Proxy Lab

수리과학부 2019-16022 박채연

1. Proxy lab의 목적

Proxy를 이용하여 HTTP requests를 파싱하여 웹 서버로 전달하며 서버로부터 응답을 받아 이를 다시 클라이언트에 전달하는 프로그램을 작성함으로써 HTTP operation과 관련된 개념을 학습하고 Proxy의 동작 원리를 이해한다. thread를 기반으로 여러 개의 requests를 처리할 수 있는 프로그램을 작성함으로써 병렬 처리가 가능한 프로그램의 작성법을 익힌다. 마지막으로 Proxy의 cache를 구현함으로써 웹 서버와 웹 브라우저 사이에서 작동하는 Proxy의 다양한 이점에 대해 이해하고자 한다.

2. 실행 결과

```
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming/proxylab-handout$ ./port-for-user.pl stu9
stu9: 24714
```

'./port-for-user.pl stu9' 실행 결과 24714라는 포트 번호를 얻었다. 따라서 24714를 proxy의 port 번호로, 24715를 tiny 웹 서버의 port 번호로 사용하였다. 한 개의 터미널에서 './proxy 24714'를 실행시키고 다른 터미널에서 './tiny 24715'를 실행시킨 후, 또 다른 터미널에서 'curl -v –proxy http://localhost:24714 http://localhost:24715/home.html'을 실행시킨 결과는 아래와 같다.

```
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming/proxylab-handout$ ./proxy 24714
```

```
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kidsl/lab1/SNU-System-Programming/proxylab-handout/tiny$ ./tiny 24715
Accepted connection from (localhost, 45528)
GET /home.html HTTP/1.0
Host: localhost:24715
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3) Gecko/20120305 Firefox/10.0.3
Accept: */*
Proxy-Connection: close

Response headers:
HTTP/1.0 200 OK
Server: Tiny Web Server
Connection: close
Content-length: 120
Content-type: text/html
```

```
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/windows/system32$ curl -v --proxy http://localhost:24714 http://localhost:24715/home.html
*   Trying 127.0.0.1:24714...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 24714 (#0)
> GET http://localhost:24715/home.html HTTP/1.1
> Host: localhost:24715
> User-Agent: curl/7.68.0
> Accept: */*
> Proxy-Connection: Keep-Alive
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: Tiny Web Server
< Connection: close
< Content-length: 120
< Content-type: text/html
<
<html>
<head><title>test</title></head>
<body>
<img align="middle" src="godzilla.gif">
Dave O'Hallaron
</body>
</html>
* Closing connection 0
```

다음으로, './driver.sh'를 통해 자동채점한 결과는 아래와 같다.

```
chaeyeon@LAPTOP-4V7LQ0S5:/mnt/c/users/kidsl/lab1/SNU-System-Programming/proxylab-handout$ ./driver.sh
*** Basic ***
Starting tiny on 17361
Starting proxy on 26700
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
```

```
*** Concurrency ***
Starting tiny on port 17286
Starting proxy on port 23022
Starting the blocking NOP server on port 11644
Trying to fetch a file from the blocking nop-server
Fetching ./tiny/home.html into ./.noproxy directly from Tiny
Fetching ./tiny/home.html into ./.proxy using the proxy
Checking whether the proxy fetch succeeded
Success: Was able to fetch tiny/home.html from the proxy.
Killing tiny, proxy, and nop-server
concurrencyScore: 15/15

*** Cache ***
Starting tiny on port 26008
Starting proxy on port 20076
Fetching ./tiny/tiny.c into ./.proxy using the proxy
Fetching ./tiny/home.html into ./.proxy using the proxy
Fetching ./tiny/csapp.c into ./.proxy using the proxy
Killing tiny
Fetching a cached copy of ./tiny/home.html into ./.noproxy
Failure: Was not able to fetch tiny/home.html from the proxy cache.
Killing proxy
cacheScore: 0/15

totalScore: 55/70
```

3. 어떻게 구현했는지

pat1과 part2는 Chapter 14 ppt pp30-31를 참고하여 main 함수와 thread_func 함수를 작성하였고, thread_assist 함수는 'tiny.c'의 doit 함수의 구조를 참고하여 작성하였다. part3은 캐시와 관련된 부분인데 관련 코드를 강의 자료에서 찾을 수 없어 학습에 많은 시간의 투자가 필요해보였다. 그러나 아직 기말고사가 모두 끝나지 않은 상태라 많은 시간을 투자하기에는 무리라고 생각해 구현하지 않았다. part1와 part2의 구현 방법은 아래와 같다.

- main

```c
/* main */
int main(int argc, char **argv) {
    struct sockaddr_in clientaddr;
    pthread_t tid;

    int clientlen = sizeof(clientaddr);
    int listenfd = Open_listenfd(argv[1]);
    while(1){
        // to avoid unintentional sharing
        int *connfdp = (int *)Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        // to handle multiple concurrent requests
         Pthread_create(&tid, NULL, thread_func, connfdp);
    }
    return 0;
}
```

위의 이미지는 main 함수이다. Open_listenfd를 통해 listenfd를 create하고, while 문에서 malloc을 이용하여 int 형 포인터인 connfd를 동적할당 한 후 connection 요청을 accept한 다음 Pthread_create 함수를 통해 각각의 request를 한 개의 thread가 handling 할 수 있도록 하였다. 이때, 각각의 thread가 한 개의 request를 handling 하기 때문에 여러 개의 요청을 동시에 처리할 수 있다. 또한, connfd가 지역 변수일 경우 thread들이 이를 공유하여 race 관련 문제가 발생할 수 있기 때문에 connfd를 동적할당하였다.

- thread func

```c
/* thread_func : handle one request */
void *thread_func(void *vargp){
    int connfd = *((int *)vargp);
    // run in detached mode to avoid fatal memory leak
    Pthread_detach(Pthread_self());
    // to close parent's connfd (threaded program)
    Free(vargp);
    thread_assist(connfd);
    Close(connfd);
    return NULL;
}
```

thread_func 함수는 main 함수의 while 문 내에서 thread를 생성할 때 호출되는 함수이다. 우선 memory leak 문제를 방지하기 위해 detached 상태로 바꾸고, listening server의 connfd를 close하기 위해 vargp를 동적할당 해제 하였다. 다음으로 thread_assist 함수를 호출하여 request를 handling 하고 connfd를 close하였다.

- thread_assist

```c
/* thread_assist: read and parse requests, forward requests to web servers,
 * read the servers' responses and forward those responses to clients */
void thread_assist(int connfd){
    int n, serverfd;
    char buf[MAXLINE], buf2[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
    char host[MAXLINE], portn[20], detailed[MAXLINE];
    rio_t rio, rio2;

    // to initialize struct rio_t and readline from it
    Rio_readinitb(&rio, connfd);
    if (!Rio_readlineb(&rio, buf, MAXLINE))
        return;
```

thread_assist 함수는 requests를 읽어 파싱하고, 이를 웹 서버로 전달한 후 서버로부터의 응답을 다시 클라이언트로 전달하는 역할을 수행한다. 위의 이미지에서는 우선 rio 구조체를 초기화하고 request를 읽어 들였다.

```c
// e.g. If "GET http://localhost:12345/home.html HTTP/1.1" is stored in buf,
// method = "GET", uri = "http://localhost:12345/home.html", version = 'HTTP/1.1'
sscanf(buf, "%s %s %s", method, uri, version);

// Only 'GET' method is implemented in this lab
if (strcasecmp(method, "GET")){
    printf("%s is not implemented\n", method);
    return;
}

// parse uri "http://localhost:12345/home.html" into host('localhost'), portn('12345'), and detailed('/home.html')
parse_uri(uri, host, portn, detailed);
```

다음으로 sscanf 함수를 사용하여 읽어들인 request를 method, uri, version으로 파싱하였다. 만약 request가 'GET http://localhost:12345/home.html HTTP/1.1'이라면 공백을 기준으로 파싱하여 method는 GET, uri는 http://localhost:12345/home.html, version은 HTTP/1.1 값을 가지게 된다. 한편, 본 랩과제에서는 GET request에 대해서만 다루고 있으므로, 만약 method 값이 GET이 아니라면 관련 메시지를 출력하고 return 하였다. 다음으로 parse_uri 함수를 이용하여 uri인 http://localhost:12345/home.html을 다시 host, portn, detailed로 파싱하였다. 이때, host는 localhost, portn은 12345, detailed는 /home.html 이라는 값을 가진다.

```c
// to create connection with server and initialize struct rio_t
serverfd = Open_clientfd(host, portn);
Rio_readinitb(&rio2, serverfd);

// to write 'GET /home.html HTTP/1.0' in the server
sprintf(buf2, "%s %s %s", "GET", detailed, http_msg);
Rio_writen(serverfd, buf2, strlen(buf2));
```

다음으로 open_clientfd 함수를 통해 서버와 connect하고 rio 패키지를 초기화하였다. 다음으로 serverfd에 "GET /home.html HTTP/1.0" 을 write 하였다. 이때, 실제 HTTP request의 HTTP version과 관계없이 항상 "HTTP/1.0"을 서버로 보내라고 되어있으므로 유의해야 한다.

```c
// to send http requests header to the server
int flagu = 0, flaga = 0, flagc = 0, flagpc = 0;
while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0){
    // end of line
    if (!strncmp(buf, "\r\n", 2)){
        Rio_writen(serverfd, buf, strlen(buf));
        break;
    }
    // host
    else if (!strncasecmp(buf, "Host", 4)){
        flagu = 1;
        Rio_writen(serverfd, buf, strlen(buf));
    }
    // user-agent
    else if (!strncasecmp(buf, "User-Agent", 10)){
        flaga = 1;
        Rio_writen(serverfd, user_agent_hdr, strlen(user_agent_hdr));
    }
    // connection
    else if (!strncasecmp(buf, "Connection", 10)){
        flagc = 1;
        Rio_writen(serverfd, connection_msg, strlen(connection_msg));
    }
    // proxy-connection
    else if (!strncasecmp(buf, "Proxy-Connection", 15)){
        flagpc = 1;
        Rio_writen(serverfd, proxyconnection_msg, strlen(proxyconnection_msg));
    }
    // other headers
    else{
        Rio_writen(serverfd, buf, strlen(buf));
    }
}
```

위의 이미지는 HTTP request header를 client로부터 읽어들여 server로 전달하는 부분이다. 이때, 주어진 조건에 의해 Host, User-Agent, Connection 과 Proxy-Connection 은 무조건 전달되어야 하므로 flagu, flaga, flagc, flagpc를 사용하여 서버로 전달하였는지 여부를 체크하였다. 한편, Host가 HTTP request에 포함되어 있으면 이를 그대로 전달해야하고, User-Agent, Connection, Proxy-Connection 값은 실제 HTTP request 값과 상관없이 정해진 값을 전달해야한다. 또한, 그 외의 헤더가 있을 경우 이를 그대로 전달해야한다.

```c
// send 'Host' request header if didn't
if (!flagu){
    char temp[20];
    sprintf(temp, "%s%s\r\n", host_msg, host);
    Rio_writen(serverfd, temp, strlen(temp));
}
// send 'User-Agent' request header if didn't
if (!flaga){
    Rio_writen(serverfd, user_agent_hdr, strlen(user_agent_hdr));
}
// send 'Connection' request header if didn't
if (!flagc){
    Rio_writen(serverfd, connection_msg, strlen(connection_msg));
}
// send 'Proxy-Connection' request header if didn't
if (!flagpc){
    Rio_writen(serverfd, proxyconnection_msg, strlen(proxyconnection_msg));
}
```

다음으로, flagu, flaga, flagc, flagpc 값을 체크하여 Host, User-Agent, Connection, Proxy-Connection 헤더를 보냈는지 여부를 체크하고 그렇지 않은 경우 이를 server로 전달하였다.

```c
// to forward responses from the server to the client
while((n = Rio_readlineb(&rio2, buf, MAXLINE)) != 0){
    Rio_writen(connfd, buf, n);
}
Close(serverfd);
```

마지막으로, 서버로부터의 응답을 다시 client로 전달하고 serverfd를 close 하였다.

- parse_uri

parse_uri 함수는 uri를 파싱하여 host 이름, port 번호, 파일 경로를 구하고 이를 각각 host, portn, detailed에 저장하는 함수이다.

```c
/* parse_uri : parse uri into host name, port number and file path */
void parse_uri(char *uri, char *host, char *portn, char *detailed){

    char *ptr;

    if (!(ptr = strstr(uri, "http://"))){
        ptr = uri;
        if (*ptr == '/')
            ptr++;
    }
    else
        ptr += 7;

    // host name
    int i = 0;
    while(*ptr != ':' && *ptr != '/'){
        host[i] = *ptr;
        ptr++;
        i++;
    }
    host[i] = 0;

    // port number
    if (*ptr != ':'){
        portn[0] = '8';
        portn[1] = '0';
        portn[2] = 0;
    }
    else{
        i = 0;
        ptr++;
        while (*ptr != '/' && *ptr != ' '){
            portn[i] = *ptr;
            i++;
            ptr++;
        }
        portn[i] = 0;
    }
```

```c
// file path
i = 0;
if (*ptr == '/'){
    while(*ptr){
        detailed[i] = *ptr;
        i++;
        ptr++;
    }
}
detailed[i] = 0;
```

4. 어려웠던 점

tiny.c에 구현되어 있는 parse_uri 함수를 참고하여 'proxy.c'에 적합한 parse_uri 함수를 구현하였다. 'GET http://localhost:12345/godzilla.jpg HTTP/1.1' 과 같은 HTTP 요청을 thread_assist 함수 내에서 공백을 기준으로 parsing 하여 각각 method, uri, version에 저장하게 되는데, uri는 조금 더 세부적인 정보로 구분될 수 있다. parse_uri 함수는 uri를 host 이름, port 번호, 파일 경로로 각각 구분해 주는 역할을 하는데, 이때, char 형 포인터인 ptr을 사용하였다. 아래 이미지의 while 문 내에서 *ptr 대신 ptr을 사용하는 실수를 저질렀고, 그 결과 proxy 실행파일을 실행했을 때 Segmentation fault가 떴다. 처음에 짠 코드가 굉장히 복잡해서 Segmentation fault가 뜰 만한 곳을 찾는데 오래걸렸고, 굉장히 사소한 실수임에도 불구하고 간단한 코드이다보니 '여기서 틀렸을 리 없다'는 생각 때문에 해당 실수를 찾아내기까지 거의 1시간이 걸렸다.

```
while(*ptr != ':' && *ptr != '/'){
    host[i] = *ptr;
    ptr++;
    i++;
}
```

올바르게 동작하는 'proxy.c'를 구현하기 위해서는 'tiny.c'의 동작 원리를 정확히 이해해야 했었다. 'proxy.c'의 main 문에서 connection file descriptor를 생성하고, thread_assist 함수 내에서 서버에 해당하는 file descriptor를 생성한다. 이들을 각각 connfd, serverfd라고 하자. 'tiny.c' 코드를 읽어보면 'GET http://localhost:12345/godzilla.jpg HTTP/1.1'과 같은 HTTP 요청을 받았을 때, 'GET /godzilla.jpg HTTP/1.0'을 serverfd에 write해야 함을 알 수 있는데, 처음에 실수로 공백을 제거하고 'GET/godzilla.jpg HTTP/1.0'을 write하여 아래와 같은 메시지가 출력되었다.

```
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/windows/system32$ curl -v --proxy http://localhost:1111 http://localhost:12345/godzilla.
jpg
*   Trying 127.0.0.1:1111...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 1111 (#0)
> GET http://localhost:12345/godzilla.jpg HTTP/1.1
> Host: localhost:12345
> User-Agent: curl/7.68.0
> Accept: */*
> Proxy-Connection: Keep-Alive
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 501 Not Implemented
< Content-type: text/html
< Content-length: 168
<
<html><title>Tiny Error</title><body bgcolor=ffffff>
501: Not Implemented
<p>Tiny does not implement this method: GET/godzilla.jpg
<hr><em>The Tiny Web server</em>
* Closing connection 0
```

아래는 위의 이미지와 같은 출력을 발생시킨 코드를 'tiny.c'에서 찾은 것이다. 공백을 주지 않아 발생한 문제라는 사실을 몰랐던 상태에서 실제로 method에 'GET/gozilla.jpg'가 저장되어 있는지 확인해보고 싶어 method를 출력하는 코드를 추가하여 확인해보고자 하였다.

```
if (strcasecmp(method, "GET")) {                    //li
    clienterror(fd, method, "501", "Not Implemented",
                "Tiny does not implement this method");
    return;
}                                                   //li
```

그러나 해당 코드를 추가하고 tiny 폴더 내에서 make 명령어를 입력하면 이미 주어진 tiny 실행 파일이 모두 사라지고 Makefile error가 떴다. gcc로 직접 컴파일을 시도해보아도 에러가 뜨긴 마찬가지였다. 따라서 'tiny.c' 파일을 변형하여 테스트를 해볼 수 없어 디버깅하기에 까다로웠다.

또한, sizeof 연산자와 strlen 함수의 차이를 혼동하여 계속 헤맸다. 아래는 'proxy.c' 파일 내의 thread_assist 함수 코드 중 일부이다. serverfd에 HTTP requests를 write 하는 과정에서 다른 부분은 모두 strlen 함수를 사용하였지만 아래의 코드에서만 Rio_writen(serverfd, buf2, strlen(buf2)) 대신 Rio_writen(serverfd, buf2, sizeof(buf2))를 썼었는데, 이로 인해 part1에서 test4, test5를 통과하지 못해 24/40의 점수가 나왔다. test4, test5에 대해 proxy와 nonproxy의 결과로 얻은 두 파일이 다르다는 메시지가 출력되었는데, 실제로 주소창에 http://localhost:12345/godzilla.jpg'를 입력하면 이미지 사진을 얻을 수 있었지만, curl 명령어를 이용하여 확인해보고자 했을 때는 이를 제대로 handling 하지 못했다.

```
sprintf(buf2, "%s %s %s", "GET", detailed, http_msg);
Rio_writen(serverfd, buf2, strlen(buf2));
```

내가 작성한 프로그램이 논리적으로 옳다고 생각하여 문제의 원인을 이해하기 어려웠다. 혹시 image 파일은 html 파일과 다른 방식으로 처리해주어야 하는지 등 사소한 것일지라도 거의 모든 것을 의심하며 코드를 고쳐보다가 하나의 Rio_writen 함수에서 strlen 함수 대신 sizeof 연산자를 사용했다는 사실을 발견하였다. 이를 strlen 함수로 바꾸어주니 part1에서 40/40이라는 점수를 얻을 수 있었다. (sizeof 연산자는 배열의 전체 크기를 반환하지만 strlen 함수는 문자열의 처음부터 null character 까지의 길이를 반환한다.)

```
while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0){
    if (!strncmp(buf, "\r\n", 2)){
        Rio_writen(serverfd, buf, strlen(buf));
        break;
    }
```

마지막으로, HTTP request header를 server로 보낼 때 tiny로부터 "₩r₩n"을 읽어들이면 이를 serverfd에 write하지 않고 바로 break문을 통해 while문을 빠져나가도록 했었는데, 이로 인해 './driver.sh'를 실행시키면 timeout이 발생하는 문제가 있었다. 따라서 "₩r₩n"을 serverfd에 write까지 한 후 while 문을 빠져나가도록 하였고 그 결과 timeout 문제를 해결할 수 있었다.

5. 새롭게 배운 점 및 신기했던 점

'./driver.sh' 를 통해 자동으로 채점할 수도 있지만 출력 결과를 확인하기 위해 리눅스의 curl 명령어를 통해 직접 테스트를 진행하였다. 총 3개의 터미널이 필요한데, 한 개는 './proxy' 실행 파일을 실행시키기 위함이고 다른 한 개는 './tiny' 실행 파일을 실행시키기 위함이다. 마지막 한 개의 터미널은 'curl' 명령어를 사용하여 프록시 서버를 통해 데이터를 전송하기 위함이다. 본 랩 과제를 수행하며 'curl' 명령어의 역할 및 특징 등을 새로 알게 되었다. 'curl'은 HTTP, HTTPS 등의 다양

한 프록시를 지원하는데, 프록시 서버를 통해 데이터를 전송하기 위해서는 '—proxy' 옵션을 사용한다. 'curl -v –proxy http://localhost:1111 http://localhost:2222/home.html'은 http://localhost:1111에서 프록시 서버를 사용하여 http://localhost:2222/home.html에 있는 리소스를 다운로드한다.

sizeof 연산자와 strlen 함수의 반환값 차이를 정확히 이해하게 되었다. char 형 배열인 'char arr[2000]' 이 정의되어 있다고 가정하자. sizeof(arr)은 배열에 저장된 값에 관계없이 arr 배열의 전체 크기인 1*2000 = 2000을 반환한다. 한편, strlen(arr)은 null character가 저장된 위치에 따라 값이 달라지는데, 만약 arr[0] = '1', arr[1] = '2', arr[2] = '₩0'이라면 strlen(arr)은 2를 반환한다.

```
chaeyeon@LAPTOP-4V7LOOS5:/mnt/c/windows/system32$ curl -v --proxy http://localhost:24714 http://localhost:24715/home.html
*   Trying 127.0.0.1:24714...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 24714 (#0)
> GET http://localhost:24715/home.html HTTP/1.1
> Host: localhost:24715
> User-Agent: curl/7.68.0
> Accept: */*
> Proxy-Connection: Keep-Alive
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: Tiny Web Server
< Connection: close
< Content-length: 120
< Content-type: text/html
<
<html>
<head><title>test</title></head>
<body>
<img align="middle" src="godzilla.gif">
Dave O'Hallaron
</body>
</html>
* Closing connection 0
```

GET의 Request header 및 Response header에 대해 자세히 알게되었다. 위의 사진은 Request와 Response의 구조를 잘 보여주는데, '>' 표시는 Request를, '<' 표시는 Response를 의미한다. Request에서 GET 다음 줄에는 Request header가 표시되는데, 이는 웹 브라우저가 웹 서버에 요청하는 내용을을 텍스트로 변환한 메시지들이다. 'Host'는 요청하려는 서버 호스트의 이름과 포트 번호를, 'User-Agent'는 클라이언트 프로그램 정보를, 'Accept'는 클라이언트가 처리 가능한 미디어 타입을 의미한다. 'Proxy-Connection'은 HTTP 연결 상태를 의미하는 Connection과 비슷한 일을 수행하는데, 오늘날 많은 HTTP 시스템에서 이 헤더를 사용하는 것을 지양한다.

Response에서 'HTTP/1.0 200 OK'는 HTTP 요청이 성공적이었음을 의미한다. Response header는 웹 서버가 브라우저에 응답하는 내용을 텍스트로 변환한 메시지로, 'Server'는 웹 서버의 종류를, 'Content-length'는 컨텐츠의 길이를, 'Content-type'는 컨텐츠의 종류를 의미한다.

또한, 본 과제에서 캐시를 구현하지는 않았지만 프록시가 캐싱을 통해 웹 서버와 브라우저 사이에서 네트워크 병목 현상을 줄여주는데 효과적으로 기여함을 알게 되었다.