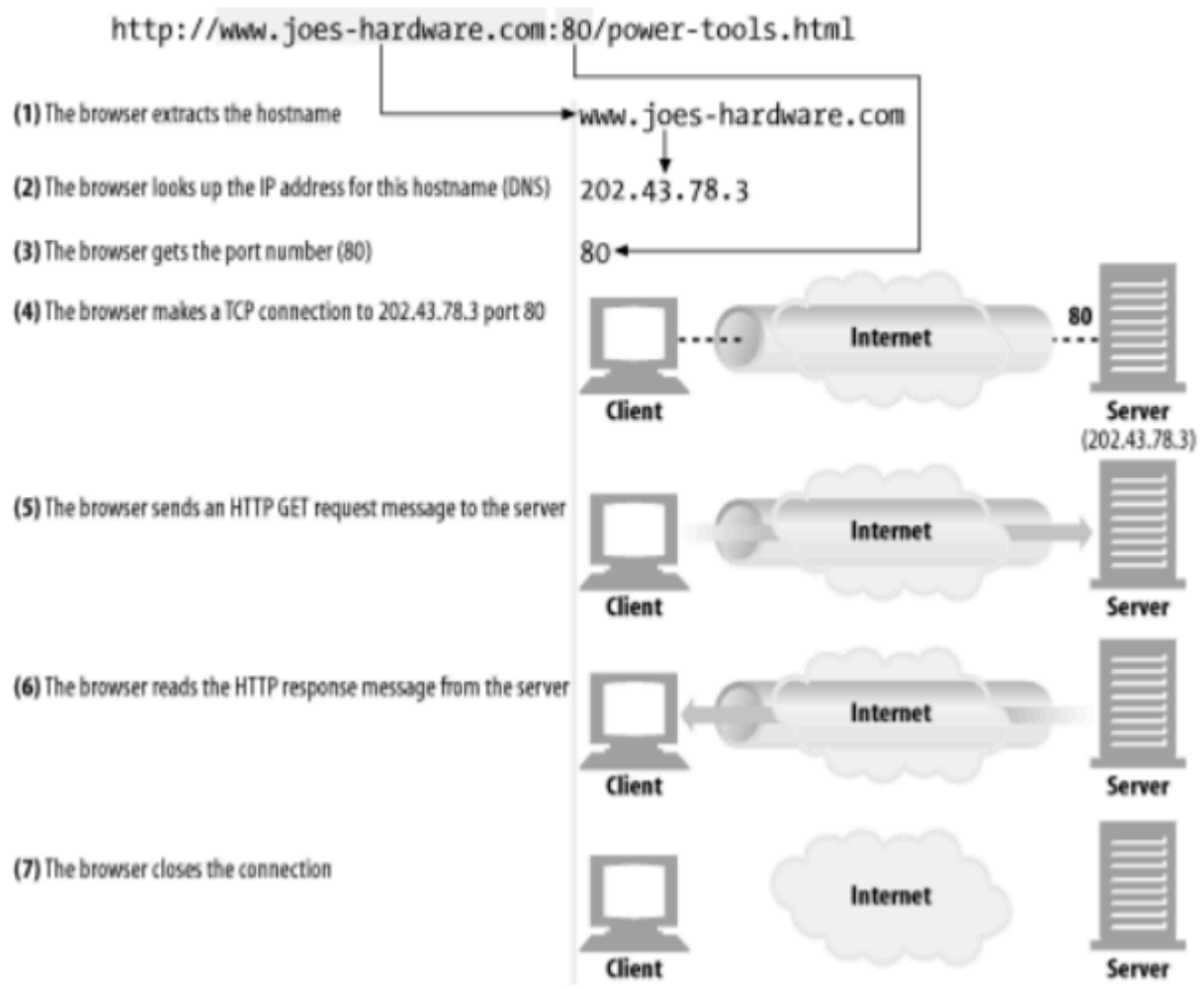
4장. 커넥션 관리

4.1 TCP 커넥션

죠의 컴퓨터 가게에서 전동공구의 최신 가격 목록을 가져온다고 해보자.

```
http://www.joes-hardware.com:80/power-tools.html
```



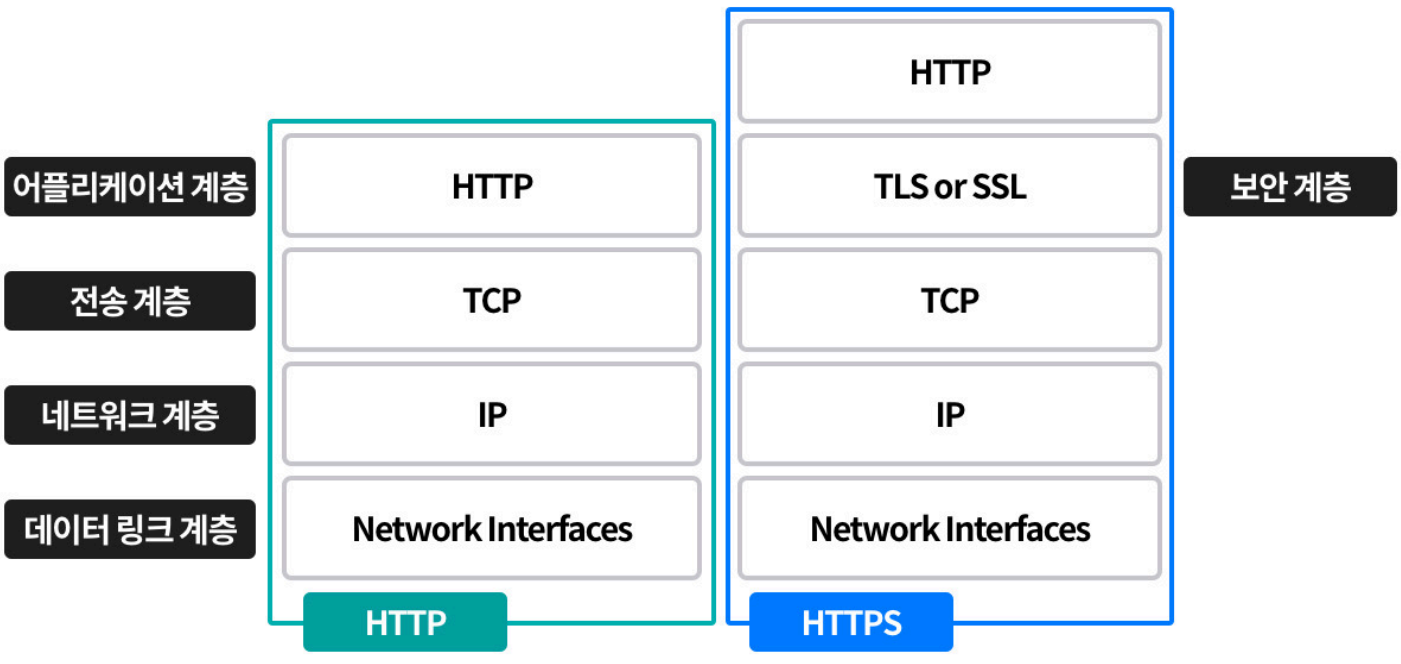
- 브라우저가 호트스명을 추출합니다.
- 브라우저가 이 호스트 명에 대한 IP 주소를 찾습니다.
- 브라우저가 포트 번호를 얻습니다.
- 브라우저가 IP주소의 포트로 TCP 커넥션을 생성합니다.
- 브라우저가 서버로 HTTP 메소드를 보냅니다.
- 브라우저가 서버에서 온 HTTP 응답 메시지를 읽습니다.
- 브라우저가 커넥션을 끊습니다.

4.1.1 신뢰할 수 있는 데이터 전송 통로인 TCP

HTTP 커넥션은 몇몇 사용 규칙을 제외하고는 TCP 커넥션에 불과하다!

TCP는 HTTP 에게 신뢰할 만한 통신 방식을 제공한다.

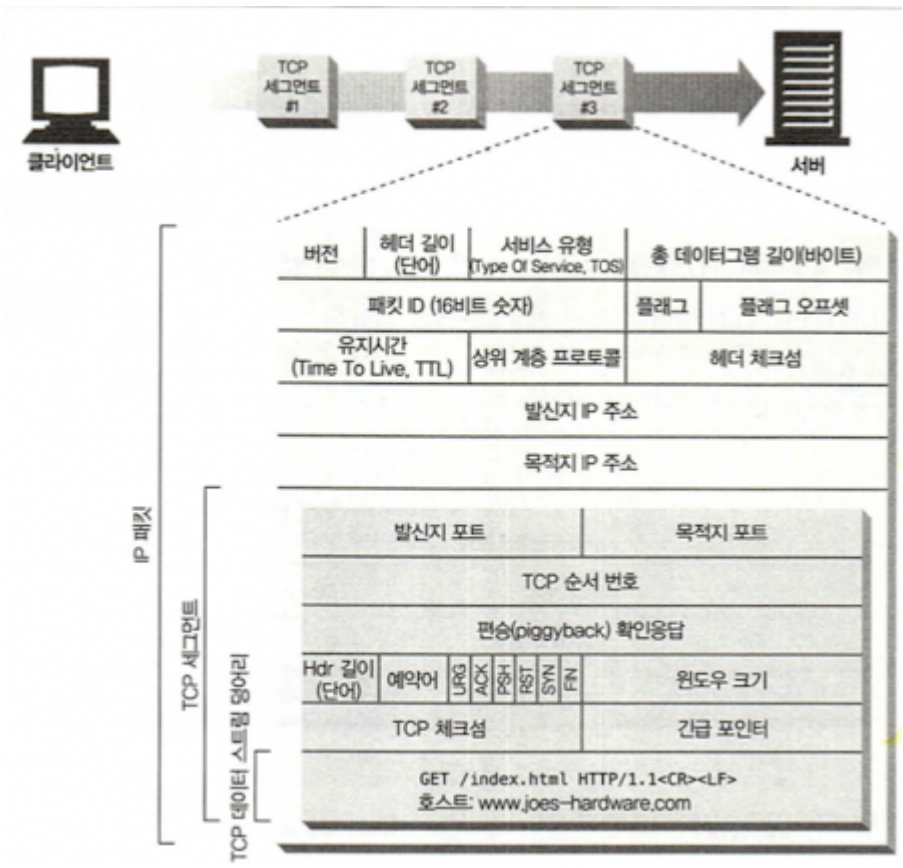
4.1.2 TCP 스트림은 세그먼트로 나뉘어 IP 패킷을 통해 전송된다



- HTTP 와 HTTPS 네트워크 프로토콜 스택

TCP 는 세그먼트 단위로 데이터 스트림을 잘게 나누고, 세그먼트를 IP 패킷이라는 봉투에 담아 인터넷을 통해 데이터를 전달한다.

아래 과정은 HTTP 프로그래머에게 보이지 않는다.



- IP 패킷은 TCP 데이터 스트림의 덩어리를 운반하는 TCP 세그먼트를 실어 나른다.

4.1.3 TCP 커넥션 유지하기

컴퓨터는 항상 TCP 커넥션을 여러 개 가지고 있다. TCP 는 포트 번호를 통해 여러 개의 커넥션 유지

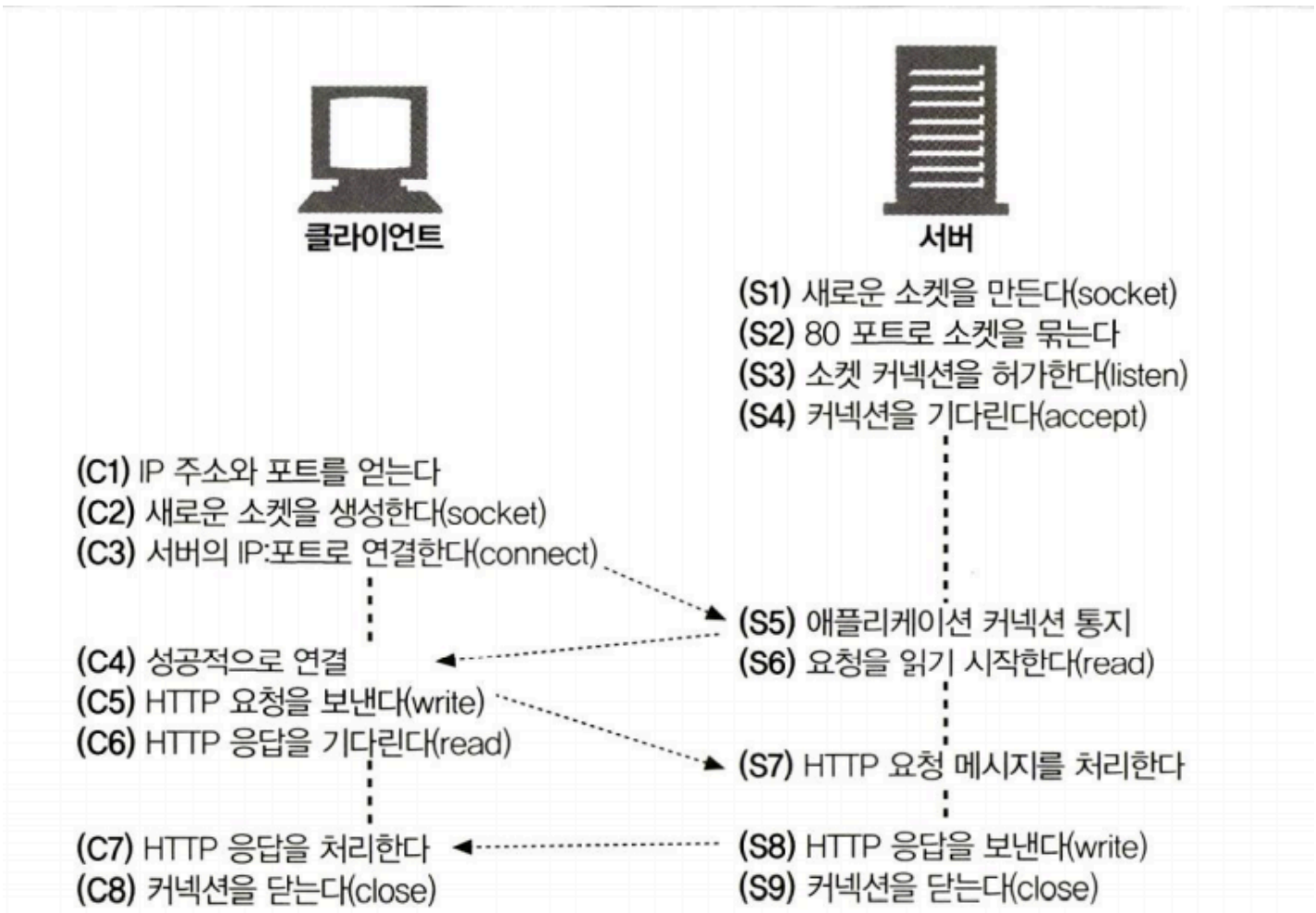
4.1.4 TCP 소켓 프로그래밍

운영체제는 TCP 커넥션의 생성과 관련된 여러 기능을 제공한다.

소켓 API 호출	설명
s = socket(〈parameters〉)	연결이 되지 않은 익명의 새로운 소켓 생성
bind(s, 〈local IP:port〉)	소켓에 로컬 포트 번호와 인터페이스 할당
connect(s, 〈remote IP:port〉)	로컬의 소켓과 원격의 호스트 및 포트 사이에 TCP 커넥션 생성
listen(s,...)	커넥션을 받아들이기 위해 로컬 소켓에 허용함을 표시
s2 = accept(s)	누군가 로컬 포트에 커넥션을 맺기를 기다림
n = read(s,buffer,n)	소켓으로부터 버퍼에 n바이트 읽기 시도
n = write(s,buffer,n)	소켓으로부터 버퍼에 n바이트 쓰기 시도
close(s)	TCP 커넥션을 완전히 끊음
shutdown(s, 〈side〉)	TCP 커넥션의 입출력만 닫음
getsockopt(s, . . .)	내부 소켓 설정 옵션값을 읽음
setsockopt(s, . . .)	내부 소켓 설정 옵션값을 변경

- TCP 커넥션 프로그래밍을 위한 공통 소켓 인터페이스 함수들

TCP API 는 기본적인 네트워크 프로토콜의 핸드셰이킹 및 모든 세부사항을 외부로부터 숨긴다.

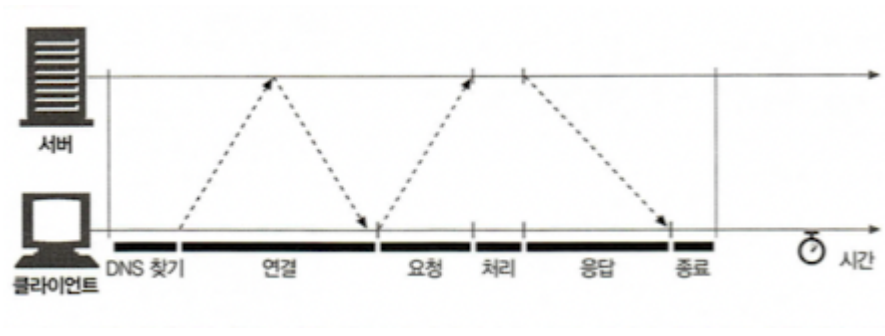


- 클라이언트와 서버가 TCP 소켓 인터페이스를 사용하여 상호작용하는 방법
- 3way 핸드셰이크와 4way 핸드셰이크 (https://jeongkyun-it.tistory.com/180)

4.2 TCP의 성능에 대한 고려

HTTP 는 TCP 위에 있는 계층이기 때문에 HTTP 트랜잭션의 성능은 그 아래 계층인 TCP 성능에 영향을 받는다.

4.2.1 HTTP 트랜잭션 지연



- HTTP 트랜잭션이 처리되는 과정

트랜잭션이 처리되는 시간은 준비과정에 비해 상당히 짧은걸 확인할 수 있음

대부분의 HTTP 지연은 TCP 네트워크 지연 때문에 발생한다.

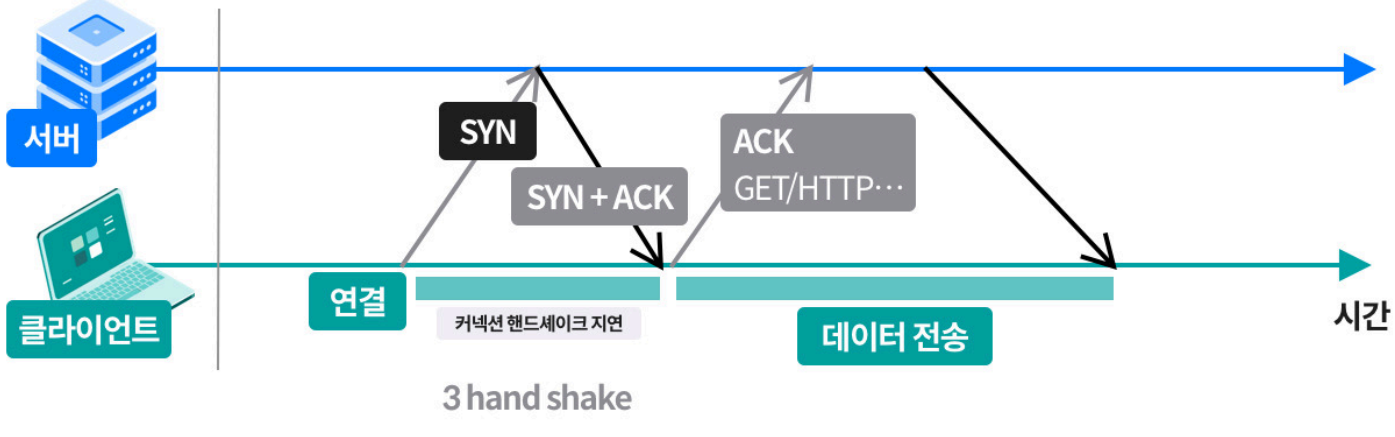
4.2.2 성능 관련 중요 요소

일반적인 TCP 관련 지연 목록

- TCP 커넥션의 핸드셰이크 설정
- 인터넷 혼잡제어를 위한 TCP 의 느린 시작
- 데이터를 한데 모아 전송하기 위한 네이글 알고리즘
- TCP 편승 확인응답을 위한 지연 알고리즘
- TIME_WAIT 지연과 포트 고갈

4.2.3 TCP 커넥션 핸드셰이크

어떤 데이터를 전송하든 새로운 TCP 커넥션을 열면 그 조건을 맞추기위해 연속으로 IP 패킷을 교환하는데.. 이게 위에 있는 TCP 핸드셰이크의 과정이다.



- TCP는 데이터를 전송하기 전, 커넥션 설정을 위해 두 개의 패킷 전송을 해야 한다.

4.2.4 확인응답 지연

인터넷 자체가 패킷 전송을 완벽히 보장하지는 않기 때문에 (인터넷 라우터는 과부하가 걸렸을 때 패킷을 맘대로 파기할 수 있다), TCP 는 성공적인 데이터 전송을 보장하기 위해 자체 확인 체계를 가진다.

checksum (중복체크)

- 각 TCP 세그먼트는 순번(시퀀스)과 데이터 무결성 체크섬을 가진다.
- 만약 송신자가 특정 시간 안에 확인응답 메시지를 받지 못하면 패킷이 파기되었거나 오류가 있는걸로 판단하고 데이터를 다시 요청한다..
- 네트워크를 좀 더 효율적으로 사용하기 위해 같은 방향의 송출 데이터 패킷에다가 확인응답패킷을 편승시키는데(끼워서 전달) 이러한 경우의 수를 늘리기 위해 확인응답지연 알고리즘을 사용
- 송신측은 수신측의 확인응답을 기다림, 수신측은 데이터를 정상수신했지만 편승시킬 패킷 찾느라 대기 → 지연

4.2.5 TCP 느린 시작(slow start)

TCP가 한번에 전송할 수 있는 패킷의 수를 제한한다.

예로, 처음 패킷을 1개만 전송 → 다음 2개 전송 → 다음 4개 전송 ...

이 혼잡제어 기능때문에 기존 커넥션에 비해 새로운 커넥션이 속도가 더 느리다.

- 이를 위해 이미 존재하는 커넥션을 재사용하는 기능이 있음 (지속커넥션)

4.2.6 네이글(Nagle) 알고리즘과 TCP_NODELAY

네이글 알고리즘은 알고리즘을 만든사람인 존 네이글의 이름을 따서 만들어졌다.

네트워크 효율을 위해 패킷 전송 전 많은 양의 TCP 데이터를 한 개의 덩어리로 합쳐서 보낸다.

네이글 알고리즘은 HTTP 성능 관련해 여러 문제를 발생시킨다.

- 크기가 작은 HTTP 메시지는 패킷을 채우지 못해 앞으로 생길지 않을지 모르는 추가 데이터를 기다리며 지연
- 확인응답 지연과 함께 쓰일 경우 형편없이 동작.
- HTTP 스택에 TCP_NODELAY 파라미터 값 설정 시 네이글 알고리즘을 비활성화 하기도 한다.
  - (실시간 응답이 필요한 서비스..)
  - 설정 시 작은 크기의 패킷이 너무 많이 생기지 않도록 큰 크기의 데이터 덩어리를 만들어야 한다.

4.2.7 TIME_WAIT의 누적과 포트 고갈

보통 실제상황에서는 문제를 발생시키진 않는다... 스킵

포트 고갈 문제를 겪지 않아도 커넥션을 너무 많이 맺거나 대기 상태로 있는 제어 블록이 많아지는 상황은 주의해야 한다.

4.3 HTTP 커넥션 관리

4.3.1 흔히 잘못 이해하는 Connection 헤더

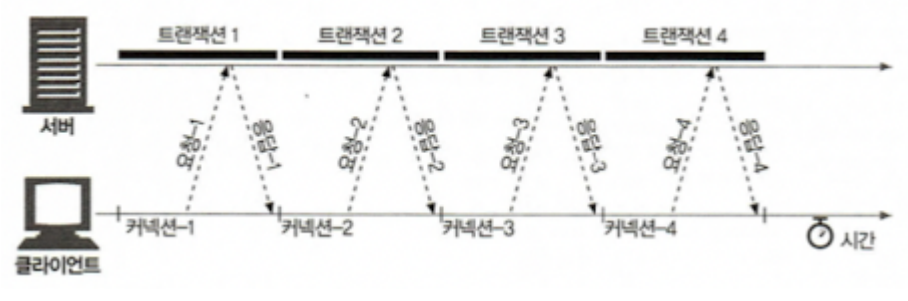
Connection 헤더에는 다음 세 가지 종류의 토큰이 전달될 수 있어 다소 혼란스러울 수 있다.

- HTTP 헤더 필드명은 이 커넥션에 해당되는 헤더들을 나열
- 임시적인 토큰 값은, 커넥션에 대한 비표준 옵션
- close 값은 커넥션이 작업이 완료되면 종료되어야 함을 의미

```
Connection : meter, close, bill-my-credit-card

- Meter 헤더를 다른 커넥션으로 전달하면 안되고
- bill-my-credit-card 옵션을 적용할 것이며
- 이 트랜잭션이 끝나면 커넥션이 끊길 것이다.
```

4.3.2 순차적인 트랜잭션 처리에 의한 지연

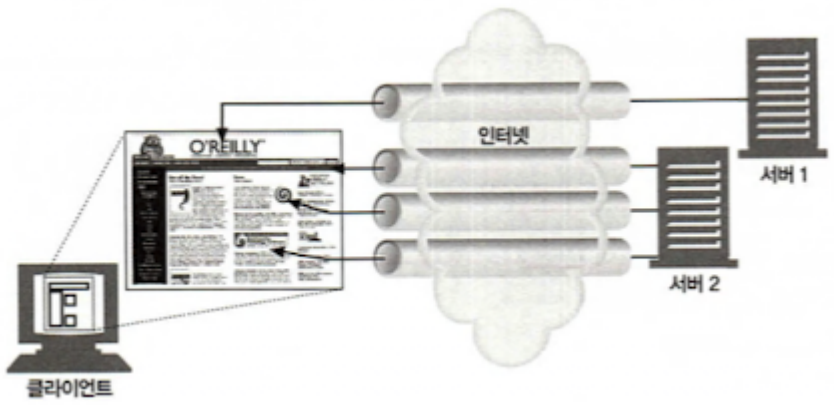


여러 이미지를 서버에서 요청해야 되는 상황을 가정해보면? 지연이 생길 수 밖에 없다.

HTTP 커넥션을 향상 시킬 수 있는 기술 4가지를 소개한다.

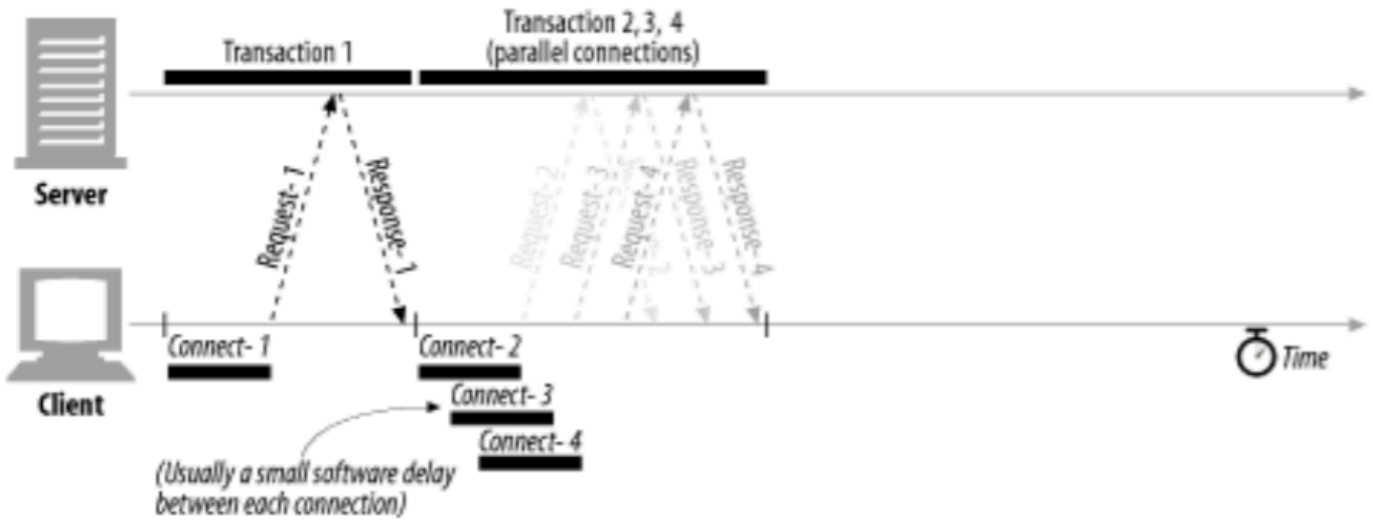
- 병렬 커넥션
  - 여러 개의 TCP 커넥션을 통한 동시 HTTP 요청
- 지속 커넥션
  - 커넥션을 맺고 끊는 데서 발생하는 지연을 제거하기 위한 TCP 커넥션의 재활용
- 파이프라인 커넥션
  - 공유 TCP 커넥션을 통한 병렬 HTTP 요청
- 다중 커넥션
  - 요청과 응답들에 대한 중재 (실험적 기술)

4.4 병렬 커넥션



말 그대로 병렬로 커넥션을 연결해 처리한다.

4.4.1 병렬 커넥션은 페이지를 더 빠르게 내려받는다



커넥션 지연이 겹치니 당연히 총 지연시간이 줄어들게 된다.

4.4.2 병렬 커넥션이 항상 더 빠르지는 않다

- 일반적인 경우엔 병렬 커넥션이 빠르지만 클라이언트의 대역폭이 좁으면.. 대부분의 시간을 전송에만 쓸 것이라 성능상의 장점이 없어짐
- 그 외 다수 커넥션은 메모리를 많이 소모하여 자체적 성능 문제를 발생시킨다.
- 현재 최신 브라우저는 병렬 커넥션 개수를 6~8개 정도 지원한다.
    - 크롬의 경우 HTTP/1.1 스펙에서 도메인당 최대 6개의 커넥션을 지원

4.5 지속 커넥션

- 웹 클라이언트는 보통 같은 사이트에 여러 개의 커넥션을 맺고 서버에 여러번 요청하게 될 것이다. → 사이트 지역성 (site locality)
- 처리가 완료된 후에도 계속 연결된 상태로 있는 TCP 커넥션을 지속 커넥션 이라 함

4.5.1 지속 커넥션 vs 병렬 커넥션

병렬 커넥션의 단점

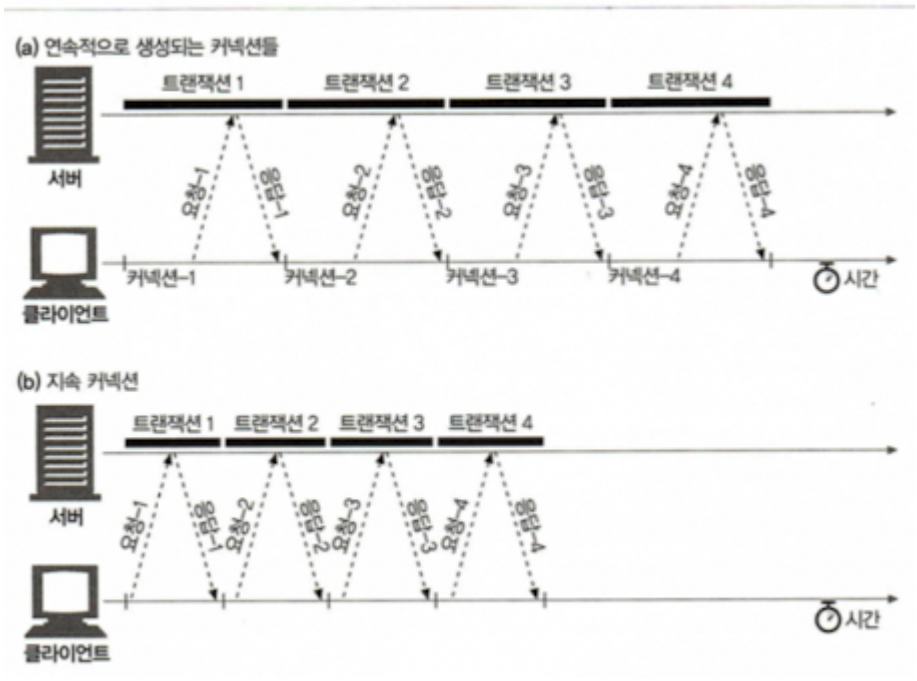
- 각 트랜잭션마다 새로운 커넥션을 맺고 끊기 때문에 시간과 대역폭 소모
- 각각의 새로운 커넥션은 TCP 느린 시작 때문에 성능 저하
- 실제로 연결할 수 있는 병렬 커넥션 수 제한

그에 비해 지속 커넥션의 장점

- 커넥션 작업과 지연 줄여줌
- 튜닝된 커넥션(TCP 느린시작 참고) 유지
- 커넥션 수 감소

지속커넥션은 병렬 커넥션과 함께 사용될 때 가장 효과적이다.

4.5.2 HTTP/1.0+ 의 Keep-Alive 커넥션



- keep-alive 커넥션의 장점, 커넥션 맺고 끊는 데 필요한 작업이 없어 시간 단축된 모습(b)

4.5.3 Keep-Alive 동작

kee-alive 는 사용하지 않기로 결정되어 HTTP/1.1 스펙에는 빠졌다.

하지만 아직도 브라우저와 서버 간 keep-alive 핸드셰이크가 널리 사용되고 있기 때문에 HTTP 애플리케이션은 그것을 처리할 수 있게 개발해야 한다.

HTTP/1.0 에서는 요청에 Connection:Keep-Alive 헤더를 포함시킨다.

- 응답 시에도 마찬가지로 포함시킴
- 응답 시 포함하지 않으면 서버가 keep-alive 지원 안함

4.5.5 Keep-Alive 커넥션 제한과 규칙

- keep-alive는 HTTP/1.0 에서 기본으로 사용되지는 않고 헤더를 포함해야한다.
- 커넥션이 끊어지기 전 엔티티 본문의 길이를 알 수 있어야 커넥션을 유지할 수 있다.
  - 엔티티 본문이 정확한 Content-Length 값과 함께 멀티파트 미디어 형식을 가지거나 청크 전송 인코딩으로 인코드 되어야 함을 뜻함
- 프록시와 게이트웨이는 Connection 헤더의 규칙을 철저히 지켜야 한다.

... HTTP/1.0 얘기는 그만 알아보자..

4.5.8 HTTP/1.1의 지속 커넥션

HTTP/1.1 에서는 keep-alive 커넥션을 지원하지 않는 대신, 설계가 더 개선된 지속 커넥션을 지원한다.

목적은 keep-alive 커넥션과 같지만 그에 비해 더 잘 동작한다.

HTTP/1.0의 keep-alive 커넥션과 달리 HTTP/1.1의 지속 커넥션은 기본으로 활성화 되어 있다.

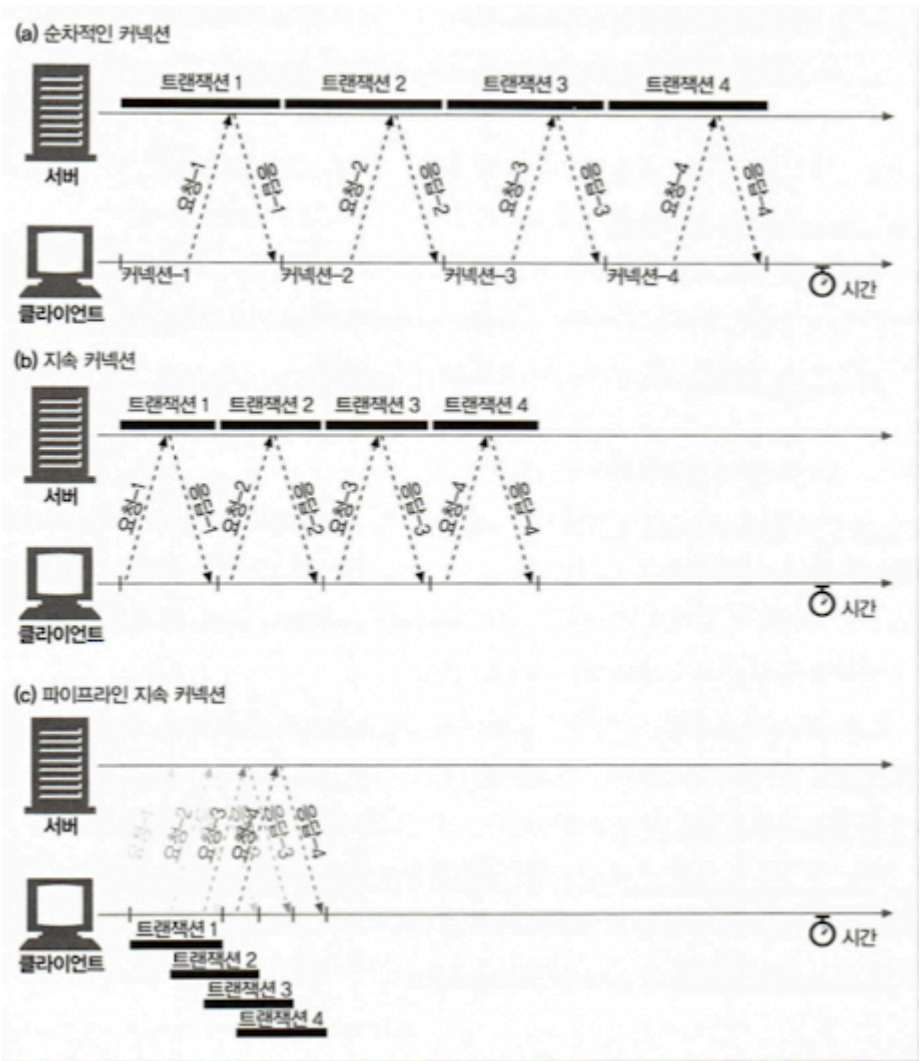
- 별도 설정을 하지 않는 한, 모든 커넥션을 지속 커넥션으로 취급한다.

HTTP/1.1 애플리케이션은 트랜잭션이 끝난 다음 커넥션을 끊으려면 Connection:close 헤더를 명시해야 한다.
Connection:close 를 안보낸다고 서버가 커넥션을 영원히 유지한다는 뜻은 아니다.

4.5.9 지속 커넥션의 제한과 규칙

- 클라이언트가 요청에 Connection:close 헤더를 포함해 보내면, 클라이언트는 그 커넥션으로 추가 요청을 보낼 수 없다.
- 커넥션에 있는 모든 메시지가 자신의 길이정보를 정확히 가지고 있을 때에만 커넥션을 지속 시킬 수 있다.
- HTTP/1.1 프록시는 클라이언트와 서버 각각에 대해 별도의 지속 커넥션을 맺고 관리해야 한다.
- HTTP/1.1 프록시 서버는 클라이언트가 커넥션 관련 기능에 대한 클라이언트의 지원 범위를 알고 있지 않는 한 지속 커넥션을 맺으면 안된다.. 하지만 현실적으로 쉽지 않고 많은 벤더가 이 규칙을 지키지 않음
- HTTP/1.1 기기는 Connection 헤더 값과 상관없이 언제든 커넥션을 끊을 수 있다.
- HTTP/1.1 애플리케이션은 중간에 끊어지는 커넥션을 복구할 수 있어야만 한다.
  - 클라이언트는 다시 보내도 문제가 없는 요청이면 가능한 다시 보내야 한다.
- 하나의 사용자 클라이언트는 서버 과부하 방지를 위해, 최소 두 개 의 지속커넥션만을 유지해야 한다. (2N)

4.6 파이프라인 커넥션



HTTP/1.1은 지속 커넥션을 통해 요청을 파이프라이닝 할 수 있다.
여러 개의 요청은 응답이 도착하기 전까지 큐에 쌓인다.

- 이는 대기 시간이 긴 네트워크 상황에서 성능을 높여준다.

4.7 커넥션 끊기에 대한 미스터리

커넥션 관리에는 명확한 기준이 없다. 이 이슈는 수많은 개발자가 알고 있는 것보다 더 미묘하며, 그에 관한 기술 문서도 별로 없다..

4.7.1 '마음대로' 커넥션 끊기

어떠한 HTTP 클라이언트, 서버, 혹은 프록시든 언제든 TCP 전송 커넥션을 끊을 수 있다.

이 상황은 파이프라인 지속 커넥션에서와 같다.

HTTP 애플리케이션은 언제든 지속 커넥션을 임의로 끊을 수 있다.

4.7.2 Content-Length 와 Truncation

각 HTTP 응답은 본문의 정확한 크기 값을 가지는 Content-Length 헤더를 가지고 있어야 한다.

4.7.3 커넥션 끊기의 허용, 재시도, 멱등성

GET, HEAD, PUT, DELETE, TRACE, OPTIONS : 멱등성을 지킴

POST: 멱등성 지키지 않음

- 멱등성 : 한번 혹은 여러번 실행했는지에 상관없이 같은 결과를 반환한다면 멱등하다고 한다.

4.7.4 우아한 커넥션 끊기

- TCP 커넥션은 양방향이다.

전체 끊기와 절반 끊기

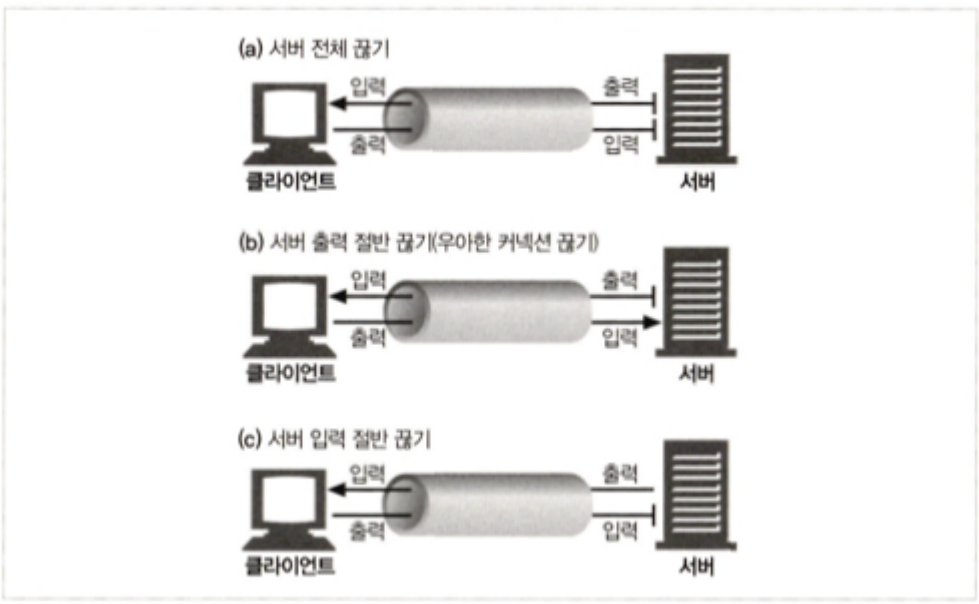


그림 4-20 전체 끊기와 절반 끊기

TCP 끊기와 리셋 에러

단순한 HTTP 애플리케이션은 전체 끊기만을 사용할 수 있다.

하지만 애플리케이션이 각기 다른 HTTP 클라이언트, 서버, 프록시와 통신 할 때, 그리고 그들과 파이프라인 지속 커넥션을 사용할 때, 기기들에 예상치 못한 쓰기 에러를 발생하는 것을 예방하기 위해 '절반끊기' 를 사용해야 한다.

- 보통은 커넥션의 출력 채널을 끊는 것이 안전하다.

    - 커넥션의 반대편에 있는 기기는 모든 데이터를 버퍼로부터 읽고 나서 데이터 전송이 끝남과 동시에 당신이 커넥션을 끊었다는걸 알게 됨
- 클라이언트에서 더는 데이터를 보내지 않는다는걸 확신할 수 없는 이상, 커넥션의 입력 채널을 끊는건 위험
    - 만약 클라이언트에서 이미 끊긴 입력 채널에 데이터 전송 시 서버의 운영체제는 한번쯤 봤을법한 메시지인‘connection reset by peer’ 메시지를 클라이언트에게 보낼 것이다.
    - 운영체제는 위 메시지를 심각한 에러로 취급하여 버퍼에 저장된 읽히지 않은 데이터를 모두 삭제 해버린다.

우아하게 커넥션 끊기

- 일반적으로 애플리케이션이 우아한 커넥션 끊기를 구현하는 건 애플리케이션 자신의 출력 채널을 먼저 끊고 다른 쪽에 있는 기기의 출력 채널이 끊기는 것을 기다리는 것이다.
- 양쪽에서 더는 데이터를 전송하지 않을 것이라 알려주면, 커넥션은 리셋의 위험 없이 온전히 종료된다.

그러나 안타깝게도 상대방이 절반 끊기를 구현했다는 보장도 없고, 내가 하는 절반 끊기를 검사해준다는 보장도 없다.

따라서 커넥션을 우아하게 끊고자 하는 애플리케이션은 출력 채널을 끊은 후, 데이터 전송이 끝났는지 확인하기 위해 입력 채널에 대해 상태 검사를 주기적으로 해야 한다.