**北京邮电大学软件学院**

**2020-2021学年第一学期实验报告**

**课程名称： 计算机网络\_\_\_\_\_\_\_\_**

**实验名称： 传输层实验 \_\_\_\_\_\_\_**

**项目完成人：**

**姓名：**\_ 徐龙\_\_**学号：**\_\_ 2018522108 \_\_\_

**指导教师：**\_\_\_\_\_\_\_ \_\_**\_\_雷友珣\_**\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**日 期： 2019 年 12 月 15 日**

1. **实验目的**

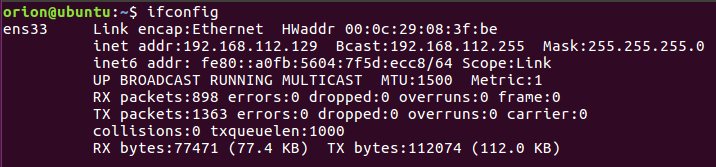
通过本实验使学生理解并掌握TCP连接，掌握TCP报文（TCP Segment）字段组成、字段的作用和使用方法。

1. **实验内容**
2. 恢复本课程实验二（“网络层实验”）中的网络环境，并使主机Host\_1和主机Host\_2在网络层连通。
3. 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。
4. 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Host\_1和主机Host\_2上。
5. 在未启动TCP服务器应用程序的情况下，在主机Host\_1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。
6. 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Host\_1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Host\_2，然后在主机Host\_1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。
7. 删除在路由器R1中设置的丢弃主机Host\_1发出的IP分组的防火墙规则，使得主机Host\_1发出IP分组能够到达主机Host\_2。在主机Host\_2启动TCP服务器应用程序，在主机Host\_1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。
8. 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。
9. 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。
10. **实验环境**
11. Linux系统主机。
12. VMware Workstations软件。
13. Wireshark软件。
14. Linux环境下Socket网络编程开发和运行环境：Java程序开发环境Eclipse。
15. **实验步骤及结果**

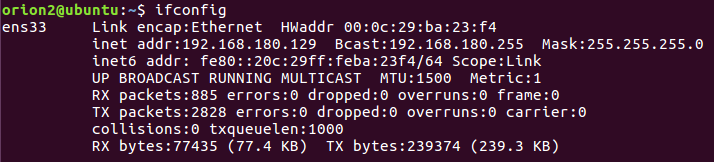
**标注：**

1. 由于做完实验二之后就删除了创建的虚拟机，因此实验三重新创建了新的虚拟机Host\_2和Host\_1。
2. 一下内容实验所需给出的TCP报文交互过程，客户端程序和服务端程序的源代码在附录中给出。
3. 注意：在实验过程中不小心将客户端部署到了虚拟机Host\_2，服务端部署到了虚拟机Host\_1，与实验指导书相反。
   * + 1. **创建两台新的虚拟机重复实验二，并使主机Host\_1和主机Host\_2在网络层连通**

使用ifconfig命令查看Host\_1和Host\_2的IP地址：

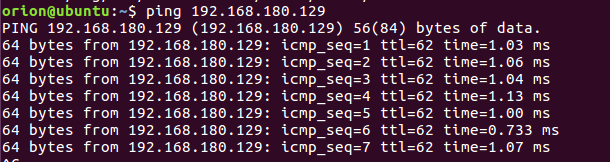


Host\_1的IP地址为：192.168.112.129.

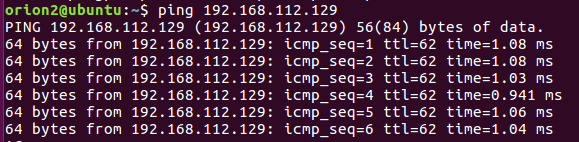


由图可知，**Host\_1的IP地址为：192.168.180.129**.

在Host\_1中ping命令测试与Host\_2的连接：



在Host\_2中ping命令测试与Host\_1的连接：



Host\_1和Host\_2已经相互连通。

* + - 1. **编写TCP客户端应用程序和TCP服务器应用程序，并部署到Host\_1和Host\_2中，查看是否能相互连通。**

程序说明：

使用Java语言对服务端程序和客户端程序进行开发。

在服务端程序中，采用了多线程机制来支持同时与多个TCP客户端程序建立TCP连接。服务端程序包含SocketService类和ServerThread类，其中SocketService线程用来监听客户端的连接请求,如果接受到一个连接请求那么创建一个新的ServerThread线程与客户端进行数据传输。一个ServerThread线程处理一个客户端的消息，并将其输出在控制台中，当客户端断开连接时，线程结束。

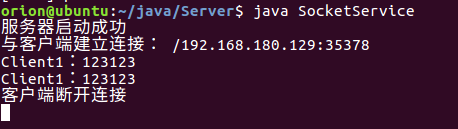
在客户端程序中，创建了三个同样的客户端实例，都是通过向服务端发送连接请求之后进行通信。其中，SocketClient1.java和SocketClient2.java两个程序代码完全一样，都能够实现无限次地向服务端发送消息，不同的是发送的消息内容不同，便于之后在服务端进行区分。

Client3.java中，只发送一次消息，而该消息的长度很长，这使得TCP协议在发送该数据的时候会进行分段操作。这是为了方便之后进行TCP协议分段数据实验的分析。

以下为程序运行实例，运行服务端程序，等待客户端发来连接请求，当与一个客户端建立连接时会输出客户端的IP地址以及端口号。

当客户端向服务端发送消息时，服务端会显示出来，而当客户端发送“quit”消息请求断开连接时，服务端便会与客户端断开连接。

服务器与一个客户端建立连接，断开连接



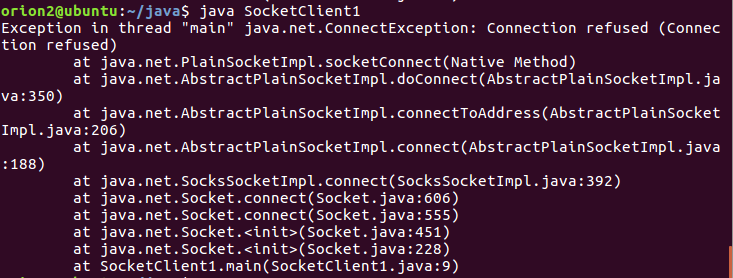
服务器与多个客户端连接的结果，并对多个客户端进行消息的处理。



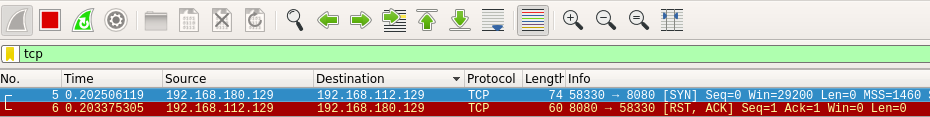
详细的程序代码请看附录

* + - 1. **在主机 Host\_2 上启动 TCP 客户端应用程序，查看客户端应用程序的执行情况，分析TCP连接建立请求消息和响应消息。**

在Host\_2中启动客户端程序，发现程序抛出异常，原因为没有运行服务端程序，客户端无法建立连接。

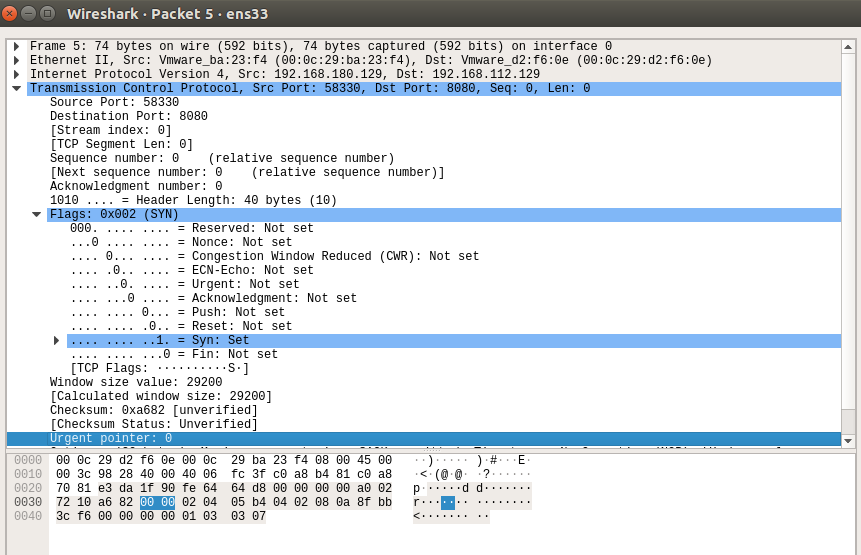


查看Wireshark抓包结果，发现在客户端和服务端均抓到了两个TCP报文，报文内容分析如下：



1. **查看客户端发往服务端的Packet5**

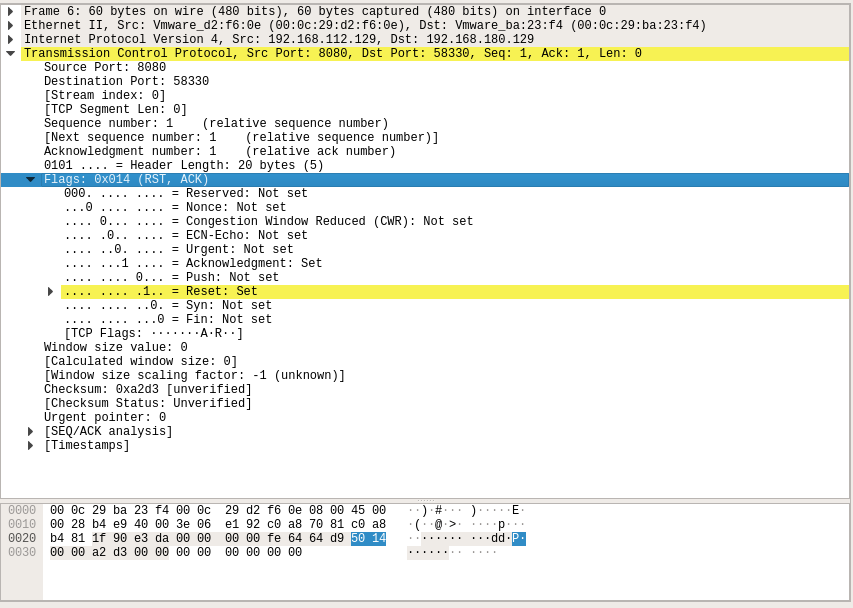
5号帧字段取值情况如下：



从IP地址和端口号可以看出这个报文由客户端发往服务端，端口号为58300。

查看报文中Flags字段的值，其中**SYN为1，RST为0**，由此可以得知该报文是一个向服务端的连接请求。

1. **查看服务端发往客户端的packet6**



从IP地址和端口号可以看出这个报文由服务端发往客户端，服务器端口号默认为8080端口。

查看报文中Flags字段的值，其中**SYN为0，RST为1，ACK为1**.该报文的ACK位为1，RST位为1由此可以得知由于服务端程序没有开启，与客户端建立连接失败，于是发送了一个响应报文，让客户端的TCP连接重置。

* + - 1. **在主机 Host\_2 上启动 TCP 客户端应用程序，观查 TCP 客户端应用程序的执行情况，分析TCP连接建立请求消息和响应消息**

在Virtual\_Router中使用iptables命令，将来自主机Host\_2的数据包丢弃，使得客户端发送的连接请求无法到达服务端：



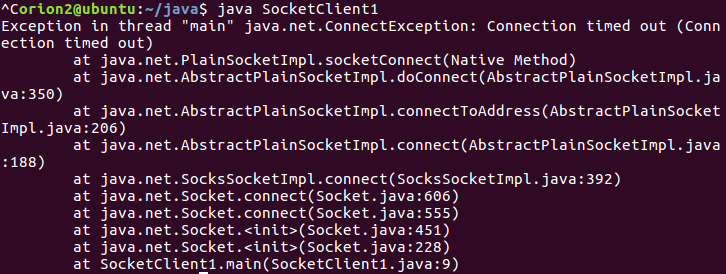
之后启动主机Host\_1中的服务端程序



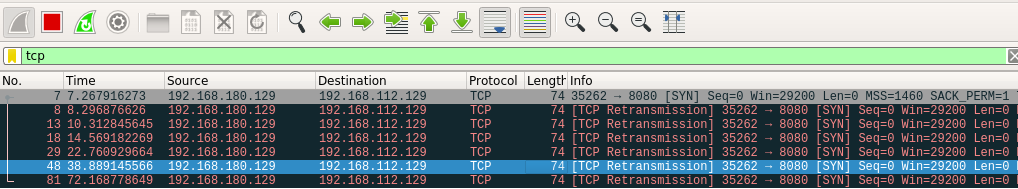
启动主机Host\_2的客户端程序



启动之后，客户端程序处于阻塞状态，直到一段时间之后，程序输出报错信息

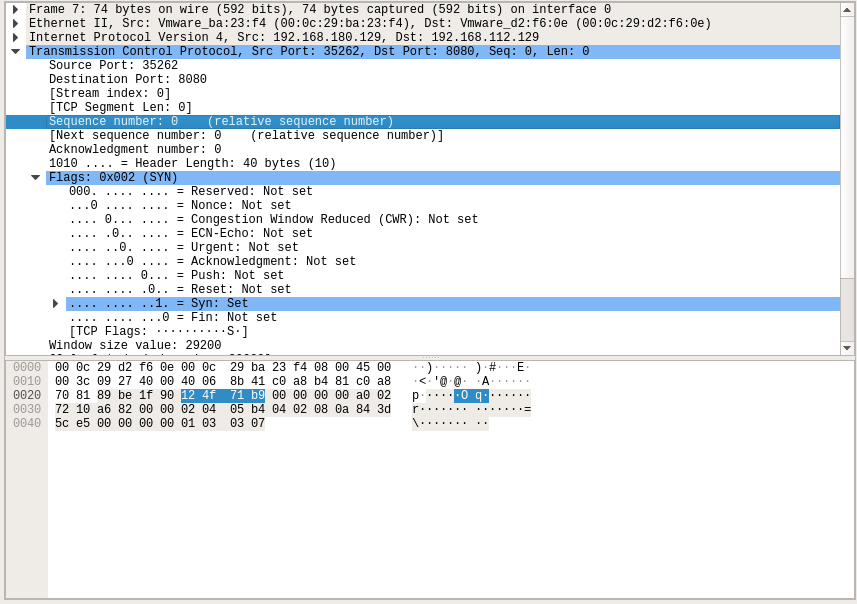


使用使用wireshark在主机Host\_2中进行抓包，分析原因，结果如下



1. **建立连接的数据包packet7**

packet7为客户端向服务端的连接请求包，具体报文内容如图所示

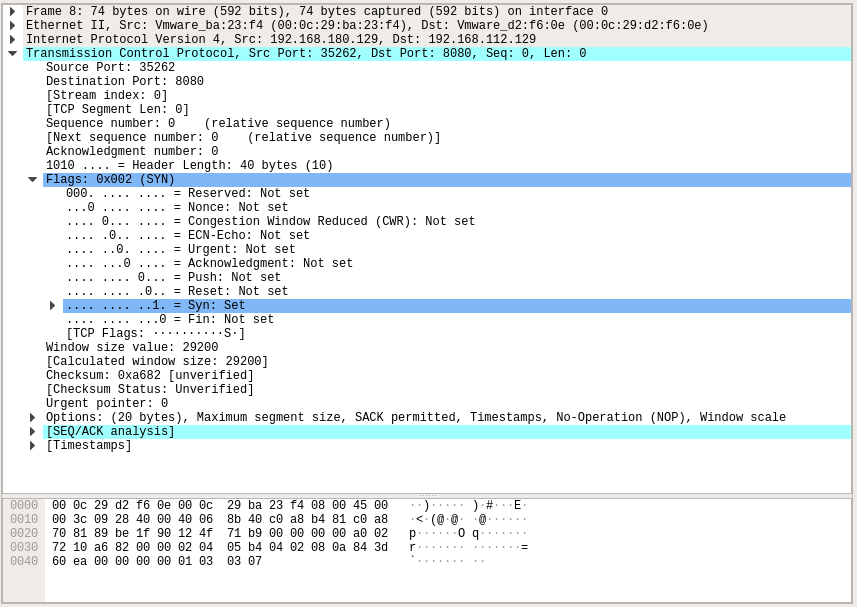


由查看结果可知，**数据包中SYN为1，ACK为0**，该数据包的作用为发送连接请求，并没有响应数据包的作用，所以SYN置为1，ACK置为 1.

由图中序列号字段对应的16进制数来看，该数据包的序列号对应的字段值为0x124f71b9，但是界面中显示的序列号为0，这是因为wireshark通常显示的是相对序列号，而不是实际的序列号，这样会更方便我们进行判断。

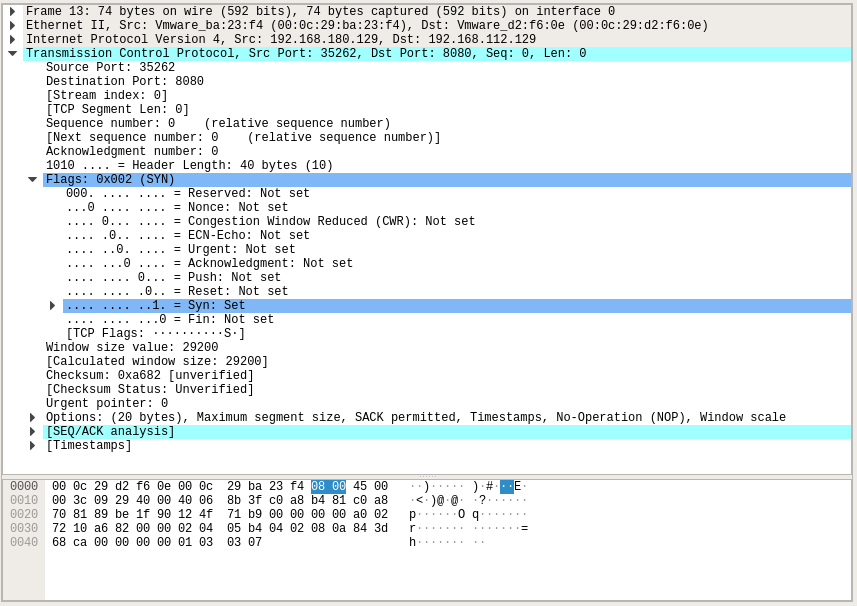
1. **TCP连接重传报文Frame8、13、29、48、81**

客户端设定计时器，如果计时器超时后服务端仍然没有给出响应，则客户端发送TCP连接重传报文，下图为Frame8中的TCP报文的具体内容：



由查看结果可知，**报文的SYN为1，ACK为0，序列号的值为0**，该报文的作用仍然是发送连接请求，因此SYN和ACK的值与第一个TCP报文相同。

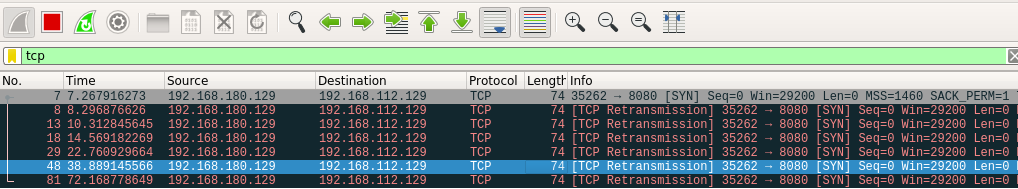
Frame8中的TCP报文的具体内容如下图所示：



由查看结果可知，**报文的SYN位为1，ACK位为0，序列号的值为0**，与上一个TCP重传报文的内容一致，由此可以判断得出，后面的所有TCP重传报文的内容都是相同的。

1. **重传时间间隔分析**

之前客户端发送重传帧的过程如下所示：



通过查看Time属性可以发现这些重传报文的时间间隔分别为：1s、2s、4s、8s、16s和32s，而当最后一个重传报文发送之后，客户端便不再发送重传报文，而是直接抛出程序异常，并结束程序。

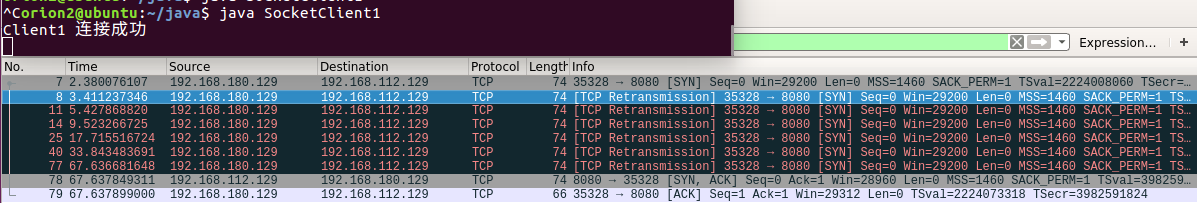
这些时间间隔可以发现规律：每发一次新的重传报文，相隔时间都会是上一个时间间隔的二倍，直到发送了六个重传报文之后，结束重传操作，进行连接重置操作。

1. **删除防火墙过滤规则，恢复客户端与服务端的连接**

在客户端处于阻塞状态，并发送重传报文的过程中，在Virtual\_Router中使用iptables命令，将刚才设置的过滤规则删除：



Wireshark所抓到的包如下所示



通过结果可以发现，客户端在发送Frame77时，服务器正确地收到了客户端的连接请求，并发送了接收报文。Frame78、79分别对应了三次握手的后面两个过程，而且这时候客户端的程序也显示已经成功地与服务端进行连接。

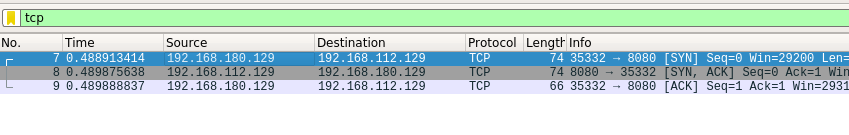
服务端此时的程序运行结果如下



这也说明服务器正常地与客户端建立了连接。

* + - 1. **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TCP报文中的IP协议头字段和TCP报文字段**

重新打开客户端程序和服务端程序，进行一次TCP连接，在wireshark中收到的TCP报文如下所示：

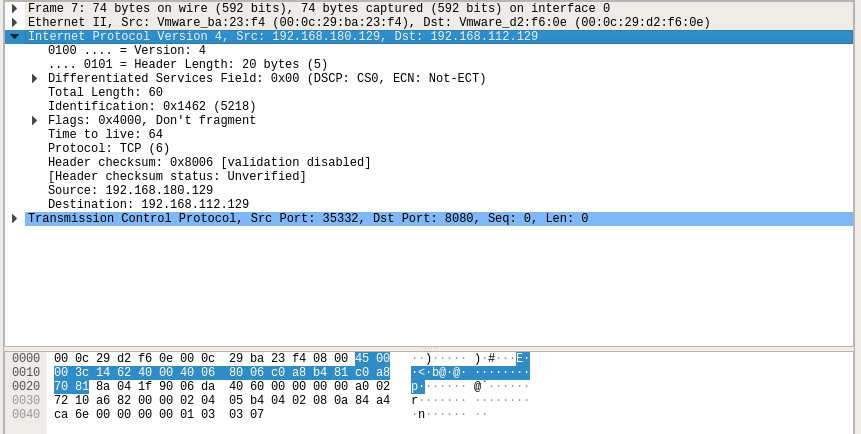


在连接建立的过程中，一共存在3个TCP报文的传输，分别对应三次握手的三个过程，下面对这三个数据帧的IP分组和TCP报文进行展示分析。

（IP协议头因为在实验一中进行了详细的字段分析，因此在这里只对第一个帧的IP协议头字段进行分析）

1. **第一次握手**

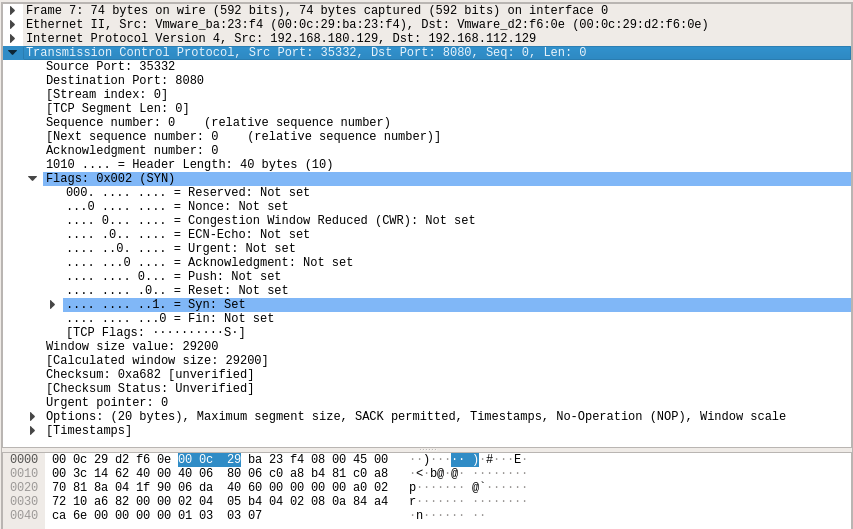
在第一次握手中，客户端向服务端发送TCP连接请求，该数据帧的IP协议头字段结构如下所示：



对于IP协议头，对下面几个字段进行分析：

* **源IP地址**：192.168.180.129，在这里表示客户端的IP地址，也就是主机Host\_2的IP地址。
* **目的IP地址**：192.168.112.129，在这里表示服务端的IP地址，也就是主机Host\_1的IP地址。
* **protocol字段**：TCP，对应的十六进制数为0x06。该字段长度为8bit，标识了网络层上层，即传输层所使用的协议。比较常用的为TCP, UDP, ICMP等。
* **IP分组头长度（IHL）字段**：0101，即20个字节，表示本IP包中包头的长度为20个字节。该字段总长度为4bit，作用是描述IP包头的长度，因为在IP包头中由变长的可选部分。一个IP包头的长度最长为“1111”，即15\*4＝60个字节。IP包头最小长度为20字节。
* **IP分组长度（Total Length）字段**：0x003c，即60，表示IP包的总长度为60个字节。该字段的长度为16bit，该字段以字节为单位计算的IP包的长度 (包括头部和数据)，所以IP包最大长度65535字节。

下面开始分析TCP报文头部的字段，该数据帧的TCP协议头字段结构如下所示：



对于TCP协议头，对下面几个字段进行分析：

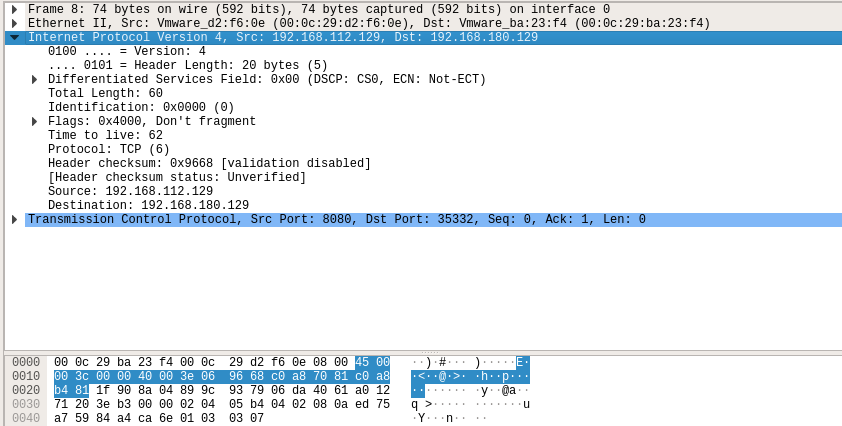
* **源端口号**：35332，即客户端使用的端口号。
* **目的端口号**：8080，即服务端程序绑定的端口号。
* **SYN比特位**：1，表示这个TCP报文请求建立一个连接。
* **ACK比特位**：0.
* **FIN比特位**：0.
* **序列号（Sequence Number）字段**：相对序列号为0，对应字段值为0x06da4060，这个序列号是客户端的初始序列号（ISN），它是系统随机生成的。
* **确认号（Acknowledgement Number）字段**：0，真实字段值也为0.

总的来说，客户端将标志位SYN置为1，随机产生一个值作为ISN，并将该数据包发送给服务端，客户端开始等待服务端确认。

1. **第二次握手**

在第二次握手中，服务端收到连接请求，向客户端发送一个同步确认消息，同时进行一次连接请求。

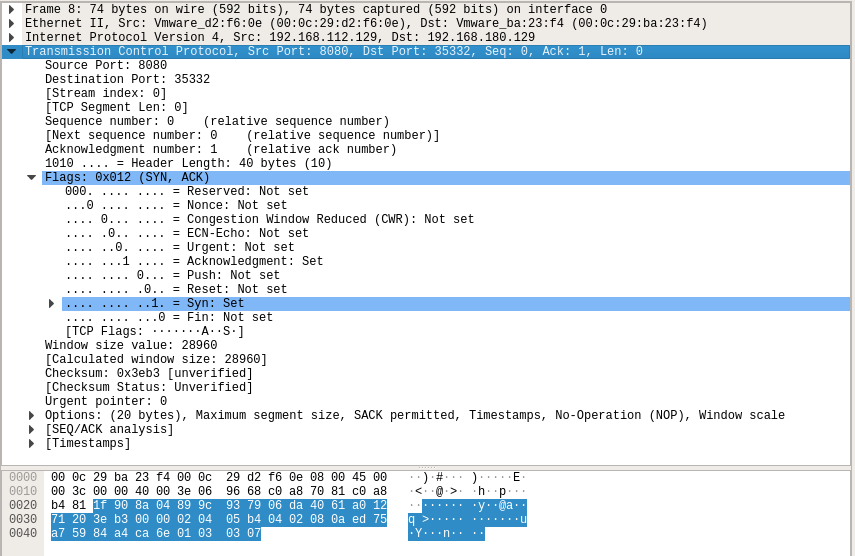
该数据帧的IP协议头字段数据如下所示：



对于IP协议头，对下面几个字段进行展示：

* **源IP地址**：192.168.112.129，在这里表示服务端的IP地址，也就是主机Host\_1的IP地址。
* **目的IP地址**：192.168.180.129，在这里表示客户端的IP地址，也就是主机Host\_2的IP地址。
* **protocol字段**：TCP，对应的十六进制数为0x06。
* **IP分组头长度（IHL）字段**：0101，即20个字节，表示本IP包中包头的长度为20个字节。
* **IP分组长度（Total Length）字段**：0x003c，即60，表示IP包的总长度为60个字节。

下面开始分析TCP报文头部的字段，该数据帧的TCP协议头字段结构如下所示：



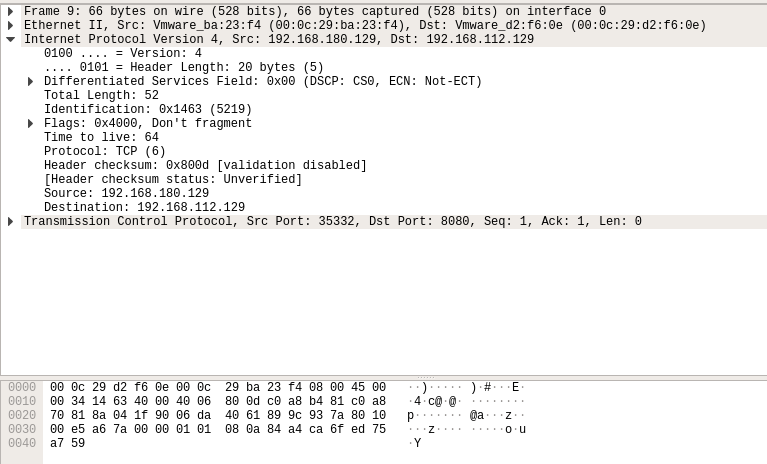
对于TCP协议头，对下面几个字段进行分析：

* **源端口号**：8080，即服务端程序绑定的端口号。即客户端使用的端口号。
* **目的端口号**：35332，即客户端使用的端口号。
* **SYN比特位**：1，表示这个TCP报文请求建立一个连接.
* **ACK比特位**：1，表示该TCP报文确认序号是否有效。
* **FIN比特位**：0.
* **序列号（Sequence Number）字段**：相对序列号为0，对应真实字段值为0x899c9379，这个序列号是服务端的初始序列号（ISN），它是系统随机生成的。
* **确认号（Acknowledgement Number）字段**：1，真实字段值为0x06da4061，该确认号是对上一个TCP报文的确认，因此该字段的值为客户端的ISN+1.

总的来说，服务端收到客户端发来的报文之后，根据SYN=1知道客户端请求建立连接，服务端在发送报文时将SYN和ACK都置1，确认号字段为客户端的ISN加一，并将该数据包发送给客户端以确认连接请求。

1. **第三次握手**

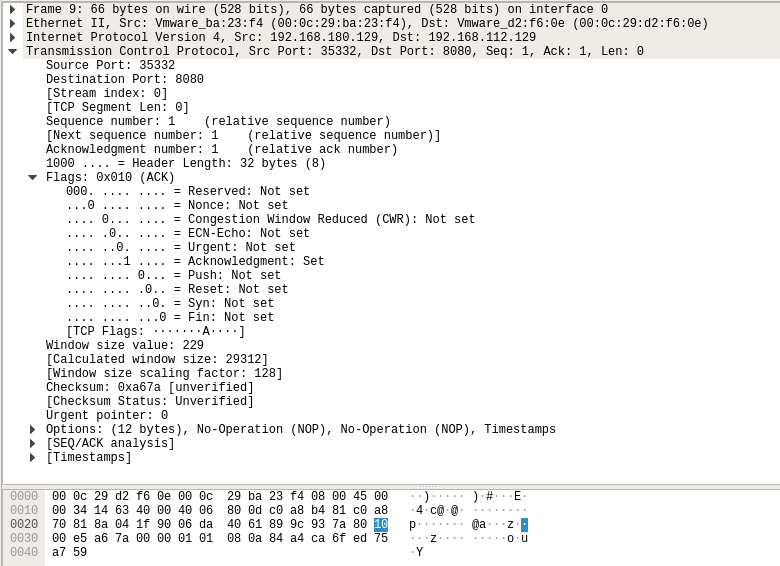
该数据帧的IP协议头字段数据如下所示：



对于IP协议头，对下面几个字段进行展示：

* **源IP地址**：192.168.180.129，在这里表示客户端的IP地址，也就是主机Host\_2的IP地址。
* **目的IP地址**：192.168.112.129，在这里表示服务端的IP地址，也就是主机Host\_1的IP地址。
* **protocol字段**：TCP，对应的十六进制数为0x06。
* **IP分组头长度（IHL）字段**：0101，即20个字节，表示本IP包中包头的长度为20个字节。
* **IP分组长度（Total Length）字段**：0x0034，即52，表示IP包的总长度为52个字节。

下面开始分析TCP报文头部的字段，该数据帧的TCP协议头字段结构如下所示：



对于TCP协议头，对下面几个字段进行分析：

* **源端口号**：35332，即客户端使用的端口号。
* **目的端口号**：8080，即服务端程序绑定的端口号。
* **SYN比特位**：0.
* **ACK比特位**：1，表示该TCP报文确认序号是否有效
* **FIN比特位**：0.
* **序列号（Sequence Number）字段**：相对序列号为1，对应真实字段值为0x06da4061，这个序列号是客户端的初始序列号（ISN）加一得到的。
* **确认号（Acknowledgement Number）字段**：1，真实字段值为0x899c9379，该确认号是对上一个TCP报文的确认，因此该字段的值为服务端的ISN+1.

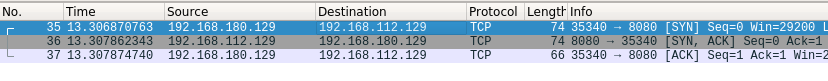
总的来说，客户端收到服务端的确认之后，检查确认号字段的值是否为客户端的ISN加一，确认之后将标志位ACK置为1，确认号字段的值为服务端ISN+1，服务端检查客户端发来的报文中确认号字段是否正确，如果正确则连接建立成功。在这之后，客户端和服务端就可以互相传输数据了。

* + - 1. **在主机Host\_1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，观察与之前的报文相比有哪些字段的取值是相同的**

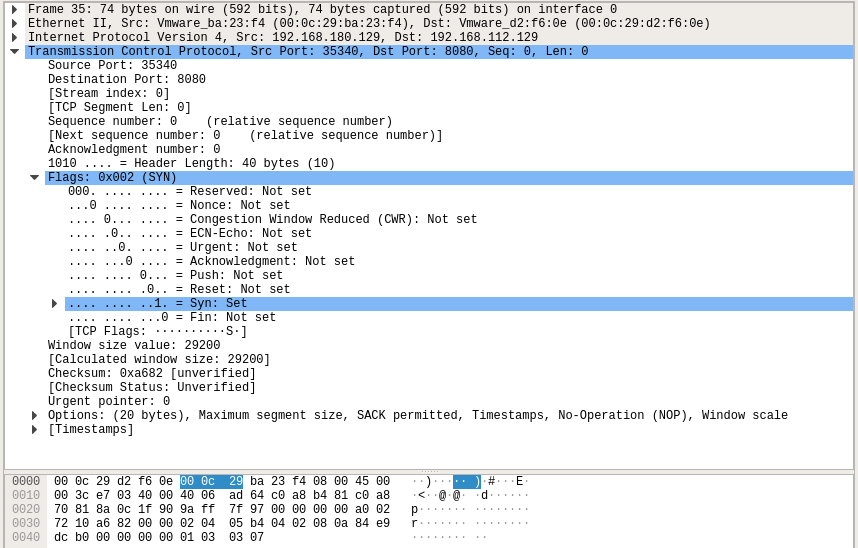
在第一个客户端与服务端建立连接的情况下，在主机Host\_2中启动第二个客户端程序，程序成功运行的结果如图所示：



在这个过程中使用wireshark捕获发送的数据包，结果如下：

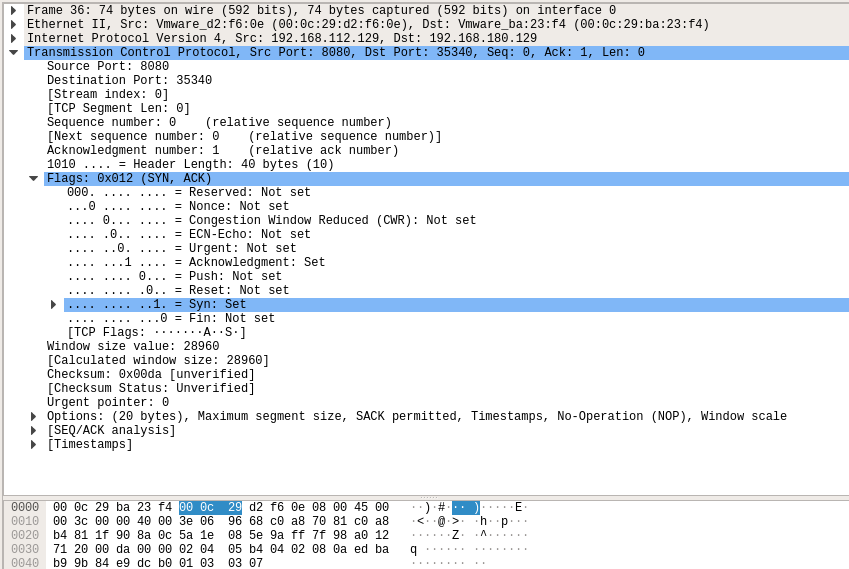


在这里对两个客户端建立的过程中发送报文内容的区别进行分析，对于第一个连接请求报文，TCP协议头内容如下：



可以发现，该客户端使用的端口号为35340，与第一个客户端的端口号不同，而且该客户端的初始序列号（ISN）为0x9aff7f97，与第一个客户端的不同。

对于第二个连接请求报文，TCP协议头内容如下：



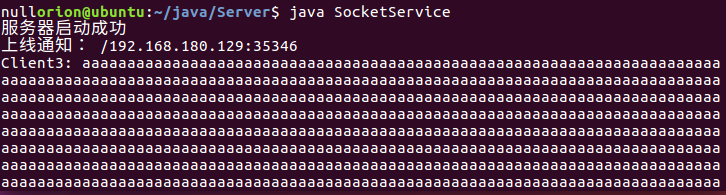
可以发现，本次服务端使用的初始序列号的值为0x5a1e085e，与第一个客户端中所使用的不同。

综上所述，在两个客户端与服务端的连接过程中，**传输的TCP报文主要区别在于客户端所使用的端口号，客户端的ISN以及服务端的ISN取值**，其他的字段取值是相同的。

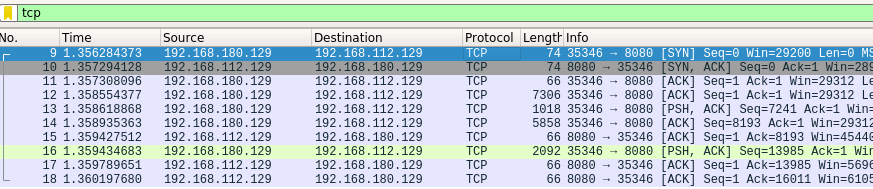
* + - 1. **触发TCP应用程序一次发送数据的数据量大于链路数据包最大长度**

在该部分中，在客户端中启动SocketClient3程序，在该程序中，客户端自动发送一个长度为16000字节的数据，使用该程序来触发TCP协议的分段操作。

服务端建立连接后收到Client3的消息如下：



说明服务器正确地收到了Client3的所有字节，下面使用wireshark对客户端进行抓包：

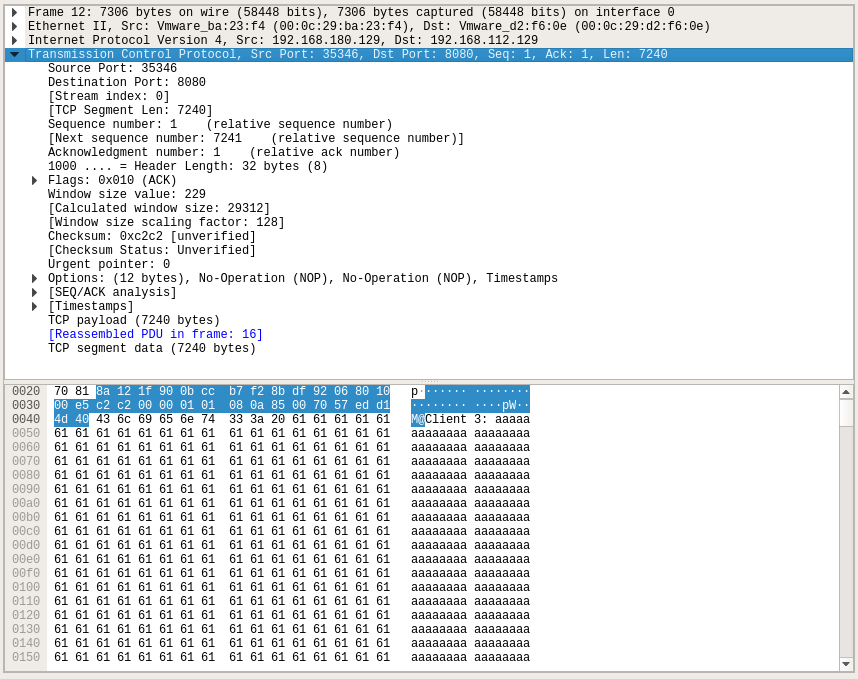


在这次抓包中，前三个帧（9号、10号、11号）帧为三次握手过程中传输的TCP报文。

查看这些报文的Length列数据，可以发现12、13、14、16号帧的长度很大，而且都是由客户端发往服务端的数据帧，因此可以判断客户端发送数据时，TCP协议进行了分段处理，并分成了这四个帧，而服务端分别对这几个帧进行确认（15、17、18号帧）。

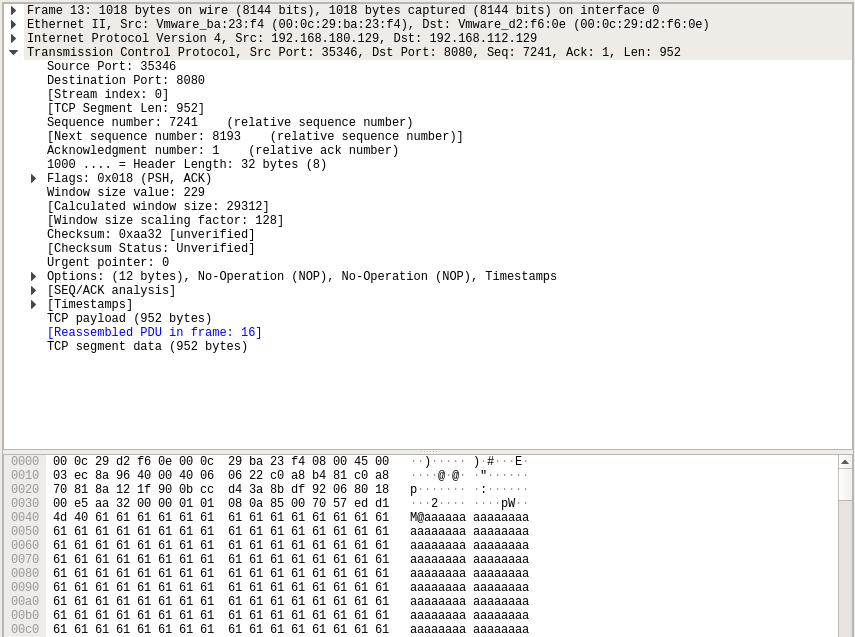
首先，对这些分段之后的帧的序列号和确认号进行分析。

对于12号帧，TCP报文的结构如下：



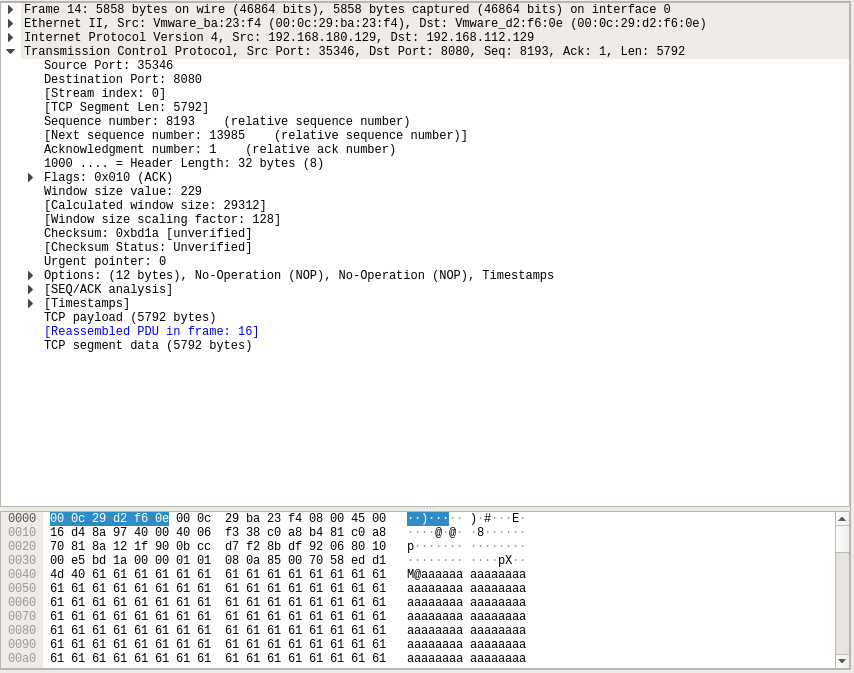
由于该报文为客户端第一次进行数据传输时发送的，因此**序列号和确认号与第三次握手时发送的报文一致**，而且都为1.而且，根据wireshark的分析，**该报文发送的数据长度为7240字节**。

对于13号帧，TCP报文的结构如下：



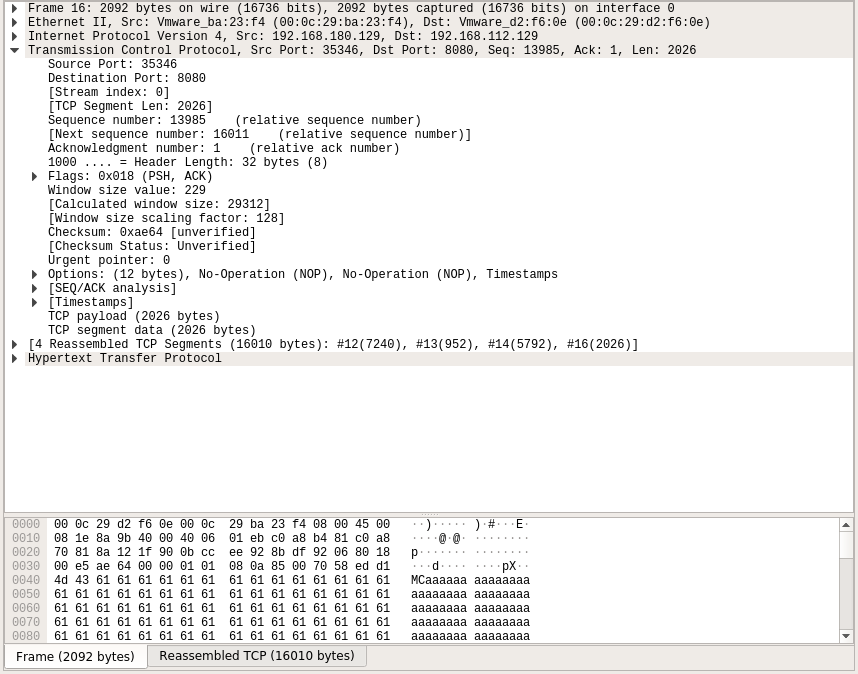
对应红色方框中的内容，**该报文的序列号为7241，这是因为该报文的序列号要在上一个报文的基础上加上其有效载荷的长度**，即7240.确认号的值为1，仍与上一个报文的一致。根据wireshark分析可知，该报文有效载荷为952字节。

对于14号帧，分析的思路也是一样的：



该报文的序列号为8193，仍然是在上一个报文的基础上加上其有效载荷的长度，即952. 确认号的值为1，仍与上一个报文的一致。

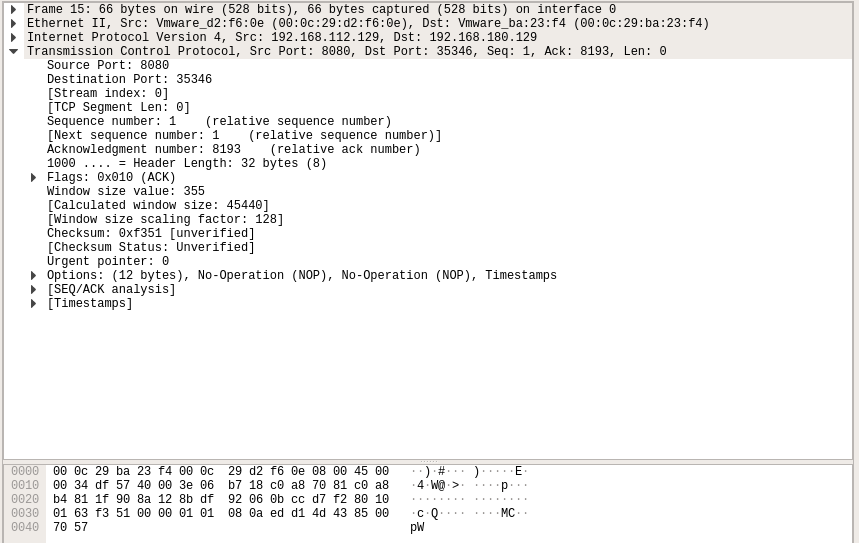
对于16号帧，结构如下：



根据wireshark分析得到的“下一个序列号”取值可知，这四个分段的总数据长度为16011字节（由于发送的串中包含“Client3:”,因此在原来的基础上多了11个字节），说明该分段发送之后，所有的长消息分段发送结束。

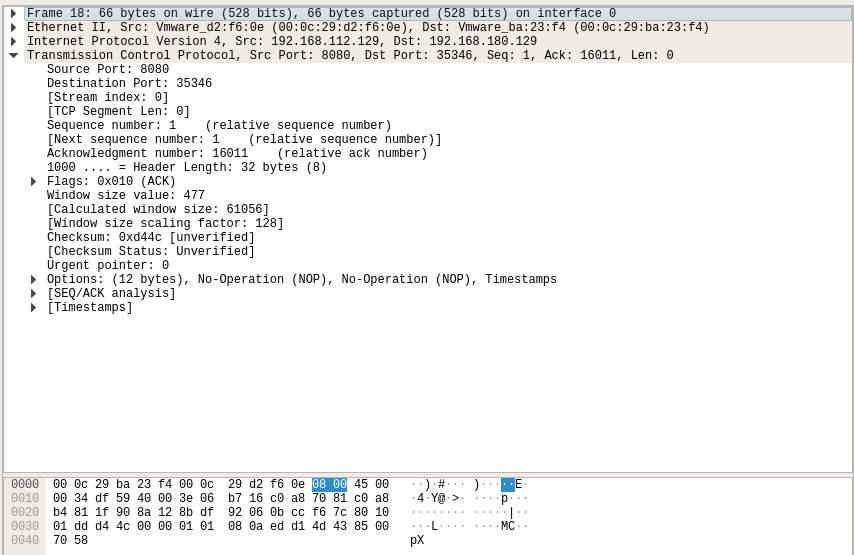
下面对服务器发送的确认帧进行分析，重点关注的是确认号的取值。

对于15号帧，结构如下：



对于服务器发送的确认帧，**序列号是与第三次握手中的报文序列号一致的**.而确认号为8193，这是因为**该报文的确认号为上一个数据包中的序列号+所该数据包中所带数据的大小**，以便客户端确认服务端是否正确地收到了全部的数据。该确认帧是对13号帧进行确认的，13号帧的序列号为7241，有效载荷的长度为952字节，因此对服务器发送对该帧的确认时，确认号为7241+952 = 8193，与之前的分析一致。

对于服务端发送的最后一个确认帧（18号帧），结构如下：

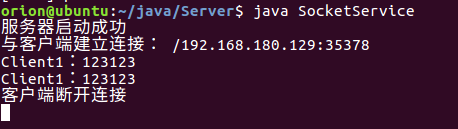


该帧是服务端发送的最后一个确认帧，该帧中TCP报文的序列号仍然与前面的确认帧保持一致，而确认号的值为16011.这说明，**服务器已经按照顺序将客户端发送的所有分段数据帧接收成功**，最后接收到的数据总长度为16011.

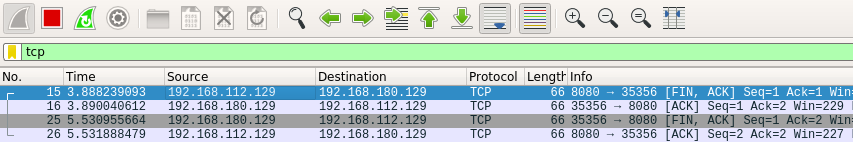
综上所述，对TCP分段数据传输的过程进行总结：

* 如果客户端发送的数据长度过长，导致组成的帧超过硬件规定的长度，那么TCP协议会进行分段处理。对这些分段数据帧，每个帧中TCP报文的序列号的值都是前一个帧的序列号值加上该帧中有效载荷的长度，这样能够保证这些分段数据帧是进行了排序的，方便服务端判断接收。
* 服务端接收到客户端发送的数据帧之后，会发送确认帧，其中确认号的值为上一个数据包中的序列号+所该数据包中所带数据的大小。
  + - 1. **关闭TCP客户端程序和TCP服务器应用程序**

在程序中结束本次连接，结果如下：

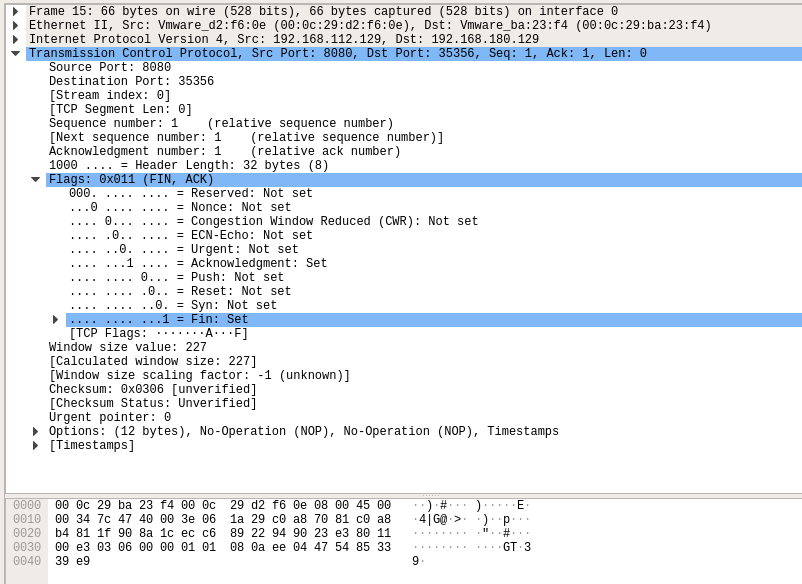


此时使用wireshark对TCP断开连接的四次挥手过程进行抓包：



下面对四次挥手过程中，传输的四个TCP报文进行分析：

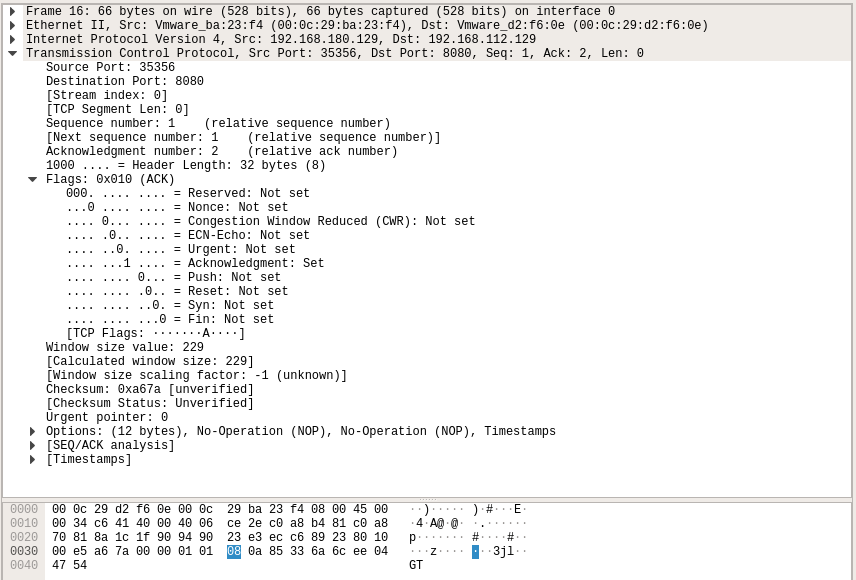
1. **15号帧：客户端发送断开连接请求**



* SYN比特位：0.
* ACK比特位：1.
* FIN比特位：1.
* 序列号字段：1.
* 确认号字段：1.

客户端想要结束连接时，通过调用socket.close()方法，向服务端发送一个FIN=1的数据包，使得服务器收到断开连接的请求

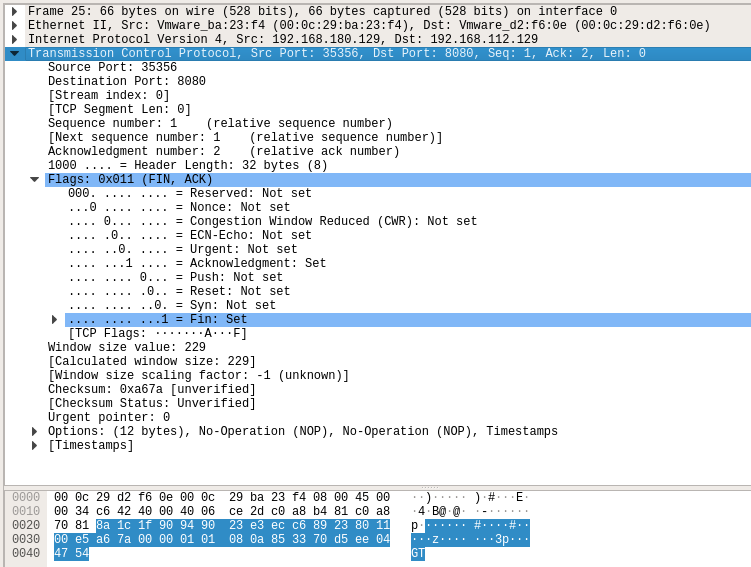
1. **16号帧：服务端发送断开确认信息**

****

* SYN比特位：0.
* ACK比特位：1.
* FIN比特位：0.
* 序列号字段：1.
* 确认号字段：2.

该报文中，ACK位的值为1，表示判断确认号是否有效，确认号的值为2，即15号帧的序列号加一。服务器收到客户端的FIN数据包之后，发送一个ACK包给客户端，确认号为上一个包序列号+1，表示是对该包的确认信息。

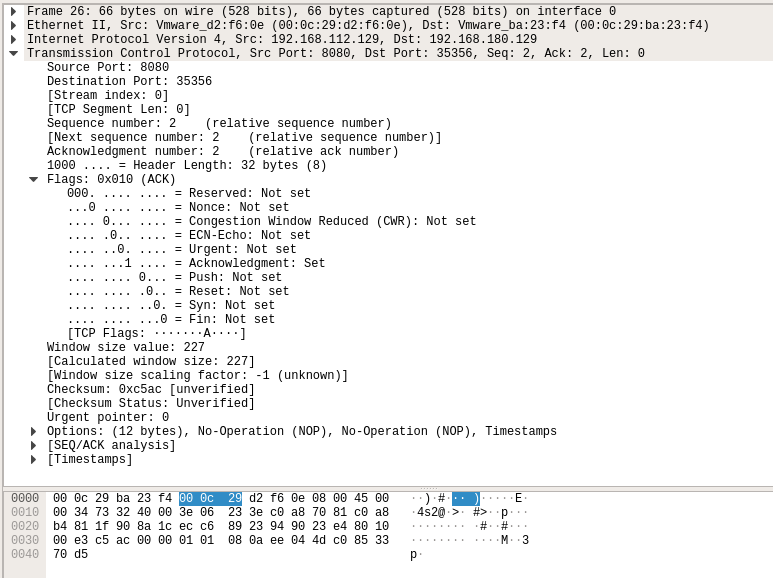
1. **25号帧：服务端发送断开连接请求**



* SYN比特位：0.
* ACK比特位：1.
* FIN比特位：0.
* 序列号字段：1.
* 确认号字段：2.

该帧的作用与15号帧相同，区别是该帧是由服务器发送的，作用是向客户端发送断开连接的请求。其中TCP报文的确认号为2，这与上一个帧（16号帧）中的确认号是相同的，序列号也与16号帧的序列号相同。

1. **25号帧：服务端发送断开连接请求**



* SYN比特位：0.
* ACK比特位：1.
* FIN比特位：0.
* 序列号字段：2.
* 确认号字段：2.

客户端收到了服务端的断开连接请求之后，发送一个ACK包到服务器。通过查看真实的序列号、确认号可以知道，该帧中的序列号取值与上一个帧（25号帧）的确认号相同，确认号的取值为上一个帧的序列号加一。

1. **附录**
   * + 1. **服务端程序：**
2. **SocketService.java**

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public** **class** SocketService {

**public** **static** **void** main(String args[])**throws** Exception {

ServerSocket serverSocket = **new** ServerSocket(8080); //服务端程序端口为8080

System.***out***.println("服务器启动成功");

**while** (**true**) {

Socket socket= serverSocket.accept(); //等待客户端发来连接请求

System.***out***.println("与客户端建立连接： " + socket.getInetAddress() + ":" +socket.getPort());

**new** Thread(**new** ServerThread(socket)).start(); //连接建立后，创建新的线程进行数据传输

}

}

}

1. **ServerThread.java**

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.net.Socket;

**public** **class** ServerThread **implements** Runnable {

**public** Socket socket;

**public** ServerThread (Socket socket) {

**this**.socket = socket;

}

@Override

**public** **void** run() {

**try** {

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));

**while** (**true**) {

String str = br.readLine(); //每一次从客户端中获取消息

**if**(str!=**null**){ //消息不为空，则输出内容

System.***out***.println(str);

}

**else**{ //消息为空，则断开本链接

System.***out***.println("客户端断开连接");

socket.close();

**break**;

}

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

* + - 1. **客户端程序：**

**1） SocketClient1.java**

**import** java.io.BufferedReader;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**import** java.net.Socket;

**public** **class** SocketClient1 {

**public** **static** **void** main(String args[])**throws** Exception{

Socket socket = **new** Socket("192.168.112.129", 8080); //服务端的IP地址以及端口号

System.***out***.println("Client1 连接成功");

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.***in***));

PrintWriter pw = **new** PrintWriter(socket.getOutputStream());

**while**(**true**){

String str = br.readLine(); //从控制台中输入将要发送的消息

**if**(str.equals("quit")){ //如果输入quit，则客户端断开连接

**break**;

}

pw.println("Client1："+str);

pw.flush();

}

socket.close();

}

}

**2） SocketClient2.java**

**import** java.io.BufferedReader;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**import** java.net.Socket;

**public** **class** SocketClient2 {

**public** **static** **void** main(String args[])**throws** Exception{

Socket socket = **new** Socket("192.168.112.129", 8080); //服务端的IP地址以及端口号

System.***out***.println("Client21 连接成功");

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.***in***));

PrintWriter pw = **new** PrintWriter(socket.getOutputStream());

**while**(**true**){

String str = br.readLine(); //从控制台中输入将要发送的消息

**if**(str.equals("quit")){ //如果输入quit，则客户端断开连接

**break**;

}

pw.println("Client2："+str);

pw.flush();

}

socket.close();

}

}

1. **SocketClient3.java**

**import** java.io.PrintWriter;

**import** java.net.Socket;

**public** **class** SocketClient3 {

**public** **static** **void** main(String args[])**throws** Exception{

Socket socket = **new** Socket("192.168.112.129", 8080); //服务端的IP地址以及端口号

System.***out***.println("Client3 连接成功");

PrintWriter pw = **new** PrintWriter(socket.getOutputStream());

StringBuffer longstr = **new** StringBuffer("Client3: "); //longstr长度为16000，由字母a填充

**for**(**int** i=0;i<1600;i++){

longstr.append("aaaaaaaaaa");

}

pw.println(longstr.toString());

pw.flush();

socket.close();

}

}