클라이언트는 맥북, 서버는 우분투 가상머신(Ubuntu server 16.04 LTS) 에서 실행시킨 결과와 스크린샷을 사용하였음.

1. PauseUntilPressAnyKey()

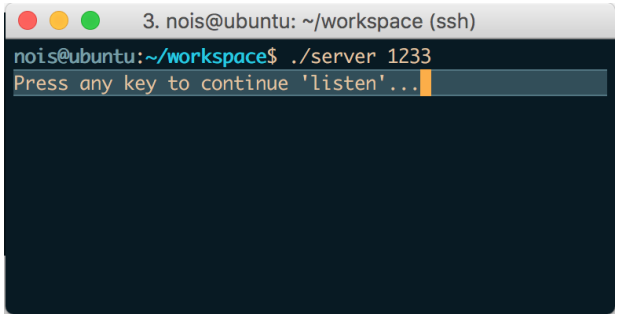
먼저, Client 와 Server 의 CO 관련 함수 전에 키 입력을 받아야 다음 함수가 동작하도록 PauseUntilPressAnyKey()라는 함수를 만들었다. 구현은 다음 그림과 같다.

```
#define ISPAUSE 0

void pauseUntilPressAnyKey(char* message) {
        if(ISPAUSE == 0) return;
        printf("Press any key to continue '%s'...", message);
        getchar();
        puts("");
}
```

비교적 간단한 함수이다. 'Message 에 해당하는 행위를 계속하려면 아무 키나 누르시오.'라는 메시지를 보여주고, 특정 키를 누르기 전까지 진행되지 않는 간단한 함수이다. #define 지시문으로 ISPAUSE 을 '0' 으로 설정해 두었는데, 이는 코드 전체를 바꾸지 않고 ISPAUSE 만 변경하여 Pause 기능을 사용하지 않으려면 0 으로, 사용하려면 0 이 아닌 값으로 변경하여 컴파일 하기 위해 만들어 두었다. 실제 함수 사용은 다음 그림과 같이 이루어진다.

```
pauseUntilPressAnyKey("listen");

if(listen(servSock, MAXPENDING) == -1) {
        DieWithError("listen() error");
}
```

이처럼 "listen"을 argument로 전달하면, 아래 그림과 같은 결과를 확인할 수 있다.

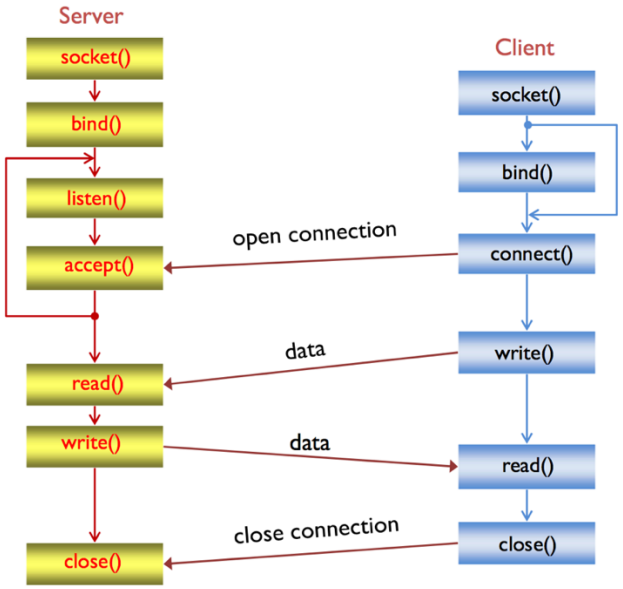


터미널에서 Press any key to continue 'listen'...이라는 메시지와 함께 아무 키나 입력하기 전까지 진행되지 않는 서버의 모습이다.

2. Funcion동작 시 TCP 동작 연관관계 분석 (실습과제 #5)

Pause함수(PauseUntilPressAnyKey)는 client의 경우 connect/send/recv/close의 4곳, Server의 경우 listen/accept/recv/send/close의 5곳의 바로 윗 라인에 각각 배치하였다. 또한, Function 동작

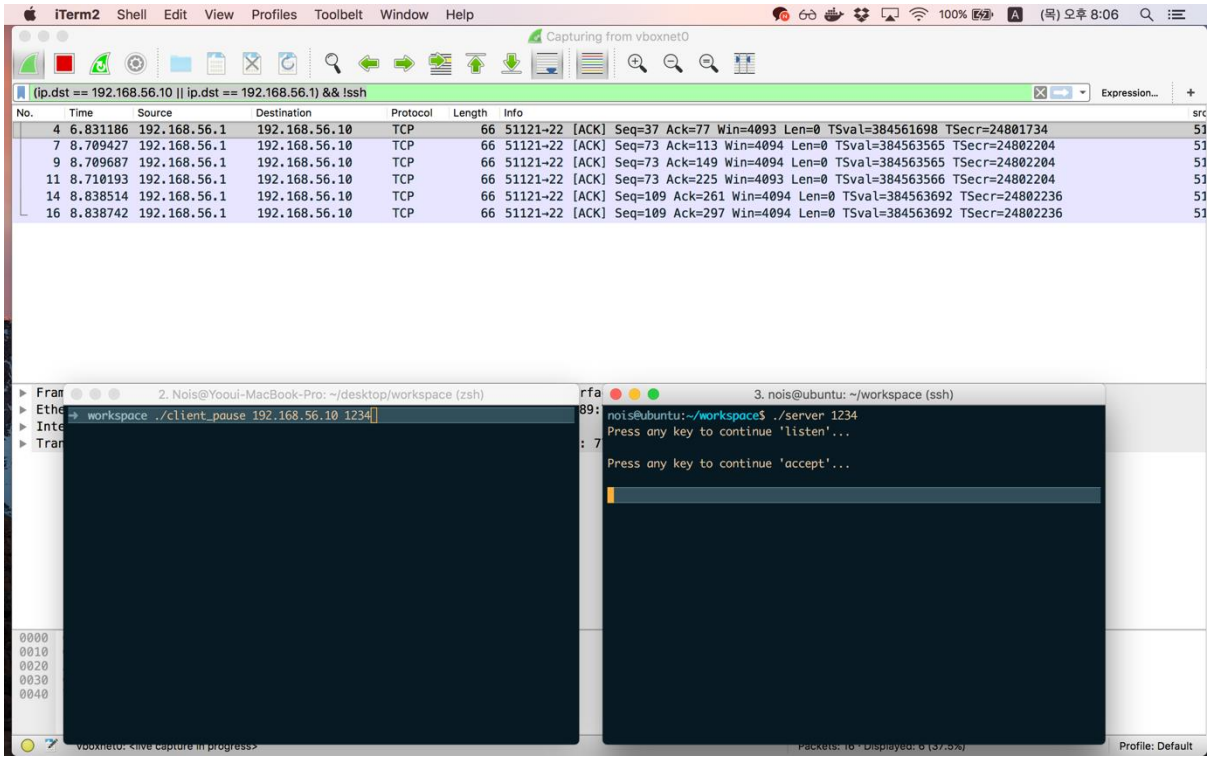
순서는 1주차 강의노트의 이미지를 참고하여 정하게 되었다.



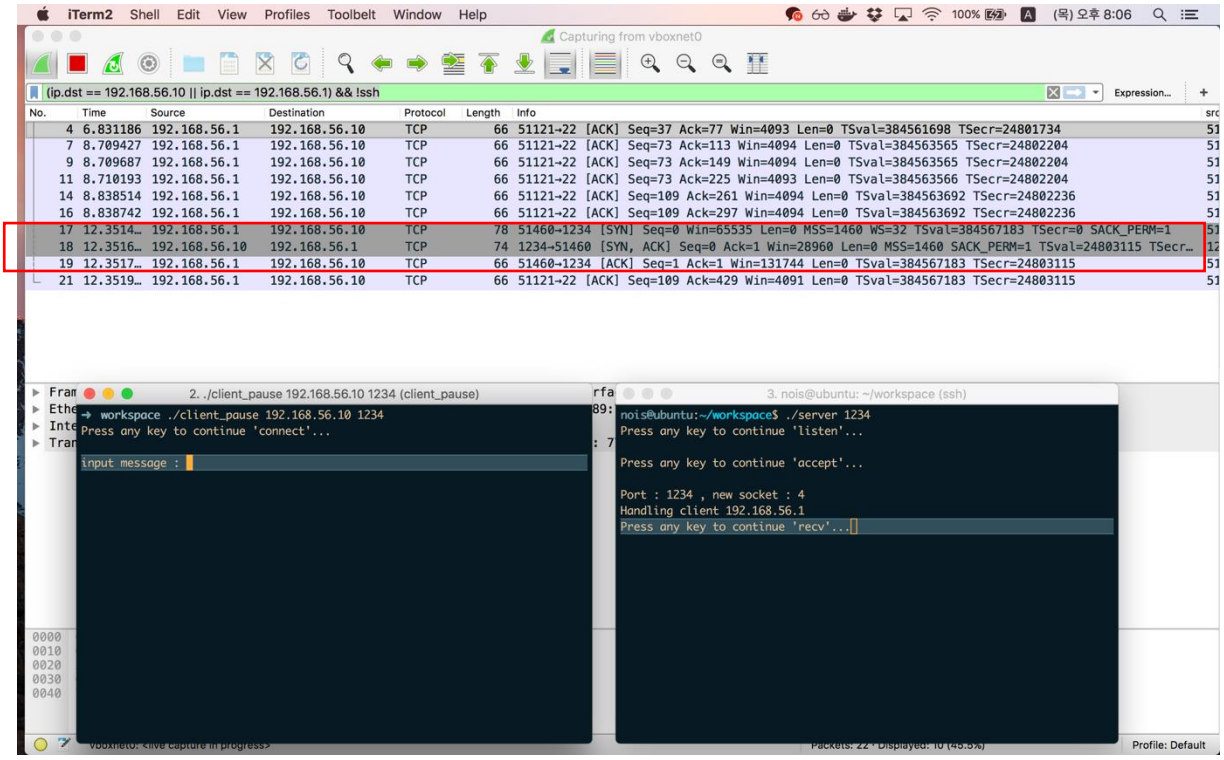
순서는 다음과 같다. 위의 그림을 참고하면서 작성한 동작 순서를 보도록 하자.

1. **Server**의 **listen, accept** 함수 실행을 전후로 각각 (wireshark를 통해) 비교한다.
2. **Client**의 **connect** 함수 실행을 전후로 비교한다.
3. Server에서 accept가 정상적으로 이루어지면 client에서 'hi' 메시지를 입력하여 클라이언트가 send를 준비하도록 한다.
4. **Client**의 **send** 함수 실행을 전후로 비교한다.
5. **Server**의 **recv** 함수 실행을 전후로 비교한다.
6. **Server**의 **send** 함수 실행을 전후로 비교한다.
7. **Client**의 **recv** 함수 실행을 전후로 비교한다.
8. **Client**에서 'quit' 메시지를 입력하여 Client가 send를 준비하도록 한다.
9. 4-7을 반복한다.
10. **Client**의 close 함수 실행을 전후로 비교한다.
11. **Server**의 close 함수 실행을 전후로 비교한다.

불필요한 패킷을 거르기 위해, 필터를 (ip.dst == 192.168.56.10 || ip.dst == 192.168.56.1) && !ssh 로 설정하였다. (여기서 192.168.56.10은 서버, 192.168.56.1은 클라이언트이다.)



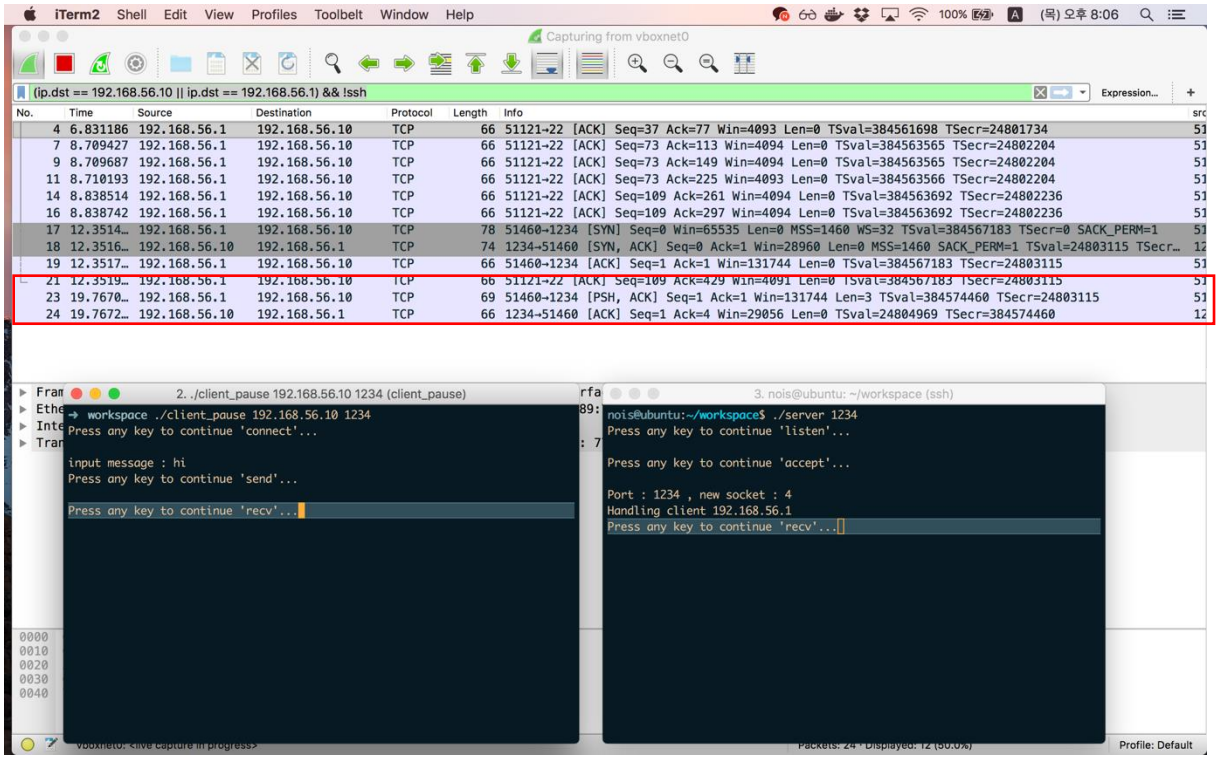
먼저, 위의 그림처럼 포트 넘버를 1234로 설정하여, 서버를 실행시킨 후 accept까지 진행하였다. 이후, 클라이언트에서 해당 서버의 ip주소와 포트 넘버를 입력하여 connect를 시도하였다.



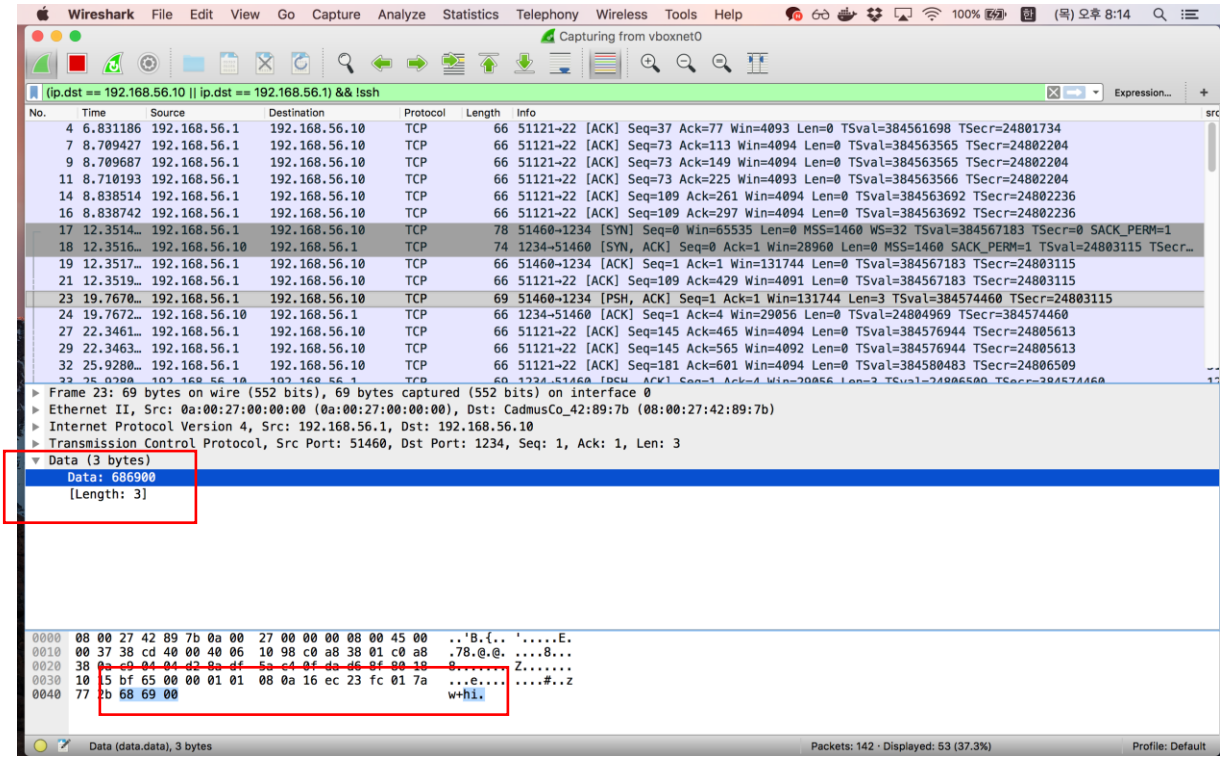
클라이언트의 connect request가 정상적으로 이루어지고, 서버에서는 클라이언트를 핸들링한다

는 메시지를 보여준다. 실제로, 와이어샤크를 보면 3way-handshake이 정상적으로 진행되었음을 확인할 수 있다.

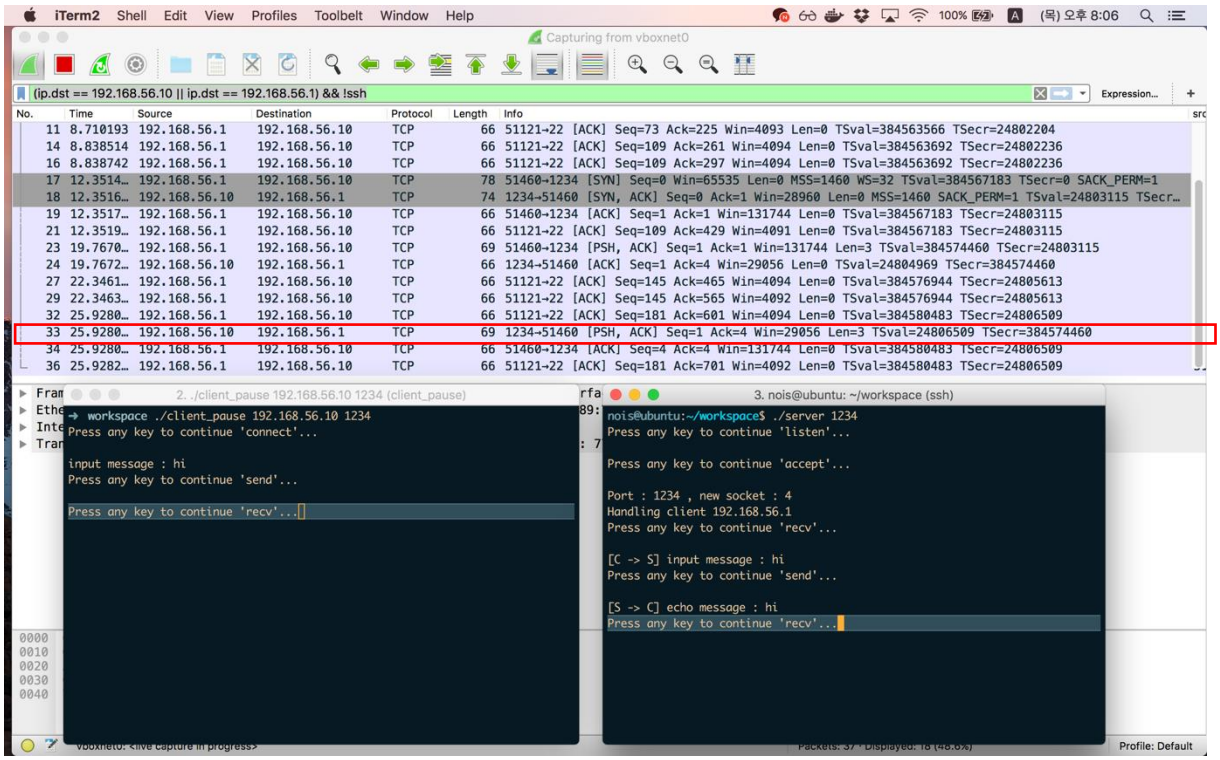
1. Client > Server [SYN] (No.17 패킷)
2. Server > Client [SYN, ACK] (No.18 패킷)
3. Client > Server [ACK]) (No.19 패킷)



다음으로, client에서 hi라는 메시지를 보내었다. 먼저 send를 하고, 서버에서 아직 recv를 하기 전 모습이다. 이때, 와이어샤크에서 23번 [PSH] 패킷을 확인할 수 있다. PSH 플래그는 데이터를 버퍼가 채워지길 기다리지 않고, Application layer로 바로 전달할 때 쓴다고 한다. 이 패킷을 좀 더 자세히 보면,

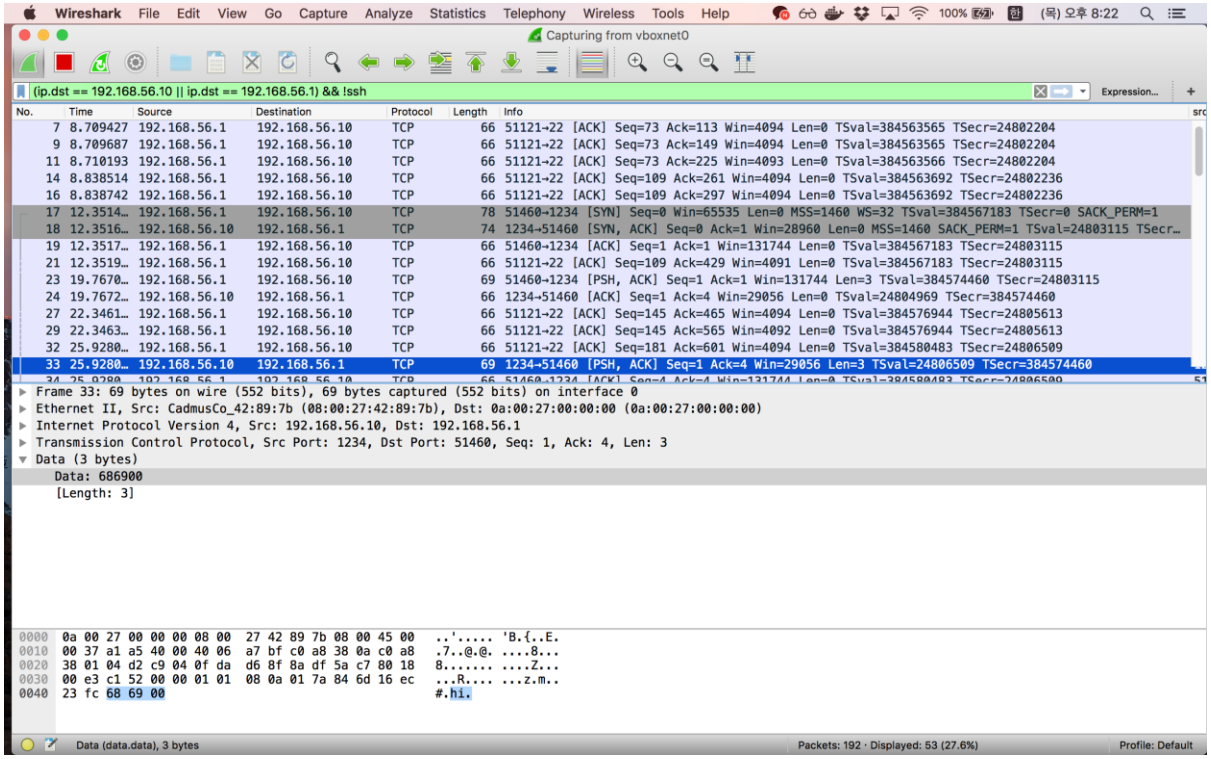


실제로 패킷의 마지막 3 byte의 UTF-8 코드가 68('h'), 69('i'), 00(NULL)로 전송되었음을 확인할 수 있다.

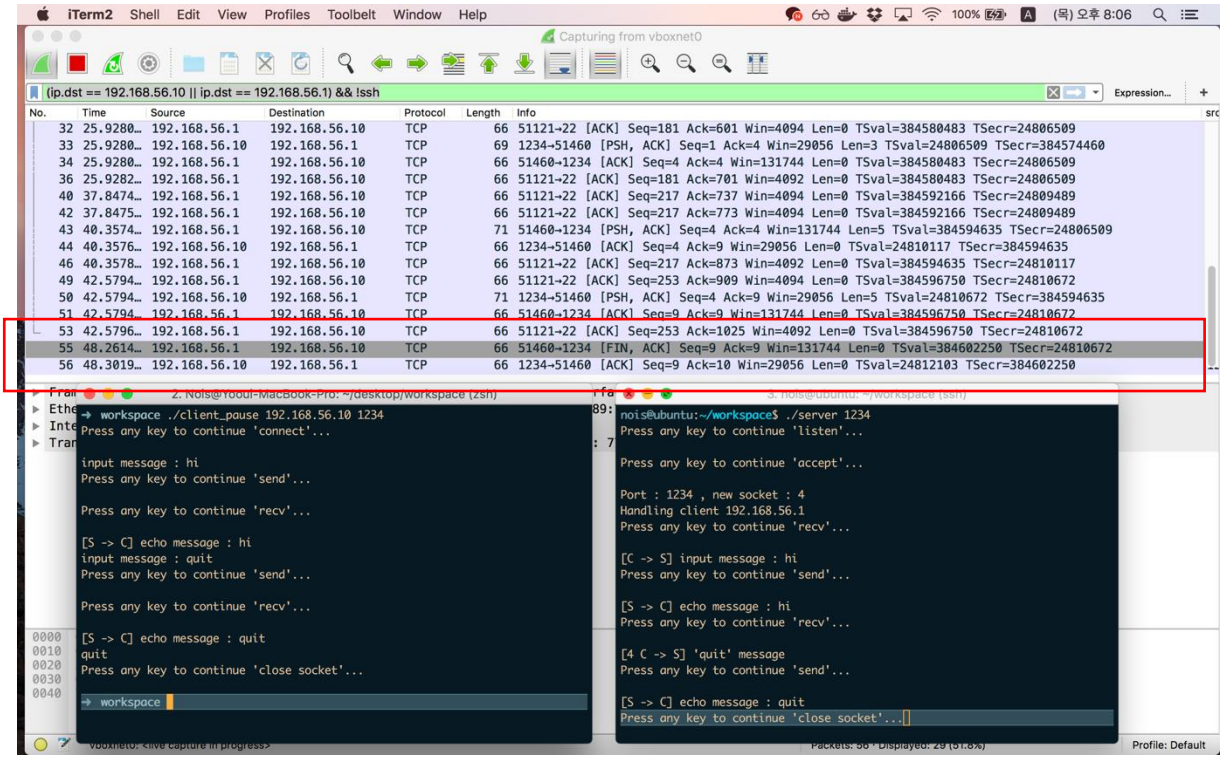


또한 서버에서 recv한 후에 send를 하면, 서버에서 [PSH] 패킷을 Client에게 보내었음을 확인할 수 있다. (33번 패킷) 이 패킷 역시, Client가 Server에게 보냈던 데이터와 같은 데이터값을 보냄을

확인할 수 있다.

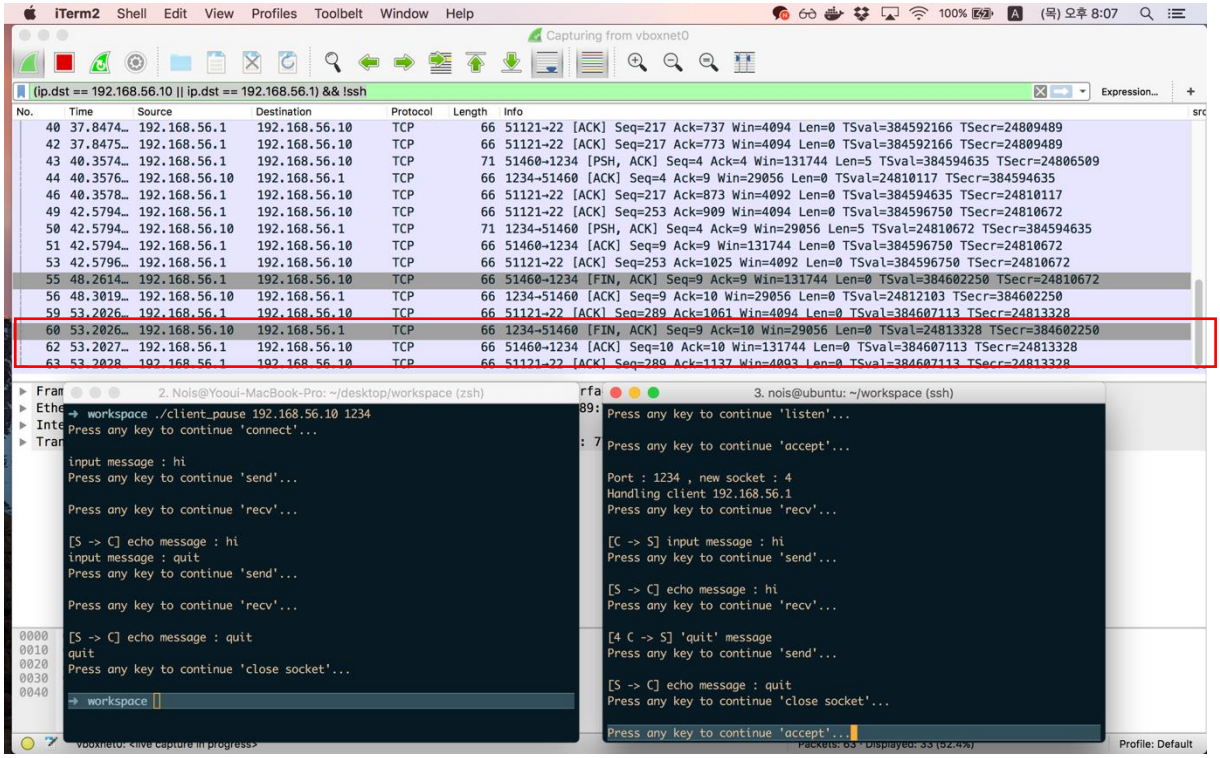


마지막으로, quit메시지를 클라이언트에서 보낼 때인데 먼저 아래 그림을 보자. (데이터를 주고받는 과정은 앞에서 hi메시지를 주고받을 때와 중복되어 따로 설명하지 않았다.)



클라이언트가 quit메시지를 전달받고, 소켓을 close하였다. 실제로 와이어샤크에서는 Client에서

Server로 FIN패킷 보내고, Server에서는 이에 대해 ACK을 한다. 다음 그림을 보자.

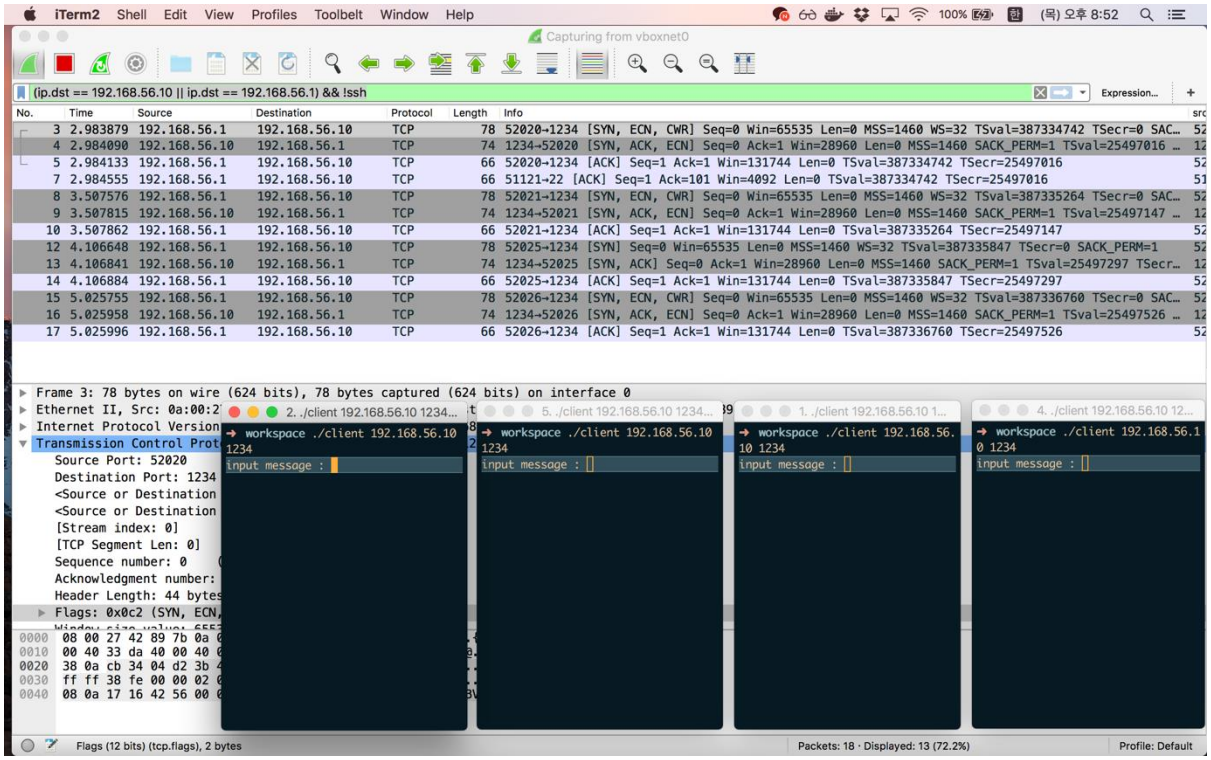


서버에서도 준비가 되면, 해당 소켓을 닫는다. 이제 서버에서 FIN메시지를 다시 보내고, 이에 대한 ACK을 보내 4way handshake를 통해 정상적으로 연결이 종료됨을 확인할 수 있었다.

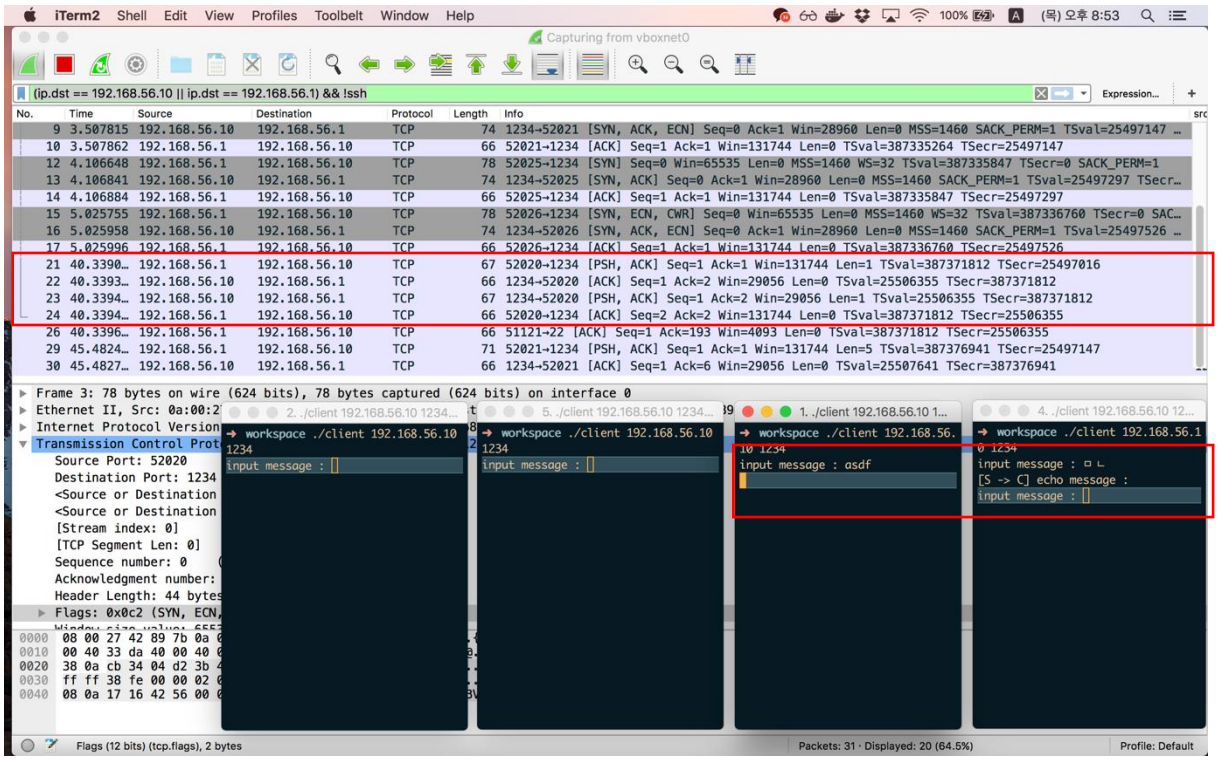
3. queueLimit 값에 따라 발생하는 상황 분석 (실습과제 #4)

동시에 하나의 client만 accept할 수 있는 iterative server의 특성 때문에, queue를 통해서 accept를 대기할 수 있다. 이 queue의 크기에 해당하는 값이 queueLimit이다. 따라서 이 queue가 가득 차있을 때 추가적으로 connect요청을 하는 client가 있으면 loss가 발생하고 retransmission이 발생한다. 따라서 동시에 여러 클라이언트가 connect 요청을 할 때, queueLimit의 값이 충분히 크지 않으면 retransmission이 발생할 확률이 높다고 생각하였다. 이 상황을 wireshark로 확인하기 위해서 queueLimit이 10일 때와 1일 때를 각각 비교해보았다.

테스트는 클라이언트에서 여러 개의 프로세스가 서버에 accept요청을 하는 방법으로 진행하였다. 다음은 queueLimit이 10일 때의 테스트 과정이다.



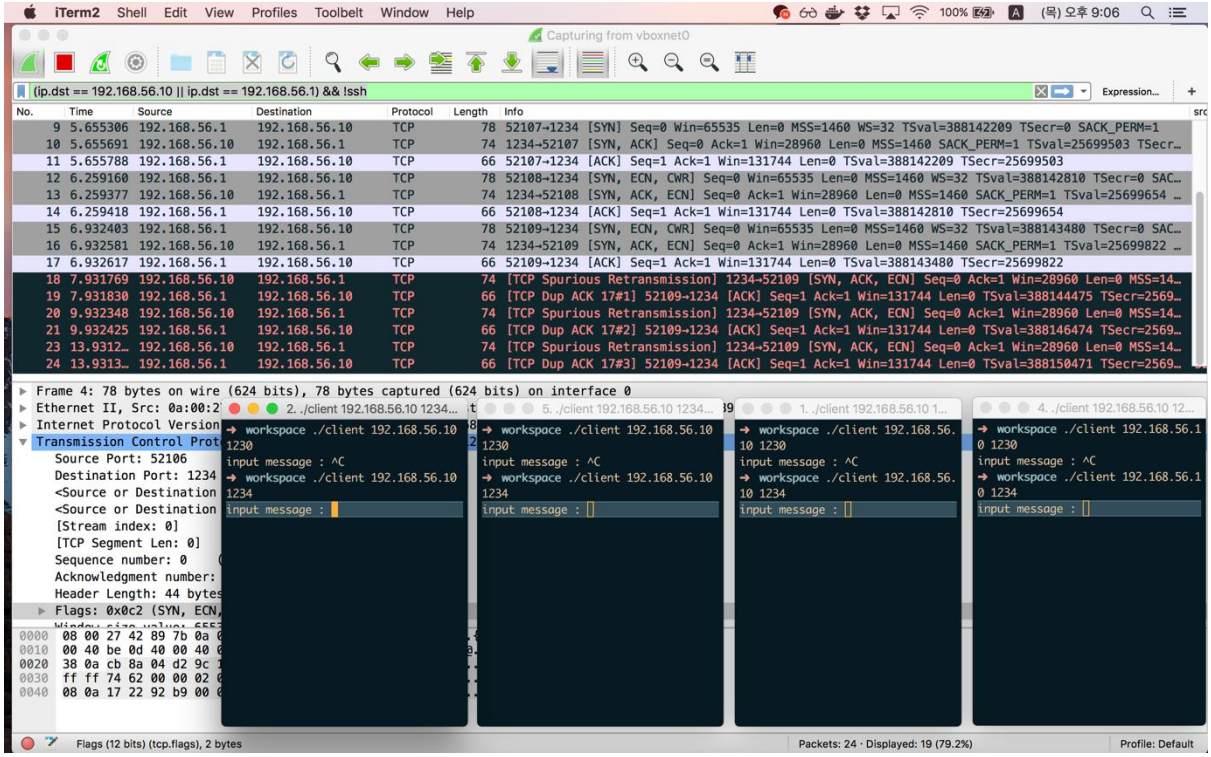
그림을 보자. 필자는 가장 오른쪽에 있는 터미널부터 4, 3, 2, 1번째로 client를 실행시켰다. 모두 정상적으로 연결된 것처럼 보이지만 가장 먼저 connect한 클라이언트만 메시지를 주고받을 수 있었다.



가장 먼저 연결된 4번째 클라이언트와 두 번째로 연결된 클라이언트로 각각 메시지를 보내보았다. 결과는 첫 번째로 연결된 클라이언트에서만 서버로부터 echo 메시지가 돌아왔다. 이는 와이어

샤크와 터미널에서 각각 확인할 수 있었다.

다음은 queueLimit이 1일 때의 테스트 과정이다.



동일한 환경에서 queueLimit을 1로 낮추어 테스트하였더니, 4번째로 연결한 클라이언트에서 Retransmission이 일어남을 확인할 수 있었다. Queue의 size가 1이므로 3번째에 연결하는 클라이언트부터 retransmission이 발생할 줄 알았는데, 그 사이에 accept처리가 이루어졌는지 4번째로 연결한 클라이언트부터 retransmission이 발생했다. 다소 의문점이 남아 몇 번 더 테스트 해보았지만, 같은 결과를 얻었다.