实验3：TCP、UDP协议分析

# 1 实验目的

（1）捕获传输层数据包，进行协议分析。

// 我们通过一个简单的UDP应用程序和一个简单的TCP应用程序

（2）使用教材第二章提供的最basic的、python版本的TCP/UDP client-server 程序，进行实验场景构建，hostos运行client、guestos运行server，或反之。

a. 如果你更熟悉其他语言，如C、C++、Java等，也可以使用其他语言的socket程序，哪门语言不重要。因为最终都要使用操作系统的TCP/IP协议栈。

# 2 实验内容及提交

## 分析TCP header各字段。



Source Port: 10942//请求端端口：10942

Destination Port: 12345//服务器端端口：12345

[Stream index: 0] //tcp流序号：0（wireshark中对源于同一tcp流的包的标记）

[TCP Segment Len: 13] //tcp报文长度：13

Sequence Number: 1(relative sequence number) //报文序列号：1（相对序号）

Sequence Number (raw): 3199161048//报文序列号：3199161048（绝对序号）

[Next Sequence Number: 14(relative sequence number)] //下一序列号：14

Acknowledgment Number: 1(relative ack number) //确认号：1（相对序号）

Acknowledgment number (raw): 3259045990//确认号：3259045990（绝对序号）

0101 … = Header Length: 20 bytes (5) //报文头部长度（数据偏移）：20bytes

Flags : 0x018 (PSH,ACK) //报文类型：PSH,ACK

000… = Reserved: Not set //保留字段

…0… = Nonce: Not set//随机数：无效（随机数(Nonce)是任意的或非重复的值，它包括在经过一个协议的数据交换中，通常为保证活跃度以及避免受重复攻击）

…0… = Congestion window Reduced (CWR): Not set

//拥塞窗口减少戳：无标记（TCP拥塞控制）

…0… =ECN-Echo: Not set //显式拥塞戳:无效（ECN-Echo与TCP拥塞控制）

…0… = Urgent: Not set //紧急指针戳：无效

…1… = Acknowledgment: Set //确认号戳：有效

…0… = Push: Set //推送戳:有效

…0… = Reset: Not set //复位戳：无效

…0. = Syn: Not set //同步戳:无效

…0 = Fin: Not set //结束戳:无效

[TCP Flags …… AP……]

window: 4106 //窗口大小：4106（TCP接收者缓冲的字节大小）

[ Calculated window size: 1051136] //计算出的窗口大小（窗口大小单位\*窗口大小）

[window size scaling factor: 256] //窗口大小换算系数

Checksum: Oxd6a1 [unverified] //校验和

[Checksum Status : Unverified] //校验状态

Urgent Pointer: 0（如果设置了URG位，这个域将被检查作为额外的指令，告诉CPU从数据包的哪里开始读取数据）

[Timestamps]//时间戳

[Time since first frame in this TCP stream: 5.837835000 seconds]// 从此TCP流中的第一帧开始的时间：5.837835000秒

[ Time since previous frame in this TCP stream: 5.837078000 seconds]// 从此TCP流中的上一帧开始的时间：5.837078000秒

## 分析 TCP 的连接建立、释放的报文交互

1. 注：连接释放，不一定是4个报文，只相信你看到的，并且对此作出解释。

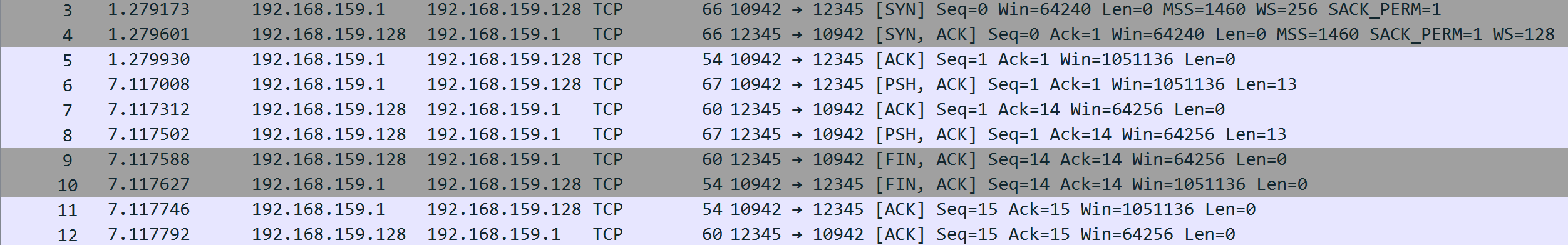
### TCP连接的建立

整个连接的建立，交互及关闭的包如图所示：

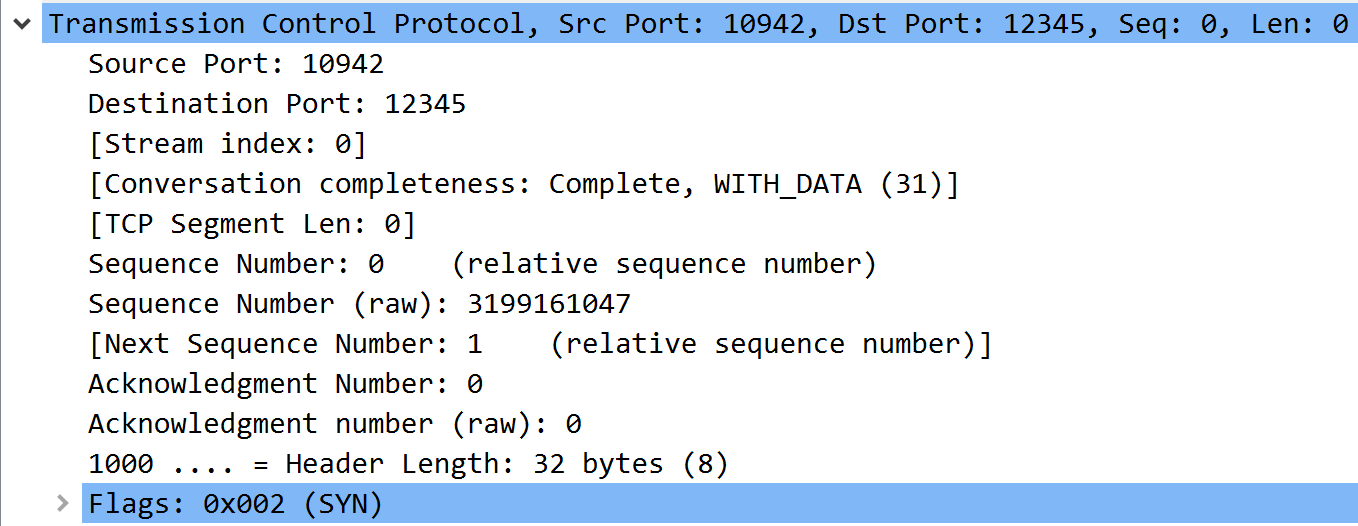
红色：TCP三次握手的建立

蓝色：TCP报文交互（输入数据并返回）

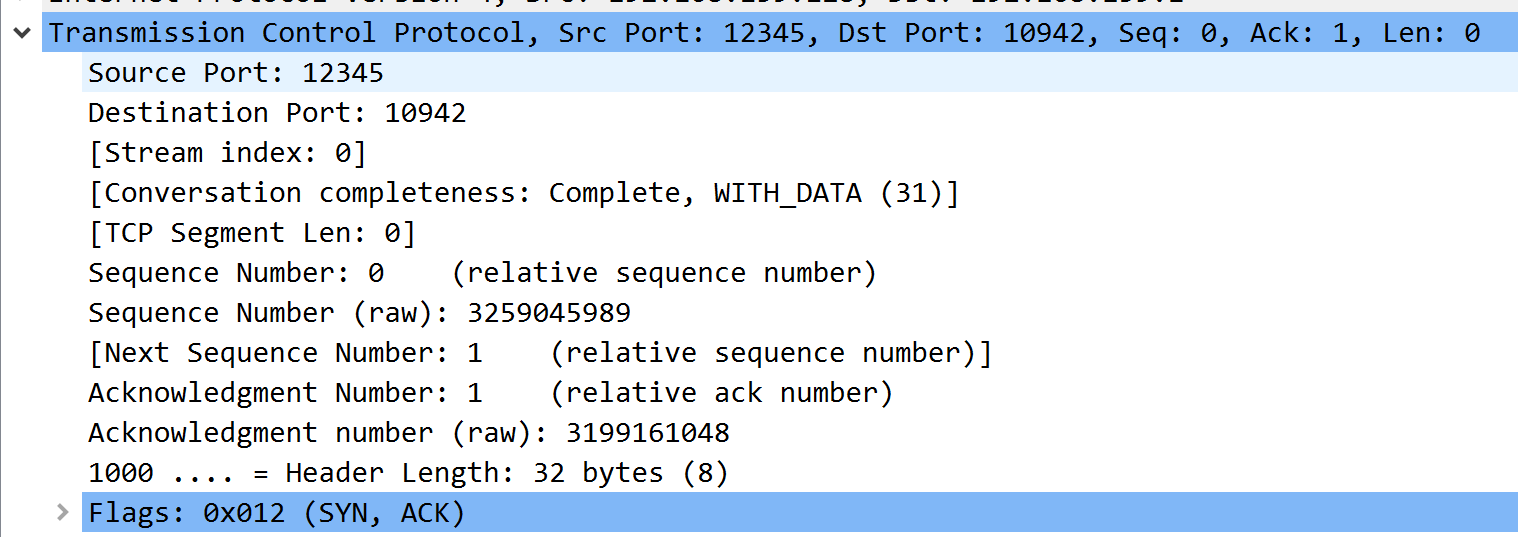
黄色：TCP四次挥手，客户端断开连接



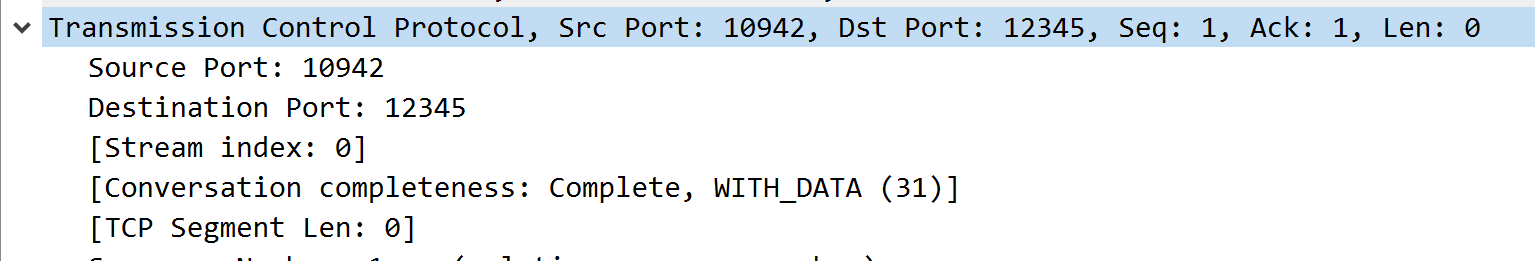
第一次握手由客户端发出，可以看到，客户端将标志位 SYN 置为 1 ，随机产生一个值SEQ = 0，并将该数据包发送给服务器，等待服务器确认； (此处WireShark为了便于分析，将序列号变成了相对序列号，从0开始)。



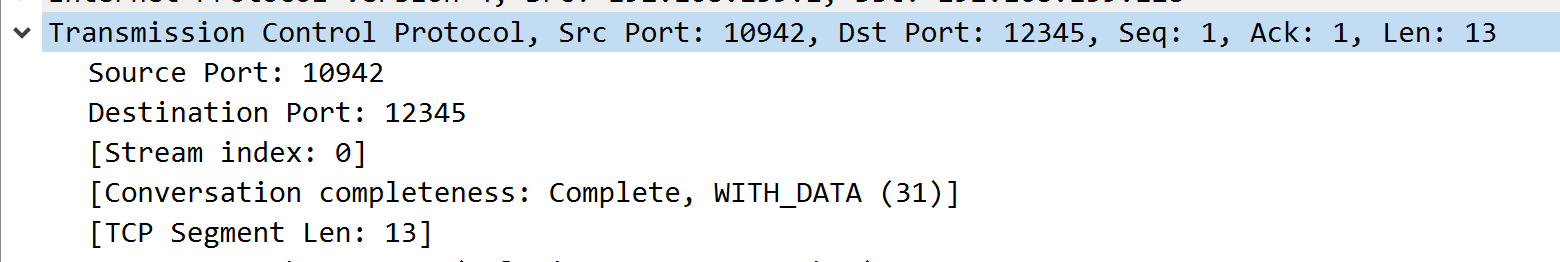
第二次握手由服务端返回，服务器将标志位 SYN 和 ACK 都置为 1 （ACK = X + 1 = 1）随机产生一个值SEQ = Y = 0，并将该数据包发送给客户端以确认连接请求。

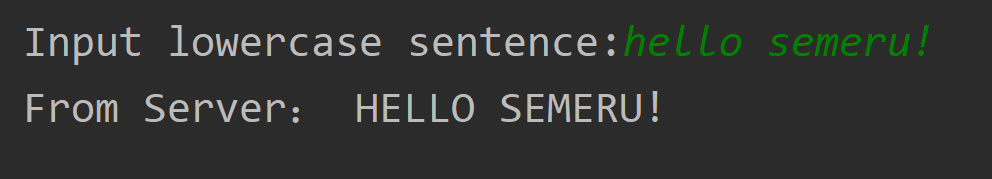


第三次握手：客户端收到确认后，检查 ACK 是否为X + 1= 1，如果正确则将标志位 ACK 置为 1 ，SEQ = 1,ACK = Y + 1 = 1，并将该数据包发送给服务器，服务器检查 ACK 是否为1 ，如果正确则连接建立成功,后续开始传输数据。



三次挥手完毕，CS之间建立连接，发送数据（Len：13）如图，我所发送的数据长度也是13个字节，正确的。



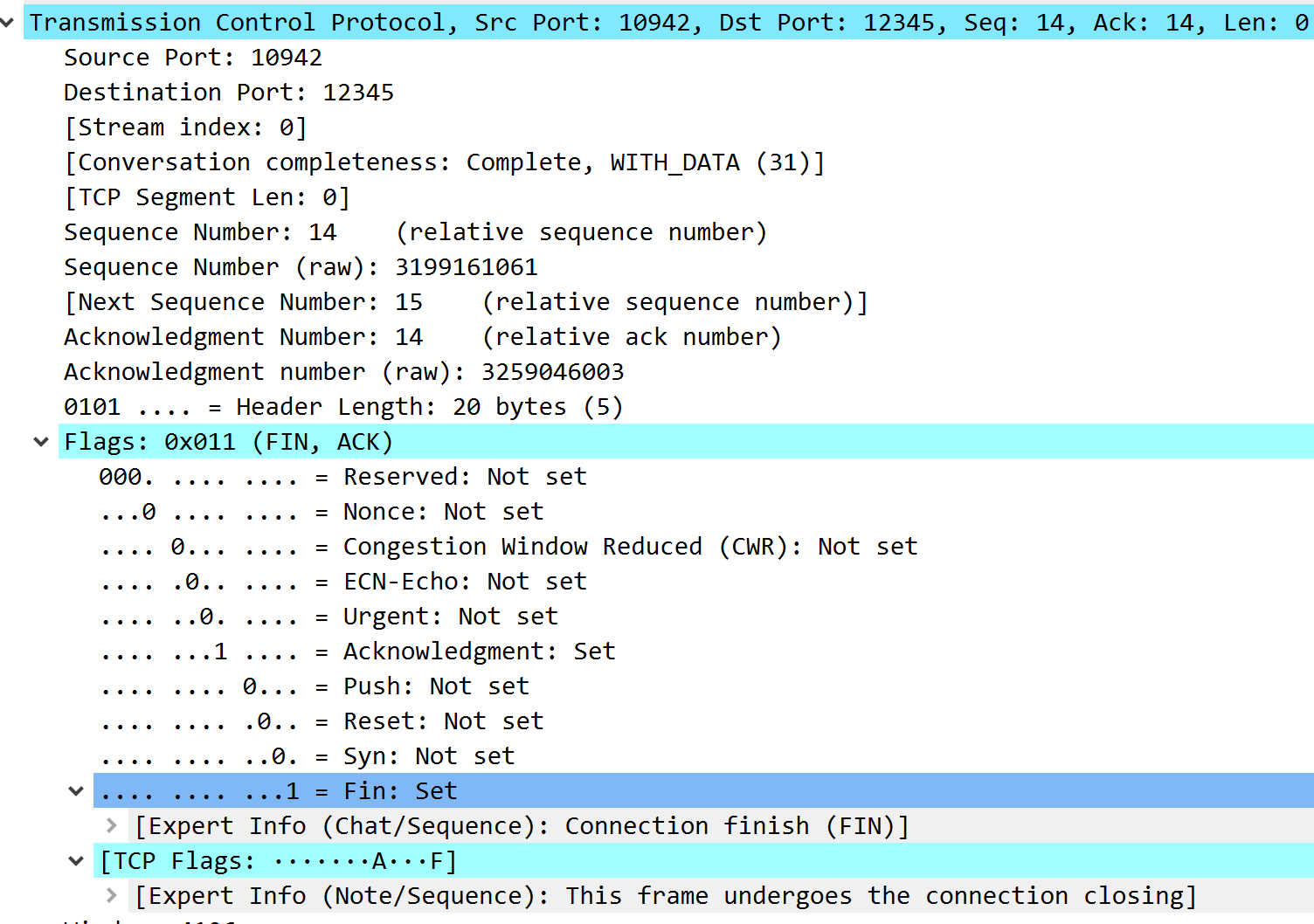


### TCP连接的释放

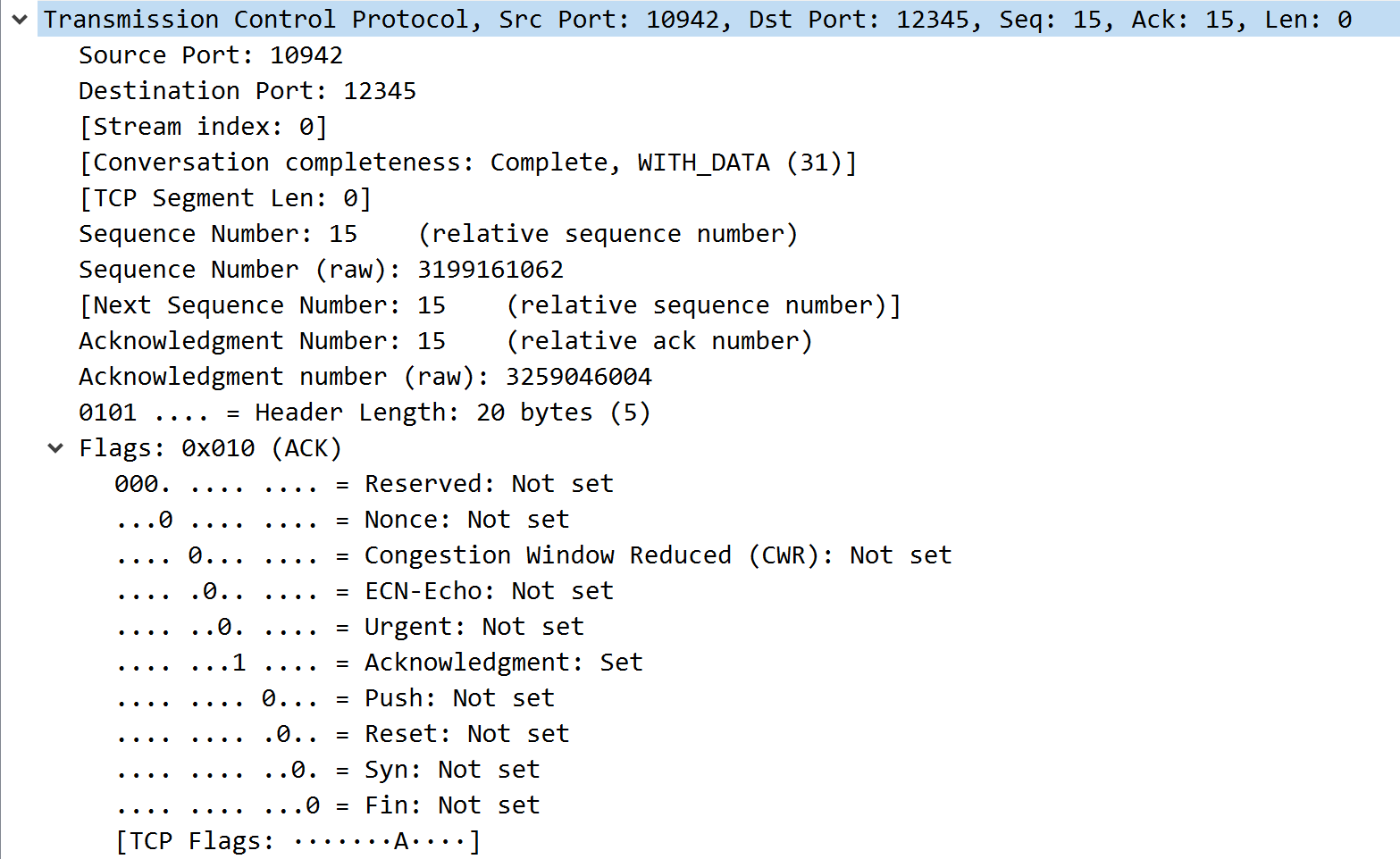
第一次挥手由服务端（主动关闭方）发出发送一个 FIN = 1 ，用来关闭主动方到被动关闭方的数据传送.



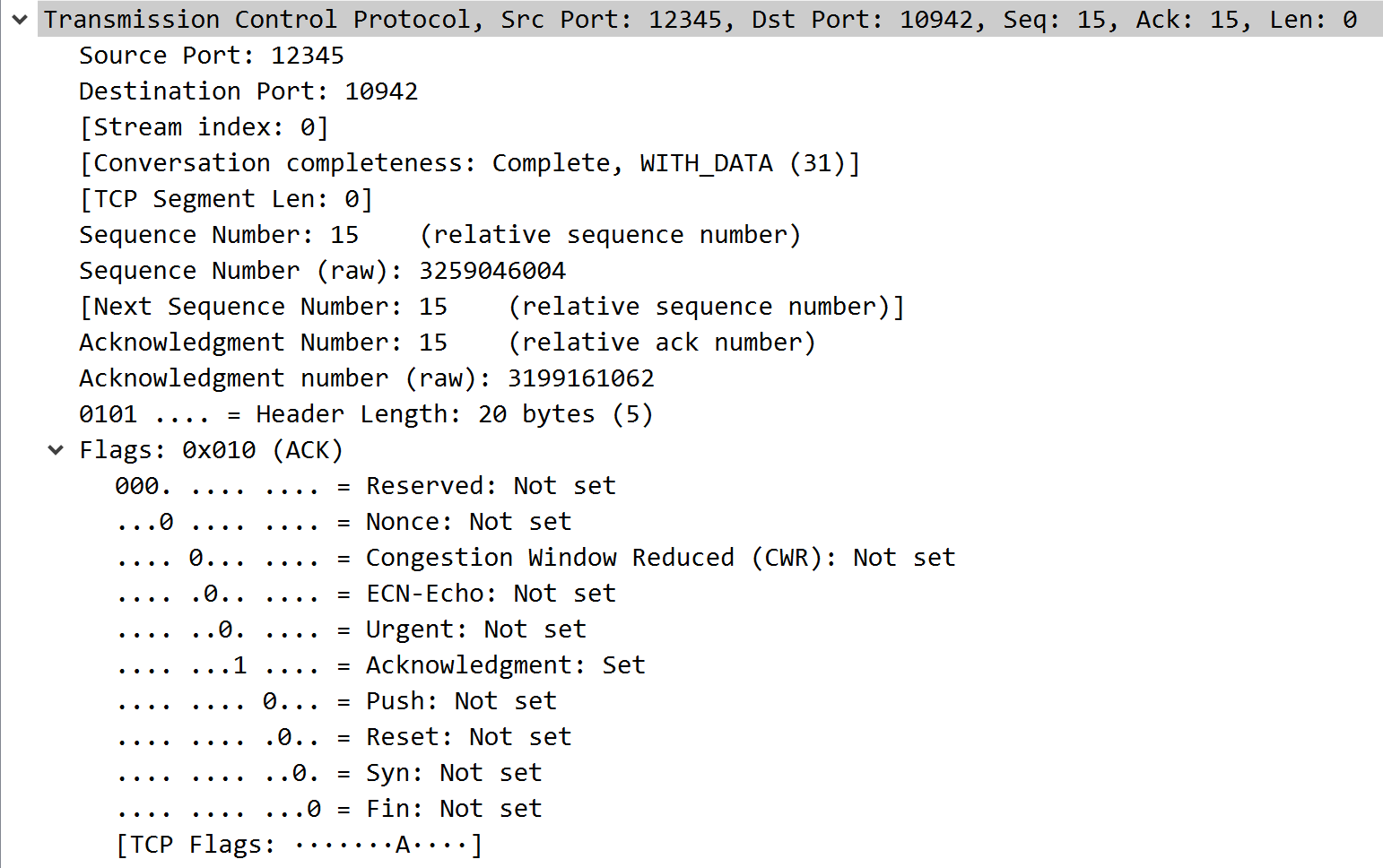
第二次挥手：客户端（被动关闭方）收到 FIN = 1包后，发送一个 ACK = 1给对方，确认序号为:收到报文序号Seq +收到报文所携带数据长度len+ 1=1 。上一个报文可能“捎带”了主动关闭方发送的最后一块数据,其长度用字段len来表示。



第三次挥手：被动关闭方发送一个 FIN = 1 ，同时ACK=1，用来关闭被动关闭方到主动关闭方的数据传送,序列号和确认序号都+1

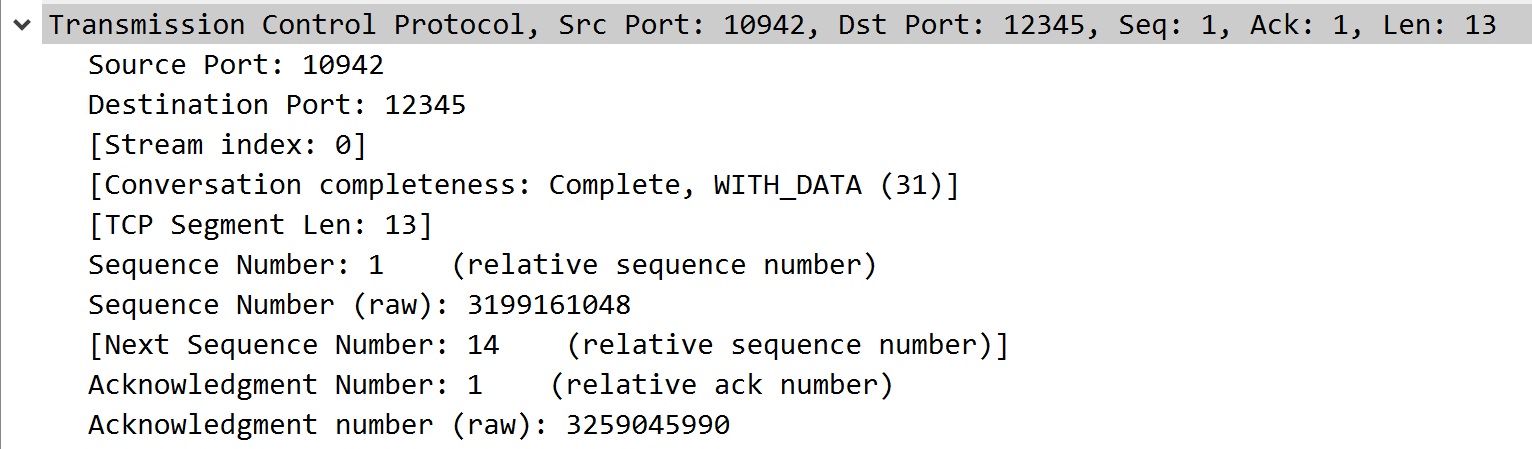


第四次挥手：主动关闭方收到 FIN 后，发送一个 ACK 给被动关闭方，确认序号为：收到报文的序号Seq + 收到报文所携带数据长度len+ 1 ，至此，完成四次挥手，双向连接关闭。

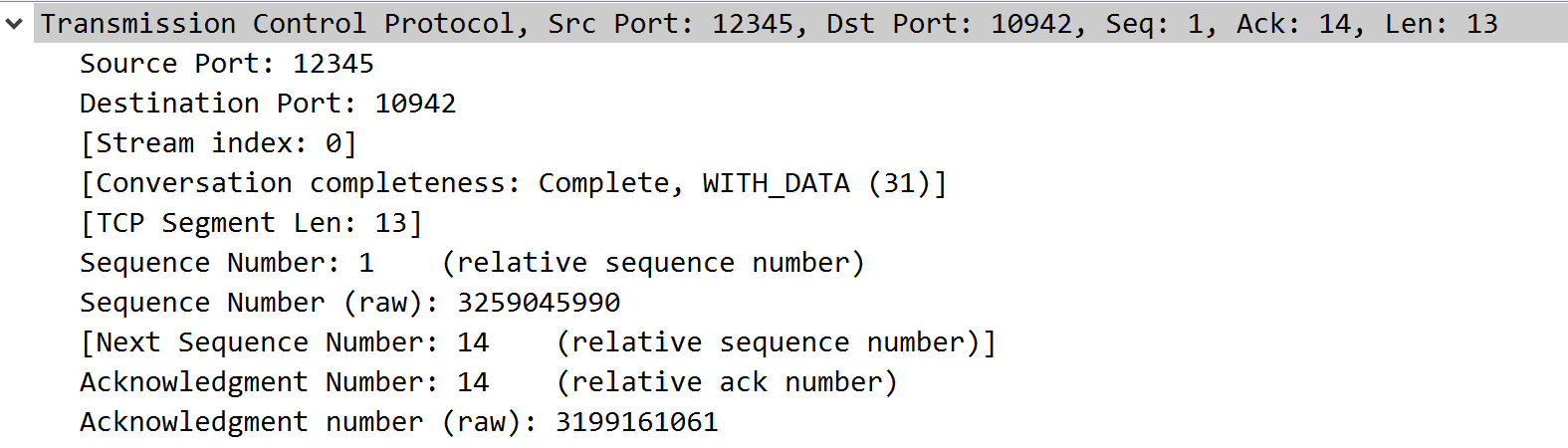


## 跟踪（若干）TCP的数据传输过程中的sequence no.、ack字段。若有，分析TCP重传等。

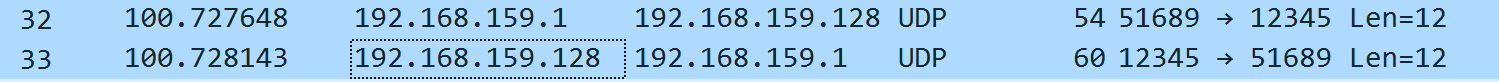
第一次：主机向虚拟机发送数据，sequence no.=1，ack.no=1，并指明了下次返回时的ack字段的值是14

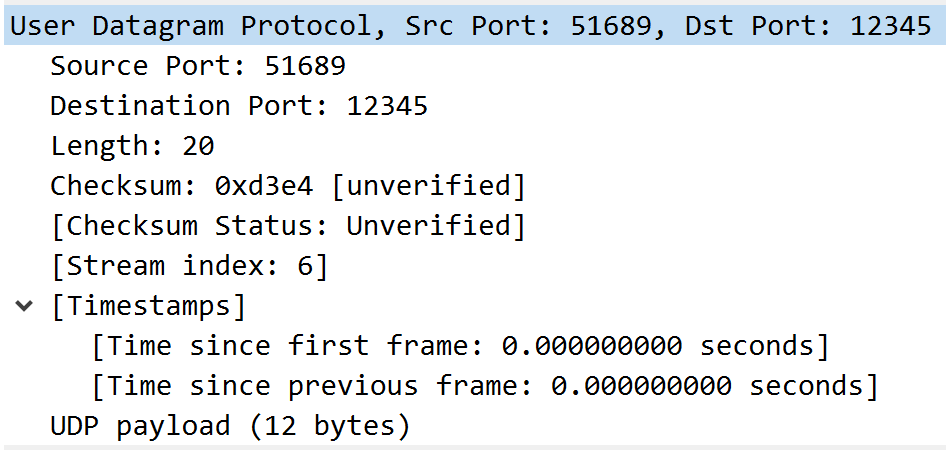


第二次：虚拟机向主机返回数据，同一次传输，序列号是同一个，并且使用了14作为ack.no，与请求报文指定的Next Sequence Number相同。



## 分析UDP header





User Datagram Protocol, Src Port: 52689,Dst Port: 12345

Source Port: 52689//请求方端口：52689

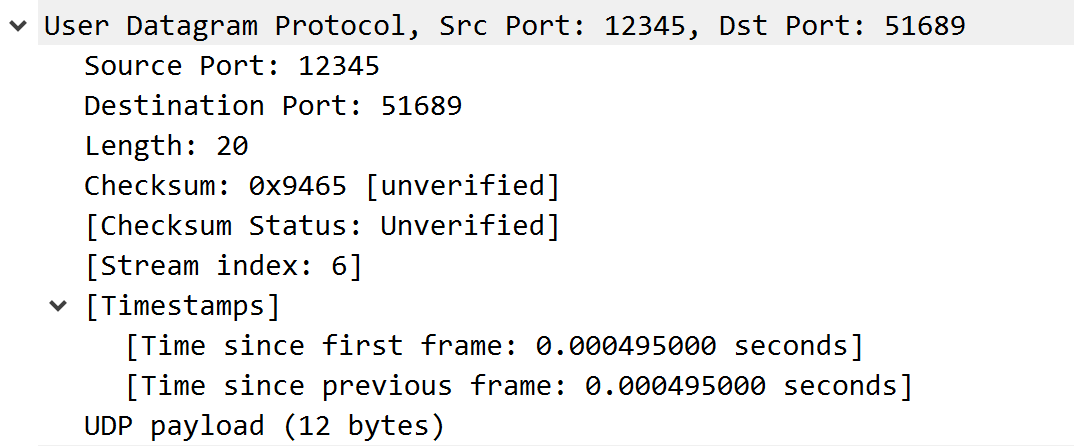
Destination Port: 12345//服务器端口：12345

Length: 20 //header长度：20

Checksum: 0xd3e4 [unverified] //校验和

[checksum Status: Unverified] //校验状态

[Stream index: 6] //流序号：6



User Datagram Protocol, Src Port: 12345,Dst Port: 52689

Source Port: 12345//请求方端口：12345

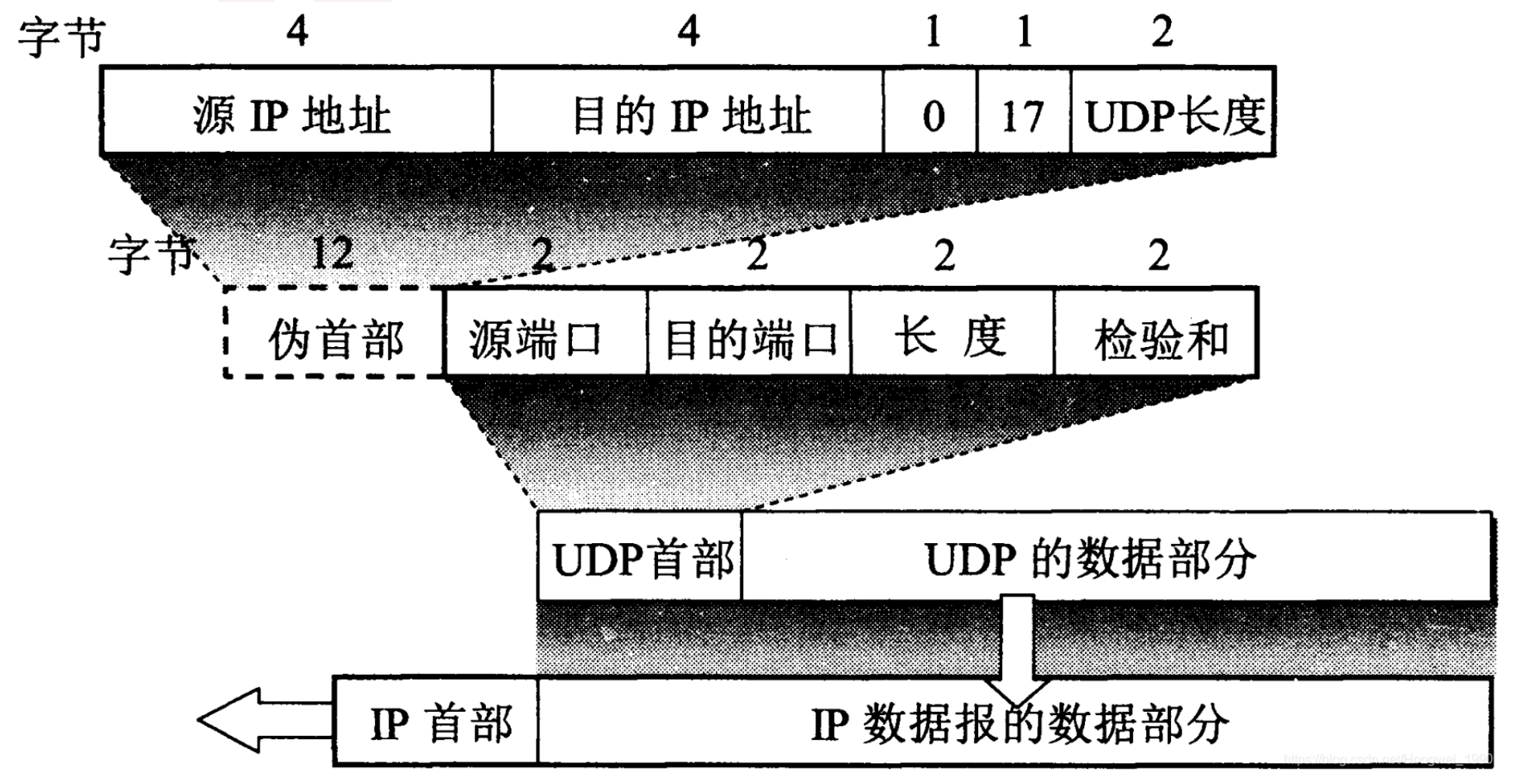
Destination Port: 52689//服务器端口：52689

Length: 20 //header长度：20

Checksum: 0x9465 [unverified] //校验和

[checksum Status: Unverified] //校验状态

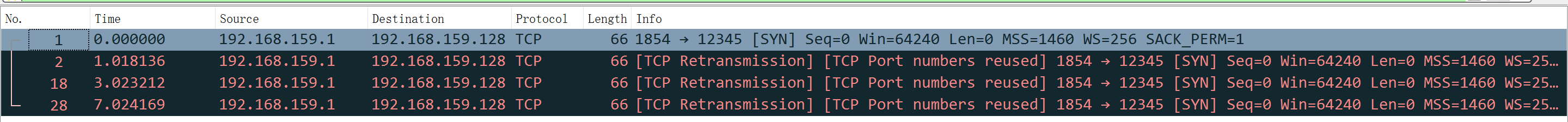
[Stream index: 6] //流序号：6

参考了[UDP 概述（主要特点，首部格式 header）\_CSDN博客](https://blog.csdn.net/Hongwei_1990/article/details/94218912)，找到Length = 20的图解：  


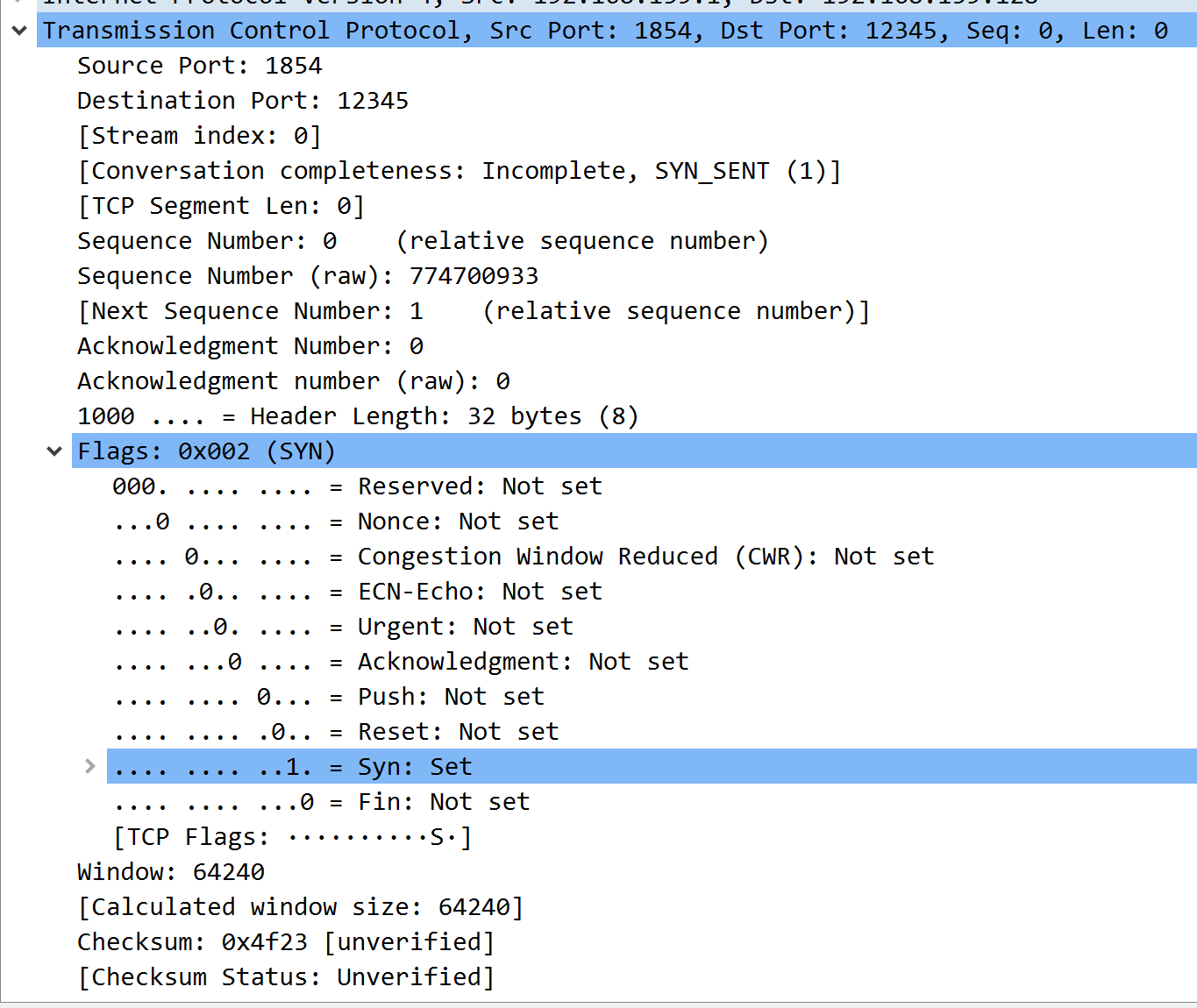
## 分别考察如下情况：

### 在TCP server关闭时，打开TCP client,就所捕获的数据包s,观察并得出什么结论？

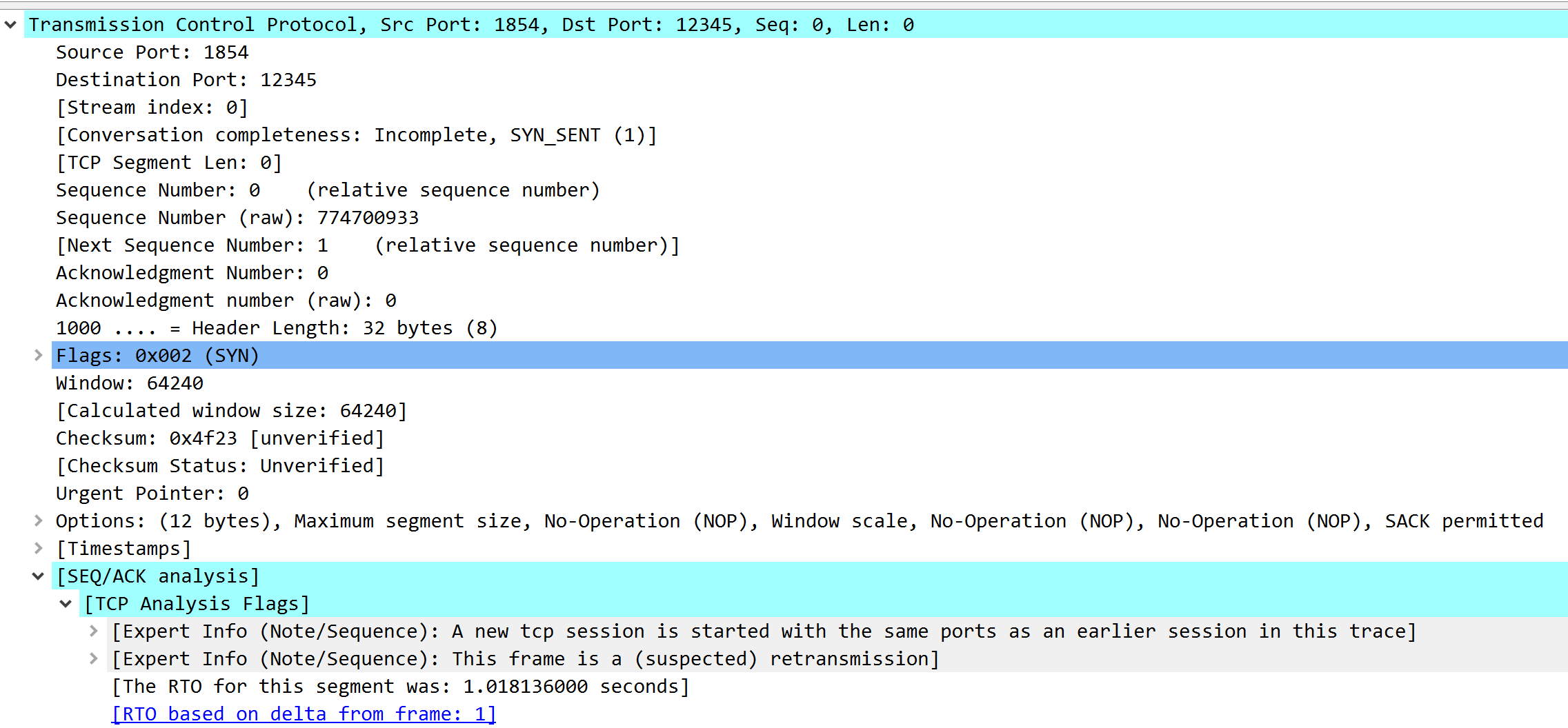
捕获的数据包



Client首先向Server 发送一个正常的TCP连接报文



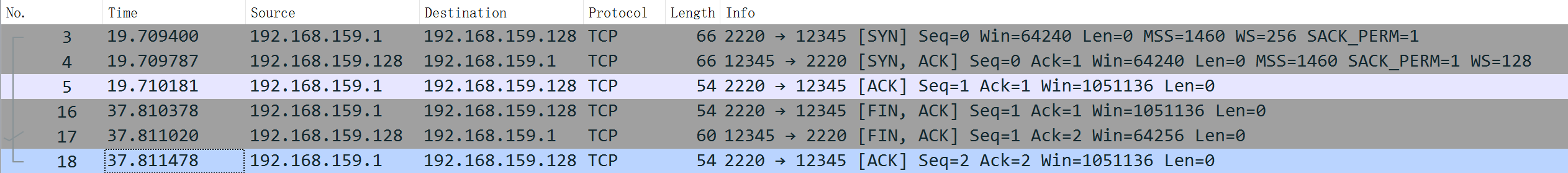
当没有收到Server的回复时，Client又连续发送了3个重传的TCP包，之后都没有收到回复后，就不再向Server 端发送报文了。重传报文的内容，头部等信息与第一次发送的报文一致，从info上都显示为retransmission报文，在红框所标注出的内容中，也标注出Client 从与第一次发送TCP报文相同的端口又重新发送了报文。



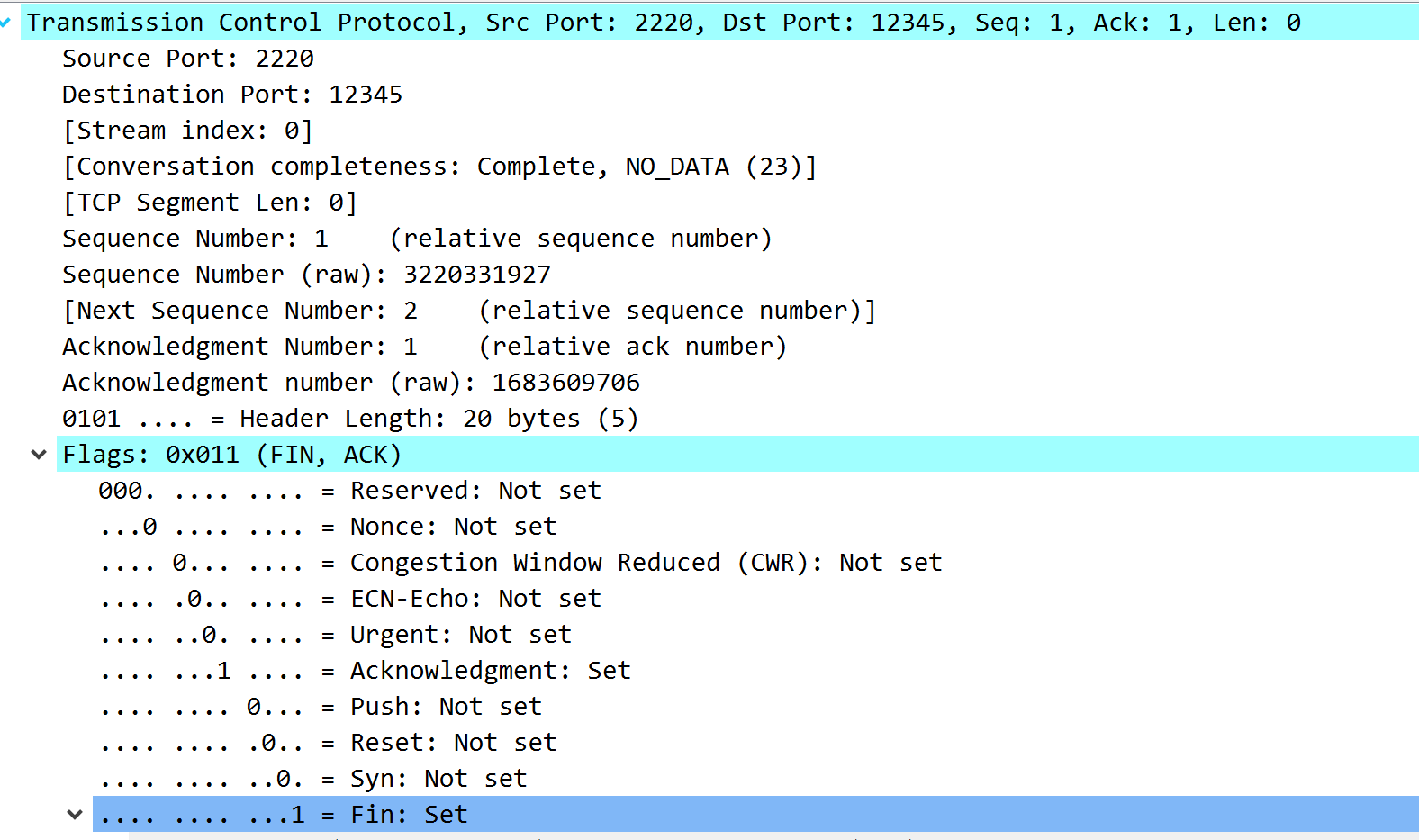
从Client端的表述来看，也说明了连接不到Server 就自动断开连接的问题



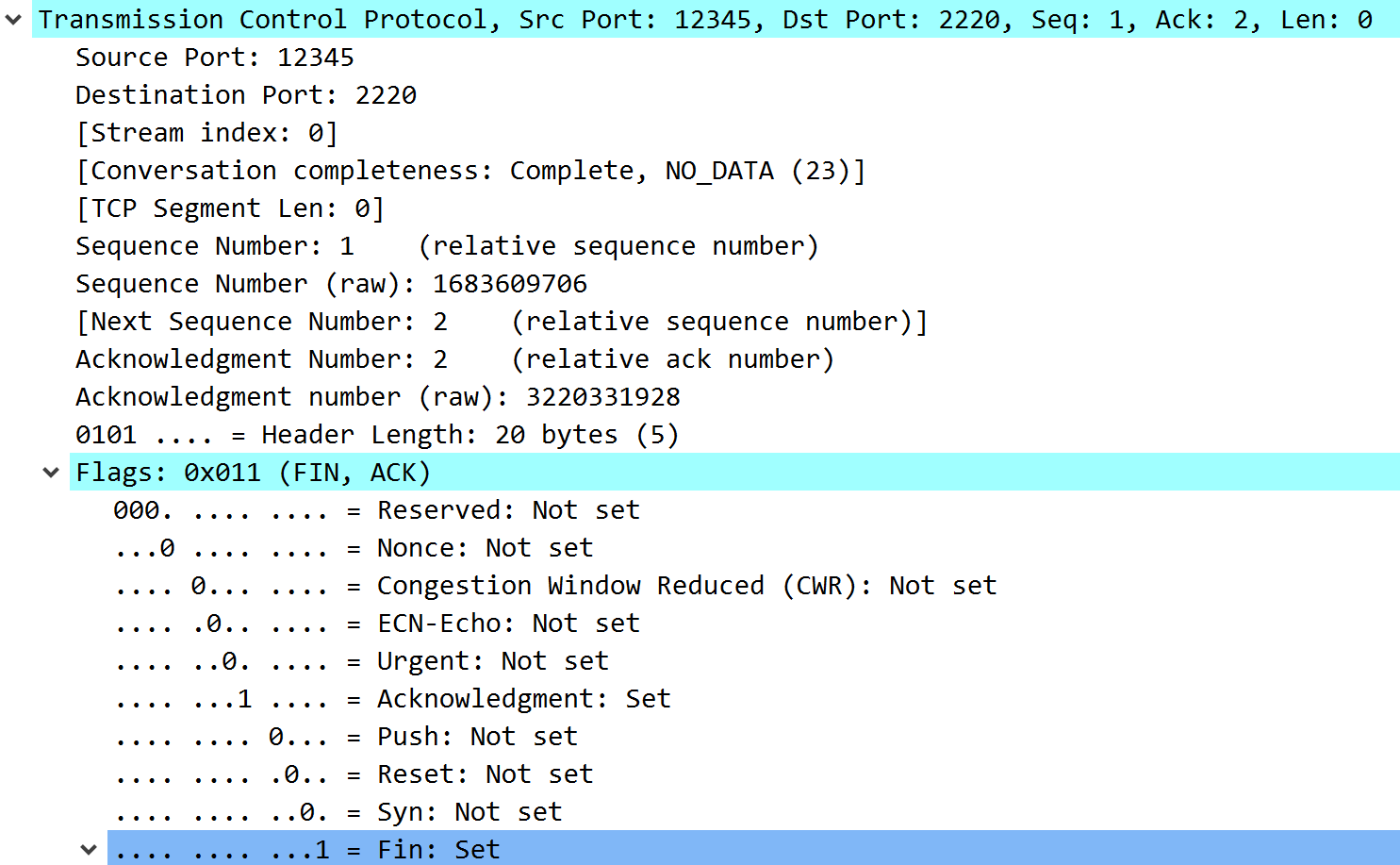
### TCP client--server处于正常交互状态时，非正常关闭client,就所捕获的数据包s,观察并得出什么结论？



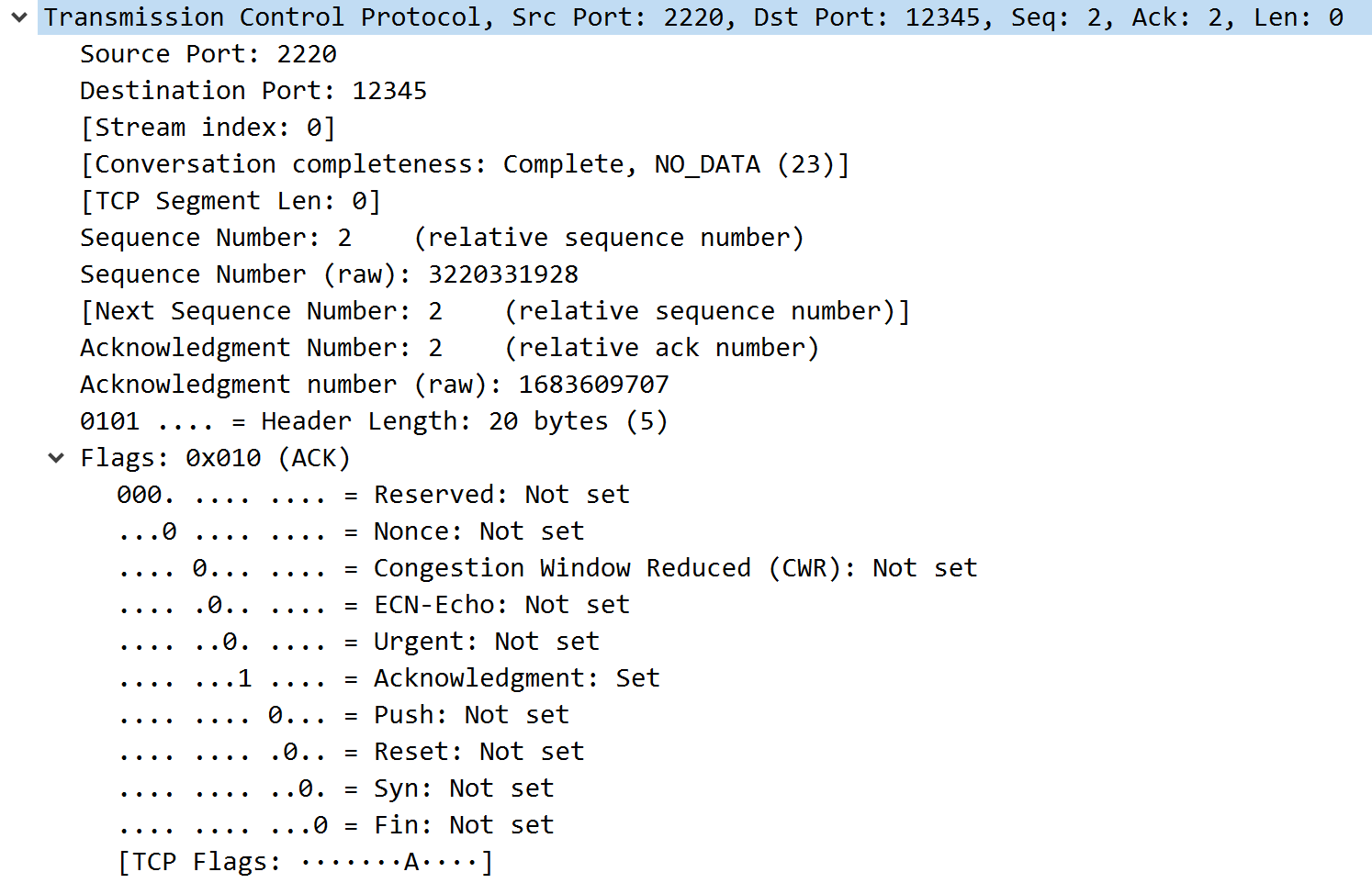
发现只有3次挥手，客户端先向服务器发送FIN报文，并设置ACK = 1，并未同步；



服务端收到后，使用相同的Sequence number 回复客户端，并将ACK + 1，但没有同步；



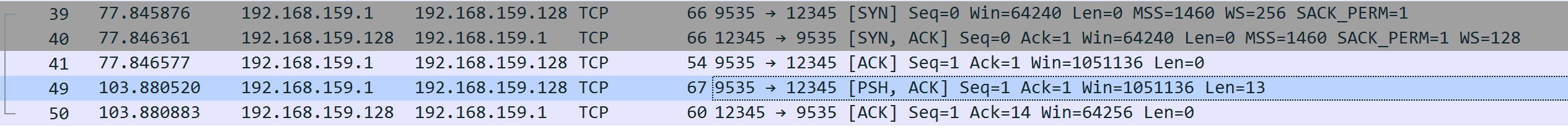
客户端收到后，知道自己的发送没有问题，对方的发送和接收没有问题，再向服务器发送回复报文后即主动关闭连接。



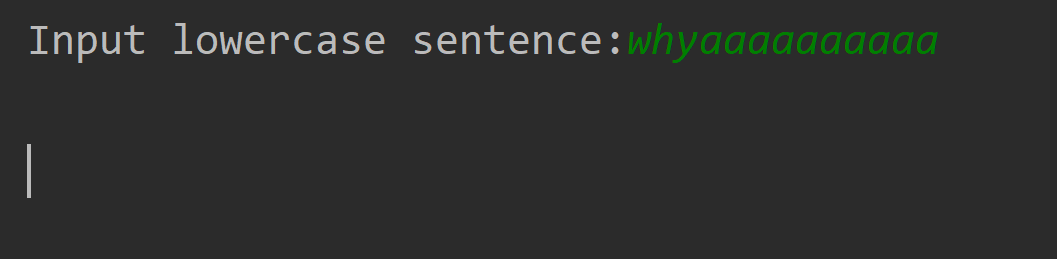
### c.TCP client-server处于正常交互状态时，打开hostOS的防火墙，“拒绝来自guestOS的TCP报文”，就所捕获的数据包s,观察并得出什么结论？

i.若你的hostOS是windows操作系统，请查阅如何编辑windows防火墙。

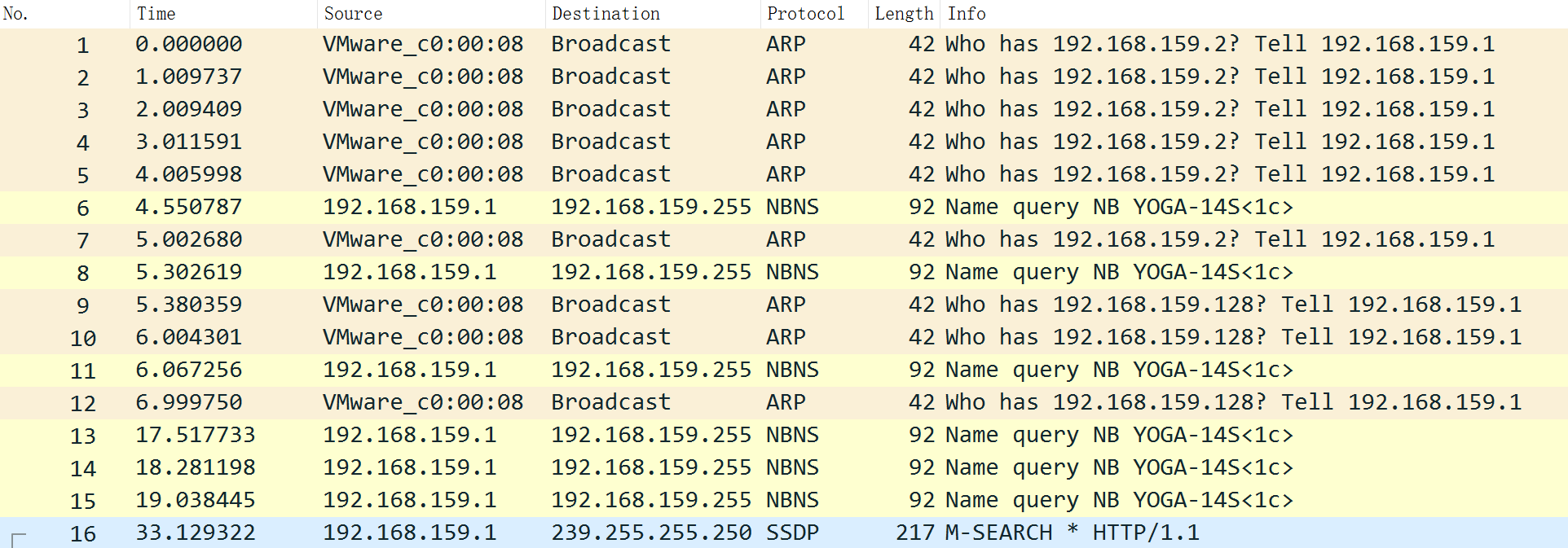
ii.解释：防火墙/firewall,意思是对in、out的网络连接进行permit或deny。还记得网络五元组吗？(source ip、source port, dest ip、dest port.,TCP/UDP),一般意义的防火墙就是针对这5个大的变量进行设置，例如“阻止来自某个P地址访问我本机的某个端口”。linux上的iptables可以细致到协议首部的逐字段级，例如仅针对TCP首部中syn bit.置为1的TCP报文。



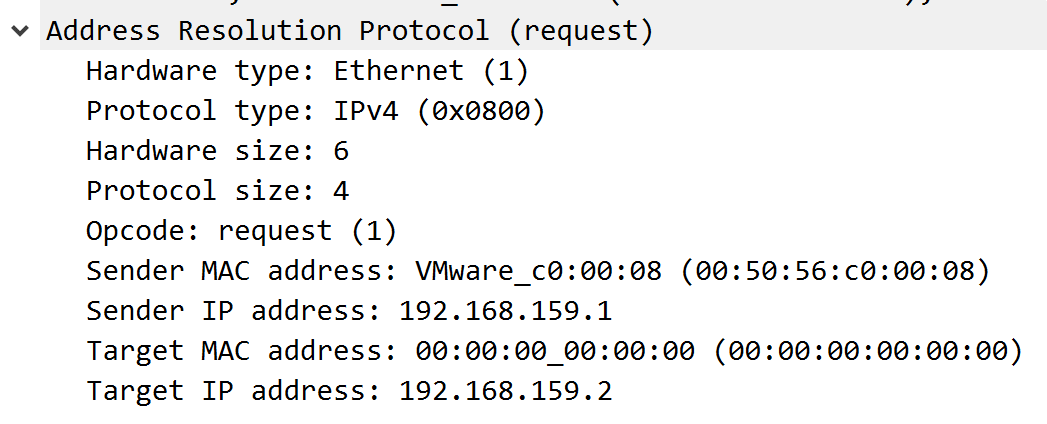
如图，只有这几个包，前面三个是三次握手建立连接，然后是hostOS向Server发送数据包，但是开启了对应端口的防火墙所以收不到答复了，客户端这边也没有显示接收到的信息。

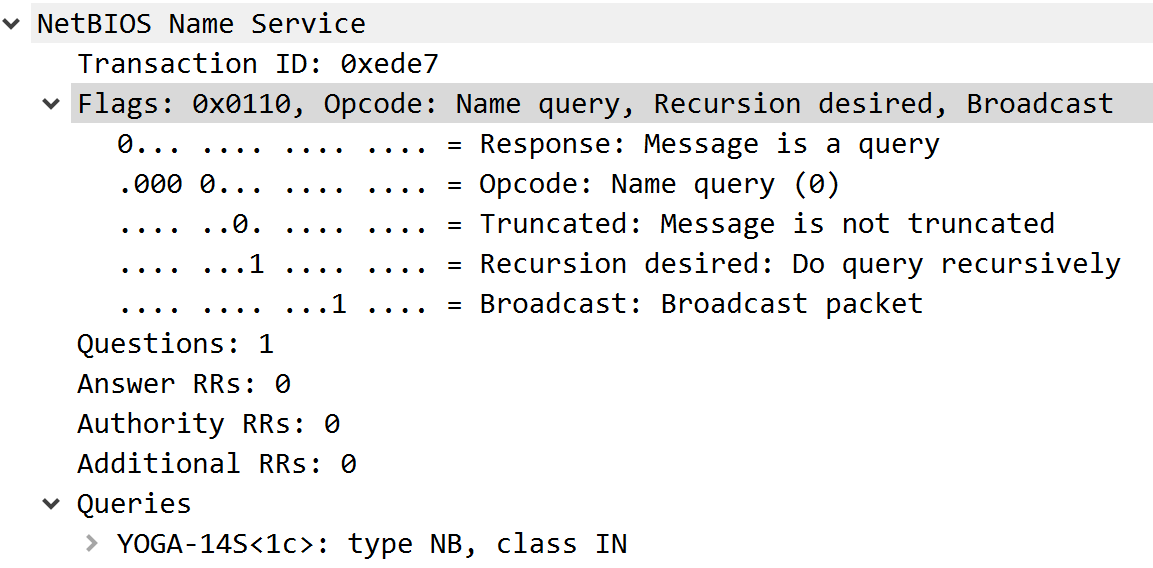


### d.运行UDP client,关闭UDP server。就所捕获的数据包s,观察并得出什么结论？

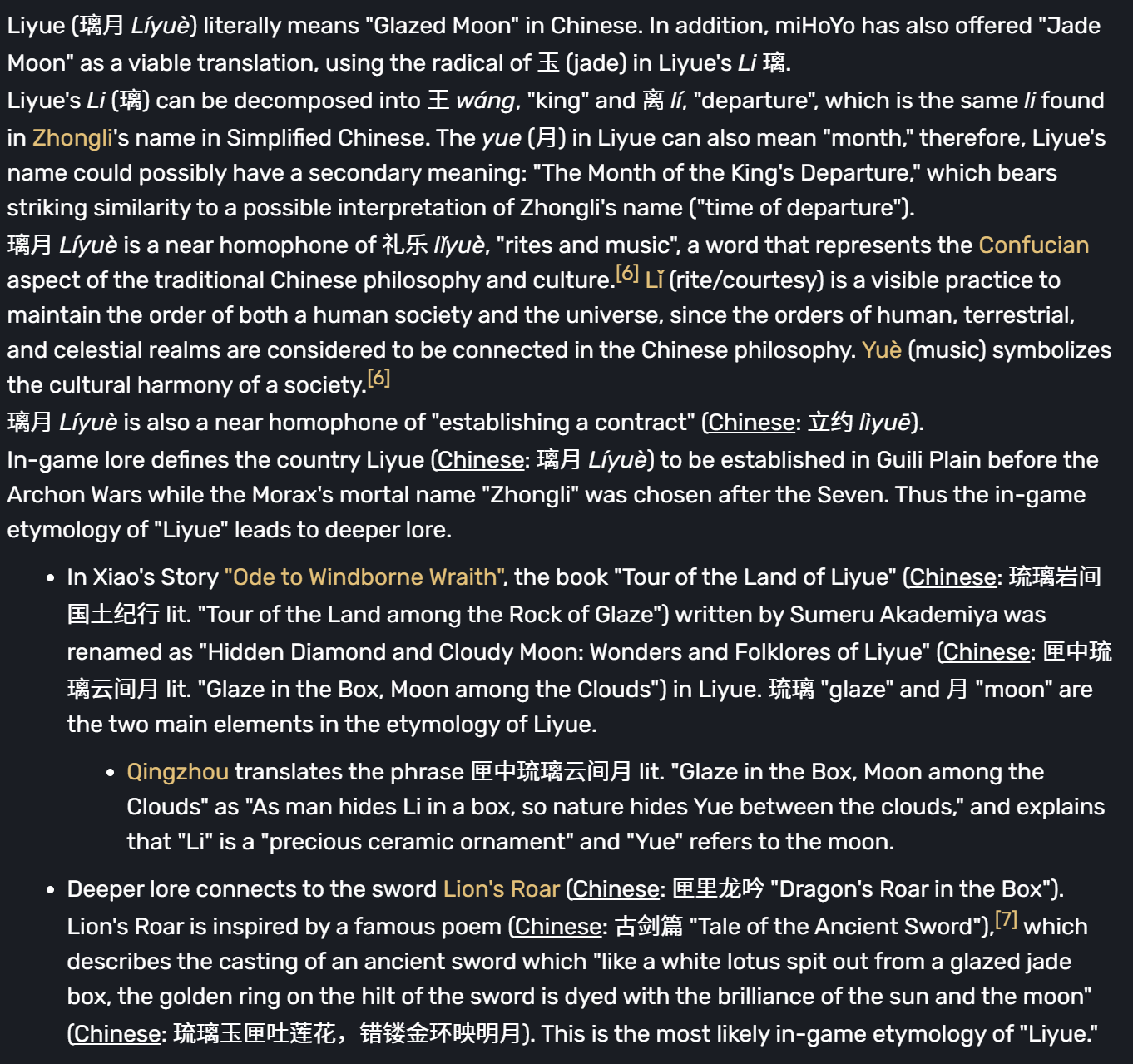


如图，主机会向网络上以广播的方式发送询问，查询虚拟机所在子网的IP，主机从属于192.168.159.1这个子网，虚拟机从属于192.168.159.2这个子网，由于没有开启服务端，主机找不到相应的IP地址对应的物理链路，故通过ARP协议寻找该IP地址对应的物理地址。如图，Sender MAC Address为本机与虚拟机的网络连接适配器，Target IP address为虚拟机的子网地址。



分析下面的数据包，发现是递归地逐级向上查询虚拟机的地址.（Do query recursively）

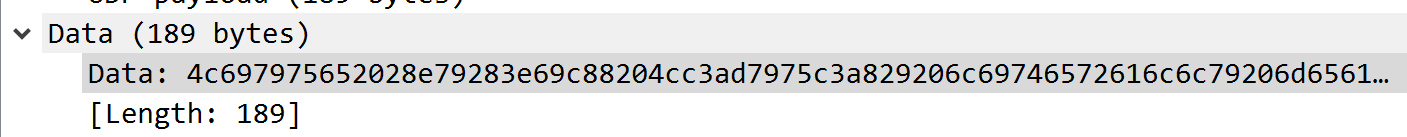
### e.同时运行UDP client server,当UDP client发送的数据size很大超过了设定的字节数时，就捕获的数据包s，观察并得出什么结论？

如图，发送以下文字（肯定超了）

由服务端解析后，返回了以下文字：b'LIYUE (\xe7\x92\x83\xe6\x9c\x88 L\xc3\x8dYU\xc3\x88) LITERALLY MEANS "GLAZED MOON" IN CHINESE. IN ADDITION, MIHOYO HAS ALSO OFFERED "JADE MOON" AS A VIABLE TRANSLATION, USING THE RADICAL OF \xe7\x8e\x89 (JADE) IN LIYUE\'S LI \xe7\x92\x83.'

('192.168.159.128', 12345)

查看wireshark发现data的长度远小于发送的数据，只有189字节。

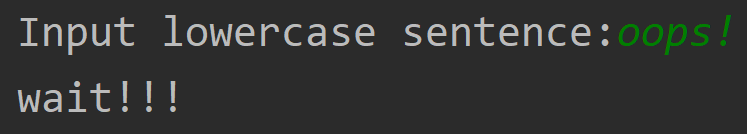


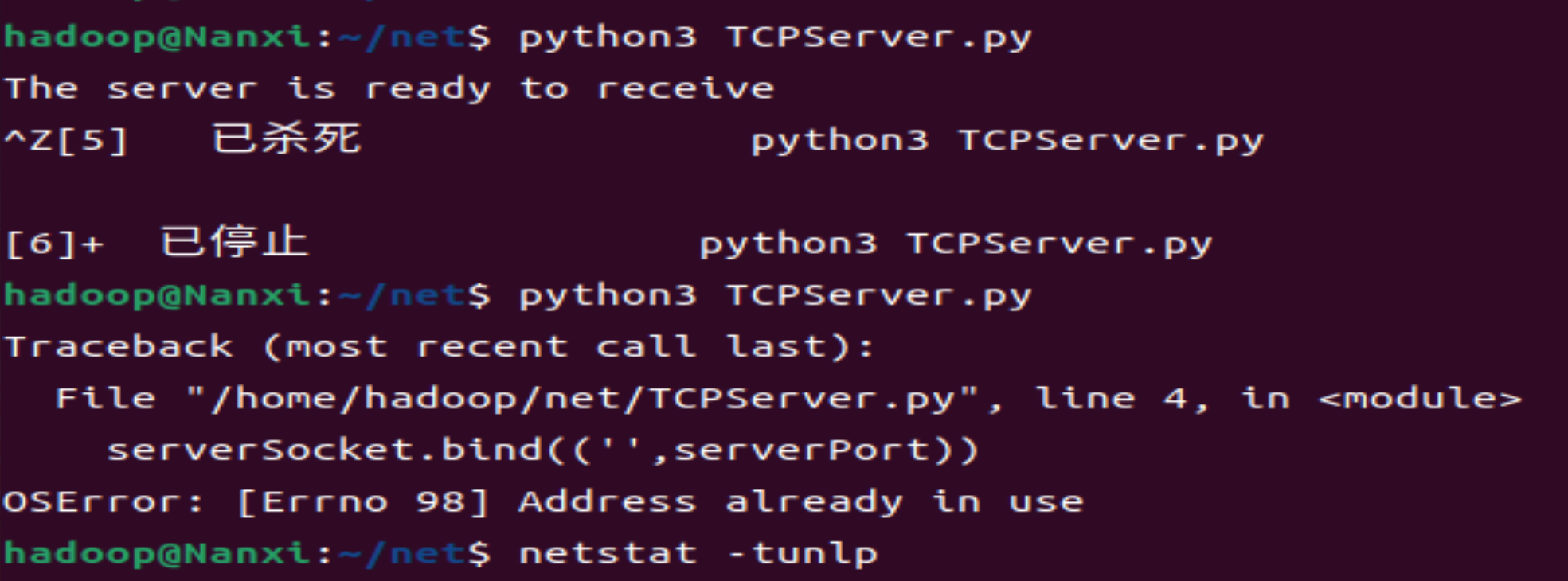
可以发现服务器只接收到了部分的字符，并没有收到全部的字符。网上查阅资料得知当我们发送的UDP数据大于1472的时候，这个时候发送方IP层就需要把数据报分成若干片，使每一片都小于MTU.而接收方IP层则需要进行数据报的重组。这样就会多做许多事情，而更严重的是，由于UDP的特性，当某一片数据传送中丢失时,接收方便无法重组数据报，这将导致丢弃整个UDP数据报。

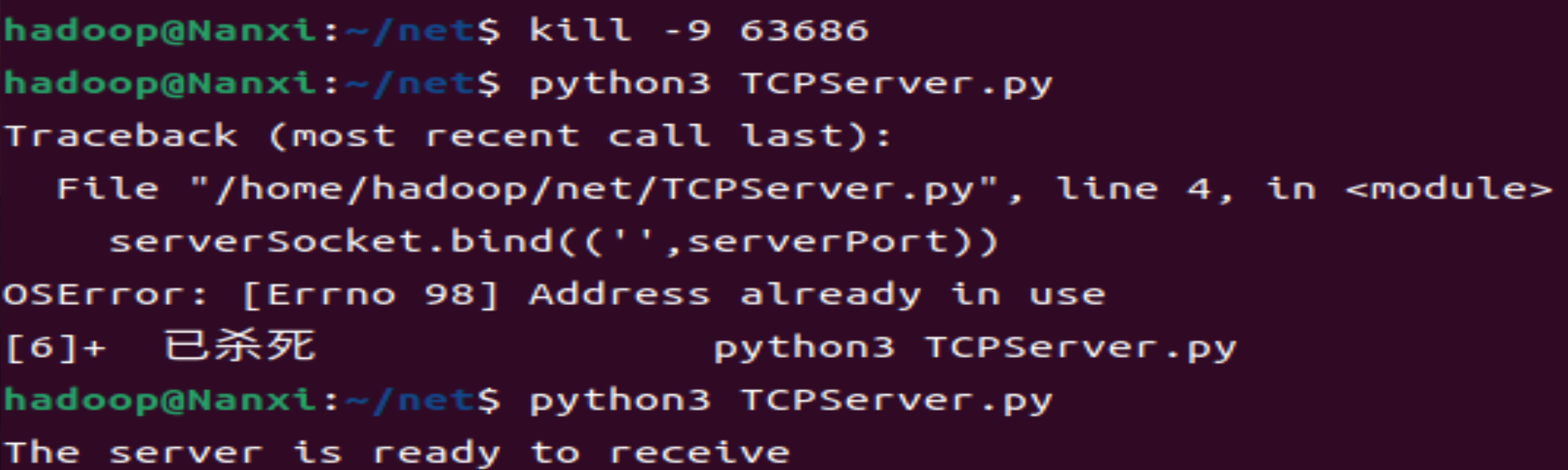
后面没有再收到其他UDP数据包，也没有收到服务端的任何信息，可能是因为后面的包都丢了吧…

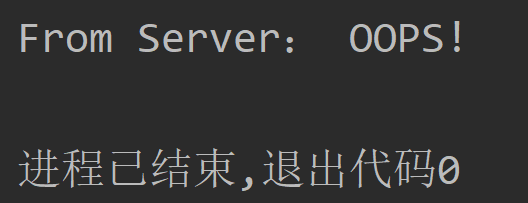
### f.你还能构建什么场景？

TCP，服务端和客户端正在正常运行并连接，但是服务器端突然断开了（Linux端终止连接，杀死进程后再重新启动Server端，继续Client 端发送的字符串，如图，仍然能够继续刚才的过程。

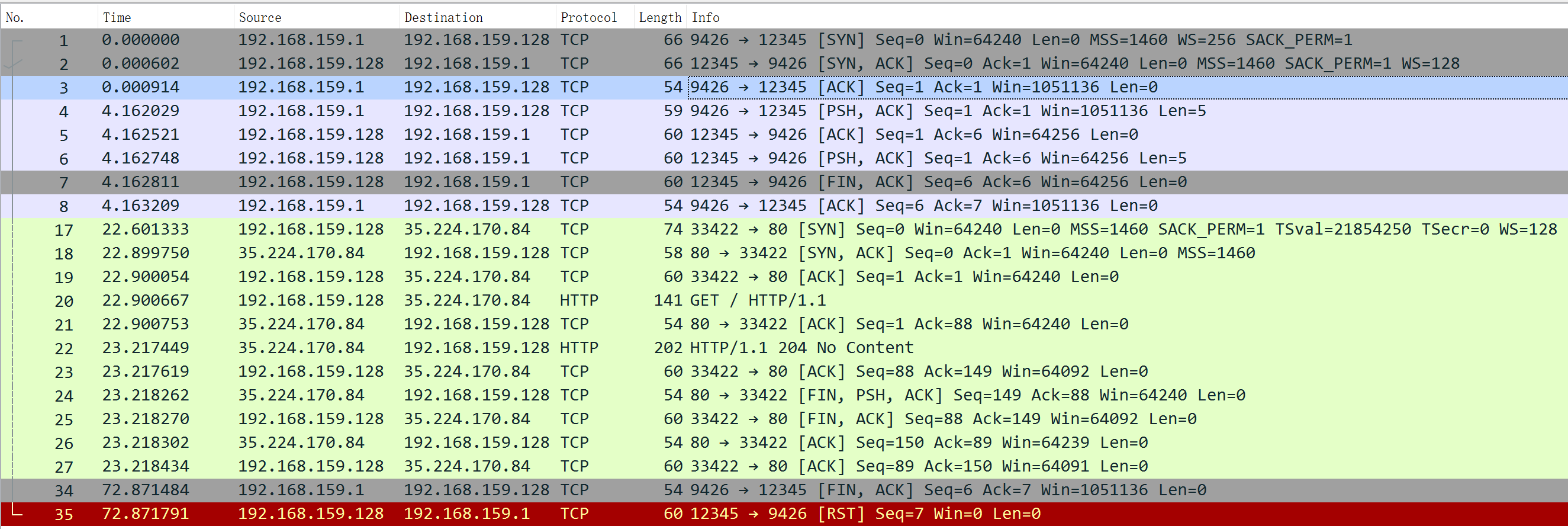




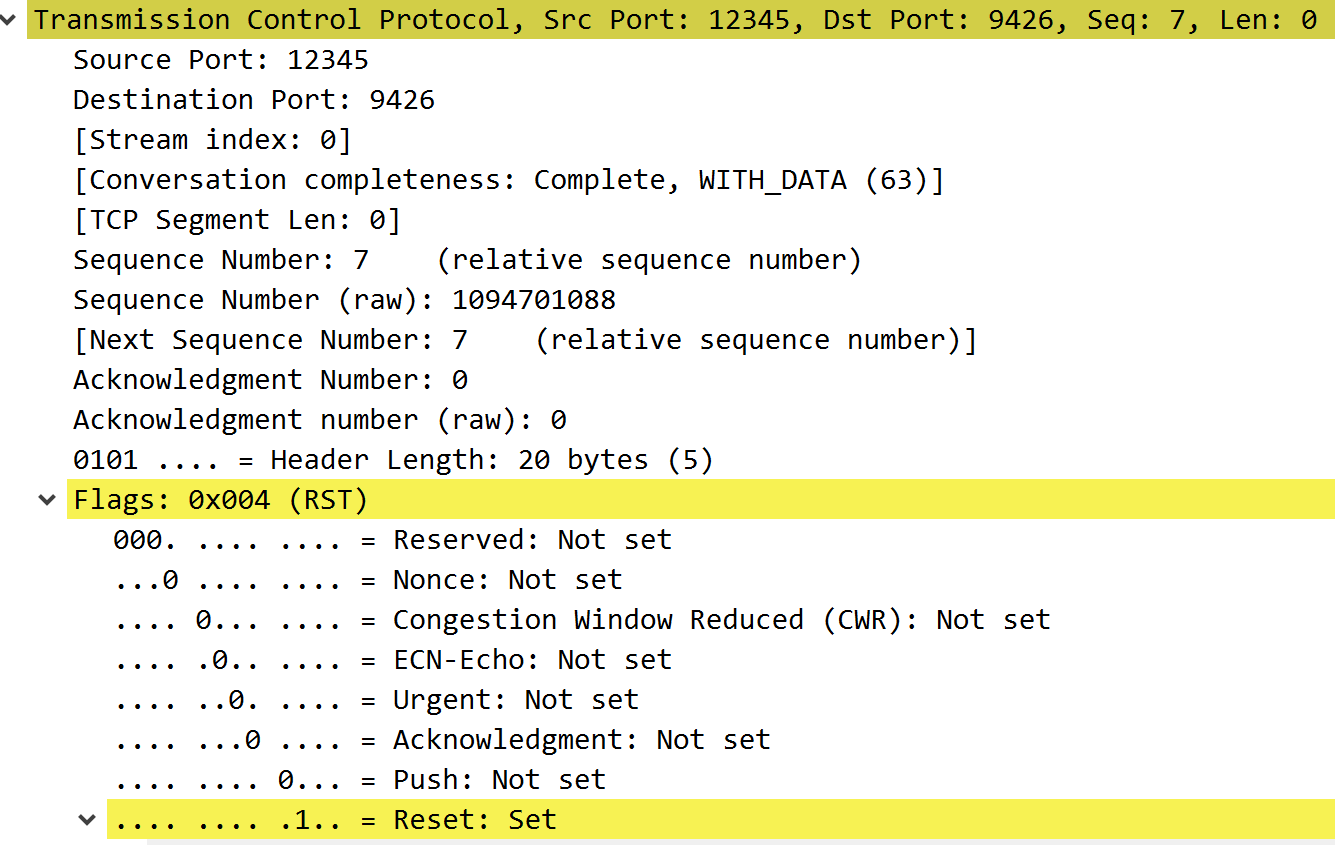




TCP报文抓包如下：

1-3是3次握手，后面手动断开Server又重启，4-6客户端再发送报文，服务端回传，传输的过程没问题，但是在服务端终止会话的时候有一个重传（7号包）（中间绿色部分的TCP不知道是什么地方的…），而且挥手的过程只进行了两次而不是4次。  


重传报文的Flags会变成RST状态，Reset标志位会变为set状态。

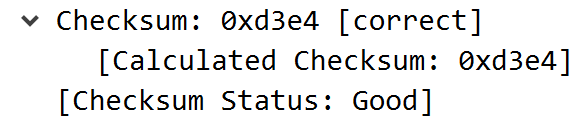
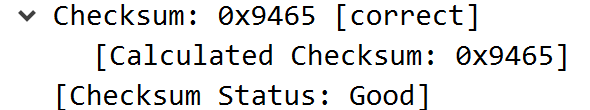


## TCP和UDP的checksum在计算时，要加上伪首部（pseudo-header），什么是伪首部？wireshark这个软件是怎么对待checksum的？

伪首部是因为这种首部并不是UDP用户数据报真正的首部。只是在计算校验和时，临时添加在UDP用户数据报前面，得到一个临时的UDP用户数据报。校验和就是按照这个临时的UDP用户数据报来计算的。

伪首部既不向下传输，也不向上递交，而仅仅是为了计算校验和。

wireshark 打开检查checksum值功能后，会自动对checksum值做出计算并比对，但不会显示伪首部的详细信息。如图：

## 本小题不用做，去google下并了解一下即可：

1. 什么是TCP的TSO？

就是靠网卡硬件来分段，计算checksum，从而解放CPU周期。

贴一个解释：TSO (TCP Segmentation Offload) 是一种利用网卡替代CPU对大数据包进行分片，降低CPU负载的技术。如果数据包的类型只能是TCP，则被称之为TSO。此功能需要网卡提供支持。TSO 是使得网络协议栈能够将大块 buffer 推送至网卡，然后网卡执行分片工作，这样减轻了CPU的负荷，其本质实际是延缓分片。这种技术在Linux中被叫做GSO(Generic Segmentation Offload)，它不需要硬件的支持分片就可使用。对于支持TSO功能的硬件，则先经过GSO功能处理，然后使用网卡的硬件分片能力进行分片；而当网卡不支持TSO功能时，则将分片的执行放在了将数据推送的网卡之前，也就是在调用网卡驱动注册的ndo\_start\_xmit函数之前。

在TCP连接的建立阶段，需要开启TSO功能。

b.什么是TCP的TCO？

## 以下选做，技能点提升。

### （1） 学习linux下iptables的基本语法

a. 以ubuntu为例，不同ubuntu发行版本，防火墙也有差异，请根据自已虚拟机里ubuntu的具体版本，google一下iptables的使用方法。

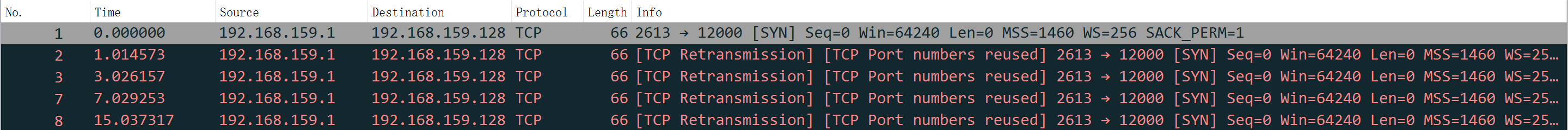
### （2） 还是仅使用第二次作业的TCP/UDPclient-server程序。

通过使用iptables过滤掉第2个syn报文、最后一个ack等，来考察TCPclient、server不同的行为（例如TCP连接建立阶段，server在多久没有收到来自于client的ack，会关闭掉连接）。

使用

sudo iptables -A OUTPUT -p tcp --tcp-flags SYN ACK --dport 12000-j DROP

命令，将虚拟机的SYN报文全部丢弃，报文情况如下（我换了一个端口12000）：



Client发送4次，收不到回复的话就不发了。