M1522.000800 System Programming
Fall 2018

System Programming Proxy Lab Report

Woohyeon Baek
2017-15782

1. <lab> Proxy Lab

프록시 랩은 HTTP 서버와 프록시 서버의 역할과 방식에 대한 이해를 목표로 한다. HTTP는 HyperText Transfer Protocol의 약자로 클라이언트와 서버 간 정보 교환 방식을 정의한다. HTTP 서버는 헤더와 바디로 구성된 HTTP 형식의 요청을 받아 그에 해당하는 응답을 클라이언트에 보낸다. 그리고 프록시 서버는 클라이언트와 서버 사이에서 매개체로서 작동하며 방화벽, 웹 정보 캐싱 등의 역할을 수행한다. 간단한 HTTP 서버와 프록시 서버를 직접 구현해 봄으로써 클라이언트와 서버 간의 정보 교환 방식에 대해 깊게 학습하고자 한다.

2. Implementation

랩은 크게 HTTP 서버의 구현, 프록시 서버의 구현, 그리고 프록시 서버에 캐시 장착, 이렇게 세 가지로 구성된다.

2.1 Implementing an HTTP Server

HTTP 서버는 매우 간단한 형태로 http.c에 구현되어 있으며 클라이언트로부터 request를 읽는 doit 함수와 요청한 정보를 response로 보내는 server_static 함수로 나뉜다. 맨 처음에 서버는 클라이언트와 연결이 될 때까지 대기한다. 클라이언트와 연결되어 request가 날아왔다면 doit 함수에서 request를 읽고 그것을 서버가 지원 가능한지부터 확인한다. 이에 필요한 정보는 모두 request header의 맨 첫 줄에 있다. 이때 UNIX I/O를 사용하여 정보를 읽으면 short count 등의 이유로 원하는 정보를 끝까지 읽은 것을 보장할 수 없다. 따라서 CSAPP에서 미리 정의한 Robust I/O 중 Rio_readlineb을 사용하여 \r\n이 나타날 때까지 한 줄을 전부 읽었다. 앞으로 설명할 여타의 데이터 쓰기 및 읽기에 대해서도 전부 Robust I/O로 처리했다. 읽은 첫 줄을 method, URI, version 형식으로 파싱했다. 서버가 지원 가능한 method는 오직 GET 하나뿐이므로 그것과 다른 method들은 모두 501 Not implemented 에러 처리를 하였다. URI에서 파일 경로를 추출하여 UNIX stat 함수를 이용하여 그 경로에 실제 파일이 존재하지 않으면 404 Not found 에러를 띄웠다. 그리고 존재한다 하더라도 디렉토리나 소켓처럼 일반적인 파일이 아니거나 파일 접근 권한이 없는 경우 403 Forbidden 에러를 보냈다. 이 모든 것을 확인하고 나면 서버는 비로소 원하는 파일의 데이터를 static하게 제공할 수 있음이 보장된다.

serve_static 함수로 이동하여 클라이언트에게 보낸 response header와 body를 만들게 된다. 항상 HTTP/1.0 버전으로 response를 보내며 서버의 이름은 간단하게 HTTP Server로 했다. 파일의 타입은 html, jpg, png, gif일 때 각각 text/html, image/jpeg, image/png, image/gif로 했으며 그 이외의 파일 타입의 경우 test/plain으로 했다. response body를 보내는 방식은 Content-Length를 정하여 그만큼 데이터를 보내는 방식과 chunk로 끊어서 보내는 방식이 있다. 구현한

간단한 HTTP 서버는 항상 static한 데이터를 보내기 때문에 UNIX stat 구조체를 통해 해당 파일의 용량을 미리 알 수 있다. 따라서 Content-Length 방식으로 데이터를 전송하도록 했다. 파일로부터 mmap으로 데이터를 읽어서 Rio_writen으로 클라이언트에게 response를 보냈다. 그리고 소켓을 닫음으로써 클라이언트와의 연결을 끊었다. 이 일련의 과정을 계속해서 반복하게 된다.

2.2 Implementing a Proxy Server

프록시 서버도 HTTP 서버와 마찬가지로 클라이언트와 연결될 때까지 기다린다. 그리고 연결되면 클라이언트로부터 request를 받아 그것을 서버에 전달하고 서버로부터 response를 받아 그것을 클라이언트에 전달하게 된다. 이 과정은 전부 proxy.c 파일의 doit 함수에 구현되었다. Robust I/O로 request를 읽고 지원하지 않는 method나 URI의 경우 에러를 내는 것은 앞선 HTTP 서버와 유사하다. method가 GET이 아니면 501 Not implemented 에러를 보내며 URI를 파싱하여 호스트와 포트, 파일 이름을 알아내는 데에 실패한 경우 400 Bad request 에러를 처리한다. 그리고 알아낸 호스트와 포트를 이용하여 프록시 서버가 다른 서버와 연결된다. 프록시 서버는 다른 모든 서버들과 연결될 수 있으며 어떤 경우에도 GET method의 경우 일반적인 기능을 제공해야 한다. 따라서 HTTP 서버처럼 403 Forbidden이나 404 Not found 처리를 따로 하지 않고 그 역할을 다른 서버에게 맡겼다. 같은 이유로 프록시 서버는 맨 첫 줄만 읽는 구현한 HTTP 서버와 달리 Rio_readlinb를 반복적으로 수행하여 클라이언트로부터 request header 전체를 읽어서 프록시 서버에 보냈다.

클라이언트의 request를 모두 서버에게 보냈다면 이제 서버로부터 response를 읽어 클라이언트에게 보낼 response의 header와 body를 구성한다. Rio_readlineb 함수를 여러 번 불러서 response header를 읽는다. 물론 header의 마지막에는 “\r\n”이 포함되도록 했다. header를 읽는 과정에서 Transfer-Encoding: chunked 또는 Content-Length: <contentLength>가 있는지 확인해야 그 후에 해당 형식에 맞춰 response body를 읽고 구성할 수 있다. 단 Transfer-Encoding의 경우 다른 형식과 합쳐져서 Transfer-Encoding: gzip, chunked와 같은 형태로 들어올 수도 있기 때문에 문장에 Transfer-Encoding과 chunked가 있는지를 따로 확인했다. Content-Length라면 해당 contentLength를 파싱하여 그 길이만큼 정확히 Rio_readnb 함수로 읽는다. 주의해야 할 것은 Rio_readlineb 함수를 사용하여 줄 단위로 읽어서는 안된다는 것이다. Rio_readlineb는 \n이 나타날 때까지 읽고 그 뒤에 \0를 삽입하게 된다. 그러나 content에는 \n가 포함될 수 있어서 반드시 byte의 수로 body를 읽도록 했다.

body 정보가 chunked된 형태로 들어온다면 이를 decode해야 전체 body 정보를 읽을 수 있게 된다. chunked의 형식은 다음과 같다.

```
<chunk size (16진법으로 표현된 N)>\r\n
<chunk data (N bytes)>\r\n
……
0\r\n
```

즉, chunk size가 0이 나올 때까지 chunk size와 \r\n을 읽고 해당 크기만큼의 chunk data와 \r\n을 읽는 것을 반복한다. chunk size는 Rio_readlineb로 읽고 sscanf에서 %x를 이용하여 chunk size를 얻었다. chunk data에도 전 상황과 마찬가지로 \r이나 \n 등이 포함될 수 있기 때문에 Rio_readnb로 n bytes를 직접 읽도록 했다. 항상 끝에 \r\n도 같이 읽어야 하므로 chunk size를 2만큼 증가시켜 읽음으로써 구현을 단순화했다.

chunked되어 서버로부터 response body가 주어질지라도 그것을 Content-Length 형식으로 변환하거나 하지 않고 그대로 서버에 chunked된 형식으로 전달하도록 했다. 따라서 Content-Length의 계산은 https://tools.ietf.org/html/rfc2616에 나와있는 방식과는 달리 여러 개의 \r\n와 chunk size 문자열의 길이도 포함되도록 했다. 그리고 chunked된 body의 크기를 decode 과정에서 알 수 없으므로 최초의 chunk의 길이의 두 배만큼 malloc하고 메모리가 부족하면 필요한 메모리 양의 두 배만큼 realloc하는 방식으로 body를 구성하도록 했다. Rio_readnb로 읽은 데이터의 맨 끝에는 \0이 없으므로 strcpy 대신 memcpy를 이용하여 n bytes를 content로 복사했다. 클라이언트에게 보낼 response header와 body 구성이 완료되면 Rio_writen으로 클라이언트에게 해당 정보를 보낸다.

이렇게 구현된 프록시 서버를 통해 http://www.pactconf.org/index.php를 접속하면 맨 마지막에 클라이언트가 POST http://ocsp.digicert.com/ HTTP/1.1 request를 서버로 보내게 된다. 이것을 프록시 서버가 받아서 501 Not implemented와 함께 clienterror 함수를 호출하여 해결할 경우 첫번째 Rio_writen 함수 호출 이후 클라이언트에서 소켓을 바로 닫아버려 그 뒤 닫혀버린 소켓에 데이터 쓰기를 하여 broken pipe 에러와 함께 SIGPIPE signal이 날아와서 서버 프로세스가 죽어버리는 문제가 발생한다. 이는 HTTPS의 인증서와 관련된 것으로 파이어폭스에 해당 HTTP 서버를 프록시 하지 않겠다고 설정함으로써 해결했다.

2.3 Caching Web Objects

클라이언트와 서버가 통신하는 사이에 프록시 서버가 있으면 주고받는 데이터를 알 수 있다. 이를 캐시에 저장한다면 이후에 클라이언트가 요청한 정보가 캐시 hit되면 프록시 서버가 실제 서버에게 요청을 보내지 않아도 클라이언트에게 데이터를 전송할 수 있으므로 클라이언트 입장에서 더욱 빠르게 데이터를 받을 있게 된다. 따라서 구현한 프록시 서버에 캐시를 장착했다.

캐시는 linked list로 구현되어 있으며 각각의 원소는 cache_block이라는 struct로 되어 있다. cache_block은 URI, response, content, content_size, next로 구성된다. URI는 find_cache_block 함수를 통해 원하는 cache_block을 찾을 때 이용되며 response는 response의 header, content는 body를 의미한다. content_size는 content의 유효한 size를 의미하며 next가 다음 cache_block의 주소를 가리킴으로써 전체적으로 linked list가 구성되도록 하는 것이다. 추가로 linked list의 맨 뒤는 rear, 맨 앞은 front라는 포인터가 가리키며 rear를 이용하여 맨 뒤에 cache_block을

삽입한다. 그리고 object_size라는 사용되는 전체 캐시량에 대한 전역변수가 있어서 이것이 MAX_OBJECT_SIZE를 넘어가면 front부터 차례로 cache_block을 삭제하여, FIFO 방식으로 replacement policy를 유지한다. object_size에는 content_size만이 반영되며 response header와 같은 부차적인 데이터는 고려하지 않는다.

add_cache_block이 불리면 가장 먼저 content length가 MAX_CONTENT_SIZE를 넘는지 확인하여 넘으면 캐시에 해당 데이터를 삽입하지 않는다. 그 다음으로 데이터가 이미 캐시에 존재하는지를 URI의 비교를 통해 확인하며 캐시에 해당 데이터가 없을 경우에만 삽입하게 된다. cache_replacement_policy라는 함수를 불러서 MAX_OBJECT_SIZE를 넘지 않고 데이터가 삽입될 수 있도록 여유 공간을 만들어준다. 전 처리 과정이 끝났으면 블록을 malloc을 통해 생성하며 해당 정보들을 복사한다. 이때 content 배열의 맨 끝에는 \0이 없고 contentLength 변수가 content의 유효한 데이터 범위를 알려주므로 strcpy 대신 memcpy를 사용하여 데이터를 복사했다. object_size를 갱신하고 linked list에 해당 블록을 삽입하는 것으로 add_cache_block 함수가 끝이 난다.

프록시 서버 코드에 이 캐시 루틴을 추가했다. 클라이언트로부터 request header의 맨 첫 줄을 읽고 URI를 파싱한 다음 그것을 이용하여 find_cache_block 함수를 통해 캐시에 있는지를 판단한다. 만약에 있으면 서버와 연결하지 않고 바로 response와 content를 Rio_writen함으로써 데이터를 클라이언트에게 바로 전송한다. 만약 캐시에 없다면 앞에서 설명한 것과 같은 루틴이 작동된다. 그리고 맨 마지막에 서버로부터 얻은 데이터를 add_cache_block를 호출함으로써 캐시에 저장하게 된다.

3. Conclusion

HTTP 서버와 프록시 서버, 그리고 캐시 기능을 구현함으로써 HTTP 형식으로 데이터를 주고받는 방법과 프록시의 역할에 대해 좀 더 깊게 이해할 수 있었다. HTTP 서버를 구현하는 것은 책에도 코드가 어느 정도 나와있고 주석도 상세해서 난이도가 높지 않았다. 그러나 프록시 서버는 참고할 만한 책이나 문서가 없어서 난이도가 꽤 있었다. response body를 구성할 때 chunked된 형식을 decode하는 과정에서 Rio_readlineb로 읽거나 맨 끝에 \0이 없는데 strcpy나 strstr, strcat 함수를 사용하는 등 잘못된 방법을 써서 시간 소모가 많았다. 또한 POST http://ocsp.digicert.com/ HTTP/1.1 request 문제에도 시간을 많이 허비하여 eTL에서 답변을 받기 전까지 해결하는데 어려움이 있었다. HTTPS와 관련된 사전 문제들을 미리 핸드아웃에 추가했으면 좋을 것 같다. 캐시 부분은 오히려 코드가 그다지 길지 않고 컴퓨터구조 수업에서 배운 개념이 있어서 별로 어렵지 않았다.

현재 HTTP 서버와 프록시 서버가 단일 쓰레드로 구현되어 있다. 랩 과제를 수행하는 데에는 이 정도면 충분하지만 파이어폭스에서 동시에 웹 서버 접근이 이루어질 경우 하나씩 처리가 된다는 단점이 있다. pthread로 보완하면 더 뛰어난 서버가 될 것 같다.