

北京邮电大学软件学院  
2019-2020 学年第一学期实验报告

课程名称: 计算机网络

项目名称: 实验三: 传输层实验

项目完成人:

姓名: 陈梦实 学号: 2017211960

指导教师: 王文东

日 期: 2019 年 12 月 22 日

## 一、实验目的

通过本实验使学生理解并掌握 TCP 连接，掌握 TCP 报文（TCP Segment）字段组成、字段的作用和使用方法。

## 二、实验内容

1. 恢复本课程实验二（“网络层实验”）中的网络环境，并使主机 H1 和主机 H2 在网络层连通。
2. 编写基于 TCP 套接字的 TCP 客户端应用程序和 TCP 服务器应用程序。TCP 客户端程序和 TCP 服务器程序可以采用不同编程语言开发；TCP 服务器应用程序支持与多个 TCP 客户端应用程序同时建立 TCP 连接；TCP 连接建立后，可以在 TCP 连接上传递结构化消息，消息中的字段取值长度为可变长度。
3. 将 TCP 客户端应用程序和 TCP 服务器应用程序分别部署到主机 H1 和主机 H2 上。
4. 在未启动 TCP 服务器应用程序的情况下，在主机 H1 上启动 TCP 客户端应用程序，观测 TCP 客户端应用程序在 TCP 连接建立过程中的收发的 TCP 报文、TCP 连接建立结果。
5. 在启动 TCP 服务器应用程序的情况下，在路由器 R1 上设置丢弃主机 H1 发出的 IP 分组的防火墙规则，使得 TCP 客户端应用程序发出的 TCP 连接建立请求无法到达主机 H2，然后在主机 H1 上启动 TCP 客户端程序，观测 TCP 客户端应用程序在 TCP 连接建立过程中的收发的 TCP 报文、TCP 连接建立结果。
6. 删除在路由器 R1 中设置的丢弃主机 H1 发出的 IP 分组的防火墙规则，使得主机 H1 发出 IP 分组能够到达主机 H2。在主机 H2 启动 TCP 服务器应用程序，在主机 H1 上分别多次运行 TCP 客户端应用程序，在 TCP 服务器和 TCP 客户端间建立多条 TCP 连接。观测 TCP 连接建立的三次握手过程、观测 TCP 连接建立后 TCP 连接上 TCP 报文传递过程。
7. TCP 连接建立后，在客户端与 TCP 服务器间传递数据，观测应用程序一次发送数据量大于链路数据包最大长度情况下的 TCP 报文的发送/接收。
8. 分别先后关闭 TCP 客户端程序和 TCP 服务器程序，观测 TCP 连接拆除过程。

### 三、实验环境

虚拟机软件: VMWare Workstations

操作系统: Linux Ubuntu 16.04

WireShark 软件 (Linux 环境)

Socket 网络编程运行环境: Eclipse 2019

Socket 网络编程开发语言: Java

### 四、实验过程与结果展示

注意: 考虑到本次实验要求给出的结果过多, 若一一给出后再在附录中描述一遍实验过程, 会导致篇幅过长! 由于要求的结果会在每一步骤中体现出来, 故本次实验报告不再将结果展示与实验过程分两部分展示; 在一步步展示实验过程时便也将实验结果给出了。

#### 4.1 恢复实验二 (网络层实验) 中的网络环境, 并使主机 H1 和主机 H2 在网络层连通

H1 的 IP 地址为:

```
h1@ubuntu:~$ ifconfig
ens33      Link encap:以太网 硬件地址
            inet 地址:192.168.30.130
```

H2 的 IP 地址为:

```
h2@ubuntu:~$ ifconfig
ens33      Link encap:以太网 硬件地址
            inet 地址:192.168.59.131
```

H1 ping H2 结果如下:

```
h1@ubuntu:~$ ping 192.168.59.131
PING 192.168.59.131 (192.168.59.131) 56(84) bytes of data.
64 bytes from 192.168.59.131: icmp_seq=1 ttl=62 time=0.944 ms
64 bytes from 192.168.59.131: icmp_seq=2 ttl=62 time=0.927 ms
64 bytes from 192.168.59.131: icmp_seq=3 ttl=62 time=1.00 ms
64 bytes from 192.168.59.131: icmp_seq=4 ttl=62 time=0.316 ms
64 bytes from 192.168.59.131: icmp_seq=5 ttl=62 time=0.372 ms
64 bytes from 192.168.59.131: icmp_seq=6 ttl=62 time=1.13 ms
64 bytes from 192.168.59.131: icmp_seq=7 ttl=62 time=0.935 ms
64 bytes from 192.168.59.131: icmp_seq=8 ttl=62 time=0.993 ms
^C
--- 192.168.59.131 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7060ms
rtt min/avg/max/mdev = 0.316/0.827/1.131/0.286 ms
```

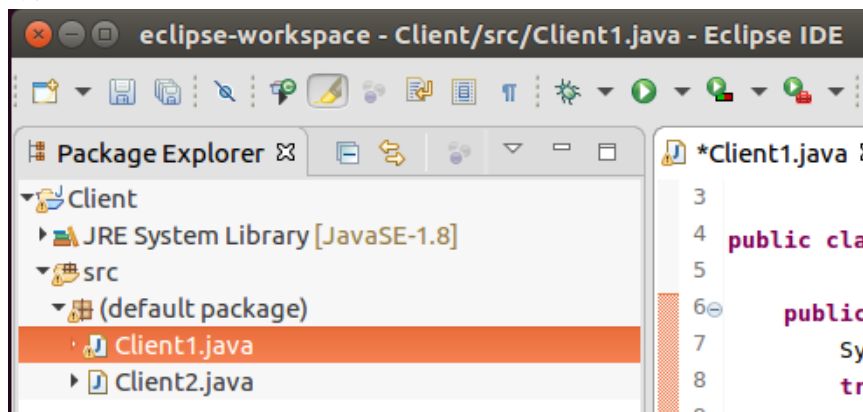
H2 ping H1 结果如下：

```
h2@ubuntu:~$ ping 192.168.30.130
PING 192.168.30.130 (192.168.30.130) 56(84) bytes of data.
64 bytes from 192.168.30.130: icmp_seq=1 ttl=62 time=0.770 ms
64 bytes from 192.168.30.130: icmp_seq=2 ttl=62 time=0.361 ms
64 bytes from 192.168.30.130: icmp_seq=3 ttl=62 time=1.00 ms
64 bytes from 192.168.30.130: icmp_seq=4 ttl=62 time=1.88 ms
64 bytes from 192.168.30.130: icmp_seq=5 ttl=62 time=0.728 ms
64 bytes from 192.168.30.130: icmp_seq=6 ttl=62 time=0.529 ms
64 bytes from 192.168.30.130: icmp_seq=7 ttl=62 time=0.590 ms
64 bytes from 192.168.30.130: icmp_seq=8 ttl=62 time=0.757 ms
^C
--- 192.168.30.130 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7109ms
rtt min/avg/max/mdev = 0.361/0.828/1.882/0.436 ms
```

可见，成功使主机 H1 和主机 H2 在网络层连通。

#### 4.2 设计并实现 TCP 客户端应用程序和 TCP 服务器应用程序，将客户端部署到主机 H1 上，将服务器部署到主机 H2 上。

对于客户端：



为方便观察后面“多个客户端连接服务器”，使用两个类（分别名为“Client1”和“Client2”）来实现，其内容除了标号不同外一模一样（实际上打开多个 Client1 效果也一样，但不方便观察）。程序如下：

##### Client1.java

```
import java.net.*;
import java.io.*;

public class Client1 {

    public static void main(String args[]) {
        System.out.println("Hello Client1!");
        try{
            //创建 Socket
            Socket socket = new Socket("192.168.59.131", 8888);
```

```

        //建立连接
        InputStreamReader Sysin = new InputStreamReader(System.in);
        BufferedReader SysBuf = new BufferedReader(Sysin);

        InputStreamReader Socin = new
        InputStreamReader(socket.getInputStream());
        BufferedReader SocBuf = new BufferedReader(Socin);

        PrintWriter Socout = new
        PrintWriter(socket.getOutputStream());

        //进行通信
        String readline = "first";
        while(!readline.equals("bye")){
            String words = SysBuf.readLine();
            Socout.println(words);
            Socout.flush();
            readline = SocBuf.readLine();
            System.out.println("Client1:" + words);
            System.out.println("Server:" + readline);
        }
        socket.close();
    } catch (Exception e) {
        System.out.println("Error:" + e);
    }
}
}

```

## Client2. java

```

import java.net.*;
import java.io.*;

public class Client2 {

    public static void main(String args[]) {
        System.out.println("Hello Client2!");
        try{

            //创建 Socket
            Socket socket = new Socket("192.168.59.131", 8888);

            //建立连接
            InputStreamReader Sysin = new InputStreamReader(System.in);
            BufferedReader SysBuf = new BufferedReader(Sysin);

```

```

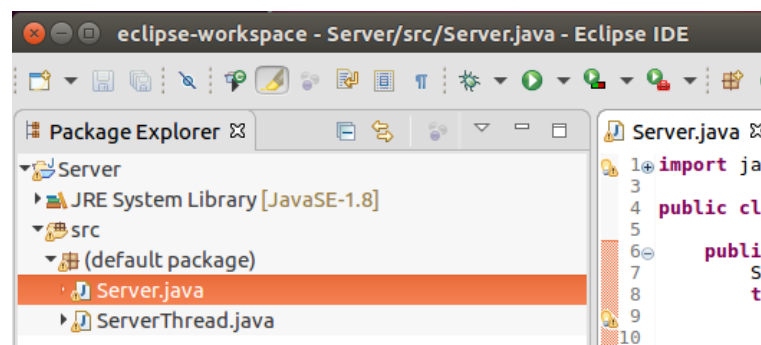
        InputStreamReader Socin = new
InputStreamReader(socket.getInputStream());
        BufferedReader SocBuf = new BufferedReader(Socin);

        PrintWriter Socout = new
PrintWriter(socket.getOutputStream());

        //进行通信
        String readline = "first";
        while(!readline.equals("bye")) {
            String words = SocBuf.readLine();
            Socout.println(words);
            Socout.flush();
            readline = SocBuf.readLine();
            System.out.println("Client2:" + words);
            System.out.println("Server:" + readline);
        }
        socket.close();
    } catch (Exception e) {
        System.out.println("Error:" + e);
    }
}
}

```

对于服务器:



由于后面要求“服务器能连接多个客户端”，本次服务器实现使用多线程思路，对于每一个客户端的连接都开辟一个线程处理。故有 Server 主程序和 Thread 的程序，程序如下：

## Server.java

```

import java.net.*;
import java.io.*;

public class Server {

```

```

public static void main(String args[]) {
    System.out.println("Hello Server!");
    try {
        ServerSocket server = new ServerSocket(8888);
        Socket socket = null;
        int count = 0;
        while(true) {
            socket = server.accept();
            InetAddress inetAddress = socket.getInetAddress();
            ServerThread thread = new
ServerThread(socket, inetAddress, count, server);
            thread.start();
            count++;
            System.out.println("Server Numbers: " + count);
        }
    } catch (Exception e) {
        System.out.println("Error"+e);
    }
}

```

## ServerThread. java

```

import java.io.*;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;

public class ServerThread extends Thread{
    ServerSocket server = null;
    Socket socket = null;
    InetAddress inetAddress = null;
    int num;
    public ServerThread(Socket socket, InetAddress inetAddress, int
num, ServerSocket server) {
        this.server=server;
        this.socket = socket;
        this.inetAddress = inetAddress;
        this.num = num;
    }

    @Override
    public void run() {

```

```

try {
    InputStreamReader Socin = new
InputStreamReader(socket.getInputStream());
    BufferedReader SocBuf = new BufferedReader(Socin);

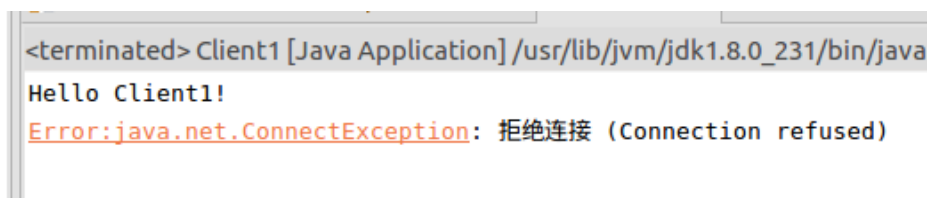
    PrintWriter Socout = new PrintWriter(socket.getOutputStream());
    String readline = SocBuf.readLine();
    while(!readline.equals("bye")) {
        System.out.println("Client:"+readline);
        Socout.println("get Client"+num+": "+(new Date()));
        Socout.flush();

        readline=SocBuf.readLine();
    }
    Socout.println("bye");
    Socout.flush();
    socket.close();
} catch(Exception e) {
    System.out.println("Error"+e);
}
}
}

```

4.3 在未启动 TCP 服务器应用程序的情况下，在主机 H1、主机 H2 上分别启动 Wireshark 软件抓取主机 H1、H2 收发的 IP 分组；然后在主机 H1 上启动 TCP 客户端应用程序，观察 TCP 客户端应用程序的执行情况，分析 Wireshark 软件抓取的 TCP 连接建立请求消息和响应消息、以及这些 TCP 报文中的 SYN 比特位、RST 比特位的取值情况。

#### 4.3.1 TCP 客户端应用程序的执行情况



图中可见，在未启动服务器，只启动 H1 中的客户端程序时，无法连接，会提示拒绝连接错误。

#### 4.3.2 TCP 报文分析

对于该过程，H1 和 H2 的 Wireshark 抓到了同样的两个包，如下：

H1:



正在捕获 ens33

| No. | Time         | Source         | Destination    | Protocol | Length | Info                        |
|-----|--------------|----------------|----------------|----------|--------|-----------------------------|
| 47  | 41.645875088 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | 58436 → 8888 [SYN] Seq=...  |
| 48  | 41.647598095 | 192.168.59.131 | 192.168.30.130 | TCP      | 60     | 8888 → 58436 [RST, ACK] ... |

H2:

\*ens33

| No. | Time         | Source         | Destination    | Protocol | Length | Info                         |
|-----|--------------|----------------|----------------|----------|--------|------------------------------|
| 21  | 15.650983731 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | 58436 → 8888 [SYN] Seq=0 ... |
| 22  | 15.651041875 | 192.168.59.131 | 192.168.30.130 | TCP      | 54     | 8888 → 58436 [RST, ACK] S... |

只取 H1 的两个包进行分析:

#### 4.3.2.1 47 号帧 连接建立请求消息 SYN=1, RST=0

Wireshark · 分组 47 · ens33

Frame 47: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0  
 Ethernet II, Src: Vmware\_01:c9:7f (00:0c:29:01:c9:7f), Dst: Vmware\_24:86:9a (00:0c:29:24:86:9a)  
 Internet Protocol Version 4, Src: 192.168.30.130, Dst: 192.168.59.131  
 Transmission Control Protocol, Src Port: 58436, Dst Port: 8888, Seq: 0, Len: 0

Source Port: 58436  
 Destination Port: 8888  
 [Stream index: 0]  
 [TCP Segment Len: 0]  
 Sequence number: 0 (relative sequence number)  
 [Next sequence number: 0 (relative sequence number)]  
 Acknowledgment number: 0  
 1010 .... = Header Length: 40 bytes (10)

Flags: 0x002 (SYN)

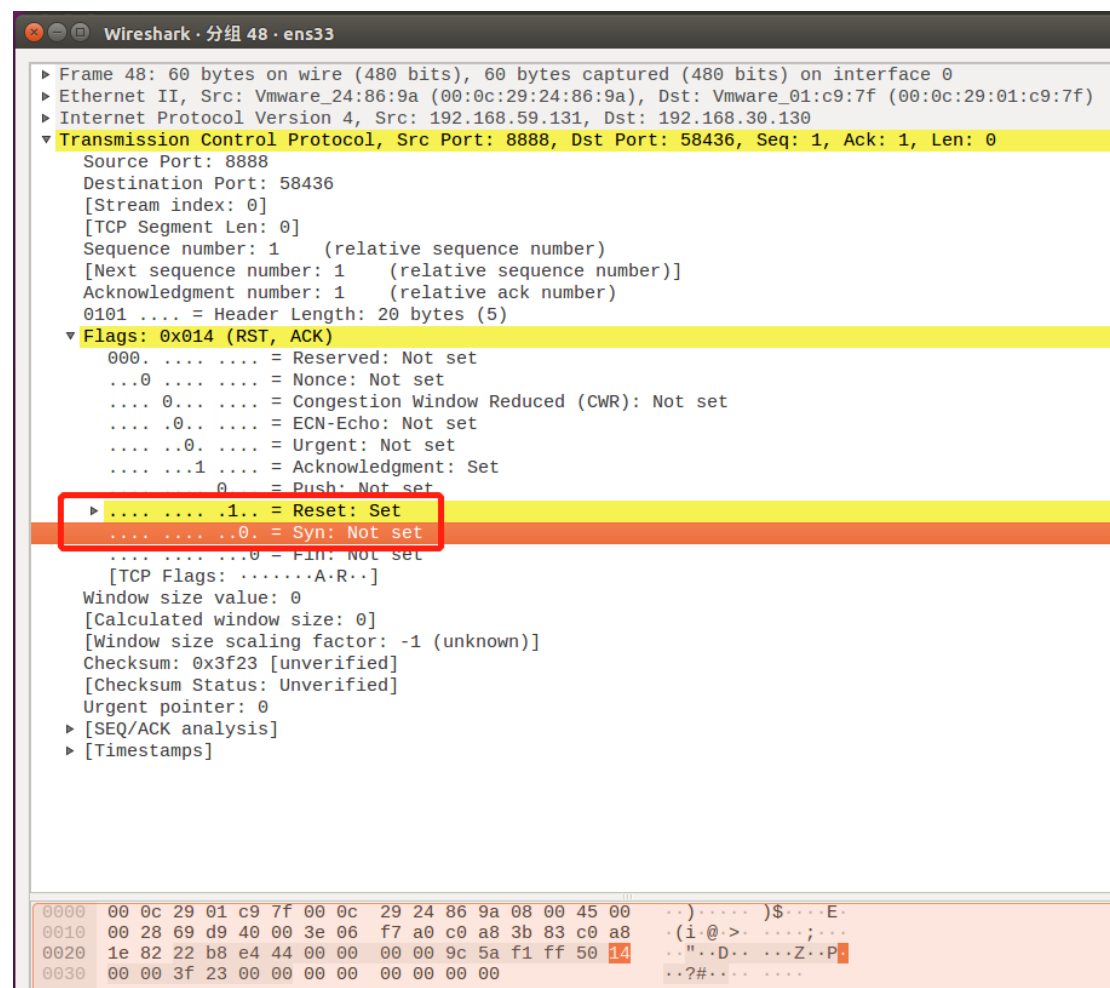
000. .... = Reserved: Not set  
 ...0 .... = Nonce: Not set  
 .... 0... = Congestion Window Reduced (CWR): Not set  
 .... 0... = ECN-Echo: Not set  
 .... 0... = Urgent: Not set  
 .... 0... = Acknowledgment: Not set  
 .... 0... = Push: Not set  
 0 .... = Reset: Not set  
 ...1. .... = Syn: Set  
 .... 0... = Fin: Not set

[TCP Flags: .....S.]  
 Window size value: 29200  
 [Calculated window size: 29200]  
 Checksum: 0xdb84 [unverified]  
 [Checksum Status: Unverified]  
 Urgent pointer: 0  
 Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale  
 [Timestamps]

0000 00 0c 29 24 86 9a 00 0c 29 01 c9 7f 08 00 45 00 ..)\$.E  
 0010 00 3c 38 3c 40 00 40 06 27 2a c0 a8 1e 82 c0 a8 <8<@ @ !\*  
 0020 3b 83 e4 44 22 b8 9c 5a f1 fe 00 00 00 00 a0 02 ;-D"-Z  
 0030 72 10 db 84 00 00 02 04 05 b4 04 02 08 0a 78 34 r.....x4  
 0040 a1 e9 00 00 00 00 01 03 03 07 .....

对于 47 号帧，它是由客户端向服务器发出的连接建立请求消息，在发出该请求时客户端并不知道服务器的状况，因此该帧与正常的“TCP 三次握手”的第一个连接建立请求报文相同，即：SYN=1, RST=0。

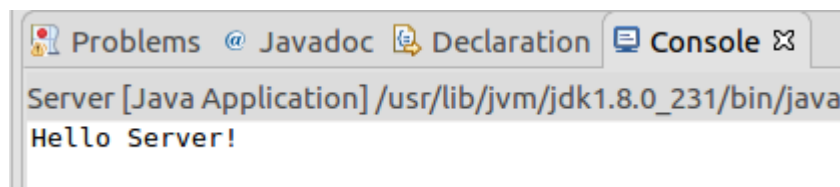
#### 4.3.2.1 48 号帧 响应消息 SYN=0, RST=1



48 号帧是服务器向客户端发送的响应消息，由于此时服务器并未启动，故连接无法成功。ACK 表示响应、RST 表示连接重置，该帧中 ACK 与 RST 同时使用，而一般当出现 RST 包时，我们便认为客户端与服务器断开了连接。故该帧的 SYN=0, RST=1。

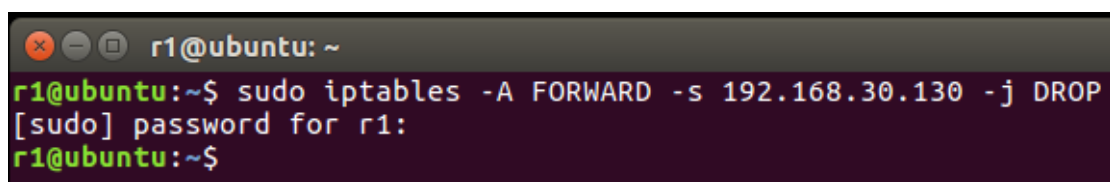
4.4 在启动 TCP 服务器应用程序的情况下，在路由器 R1 上使用 iptables 命令防火墙规则，设置丢弃从主机 H1 发出的 IP 分组，使得 TCP 客户端应用程序发出的 TCP 连接请求消息无法到达主机 H2。然后在主机 H1 上启动 TCP 客户端程序，观察 TCP 客户端应用程序的执行情况，分析 Wireshark 软件抓取的 TCP 连接建立请求消息、这些 TCP 报文中的 SYN 比特位、ACK 比特位、序列号字段的取值情况、以及重发的 TCP 连接请求分组的发送时间间隔。在某一次 H1 尝试重连时，删除刚才设置的过滤规则，使得 TCP 客户端重发的连接请求报文能够到达 TCP 服务器，从而 TCP 服务器能够回复 TCP 连接响应消息，从而接下来 TCP 客户端回复 TCP 连接响应消息，这样完成了 TCP 连接建立的三次握手过程。

#### 4.4.1 启动 H2 中的服务器应用程序



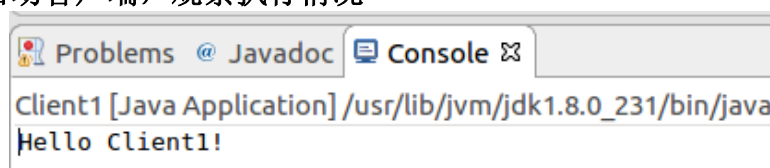
可见，服务器已成功启动，正在等待客户端连接。

#### 4.4.2 在路由器 R1 上使用 iptables 设置防火墙



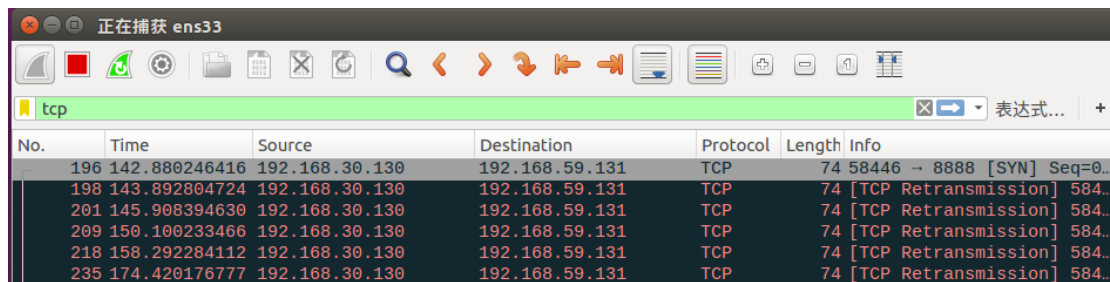
可见，过滤规则设置成功。

#### 4.4.2 H1 启动客户端，观察执行情况



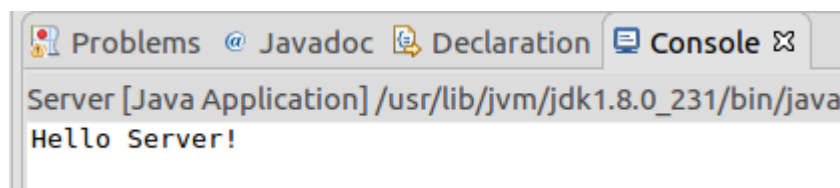
此时于 H1 中成功启动客户端 Client1，由于过滤规则致使建立连接请求无

法送达服务器端，因此无法获得服务器端返回的响应消息，故不会提示“拒绝连接”，而是尝试重传，Wireshark 抓包如下：



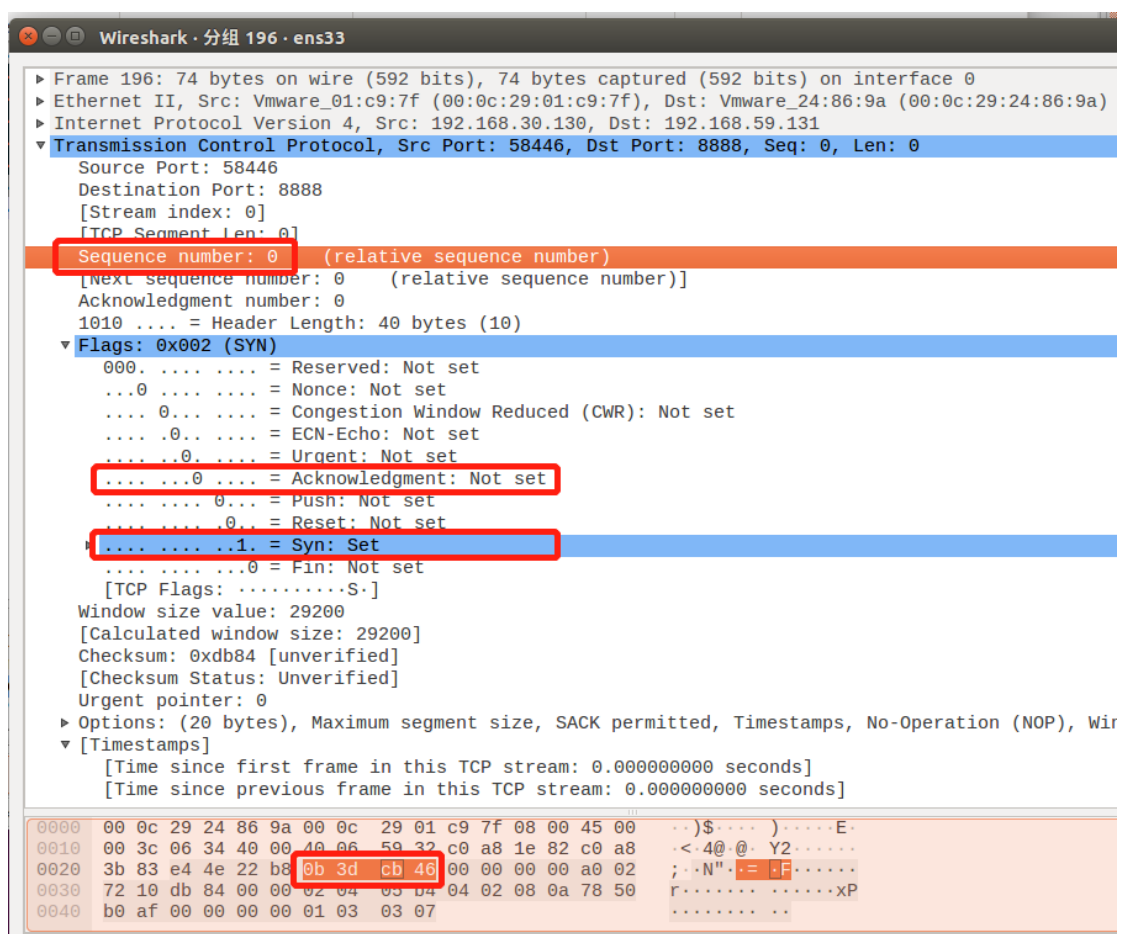
| No. | Time          | Source         | Destination    | Protocol | Length | Info                        |
|-----|---------------|----------------|----------------|----------|--------|-----------------------------|
| 196 | 142.880246416 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | 58446 → 8888 [SYN] Seq=0... |
| 198 | 143.892804724 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |
| 201 | 145.908394630 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |
| 209 | 150.100233466 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |
| 218 | 158.292284112 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |
| 235 | 174.420176777 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |

此时，H2 中的服务器应用程序没有收到建立连接请求，故无反应，如下：



#### 4.4.3 分析 Wireshark 抓到的包

##### ① 196 号帧 建立连接请求报文 SYN=1, ACK=0, 序列号：0



Wireshark · 分组 196 · ens33

Frame 196: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0

Ethernet II, Src: Vmware\_01:c9:7f (00:0c:29:01:c9:7f), Dst: Vmware\_24:86:9a (00:0c:29:24:86:9a)

Internet Protocol Version 4, Src: 192.168.30.130, Dst: 192.168.59.131

Transmission Control Protocol, Src Port: 58446, Dst Port: 8888, Seq: 0, Len: 0

Source Port: 58446

Destination Port: 8888

[Stream index: 0]

[TCP Segment Len: 0]

Sequence number: 0 (relative sequence number)

[Next sequence number: 0 (relative sequence number)]

Acknowledgment number: 0

1010 .... = Header Length: 40 bytes (10)

Flags: 0x002 (SYN)

000. .... = Reserved: Not set

...0 .... = Nonce: Not set

.... 0... = Congestion Window Reduced (CWR): Not set

.... 0... = ECN-Echo: Not set

.... 0... = Urgent: Not set

.... 0... = Acknowledgment: Not set

.... 0... = Push: Not set

.... 0... = Reset: Not set

.... 1... = Syn: Set

.... 0... = Fin: Not set

[TCP Flags: .....S.]

Window size value: 29200

[Calculated window size: 29200]

Checksum: 0xdb84 [unverified]

[Checksum Status: Unverified]

Urgent pointer: 0

Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Wir

[Timestamps]

[Time since first frame in this TCP stream: 0.000000000 seconds]

[Time since previous frame in this TCP stream: 0.000000000 seconds]

0000 00 0c 29 24 86 9a 00 0c 29 01 c9 7f 08 00 45 00 ..)\$... )...E

0010 00 3c 06 34 40 00 40 06 59 32 c0 a8 1e 82 c0 a8 <.4@. Y2...

0020 3b 83 e4 4e 22 b8 0b 3d cb 46 00 00 00 00 a0 02 ;.N".= xE.....

0030 72 10 db 84 00 00 02 04 05 04 04 02 08 0a 78 50 r.....xP

0040 b0 af 00 00 00 00 01 03 03 07 .....

该报文是 TCP 三次握手中的第一次握手，是由客户端发向服务器的“建立连接请求”，这是一个 SYN 标记的包，告诉服务器请求建立连接，故其 SYN 为 1，ACK 为 0。



根据上面两个重传的帧与一开始的帧的内容对比可见，重传的帧并未作任何改动。

下面分析重传间隔时间：

|     | Time          | Source         | Destination    | Protocol | Length | Info                        |
|-----|---------------|----------------|----------------|----------|--------|-----------------------------|
| 196 | 142.880246416 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | 58446 → 8888 [SYN] Seq=0... |
| 198 | 143.892804724 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |
| 201 | 145.908394630 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |
| 209 | 150.100233466 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |
| 218 | 158.292284112 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |
| 235 | 174.420176777 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584... |

上图是第一次握手报文与五次重传帧的时间，可以看出大致时间间隔为：1 秒、2 秒、4 秒、8 秒、16 秒。不难发现其规律是：第  $i$  次重传与上一次的时间间隔为  $2^{(i-1)}$ ， $i \geq 1$ 。据此推断，若过滤规则继续存在，则下一次重传可能是 32 秒之后。（也可能连接超时就不重传了）

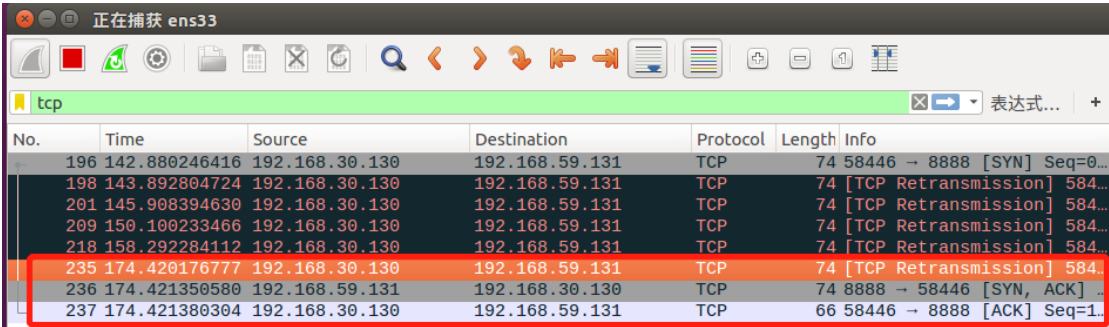
更为精准的时间数据可以在报文详细中最下面的 Timestamps 中看到。

#### 4.4.4 在某一次重传时删除 R1 的过滤规则，使连接成功

在 235 号帧（即第 5 次重传）前，于 R1 中删除过滤规则，删除方法如下

```
sudo iptables -D FORWARD -s 192.168.30.130 -j DROP
```

此时可见，第五次重传后成功进行 TCP 三次握手：



| No. | Time          | Source         | Destination    | Protocol | Length | Info                             |
|-----|---------------|----------------|----------------|----------|--------|----------------------------------|
| 196 | 142.880246416 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | 58446 → 8888 [SYN] Seq=0...      |
| 198 | 143.892804724 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584...      |
| 201 | 145.908394630 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584...      |
| 209 | 150.100233466 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584...      |
| 218 | 158.292284112 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584...      |
| 235 | 174.420176777 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | [TCP Retransmission] 584...      |
| 236 | 174.421350580 | 192.168.59.131 | 192.168.30.130 | TCP      | 74     | 8888 → 58446 [SYN, ACK] Seq=1... |
| 237 | 174.421380304 | 192.168.30.130 | 192.168.59.131 | TCP      | 66     | 58446 → 8888 [ACK] Seq=1...      |

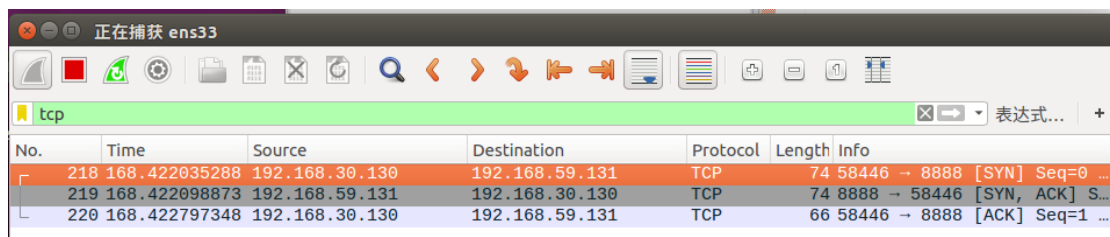
查看 H2 的服务器应用程序，也可证明连接成功。

```
Server [Java Application] /usr/lib/jvm/jdk1.8.0_231/bin/java (
Hello Server!
Server Numbers: 1
```



4.5 观察 TCP 连接建立的三次握手过程中的 TCP 报文交互过程: 分析携带 TCP 报文的 IP 分组的源 IP 地址、目的 IP 地址、协议(protocol) 字段、IP 分组头长度 (IHL) 字段、IP 分组长度 (Total Length) 字段取值; 分析 TCP 连接建立过程中的每个 TCP 报文的源端口号、目的端口号、SYN 比特位、ACK 比特位、FIN 比特位、序列号 (Sequence Number) 字段、确认号 (Acknowledgement Number) 字段的取值。

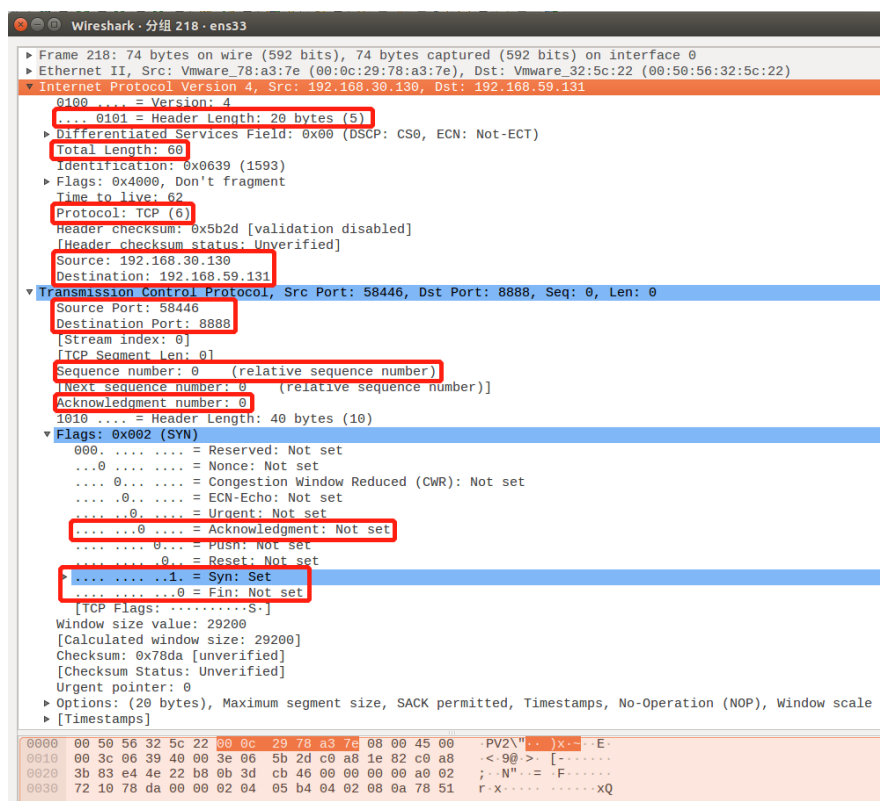
本步中对 H2 使用 Wireshark 抓包观察到的三次握手进行分析, 如下图:



| No. | Time          | Source         | Destination    | Protocol | Length | Info                         |
|-----|---------------|----------------|----------------|----------|--------|------------------------------|
| 218 | 168.422035288 | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | 58446 → 8888 [SYN] Seq=0 ... |
| 219 | 168.422098873 | 192.168.59.131 | 192.168.30.130 | TCP      | 74     | 8888 → 58446 [SYN, ACK] S... |
| 220 | 168.422797348 | 192.168.30.130 | 192.168.59.131 | TCP      | 66     | 58446 → 8888 [ACK] Seq=1 ... |

由于对 IP 分组各字段的分析已于实验一中做过, 本处只取一帧进行分析, 其他两个帧同理看待即可。

218 号帧 第一次握手: 客户端向服务端发出建立连接请求



```
Frame 218: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
Ethernet II, Src: Vmware_78:a3:7e (00:0c:29:78:a3:7e), Dst: Vmware_32:5c:22 (00:50:56:32:5c:22)
Internet Protocol Version 4, Src: 192.168.30.130, Dst: 192.168.59.131
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
  Total Length: 60
  Identification: 0x0639 (1593)
  Flags: 0x4000, Don't fragment
  Time to live: 62
  Protocol: TCP (6)
  Header checksum: 0x5b2d [validation disabled]
  [Header checksum status: Unverified]
  Source: 192.168.30.130
  Destination: 192.168.59.131
Transmission Control Protocol, Src Port: 58446, Dst Port: 8888, Seq: 0, Len: 0
  Source Port: 58446
  Destination Port: 8888
  [Stream index: 0]
  [TCP Segment Len: 0]
  Sequence number: 0 (relative sequence number)
  [Next sequence number: 0 (relative sequence number)]
  Acknowledgment number: 0
  1010 .... = Header Length: 40 bytes (10)
  Flags: 0x002 (SYN)
  000. .... = Reserved: Not set
  ...0 .... = Nonce: Not set
  .... 0... = Congestion Window Reduced (CWR): Not set
  .... 0... = ECN-Echo: Not set
  .... 0... = Urgent: Not set
  .... 0... = Acknowledgment: Not set
  .... 0... = Push: Not set
  .... 0... = Reset: Not set
  .... 0... = Syn: Set
  .... 0... = Fin: Not set
  [TCP Flags: .....S.]
  Window size value: 29200
  [Calculated window size: 29200]
  Checksum: 0x78da [unverified]
  [Checksum Status: Unverified]
  Urgent pointer: 0
  Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
  [Timestamps]
```

IP 分组

源 IP 地址: 192.168.30.130 (c0 a8 1e 82)

顾名思义，发送方的 IP 地址

目的 IP 地址: 192.168.59.131 (c0 a8 3b 83)

顾名思义，接收方的 IP 地址

协议 (protocol) 字段: TCP (06)

指 IP 协议的版本，此处为 TCP。

IP 分组头长度 (IHL) 字段: 20 字节

占 4 位，该字段的单位是 32 位字 (1 个 32 位字长是 4 字节)，因此当 IP 报头长度为 1111 时，报头长度就达到最大值 60 字节。当 IP 分组的首部长度不是 4 字节的整数倍是，就需要对填充域加以填充。最常用的报头长度为 20 位 (报头长度值为 0101)，这时不使用任何选项。

IP 分组长度 (Total Length) 字段: 60 (00 3c)

指报头和数据之和的长度，单位是字节。

## TCP 报文

TCP 报文源端口号: 58446 (e4 4e)

TCP 报文目的端口号: 8888 (22 b8)

SYN 比特位: 1

ACK 比特位: 0

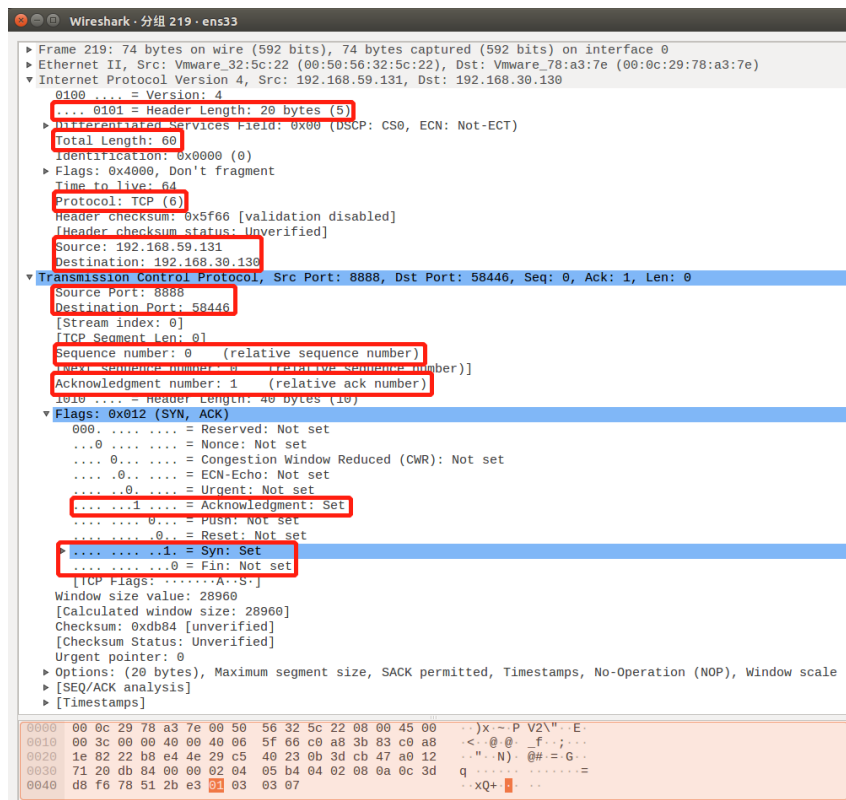
FIN 比特位: 0

序列号字段: 0 (相对序列号，真实序列号为 0b 3d cb 46)

确认号字段: 0

客户端向服务器发送一个同步数据包请求建立连接，该数据包中，初始序列号 (ISN) 是客户端随机产生的一个值 (此处使用相对序列号，为 0)，确认号是 0；

## 219 号帧 第二次握手：服务器向客户端发送响应信息





## IP 分组

源 IP 地址: 192.168.59.131 (c0 a8 3b 83)

目的 IP 地址: 192.168.30.130 (c0 a8 1e 82)

协议 (protocol) 字段: TCP (06)

IP 分组头长度 (IHL) 字段: 20 字节

IP 分组长度 (Total Length) 字段: 60 (00 3c)

## TCP 报文

TCP 报文源端口号: 8888 (22 b8)

TCP 报文目的端口号: 58446 (e4 4e)

SYN 比特位: 1

ACK 比特位: 1

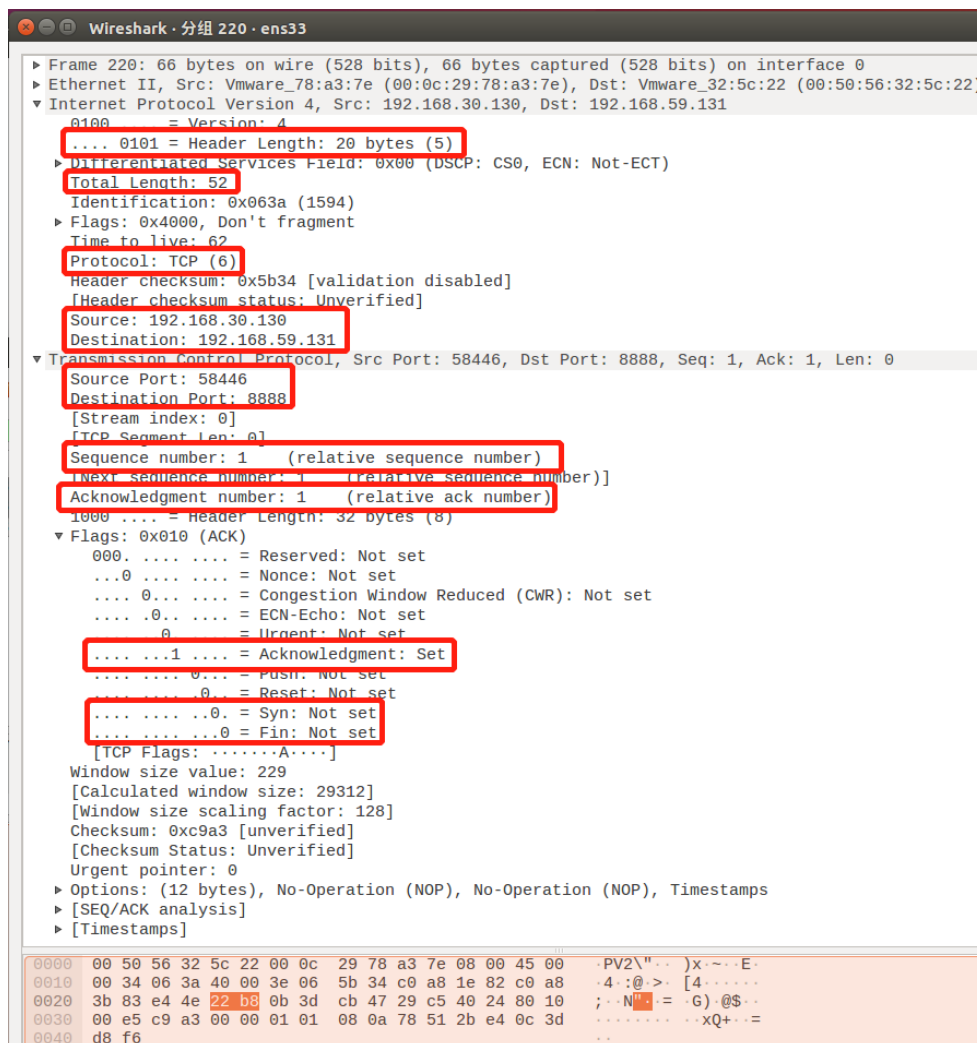
FIN 比特位: 0

序列号字段: 0 (相对序列号, 真实序列号为: 29 c5 40 23)

确认号字段: 1 (相对确认号, 真实确认号为: 0b 3d cb 47)

服务器收到这个同步请求数据包后, 会对客户端进行一个同步确认。这个数据包中, 序列号 (ISN) 是服务器随机产生的一个值 (此处使用相对序列号, 为 0), 确认号是客户端的初始序列号+1;

## 220 号帧 第三次握手: 客户端向服务器发送确认信息



## IP 分组

源 IP 地址: 192.168.30.130 (c0 a8 1e 82)

目的 IP 地址: 192.168.59.131 (c0 a8 3b 83)

协议 (protocol) 字段: TCP (06)

IP 分组头长度 (IHL) 字段: 20 字节

IP 分组长度 (Total Length) 字段: 52 (00 34)

## TCP 报文

TCP 报文源端口号: 58446 (e4 4e)

TCP 报文目的端口号: 8888 (22 b8)

SYN 比特位: 0

ACK 比特位: 1

FIN 比特位: 0

序列号字段: 1 (相对序列号, 真实序列号为: 0b 3d cb 47)

确认号字段: 1 (相对确认号, 真实确认号为: 29 c5 40 24)

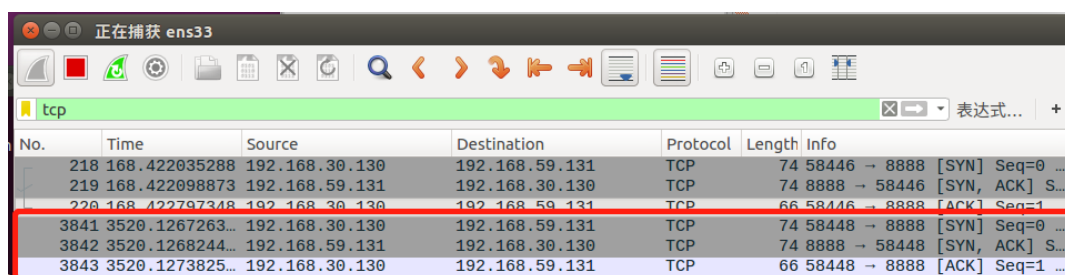
客户端收到这个同步确认数据包后, 再对服务器进行一个确认。该数据包中, 序列号是上一个同步请求数据包中的确认号值, 确认号是服务器的初始序列号 +1。

4.6 在主机 H1 上再启动一个 TCP 客户端应用程序建立与 TCP 服务器的 TCP 连接, 观察新建立的 TCP 连接的三次握手过程中的 TCP 报文的字段取值与本实验步骤 5 中的 TCP 连接建立的三次握手过程中的 TCP 报文字段的取值有哪些字段的取值是相同的。

于 H1 中启动另一个 TCP 客户端应用程序 Client2, 如下:

```
Client2 [Java Application] /usr/lib/jvm/jdk1.8.0_231/bin/java (
Hello Client2!
```

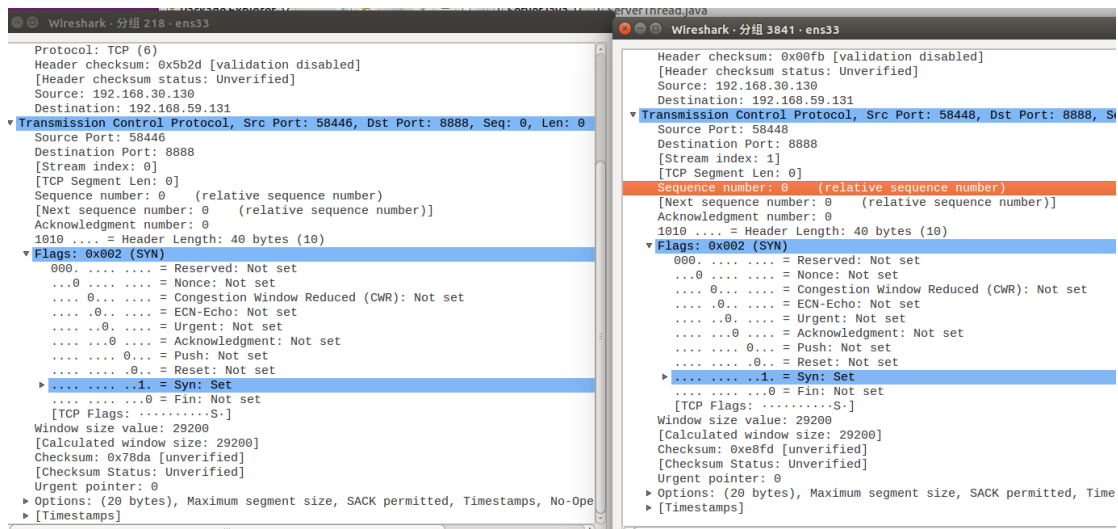
于 H2 的 Wireshark 中成功抓包新的三次握手, 如下:



| No.  | Time            | Source         | Destination    | Protocol | Length | Info                         |
|------|-----------------|----------------|----------------|----------|--------|------------------------------|
| 218  | 168.422035288   | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | 58446 → 8888 [SYN] Seq=0 ... |
| 219  | 168.422098873   | 192.168.59.131 | 192.168.30.130 | TCP      | 74     | 8888 → 58446 [SYN, ACK] S... |
| 220  | 168.422797348   | 192.168.30.130 | 192.168.59.131 | TCP      | 66     | 58446 → 8888 [ACK] Seq=1 ... |
| 3841 | 3520.1267263... | 192.168.30.130 | 192.168.59.131 | TCP      | 74     | 58448 → 8888 [SYN] Seq=0 ... |
| 3842 | 3520.1268244... | 192.168.59.131 | 192.168.30.130 | TCP      | 74     | 8888 → 58448 [SYN, ACK] S... |
| 3843 | 3520.1273825... | 192.168.30.130 | 192.168.59.131 | TCP      | 66     | 58448 → 8888 [ACK] Seq=1 ... |

下面按三次握手划分，将两次 TCP 连接建立的报文进行两两比对：

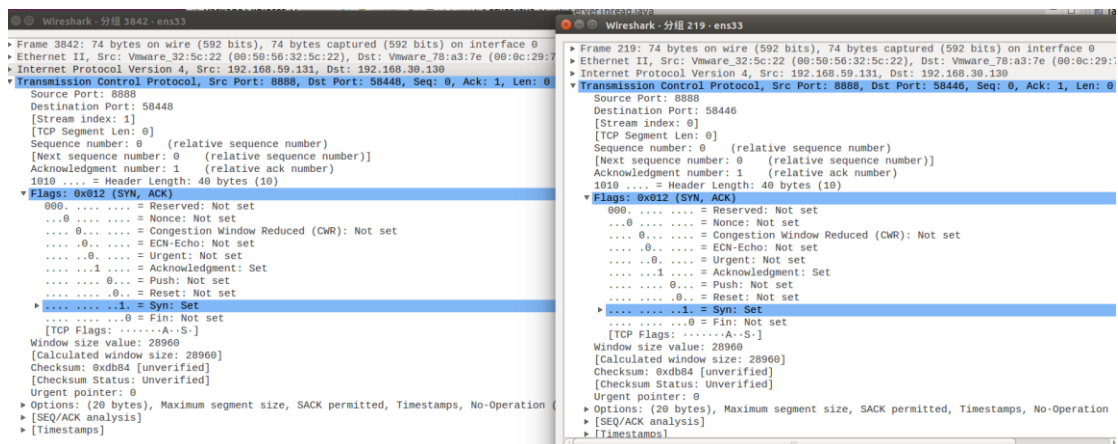
### 第一次握手：



可见，二者目的端口号、TCP Segment Len、相对序列号、相对下一序列号、确认号、Header Length、Flags 的所有字段取值（如 SYN、ACK）、Windows size value、Calculated window size、Checksum Status、Urgent pointer 的取值都相同。

而二者的源端口号不同，Stream index 也不同。

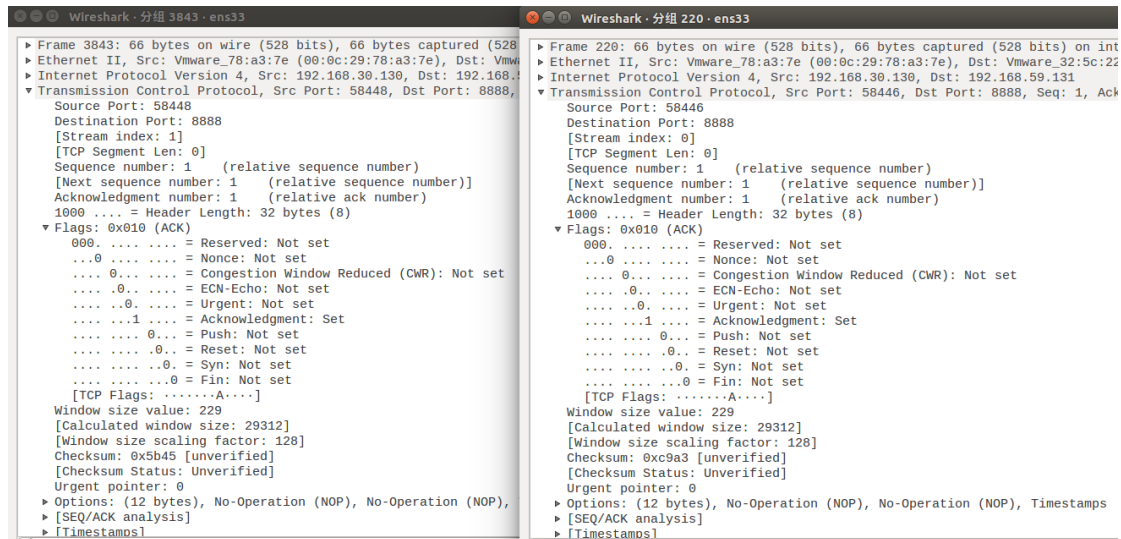
### 第二次握手：



可见，二者目的端口号、TCP Segment Len、相对序列号、相对下一序列号、相对确认号、Header Length、Flags 的所有字段取值（如 SYN、ACK）、Windows size value、Calculated window size、Checksum Status、Urgent pointer 的取值都相同。

而二者的源端口号不同，Stream index 也不同。

### 第三次握手：



可见，二者目的端口号、TCP Segment Len、相对序列号、相对下一序列号、相对确认号、Header Length、Flags 的所有字段取值（如 SYN、ACK）、Windows size value、Calculated window size、Checksum Status、Urgent pointer 的取值都相同。

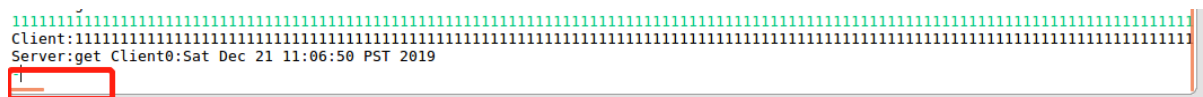
而二者的源端口号不同，Stream index 也不同。

综上可见，在两次 TCP 连接建立的过程中，TCP 报文的源端口号、Stream index、真实序列号、真实下一序列号、真实下一确认号及 Checksum 可能会产生不同。

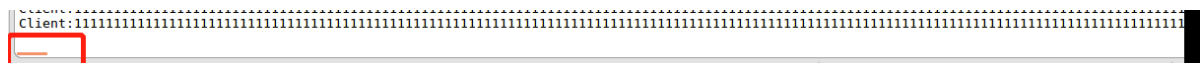
4.7 TCP 连接建立后，触发 TCP 应用程序一次发送数据的数据量大于链路数据包最大长度，通过 Wireshark 抓包分析与此对应的 TCP 连接上 TCP 报文的收发，判断 TCP 协议是否对发送的数据进行了分段处理，分析这些 TCP 报文的序列号（Sequence Number）字段、确认号（Acknowledgement Number）字段的取值。

下面尝试使用客户端向服务器发送长达 15841 字节的数据（用“1”填充），来判断 TCP 协议是否对发送的数据进行了分段处理。

客户端发送情况如下（观察红框中的拖动条可知有很多“1”）：



服务器接收情况如下（观察红框中的拖动条可知有很多“1”）：



注意：本次实验所编写的客户端程序与服务器程序的流程是：客户端向服务器发送信息，服务器接收信息后显示到命令行中，并向客户端发送接收信息（客户端名+get+当前时间）。

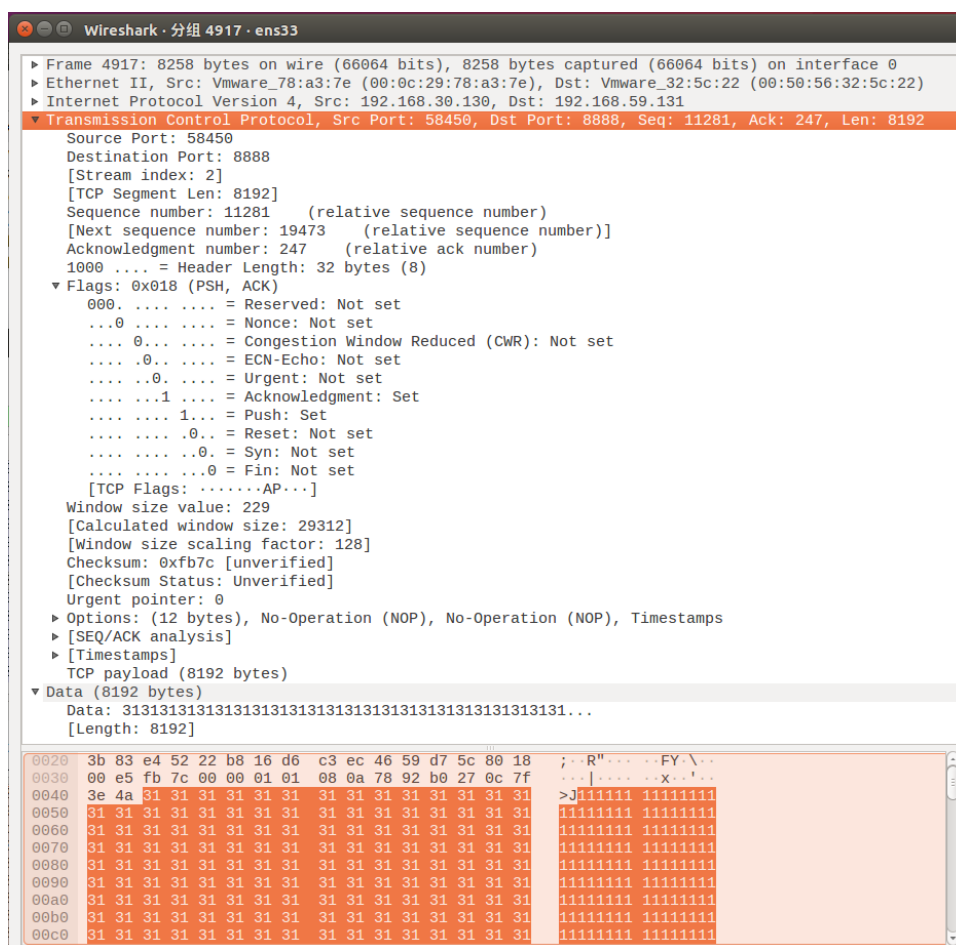
H2 中 Wireshark 抓包情况如下:

|      |                 |                |                |     |                                 |
|------|-----------------|----------------|----------------|-----|---------------------------------|
| 4917 | 4462.0923811... | 192.168.30.130 | 192.168.59.131 | TCP | 8258 58450 → 8888 [PSH, ACK]... |
| 4918 | 4462.0924016... | 192.168.59.131 | 192.168.30.130 | TCP | 66 8888 → 58450 [ACK] Seq=...   |
| 4919 | 4462.0935830... | 192.168.30.130 | 192.168.59.131 | TCP | 7715 58450 → 8888 [PSH, ACK]... |
| 4920 | 4462.0936183... | 192.168.59.131 | 192.168.30.130 | TCP | 66 8888 → 58450 [ACK] Seq=...   |
| 4921 | 4462.0950631... | 192.168.59.131 | 192.168.30.130 | TCP | 107 8888 → 58450 [PSH, ACK]...  |
| 4922 | 4462.0973293... | 192.168.30.130 | 192.168.59.131 | TCP | 66 58450 → 8888 [ACK] Seq=...   |

根据红框可见，该次数据被 TCP 协议分成两段发送。在服务器接收完毕后，向客户端发送接受信息，客户端返回 ACK（即 4921 与 4922 号帧）。

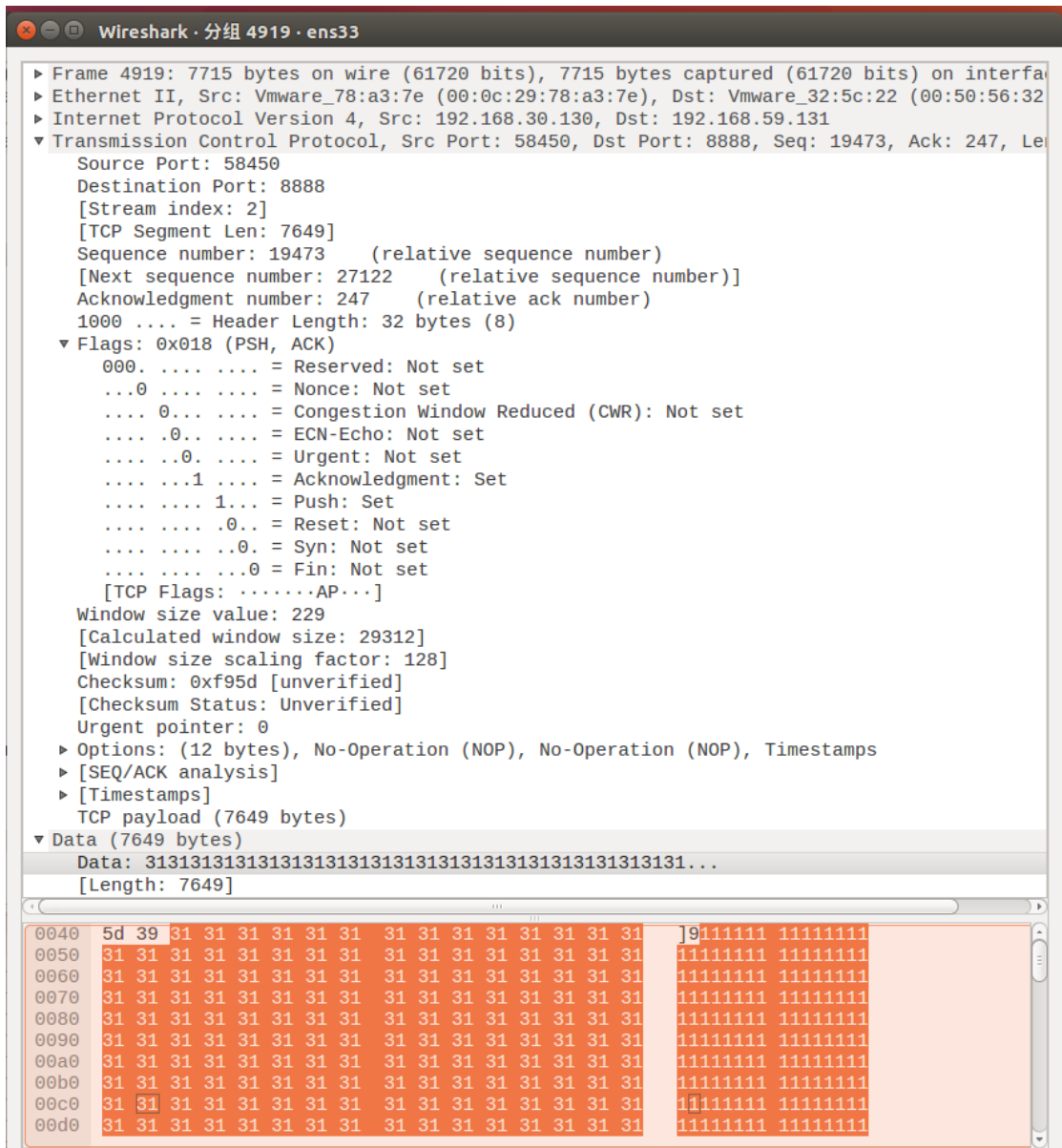
由于之前经过多次测试，所以相对序列号开始时已经大于 1 万，但不妨碍分析。

对这两个分段发送的 TCP 报文进行分析，首先看 4917 号帧：



由于前面已经发送了 11280 长度的数据，故本次序列号从 11281 开始，下一个序列号为 19473,  $19473-11281=8192$ ，恰好为本包携带的数据长度。而确认号按规律是 247。

随后是 4919 号帧:



可见, 其序列号为 4917 中的“下一序列号”19473, 且它的下一序列号是 27122, 二者做减法即是数据长度 7649 字节。

由于是一个数据包进行分段处理，故 ACK 是相同的。

数据分段中的序列号可以保证所有传输的数据按照正常的次序进行重组,而且通过确认保证数据传输的完整性。例如,客户端发送了这两个,若服务器只接受到了后半包,则只会发回后半包的确认;客户端未接到 11281 序列号的包的返回确认消息,在超时后则会重发这个包。服务器接到这个包后,根据二者序列号将数据进行组合,返回 ACK,同时将组合好的数据给出。



4.8 关闭 TCP 客户端程序和 TCP 服务器应用程序，通过 Wireshark 抓包分析 TCP 连接拆除过程、TCP 连接拆除过程中交互的 TCP 报文、每个报文的 SYN 比特位、ACK 比特位、FIN 比特位、序列号（Sequence Number）字段、确认号（Acknowledgement Number）字段的取值。

当 H1 中 TCP 客户端应用程序向 H2 中 TCP 服务器应用程序发送信息 “bye” 时，正常结束本次 TCP 连接，程序运行结果如下。

```
<terminated> Client2 [Java Application] /usr/lib/jvm/jdk1.8.0_231/bin/java (
Hello Client2!
bye
Client:bye
Server:bye
```

此时在 H2 的 Wireshark 中抓包得到 TCP 断开连接的四次握手如下：

|    |              |                |                |     |                                 |           |
|----|--------------|----------------|----------------|-----|---------------------------------|-----------|
| 26 | 14.760111654 | 192.168.59.131 | 192.168.30.130 | TCP | 66 8888 → 58490 [FIN, ACK] S... | Co<br>0.2 |
| 27 | 14.761027016 | 192.168.30.130 | 192.168.59.131 | TCP | 66 58490 → 8888 [ACK] Seq=5 ... |           |
| 28 | 14.761847096 | 192.168.30.130 | 192.168.59.131 | TCP | 66 58490 → 8888 [FIN, ACK] S... |           |
| 29 | 14.761912840 | 192.168.59.131 | 192.168.30.130 | TCP | 66 8888 → 58490 [ACK] Seq=6 ... |           |

下面逐步解析这四次握手：

## 第一步：服务器向客户端发送终止数据包

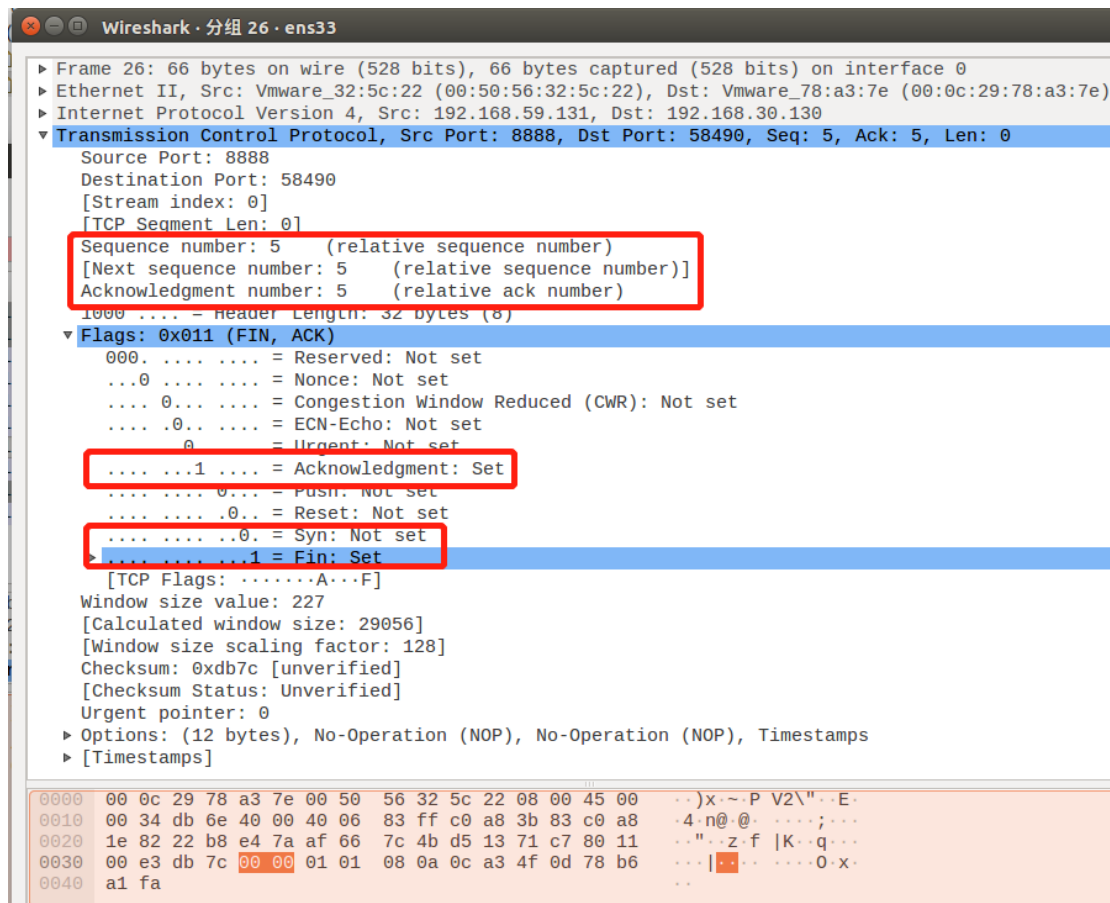
SYN: 0

ACK: 1

FIN: 1

序列号: 5

确认号: 5



服务器完成它的数据发送任务后，会主动向客户端发送一个终止数据包，以关闭在这个方向上的 TCP 连接。

该数据包中，序列号为客户端发送的上一个数据包中的确认号值（5），而确认号为服务器发送的上一个数据包中的序列号（1）+该数据包所带的数据的大小（4 字节）；



## 第二步：客户端向服务器发送确认信息

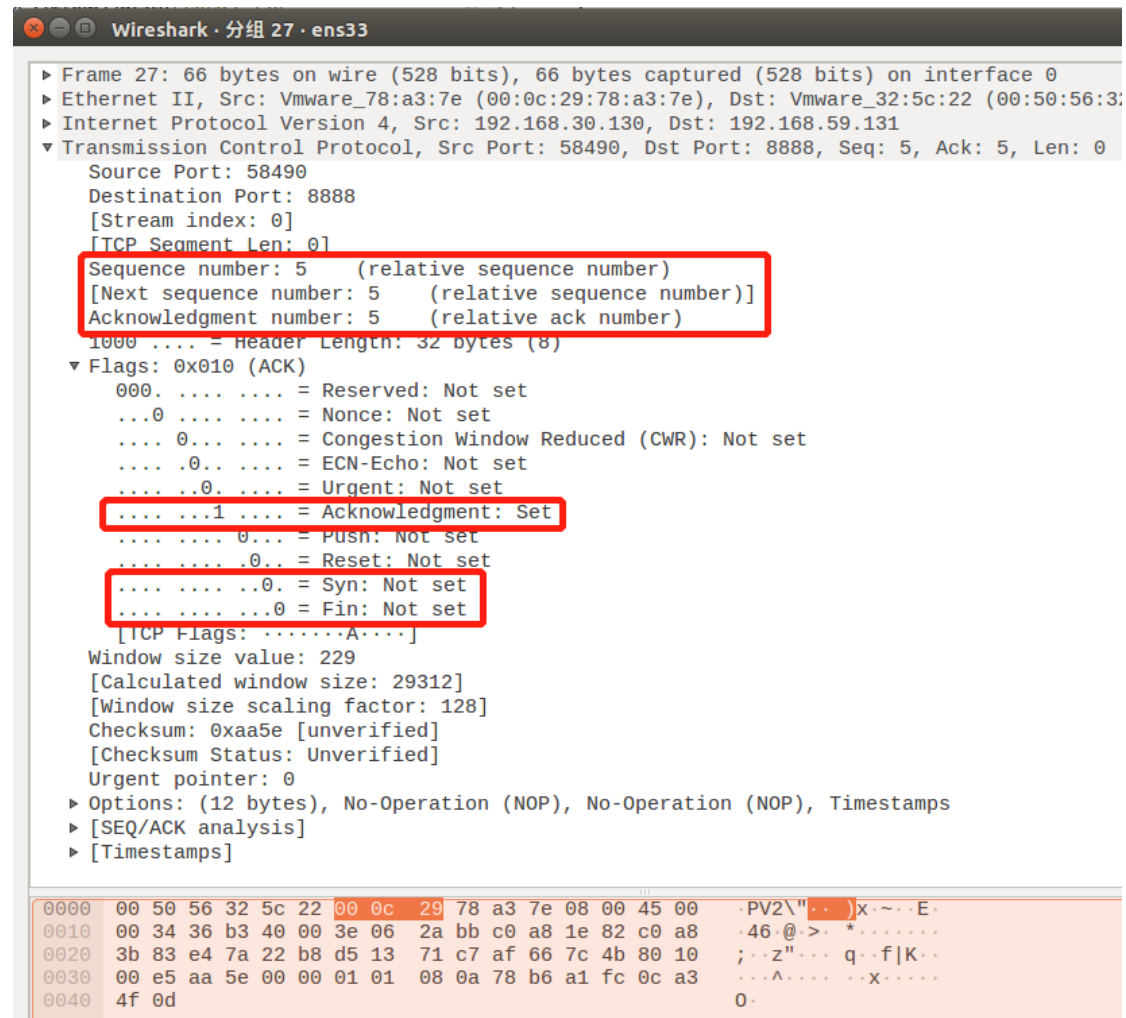
SYN: 0

ACK: 1

FIN: 0

序列号: 5

确认号: 5



客户端收到服务器发送的终止数据包后，将对服务器发送确认信息，以关闭该方向上的 TCP 连接。

这时的数据包中，序列号为第 1 步中的确认号值（5），而确认号为第 1 步的数据包中的序列号（5）；

### 第三步：客户端向服务器发送终止数据包

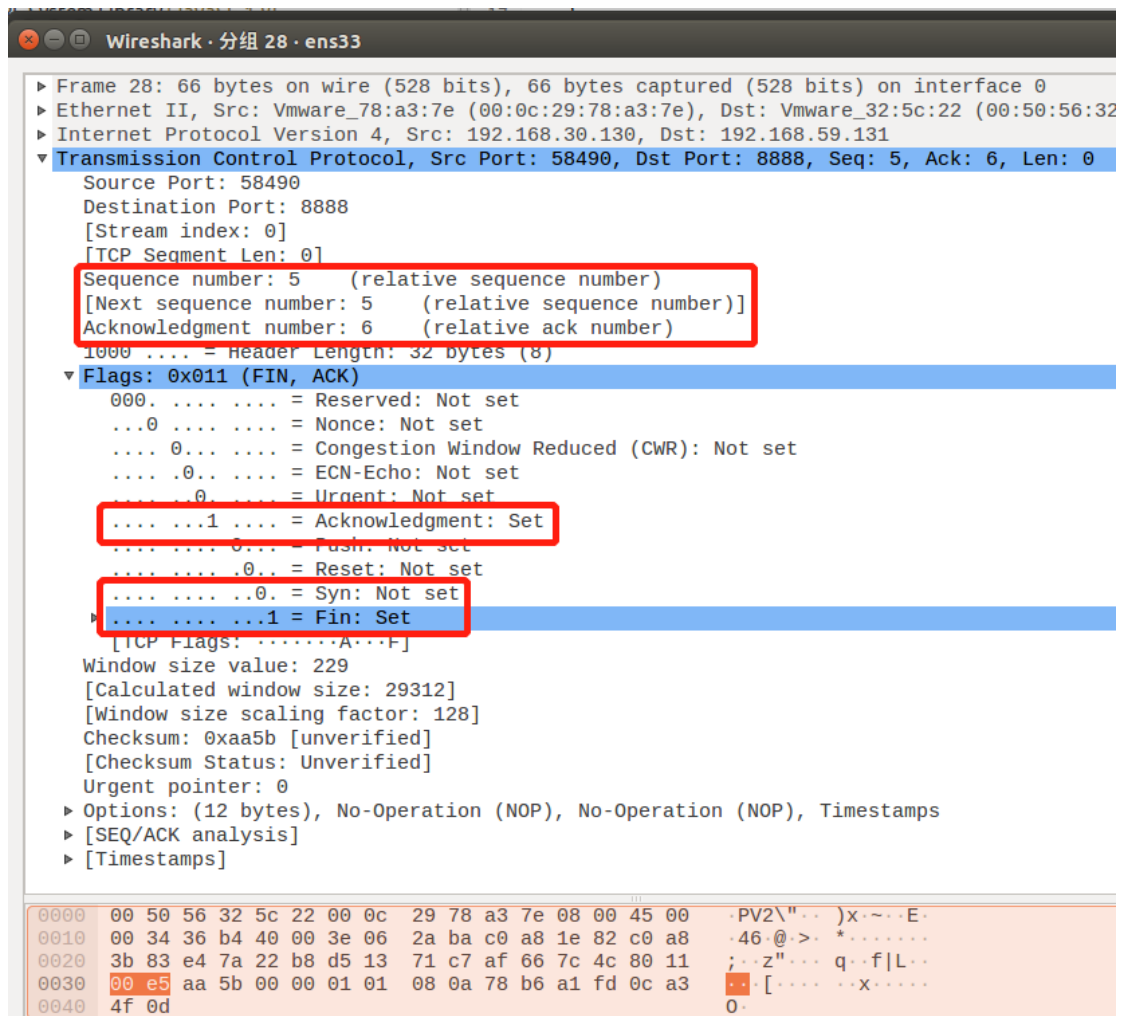
SYN: 0

ACK: 1

FIN: 1

序列号: 5

确认号: 6



同理，客户端完成它的数据发送任务后，就也会向服务器发送一个终止数据包，以关闭在这个方向上的 TCP 连接，该数据包中，序列号为服务器发送的上一个数据包中的确认号值（5），而确认号为客户端发送的上一个数据包中的序列号（5）+该数据包所带数据的大小（0）；

#### 第四步：服务器向客户端发送确认信息

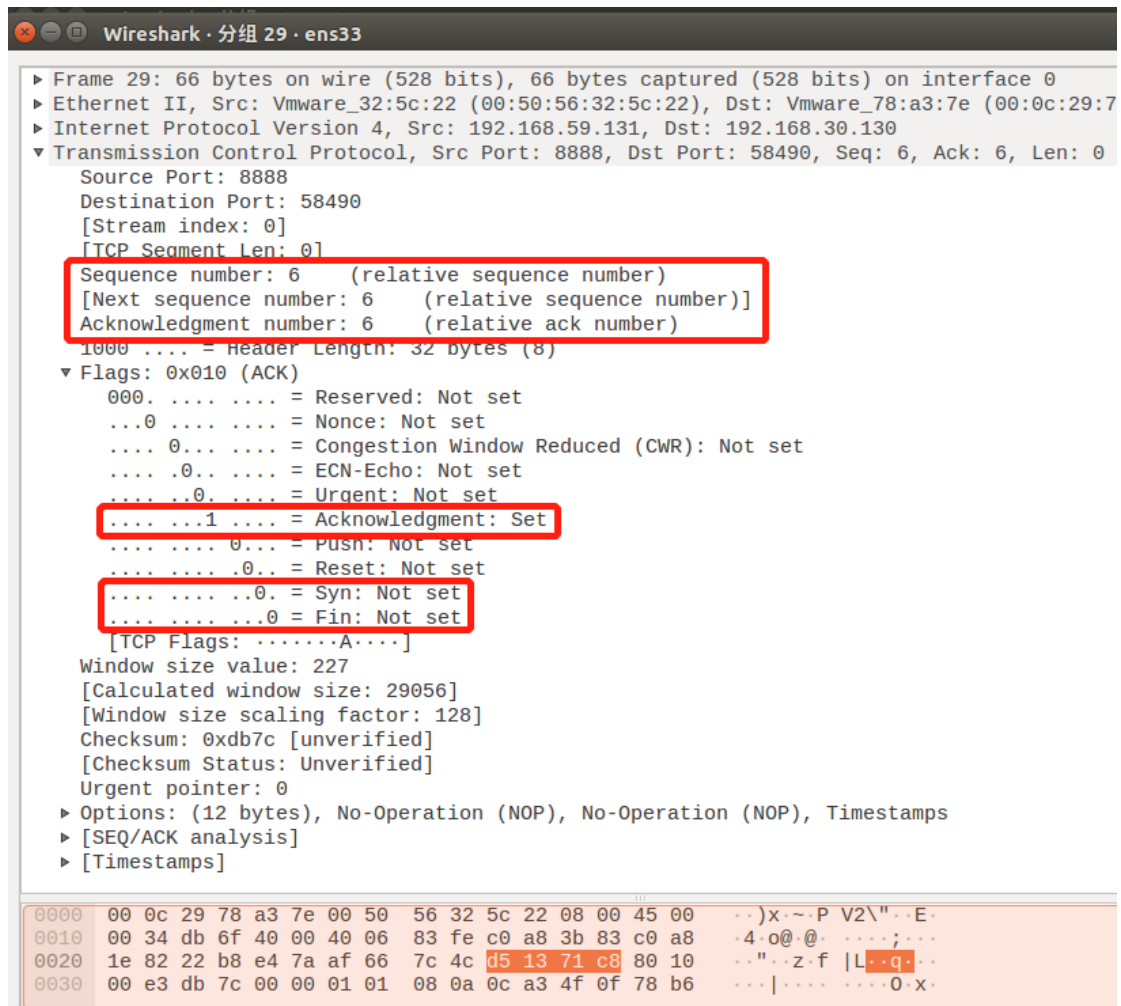
SYN: 0

ACK: 1

FIN: 0

序列号: 6

确认号: 6



服务器收到客户端发送的终止数据包后，将对客户端发送确认信息，以关闭该方向上的 TCP 连接。这时在数据包中，序列号为第 3 步中的确认号值（6），而确认号为第 3 步数据包中的序列号（5）+1；

#### 总结：

为了释放一个连接，任何一方都可以发送一个设置了 FIN 标志位的 TCP 段，这表示它已经没有数据要发送了。当 FIN 段被另一方确认后，这个方向上的连接就被关闭，不再发送任何数据。然而，另一个方向上或许还在继续着无限的数据流。当两个方向都关闭以后，连接才算彻底被释放。

## 五、心得体会

通过本次实验，我了解并掌握了 TCP 连接开始的三次握手、中间传输和结束的四次挥手的相关流程，也因此懂得了服务器端和客户端状态的变化。

同时，对于 TCP 报文中的信息如源端口、目的端口、序列号、确认号、SYN、ACK、FIN 等值也有了更深的理解。

此外，对 linux 中的过滤规则设置有了了解，也对如何在 linux 下安装 eclipse 并使用 java 进行 socket 编程掌握得更好了。