시스템 프로그래밍

Proxy 2-4

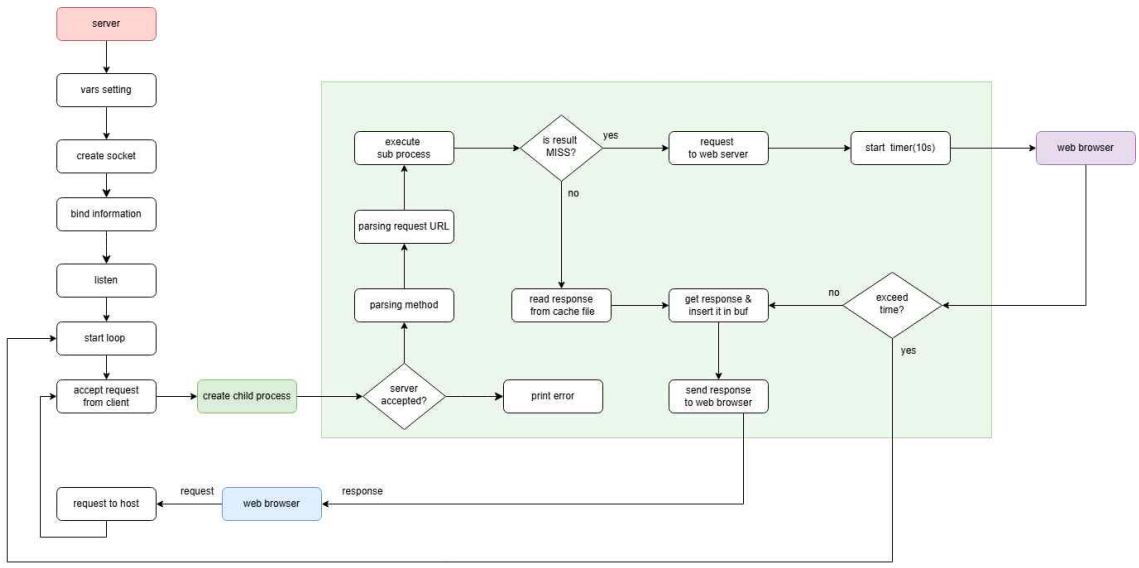
최상호 교수님
2020202047
이정한

Proxy 2-4

Introduction

이번 프로젝트는 저번 2-3과 유사하게, 웹 프록시 서버를 구현하고 이를 통해 캐시 기반의 효율적인 HTTP 응답 제공 방식을 실현하는 데 있다. 구현된 프록시 서버는 클라이언트(브라우저)로부터 요청을 수신하면, 먼저 해당 요청에 대한 캐시파일이 존재하는지 확인한다. 만약 요청이 캐시에 존재하는 경우 서버는 로컬 파일내에 생성된 데이터를 직접 읽어 응답함으로써 서버와의 불필요한 통신을 줄이고 응답 속도를 향상시킨다. 반대로 캐시에 존재하지 않는 경우에는 프록시 서버가 원격 서버에 직접 요청을 보내고 그 응답을 클라이언트에 전달 및 캐시에 저장하여 다음 동작에 활용할 수 있다.
이번 과제에서 특히 중점을 둔 부분은 HTTP 응답 데이터를 처리하는 과정에서의 동적 메모리 할당이다. 실제 네트워크 환경에서는 응답의 크기가 가변적이기 때문에, 고정된 크기의 버퍼로는 완전한 데이터를 수신하지 못하거나 메모리 낭비가 발생할 수 있다. 따라서 본 구현에서는 수신한 데이터의 크기에 따라 버퍼를 동적으로 확장하며 안전하게 데이터를 수신하고 이를 기반으로 정확하게 캐시 파일을 작성하는데 초점을 맞췄다.

Flow Chart



(2-3과 동일)

Pseudo code

[main]

```
Main() {
    // STEP 1: 초기 세팅
    call vars_setting()
        - getHomeDir(home) 호출하여 home 디렉토리 경로 설정
        - snprintf()로 cachePath, logPath 지정
        - ensureDirExist(cachePath, 0777), ensureDirExist(logPath, 0777)
        - 로그파일 존재 여부 확인 → 없으면 createFile()
        - make_dir_path(logPath, logfileName) 후 init_log()로 열기
        - hit_count, miss_count = 0, start_time 설정

    // STEP 2: 소켓 설정
    socket_fd = socket(PF_INET, SOCK_STREAM, 0)
    server_addr 설정 (AF_INET, INADDR_ANY, PORTNO)
    bind(socket_fd, server_addr)
    listen(socket_fd, 5)
    signal(SIGCHLD, handler)  // 좀비 프로세스 수거 핸들러 등록

    // STEP 3: 클라이언트 요청 처리 루프
    while (true) {
        accept client_fd from socket_fd
        if client_fd < 0:
            → print error
            → continue

        read(client_fd, buf, BUFFSIZE)
        if read 실패 or client 종료:
            → print error
            → close(client_fd)
            → continue

        copy buf to tmp
        url = get_parsing_url(tmp)
        if url == NULL:
            → close(client_fd)
            → continue
```

```
        if is_filtered_url(url):
            → free(url)
            → close(client_fd)
            → continue

        pid = fork()
        if pid == -1:
            → close(client_fd)
            → continue

        if pid == 0:  // 자식 프로세스
            sub_start_time = time()
            result = sub_process(url, log_fp, cachePath, client_fd)
            if result == MAIN_REQUEST:
                → process_count++
            → exit(0)

        // 부모 프로세스
        close(client_fd)
        free(url)
    }

    // STEP 4: 자원 정리
    close(socket_fd)
    return 0
}
```

[sub_process]

```
sub_process(input_url, pid, log_fp, cachePath, sub_start_time,
                            hit_count, miss_count, buf, buf_size) {
    if input_url == NULL:
        return PROCESS_UNKNOWN

    // STEP 1: 해시 및 경로 설정
    sha1_hash(input_url, hashed_url)
    subdir = hashed_url 앞 3자리
    fileName = 나머지 37자리
    subCachePath = make_dir_path(cachePath, subdir)
    cache_full_path = subCachePath/fileName

    result = is_file_hit(subCachePath, fileName)
    signal(SIGALRM, timeout_handler)  // 10초 타임아웃 처리 등록

    // STEP 2: Check if request is for static sub-resource
    if trimmed_url ends with static file extension:
        is_sub_request ← true
    else:
        is_sub_request ← false

    // STEP 3: Handle cache MISS
    if result is MISS:
        host_ip ← resolve host to IP
        if resolution fails:
            return PROCESS_UNKNOWN

        connection ← connect to host_ip
        if connection fails:
            return PROCESS_UNKNOWN

        send HTTP GET request to connection
        start timeout alarm

        response ← receive full HTTP response from connection
        stop timeout alarm

        create cache directory and file if needed
        write response to cache file
        send response to client

        log_contents ← generate MISS log

    // STEP 4: Handle cache HIT
    else if result is HIT:
        open cache file
        read response from cache
        send response to client

        log_contents ← generate HIT log

    // STEP 5: Unknown cache result
    else:
        return PROCESS_UNKNOWN

    write_log_contents(log_fp, log_contents)
    free 동적 메모리들 (subCachePath, log_contents, full_path 등)

    return result
}
```

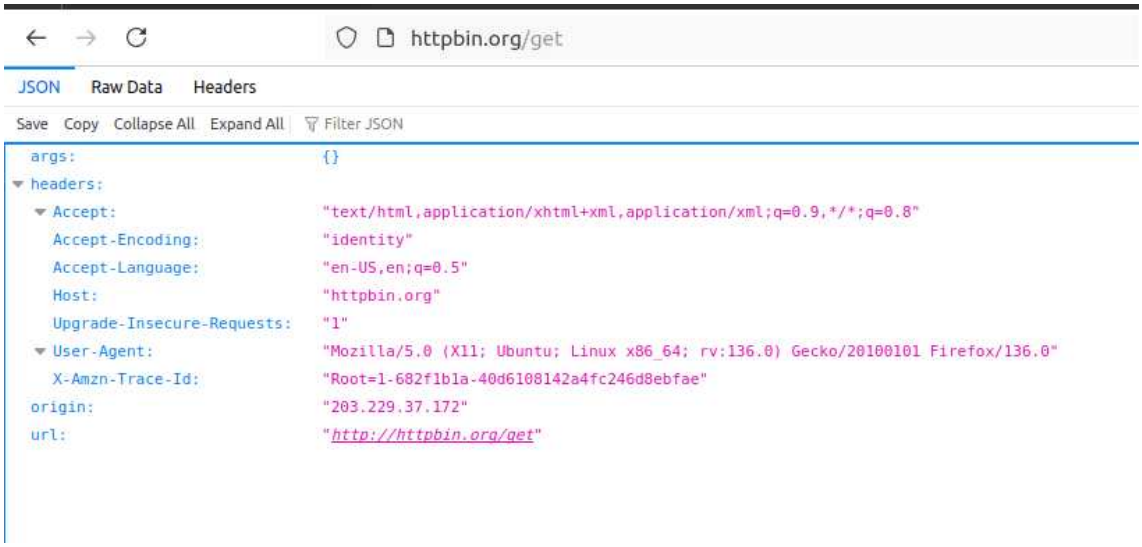
[receive_response]

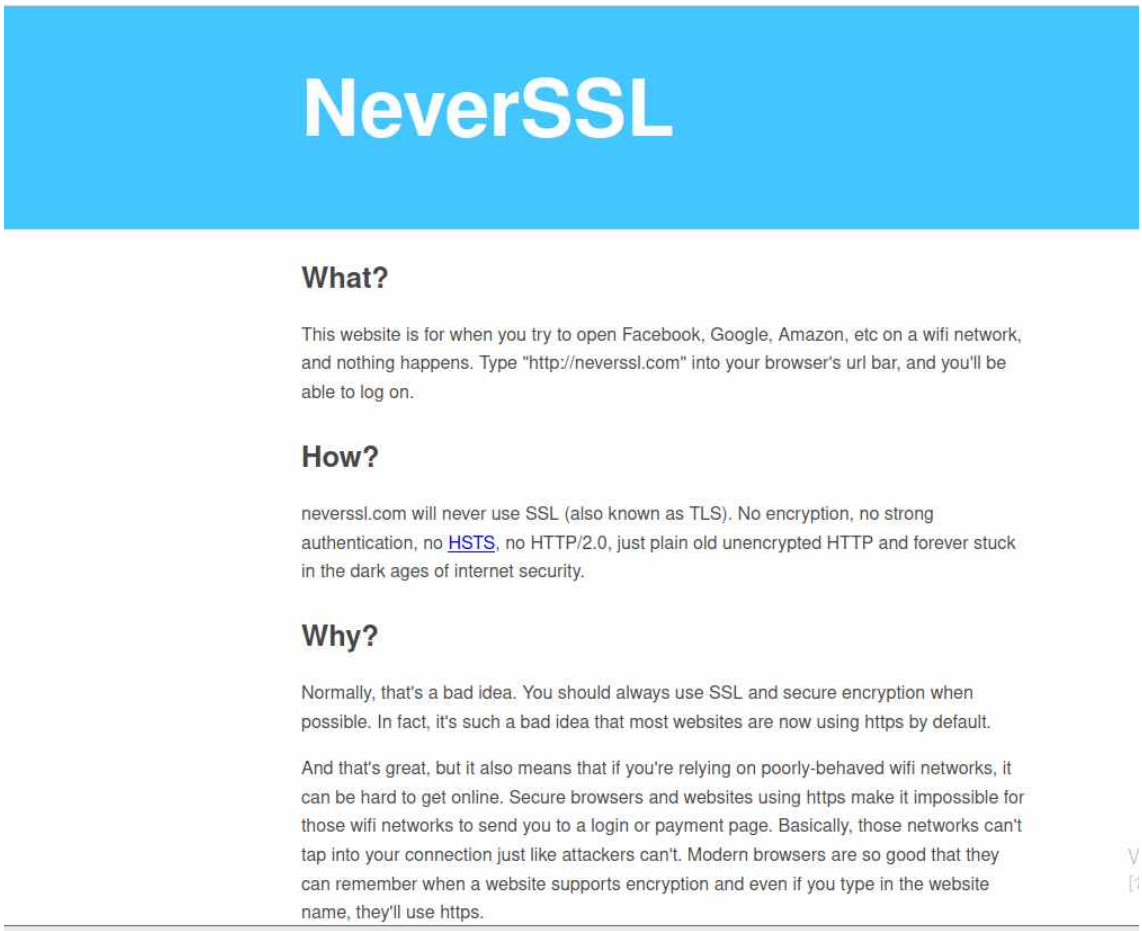
```
receive_http_response(sockfd, out_size) {
    // STEP 1: 초기 설정
    buffer = NULL
    buffer_size = 0
    total_received = 0
    content_length = 0
    header_parsed = false
    header_end = NULL

    // STEP 2: 응답 수신 루프
    while (true) {
        buffer = realloc(buffer, buffer_size + READ_BLOCK_SIZE + 1)
        if realloc 실패:
            → print "realloc failed"
            → return NULL

        n = read(sockfd, buffer + total_received, READ_BLOCK_SIZE)
        if n < 0:
            → print "read failed"
            → free(buffer)
            → return NULL

        else if n == 0:
            if header_parsed && total_received >= content_length:
                → break  // 수신 완료
            else:
                → wait 5ms (usleep)
                → continue  // 수신 대기
```

```
        // STEP 3: 응답 헤더 파싱
        if header_parsed == false:
            header_end = strstr(buffer, "\r\n\r\n")
            if header_end == NULL:
                → continue  // 아직 헤더 끝 못 찾음

            header_parsed = true

            cl_ptr = strstr(buffer, "Content-Length:")
            if cl_ptr == NULL:
                → print "Content Length not found"
                → return NULL

            cl_ptr += strlen("Content-Length:")
            공백 문자 건너뛰기
            content_length = atoi(cl_ptr)

            header_size = (header_end - buffer) + 4
            content_length += header_size

        // STEP 4: 전체 응답 수신 완료 여부 확인
        if header_parsed && total_received >= content_length:
            → break
    }

    // STEP 5: 결과 반환
    if out_size != NULL:
        *out_size = total_received

    return buffer
}
```

receive response 의 무한루프 로직은 다음과 같다.
(1) 버퍼 공간 확장(READ_BLOCK_SIZE)만큼 더 할당)
(2) 소켓으로부터 데이터 추가 수신
(3) 서버 측에서 연결을 종료
(3.1) 응답받은 메세지 양 >= 헤더의 content length : 루프 탈출
(3.2) 아직 응답받지 못한 경우를 위해 재기
(4) 정상적으로 응답받은 경우, 해당 회차의 루프에서 받아온 만큼 사이즈 동기화
(5) 헤더 파싱 -> content length 파싱
(5.1) 헤더 파싱 후 루프 탈출 조건 성립확인
(6) 응답 크기 반환

결과화면






```
[Hit] Server PID : 2489 | /home/kw2020202047/cache/7fa/28472937cfbc36ea28b2c1f236003ff0e29d1-[2025/05/22,
05:38:21]
[Hit]textfiles.com/
[Miss] Server PID : 2600 | httpbin.org/get-[2025/05/22, 05:39:54]
[Hit] Server PID : 2645 | /home/kw2020202047/cache/e3d/87d83b3678bed4384d63f4a7be04a7a23277a-[2025/05/22,
05:39:58]
[Hit]httpbin.org/get
[Miss] Server PID : 2739 | www.catb.org/jargon/-[2025/05/22, 05:40:49]
```

고찰

이번 프록시 서버 구현 과정에서 가장 핵심적인 과제는 메모리 관리와 캐시 저장의 안정성 확보였다. 2-3에서는 고정된 크기의 버퍼를 사용하여 HTTP응답을 수신하도록 구현했으나 이번 과제를 테스트하는 도중 결과가 잘못되었다는 것을 확인하였다. 캐시 파일 내부까지 정확히 확인해 본 결과 웹 페이지에 대한 응답을 처리하던 중 응답 전체를 가져오지 못한 채 일부 데이터만 저장된 상태에서 캐시에 기록되고 클라이언트도 잘못된 데이터가 전달되는 문제가 발생했다. 브라우저에 결과가 나온 것을 제대로 된 환경에서 테스트 하지 않아 발생한 오류였다.

이 문제의 핵심은 메모리를 고정된 크기로 한정했기 때문에 데이터의 전부를 수신하지 못하는 것이다. 해결을 위해서는 수신한 데이터의 양에 따라 버퍼를 동적으로 재할당해 나가는 방식이 필요했다. realloc을 이용하여 일정 블록 단위로 버퍼를 확장하고, 매번 현재까지 수신된 양을 기준으로 처리함으로써 전체 응답을 안전하게 수신할 수 있도록 개선했다. 이 과정에서 메모리를 조금이라도 잘못 관리한다면, 예를 들어 포인터를 잃어버리거나 충분히 null-terminate 하지 않으면 다시 응답을 제대로 파싱하지 못하는 문제가 이어졌기 때문에 신중하게 디버깅하며 구현을 진행했다.

이번 과제를 구현하면서 알게 된점으로, 브라우저에 요청한 url에는 부가적으로 필요한 요청들이 존재한다는 것이었다. 예를 들어 favicon 이미지 파일 등 클라이언트가 명시적으로 요청하지 않아도 브라우저가 자동으로 요청을 보내는 리소스들이 이에 해당한다. 처음에는 로그에 나타나는 수많은 요청들이 잘못된 것들인 줄 알았지만 웹 브라우저에 부족한 결과화면과 로그에 작성된 .ico, .png, .js, .css 등 다양한 정적 리소스가 브라우저에 의해 자동으로 요청된다는 사실을 깨달았다. 이 요청들까지 모두 로그에 기록되면 복잡해지기 대문에 strstr()을 활용해 부가적인 요청들은 필터링하여 로그 작성에 제외하였다.

Reference