



[Memo] proxy server 과제 번역본

☰ 태그	Crafton Jungle
☑ _visible	☑
📅 게시일	@2025/07/03
🕒 수정일	@2025년 7월 3일 오후 3:05

1. 소개

웹 프록시는 웹 브라우저와 최종 서버 사이에서 중간자 역할을 하는 프로그램입니다. 브라우저가 웹 페이지를 요청할 때, 최종 서버에 직접 연결하지 않고 프록시를 거쳐 요청을 보냅니다. 프록시는 이 요청을 서버에 전달하고, 서버로부터 응답을 받은 뒤 이를 다시 브라우저에 전달합니다.

프록시는 다양한 목적으로 활용됩니다. 예를 들어, 방화벽 뒤에 있는 브라우저가 외부 서버에 접근할 때 프록시를 통해서만 가능하도록 하거나, 프록시가 요청에서 식별 정보를 제거하여 브라우저를 익명화(익명화 도구로 사용)할 수 있습니다. 또한 프록시는 웹 객체를 저장(캐싱)하여, 동일한 요청이 들어오면 서버와 다시 통신하지 않고 캐시된 데이터를 제공할 수도 있습니다.

이 실습에서는 웹 객체를 캐시할 수 있는 간단한 HTTP 프록시를 작성하게 됩니다. 실습은 세 부분으로 구성됩니다:

1. 클라이언트 요청을 수신하고 파싱한 후, 서버에 전달하고 응답을 다시 클라이언트로 전달하는 **기본 프록시 구현**
2. 동시에 여러 클라이언트 요청을 처리하는 **동시성 구현**
3. **최근에 접근한 웹 콘텐츠를 메모리에 캐시하여 재사용하는 기능 구현**

2. 운영 및 제출 안내

- 이 프로젝트는 **개인 과제**입니다.
- 제출 마감일 및 최종 제출 가능일이 명시되어 있습니다.

(과제 할당: 11월 19일 목요일 / 마감: 12월 8일 화요일 / 최종 제출: 12월 11일 금요일)

3. 배포 파일 안내

(이 부분은 교수진이나 조교가 작성해야 할 사이트별 배포 안내입니다.)

리눅스 머신의 보호된 디렉토리에 `proxylab-handout.tar` 파일을 복사한 뒤, 다음 명령어로 압축을 풉니다:

```
linux> tar xvf proxylab-handout.tar
```

이 명령어를 실행하면 `proxylab-handout` 이라는 디렉토리가 생성됩니다. 이 디렉토리 안의 `README` 파일에는 각 파일에 대한 설명이 포함되어 있습니다.

4. Part I: 순차적 웹 프록시 구현

첫 번째 단계는 **HTTP/1.0 GET 요청**을 처리하는 **기본적인 순차적 프록시**를 구현하는 것입니다.

다른 요청 방식(예: POST)은 **선택 사항**이며 구현하지 않아도 됩니다.

프록시 프로그램을 실행하면, ****명령행 인자(argument)****로 지정된 포트 번호를 사용하여 **들어오는 연결을 수신**해야 합니다. 연결이 설정되면, 프록시는 **클라이언트로부터 전체 요청을 읽고 이를 파싱**해야 합니다.

요청이 **유효한 HTTP 요청인지 확인**한 다음, 해당 요청에 맞는 **웹 서버와 연결을 설정**하고, ****클라이언트가 요청한 객체(object)****를 요청해야 합니다. 서버의 응답을 받은 뒤, **그 응답을 다시 클라이언트로 전달**합니다.

4.1 HTTP/1.0 GET 요청

사용자가 브라우저 주소창에 다음과 같이 입력할 수 있습니다:

```
http://www.cmu.edu/hub/index.html
```

이 경우 브라우저는 다음과 같은 HTTP 요청을 프록시에 보냅니다:

```
GET http://www.cmu.edu/hub/index.html HTTP/1.1
```

프록시는 이 요청을 파싱하여 **최소한 다음 정보를 추출**해야 합니다:

- 호스트 이름: `www.cmu.edu`
- 경로(path): `/hub/index.html`

프록시는 위 정보를 이용하여 해당 호스트에 연결을 생성하고, 다음과 같은 형식의 요청을 서버에 보냅니다:

```
GET /hub/index.html HTTP/1.0
```

◆ 참고: HTTP 요청의 각 줄은 `\r\n` (캐리지 리턴 + 줄바꿈)으로 끝나야 하며, 전체 요청은 빈 줄 `"\r\n"`로 종료됩니다.

현대 브라우저는 **HTTP/1.1** 형식으로 요청을 보내지만, **프록시는 이를 수신하여 HTTP/1.0 형식으로 변환하여 서버에 전달**해야 합니다.

📌 주의사항: HTTP/1.0 GET 요청만 보더라도 요청 형식이 복잡할 수 있습니다. 가능한 한 **robust(탄탄하고 오류에 강한)한 요청 파서(parser)**를 작성해야 하며, RFC 1945 문서에 따라 구현하는 것이 이상적입니다. 다만, 멀티라인 헤더는 처리하지 않아도 됩니다.

4.2 요청 헤더

이 실습에서 중요한 요청 헤더는 다음과 같습니다:

- **Host 헤더 (항상 포함)**

예:

```
Host: www.cmu.edu
```

브라우저가 이 헤더를 이미 포함하는 경우, **동일한 값을 그대로 사용**하세요.

- **User-Agent 헤더**

항상 다음 값을 사용할 수 있습니다:

```
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3) Gecko/20120305 Firefox/10.0.3
```

(실제 전송 시에는 **한 줄로 연결**해서 보내야 합니다.)

- **Connection 헤더 (항상 포함)**

```
Connection: close
```

- **Proxy-Connection 헤더 (항상 포함)**

```
Proxy-Connection: close
```

◆ 설명: 위 두 헤더는 연결을 요청/응답 이후에 종료할 것임을 알립니다. 각 요청마다 새로운 연결을 여는 것이 권장되며, close를 명시함으로써 서버에 이 사실을 알립니다.

💡 proxy.c 코드 안에 위 User-Agent 문자열이 ****상수(constant)****로 정의되어 있으므로 편리하게 사용할 수 있습니다.

추가적으로 브라우저가 보낸 **기타 헤더**가 있다면, **수정하지 말고 그대로 서버에 전달**해야 합니다.

🔧 4.3 포트 번호

이 실습에서 중요한 포트는 두 종류입니다:

1. HTTP 요청 포트

- 예: `http://www.cmu.edu:8080/...`
- 포트가 명시된 경우, 해당 포트(예: 8080)로 연결을 시도해야 하며, 포트가 생략된 경우에는 **기본 포트 80번**을 사용합니다.

2. 프록시 리스닝 포트

- 프록시는 클라이언트 요청을 수신하기 위해 **명령행 인자로 지정된 포트**에서 대기해야 합니다.
- 예시 실행:

```
linux> ./proxy 15213
```

🧠 포트 번호는 1024보다 크고 65535보다 작은 비특권 포트여야 하며, 중복 사용을 방지해야 합니다.

제공된 스크립트 `port-for-user.pl` 을 통해 개인용 포트를 생성할 수 있습니다:

```
linux> ./port-for-user.pl droh
droh: 45806
```

이 때 생성되는 포트 번호는 **짝수**이며, 필요 시 인접 포트(짝수+1)도 함께 사용할 수 있습니다.

5. Part II: 동시적(Concurrent) 요청 처리

기본적인 순차 프록시가 동작하도록 구현한 후에는, 이를 확장하여 **동시에 여러 요청을 처리할 수 있도록** 수정해야 합니다.


가장 간단한 방식은 **새로운 연결 요청이 있을 때마다 새로운 스레드를 생성하여 처리**하는 것입니다.

다른 설계 방법도 가능하지만, 예를 들어 교재의 **12.5.5절에 소개된 prethreaded 서버 모델**도 사용할 수 있습니다.

스레드 관련 주의사항:

- *스레드는 분리 모드(detached mode)**로 실행되어야 하며, 이를 통해 ****메모리 누수(memory leak)****를 방지할 수 있습니다.
- `open_clientfd` 와 `open_listenfd` 함수는 CS:APP3e 교재에 설명되어 있으며, 최신 표준 기반의 `getaddrinfo` 함수를 사용하므로 스레드 안전(thread-safe) 합니다.

이 파트에서는 스레드 프로그래밍과 동시성 처리의 핵심 개념을 실제 구현을 통해 학습하게 됩니다.

 **힌트:** Pthreads API의 `pthread_create`, `pthread_detach` 등을 활용하게 될 것입니다.

6. Part III: 웹 객체 캐싱


마지막 단계에서는, 프록시에 **최근에 사용된 웹 객체를 메모리에 저장하는 캐시 기능**을 추가하게 됩니다.

HTTP는 원래 서버가 객체에 대해 어떻게 캐시해야 할지 지시하고, 클라이언트가 캐시 사용 방식을 지정할 수 있도록 정의된 **복잡한 캐싱 모델**을 갖고 있습니다.

하지만 이 실습에서는 **간단한 방식의 캐싱**만 구현합니다.

프록시는 서버로부터 웹 객체를 받을 때, **클라이언트로 전송하면서 동시에 메모리에 저장**합니다.

이후 **다른 클라이언트가 동일한 객체를 요청하면**, 프록시는 다시 서버에 연결하지 않고 **캐시된 객체를 재전송**합니다.

 **단, 모든 객체를 캐싱하면 메모리가 무한히 필요하므로, 적절한 제한이 필요합니다.**

6.1 최대 캐시 크기

프록시 캐시 전체의 최대 크기는 다음과 같이 제한됩니다:

```
MAX_CACHE_SIZE = 1 MiB (메가바이트)
```

이 크기를 계산할 때는 웹 객체 자체의 바이트 수만 포함해야 하며,

메타데이터(예: 구조체, 포인터 등)는 포함하지 않습니다.

6.2 개별 객체의 최대 크기

캐시에 저장할 수 있는 개별 웹 객체의 최대 크기는 다음과 같습니다:

```
MAX_OBJECT_SIZE = 100 KiB (킬로바이트)
```

이 두 가지 제한값은 proxy.c 파일에서 ****매크로(macros)****로 정의되어 있습니다.

💡 캐시 구현 방법

가장 간단하고 안정적인 구현 방식은 다음과 같습니다:

1. 각 연결에 대해 ****버퍼(buffer)****를 할당합니다.
2. 서버로부터 받은 데이터를 버퍼에 누적합니다.
3. 만약 버퍼 크기가 **최대 객체 크기를 초과**하면, 해당 데이터를 **캐시하지 않고 폐기**합니다.
4. 전체 응답을 **최대 크기 이내로 모두 수신**한 경우에는, 이를 **캐시에 저장**합니다.

이 방법을 사용할 경우, 웹 객체 저장에 사용되는 총 메모리의 상한은 다음과 같습니다:

```
MAX_CACHE_SIZE + (T * MAX_OBJECT_SIZE)
```

여기서 **T**는 동시에 열려 있는 최대 연결 수입니다.

🔄 캐시 제거 정책 (Eviction Policy)

캐시는 가장 최근에 사용되지 않은 객체부터 제거하는 **LRU(Least Recently Used)** 정책을 사용해야 합니다.

완벽한 LRU가 아니어도 괜찮지만, 비슷한 수준으로 동작해야 합니다.

💡 참고: 객체를 읽거나 쓰는 모든 작업은 “사용”으로 간주됩니다.

🔒 동기화 (Synchronization)

- 캐시에 대한 접근은 **스레드 안전(Thread-safe)** 해야 합니다.
- 특히, 여러 스레드가 동시에 캐시를 **읽을 수 있어야 하며**,
쓰기 작업은 **단일 스레드만 허용**되어야 합니다.

❌ 캐시 전체에 하나의 큰 락(lock)을 걸어 동기화하는 방식은 허용되지 않습니다.




이를 위해 다음과 같은 기법을 고려할 수 있습니다:

- 캐시 분할(partitioning)
- **pthread**의 **reader-writer 락** 사용
- 세마포어를 이용해 **사용자 정의 reader-writer 락** 구현

엄격한 LRU를 구현할 필요는 없기 때문에, 여러 리더를 허용하는 구조로 유연하게 설계할 수 있습니다.

7. 평가 (Evaluation)

이 과제는 총 70점 만점으로 평가되며, 평가 항목은 다음과 같습니다:

-  **기본 기능 구현 (Basic Correctness): 40점**
 - 기본 프록시 기능이 제대로 작동하는지 확인 (자동 채점)
-  **동시성 (Concurrency): 15점**
 - 여러 요청을 동시에 처리할 수 있는지 평가 (자동 채점)
-  **캐시 기능 (Cache): 15점**
 - 캐시가 올바르게 동작하는지 평가 (자동 채점)

7.1 자동 채점 (Autograding)

배포된 핸드아웃 디렉토리에는 **자동 채점 도구 `driver.sh` **가 포함되어 있습니다.

이 스크립트는 위의 세 항목(BasicCorrectness, Concurrency, Cache)에 대해 채점을 수행합니다.

사용 방법:

```
linux> ./driver.sh
```

|  자동 채점 도구는 Linux 환경에서 실행해야 합니다.

7.2 견고성 (Robustness)

당신의 프로그램은 다음의 조건을 만족해야 합니다:

- **에러 및 비정상 입력에 강하고,**
- **장시간 실행되는 프로세스로 동작하며,**
- 예외 상황에서도 절대 프로그램이 **갑자기 종료되지 않아야** 합니다.


이 외에도 다음과 같은 사항이 견고성에 포함됩니다:

- **세그멘테이션 폴트 방지**
- **메모리 누수 없음**
- **파일 디스크립터 누수 없음**

8. 테스트 및 디버깅 (Testing and Debugging)

자동 채점 도구 외에는 별도의 **샘플 입력**이나 **테스트 프로그램**이 제공되지 않습니다.

따라서, 직접 테스트 케이스를 설계하거나 **테스트 스크립트(테스트 하니스)**를 작성해야 합니다.

|  **실전 환경에서도 운영 조건이 명확하지 않거나 레퍼런스 구현이 없는 경우가 많기 때문에,**
이런 테스트 역량은 매우 중요한 실무 능력입니다.

다행히도, 테스트와 디버깅을 도와줄 다양한 도구들이 존재합니다.

아래에 도구별 설명을 제공합니다:

8.1 Tiny 웹 서버

핸드아웃 디렉토리에는 **CS:APP Tiny 웹 서버 소스 코드**가 포함되어 있습니다.

- 고급 기능은 부족하지만, 프록시 개발을 위한 **수정이 쉬운 구조**입니다.
- 프록시의 출발점으로도 사용 가능하며,
- 자동 채점 도구(driver.sh)도 이 서버를 통해 페이지를 요청합니다.

8.2 telnet

교재 11.5.3절에 나온 것처럼, **telnet을 이용해 프록시에 직접 HTTP 요청을 전송**할 수 있습니다.

프록시의 기본 동작을 점검하기에 유용한 수단입니다.

8.3 curl

curl은 서버나 프록시에 HTTP 요청을 전송할 수 있는 매우 유용한 디버깅 도구입니다.

예시:

- 프록시가 **15214** 포트에서 실행 중이고, Tiny 웹 서버가 **15213** 포트에서 실행 중인 경우:

```
linux> curl -v --proxy http://localhost:15214 http://localhost:15213/home.html
```

출력 예시:

```
> GET http://localhost:15213/home.html HTTP/1.1
> Host: localhost:15213
> Proxy-Connection: Keep-Alive
< HTTP/1.0 200 OK
< Server: Tiny Web Server
< Content-length: 120
...
<html>...</html>
```

8.4 netcat (nc)

netcat 또는 **nc**는 매우 강력한 네트워크 유틸리티입니다.

- telnet처럼 **서버에 요청을 보내는 클라이언트로 사용할 수 있으며,**
- 반대로 **서버 역할을 수행**할 수도 있습니다.

예시:

- 프록시가 **catshark** 머신에서 **12345** 포트로 실행 중인 경우:

```
sh> nc catshark.ics.cs.cmu.edu 12345
GET http://www.cmu.edu/hub/index.html HTTP/1.0
```

netcat을 서버로 실행하기:

```
sh> nc -l 12345
```

이 상태에서 프록시를 통해 요청을 보내면, **프록시가 netcat 서버에 보낸 정확한 요청 내용을 확인할 수 있습니다.**

8.5 웹 브라우저

프록시의 동작을 최종적으로 ****실제 웹 브라우저(예: Mozilla Firefox)****로 테스트하는 것이 좋습니다.

- Firefox 설정에서 프록시를 등록할 수 있습니다:

Preferences > Advanced > Network > Settings

- 제한된 기능이지만, 대부분의 웹사이트를 **프록시를 통해 탐색 가능**합니다.

! 단, 브라우저 자체도 캐시를 사용하므로, 프록시의 캐시를 테스트할 때는 브라우저 캐시를 반드시 비활성화하세요.

9. 제출 방법 (Handin Instructions)

Makefile에는 **최종 제출용 파일을 생성**하는 기능이 포함되어 있습니다.

작업 디렉토리에서 다음 명령어를 실행하세요:

```
linux> make handin
```

이 명령어를 실행하면 상위 디렉토리에 다음과 같은 파일이 생성됩니다:

```
../proxylab-handin.tar
```

이 **.tar** 파일을 제출하시면 됩니다.

◆ SITE-SPECIFIC: 이 부분은 교수나 조교가, 각 학생이 proxylab-handin.tar 파일을 어떻게 제출해야 하는지 안내하는 내용을 직접 작성해야 합니다.

참고 자료

- 교재의 **10~12장**에는 다음과 같은 유용한 정보가 포함되어 있습니다:
 - 시스템 수준의 I/O
 - 네트워크 프로그래밍
 - HTTP 프로토콜
 - 병렬/동시 프로그래밍
- RFC 1945** 문서는 HTTP/1.0 프로토콜에 대한 완전한 명세서입니다:
<http://www.ietf.org/rfc/rfc1945.txt>

10. 힌트 (Hints)

프록시를 구현할 때 도움이 되는 팁은 다음과 같습니다:

- 📖 **교재 10.11절**에 설명된 것처럼, 소켓 입출력 시 ****표준 I/O 함수(fprintf, scanf 등)****를 사용하는 것은 위험합니다.
대신, 핸드아웃 디렉토리의 `csapp.c`에 포함된 **Robust I/O(RIO) 패키지**를 사용하세요.
- ⚠️ `csapp.c`에 있는 ****에러 처리 함수(error-handling functions)****는 프록시에 적합하지 않습니다.
프록시는 **서버처럼 장시간 실행되는 프로그램**이므로, **에러 발생 시 즉시 종료하는 방식은 바람직하지 않습니다.**
→ 에러 처리 함수를 **수정하거나 직접 구현**해야 합니다.
- 🛠️ 핸드아웃 디렉토리의 파일은 **자유롭게 수정**할 수 있습니다.
예: 캐시 관련 기능을 `cache.c`, `cache.h`로 모듈화
→ 새 파일을 추가한 경우, `Makefile`도 함께 수정해야 합니다.
- 🚫 교재 964페이지의 사이드노트에 따라, **SIGPIPE 시그널을 무시**해야 하며,
`write()` 함수에서 `EPIPE` 오류가 발생해도 **정상적으로 처리**해야 합니다.
- ❌ 소켓이 미리 닫힌 상태에서 `read()` 호출 시, `1`을 반환하고 `errno`가 `ECONNRESET`일 수 있습니다.
이 경우에도 프록시가 종료되지 않도록 처리해야 합니다.
- 🖼️ 웹에는 **이진(binary) 데이터**도 많습니다 (예: 이미지, 영상).
네트워크 I/O 함수 선택 시 **텍스트뿐 아니라 바이너리 데이터도** 올바르게 처리할 수 있어야 합니다.
- 🌐 브라우저가 보내는 요청이 **HTTP/1.1**이더라도, **항상 HTTP/1.0**으로 변환하여 서버에 전달해야 합니다.

🎉 마지막 인사

Good luck! (행운을 빕니다!)