

컴퓨터네트워크 Task2

학과: 소프트웨어학과

학번: 2022125057

이름: 조재현

이메일: [cjh030808@kau.kr](mailto:cjh030808@kau.kr)

**1. 개요**

웹 서버와 브라우저 간 통신에서 발생하는 패킷을 Wireshark로 캡처하고, Application Layer(HTTP)와 Transport Layer(TCP)의 주요 특징을 분석한다. 이를 통해 웹 통신 흐름과 계층 간 연동 과정을 이해하는 것이 목적이다.

분석대상: <http://kau.ac.kr>

**2. 실습 과정**

Wireshark 실행 후 이더넷 환경에서 캡쳐 시작

브라우저에서 <http://kau.ac.kr> 접속 -> /index/main.php 요청

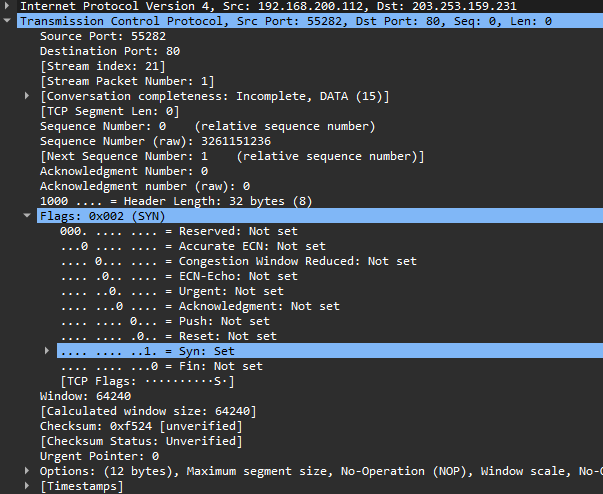
필터를 통해 해당 서버 ip인 (ip.addr == 203.253.159.231)만 분석



**3. 패킷 분석**

**TCP 3-way handshake**

SYN (클라이언트 -> 서버)

****

클라이언트: 192.168.200.112:55282

서버: 203.253.159.231:80

Flags: SYN=1

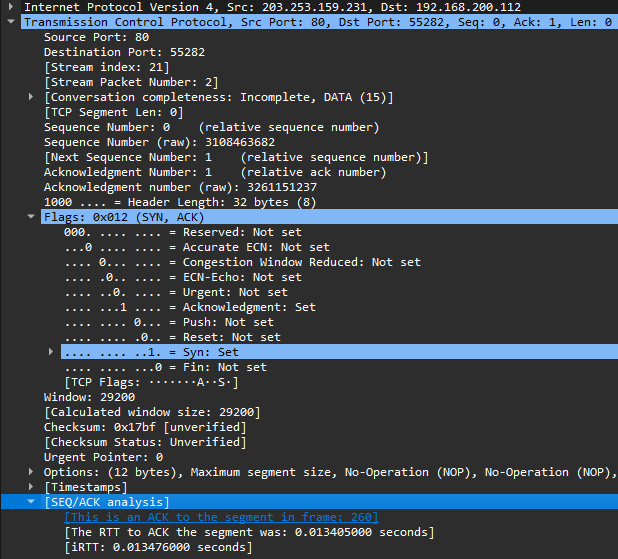
Initiative Sequence Num: 3261151236 (무작위 생성)

Window Size: 64240 (rwnd)

Checksum: 0xf524

이 패킷은 클라이언트가 서버로 TCP 연결을 요청하는 세그먼트임.

SYN+ACK (서버 → 클라이언트)



Flags: SYN=1, ACK=1

Acknowledgment Number: 3261151237 (클라이언트의 Initiative Sequence Num+1)

Initiative Sequence Num(서버): 3108463682 (무작위 생성)

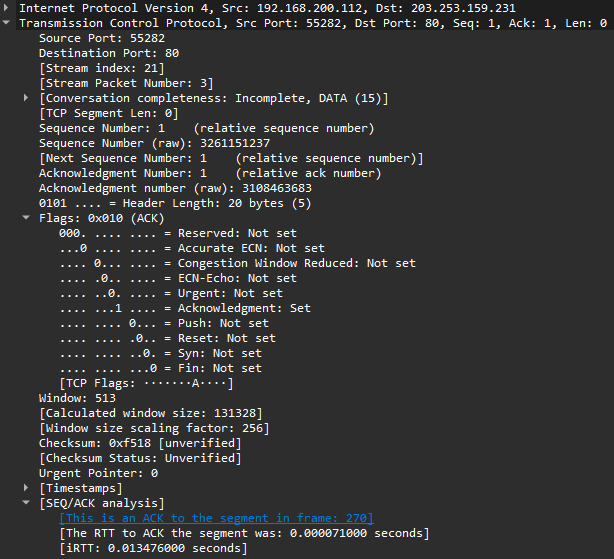
Window Size: 29200 (rwnd)

Checksum: 0x17bf

RTT: 약 13.47 ms

서버는 클라이언트의 SYN에 응답하며, 자신의 연결 요청을 함께 보냄.

ACK(클라이언트 -> 서버)



Flags: ACK=1

Sequence Number: 3261151237 (클라이언트 ISN+1)

Acknowledgment Number: 3108463683 (서버 ISN+1)

Window Size (raw): 513

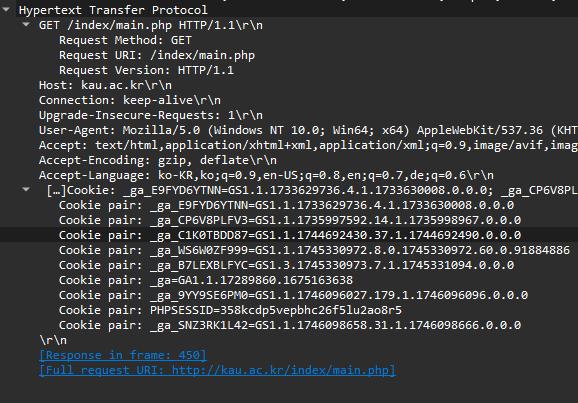
Window Scaling: 256 (최종 rwnd=513×256=131328)

Checksum: 0xf518

클라이언트는 서버의 SYN+ACK에 응답하며 3-way Handshake를 완료함.

실제로 ISN에 +1 씩의 seq # 과 ack # 을 가지는 것을 확인하여 TCP의 seq, ack number 특성을 확인할 수 있었음

**HTTP Request**



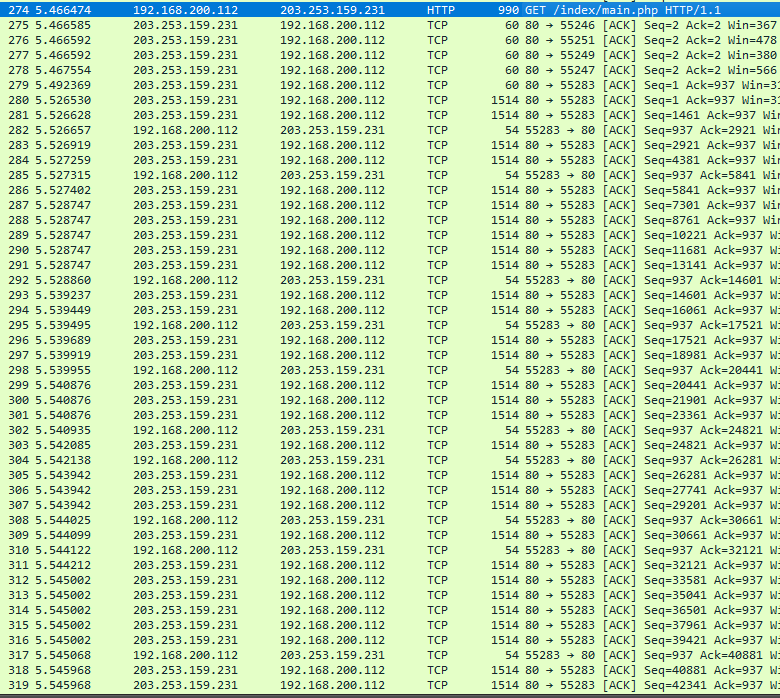
GET메서드로 /index/main.php요청

Connection: keep-alive: Persistent HTTP 요청

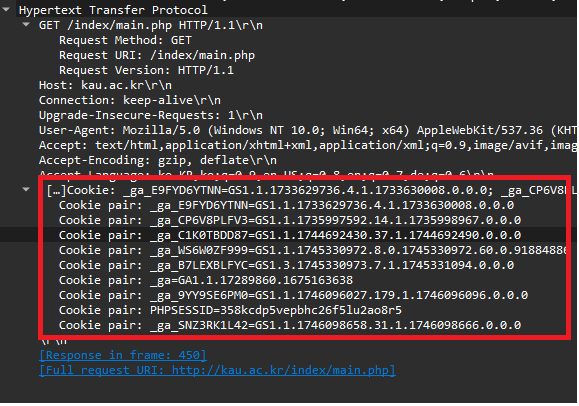
Cookie: 세션 유지 및 추적

Persistent Connection은 단일 TCP 연결을 유지하여 여러 HTTP 요청과 응답을 순차적으로 처리할 수 있게 하여, 매번 새로운 연결을 설정하는 오버헤드를 줄이는 방식임. 이로 인해 연결 설정 시간과 리소스 소모가 감소되고, 전반적인 응답 시간이 개선될 수 있음. (persistent HTTP: RTT + (RTT + file transmission time) \* N, Non-persistent HTTP: (2RTT + file transmission time)\*N)

HTTP는 기본적으로 무상태(stateless) 프로토콜이지만, Cookie를 사용하여 세션 상태를 유지함. 서버는 클라이언트에 고유한 쿠키 값을 전송하며, 클라이언트는 이 값을 저장한 후 이후 요청 시 자동으로 함께 전송함. 이러한 쿠키 값은 사용자별 세션 식별자로 활용되어, 서비스의 지속성을 보전해주는 기능을 함.

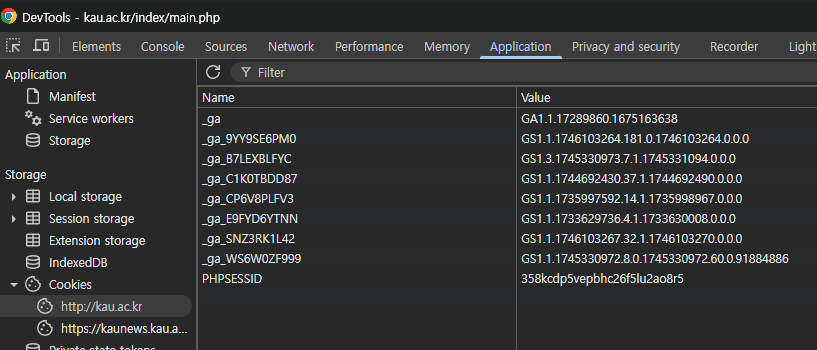
****Connection: keep-alive요청 후, 동일 TCP 연결(포트 55283)에서 다수의 응답 세그먼트가 전송되는 것으로 미루어보아 persistent HTTP로 동작하는 것을 알 수 있음.

284번 패킷에서 Seq# = 4381의 데이터를 보냈는데 Ack# = 5841을 보낸 것을 미루어보아 누적 ACK이 확인됨. 그리고 데이터를 받을 때마다 ACK를 보내는 것이 아닌 일정 시간 지연 후 ACK를 전송하는 delayed ack을 사용하고 있는 것도 확인할 수 있음.

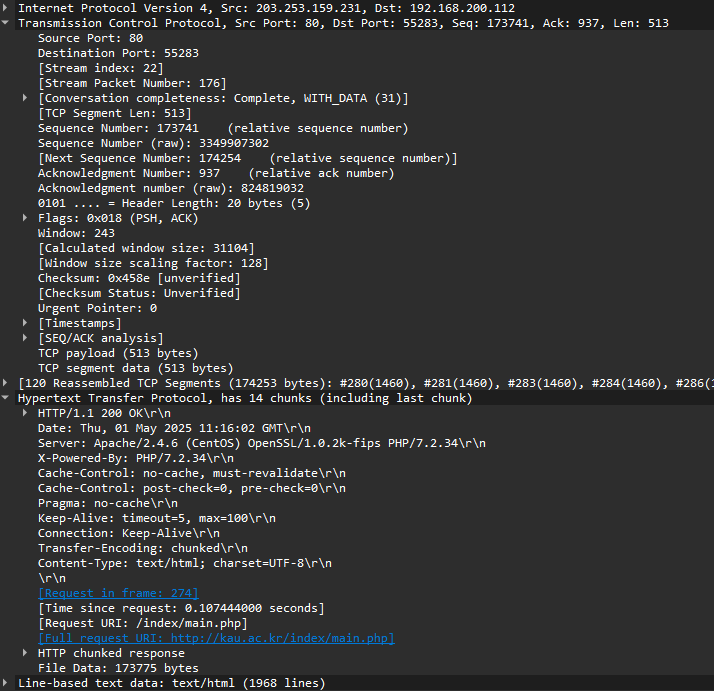


Cookie헤더에 Google Analytics(\_ga, \_gid) 및 PHPSESSID 포함

개발자도구에서 Application에 들어가보면 해당 쿠키가 잘 저장되어 있는 것을 확인할 수 있었음.



**HTTP response**



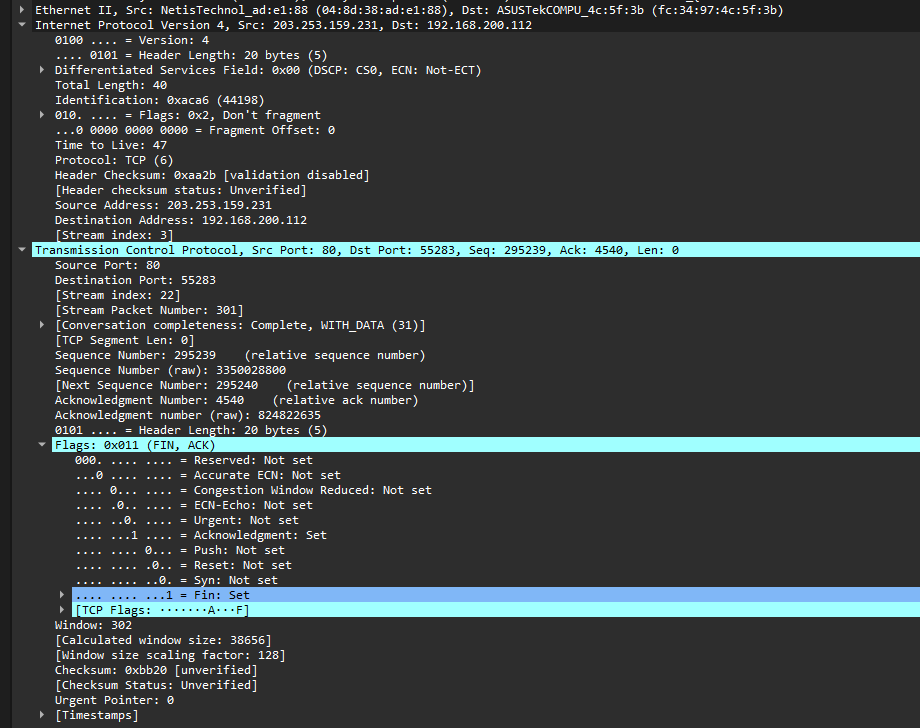
Status: 200 OK

헤더: Pragma: no-cache (캐시 무시 요청)

콘텐츠: 서버에서 새로운 HTML 데이터 전송

캐시 유효 시 304 Not Modified 응답 가능, 혹은 오브젝트를 찾지 못했다면 404 와 같은 status code로 알 수 있을 것.

checksum을 계산할 때, 데이터는 16비트 단위로 처리되야 함. checksum을 구할 때 두 개의 16비트 정수를 더해서 구하는 방식이기 때문. 해당 File Data가 173775 bytes 이므로 checksum인 0x458e 를 도출하기 위해 00000000, 즉 1byte의 padding으로 16비트를 만들어주었을 것임.

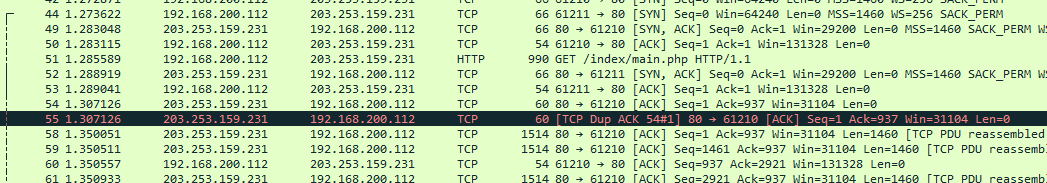
**TCP 연결 종료**

Flag: FIN: 1, ACK 1

FIN과 ACK 가 각각 이뤄져야만 하는 줄 알았지만 만약 더 보낼 내용이 없다면 같이 보낼 수도 있다고 함.

Wireshark에서는 두 개의 플래그를 한 세그먼트에 결합하여 전송하는 최적화가 표시됨.

**Duplicate Ack**

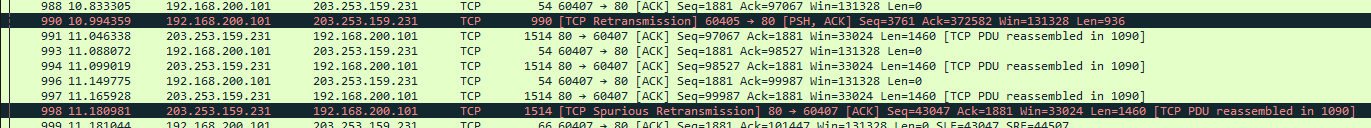
****duplicate ack 이 발생하는 경우도 캡쳐하기 위해 여러 번 재접속하여 dup ack을 캡쳐함.

ACK#= 937인 패킷으로, 중복 ACK(Duplicate ACK)에 해당함. 이 패킷은 Seq#= 1, Len: 0이며, 수신 측이 새로운 데이터를 받지 못한 채 같은 ACK 번호를 연속적으로 전송하고 있는 것을 의미함. 이는 송신 측에 손실된 세그먼트가 존재함을 암시하며, 수신자는 가장 마지막으로 정상 수신한 바이트 다음 번호(937)를 계속해서 알려주는 중임.

이러한 중복 ACK 발생 시, TCP는 손실을 감지하고 혼잡 제어 알고리즘에 따라 행동함. 학습한 두 가지 알고리즘인 TCP Tahoe와 TCP Reno의 동작 차이를 비교하며 각 행동을 설명하겠음.

TCP Tahoe  
중복 ACK가 3번 이상 감지되면 Fast Retransmit를 수행하고, ssthresh는 cwnd의 절반 값을 가지며 cwnd를 1로 줄이며 다시 Slow Start상태로 돌아감.

TCP Reno  
동일하게 중복 ACK 3회를 감지하면 Fast Retransmit를 수행하되, ssthresh는 기존 cwnd 절반으로 떨어트리고 cwnd = ssthresh + 3 상태로 Fast Recovery상태로 진입하여 패킷을 전송한 후 new ack를 받았을 때 다시 cwnd = ssthresh 로 떨어트려서 congestion avoidance 상태로 들어갈 것임.

**TCP retransmission**

TCP retransmission

990번 패킷은 송신자(192.168.200.101)가 이전에 전송했던 데이터를 동일한 seq# 로 다시 전송한 일반적인 TCP 재전송(TCP Retransmission) 사례임.  
Seq: 3761, Len: 936으로, 클라이언트가 보낸 해당 구간의 데이터에 대한 ACK이 오지 않자, timeout이 발생함에 따라 다시 전송된 것으로 보임.

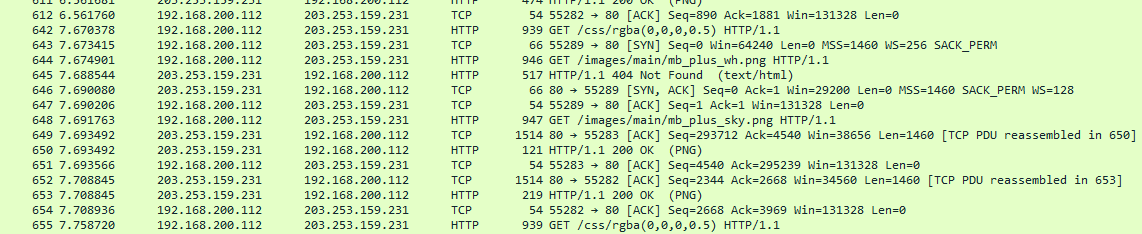
TCP Spurious Retransmission

실제 패킷 손실이 발생하지 않았음에도, 네트워크 지연이나 ACK의 도착 누락 등을 송신자가 손실로 오인하여 재전송을 수행하는 현상임. 이러한 현상은 Premature Timeout과 밀접한 관련이 있다고 볼 수 있음.  
즉, ACK가 도착하기 전에 timeout이 발생하면 송신자는 재전송을 시작하게 되며, 이로 인해 불필요한(spurious) 재전송이 발생함.  
만약 ACK 유실이 아닌 premature timeout이 맞다면 이는 timeout interval이 과도하게 짧게 설정된 상태로, 네트워크 지연에 대한 편차가 큰 상황에서 발생할 가능성이 높음.  
결국 devRTT(편차)가 커지면 timeout interval 또한 증가하게 되어, 이후에는 이런 오차가 보정되는 효과가 나타날 것임을 예측해볼 수 있음.

두 경우(spurious retransmission에서 ACK 유실이 아니라고 가정) 모두 timeout이 발생했기 때문에 혼잡 제어 알고리즘이 3 duplicate ack과는 다른 양상을 띌 것임.

TCP Tahoe, Reno  
ssthresh 값은 기존 cwnd의 절반으로 떨어질 것이고, cwnd 값은 1로 떨어진다. slow start 상태로 바뀌며 ssthresh 값을 만나면 congestion avoidance 상태로 전이됨.

**Multiplexing, Demultiplexing**

Multiplexing

641–643번 패킷 흐름 (192.168.200.112:55282 → 203.253.159.231:80)

644–647번 패킷 흐름 (192.168.200.112:55289 → 203.253.159.231:80)

이렇게 55282번 포트와 55289번 포트 두 개가 동시에, 그리고 순서대로 교차되어 같은 물리적 경로(192.168.200.112→203.253.159.231:80)로 전송되고 있음. 이를 토대로 TCP의 multiplexing을 확인할 수 있었음.

Demultiplexing

Server 입장에서는 들어오는 모든 패킷의 destination port가 80이므로 소켓으로 전달.

응답을 보낼 때는, 54 bytes ACK나 200 OK 메시지를 각각 source port (80) → destination port (55282 vs 55289)로 나눠 보내고 있고, 클라이언트는 각 세그먼트의 destination port(55282 vs 55289)를 보고 분배할 것임.

이처럼 전달된 세그먼트를 destination port기준으로 각각의 애플리케이션 소켓으로 나누어 주는 동작으로 demultiplexing을 확인할 수 있었음.

**4. 결론**

이번 실습을 통해 실제 패킷을 통해 application layer, transport layer이 어떻게 구현되는지 구체적으로 확인할 수 있었음. 특히, 3-way handshake 및 연결 종료 시 플래그 구조, 그리고 중복 ACK와 재전송 시점에서의 TCP 혼잡 제어 알고리즘(Tahoe, Reno)의 차이점을 비교 분석할 수 있었음. 3 duplicate ack, retransmission 등의 가능한 많은 상황을 연출해보며 여러 가지 상황에 대한 패킷 분석과 개념을 활용한 흐름 이해를 할 수 있었음.

또한, HTTP 프로토콜의 동작 원리를 요청-응답 방식, 쿠키 저장 방식 등을 통해 명확히 이해할 수 있었으며, Stateless한 특성을 Cookie를 통해 보완하는 구조도 확인할 수 있었음.

**5. 느낀점**

실제로 Wireshark를 통해 패킷을 분석하며, 단순히 "웹 사이트에 접속한다"는 행위 뒤에 복잡하고 정교한 절차가 숨어 있다는 점이 매우 인상 깊었음. 한편으로는, 중복 ACK나 재전송 패킷과 같은 상황을 정확히 해석하기 위해 더 정밀한 조건 설정과 반복 실험이 필요하다는 점도 느꼈음.

아무리 재접속을 해봐도 Flow Control 상황이 발생하지 않아서 추후에는 이런 부분을 더 명확히 확인해보고 싶다고 생각하게 됨. 또한 네트워크 과목을 대단히 재밌게 수강하고 있는데 이렇게 배운 내용을 실제로 패킷을 하나하나 분석해보고 개념에 대칭시켜보니 더욱 흥미롭고 기억에 남는 것도 많은 것 같다고 생각하게 됨.