M1522.000800 System Programming
Fall 2018

System Programming Proxy Lab Report

Daeeun Kim
2014-15395

1. Proxy Lab

Proxy Lab은 GET요청을 처리하는 간단한 HTTP서버를 만들어 보고, 사용자 클라이언트로부터 요청을 받아 서버로 전달하며 서버로부터의 응답을 사용자에게 전달하는 간단한 프록시 서버를 만들어 보며 HTTP통신과 네트워크 프로그래밍에 대한 이해를 높이는 게 목적이다.

2. Implementation

프록시랩은 크게 세 단계로 이루어져 있다. 먼저, 파일에 대한 GET요청에 대해 static content를 제공하는 간단한 HTTP서버를 작성하였다. 그리고, 사용자의 클라이언트 (Gentoo VM상의 파이어폭스)로부터 요청을 받아 서버로 전달하고, 그 응답을 다시 사용자에게 전달하는 프록시 서버를 작성했다. 마지막으로, 이전과 같은 요청에 대한 응답시간을 줄이기 위해 캐싱 기능을 추가하여, 서버로 받은 응답을 캐시에 저장하도록 하였다.

2.1 HTTP 서버

main함수의 경우 CSAPP교재의 echo서버를 기반으로 작성한 뒤, 스펙에 맞게 변경시켰다. 먼저, main함수의 인자로 받는 포트번호가 숫자가 아닐 경우 에러를 출력하도록 하였다. csapp.c에 정의된 Open_listenfd함수를 이용해서 서버를 열고, 루프를 돌며 사용자로부터 요청을 받아서 처리하도록 구현하였다. 교재의 echo서버가 출력 용도로 사용했던 Getnameinfo 함수와 printf함수는 삭제하였다.

doit함수는 사용자로부터 입력을 받아 GET명령어가 아닐 경우 501 Not implemented에러를 출력한다. 대소문자에 관계없이 잘 작동하도록 하기 위해 명령어 문자열 비교 시 strcasecmp함수를 이용하였다. 이후 뼈대 코드에서 작성된 parse_uri함수를 이용해 입력으로 받은 uri를 파싱한 뒤, 파일이름을 비교하고, 파일이 디렉토리인지 확인하여 에러가 생기는 각각의 입력에 대해 404 Not found와 403 Forbidden 에러를 반환하였다. 정상적인 요청에 대해서는 serve_static함수를 이용해 응답을 보냈다.

get_filetype함수는 파일명의 확장자를 확인해 파일의 종류를 판단한다. 확장자가 html, gif, png, jpg 인 경우 각각의 파일 타입을 지정했으며, 이에 해당하지 않는 경우 text/plain을 사용했다.

serve_static함수는 교재의 serve_static함수를 참고하였다. 먼저 작성한 get_filetype함수를 이용해 파일의 정보를 얻은 뒤, HTTP/1.0 200 OK라는 응답 문장을 시작으로, 응답 헤더들을 구성하고 Mmap을 이용해 파일 데이터를 클라이언트로 전송하였다.

2.2 Proxy 서버

proxy서버의 main함수는 2.1에서 작성하였던 HTTP서버와 비슷하게 구성하였다. 초기 구현에서는 단일 프로세스 / 단일 스레드로 구현하여 클라이언트로부터 받은 요청을 순차적으로 처리하도록 하였다.

클라이언트로부터 요청을 받으면 doit함수를 실행한다. 먼저 Request Line을 입력받아 line1스트링에 저장하고, sscanf함수를 이용하여 method, uri, version으로 나눴다. GET메소드가 아닐 경우 501에러를 출력했으며, GET메소드가 들어온 경우 parse_uri_proxy함수를 이용해 입력으로 받은 uri를 host와 port그리고 filename로 파싱하였다. 파싱 과정에서 문제가 생긴 경우 400 Bad Request에러를 반환했다. 파싱이 정상적으로 진행되지 않은 경우, 남아있는 입력을 모두 제거해야 프록시 서버가 이후의 입력에 대해 정상적으로 작동하기 때문에 clienterror함수로 에러를 반환하기 전 print_requsthdrs함수를 호출하여 남아있는 입력을 모두 제거하였다. 파싱이 정상적으로 진행된 경우 역시 print_requesthdrs함수를 이용하여 남아있는 불필요한 입력을 읽어들인 뒤 무시했다. 이렇게 함으로써 서버에 요청을 보낼 때 브라우저 종류나 버전과 같은 클라이언트의 정보를 숨기는게 가능해진다.
open_clientfd함수를 이용해 앞에서 파싱한 host와 port정보를 이용해 서버에 접속하였고, line1 스트링에 저장해 두었던 request를 그대로 서버에 전달하였고, 다음 줄에 host와 port로 구성한 Host 헤더를 같이 보냈다.
반복문을 이용하여 서버로부터 받은 응답들을 조합하여 클라이언트에게 만들 응답을 구성하였다. 헤더의 종료를 뜻하는 빈 줄이 나올 때 까지 반복하였으며, 이 과정에서 “Transfer-Encoding:”이라는 문자열이나 “Content-Length:”라는 문자열이 등장하면, 해당 부분은 응답의 형식을 결정할 수 있다. Transfer-Encoding 헤더가 등장한 경우 isChunked변수를 1로 설정해, 뒤에 이어서 받을 response body의 입력을 chunked모드로 받을 수 있도록 하였으며, Content-Length헤더가 등장한 경우, sscanf 함수를 이용해 contentLength를 얻었다. 이 두 가지 문자열을 비교하는 과정에서 대-소문자로 인한 문제가 발생하지 않도록 하기 위해 문자열 비교에서 strncasecmp함수를 이용하였다. 빈 줄까지 읽어들여 response header를 구성하고 클라이언트에게 전달했다.

서버로부터 받은 Response body를 읽어들이는 과정ㅇ에서 isChunked플래그를 이용해 동작 방식을 결정하였다. Content-Length를 통해 하나의 묶음으로 데이터가 들어온 경우, Rio_writen 함수를 이용해 contentLength만큼을 읽어들였다. chunked데이터가 들어온 경우, 한 줄씩 읽으며 다음에 이어질 데이터의 크기(size)를 얻고, 라인 종료 문자인 \r\n까지 포함하여 총 size+2글자를 읽었다. 이렇게 한 줄 한 줄 읽어가며, 내용을 contentBuffer에 누적해가는데, 새로운 chunked블럭이 들어올 때마다 realloc함수를 이용하여 버퍼의 크기를 재조정했다. 0\r\n

을 마지막으로 chunked데이터 구성을 마쳤다. 이렇게 만든 contentBuffer를 클라이언트로 전달하며 doit 함수가 종료된다.

parse_uri_proxy함수는, GET method뒤에 이어져 들어오는 uri를 host와 port 그리고 filename으로 분리하는 작업을 한다. 먼저, 이 proxy함수는 http프로토콜에 대해서만 동작하기 때문에 포트는 80을 초기값으로 설정하고 진행하였다.
파싱 첫 부분에서 sscanf함수를 이용해 정규식 "http://%[^/]/%s"를 만족하는 입력에 대해 host와 port묶음인 t_hostport와 t_file로 분리하였다. t_hostport에 대해서는 strstr함수를 이용해 콜론 문자가 들어있는지 확인하여 포트정보가 포함되어 있을 경우, atoi함수를 이용하여 포트를 업데이트했다.

이렇게 작성한 프록시 서버가 제대로 동작하는지 확인하기 위해 주어진 두 개의 사이트로 테스트를 진행했다. 첫 번째 사이트(columbia.edu)의 경우 정상적으로 작동했지만, 두 번째 사이트(pactconf.org)는 절반정도 로딩이 잘 진행되다가 갑자기 프록시 서버가 죽는 현상이 발생하였다. 프록시 서버가 다운되며 open_clientfd함수에 문제가 있다는 에러를 출력했으며, 서버와의 연결이 가끔씩 실패한다는걸 알 수 있었다. 이러한 경우를 방지하고 안정적으로 동작하게 만들기 위해 while문을 이용해 open_clientfd함수를 성공할 때까지 반복적으로 호출하게 만들었으며, 무한루프를 방지하기 위해 retry횟수를 저장하는 변수를 만들어 최대 100번의 재접속 시도를 하도록 구성하였다.
변경된 프록시 서버는 대부분의 경우 잘 작동했지만 간헐적으로 broken pipe에러를 띄우며 종료되었는데, 이 중 대부분은 사이트 로딩 중에 firefox브라우저를 강제로 종료할 때 발생했다. 이러한 상황에서도 프록시 서버가 안정적으로 동작할 수 있도록 fork()를 도입하여 클라이언트에 대한 서비스를 child프로세스에서 진행할 수 있도록 하였으며, child프로세스 reapping을 위해 SIGCHLD시그널에 대한 핸들러를 구성하였다.

2.3 Caching

캐싱을 구현하는 과정에서, 프록시 서버가 fork를 이용해 별개의 프로세스에서 request를 처리하면 프록시 서버내에 cache가 공유되지 않음을 깨달았다. 따라서 fork와 execv로 구성된 proxy.c를 멀티스레드를 이용하는 방식으로 변경하였다. doit함수에서 파싱이 정상적으로 진행된 이후에 find_cache_block함수를 이용하여 데이터가 저장되었는지 확인하고, 캐시가 있다면 해당 정보를 바로 클라이언트로 전송하고 종료하도록 하였다. 캐시가 없어서 서버로부터 정보를 받은 경우, add_cache_block함수를 이용하여 캐시에 저장하였다.
캐시 구현에는 Queue를 이용하였다. find_cache_block은 단순히 선형탐색을 이용하였고, cache_replacement_policy함수는 dequeue를 이용하였다. 캐시 메모리의 크기를 관리하기 위해 전역변수 cache_size를 도입하였다.

add_cache_block함수는 doit함수에서 allocate된 content를 그대로 받아오기 때문에, 최대 캐시블럭 사이즈보다 큰 content 등이 들어왔을 때 content를 free하고 종료한다. 에러케이스가 아닌, 정상작동하는 경우에는 cache_size를 새 contentLength만큼 증가시키고, 이 값이 최대 캐시 용량보다 작아질 때까지 cache_replacement_policy 함수를 호출한다.
캐시 용량을 절약하기 위하여 enqueue과정에서는 uri, response에 대해 해당 문자열 크기+1만큼 malloc하여 문자열을 관리하였고, 이렇게 생성된 메모리는 dequeue과정에서 free되도록 구현하였다.
이렇게 구현한 cache가 대부분의 경우 정상 작동했지만, cache.c 내부에 queue 자료구조의 포인터와 cache_size전역변수에 접근하기 때문에 thread-unsafe하다. 따라서 전역변수와 자료구조에 접근하는 부분에 semaphore를 도입하여 thread-safe하게 수정하였다. 전체적인 구조는 교재의 readers-writers problem을 참고하였으며, find_cache_block함수를 reader, cache_replacement_policy 함수와 add_cache_block함수를 writer라고 놓고 semaphore를 적용하였다.

마지막으로, 2.1에서 구현한 http서버도 다중 요청에 효율적으로 응답할 수 있도록 프록시 서버와 마찬가지로 멀티스레드를 이용하도록 구조를 변경하였다.

3. Conclusion

첫 번재 HTTP서버 구현부터 CSAPP책을 참고하며 차근차근 진행해 나가면서 초기 버전을 완성하기 까지는 크게 막힌 부분 없이 잘 진행되었다. 또한 뼈대코드에 주석으로 에러 처리와 함수의 동작 과정이 자세하게 설명되어있어 방향을 쉽게 잡을 수 있었다. 하지만 open_listenfd함수가 서버 연결에 실패한다거나 갑작스럽게 사용자 클라이언트와 연결이 끊어지면서 생기는 broken pipe에러 등 예상하지 못한 문제가 생각보다 많이 발생했는데, 로컬에서 돌아가는 다른 프로그램과 다르게 네트워크 프로그램에서는 여러 가지를 함께 고려하며 안정적으로 동작하게 구현해야 한다는 점을 배웠다. 이러한 부분에서 문제의 원인을 파악하기 어려웠으며 많은 시간을 투자해 구현을 여러 번 변경하고 나서야 해결이 되었다.
마지막으로, cache를 thread-safe하게 변경하는 과정에서, enqueue, dequeue등 여러 함수들이 서로 호출하는 관계를 가지고 있어 데드락 상태에 들어가는 경우가 많았는데, 이를 해결하고 필요한 부분에만 semaphore를 도입하는게 까다로웠다.