컴퓨터 네트워크 HW_2

TCP 기반 소켓프로그래밍 작성

20142772 최승호

TCP 소켓프로그래밍

- 기본 설정

- 각 요청에 대한 클라이언트 / 서버 코드와 응답

- 와이어샤크 스크린샷 및 분석
  전체적인 프로세스로 .100 <-> .102 와 요청 응답 한 것입니다.

180957	1139.223116	192.168.219.102	192.168.219.100	HTTP	253 HEAD / HTTP/1.1
180958	1139.225756	192.168.219.100	192.168.219.102	HTTP	154 HTTP/1.1 200 OK
180966	1139.349769	192.168.219.102	192.168.219.100	HTTP	252 GET / HTTP/1.1
180967	1139.352470	192.168.219.100	192.168.219.102	HTTP	494 HTTP/1.1 200 OK (text/html)Continuation
180975	1139.456678	192.168.219.102	192.168.219.100	HTTP	258 GET /random HTTP/1.1
180976	1139.459046	192.168.219.100	192.168.219.102	HTTP	500 HTTP/1.1 404 Not Found (text/html)Continuation
180986	1139.530497	192.168.219.100	192.168.219.102	HTTP	54 HTTP/1.1 400 Bad Request

기본 설정

- 예제 페이지 index.html 404error.html 파일도 첨부하였습니다.
- 서버 포트는 12000으로 하였고, 집 와이파이 주소를 default 값으로 넣었습니다.
- 테스트 하실려면 서버/클라이언트 두 군데 192.168.218.100 주소를 변경하시고 진행하시면 됩니다.

1. 정상적인 HEAD 요청

```
# 1. 정상적인 head 요청
request_message  = 'HEAD / HTTP/1.1\r\n'
request_message += 'Host: 192.168.219.100:12000\r\n'
request_message += 'User-Agent: Mozilla/5.0 (Windows NT 10.0;
request_message += 'Connection: Keep-Alive\n\n'
create_socket_and_send_message(request_message)
```

```
Hypertext Transfer Protocol
  HEAD / HTTP/1.1\r\n
  Host: 192.168.219.100:12000\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537
  Connection: Keep-Alive\n
```

응답

```
# 헤더 만들어 주기
response_header = 'HTTP/1.1 200 OK\r\n'
response_header += 'Content-Type: text/html\r\n'
response_header += 'Content-Length: {}\r\n'.format(str(len(response_data)))
response_header += 'Date: {}\n\n'.format(datetime.now().strftime('%a, %d %b

# HEAD 요청일 경우
if request_headers[0] == 'HEAD':
    connectionSocket.send(response_header.encode('utf-8'))
```

```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
  Content-Type: text/html\r\n
  Content-Length: 266\r\n
  Date: Wed, 27 May 2020 17:14:22 KST\n
  \n
```

분석

정상적인 HEAD 요청을 수행하였습니다.
정상적인 요청에 따라 200 상태코드를 응답하였고, head요청에 따라 index.html에 대한 데이터는 주지 않았고 header에는 content-type, content-length, date를 직접 설정하여 응답을 주었습니다.
코드에서 해놓은 설정이 와이어샤크에서 그대로 나온 것을 확인할 수 있었습니다.

2. 정상적인 GET 요청

```
# 2. 정상적인 get 요청
request_message  = 'GET / HTTP/1.1\r\n'
request_message += 'Host: 192.168.219.100:12000\r\n'
request_message += 'User-Agent: Mozilla/5.0 (Windows NT 1
request_message += 'Connection: Keep-Alive\n\n'
create_socket_and_send_message(request_message)
```

```
Hypertext Transfer Protocol
> GET / HTTP/1.1\r\n
  Host: 192.168.219.100:12000\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) Ap
  Connection: Keep-Alive\n
  \n
  [Full request URI: http://192.168.219.100:12000/]
```

응답

```
# GET 인경우
elif request_headers[0] == 'GET':
    response_message = response_header
    response_message += response_data
    connectionSocket.send(response_message.encode('utf-8'))
```

```
Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
    Content-Type: text/html\r\n
  > Content-Length: 266\r\n
    Date: Wed, 27 May 2020 17:14:22 KST\n
    \n
    [HTTP response 1/1]
    [Time since request: 0.001539000 seconds]
    [Request in frame: 36]
    [Request URI: http://192.168.219.100:12000/]
    File Data: 266 bytes
> Line-based text data: text/html (10 lines)
Hypertext Transfer Protocol
  > Data (440 bytes)
```

분석

정상적인 GET 요청을 수행하였습니다.
앞선 HEAD 코드에서 data를 추가하여 응답하였습니다. Data는 index.html파일로 예시 페이지 입니다. 그래서 응답의 와이어 샤크 캡처화면을 보면 아래에 data부분이 추가되어 정상적인 200 상태코드로 응답하였습니다.

3. 잘못된 주소 요청

```
# 3. 잘못된 페이지 get 요청
request_message  = 'GET /random HTTP/1.1\r\n' # 없는 페이지 요청
request_message += 'Host: 192.168.219.100:12000\r\n'
request_message += 'User-Agent: Mozilla/5.0 (Windows NT 10.0; Wi
request_message += 'Connection: Keep-Alive\n\n'
create_socket_and_send_message(request_message)
```

```
Hypertext Transfer Protocol
  GET /random HTTP/1.1\r\n
  Host: 192.168.219.100:12000\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) Ap
  Connection: Keep-Alive\n
  \n
  [Full request URI: http://192.168.219.100:12000/random]
```

응답

```
# 주소가 틀린 경우
else:
    # 404error.html 페이지 불러오기
    file_name = './404error.html'
    f = open(file_name)
    response_data = f.read()
    response_header = 'HTTP/1.1 404 Not Found\r\n'
    response_header += 'Content-Type: text/html\r\n'
    response_header += 'Content-Length: {}\r\n'.format(str(l
    response_header += 'Date: {}\n\n'.format(datetime.now().
    response_message = response_header
    response_message += response_data
    connectionSocket.send(response_message.encode('utf-8'))
```

```
Hypertext Transfer Protocol
  HTTP/1.1 404 Not Found\r\n
  Content-Type: text/html\r\n
  Content-Length: 279\r\n
  Date: Wed, 27 May 2020 17:14:22 KST\n
  \n
  [HTTP response 1/1]
  [Time since request: 0.001604000 seconds]
  [Request in frame: 45]
  [Request URI: http://192.168.219.100:12000/random]
  File Data: 279 bytes
Line-based text data: text/html (10 lines)
Hypertext Transfer Protocol
  Data (446 bytes)
```

분석

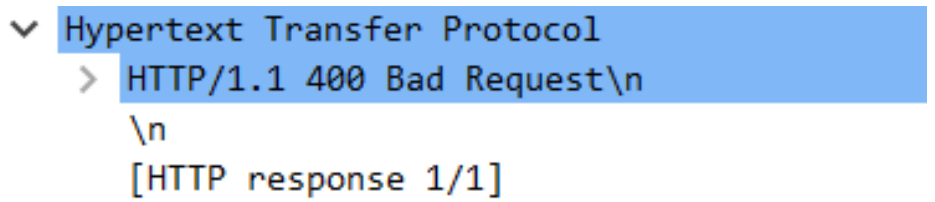
잘못된 경로로 GET 요청을 수행하였습니다.
Random이라는 없는 페이지를 요청하여 로컬에 만들어 놓은 404error.html과 404 상태코드로 응답하였습니다.
Get요청과 유사해 보이나 실제로는 서버에 index.html을 제외한 여러 페이지가 있으므로 없는 페이지를 보여줄 때는 예시처럼 404 상태 코드와 404 에러 페이지를 응답하여야 합니다.

4. 잘못된 프로토콜
OR 잘못된 요청 메소드

```
# 4. 프로토콜이나 요청 메소드 잘못 요청
request_message  = 'POST / HTTTTP/1.1\r\n'
request_message += 'Host: 192.168.219.100:12000\r\n'
request_message += 'User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
request_message += 'Connection: Keep-Alive\n\n'
create_socket_and_send_message(request_message)
```

응답

```
# 프로토콜이나 요청이 틀릴 경우
else:
    response_header = 'HTTP/1.1 400 Bad Request\n\n'
    connectionSocket.send(response_header.encode('utf-8'))
```



분석

잘못된 프로토콜이나 잘못된 요청을 했을 경우 400 bad request를 응답하였습니다. 이번에는 따로 페이지를 만들지 않고 필수적인 header만 설정하여 응답하였습니다. Get요청만 응답이 가능하여 post/delete/update 등을 요청하거나 htttp/1.0같이 잘못된 프로토콜명과 버전을 사용하면 400 상태코드를 응답하는 것으로 하였습니다. 요청 부분은 잘못된 요청으로 와이어샤크에 잡히지 않았습니다.

5. 서버 에러

```
try:
    # 수신
    message = connectionSocket.recv(65535).decode()
    request_headers = message.split()
    # print(int(request_headers)) # 500 에러 서버 띄우기
```

```
Hypertext Transfer Protocol
  HEAD / HTTP/1.1\r\n
  Host: 192.168.219.100:12000\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) A
  Connection: Keep-Alive\n
  \n
  [Full request URI: http://192.168.219.100:12000/]
```

응답

```
# 아예 서버에러 500
# 로직이 잘못 되거나 통신이 잘못 됐을 경우
except:
    response_header = 'HTTP/1.1 500 Internal Server Error\n\n'
    connectionSocket.send(response_header.encode('utf-8'))
    connectionSocket.close()
    sys.exit()
```

```
Hypertext Transfer Protocol
  HTTP/1.1 500 Internal Server Error\n
  \n
  [HTTP response 1/1]
```

분석

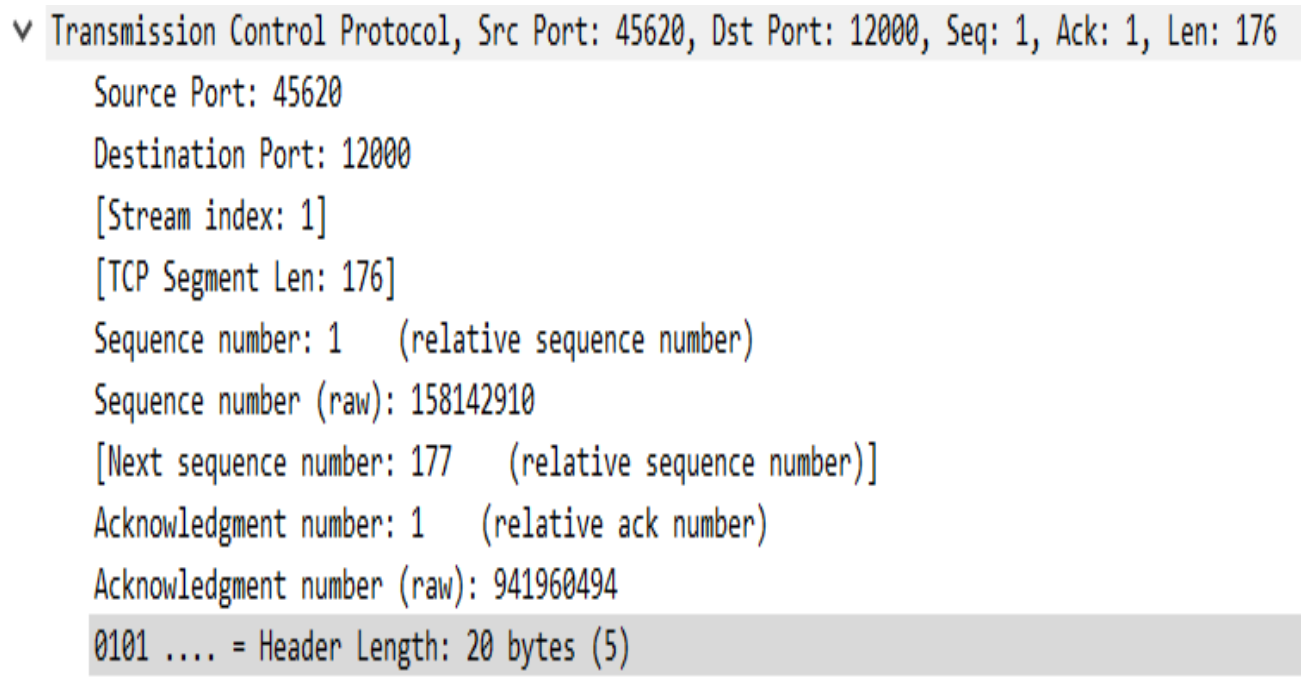
앞서 1번의 요청 코드를 그대로 사용하였습니다. 차이점은 서버 코드에서 왼쪽 첫번째 캡처사진에 주석부분을 해제해주어 강제로 서버에서 에러가 나오게 하였습니다. 400, 404와 다르게 내부적인 서버로직이 잘못됐거나 통신상 에러일 경우만 500 상태코드를 반환하도록 하였습니다.

TCP 소켓프로그래밍 헤더 분석

- 첫번째 HEAD요청 / 응답으로 TCP 헤더 분석을 진행하였습니다.
- 요청에 따른 TCP 차이는 없어 한가지 케이스만 분석하겠습니다.
- 요청 – 3way handshake & close connection – 응답 순으로 설명하겠습니다.

TCP 소켓프로그래밍 헤더 분석

TCP protocol 요청



분석 1

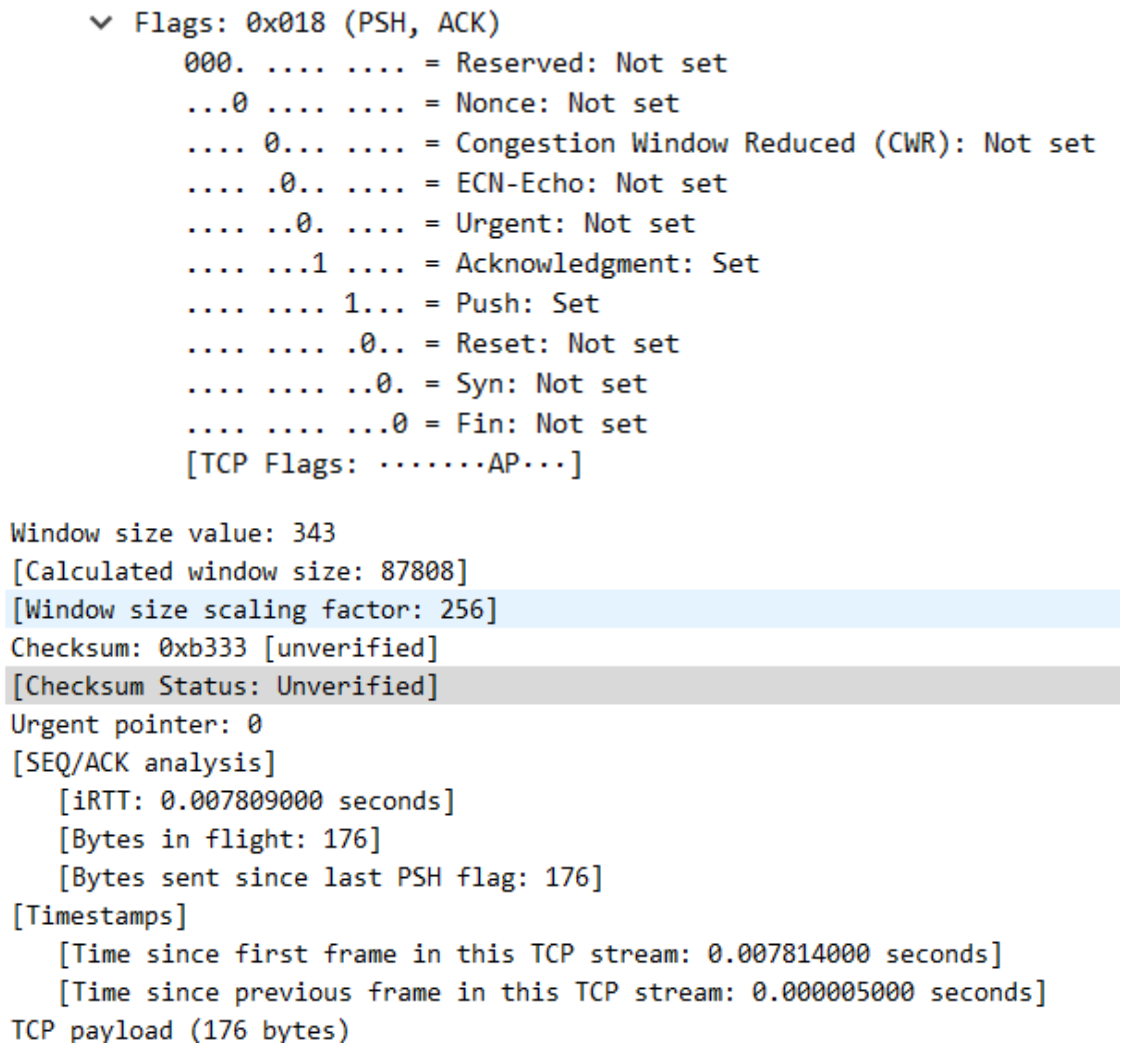
첫번째로는 source port: 45620으로 제가 요청한 주소의 port번호이고, destination port는 12000으로 server socket에서 임의로 설정해준 port번호가 나왔습니다.

Sequence number는 1로 특정 tcp 세그먼트를 식별할 수 있습니다. 통신 스트림 일부가 분실되면 확인을 위해 수신자를 사용가능하게 합니다.
이 때 next seq num을 tcp segment크기만큼 증가한 177로 변경해 줍니다.

Ack는 1로 다음번에 기대되는 순차번호로, 응답코드에서 확인할 예정입니다.

Header length는 20으로 80byte의 TCP헤더의 길이를 뜻합니다. 4byte(32bit)씩 증가하고 헤더의 길이가 가변적이므로 필요한 필드입니다.

분석 2



Reserved는 현재 사용하지 않는 필드입니다.
플래그 필드로 앞에 2bit는 사용하지 않고 6bit를 파헤쳐보면
Urgent: 긴급 포인터(잘 사용 x, 1일 경우 특정 패킷의 데이터를 우선처리 하고위해)
Ack: 확인응답패킷 1이면 확인 응답이라는 것이다
PSH: 네트워크에서 버퍼링 우회와 데이터 즉시 통과(잘 사용 x)
(버퍼에 유지되면 안된다는 것을 표시)
RST: 연결 닫기(tcp 연결 종료)
SYN: 동기화 순차번호(1이면 핸드셰이크의 syn단계를 의미)
FIN: 트랜잭션 종료

Window size는 TCP 수신 버퍼의 바이트 크기를 의미하고 최대값은 65,535입니다. 0이면 공간이 없다는 것입니다.

Checksum은 1의 보수를 취해 에러를 체크하는 방법으로
checksum field는 TCP 헤더와 데이터의 내용뿐만 아니라 IP 헤더로 부터 파생된 의사 헤더에 대한 내용을 수행합니다.

나머지는 긴급포인터 필드와 요청에 걸린 시간을 의미합니다.

TCP 소켓프로그래밍 헤더 분석

TCP 3-way handshake & close connection

클라이언트: 192.168.219.103
서버: 192.168.219.100

Time	Source	Destination	Protocol	Length	Info
0.864831	192.168.219.103	192.168.219.100	TCP	74	45620 → 12000 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=2
0.864909	192.168.219.100	192.168.219.103	TCP	66	12000 → 45620 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 S
0.872640	192.168.219.103	192.168.219.100	TCP	54	45620 → 12000 [ACK] Seq=1 Ack=1 Win=87808 Len=0
0.872645	192.168.219.103	192.168.219.100	HTTP	230	HEAD / HTTP/1.1
0.873587	192.168.219.100	192.168.219.103	HTTP	154	HTTP/1.1 200 OK
0.873658	192.168.219.100	192.168.219.103	TCP	54	12000 → 45620 [FIN, ACK] Seq=101 Ack=177 Win=131072 Len=0
0.879569	192.168.219.103	192.168.219.100	TCP	54	45620 → 12000 [ACK] Seq=177 Ack=101 Win=87808 Len=0
0.928955	192.168.219.103	192.168.219.100	TCP	54	45620 → 12000 [ACK] Seq=177 Ack=102 Win=87808 Len=0
11.871320	192.168.219.103	192.168.219.100	TCP	54	45620 → 12000 [FIN, ACK] Seq=177 Ack=102 Win=87808 Len=0
11.871372	192.168.219.100	192.168.219.103	TCP	54	12000 → 45620 [ACK] Seq=102 Ack=178 Win=131072 Len=0

분석

1. 앞서 말한 플래그 비트 중 syn비트에 1을 넣고 seq=0 을 mss와 window scale까지 정해서 서버로 보냈습니다.
- Mss는 maximum segment size로 mtu – ip/tcp 헤더 크기 입니다.

2. 서버측에서 syn비트=1, seq=0 ack=클라이언트seq+1 로 1로 응답하였고

3. ack=서버seq +1로 1이되었고, ackbit를 1로 설정하여 메세지를 보내 connection을 생성하였습니다.

4. 그 후 http 요청/응답 (앞서 분석한 요청 tcp 헤더 분석과 같습니다.) 을 정상적으로 받고 마지막에 finbit=1로 설정하여 서버에서 먼저 seq=101 ack=177로 요청하여 , 클라이언트에서 seq=177, ack=102 응답(우아한 종료라고 칭함)하여 tcp connection이 정상 종료되었음을 알 수 있었습니다.

TCP 소켓프로그래밍 헤더 분석

TCP protocol 응답

```
Transmission Control Protocol, Src Port: 12000, Dst Port: 45620, Seq: 1, Ack: 177, Len: 100
    Source Port: 12000
    Destination Port: 45620
    [Stream index: 1]
    [TCP Segment Len: 100]
    Sequence number: 1    (relative sequence number)
    Sequence number (raw): 941960494
    [Next sequence number: 101    (relative sequence number)]
    Acknowledgment number: 177    (relative ack number)
    Acknowledgment number (raw): 158143086
    0101 .... = Header Length: 20 bytes (5)
  Flags: 0x018 (PSH, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 1... = Push: Set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·······AP···]
```

분석 1

Source랑 dest 포트가 반대로 된 것을 확인할 수 있습니다.

Sequence number는 1로 앞서 설명한 요청의 ack num과 같습니다. handshake진행 후 첫 seq입니다.

Ack는 177로 다음번에 기대되는 순차번호입니다.
= 다음번 요청측의 seqnum
= 이전 요청측의 next seq num

플래그와 header length의 설명은 동일하니 넘어가겠습니다.

TCP 소켓프로그래밍 헤더 분석

```
Window size value: 512
[Calculated window size: 131072]
[Window size scaling factor: 256]
Checksum: 0xb6c8 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
[SEQ/ACK analysis]
   [This is an ACK to the segment in frame: 7]
   [The RTT to ACK the segment was: 0.000942000 seconds]
   [iRTT: 0.007809000 seconds]
   [Bytes in flight: 100]
   [Bytes sent since last PSH flag: 100]
[Timestamps]
   [Time since first frame in this TCP stream: 0.008756000 seconds]
   [Time since previous frame in this TCP stream: 0.000942000 seconds]
TCP payload (100 bytes)
```

분석 2 및 결론

앞서 요청에서 말한 설명들과 동일합니다.

http 프로토콜을 직접 짜보며 어떤 식으로 요청들을 parsing해야 할지 고민해보는 코딩이였습니다.

직접 소켓프로그래밍을 구현하여 tcp와 http 프로토콜을 시간순으로 나열해보니 어떤식으로 연결 되고 어떤식으로 데이터를 보내고 종료되는지 확인하는 좋은 경험이였습니다.

- Tcp 3-handshake를 통한 connection설정
- 서버와 클라이언트의 seq num, ack 번호 규칙
- 플래그 비트 syn ack fin의 이해
- Close connection 하는 방법

등을 이해했습니다.

이제 조금 tcp의 감이 잡히는 것 같습니다!