HTTP & HTTPS

- **HTTP(HyperText Transfer Protocol)**

  인터넷 상에서 클라이언트와 서버가 자원을 주고 받을 때 쓰는 통신 규약

HTTP는 텍스트 교환이므로, 누군가 네트워크에서 신호를 가로채면 내용이 노출되는 보안 이슈가 존재한다.

이런 보안 문제를 해결해주는 프로토콜이 **'HTTPS'**

- **HTTPS(HyperText Transfer Protocol Secure)**

  인터넷 상에서 정보를 암호화하는 SSL 프로토콜을 사용해 클라이언트와 서버가 자원을 주고 받을 때 쓰는 통신 규약

HTTPS는 텍스트를 암호화한다. (공개키 암호화 방식으로!) : 공개키 설명

HTTPS 통신 흐름

1. 애플리케이션 서버(A)를 만드는 기업은 HTTPS를 적용하기 위해 공개키와 개인키를 만든다.
2. 신뢰할 수 있는 CA 기업을 선택하고, 그 기업에게 내 공개키 관리를 부탁하며 계약을 한다.

CA 란? : Certificate Authority로, 공개키를 저장해주는 신뢰성이 검증된 민간기업

3. 계약 완료된 CA 기업은 해당 기업의 이름, A서버 공개키, 공개키 암호화 방법을 담은 인증서를 만들고, 해당 인증서를 CA 기업의 개인키로 암호화해서 A서버에게 제공한다.
4. A서버는 암호화된 인증서를 갖게 되었다. 이제 A서버는 A서버의 공개키로 암호화된 HTTPS 요청이 아닌 요청이 오면, 이 암호화된 인증서를 클라이언트에게 건내준다.
5. 클라이언트가 `main.html` 파일을 달라고 A서버에 요청했다고 가정하자. HTTPS 요청이 아니기 때문에 CA기업이 A서버의 정보를 CA 기업의 개인키로 암호화한 인증서를 받게 된다.

CA 기업의 공개키는 브라우저가 이미 알고있다. (세계적으로 신뢰할 수 있는 기업으로 등록되어 있기 때문에, 브라우저가 인증서를 탐색하여 해독이 가능한 것)

6. 브라우저는 해독한 뒤 A서버의 공개키를 얻게 되었다.
7. 클라이언트가 A서버와 HandShaking 과정에서 주고받은 난수를 조합하여 pre-master-key(대칭키) 를 생성한 뒤, A서버의 공개키로 해당 대칭키를 암호화하여 서버로 보냅니다.
8. A서버는 암호화된 대칭키를 자신의 개인키로 복호화 하여 클라이언트와 동일한 대칭키를 획득합니다.

9. 이후 클라이언트-서버사이의 통신을 할 때 주고받는 메세지는 이 pre-master-key(대칭키)를 이용하여 암호화, 복호화를 진행합니다.

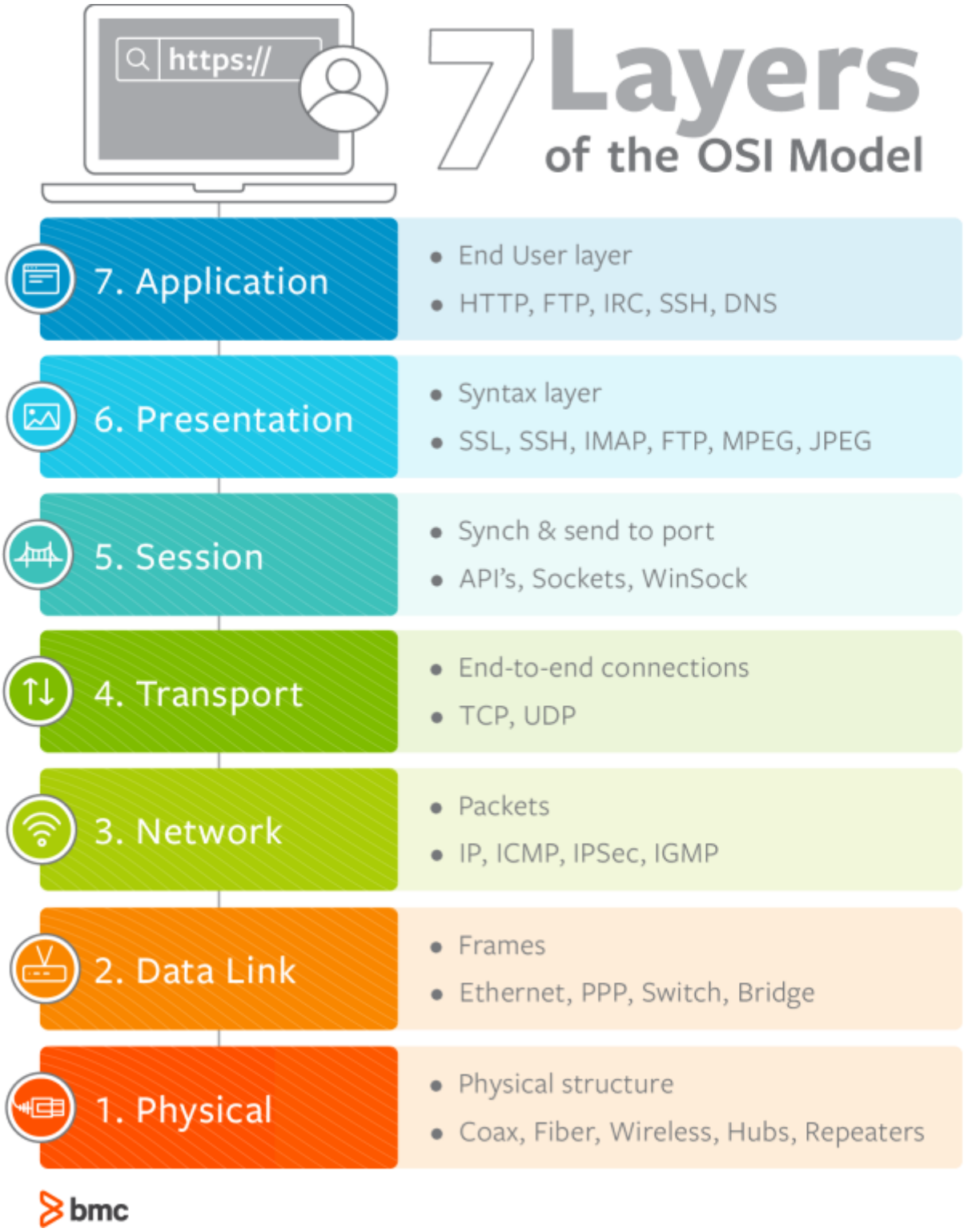
HTTPS도 무조건 안전한 것은 아니다. (신뢰받는 CA 기업이 아닌 자체 인증서 발급한 경우 등)

이때는 HTTPS지만 브라우저에서 주의 요함, 안전하지 않은 사이트와 같은 알림으로 주의 받게 된다.

[참고사항]

링크

OSI 7 계층



7계층은 왜 나눌까?

통신이 일어나는 과정을 단계별로 알 수 있고, 특정한 곳에 이상이 생기면 그 단계만 수정할 수 있기 때문이다.

1) 물리(Physical)

> 리피터, 케이블, 허브 등

단지 데이터 전기적인 신호로 변환해서 주고받는 기능을 진행하는 공간

즉, 데이터를 전송하는 역할만 진행한다.

2) 데이터 링크(Data Link)

> 브릿지, 스위치 등

물리 계층으로 송수신되는 정보를 관리하여 안전하게 전달되도록 도와주는 역할

Mac 주소를 통해 통신한다. 프레임에 Mac 주소를 부여하고 에러검출, 재전송, 흐름제어를 진행한다.

3) 네트워크(Network)

> 라우터, IP

데이터를 목적지까지 가장 안전하고 빠르게 전달하는 기능을 담당한다.

라우터를 통해 이동할 경로를 선택하여 IP 주소를 지정하고, 해당 경로에 따라 패킷을 전달해준다.

라우팅, 흐름 제어, 오류 제어, 세그먼테이션 등을 수행한다.

4) 전송(Transport)

> TCP, UDP

TCP와 UDP 프로토콜을 통해 통신을 활성화한다. 포트를 열어두고, 프로그램들이 전송을 할 수 있도록 제공해준다.

- TCP : 신뢰성, 연결지향적
- UDP : 비신뢰성, 비연결성, 실시간

5) 세션(Session)

> API, Socket

데이터가 통신하기 위한 논리적 연결을 담당한다. TCP/IP 세션을 만들고 없애는 책임을 지니고 있다.

6) 표현(Presentation)

> JPEG, MPEG 등

데이터 표현에 대한 독립성을 제공하고 암호화하는 역할을 담당한다.

파일 인코딩, 명령어를 포장, 압축, 암호화한다.

7) 응용(Application)

> HTTP, FTP, DNS 등

최종 목적지로, 응용 프로세스와 직접 관계하여 일반적인 응용 서비스를 수행한다.

사용자 인터페이스, 전자우편, 데이터베이스 관리 등의 서비스를 제공한다.

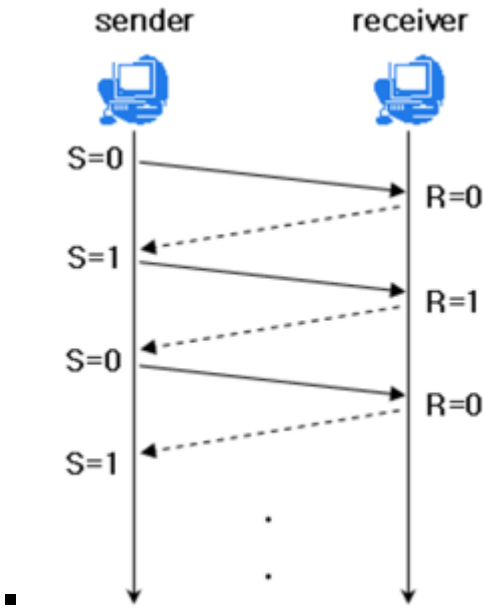
TCP (흐름제어/혼잡제어)

들어가기 전

- TCP 통신이란?
  - 네트워크 통신에서 신뢰적인 연결방식
  - TCP는 기본적으로 unreliable network에서, reliable network를 보장할 수 있도록 하는 프로토콜
  - TCP는 network congestion avoidance algorithm을 사용
- reliable network를 보장한다는 것은 4가지 문제점 존재
  - 손실 : packet이 손실될 수 있는 문제
  - 순서 바뀜 : packet의 순서가 바뀌는 문제
  - Congestion : 네트워크가 혼잡한 문제
  - Overload : receiver가 overload 되는 문제
- 흐름제어/혼잡제어란?
  - 흐름제어 (endsystem 대 endsystem)
    - 송신측과 수신측의 데이터 처리 속도 차이를 해결하기 위한 기법
    - Flow Control은 receiver가 packet을 지나치게 많이 받지 않도록 조절하는 것
    - 기본 개념은 receiver가 sender에게 현재 자신의 상태를 feedback 한다는 점
  - 혼잡제어 : 송신측의 데이터 전달과 네트워크의 데이터 처리 속도 차이를 해결하기 위한 기법
- 전송의 전체 과정
  - Application layer : sender application layer가 socket에 data를 씀.
  - Transport layer : data를 segment에 감싼다. 그리고 network layer에 넘겨줌.
  - 그러면 아랫단에서 어쨋든 receiving node로 전송이 됨. 이 때, sender의 send buffer에 data를 저장하고, receiver는 receive buffer에 data를 저장함.
  - application에서 준비가 되면 이 buffer에 있는 것을 읽기 시작함.
  - 따라서 flow control의 핵심은 이 receiver buffer가 넘치지 않게 하는 것임.
  - 따라서 receiver는 RWND(Receive WiNDow) : receive buffer의 남은 공간을 홍보함

1. 흐름제어 (Flow Control)

- 수신측이 송신측보다 데이터 처리 속도가 빠르면 문제없지만, 송신측의 속도가 빠를 경우 문제가 생긴다.
- 수신측에서 제한된 저장 용량을 초과한 이후에 도착하는 데이터는 손실 될 수 있으며, 만약 손실 된다면 불필요하게 응답과 데이터 전송이 송/수신 측 간에 빈번히 발생한다.
- 이러한 위험을 줄이기 위해 송신 측의 데이터 전송량을 수신측에 따라 조절해야한다.
- 해결방법
  - Stop and Wait : 매번 전송한 패킷에 대해 확인 응답을 받아야만 그 다음 패킷을 전송하는 방법

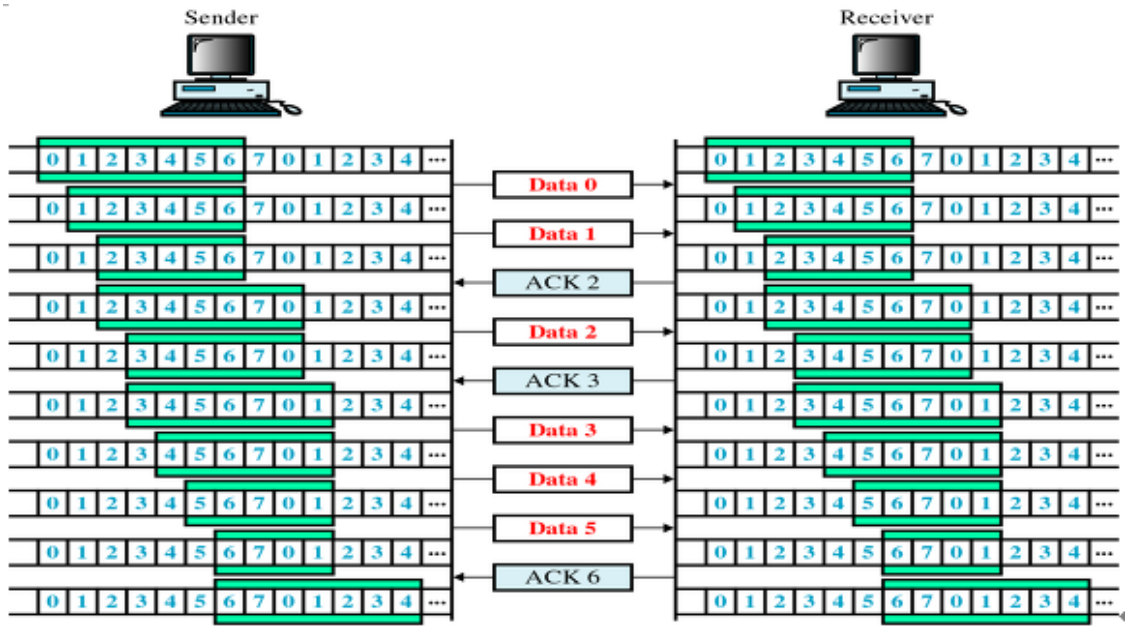


- Sliding Window (Go Back N ARQ)
  - 수신측에서 설정한 윈도우 크기만큼 송신측에서 확인응답없이 세그먼트를 전송할 수 있게 하여 데이터 흐름을 동적으로 조절하는 제어기법
  - 목적 : 전송은 되었지만, acked를 받지 못한 byte의 숫자를 파악하기 위해 사용하는 protocol

    LastByteSent - LastByteAcked <= ReceivecWindowAdvertised

    (마지막에 보내진 바이트 - 마지막에 확인된 바이트 <= 남아있는 공간) ==

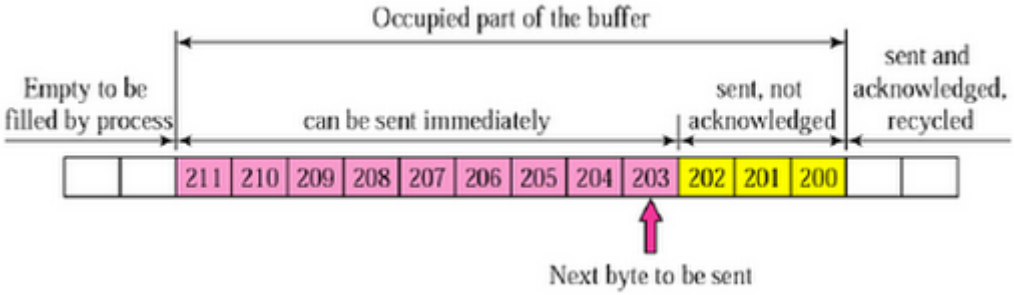
    (현재 공중에 떠있는 패킷 수 <= sliding window)
- 동작방식 : 먼저 윈도우에 포함되는 모든 패킷을 전송하고, 그 패킷들의 전달이 확인되는대로 이 윈도우를 옆으로 옮김으로써 그 다음 패킷들을 전송



- Window : TCP/IP를 사용하는 모든 호스트들은 송신하기 위한 것과 수신하기 위한 2개의 Window를 가지고 있다. 호스트들은 실제 데이터를 보내기 전에 '3 way handshaking'을 통해 수신 호스트의 receive window size에 자신의 send window size를 맞추게 된다.

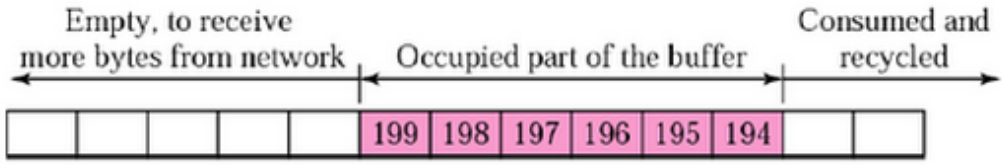
- 세부구조

1. 송신 버퍼

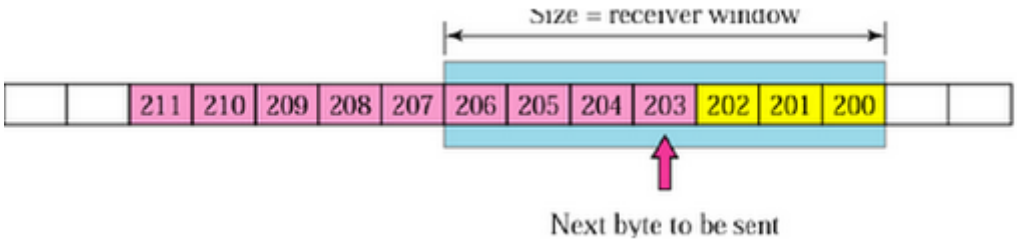


-
  - 200 이전의 바이트는 이미 전송되었고, 확인응답을 받은 상태
  - 200 ~ 202 바이트는 전송되었으나 확인응답을 받지 못한 상태
  - 203 ~ 211 바이트는 아직 전송이 되지 않은 상태

2. 수신 윈도우

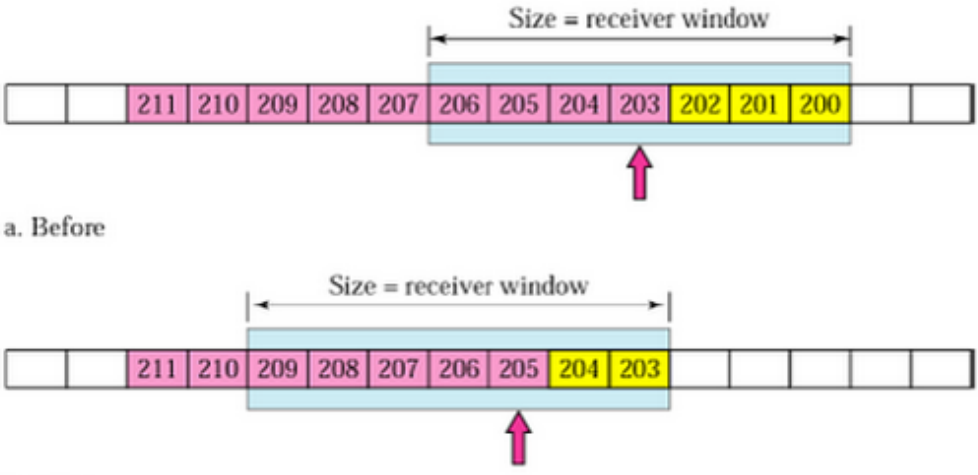


3. 송신 윈도우



  - 수신 윈도우보다 작거나 같은 크기로 송신 윈도우를 지정하게되면 흐름제어가 가능하다.

4. 송신 윈도우 이동

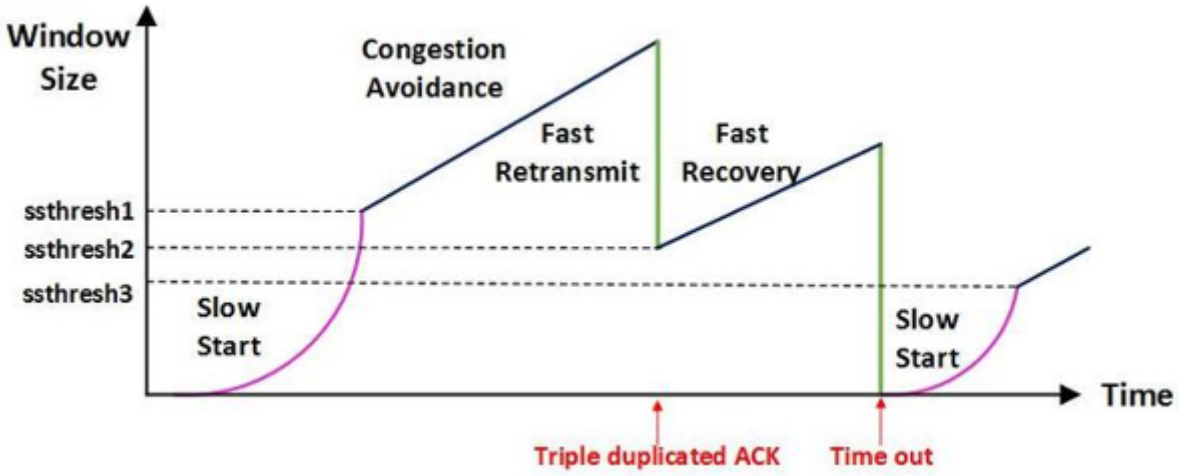


  - Before : 203 ~ 204를 전송하면 수신측에서는 확인 응답 203을 보내고, 송신측은 이를 받아 after 상태와 같이 수신 윈도우를 203 ~ 209 범위로 이동
  - after : 205 ~ 209가 전송 가능한 상태

5. Selected Repeat

2. 혼잡제어 (Congestion Control)

- 송신측의 데이터는 지역망이나 인터넷으로 연결된 대형 네트워크를 통해 전달된다. 만약 한 라우터에 데이터가 몰릴 경우, 자신에게 온 데이터를 모두 처리할 수 없게 된다. 이런 경우 호스트들은 또 다시 재전송을 하게되고 결국 혼잡만 가중시켜 오버플로우나 데이터 손실을 발생시키게 된다. 따라서 이러한 네트워크의 혼잡을 피하기 위해 송신측에서 보내는 데이터의 전송속도를 강제로 줄이게 되는데, 이러한 작업을 혼잡제어라고 한다.
- 또한 네트워크 내에 패킷의 수가 과도하게 증가하는 현상을 혼잡이라 하며, 혼잡 현상을 방지하거나 제거하는 기능을 혼잡제어라고 한다.
- 흐름제어가 송신측과 수신측 사이의 전송속도를 다루는데 반해, 혼잡제어는 호스트와 라우터를 포함한 보다 넓은 관점에서 전송 문제를 다루게 된다.
- 해결 방법



  - AIMD(Additive Increase / Multiplicative Decrease)
    - 처음에 패킷을 하나씩 보내고 이것이 문제없이 도착하면 window 크기(단위 시간 내에 보내는 패킷의 수)를 1씩 증가시켜가며 전송하는 방법
    - 패킷 전송에 실패하거나 일정 시간을 넘으면 패킷의 보내는 속도를 절반으로 줄인다.
    - 공평한 방식으로, 여러 호스트가 한 네트워크를 공유하고 있으면 나중에 진입하는 쪽이 처음에는 불리하지만, 시간이 흐르면 평형상태로 수렴하게 되는 특징이 있다.
    - 문제점은 초기에 네트워크의 높은 대역폭을 사용하지 못하여 오랜 시간이 걸리게 되고, 네트워크가 혼잡해지는 상황을 미리 감지하지 못한다. 즉, 네트워크가 혼잡해지고 나서야 대역폭을 줄이는 방식이다.
  - Slow Start (느린 시작)
    - AIMD 방식이 네트워크의 수용량 주변에서는 효율적으로 작동하지만, 처음에 전송 속도를 올리는데 시간이 오래 걸리는 단점이 존재했다.
    - Slow Start 방식은 AIMD와 마찬가지로 패킷을 하나씩 보내면서 시작하고, 패킷이 문제없이 도착하면 각각의 ACK 패킷마다 window size를 1씩 늘려준다. 즉, 한 주기가 지나면 window size가 2배로 된다.
    - 전송속도는 AIMD에 반해 지수 함수 꼴로 증가한다. 대신에 혼잡 현상이 발생하면 window size를 1로 떨어뜨리게 된다.
    - 처음에는 네트워크의 수용량을 예상할 수 있는 정보가 없지만, 한번 혼잡 현상이 발생하고 나면 네트워크의 수용량을 어느 정도 예상할 수 있다.

        - 그러므로 혼잡 현상이 발생하였던 window size의 절반까지는 이전처럼 지수 함수 꼴로 창 크기를 증가시키고 그 이후부터는 완만하게 1씩 증가시킨다.
    - Fast Retransmit (빠른 재전송)
        - 빠른 재전송은 TCP의 혼잡 조절에 추가된 정책이다.
        - 패킷을 받는 쪽에서 먼저 도착해야할 패킷이 도착하지 않고 다음 패킷이 도착한 경우에도 ACK 패킷을 보내게 된다.
        - 단, 순서대로 잘 도착한 마지막 패킷의 다음 패킷의 순번을 ACK 패킷에 실어서 보내게 되므로, 중간에 하나가 손실되게 되면 송신 측에서는 순번이 중복된 ACK 패킷을 받게 된다. 이것을 감지하는 순간 문제가 되는 순번의 패킷을 재전송 해줄 수 있다.
        - 중복된 순번의 패킷을 3개 받으면 재전송을 하게 된다. 약간 혼잡한 상황이 일어난 것이므로 혼잡을 감지하고 window size를 줄이게 된다.
    - Fast Recovery (빠른 회복)
        - 혼잡한 상태가 되면 window size를 1로 줄이지 않고 반으로 줄이고 선형증가시키는 방법이다. 이 정책까지 적용하면 혼잡 상황을 한번 겪고 나서부터는 순수한 AIMD 방식으로 동작하게 된다.

[ref]

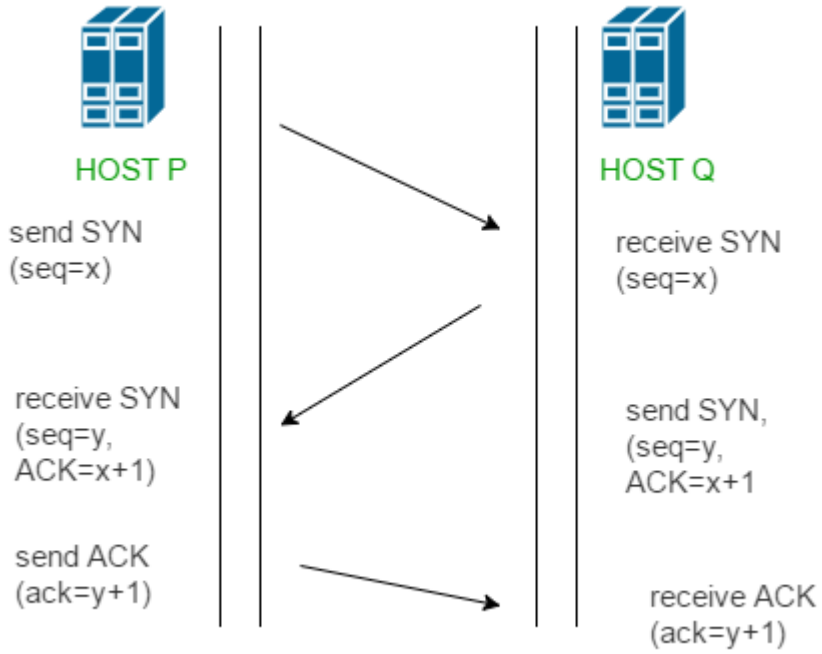
- https://www.brianstorti.com/tcp-flow-control/
- https://www.brianstorti.com/tcp-flow-control/

[TCP] 3 way handshake & 4 way handshake

> 연결을 성립하고 해제하는 과정을 말한다

3 way handshake - 연결 성립

TCP는 정확한 전송을 보장해야 한다. 따라서 통신하기에 앞서, 논리적인 접속을 성립하기 위해 3 way handshake 과정을 진행한다.

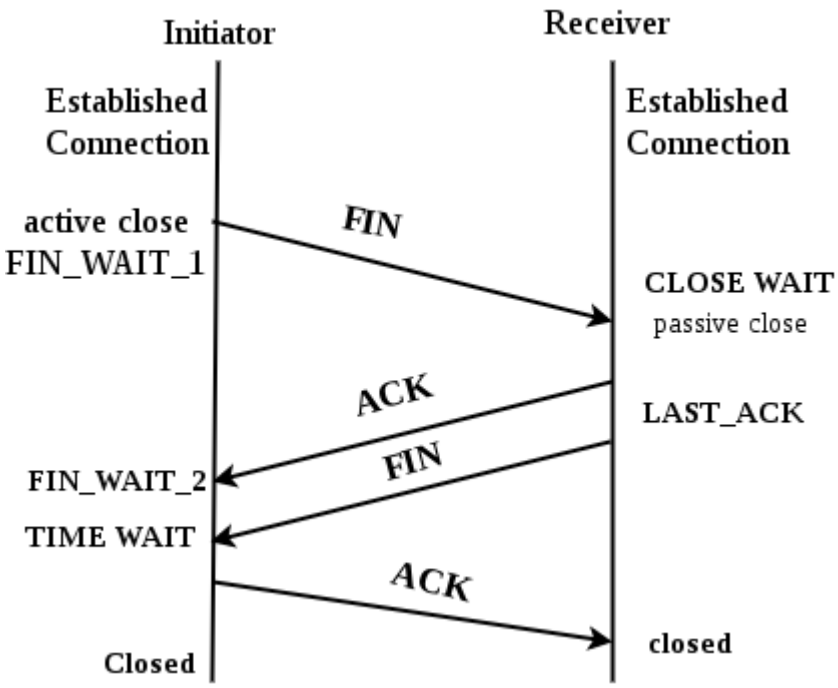


1. 클라이언트가 서버에게 SYN 패킷을 보냄 (sequence : x)
2. 서버가 SYN(x)을 받고, 클라이언트로 받았다는 신호인 ACK와 SYN 패킷을 보냄 (sequence : y, ACK : x + 1)
3. 클라이언트는 서버의 응답은 ACK(x+1)와 SYN(y) 패킷을 받고, ACK(y+1)를 서버로 보냄

이렇게 3번의 통신이 완료되면 연결이 성립된다. (3번이라 3 way handshake인 것)

4 way handshake - 연결 해제

연결 성립 후, 모든 통신이 끝났다면 해제해야 한다.



1. 클라이언트는 서버에게 연결을 종료한다는 FIN 플래그를 보낸다.
2. 서버는 FIN을 받고, 확인했다는 ACK를 클라이언트에게 보낸다. (이때 모든 데이터를 보내기 위해 CLOSE_WAIT 상태가 된다)
3. 데이터를 모두 보냈다면, 연결이 종료되었다는 FIN 플래그를 클라이언트에게 보낸다.
4. 클라이언트는 FIN을 받고, 확인했다는 ACK를 서버에게 보낸다. (아직 서버로부터 받지 못한 데이터가 있을 수 있으므로 TIME_WAIT을 통해 기다린다.)

- 서버는 ACK를 받은 이후 소켓을 닫는다 (Closed)
- TIME_WAIT 시간이 끝나면 클라이언트도 닫는다 (Closed)

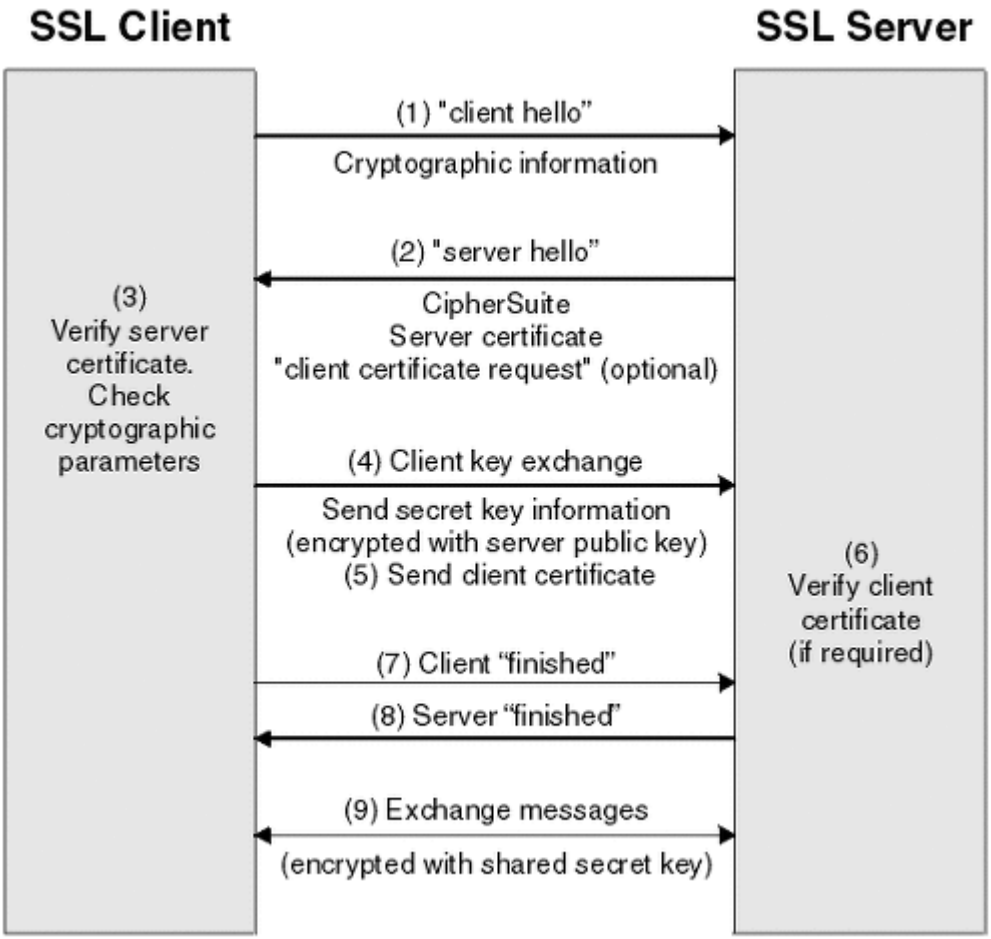
이렇게 4번의 통신이 완료되면 연결이 해제된다.

[참고 자료]

링크

TLS/SSL HandShake

> HTTPS에서 클라이언트와 서버간 통신 전
> SSL 인증서로 신뢰성 여부를 판단하기 위해 연결하는 방식



진행 순서

1. 클라이언트는 서버에게 `client hello` 메시지를 담아 서버로 보낸다. 이때 암호화된 정보를 함께 담는데, 버전, 암호 알고리즘, 압축 방식 등을 담는다.

2. 서버는 클라이언트가 보낸 암호 알고리즘과 압축 방식을 받고, 세션 ID와 CA 공개 인증서를 `server hello` 메시지와 함께 담아 응답한다. 이 CA 인증서에는 앞으로 통신 이후 사용할 대칭키가 생성되기 전, 클라이언트에서 handshake 과정 속 암호화에 사용할 공개키를 담고 있다.

3. 클라이언트 측은 서버에서 보낸 CA 인증서에 대해 유효한 지 CA 목록에서 확인하는 과정을 진행한다.

4. CA 인증서에 대한 신뢰성이 확보되었다면, 클라이언트는 난수 바이트를 생성하여 서버의 공개키로 암호화한다. 이 난수 바이트는 대칭키를 정하는데 사용이 되고, 앞으로 서로 메시지를 통신할 때 암호화하는데 사용된다.

5. 만약 2번 단계에서 서버가 클라이언트 인증서를 함께 요구했다면, 클라이언트의 인증서와 클라이언트의 개인키로 암호화된 임의의 바이트 문자열을 함께 보내준다.

6. 서버는 클라이언트의 인증서를 확인 후, 난수 바이트를 자신의 개인키로 복호화 후 대칭 마스터 키 생성에 활용한다.

7. 클라이언트는 handshake 과정이 완료되었다는 `finished` 메시지를 서버에 보내면서, 지금까지 보낸 교환 내역들을 해싱 후 그 값을 대칭키로 암호화하여 같이 담아 보내준다.

8. 서버도 동일하게 교환 내용들을 해싱한 뒤 클라이언트에서 보내준 값과 일치하는 지 확인한다. 일치하면 서버도 마찬가지로 `finished` 메시지를 이번에 만든 대칭키로 암호화하여 보낸다.

9. 클라이언트는 해당 메시지를 대칭키로 복호화하여 서로 통신이 가능한 신뢰받은 사용자란 걸 인지하고, 앞으로 클라이언트와 서버는 해당 대칭키로 데이터를 주고받을 수 있게 된다.

[참고 자료]

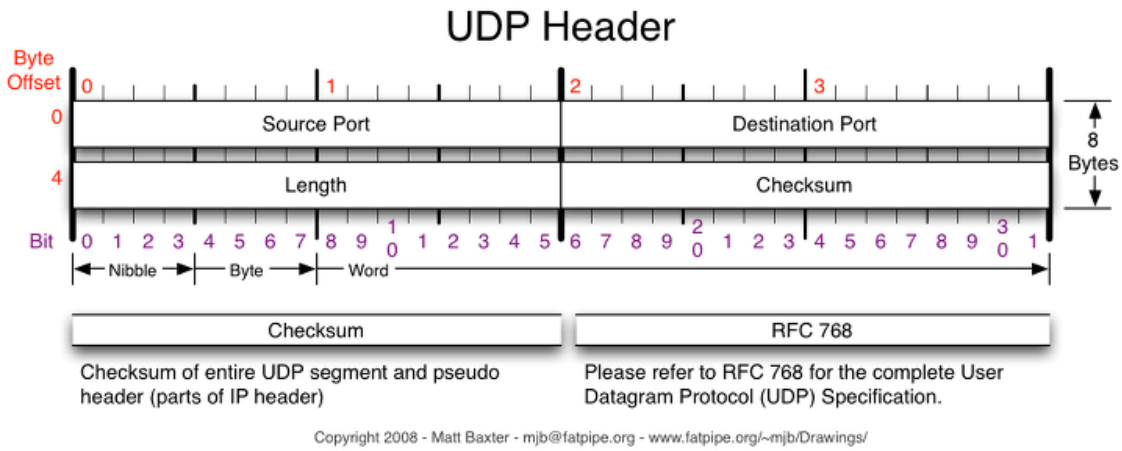
- 링크

2019.08.26.(월) [BYM] UDP란?

들어가기 전

- UDP 통신이란?
    - User Datagram Protocol의 약자로 데이터를 데이터그램 단위로 처리하는 프로토콜이다.
    - 비연결형, 신뢰성 없는 전송 프로토콜이다.
    - 데이터그램 단위로 쪼개면서 전송을 해야하기 때문에 전송 계층이다.
    - Transport layer에서 사용하는 프로토콜.
- TCP와 UDP는 왜 나오게 됐는가?
    1. IP의 역할은 Host to Host (장치 to 장치)만을 지원한다. 장치에서 장치로 이동은 IP로 해결되지만, 하나의 장비안에서 수많은 프로그램들이 통신을 할 경우에는 IP만으로는 한계가 있다.
    2. 또한, IP에서 오류가 발생한다면 ICMP에서 알려준다. 하지만 ICMP는 알려주기만 할 뿐 대처를 못하기 때문에 IP보다 위에서 처리를 해줘야 한다.
    - 1번을 해결하기 위하여 포트 번호가 나오게 됐고, 2번을 해결하기 위해 상위 프로토콜인 TCP와 UDP가 나오게 되었다.
    - *ICMP : 인터넷 제어 메시지 프로토콜로 네트워크 컴퓨터 위에서 돌아가는 운영체제에서 오류 메시지를 전송받는데 주로 쓰임
- 그렇다면 TCP와 UDP가 어떻게 오류를 해결하는가?
    - TCP : 데이터의 분실, 중복, 순서가 뒤바뀜 등을 자동으로 보정해줘서 송수신 데이터의 정확한 전달을 할 수 있도록 해준다.
    - UDP : IP가 제공하는 정도의 수준만을 제공하는 간단한 IP 상위 계층의 프로토콜이다. TCP와는 다르게 에러가 날 수도 있고, 재전송이나 순서가 뒤바뀔 수도 있어서 이 경우, 어플리케이션에서 처리하는 번거로움이 존재한다.
- UDP는 왜 사용할까?
    - UDP의 결정적인 장점은 데이터의 신속성이다. 데이터의 처리가 TCP보다 빠르다.
    - 주로 실시간 방송과 온라인 게임에서 사용된다. 네트워크 환경이 안 좋을때, 끊기는 현상을 생각하면 된다.
- DNS(Domain Name Service)에서 UDP를 사용하는 이유
    - Request의 양이 작음 -> UDP Request에 담길 수 있다.
    - 3 way handshaking으로 연결을 유지할 필요가 없다. (오버헤드 발생)
    - Request에 대한 손실은 Application Layer에서 제어가 가능하다.
    - DNS : port 53번
    - But, TCP를 사용할 때가 있다! 크기가 512(UDP 제한)이 넘을 때, TCP를 사용해야한다.

1. UDP Header



- - Source port : 시작 포트 - Destination port : 도착지 포트 - Length : 길이 - _Checksum_ : 오류 검출 - 중복 검사의 한 형태로, 오류 정정을 통해 공간이나 시간 속에서 송신된 자료의 무결성을 보호하는 단순한 방법이다.

- 이렇게 간단하므로, TCP 보다 용량이 가볍고 송신 속도가 빠르게 작동됨.
- 그러나 확인 응답을 못하므로, TCP보다 신뢰도가 떨어짐.
- UDP는 비연결성, TCP는 연결성으로 정의할 수 있음.

DNS과 UDP 통신 프로토콜을 사용함.

DNS는 데이터를 교환하는 경우임

이때, TCP를 사용하게 되면, 데이터를 송신할 때까지 세션 확립을 위한 처리를 하고, 송신한 데이터가 수신되었는지 점검하는 과정이 필요하므로, Protocol overhead가 UDP에 비해서 큼.

DNS는 Application layer protocol임.

모든 Application layer protocol은 TCP, UDP 중 하나의 Transport layer protocol을 사용해야 함.

(TCP는 reliable, UDP는 not reliable임) / DNS는 reliable해야할 것 같은데 왜 UDP를 사용할까?

사용하는 이유

1. TCP가 3-way handshake를 사용하는 반면, UDP는 connection 을 유지할 필요가 없음.
2. DNS request는 UDP segment에 꼭 들어갈 정도로 작음.

   DNS query는 single UDP request와 server로부터의 single UDP reply로 구성되어 있음.
3. UDP는 not reliable이나, reliability는 application layer에 추가될 수 있음. (Timeout 추가나, resend 작업을 통해)

DNS는 UDP를 53번 port에서 사용함.

그러나 TCP를 사용하는 경우가 있음.

Zone transfer 을 사용해야하는 경우에는 TCP를 사용해야 함.

(Zone Transfer : DNS 서버 간의 요청을 주고 받을 때 사용하는 transfer)

만약에 데이터가 512 bytes를 넘거나, 응답을 못받은 경우 TCP로 함.

[ref]

- https://www.geeksforgeeks.org/why-does-dns-use-udp-and-not-tcp/
- https://support.microsoft.com/en-us/help/556000
- https://www.scaler.com/topics/domain-name-system/

Blocking I/O & Non-Blocking I/O

> I/O 작업은 Kernel level에서만 수행할 수 있다. 따라서, Process, Thread는 커널에게 I/O를 요청해야 한다.

1. **Blocking I/O**

   I/O Blocking 형태의 작업은

   (1) Process(Thread)가 Kernel에게 I/O를 요청하는 함수를 호출

   (2) Kernel이 작업을 완료하면 작업 결과를 반환 받음.

   - 특징
     - I/O 작업이 진행되는 동안 user Process(Thread) 는 자신의 작업을 중단한 채 대기
     - Resource 낭비가 심함
       (I/O 작업이 CPU 자원을 거의 쓰지 않으므로)

   `여러 Client 가 접속하는 서버를 Blocking 방식으로 구현하는 경우` -> I/O 작업을 진행하는 작업을 중지 -> 다른 Client가 진행중인 작업을 중지하면 안되므로, client 별로 별도의 Thread를 생성해야 함 -> 접속자 수가 매우 많아짐

   이로 인해, 많아진 Threads 로 *컨텍스트 스위칭 횟수가 증가함,,, 비효율적인 동작 방식*

2. **Non-Blocking I/O**

   I/O 작업이 진행되는 동안 User Process의 작업을 중단하지 않음.

   - 진행 순서

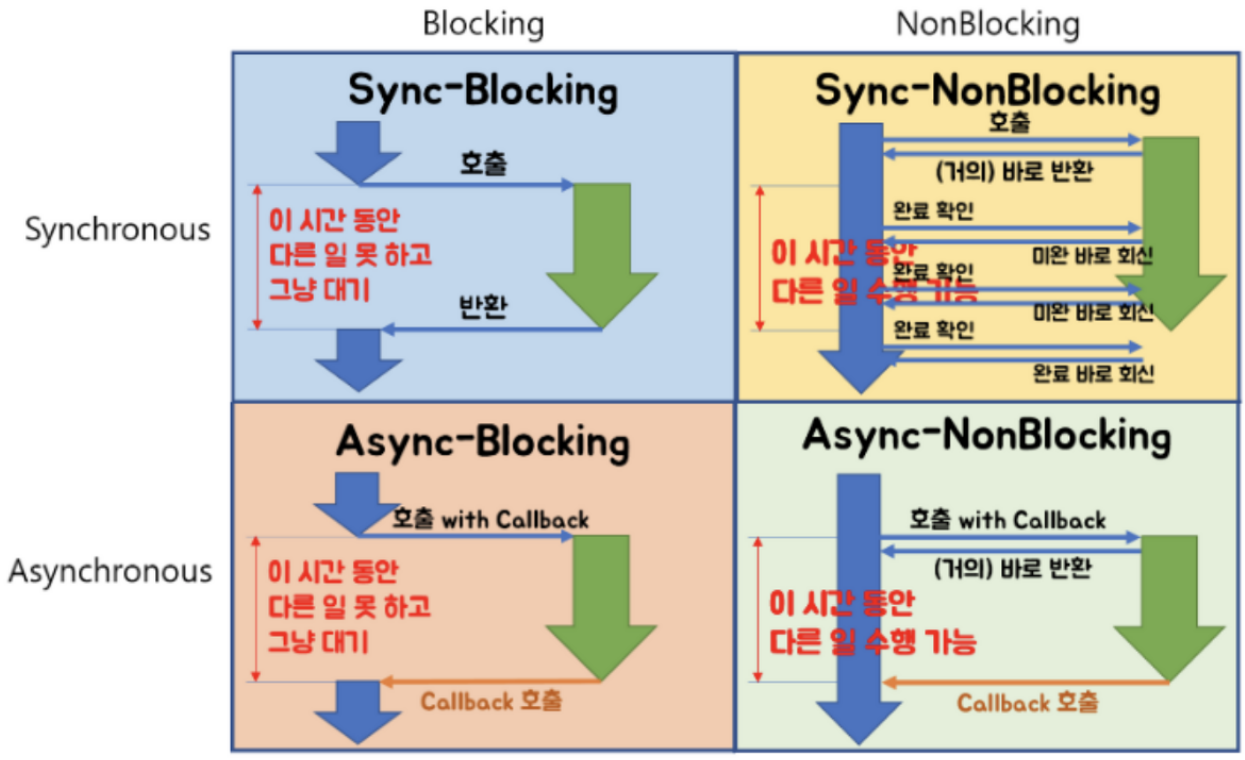
     1. User Process가 recvfrom 함수 호출 (커널에게 해당 Socket으로부터 data를 받고 싶다고 요청함)
     2. Kernel은 이 요청에 대해서, 곧바로 recvBuffer를 채워서 보내지 못하므로, "EWOULDBLOCK"을 return함.
     3. Blocking 방식과 달리, User Process는 다른 작업을 진행할 수 있음.
     4. recvBuffer에 user가 받을 수 있는 데이터가 있는 경우, Buffer로부터 데이터를 복사하여 받아옴.

        > 이때, recvBuffer는 Kernel이 가지고 있는 메모리에 적재되어 있으므로, Memory간 복사로 인해, I/O보다 훨씬 빠른 속도로 data를 받아올 수 있음.

     5. recvfrom 함수는 빠른 속도로 data를 복사한 후, 복사한 data의 길이와 함께 반환함.

[Network] Blocking/Non-blocking & Synchronous/Asynchronous

> 동기/비동기는 우리가 일상 생활에서 많이 들을 수 있는 말이다.
>
> Blocking과 Synchronous, 그리고 Non-blocking과 Asysnchronous를 서로 같은 개념이라고 착각하기 쉽다.
>
> 각자 어떤 의미를 가지는지 간단하게 살펴보자



homoefficio님 블로그에 나온 2대2 매트릭스로 잘 정리된 사진이다. 이 사진만 보고 모두 이해가 된다면, 차이점에 대해 잘 알고 있는 것이다.

Blocking/Non-blocking

블럭/논블럭은 간단히 말해서 호출된 함수가 호출한 함수에게 제어권을 건네주는 유무의 차이라고 볼 수 있다.

함수 A, B가 있고, A 안에서 B를 호출했다고 가정해보자. 이때 호출한 함수는 A고, 호출된 함수는 B가 된다. 현재 B가 호출되면서 B는 자신의 일을 진행해야 한다. (제어권이 B에게 주어진 상황)

- **Blocking** : 함수 B는 내 할 일을 다 마칠 때까지 제어권을 가지고 있는다. A는 B가 다 마칠 때까지 기다려야 한다.
- **Non-blocking** : 함수 B는 할 일을 마치지 않았어도 A에게 제어권을 바로 넘겨준다. A는 B를 기다리면서도 다른 일을 진행할 수 있다.

즉, 호출된 함수에서 일을 시작할 때 바로 제어권을 리턴해주느냐, 할 일을 마치고 리턴해주느냐에 따라 블럭과 논블럭으로 나누어진다고 볼 수 있다.

Synchronous/Asynchronous

동기/비동기는 일을 수행 중인 동시성에 주목하자

아까처럼 함수 A와 B라고 똑같이 생각했을 때, B의 수행 결과나 종료 상태를 A가 신경쓰고 있는 유무의 차이라고 생각하면 된다.

- **Synchronous** : 함수 A는 함수 B가 일을 하는 중에 기다리면서, 현재 상태가 어떤지 계속 체크한다.
- **Asynchronous** : 함수 B의 수행 상태를 B 혼자 직접 신경쓰면서 처리한다. (Callback)

즉, 호출된 함수(B)를 호출한 함수(A)가 신경쓰는지, 호출된 함수(B) 스스로 신경쓰는지를 동기/비동기라고 생각하면 된다.

비동기는 호출시 Callback을 전달하여 작업의 완료 여부를 호출한 함수에게 답하게 된다. (Callback이 오기 전까지 호출한 함수는 신경쓰지 않고 다른 일을 할 수 있음)

위 그림처럼 총 4가지의 경우가 나올 수 있다. 이걸 좀 더 이해하기 쉽게 Case 별로 예시를 통해 보면서 이해하고 넘어가보자

```
상황 : 치킨집에 직접 치킨을 사러감
```

1) Blocking & Synchronous

```
나 : 사장님 치킨 한마리만 포장해주세요
사장님 : 네 금방되니까 잠시만요!
나 : 넹
-- 사장님 치킨 튀기는 중--
나 : (아 언제 되지?..궁금한데 그냥 멀뚱히 서서 치킨 튀기는거 보면서 기다림)
```

2) Blocking & Asynchronous

```
나 : 사장님 치킨 한마리만 포장해주세요
사장님 : 네 금방되니까 잠시만요!
나 : 앗 넹
-- 사장님 치킨 튀기는 중--
나 : (언제 되는지 안 궁금함, 잠시만이래서 다 될때까지 서서 붙잡힌 상황)
```

3) Non-blocking & Synchronous

```
나 : 사장님 치킨 한마리만 포장해주세요
사장님 : 네~ 주문 밀려서 시간 좀 걸리니까 볼일 보시다 오세요
나 : 넹
-- 사장님 치킨 튀기는 중--
(5분뒤) 나 : 제꺼 나왔나요?
사장님 : 아직이요
(10분뒤) 나 : 제꺼 나왔나요?
사장님 : 아직이요ㅠ
(15분뒤) 나 : 제꺼 나왔나요?
사장님 : 아직이요ㅠㅠ
```

4) Non-blocking & Asynchronous

```
나 : 사장님 치킨 한마리만 포장해주세요
사장님 : 네~ 주문 밀려서 시간 좀 걸리니까 볼일 보시다 오세요
나 : 넹
-- 사장님 치킨 튀기는 중--
나 : (앉아서 다른 일 하는 중)
...
사장님 : 치킨 나왔습니다
나 : 잘먹겠습니다~
```

[참고 사항]

- 링크
- 링크

대칭키 & 공개키

대칭키(Symmetric Key)

> 암호화와 복호화에 같은 암호키(대칭키)를 사용하는 알고리즘

동일한 키를 주고받기 때문에, 매우 빠르다는 장점이 있음

but, 대칭키 전달과정에서 해킹 위험에 노출

공개키(Public Key)/비대칭키(Asymmetric Key)

> 암호화와 복호화에 사용하는 암호키를 분리한 알고리즘

대칭키의 키 분배 문제를 해결하기 위해 고안됨.(대칭키일 때는 송수신자 간만 키를 알아야하기 때문에 분배가 복잡하고 어렵지만 공개키와 비밀키로 분리할 경우, 남들이 알아도 되는 공개키만 공개하면 되므로)

자신이 가지고 있는 고유한 암호키(비밀키)로만 복호화할 수 있는 암호키(공개키)를 대중에 공개함

공개키 암호화 방식 진행 과정

1. A가 웹 상에 공개된 'B의 공개키'를 이용해 평문을 암호화하여 B에게 보냄
2. B는 자신의 비밀키로 복호화한 평문을 확인, A의 공개키로 응답을 암호화하여 A에개 보냄
3. A는 자신의 비밀키로 암호화된 응답문을 복호화함

하지만 이 방식은 Confidentiallity만 보장해줄 뿐, Integrity나 Authenticity는 보장해주지 못함

-> 이는 MAC(Message Authentication Code)나 전자 서명(Digital Signature)으로 해결 (MAC은 공개키 방식이 아니라 대칭키 방식임을 유의! T=MAC(K,M) 형식)

대칭키에 비해 암호화 복호화가 매우 복잡함

(암호화하는 키가 복호화하는 키가 서로 다르기 때문)

대칭키와 공개키 암호화 방식을 적절히 혼합해보면? (하이브리드 방식)

> SSL 탄생의 시초가 됨

```
1. A가 B의 공개키로 암호화 통신에 사용할 대칭키를 암호화하고 B에게 보냄
2. B는 암호문을 받고, 자신의 비밀키로 복호화함
3. B는 A로부터 얻은 대칭키로 A에게 보낼 평문을 암호화하여 A에게 보냄
```

```
4. A는 자신의 대칭키로 암호문을 복호화함
5. 앞으로 이 대칭키로 암호화를 통신함
```

즉, 대칭키를 주고받을 때만 공개키 암호화 방식을 사용하고 이후에는 계속 대칭키 암호화 방식으로 통신하는 것!

로드 밸런싱(Load Balancing)

> 둘 이상의 CPU or 저장장치와 같은 컴퓨터 자원들에게 작업을 나누는 것

요즘 시대에는 웹사이트에 접속하는 인원이 급격히 늘어나게 되었다.

따라서 이 사람들에 대해 모든 트래픽을 감당하기엔 1대의 서버로는 부족하다. 대응 방안으로 하드웨어의 성능을 올리거나(Scale-up) 여러대의 서버가 나눠서 일하도록 만드는 것(Scale-out)이 있다. 하드웨어 향상 비용이 더욱 비싸기도 하고, 서버가 여러대면 무중단 서비스를 제공하는 환경 구성이 용이하므로 Scale-out이 효과적이다. 이때 여러 서버에게 균등하게 트래픽을 분산시켜주는 것이 바로 **로드 밸런싱**이다.

로드 밸런싱은 분산식 웹 서비스로, 여러 서버에 부하(Load)를 나누어주는 역할을 한다. Load Balancer를 클라이언트와 서버 사이에 두고, 부하가 일어나지 않도록 여러 서버에 분산시켜주는 방식이다. 서비스를 운영하는 사이트의 규모에 따라 웹 서버를 추가로 증설하면서 로드 밸런서로 관리해주면 웹 서버의 부하를 해결할 수 있다.

로드 밸런서가 서버를 선택하는 방식

- 라운드 로빈(Round Robin) : CPU 스케줄링의 라운드 로빈 방식 활용
- Least Connections : 연결 개수가 가장 적은 서버 선택 (트래픽으로 인해 세션이 길어지는 경우 권장)
- Source : 사용자 IP를 해싱하여 분배 (특정 사용자가 항상 같은 서버로 연결되는 것 보장)

로드 밸런서 장애 대비

서버를 분배하는 로드 밸런서에 문제가 생길 수 있기 때문에 로드 밸런서를 이중화하여 대비한다.

> Active 상태와 Passive 상태

[참고자료]

- 링크
- 링크