[9]https://www.mpeg.org/standards/MPEG-DASH/

To play the content, the DASH client first obtains the `.mpd` file, which contains the media types, resolutions, bandwidths, and many other characteristics of the content. Using this information, the DASH client selects the appropriate encoded alternative and starts streaming the content by fetching the segments using HTTP GET requests.

After appropriate buffering to allow for network throughput variations, the client continues fetching the subsequent segments and monitors the bandwidth fluctuations of the network. Depending on its measurements, the client decides how to adapt to the available bandwidth by fetching segments of different alternatives (with lower or higher bitrate) to maintain an adequate buffer.

Nowadays, many online platforms, such as *YouTube* or *NTU COOL*, use DASH to deliver multimedia content.

1.3 System Architecture of the Assignment

In this assignment, you need to design both the HTTP server and the client to achieve the following goals:

- The server can serve static web page `index.html`, `uploadf.html`, `uploadv.html`.
- The server can serve dynamic web pages `listf.rhtml`, `listv.rhtml`, `player.rhtml`.
- The server can serve text/binary files (including DASH videos).
- The client can upload files and videos to the server.
- The client can fetch files from the server.

Figure 2 illustrates the system architecture in this assignment. The server and the client use `HTTP 1.1` as the protocol for information exchange. The server serves multiple types of files, including the DASH videos. There might be multiple, various clients (e.g., the client implemented in Section 3 and browsers like *Google Chrome*, *Microsoft Edge*) establishing connections with the server at the same time. In Section 2 and Section 3, we will go through the implementation details of the server and the client separately.

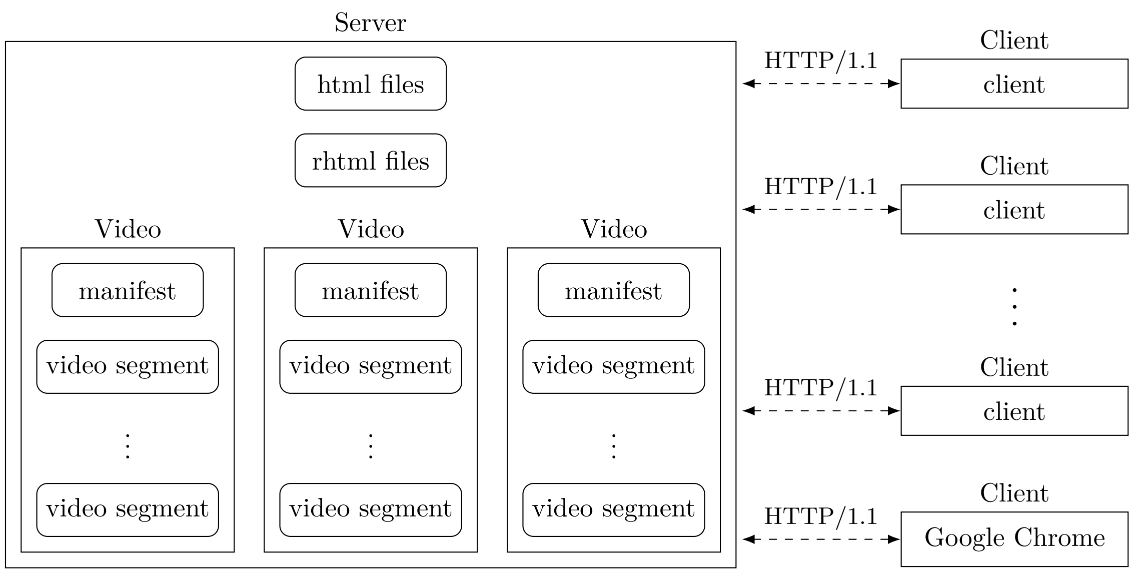

Figure 2: System architecture for Assignment 2

2 The Server

2.1 Arguments

The program server takes exactly one argument, `port`. Otherwise, it should output `Usage: ./server [port]` to `stderr` and immediately terminate the program with return code `-1`, as shown in Listing 3.

```
$ ./server
Usage: ./server [port]
```

Listing 3: Server argument

2.2 Connections

2.2.1 Handling Multiple Clients

You can choose one of the following three methods for implementation:

- **POSIX Threads**: allows a program to control multiple different flows of work that overlap in time. (not recommended)
- `select()`: allows a program to monitor multiple file descriptors, waiting until one or more descriptors become ready to perform an I/O operation (e.g., `read()`, `write()`) without blocking.
- `poll()`: a more scalable solution than `select()`, as it does not limit the number of file descriptors.

The server needs to be able to maintain a total of 100 connections simultaneously, where any socket can transmit data at any time until the client intentionally terminates the connection.

2.2.2 Receiving HTTP Requests

After sending the response, the server needs to handle connections differently based on the options specified in the `Connection` field of the request header. There are two possible options of `Connection` field:

- `Keep-Alive`: indicates that the client would like to keep the connection open.
- `Close`: indicates that either the client or the server would like to close the connection.

Connection options are case-insensitive. By default, your server (compliant with `HTTP 1.1`) should keep the connection open unless `Connection: Close` is set.

2.2.3 Sending HTTP Responses

Here are the HTTP response header types that your server needs to contain:

- `Server`[10]: contains information about the software used by the origin server to handle the request. Your server should always respond with this header, and the value should be `CN2023Server/1.0`.
- `Content-Type`[11]: indicates the media type of the entity-body sent to the recipient. Refer to Section 2.6 for information about MIME-Type. Your server should respond with this header if the body is not empty.
- `Content-Length`[12]: indicates the size of the entity-body in decimal number of bytes. Your server should always respond with this header, even if the body is empty.[13]
- `Allow`[14]: lists the set of methods supported by the resource. Your server should respond with this header if the status code is `405 Method Not Allowed`. (Refer to Section 2.5.4 for an example of `405 Method Not Allowed` response.)

2.3 Security

2.3.1 Authentication

You need to implement HTTP basic authentication[15] on your server. If an endpoint mentioned in Section 2.4 requires authentication, the server must verify whether the client has sent a valid credential before sending the response. If not, respond with a `401 Unauthorized` status code instead.

[10]https://datatracker.ietf.org/doc/html/rfc2616#section-14.38
[11]https://datatracker.ietf.org/doc/html/rfc2616#section-14.17
[12]https://datatracker.ietf.org/doc/html/rfc2616#section-14.13
[13]https://datatracker.ietf.org/doc/html/rfc2616#section-4.3
[14]https://datatracker.ietf.org/doc/html/rfc2616#section-14.7
[15]https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

The status code `401 Unauthorized` should be sent with an HTTP `WWW-Authenticate` response header that contains information about how the client can request the resource again after prompting the user for authentication credentials.[16] (Refer to Section 2.5 for information about status codes.)

The server should accept HTTP basic authentication, and the realm should be `{your student id}`. (Refer to Section 2.5.2 for an example of `401 Unauthorized` response.)

The file `hw2/secret` stores the valid credentials. Usernames and passwords are separated by ":", and each pair is separated by a new line. Listing 4 shows an example of it.

```
username1:password1
username2:password2
username3:password3
```

Listing 4: an example of file `secret`

The username and password are case-sensitive. Their lengths are no more than 20 characters.

2.3.2 Keep the Secret

Due to security concerns, the server must not deliver any content outside the `hw2/web` directory. Therefore, `hw2/secret` should not be accessed by any endpoint mentioned in Section 2.4.

2.4 Routing

This section explains the server's approach to managing requests for different resources. Routing is the mechanism by which requests (as specified by a URL and HTTP method) are routed to the code that handles them.[17] The routing table for the server in this assignment is shown in Table 1.

2.4.1 Route: /

This endpoint only allows GET requests and responds with the provided template `hw2/web/index.html`. You need to edit `hw2/web/index.html`, filling in your name and id in the table.

2.4.2 Route: /upload/file

This endpoint only allows GET requests and responds with the provided template `hw2/web/uploadf.html` after authentication. Users can upload files to the server using this endpoint in cooperation with `/api/file`.

2.4.3 Route: /upload/video

This endpoint only allows GET requests and responds with the provided template `hw2/web/uploadv.html` after authentication. Users can upload files to the server using this endpoint in cooperation with `/api/video`.

[16] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401

[17] https://www.oreilly.com/library/view/web-development-with/9781491902288/ch14.html

Route	Methods	Description
/	GET	serve `index.html`
/upload/file	GET	serve `uploadf.html`
/upload/video	GET	serve `uploadv.html`
/file/	GET	use template `listf.rhtml` to list files
/video/	GET	use template `listv.rhtml` to list videos
/video/{videoname}	GET	use template `player.rhtml` to play a video
/api/file	POST	upload file to server
/api/file/{filepath}	GET	get the file from server
/api/video	POST	upload video file to server
/api/video/{filepath}	GET	get the video file from server

Table 1: Server routing table

2.4.4 Route: /file/

This endpoint only allows GET requests and responds with the provided template `listf.rhtml`. Before the server sends it to the client, the file should be preprocessed as follows:

- `<?FILE_LIST?>`: Replace this pseudo-HTML tag with HTML table rows that contain all the files the server has under `hw2/web/files`.

  Example: Assume there are two files, `example.txt` and `sample.png`, under `hw2/web/files`. You should replace `<?FILE_LIST?>` with

  ```
  <tr><td><a href="/api/file/example.txt">example.txt</a></td></tr>
  <tr><td><a href="/api/file/sample.png">sample.png</a></td></tr>
  ```

2.4.5 Route: /video/

This endpoint only allows GET requests and responds with the provided template `listv.rhtml`. Before the server sends it to the client, the file should be preprocessed as follows:

- `<?VIDEO_LIST?>`: Replace this pseudo-HTML tag with HTML table rows that contain all the videos the server has under `hw2/web/videos`.

  Example: Assume there are two video folders, `demoA` and `demoB`, under `hw2/web/videos`. You should replace `<?VIDEO_LIST?>` with

  ```
  <tr><td><a href="/video/demoA">demoA</a></td></tr>
  <tr><td><a href="/video/demoB">demoB</a></td></tr>
  ```

2.4.6 Route: /video/{videoname}

This endpoint only allows GET requests and responds with the provided template `player.rhtml`. Before the server sends it to the client, the file should be preprocessed:

- `<?VIDEO_NAME?>`: Replace this pseudo-HTML tag with the video name (i.e., the name of the video folder)
- `<?MPD_PATH?>`: Replace this pseudo-HTML tag with the MPD URL (*Hint:* You can use the endpoint `/api/video/{filepath}`)

2.4.7 Route: /api/file

This endpoint only allows POST requests with content type `multipart/form-data` and requires authentication. If the request is valid, the server should save the uploaded file into `hw2/web/files`, and return `200 OK` with the plain text body `File Uploaded\n`.

This endpoint should accept both binary and text files that are no more than 200MB. The server should overwrite the files if they already exist.

2.4.8 Route: /api/file/{filepath}

This endpoint only allows GET requests and responds with the corresponding file under `hw2/web/files` if it exists.

2.4.9 Route: /api/video

This endpoint only allows POST requests with content type `multipart/form-data` and requires authentication. If the request is valid, the server should save the uploaded file into a directory used to hold temporary files, `hw2/web/tmp`, and return `200 OK` with the plain text body `Video Uploaded\n` immediately. Then, the server converts the video file into DASH stream. (Refer to Section 2.7 for information about video transcoding.)

This endpoint should accept video files that are no more than 200MB. The server should overwrite the files and folders if they already exist.

2.4.10 Route: /api/video/{filepath}

This endpoint only allows GET requests and responds with the corresponding file under `hw2/web/videos` if it exists.

2.5 Status Codes

The server should respond with appropriate HTTP status code according to the following list:

- `200 OK`: if the requested resource is found and sent successfully.
- `401 Unauthorized`: if the client is lack of valid credentials.
- `404 Not Found`: if the requested resource is not found.
- `405 Method Not Allowed`: if the request method is not allowed.
- `500 Internal Server Error`: if an unexpected error occurred. (This should not happen.)

However, multiple conditions might be true at the same time. Figure 3 shows how to determine what status code you should return.

2.5.1 200 OK

Listing 5 is an example of HTTP `200 OK` response.

```
HTTP/1.1 200 OK\r\n
Server: CN2023Server/1.0\r\n
Content-Type: text/plain\r\n
Content-Length: 3\r\n
\r\n
OK\n
```

Listing 5: An example of HTTP 200 response

2.5.2 401 Unauthorized

Listing 6 is an example of HTTP `401 Unauthorized` response. The server should send the response with a plain text body `Unauthorized\n`.

```
HTTP/1.1 401 Unauthorized\r\n
Server: CN2023Server/1.0\r\n
WWW-Authenticate: Basic realm="B10902999"\r\n
Content-Type: text/plain\r\n
Content-Length: 13\r\n
\r\n
Unauthorized\n
```

Listing 6: An example of HTTP 401 response

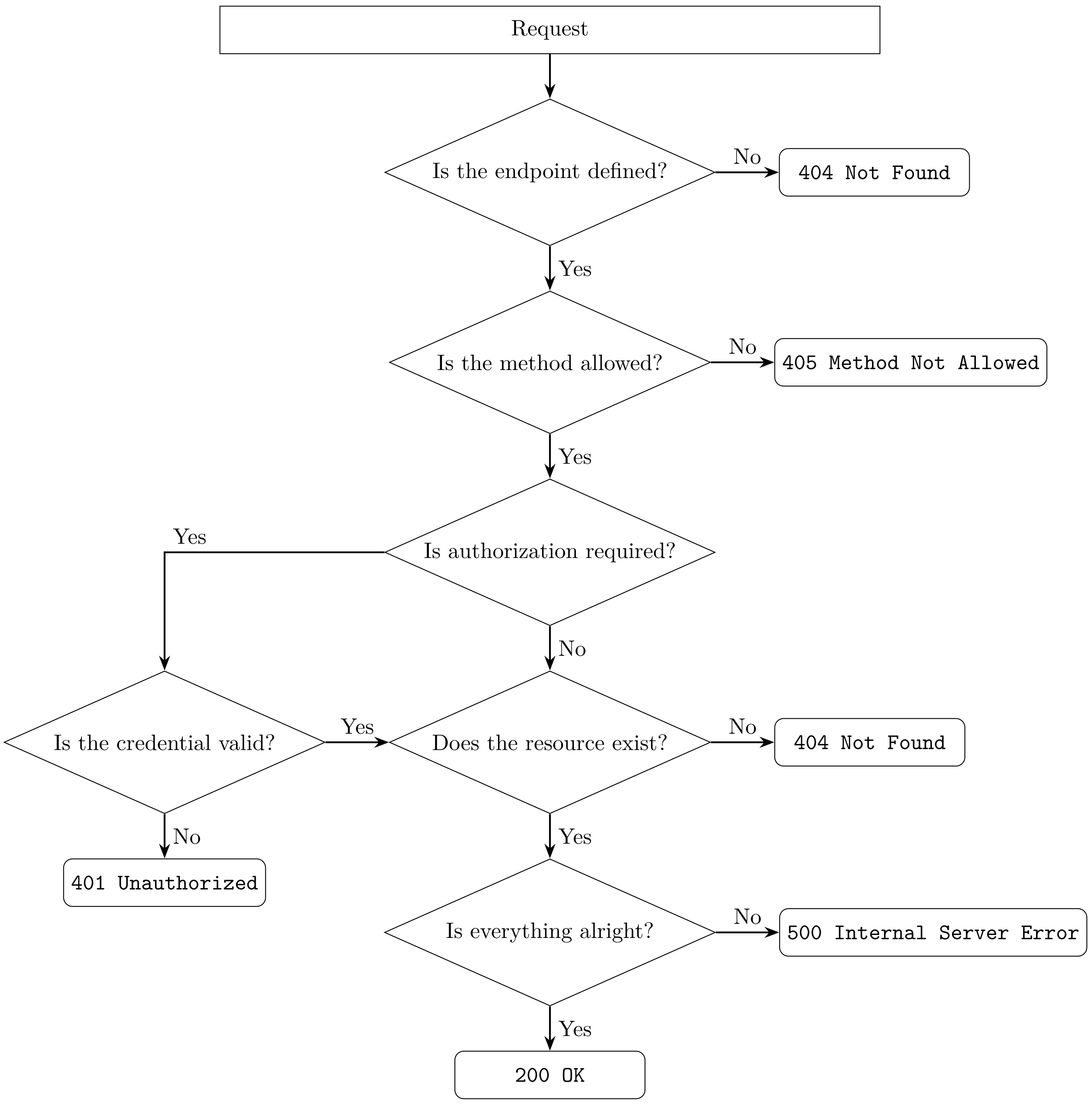

Figure 3: Decision-making process of status code

2.5.3 404 Not Found

Listing 7 is an example of HTTP `404 Not Found` response. The server should send the response with a plain text body `Not Found\n`.

```
HTTP/1.1 404 Not Found\r\n
Server: CN2023Server/1.0\r\n
Content-Type: text/plain\r\n
Content-Length: 10\r\n
\r\n
Not Found\n
```

Listing 7: An example of HTTP 404 response

2.5.4 405 Method Not Allowed

Listing 8 is an example of HTTP `405 Method Not Allowed` response. The server should send the response with an empty body.

```
HTTP/1.1 405 Method Not Allowed\r\n
Server: CN2023Server/1.0\r\n
Allow: GET\r\n
Content-Length: 0\r\n
\r\n
```

Listing 8: An example of HTTP 405 response

2.5.5 500 Internal Server Error

Listing 9 is an example of HTTP `500 Internal Server Error` response. The server should send the response with an empty body.

```
HTTP/1.1 500 Internal Server Error\r\n
Server: CN2023Server/1.0\r\n
Content-Length: 0\r\n
\r\n
```

Listing 9: An example of HTTP 500 response

2.6 MIME Type

You need to select the appropriate MIME Type for the response header field `Content-Type` based on the extension of the file that the server is going to send. Table 2 shows the mapping between MIME types and file extensions.

MIME Type	File Extension
text/html	`.html` (including `.rhtml`)
video/mp4	`.mp4`, `.m4v`
video/iso.segment	`.m4s`
audio/mp4	`.m4a`
application/dash+xml	`.mpd`
text/plain	(default type)

Table 2: MIME type

2.7 Converting Video to DASH

By Section 1.2, you should already have a basic understanding of how DASH streaming works. Since this course is not about video encoding, we have provided the script that you will need in Listing 10. Replace `<VIDEO>` with the video filename. Save the output files into `hw2/web/videos/{videoname}` and name the manifest (`.mpd`) file as dash.mpd (full path will be: `hw2/web/videos/{videoname}/dash.mpd`). Ensure that your server can still handle other requests while converting the video file.

```
$ ffmpeg -re -i <VIDEO>.mp4 -c:a aac -c:v libx264 \
-map 0 -b:v:1 6M -s:v:1 1920x1080 -profile:v:1 high \
-map 0 -b:v:0 144k -s:v:0 256x144 -profile:v:0 baseline \
-bf 1 -keyint_min 120 -g 120 -sc_threshold 0 -b_strategy 0 \
-ar:a:1 22050 -use_timeline 1 -use_template 1 \
-adaptation_sets "id=0,streams=v id=1,streams=a" -f dash \
<PATH>/dash.mpd
```

Listing 10: FFmpeg script

3 The Client

3.1 Arguments

The program client accepts either 2 or 3 arguments, `host`, `port` and `username:password` (optional). If the correct number of arguments is not provided, it should output `Usage: ./client [host] [port] [username:password]` to `stderr` and immediately terminate the program with return code `-1`, as shown in Listing 11.

`username:password` is a string that contains a username and a password separated by ”`:`”. (Refer to Section 2.3.1 for information about HTTP authentication.)

```
$ ./client
Usage: ./client [host] [port] [username:password]
```

Listing 11: Client arguments

3.2 Connections

3.2.1 Resolving Domain Names

If `host` is a domain name (e.g., `csie.ntu.edu.tw`), the client needs to convert it into the corresponding IP address in order to create sockets for it.

3.2.2 Sending HTTP Requests

Here are the HTTP request header types that your client needs to contain:

- `Host`[18]: specifies the Internet host and port number of the resource being requested. Your client should always request with this header.
- `User-Agent`[19]: contains information about the user agent originating the request. Your client should always request with this header, and the value should be `CN2023Client/1.0`.
- `Connection`[20]: indicates whether the network connection stays open after the current transaction finishes. Your client should always request with this header, and the value should be `keep-alive`.
- `Content-Type`: indicates the media type of the entity-body sent to the recipient. Your client should respond with this header if the body is not empty.
- `Content-Length`: indicates the size of the entity-body in decimal number of bytes. Your client should respond with this header if the body is not empty.

[18]https://datatracker.ietf.org/doc/html/rfc2616#section-14.23
[19]https://datatracker.ietf.org/doc/html/rfc2616#section-14.43
[20]https://datatracker.ietf.org/doc/html/rfc2616#section-14.10

3.3 Commands

Each command you need to implement will be explained in detail in this section. Please carefully follow the instructions for the output formats.

You are required to implement a simple shell with an interactive interface similar to bash. The prompt should start with the symbol "> " (including a space) at the beginning of each command.

For each command, if the command finishes without any errors, the client should print out `Command succeeded.` to `stdout`. Otherwise, `Command failed.` should be printed out to `stderr`. If a command is incomplete, the client should print out usage information to `stderr`. If a command is not listed above, the client should print out `Command Not Found.` to `stderr`.

3.3.1 put

For the `put` command, the client should upload the file specified in the argument to the server endpoint `/api/file`. There won' t be multiple clients putting the same file simultaneously.

```
> put
Usage: put [file]
> put a.file.that.does.not.exist
Command failed.
> put hello world.txt
Command succeeded.
```

Listing 12: an example of command "`put`"

3.3.2 putv

For the `putv` command, the client should upload the video file specified in the argument to the server endpoint `/api/video`. There won' t be multiple clients putting the same video file simultaneously.

```
> putv
Usage: putv [file]
> putv a.mp4.that.does.not.exist
Command failed.
> putv my-video(1).mp4
Command succeeded.
```

Listing 13: an example of command "`putv`"

3.3.3 get

For the `get` command , the client should download the file specified in the arguments from the server endpoint `/api/file/{filepath}`. The downloaded file should be saved in `hw2/files`.

```
> get
Usage: get [file]
> get a.file.that.does.not.exist
Command failed.
> get hello world.txt
Command succeeded.
```

Listing 14: an example of command "get"

3.3.4 quit

For the `quit` command, the client should close the socket to the server and terminate the process.

```
> quit
Bye.
```

Listing 15: an example of command "quit"

4 Grading

4.1 (85%) Codes

4.1.1 Submission

- If the path does not exist (e.g., `hw2/web/files`), your programs need to create it automatically.
- Make sure your files can be successfully compiled. (Only C/C++ will be accepted)
- You are not allowed to use **ANY** third-party libraries that we didn't mention in the specifications.
- The repository structure should be the same as those in Figure 4. (without unnecessary files, e.g., report, video files, and execution files, and folders, such as `hw2/files`, `hw2/web/files`, `hw2/web/tmp`, `hw2/web/videos`)
- You need to submit your codes to GitHub Classroom.
- If multiple versions exist, the latest one will be counted.
- If you commit to GitHub classroom after the deadline, your score will be deducted 20 points per day. You might want to think twice before you do that.

```
.
├─ .gitignore
├─ Dockerfile
├─ docker-compose.yml
└─ hw2/
   ├─ utils/
   │  ├─ base64.c
   │  ├─ base64.h
   │  └─ (Other files if needed)
   ├─ web/
   │  ├─ index.html
   │  ├─ listf.rhtml
   │  ├─ listv.rhtml
   │  ├─ player.rhtml
   │  ├─ uploadf.html
   │  └─ uploadv.html
   ├─ client.c
   ├─ makefile
   ├─ server.c
   └─ (Other files if needed)
```

Figure 4: Repository structure

4.1.2 Test cases

We will use multiple methods to test your server and client for all the functionalities that have been mentioned in the specification. Please note that not only should your own server and client be able to exchange data, but you also need to ensure that your server and client can interact with other servers and clients according to the assignment specifications. Any attempt to circumvent the assignment specifications, including the HTTP protocol, in transmitting data will be considered illegal.

The test cases of the server and the client are shown in Table 3 and Table 4, respectively.

Case		Points	Description
Features	Home Page	4	Test connectivity of `/`.
	List	6	Test connectivity of `/file/` and `/video/`.
	Video Player	3	Test connectivity of `/video/{videoname}`.
	File Upload	14	Test functionality of file/video upload.
	File Download	8	Test functionality of file/video download.
Implementation	URL Encoding/Decoding	5	Check if percent-encoding works as expected.
	Content Type	5	Check if `Content-Type` header is set correctly.
	Content Length	5	Check if `Content-Length` header is set correctly.
	Connection	5	Check if `Connection header` works as expected.
	Multiple Clients	10	Run stress tests on multiple connections.

Table 3: Server evaluation

Case		Points	Description
Commands	Put	5	Test functionality of command `put`
	Put video	5	Test functionality of command `putv`.
	Get	5	Test functionality of command `get`.
	Quit	1	Test functionality of command `quit`.
Implementation	URL Encoding/Decoding	2	Check if percent-encoding works as expected.
	Connection	2	Check if the connection is reused.

Table 4: Client evaluation

4.2 (15%) Report

4.2.1 Submission

You should upload your report (as a **.pdf** file) to Gradescope with the filename `{studentID}_hw2.pdf` (e.g., `B10902999_hw2.pdf`).

4.2.2 Contents

In your report, please answer the following questions.

1. (5%) Draw a flowchart of the video streaming and explain how it works in detail. (*Hint:* You can use something like *Chrome DevTools* to observe and collect data about the network flow.)
2. (5%) What are the differences between serving MP4 videos and DASH files on an HTTP server? Answer this question by considering both theoretical and practical (i.e., your server) aspects.

3. (5%) Explain the workflow of HTTP authentication through your implementation (you don't need to dive into C/C++ code). Is HTTP basic authentication secure enough? Why, or why not? If not, what alternative methods can be used?

4.3 Bonus: Git Commit History

To better track everyone's progress and improve this course, we encourage you to commit your codes to GitHub Classroom regularly after completing some parts of this assignment. Make sure your commit messages are meaningful.

You can decide whether to do the bonus part or not. If you want, please regularly (but not intensively) commit your codes to GitHub Classroom and answer the following questions in your report.

- How did you utilize Git for development in this assignment?
- What benefits did it bring?
- Is it a better way for you to submit homework via GitHub Classroom than via traditional ways (e.g., submit a .zip file to NTU Cool)? Why or Why not?

Once you join this part, we will give you a bonus of 5 points for Assignment 2.