Proxy Lab

本文档结合handout、 自己的实验过程、CSAPP 3rd edition、recitation lectures[1]，列出解题流程和思路。

Preparation

1. 学习CSAPP Part III Interaction and Communication between Programs.
   其中与lab关系密切的章节有:

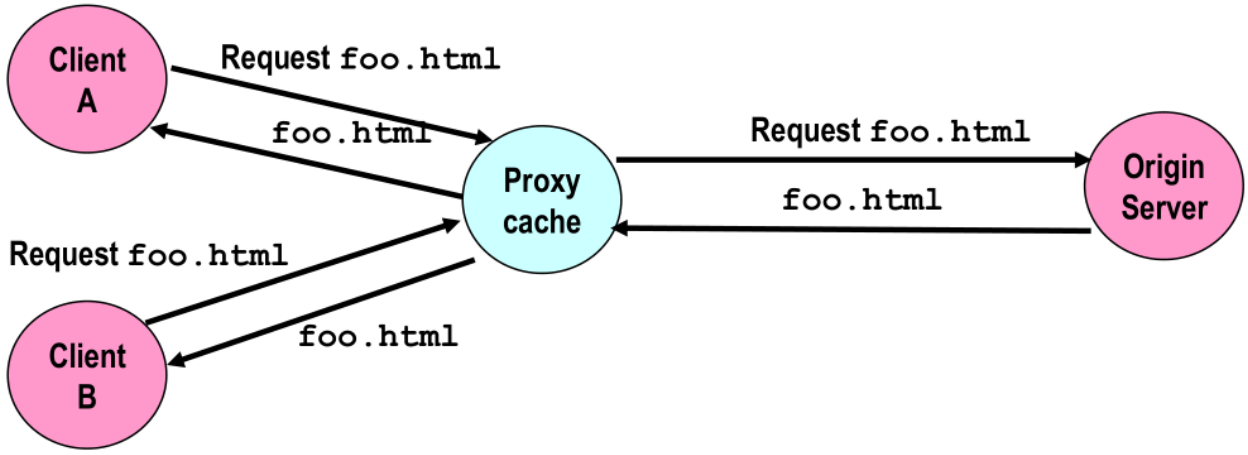
- 10.5 Robust Reading and Writing with the Rio Package
- 11.4 The Sockets Interface - 11.6 Putting It Together: The Tiny Web Server
- 12.3 Concurrent Programming with Threads - 12.5 Synchronizing Threads with Semaphores

2. **细读一遍handout** - the most Important thing!

Proxy Introduction

Why Proxy?[1-1]

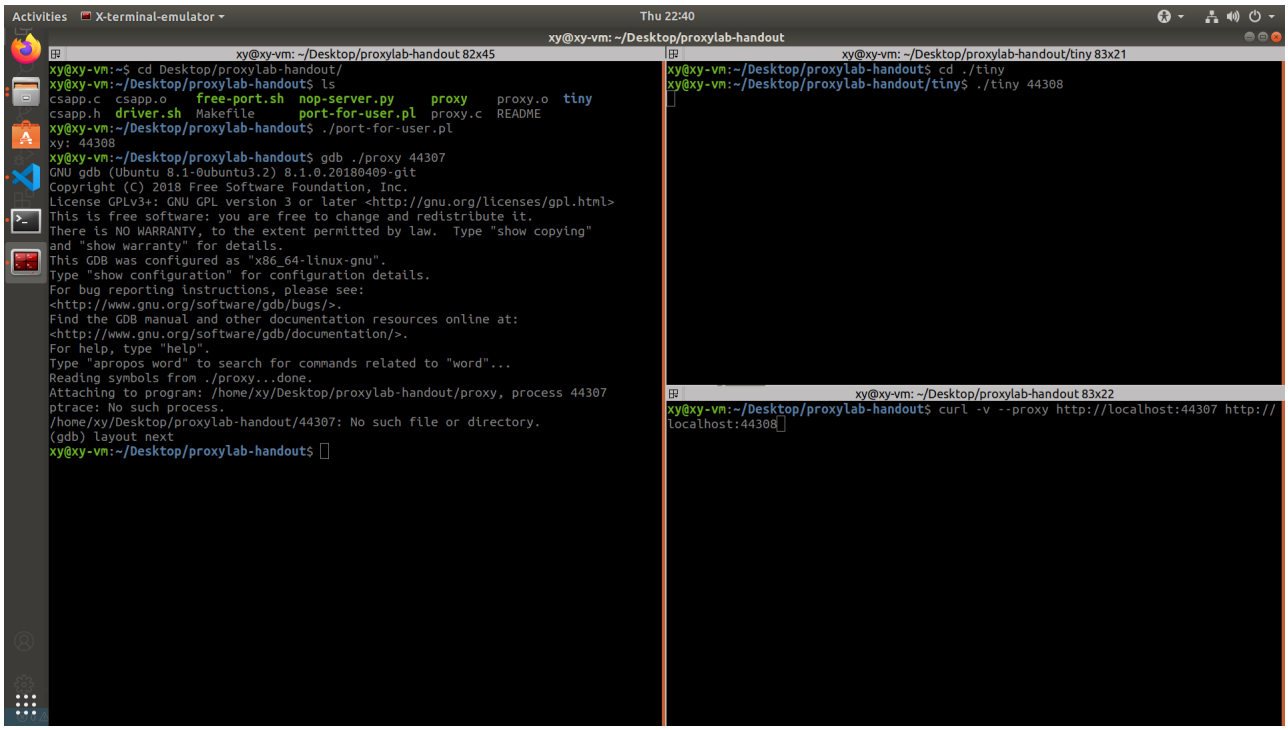
- Proxy acts as a server when connecting to a client and act as a client when connecting to remote web servers.
- It can perform useful functions as requests and responses pass by. Examples: Caching, logging, anonymization, filtering, transcoding



Debug Method

GDB debug

- Open up three terminals: for Tiny server, gdb proxy and curl



- Can make multiple requests, but need more terminals for multiple instances of the Tiny server
- If the data is corrupted, need to manually inspect lines of gibberish binary data to check error

You can use `curl` to generate HTTP requests to any server, including your own proxy. For example, if your proxy and Tiny are both running on the local machine, Tiny is listening on port 44308, and proxy is listening on port 44307, then you can request a page from Tiny via your proxy using the following curl command:

```
$ curl -v --proxy http://localhost:44307 http://localhost:44308/home.html
```

Sometimes, printing data is easier to locate the bug than using GDB. For example, print successful / fail information when proxy receives request/response.

PXYDRIVE debug

A better choice than GDB. It is a REPL for testing your proxy implementation. Refer to Recitation 12[1-2] to see its basic usage and test for PART I, refer to Recitation 14[1-3] to see the test for PART II and PART III .

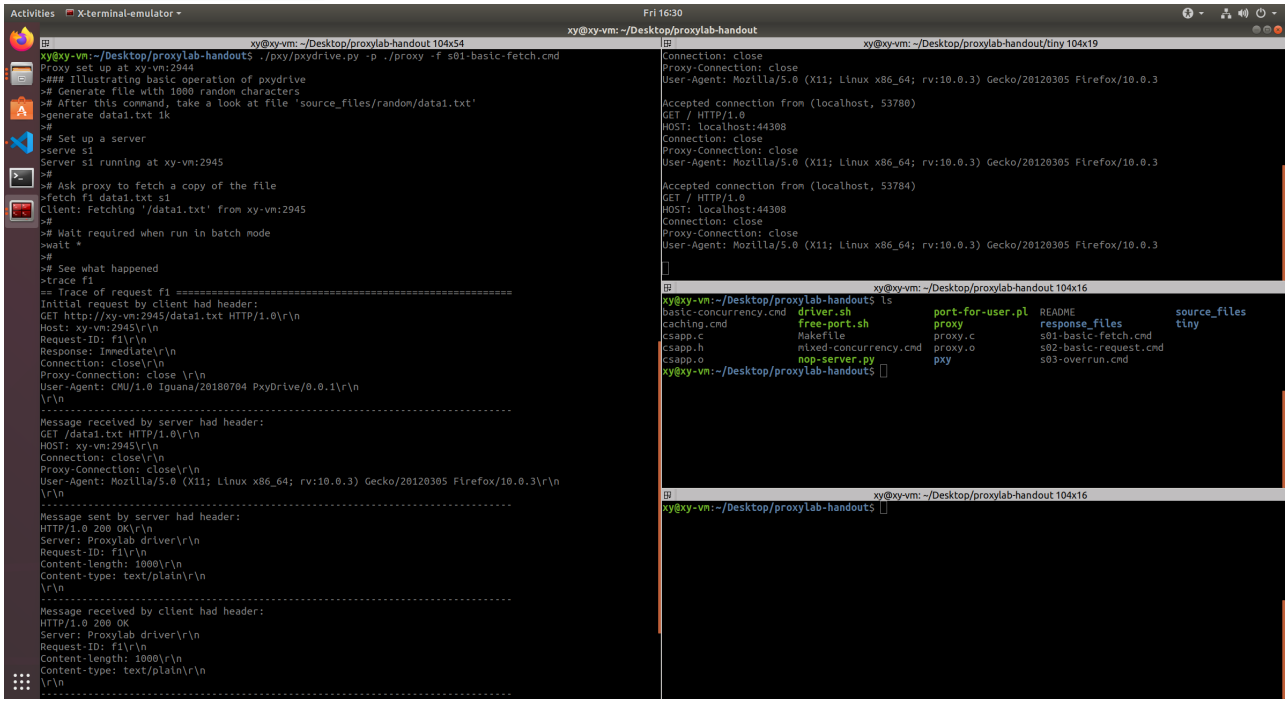
How to use PXYDRIVE?

1. download ./tar

```
# test scripts for PART I
$ wget http://www.cs.cmu.edu/~213/activities/pxydrive-tutorial.tar
# test scripts for PART II and PART III
$ wget http://www.cs.cmu.edu/~213/activities/pxydrive-tutorial2.tar

$ tar -xvf pxydrive-tutorial.tar
$ cd pxydrive-tutorial
```

3. copy `./pxy` directory and all `.cmd` files to the lab working directory
4. test `./proxy` using the following command. Note: `-f + one of ./cmd files` e.g., `s01_basic_fetch.cmd` is the most basic functions that the proxy should provide.



./driver.sh debug

use autograder `driver.sh`, in the `proxylab-handout` directory:

```
$./driver.sh
```

TINY web server

Tiny assumes that the home directory for static content is its current directory and that the home directory for executables is `./cgi-bin`. Any URI that contains the string `cgi-bin` is assumed to denote a request for dynamic content. The default filename is `./home.html`.

Tiny serves five common types of static content: HTML files, unformatted text files, and images encoded in GIF, PNG, and JPEG formats.

Tiny serves any type of dynamic content by forking a child process and then running a CGI program in the context of the child.

Part I: Implementing a sequential web proxy

General Procedure

The first step is implementing a basic sequential proxy that handles HTTP/1.0 GET requests. I only implemented GET request.

- When started, proxy should **listen** for incoming connections on a port whose number will be specified on the command line.

```
    // open a listening socket
    listenfd = Open_listenfd(argv[1]);
    // execute the typical infinite server loop
    while (1) {
        clientlen = sizeof(clientaddr);
        // repeatedly accept a connection request
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen, hostname, MAXLINE, port,
MAXLINE, 0);
        printf("Accepted connection from (%s, %s)\n", hostname, port);
        // perform a transaction
        doit(connfd);
        // close its end of connection
        Close(connfd);
    }
```

- Once a connection is established, proxy should **read** the entirety of the request from the client and **parse** the request. the function `doit()` performs this transaction. The first part of `doit()` is to read and parse the request.

```
    /* Read request line and headers */
    Rio_readinitb(&rio, fd);
    if (!Rio_readlineb(&rio, buf, MAXLINE))
        return;
```

```
    printf("%s", buf);
    /* parse request */
    sscanf(buf, "%s %s %s", method, uri, version);

    /* simply read and ignore any request headers */
    read_requesthdrs(&rio);

    /* Parse URI from GET request */
    parse_uri(uri, hostname, port, filename);
```

- It should determine whether the client has sent a valid HTTP request; if so, it can then **establish** its own connection to the appropriate web server then **request** the object the client specified. Thus, the second part of `doit()` is to check. Since I only implemented GET request, `doit()` only checks whether the method is GET or not.
- Finally, proxy should **read** the server's response and **forward** it to the client. Hence, the third part of `doit()` is as follows.

```
    // the proxy serves as client now
    int clientfd = Open_clientfd(hostname, port);
    // send request to server
    write_request(clientfd, method, filename, hostname, port);

    Rio_readnb(&rio, buf, rio.rio_cnt);
    // read response from the server
    rio_t rio_server;
    Rio_readinitb(&rio_server, clientfd);
    // send server response to client
    while( Rio_readlineb(&rio_server, buf, MAXLINE) != 0 )
        Rio_writen(rio.rio_fd, buf, strlen(buf));

    Close(clientfd);
```

request line parser

URI is the suffix of the corresponding URL that includes the filename and optional arguments. e.g.

- URL: http://localhost:15213/home.html
- URI: localhost:15213/home.html

```
void parse_url(char *url, char *hostname, char *port, char *filename)
{
    char *ptr;
    // ignore http://
    if( strstr(url, "http://") ) url = url + 7;

    // find the filename
    if ((ptr = index(url, '/'))) {
        strcpy(filename, ptr);
        *ptr = '\0';
    }
    else {
        strcpy(filename, "/index.html"); // default home page
    }

    // find the port
    if ((ptr = index(url, ':'))) {
        strcpy(port, ptr+1);
        *ptr='\0';
    }

    // the rest is the hostname
    strcpy(hostname, url);
}
```

Request headers

- how to read the request header from the client
- how to send request header to the server

```
void write_request(int fd, char *method, char *filename, char *hostname, char
*port)
{
    char buf[MAXLINE];

    sprintf(buf, "%s %s HTTP/1.0\r\n", method, filename);
    // user is very likely not to provide port
    if (port[0] != '\0')
```

```
        sprintf(buf, "%sHOST: %s:%s\r\n", buf, hostname, port);
    else
        sprintf(buf, "%sHOST: %s\r\n", buf, hostname);
    sprintf(buf, "%sConnection: close\r\n", buf);
    sprintf(buf, "%sProxy-Connection: close\r\n", buf);
    sprintf(buf, "%s%s\r\n", buf, user_agent_hdr);
    Rio_writen(fd, buf, strlen(buf));
    return;
}
```

Port numbers

- HTTP request ports is an optional field in the URL of an HTTP request.
- proxy has two ports: one for client, and one for server

Reminder

wrapper functions `Rio_` call `unix_error` when error happens, which will call `exit(0)` . This will make proxy terminate. Because once a server begins accepting connections, it is not supposed to terminate. Simply comment `exit(0)` out and leave program return. Calling a thread-level exit, e.g., `pthread_exit(NULL)`, may have some unexpected results.

When sending data to a closed socket connection twice, the system kernal will raise a SIGPIPE signal. The default action of SIGPIPE handler is terminating the process.

Part I debug

First, test PXYDRIVE

```
xy@xy-vm:~/Desktop/proxylab-handout$ ./pxy/pxydrive.py -p ./proxy -f s01-basic-fetch.cmd
xy@xy-vm:~/Desktop/proxylab-handout$ ./pxy/pxydrive.py -p ./proxy -f  s02-basic-request.cmd
xy@xy-vm:~/Desktop/proxylab-handout$ ./pxy/pxydrive.py -p ./proxy -f s03-overrun.cmd
```

Second, test `./driver.sh`

```
xy@xy-vm:~/Desktop/proxylab-handout$ ./driver.sh
*** Basic ***
Starting tiny on 10138
Starting proxy on 21340
1: home.html
   Fetching ./tiny/home.html into ./.proxy using the proxy
   Fetching ./tiny/home.html into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
2: csapp.c
   Fetching ./tiny/csapp.c into ./.proxy using the proxy
   Fetching ./tiny/csapp.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
3: tiny.c
   Fetching ./tiny/tiny.c into ./.proxy using the proxy
   Fetching ./tiny/tiny.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
4: godzilla.jpg
   Fetching ./tiny/godzilla.jpg into ./.proxy using the proxy
   Fetching ./tiny/godzilla.jpg into ./.noproxy directly from Tiny
   Comparing the two files
   Failure: Files differ.
5: tiny
   Fetching ./tiny/tiny into ./.proxy using the proxy
   Fetching ./tiny/tiny into ./.noproxy directly from Tiny
   Comparing the two files
   Failure: Files differ.
Killing tiny and proxy
basicScore: 24/40
```

注释掉./driver.sh里的`clear_dirs`调用，以保留测试结果。测试结果保存在`./.proxy`和`./.noproxy`两个隐藏路径中。对比发现两个`godzilla.jpg`的确不一样，经由proxy得到的`godzilla.jpg`丢失了很多内容，感觉是由proxy发送到客户端的过程中出了问题。

尝试将`doit()`中`Rio_writen(rio.rio_fd, buf, strlen(buf));`改为`Rio_writen(rio.rio_fd, buf, n)`。bug修复！

```
xy@xy-vm:~/Desktop/proxylab-handout$ ./driver.sh
*** Basic ***
Starting tiny on 1418
Starting proxy on 20064
1: home.html
   Fetching ./tiny/home.html into ./.proxy using the proxy
   Fetching ./tiny/home.html into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
2: csapp.c
   Fetching ./tiny/csapp.c into ./.proxy using the proxy
   Fetching ./tiny/csapp.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
3: tiny.c
   Fetching ./tiny/tiny.c into ./.proxy using the proxy
   Fetching ./tiny/tiny.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
4: godzilla.jpg
   Fetching ./tiny/godzilla.jpg into ./.proxy using the proxy
   Fetching ./tiny/godzilla.jpg into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
5: tiny
   Fetching ./tiny/tiny into ./.proxy using the proxy
   Fetching ./tiny/tiny into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
Killing tiny and proxy
basicScore: 40/40
```

Part II: Dealing with multiple concurrent requests

alter the above sequential proxy to simultaneously handle multiple requests. This part is the easiest.

How to implement a concurrent server?

- spawn a new thread to handle each new connection request. CSAPP 12.3.8

- the prethreading server. CSAPP 12.5.5.
- etc.

Note:

- threads should run in **detached mode** to avoid memory leaks.
- The `open_clientfd` and `open_listenfd` functions described in the CS:APP3e textbook are based on the modern and protocol-independent `getaddrinfo` function, and thus are thread safe.

Solution 1: spawn a new thread to handle each new connection request.

```
int main(){
    /* skip */
    pthread_t tid;
    // open a listening socket
    listenfd = Open_listenfd(argv[1]);
    // execute the typical infinite server loop
    while (1) {
        clientlen = sizeof(clientaddr);
        // to avoid the potentially deadly race
        connfdp = Malloc(sizeof(int));
        // repeatedly accept a connection request
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen, hostname, MAXLINE, port,
MAXLINE, 0);
        printf("Accepted connection from (%s, %s)\n", hostname, port);
        // perform a transaction
        pthread_create(&tid, NULL, thread, connfdp);
    }
}

void* thread(void *vargp){
    int connfd = *((int *) vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    doit(connfd);
    Close(connfd);
}
```

Part II debug

```
xy@xy-vm:~/Desktop/proxylab-handout$ ./pxy/pxydrive.py -p ./proxy -f basic-concurrency.cmd
xy@xy-vm:~/Desktop/proxylab-handout$ ./pxy/pxydrive.py -p ./proxy -f  mixed-concurrency.cmd
```

ALL TESTS PASSED

Part III: Caching web objects

Add a cache to your proxy that stores recently-used Web objects in memory.

Why Cache?

- proxy receives a web object from a server,
- proxy cache this web object in memory as it transmits the object to the client.
- If another client requests the same object from the same server, it can simply resend the cached object.

Note:

- can't cache every object that is ever requested, otherwise, an unlimited amount of memory is required.
- should have both **a maximum cache size** and **a maximum cache object size**. Otherwise, one giant object will consume the entire cache.
- `MAX_CACHE_SIZE = 1 MiB` only the actual web objects
- `MAX_OBJECT_SIZE = 100 KiB`

拓展: HTTP Cache的详细讲解[2]

Handout Suggestions on how to implement Cache

- allocate a buffer for each active connection and accumulate data as it is received from the server.
- If the size of the buffer ever exceeds the `MAX_OBJECT_SIZE`, the buffer can be discarded.
- If the entirety of the web server's response is read before the maximum object size is exceeded, then the object can be cached. Using this scheme, the maximum

amount of data your proxy will ever use for web objects is the following:

$$MAX\_CACHE\_SIZE + T * MAX\_OBJECT\_SIZE$$

where T is the maximum number of active connections. (I have no idea on how to use this.)

- employ an **approximating LRU** eviction policy, e.g., clock algorithm.
- How to implement synchronization?
  - protecting accesses to the cache with one large exclusive lock is not an acceptable solution.
  - partitioning the cache, using Pthreads readers-writers locks,
  - or using semaphores to implement your own readers-writers solution.
  - Note that **both reading** an object and **writing** it count as using the object.

My approach

Design data structure and functions

`cache`可以是一个数组，其中元素是自定义的`cache_block`结构体。

1. 首先根据功能来设计接口函数:
   简单来讲，我们需要在当proxy收到client request, 根据URL查找cache → `cache_find()`
   - 找不到时，说明是个全新的request，那么存入proxy 接收到的新的server response及URL → `cache_put()`
   - 找到时，直接将保存的server response发给client
2. 最简单的做法就是声明一个全局变量`cache`
3. 然后根据需求设计结构体:
   - `cache_block`需要存储server response和URL → 两个字符串`cache_obj`和`cache_url`
   - LRU → `timestamp`，越小越是更近才访问过，0最小
   - synchronization，采用CSAPP 12.5.4 对第一类读者-写者问题的解决方法 → `readcnt`, `mutex`, `w`
   - 在遍历`cache`查找时，多一个flag会更容易判断 → `isEmpty`
4. C中没有类，可以用一个初始化函数→ `cache_init()`
5. LRU → `cache_eviction()`, `cache_update()`

```
typedef struct
{
    int isEmpty;
    int stamp;
    int readcnt;
```

```
    sem_t mutex;
    sem_t w;
    char cache_obj[MAX_OBJECT_SIZE];
    char cache_url[MAXLINE];
} cache_block;

// global variable
cache_block cache[CACHE_OBJS_COUNT];    // cache
char ret_obj[MAX_OBJECT_SIZE];          // the requested object

/* public cache function */
void cache_init();
const char * cache_find(char *url);
void cache_put(char *url,char *buf);
/* private cache functions */
int cache_eviction();
void cache_update(int idx);
```

function declaration

`cache_init()`

只需对每一个block初始化即可。注意`timestamp`及信号量的初值选择。

`cache_find()`

- 说白了就是遍历`cache`。
- 这是读操作，要注意信号量的加锁和解锁。
- 当查找到时，直接返回`cache`中的`cache_block`会再次产生读者-写者问题，那就不如在查找过程中，将想要的block内容复制到另一个字符串中，再返回这个字符串的地址。→`ret_obj`
- 如果找到了，势必要访问这个block，那么就要更新LRU timestamp →call `cache_update()`

`cache_put()`

- 首先选出插入位置→ call `cache_eviction()`。
- 这是写操作，注意信号量的加锁和解锁。
- 写操作也要更新LRU的timestamp→call `cache_update()`

`cache_eviction()`

- 遍历`cache`找出时间戳最小的block
- 读操作，要特别注意信号量
- 如果有空的block，直接返回index，注意返回前也要操作信号量
- 如果没有，那就选出时间戳最大的那个

`cache_update()`

- 遍历`cache`更改每个block的时间戳
- 一个block的时间戳归零，其余+1
- 写操作，注意信号量

Improvement

可以用链表，这样就不用每次都遍历一遍了。

---

1. http://www.cs.cmu.edu/afs/cs/academic/class/15213-f18/www/recitations/↩↩↩↩
2. https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html↩