

# 復旦大學

数据通信与计算机网络 Lab3:运输层

王傲

15300240004

数据通信与计算机网络

COMP130017. 01

指导教师：肖晓春

**目录:**

0. 目录 .....	2
1. TCP 协议 .....	3
1.1 TCP 协议介绍实验 .....	3
1.1.1 .....	3
1.1.2 .....	3
1.1.3 .....	5
1.1.4 .....	6
1.1.5 .....	6
1.1.6 .....	8
1.2 TCP 重传 .....	8
1.2.1 .....	8
1.2.2 .....	9
1.2.3 .....	10
1.2.4 .....	11
1.2.5 .....	12
1.2.6 .....	13
1.2.7 .....	14
1.2.8 .....	15
1.2.9 .....	16
1.3 TCP 与 UDP 的比较 .....	16
1.3.1 .....	18
1.3.2 .....	18
1.3.3 .....	19
1.4 研究与讨论 .....	20
1.4.1 .....	20
1.4.2 .....	20
2. TCP 流和 UDP 流的竞争 .....	21
2.1 .....	21
2.2 .....	23
2.3 .....	23
2.4 .....	24
2.5 .....	26
3. 参考资料 .....	27

## 1. TCP 协议

### 1.1 TCP 协议介绍实验

实验步骤：

实验平台为 Macbook Pro (Mid 2015) , 使用 Wireshark 版本为 2.4.1, 使用 VMware Fusion 10.0.1 装载 Windows 10 1703 虚拟机。服务器版本为 Windows Server 2012 R2。

本机和服务器连在复旦内网内，本机 IP 地址为 10.221.121.48，服务器地址为 10.141.208.237。由于主机使用 DHCP 协议，在局域网内的 IP 地址可能会变动。

在服务器 10.141.208.237 上执行 `pcattcp -r` 指令进行监听，在本机上执行 `pcattcp -t -n 1`

**10.141.208.237** 指令向服务器发送一个 8192 字节大小的缓冲区的数据。结果保存在 TCP.pcapng 中。

相关网络进程无法完全关闭，用 Wireshark 查看包结果时可能需要使用相关过滤器。

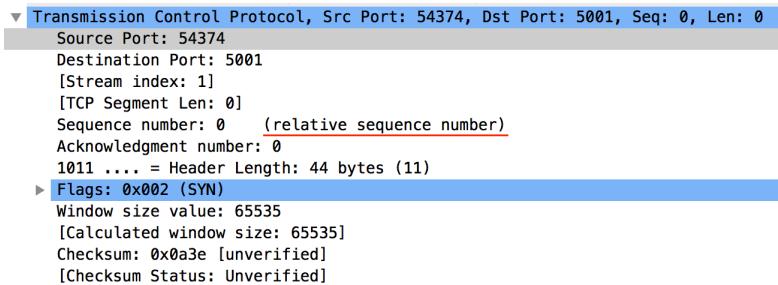
实验分析：

**1.1.1 对照TCP报文段格式，找到TCP三次握手连接报文（SYN, SYNACK, ACK），将列表框中的这三个分组截图，分别找到SYN和SYNACK对应的序列号，以及确认号。**

截图如下：

45 1.663562 10.221.121.48	10.141.208.237	TCP	78 54374 → 5001 [SYN] Seq=0 Win=65535 Len=32 TSval=965510406 TSecr=0 SACK_PERM=1
46 1.667566 10.141.208.237	10.221.121.48	TCP	74 5001 → 54374 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1260 WS=256 SACK_PERM=1 TSval=425089473 TSecr=5
47 1.667636 10.221.121.48	10.141.208.237	TCP	66 54374 → 5001 [ACK] Seq=1 Ack=1 Win=132288 Len=0 TSval=965510410 TSecr=425089473
48 1.668623 10.221.121.48	10.141.208.237	TCP	1314 54374 → 5001 [ACK] Seq=1 Ack=1 Win=132288 Len=1248 TSval=965510410 TSecr=425089473

这里序号是相对序号 (SYN报文序号为0) , 而不是绝对序号, 可能由Wireshark进行了转换:



可以看见，主机 10.221.121.48 发送了第一次握手 SYN 报文，报文序号为 0，没有 ACK 报文；服务器 10.141.208.237 接收到了第一次握手报文，向主机发送了第二次握手 SYN+ACK 报文，序号 Seq 为 0，表示是己方第一次发送，确认号 Ack 为 1，表明收到了主机的 0 号报文且报文长度为 0，期望下次收到的字节序号为 1；主机收到了第二次握手报文，向服务器发送了第三次握手 ACK 报文，序列号 Seq 为 1，符合服务器确认号的要求，确认号 Ack 为 1，表明收到了服务器的 0 号报文且报文长度为 0，期望下次收到的字节序号为 1。

这里特别截图了第四个报文，可以看见第三次握手 ACK 报文长度为 0，第四个报文同样由主机发出，序列号和确认号与第三次握手相同，只不过这次长度为 1248，包含了数据。

**1.1.2 注意三次握手也会用来协商连接的某些属性，如：MSS，观察并指出这些属性在TCP报文段的位置。**

TCP报文的格式如下：



可以看见，很多对于连接来说非常重要的属性，比如端口号、数据偏移、窗口大小等，直接被放在了表头里。其他的一些属性，比如MSS，被放在了“选项”（Option）中。

具体的，第一次握手时，规定了MSS为1460个字节（最大值）：

```
▼ Transmission Control Protocol, Src Port: 54374, Dst Port: 5001, Seq: 0, Len: 0
  Source Port: 54374
  Destination Port: 5001
  [Stream index: 1]
  [TCP Segment Len: 0]
  Sequence number: 0      (relative sequence number)
  Acknowledgment number: 0
  1011 .... = Header Length: 44 bytes (11)
  ▶ Flags: 0x002 (SYN)
  Window size value: 65535
  [Calculated window size: 65535]
  Checksum: 0x0a3e {unverified}
  [Checksum Status: Unverified]
  Urgent pointer: 0
  ▶ Options: (24 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP), Timestamps, SACK permitted, End of Option List (EOL)
    ▶ TCP Option - Maximum segment size: 1460 bytes
    ▶ TCP Option - No-Operation (NOP)
    ▶ TCP Option - Window scale: 5 (multiply by 32)
    ▶ TCP Option - No-Operation (NOP)
    ▶ TCP Option - No-Operation (NOP)
    ▶ TCP Option - Timestamps: TStamp 965510406, TSectr 0
    ▶ TCP Option - SACK permitted
    ▶ TCP Option - End of Option List (EOL)
```

第二次握手时，将MSS规定为1260个字节：

```
▼ Transmission Control Protocol, Src Port: 5001, Dst Port: 54374, Seq: 0, Ack: 1, Len: 0
  Source Port: 5001
  Destination Port: 54374
  [Stream index: 1]
  [TCP Segment Len: 0]
  Sequence number: 0      (relative sequence number)
  Acknowledgment number: 1      (relative ack number)
  1010 .... = Header Length: 40 bytes (10)
  ▶ Flags: 0x012 (SYN, ACK)
  Window size value: 8192
  [Calculated window size: 8192]
  Checksum: 0x22f4 {unverified}
  [Checksum Status: Unverified]
  Urgent pointer: 0
  ▶ Options: (20 bytes), Maximum segment size, No-Operation (NOP), Window scale, SACK permitted, Timestamps
    ▶ TCP Option - Maximum segment size: 1260 bytes
    ▶ TCP Option - No-Operation (NOP)
    ▶ TCP Option - Window scale: 8 (multiply by 256)
    ▶ TCP Option - SACK permitted
    ▶ TCP Option - Timestamps: TStamp 425089473, TSectr 965510406
    ▶ [SEQ/ACK analysis]
```

所以第三次握手结束后，第四次握手开始传输数据时，数据长度为1248个字节，没有超过1260：

```
▼ Transmission Control Protocol, Src Port: 54374, Dst Port: 5001, Seq: 1, Ack: 1, Len: 1248
  Source Port: 54374
  Destination Port: 5001
  [Stream index: 1]
  [TCP Segment Len: 1248]
  Sequence number: 1      (relative sequence number)
  [Next sequence number: 1249      (relative sequence number)]
  Acknowledgment number: 1      (relative ack number)
  1000 .... = Header Length: 32 bytes (8)
```

所以我们可以看出，三次握手时就已经开始确定连接的一些属性，不同的属性位置不同，最基础、最重要的属性直接定义在表头中，其他的属性作为可选项放在选项Option中。没有属性放在可选项中的话，Option就会被0填充。

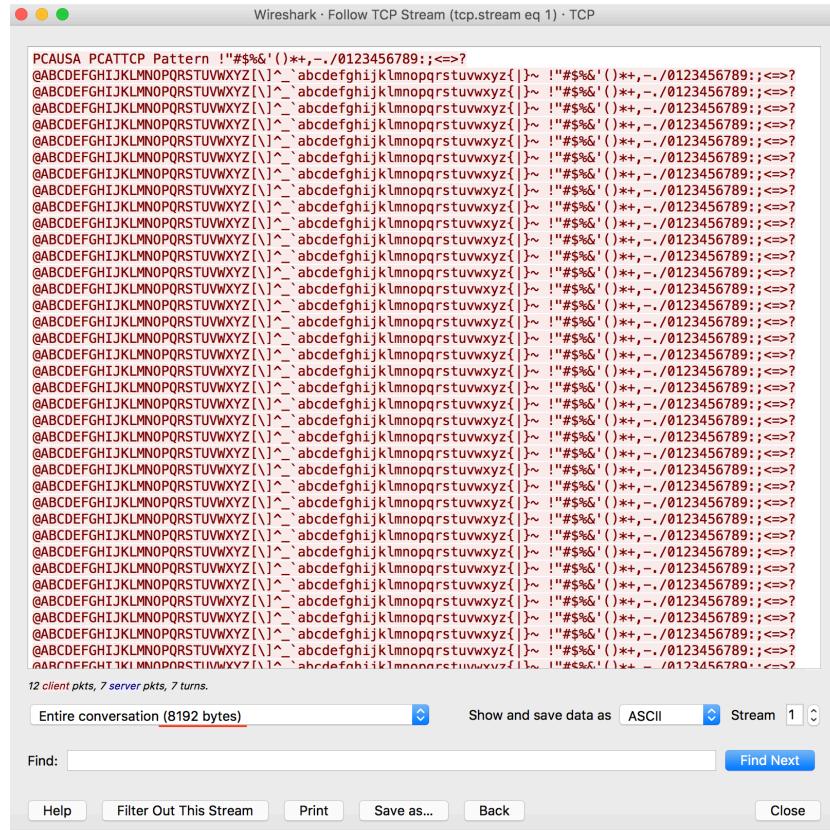
### 1.1.3 观察TCP数据流，计算一共应用层发送数据大小，使用 分析 ->追踪TCP流 观察发生的双向TCP或者单向TCP内容。

由于是主机向服务器发送数据，因此使用过滤规则 `ip.src==10.221.121.48 and tcp.port==5001` 来找到发送的数据包，计算长度：

No.	Time	Source	Destination	Protocol	Length	Info
45	1.663562	10.221.121.48	10.141.208.237	TCP	78	54374 → 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=965510406 TSecr=0 SACK_PERM=1
47	1.667636	10.221.121.48	10.141.208.237	TCP	66	54374 → 5001 [ACK] Seq=1 Ack=1 Win=132288 Len=0 TSval=965510410 TSecr=425089473
48	1.668623	10.221.121.48	10.141.208.237	TCP	1314	54374 → 5001 [ACK] Seq=1 Ack=1 Win=132288 Len=1248 TSval=965510410 TSecr=425089473
49	1.668624	10.221.121.48	10.141.208.237	TCP	278	54374 → 5001 [PSH, ACK] Seq=1249 Ack=1 Win=132288 Len=212 TSval=965510410 TSecr=425089473
50	1.668670	10.221.121.48	10.141.208.237	TCP	1314	54374 → 5001 [ACK] Seq=1461 Ack=1 Win=132288 Len=1248 TSval=965510411 TSecr=425089473
51	1.668670	10.221.121.48	10.141.208.237	TCP	278	54374 → 5001 [PSH, ACK] Seq=2789 Ack=1 Win=132288 Len=212 TSval=965510411 TSecr=425089473
52	1.668678	10.221.121.48	10.141.208.237	TCP	1314	54374 → 5001 [ACK] Seq=2921 Ack=1 Win=132288 Len=1248 TSval=965510411 TSecr=425089473
53	1.668679	10.221.121.48	10.141.208.237	TCP	278	54374 → 5001 [PSH, ACK] Seq=169 Ack=1 Win=132288 Len=212 TSval=965510411 TSecr=425089473
58	1.671659	10.221.121.48	10.141.208.237	TCP	1314	54374 → 5001 [ACK] Seq=4381 Ack=1 Win=132288 Len=1248 TSval=965510413 TSecr=425089473
59	1.671659	10.221.121.48	10.141.208.237	TCP	1314	54374 → 5001 [ACK] Seq=5629 Ack=1 Win=132288 Len=1248 TSval=965510413 TSecr=425089473
61	1.672698	10.221.121.48	10.141.208.237	TCP	1314	54374 → 5001 [ACK] Seq=6877 Ack=1 Win=132288 Len=1248 TSval=965510414 TSecr=425089474
62	1.672699	10.221.121.48	10.141.208.237	TCP	134	54374 → 5001 [FIN, PSH, ACK] Seq=8125 Ack=1 Win=132288 Len=68 TSval=965510414 TSecr=425089474
67	1.678055	10.221.121.48	10.141.208.237	TCP	66	54374 → 5001 [ACK] Seq=8194 Ack=2 Win=132288 Len=0 TSval=965510419 TSecr=425089474

注意这里不能使用 Length 中的长度，因为包含了表头；要使用报文中 Len 的值，这是真正的数据段的长度。应用层发送的数据大小为  $6 \times 1248 + 3 \times 212 + 68 = 8192$  个字节。由于使用 pcattcp 的指令是发送一个缓冲区的数据，缓冲区大小为 8192 个字节，这个结果与发送的数据相符。

TCP 数据流如下：

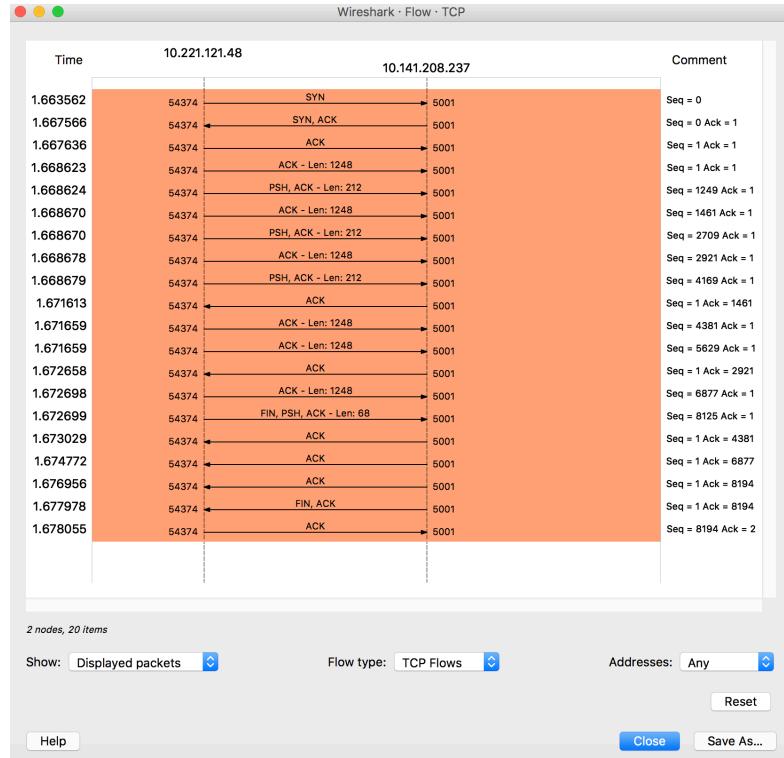


可以看见，Wireshark 统计发送的数据也为 8192 个字节，证明了这个结果的正确性。

发送的内容是一些无意义的数据，是由一些字符拼接生成的字符串。

### 1.1.4 使用 统计->流量图 观察TCP的所有通信过程，以及序列号和确认号。

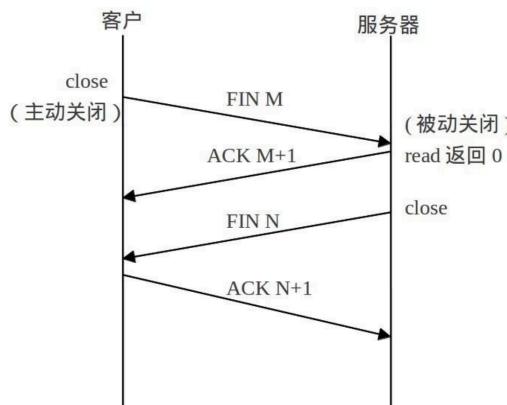
截图如下：



通信过程主要是主机10.221.121.48通过54374端口和服务器10.141.208.237通过5001端口完成的。主机向服务器发送8192个字节的数据。这里序号Seq是每次己方发送的字节流中第一个字节的编号，确认号Ack是期望从对方收到的下一个字节的编号。从这里可以看出，主机发送数据是不等待接收到ACK报文便直接发送下一个包的，所以所有发送的数据包的Seq逐步增长，而Ack均为1。同样的，服务器接收到数据包后，Ack逐步增长，而Seq均为1。直到发送方发送FIN，开始断开连接，接收方发送FIN+ACK，发送方最后发送ACK时Ack才增长到2。

### 1.1.5 观察TCP的连接关闭：找到FIN标志的报文段，说明其关闭过程。

TCP使用四次挥手断开连接：



这里因为是主机向服务器发送数据，主机相当于客户端，主动发起FIN，服务器被动接收。

第一次挥手：

具体的，主机在发送数据结束后，发送FIN报文：

```
▼ Transmission Control Protocol, Src Port: 54374, Dst Port: 5001, Seq: 8125, Ack: 1, Len: 68
  Source Port: 54374
  Destination Port: 5001
  [Stream index: 1]
  [TCP Segment Len: 68]
  Sequence number: 8125 (relative sequence number)
  [Next sequence number: 8194 (relative sequence number)]
  Acknowledgment number: 1 (relative ack number)
  1000 .... = Header Length: 32 bytes (8)
  ▶ Flags: 0x019 (FIN, PSH, ACK)
  Window size value: 4134
  [Calculated window size: 132288]
  [Window size scaling factor: 32]
  Checksum: 0xfc5a [unverified]
  [Checksum Status: Unverified]
  Urgent pointer: 0
  ▶ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  ▶ [SEQ/ACK analysis]
  TCP payload (68 bytes)
```

这里FIN报文中仍然包含着一部分数据。报文的序列号为8125，数据段长度为68，所以服务器Ack的值应为 $8125 + 68 + 1 = 8194$ 。

第二次挥手：

服务器接收到主机发送的FIN报文，发送ACK报文：

```
▼ Transmission Control Protocol, Src Port: 5001, Dst Port: 54374, Seq: 1, Ack: 8194, Len: 0
  Source Port: 5001
  Destination Port: 54374
  [Stream index: 1]
  [TCP Segment Len: 0]
  Sequence number: 1 (relative sequence number)
  Acknowledgment number: 8194 (relative ack number)
  1000 .... = Header Length: 32 bytes (8)
  ▶ Flags: 0x010 (ACK)
  Window size value: 258
  [Calculated window size: 66048]
  [Window size scaling factor: 256]
  Checksum: 0x4fed [unverified]
  [Checksum Status: Unverified]
  Urgent pointer: 0
  ▶ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  ▶ [SEQ/ACK analysis]
```

这里ACK报文中Ack的值为8194，证明是对FIN报文的回应。

第三次挥手：

服务器发送ACK报文后，向主机发送FIN报文：

```
▼ Transmission Control Protocol, Src Port: 5001, Dst Port: 54374, Seq: 1, Ack: 8194, Len: 0
  Source Port: 5001
  Destination Port: 54374
  [Stream index: 1]
  [TCP Segment Len: 0]
  Sequence number: 1 (relative sequence number)
  Acknowledgment number: 8194 (relative ack number)
  1000 .... = Header Length: 32 bytes (8)
  ▶ Flags: 0x011 (FIN, ACK)
  Window size value: 258
  [Calculated window size: 66048]
  [Window size scaling factor: 256]
  Checksum: 0x4fec [unverified]
  [Checksum Status: Unverified]
  Urgent pointer: 0
  ▶ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
```

注意Ack的值仍为8194，与第二次挥手相同。发送FIN后，服务器断开连接。

第四次挥手：

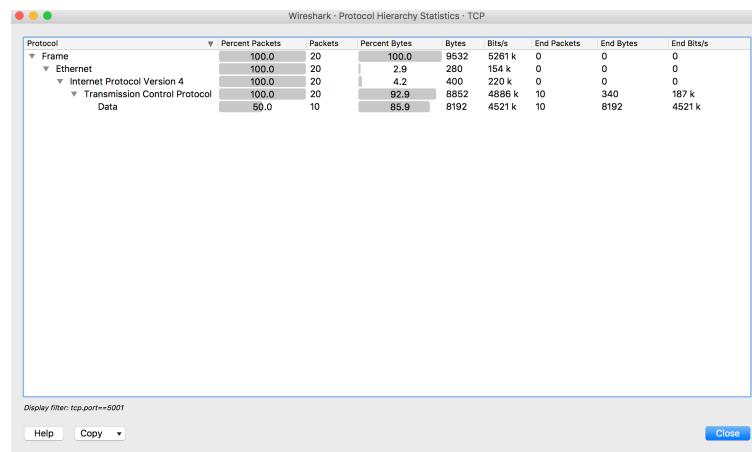
主机接收到服务器发送的FIN报文，向服务器发送ACK报文，结束这次TCP连接：

```
▼ Transmission Control Protocol, Src Port: 54374, Dst Port: 5001, Seq: 8194, Ack: 2, Len: 0
  Source Port: 54374
  Destination Port: 5001
  [Stream index: 1]
  [TCP Segment Len: 0]
  Sequence number: 8194 (relative sequence number)
  Acknowledgment number: 2 (relative ack number)
  1000 .... = Header Length: 32 bytes (8)
  ► Flags: 0x010 (ACK)
  Window size value: 4134
  [Calculated window size: 132288]
  [Window size scaling factor: 32]
  Checksum: 0x40c3 [unverified]
  [Checksum Status: Unverified]
  Urgent pointer: 0
  Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  ► [SEQ/ACK analysis]
```

这次ACK报文的Seq为8194，证明主机收到了服务器发送的FIN报文；Ack为2，对应服务器发送的Seq为1。四次挥手结束，TCP连接断开。

### 1.1.6 使用 统计->协议分级 菜单来观察统计信息。

截图如下：



可以看出最高等级的协议是TCP协议，数据有8192个字节。

## 1.2 TCP重传

实验步骤：

接收方使用指令 `pcattcp -r -l 1000` 将缓冲区大小设为1000个字节；发送方使用指令 `pcattcp -t -l 1000 -n 50 IP address` 向接收方发送50个1000字节。这里尝试了很多次，很难抓到要求的包（重传的包只有一两个，而且没有快速重传），所以使用提供的data中的pcattcp\_retrans\_r.cap和pcattcp\_retrans\_t.cap。

发送方IP地址为192.168.0.100，接收方IP地址为192.168.0.102。

实验分析：

### 1.2.1 指出三次握手过程中C/S协商SACK字段。

SACK (Selective Acknowledgment, 选择性确认)技术, 使 TCP 只重新发送交互过程中丢失的包, 不用像 Go-Back-N 协议一样发送后续所有的包, 而且提供相应机制使接收方能告诉发送方哪些数据丢失, 哪些数据重发了, 哪些数据已经提前收到了。如此大大提高了客户端与服务器端数据交互的效率。

这里 SACK 只出现在 SYN 位置为 1 的包中, 即第一次握手的 SYN 报文和第二次握手的 SYN+ACK 报文。

第一次握手:

```

> Internet Protocol Version 4, Src: 192.168.0.100, Dst: 192.168.0.102
  ▼ Transmission Control Protocol, Src Port: 4480, Dst Port: 5001, Seq: 0, Len: 0
    Source Port: 4480
    Destination Port: 5001
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0 (relative sequence number)
    Acknowledgment number: 0
    0111 .... = Header Length: 28 bytes (7)
    ▶ Flags: 0x002 (SYN)
    Window size value: 64240
    [Calculated window size: 64240]
    Checksum: 0xe40e [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    ▶ Options: (8 bytes), Maximum segment size, No-Operation (NOP), No-Operation (NOP), SACK permitted
      ▶ TCP Option - Maximum segment size: 1460 bytes
      ▶ TCP Option - No-Operation (NOP)
      ▶ TCP Option - No-Operation (NOP)
      ▶ TCP Option - SACK permitted
        Kind: SACK Permitted (4)
        Length: 2

```

第二次握手:

```

> Internet Protocol Version 4, Src: 192.168.0.102, Dst: 192.168.0.100
  ▼ Transmission Control Protocol, Src Port: 5001, Dst Port: 4480, Seq: 0, Ack: 1, Len: 0
    Source Port: 5001
    Destination Port: 4480
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0 (relative sequence number)
    Acknowledgment number: 1 (relative ack number)
    0111 .... = Header Length: 28 bytes (7)
    ▶ Flags: 0x012 (SYN, ACK)
    Window size value: 17520
    [Calculated window size: 17520]
    Checksum: 0x9de1 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    ▶ Options: (8 bytes), Maximum segment size, No-Operation (NOP), No-Operation (NOP), SACK permitted
      ▶ TCP Option - Maximum segment size: 1460 bytes
      ▶ TCP Option - No-Operation (NOP)
      ▶ TCP Option - No-Operation (NOP)
      ▶ TCP Option - SACK permitted
        Kind: SACK Permitted (4)
        Length: 2
    ▶ [SEQ/ACK analysis]

```

这里 SACK 出现在选项 Option 中, 类型值为 4, 长度为 2。客户端和服务器通过前两次握手进行协商, 分别表明支持 SACK, 即选择重传。

### 1.2.2 使用显示过滤器 `tcp.analysis.retransmission` 来查看重传数据。

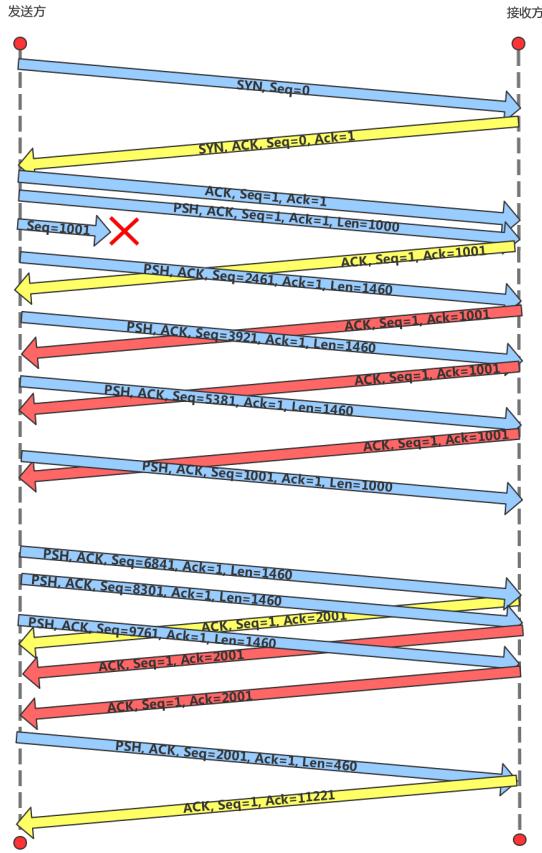
重传数据如下:

No.	Time	Source	Destination	Protocol	Length	Info
12	0.159616	192.168.0.100	192.168.0.102	TCP	1054	[TCP Fast Retransmission] 4480 → 5001 [PSH, ACK] Seq=1001 Ack=1 Win=64240 Len=1000
19	0.177841	192.168.0.100	192.168.0.102	TCP	514	[TCP Fast Retransmission] 4480 → 5001 [ACK] Seq=2001 Ack=1 Win=64240 Len=460
27	0.191856	192.168.0.100	192.168.0.102	TCP	834	[TCP Fast Retransmission] 4480 → 5001 [PSH, ACK] Seq=11221 Ack=1 Win=64240 Len=780
39	0.225238	192.168.0.100	192.168.0.102	TCP	754	[TCP Fast Retransmission] 4480 → 5001 [PSH, ACK] Seq=21301 Ack=1 Win=64240 Len=700
41	1.828318	192.168.0.100	192.168.0.102	TCP	1514	[TCP Retransmission] 4480 → 5001 [PSH, ACK] Seq=22001 Ack=1 Win=64240 Len=1460
53	1.863841	192.168.0.100	192.168.0.102	TCP	534	[TCP Fast Retransmission] 4480 → 5001 [PSH, ACK] Seq=31521 Ack=1 Win=64240 Len=480
55	3.140244	192.168.0.100	192.168.0.102	TCP	1514	[TCP Retransmission] 4480 → 5001 [PSH, ACK] Seq=32001 Ack=1 Win=64240 Len=1460
67	3.175413	192.168.0.100	192.168.0.102	TCP	314	[TCP Fast Retransmission] 4480 → 5001 [PSH, ACK] Seq=41741 Ack=1 Win=64240 Len=260
69	4.015580	192.168.0.100	192.168.0.102	TCP	1514	[TCP Retransmission] 4480 → 5001 [PSH, ACK] Seq=42001 Ack=1 Win=64240 Len=1460

重传均由发送方发起, 包括重传和快速重传。

### 1.2.3 对照发送方和接收方的分组信息，完成双方通信的图示化。

如图：



这里分析前20个包。蓝色表示发送方发送的数据，黄色表示正常的ACK包，红色表示冗余的ACK包。前20个包里发生了两次快速重传。

第一次，发送方发送的Seq=1001, Len=1460（长度由下一个发送方发送的包的Seq判断）的包由于某些原因丢失，之后接收方每收到一个包，便发送一个Ack为1001的包表示未接收到；连续发送三次后（第二个红色箭头），发送方收到三个冗余Ack，开启快速重传，重新发送了一个Seq=1001的包。这里有几点要注意：1. 第一次快速重传的包中数据的长度Len是1000而不是1460，这也是第二次快速重传的原因。2. 发送方接收到三个冗余Ack就立即重传了，但是在重传的包到达接收方之前，接收方又收到了一个包，重新发送了一次冗余Ack，所以一共发送了四次冗余Ack（对应图中一个黄箭头和三个红箭头）。

第二次，接收方收到了前两千个字节，发送Ack为2001的包。而发送方从2461开始已经发送到了6841。由于发送方并不根据接收方的Ack来确定下一次发送的内容，所以没有发送Seq为2001的包。收到三个冗余Ack后，快速重传重新发送Seq为2001的包。这里同样有几点要注意：1. 第二次快速重传的包Seq为2001，Len为460，正好接上之前发送的Seq为2461的包。2. 双方都支持SACK，支持选择重传而不是Go-Back-N，所以一旦都到齐后，接收方可以读取出缓存中的数据，下一次的Ack直接从11221开始。

### 1.2.4 在分组中找到快速重传和重传的数据包，分析其产生的不同原因，为什么有这样的区别。

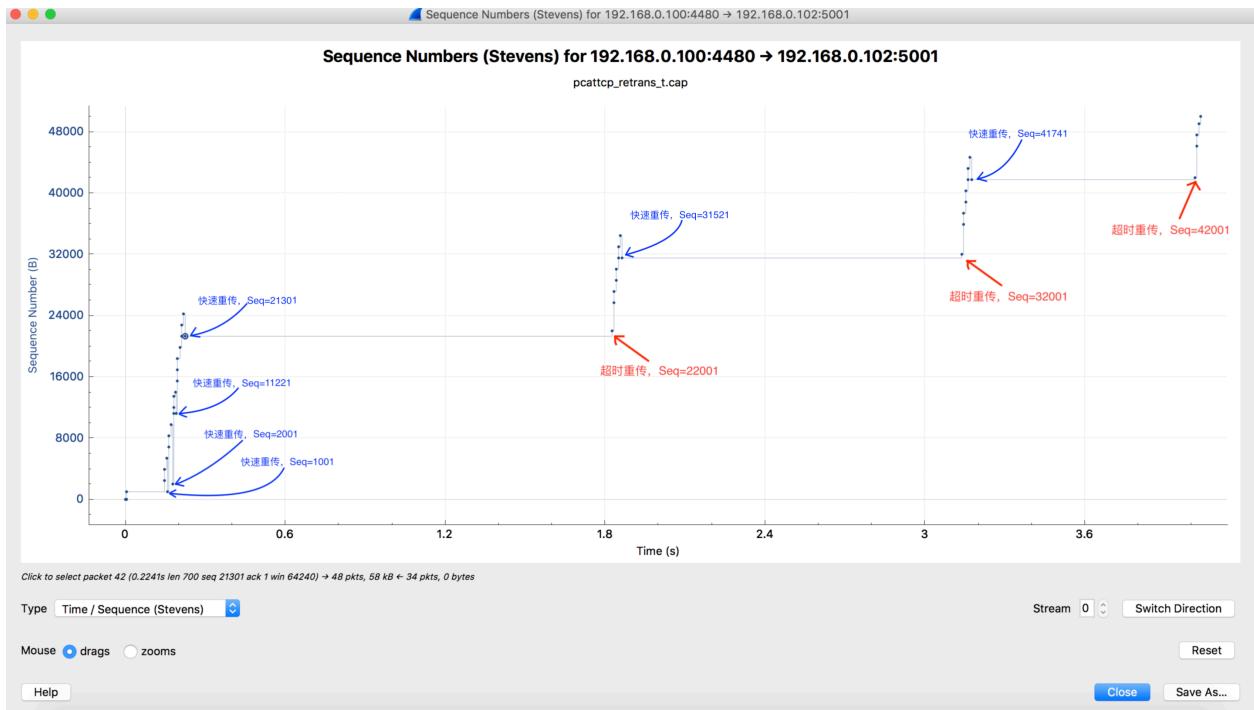
如下：

49 1.852590 192.168.0.100 192.168.0.102 TCP 1514 [TCP Previous segment not captured] 4480 → 5001 [PSH, ACK] Seq=32981 Ack=1 Win=1460
50 1.853709 192.168.0.102 192.168.0.100 TCP 66 [TCP Dup ACK 48#1] 5001 → 4480 [ACK] Seq=1 Ack=31521 Win=17520 Len=0 SLE=32981
51 1.858979 192.168.0.100 192.168.0.102 TCP 1514 4480 → 5001 [PSH, ACK] Seq=34441 Ack=1 Win=64240 Len=1460
52 1.860093 192.168.0.102 192.168.0.100 TCP 66 [TCP Dup ACK 48#2] 5001 → 4480 [ACK] Seq=1 Ack=31521 Win=17520 Len=0 SLE=32981
53 1.863841 192.168.0.100 192.168.0.102 TCP 534 [TCP Fast Retransmission] 4480 → 5001 [PSH, ACK] Seq=31521 Ack=1 Win=64240 Len=1460
54 2.046302 192.168.0.102 192.168.0.100 TCP 66 5001 → 4480 [ACK] Seq=1 Ack=32001 Win=17040 Len=0 SLE=32981 SRE=35901
55 3.140244 192.168.0.100 192.168.0.102 TCP 1514 [TCP Retransmission] 4480 → 5001 [PSH, ACK] Seq=32001 Ack=1 Win=64240 Len=1460

出现快速重传的原因最有可能的是数据包丢失，接收方发送三个冗余Ack后引起发送方快速重传。除此之外，发送方数据到达接收方的时间过长导致接收方发送三个Ack后才到也会产生快速重传（尽管可能性很小）。具体的快速重传的分析见1.2.3。

当抓到2次同一包数据时，Wireshark判断发生了重传，同时Wireshark没有抓到初传包的反馈Ack，因此Wireshark判定重传有效，标记为TCP Retransmission。基于软件的判定机制，如果抓包点在发送方的话，TCP重传一般为上行包，如果这时发送方没有收到接收方的反馈Ack，无从知晓接收方是否收到了初传包，RTO超时后触发客户端重传。此时存在2种情况：1. 服务端收到了初传包，只是下发的反馈Ack丢包，客户端没有收到；2. 服务端没有收到初传包，因此也就没有下发反馈Ack。

快速重传和重传的情况如下：



从图中可以看出，快速重传（有序号下降的地方）有6次，超时重传（平的部分）有3次。部分快速重传由于丢包造成，接收方发送三次冗余ACK导致快速重传；另一些由于重发的包被截断而不完整（比如丢失的包Len=1460，重发的包Len=1000），而发送方没有发送剩余部分，接收方发送

最有可能触发重传的原因是由于某些因素造成的计时器超时（丢包的话更有可能触发快速重传）。以图中55号重传包为例：

```

▼ Transmission Control Protocol, Src Port: 4480, Dst Port: 5001, Seq: 32001, Ack: 1, Len: 1460
  Source Port: 4480
  Destination Port: 5001
  [Stream index: 0]
  [TCP Segment Len: 1460]
  Sequence number: 32001      (relative sequence number)
  [Next sequence number: 33461      (relative sequence number)]
  Acknowledgment number: 1      (relative ack number)
  0101 .... = Header Length: 20 bytes (5)
  ▶ Flags: 0x018 (PSH, ACK)
  Window size value: 64240
  [Calculated window size: 64240]
  [Window size scaling factor: -2 (no window scaling used)]
  Checksum: 0xaaf9c [unverified]
  [Checksum Status: Unverified]
  Urgent pointer: 0
  ▼ [SEQ/ACK analysis]
    [iRTT: 0.003267000 seconds]
    [Bytes in flight: 3900]
    [Bytes sent since last PSH flag: 1460]
  ▼ [TCP Analysis Flags]
    ▶ [Expert Info (Note/Sequence): This frame is a (suspected) retransmission]
    [The RTO for this segment was: 1.281265000 seconds]
    [RTO based on delta from frame: 51]
  TCP payload (1460 bytes)

```

图中信息说明了这次重传是由于 RTO 超时导致，初传包序号为 51，超时时间接近 1.28s。

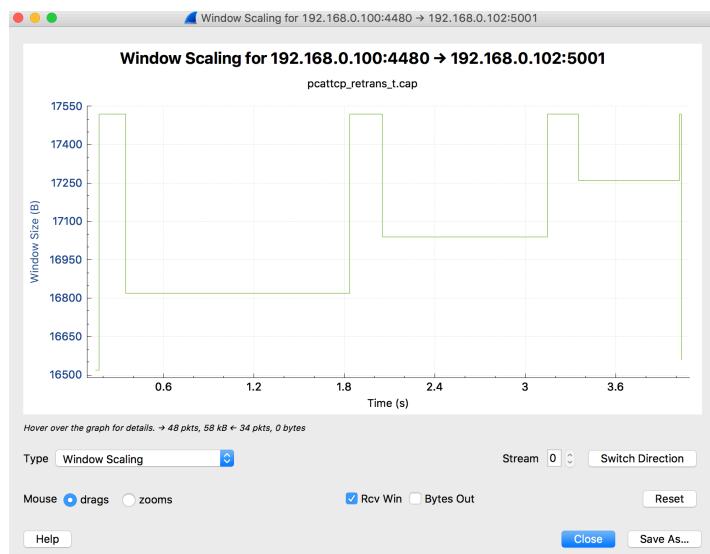
这样区别的原因在于同样对于丢包造成的重传，重传需要计时器等待一段时间才能重传，而快速重传只需要收到三个冗余 ACK 就重传。从图中可以看出，快速重传比重传要快很多。计时器定时是最基础的保证可靠的手段，而且能覆盖快速重传不能覆盖的情况，而快速重传则是为了提高效率。

### 1.2.5 查看接收方窗口大小变化情况，分析这些重传的原因是否有流量控制导致的？

从列表框中查看接收方的窗口大小变化情况：

47 1.846161	192.168.0.100	192.168.0.102	TCP	1514 4480 → 5001 [PSH, ACK] Seq=30061 Ack=1 Win=64240 Len=1460
48 1.847280	192.168.0.102	192.168.0.100	TCP	54 5001 → 4480 [ACK] Seq=1 Ack=31521 Win=17520 Len=0
49 1.852596	192.168.0.100	192.168.0.102	TCP	1514 [TCP Previous segment not captured] 4480 → 5001 [PSH, ACK] Seq=32981 Ack=1 Win=64240 Len=1460
50 1.853709	192.168.0.102	192.168.0.100	TCP	66 [TCP Dup ACK 48#1] 5001 → 4480 [ACK] Seq=1 Ack=31521 Win=17520 Len=0 SLE=32981 SRE=34441
51 1.858979	192.168.0.100	192.168.0.102	TCP	1514 4480 → 5001 [PSH, ACK] Seq=34441 Ack=1 Win=64240 Len=1460
52 1.860093	192.168.0.102	192.168.0.100	TCP	66 [TCP Dup ACK 48#2] 5001 → 4480 [ACK] Seq=1 Ack=31521 Win=17520 Len=0 SLE=32981 SRE=35901
53 1.863841	192.168.0.100	192.168.0.102	TCP	534 [TCP Fast Retransmission] 4480 → 5001 [PSH, ACK] Seq=31521 Ack=1 Win=64240 Len=480
54 2.046302	192.168.0.102	192.168.0.100	TCP	66 5001 → 4480 [ACK] Seq=1 Ack=32001 Win=17520 Len=0 SLE=32981 SRE=35901
55 3.140244	192.168.0.100	192.168.0.102	TCP	1514 [TCP Retransmission] 4480 → 5001 [PSH, ACK] Seq=32001 Ack=1 Win=64240 Len=1460
56 3.141567	192.168.0.102	192.168.0.100	TCP	54 5001 → 4480 [ACK] Seq=1 Ack=35901 Win=17520 Len=0
57 3.146620	192.168.0.100	192.168.0.102	TCP	1514 4480 → 5001 [PSH, ACK] Seq=35901 Ack=1 Win=64240 Len=1460

接收方窗口变化如图：



滑动窗口大小表明了接收方所能提供的缓冲区的大小。从包内信息可以看出，接收方的窗口大小一直在17000个字节上下。这比MSS的1460个字节大很多。所以，重传的原因应该与流量控制无关。

### 1.2.6 找到一个失序的分组，指出失序分组在接收方是如何选择确认的？

对于显式的 TCP Out-Of-Order 包，尝试自己抓取相应的包：

```
213 3.262745 211.157.135.205 10.221.121.48 TCP 1514 443 - 61268 [ACK] Seq=29415 Ack=1751 Win=66608 Len=1448 TSval=3869204354 TSeср=997366304 [TCP segment of a retransmission]
214 3.262752 211.157.135.205 10.221.121.48 TCP 1514 [TCP Previous segment not captured] 443 - 61268 [ACK] Seq=32311 Ack=1751 Win=66608 Len=1448 TSval=3869204354 TSeср=997366304
215 3.262754 211.157.135.205 10.221.121.48 TCP 1514 [TCP Out-of-Order] 443 - 61268 [ACK] Seq=30863 Ack=1751 Win=66608 Len=1448 TSval=3869204354 TSeср=997366304
```

正常来讲，发送方发送的包的序号 Seq 是逐步增长的。然而由于网络拥塞、时延太长、路径不同等原因，先产生的包可能后到达接收方，就产生了失序。对于同样的发送方，当一个分组的序号小于前面的分组的序号时，接收方就判断产生了失序。

图中 211.157.135.205 发送给主机 10.221.121.48 的报文，其中一个序号是 32311，下一个是 30863，小于前一个，所以接收方判断产生了失序。

如果是选择确认协议（SACK）的话，接收方会将失序分组放在缓存中，并发送包含期待字节所在分组的序号和已经接收到的失序分组的边界的 ACK 包，直到获得想要的包。之后，接收方在缓存中整理后按顺序将数据传递给上层调用者，并发送正确的期待的下一个字节的序号。

SACK 可以完成选择确认功能：

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.0.100	192.168.0.102	TCP	62	4480 - 5001 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
2	0.000432	192.168.0.102	192.168.0.100	TCP	62	5001 - 4480 [SYN, ACK] Seq=0 Ack=1 Win=17520 Len=0 MSS=1460 SACK_PERM=1
3	0.003267	192.168.0.100	192.168.0.102	TCP	68	4480 - 5001 [ACK] Seq=1 Ack=1 Win=64240 Len=0
4	0.005606	192.168.0.100	192.168.0.102	TCP	1054	4480 - 5001 [PSH, ACK] Seq=1 Ack=1 Win=64240 Len=1000
5	0.143277	192.168.0.102	192.168.0.100	TCP	54	5001 - 4480 [ACK] Seq=1 Ack=1001 Win=16520 Len=0
6	0.158784	192.168.0.100	192.168.0.102	TCP	1514	[TCP Previous segment not captured] 4480 - 5001 [PSH, ACK] Seq=2461 Ack=1 Win=64240 Len=1460
7	0.151839	192.168.0.102	192.168.0.100	TCP	66	[TCP Dup ACK 5#1] 5001 - 4480 [ACK] Seq=1 Ack=1001 Win=16520 Len=0 SLE=2461 SRE=3921
8	0.152666	192.168.0.100	192.168.0.102	TCP	1514	4480 - 5001 [PSH, ACK] Seq=3921 Ack=1 Win=64240 Len=1460
9	0.155729	192.168.0.102	192.168.0.100	TCP	66	[TCP Dup ACK 5#2] 5001 - 4480 [ACK] Seq=1 Ack=1001 Win=16520 Len=0 SLE=2461 SRE=5381
10	0.157514	192.168.0.100	192.168.0.102	TCP	1514	4480 - 5001 [PSH, ACK] Seq=5381 Ack=1 Win=64240 Len=1460
11	0.158627	192.168.0.102	192.168.0.100	TCP	66	[TCP Dup ACK 5#3] 5001 - 4480 [ACK] Seq=1 Ack=1001 Win=16520 Len=0 SLE=2461 SRE=6841
12	0.159616	192.168.0.100	192.168.0.102	TCP	1054	[TCP Fast Retransmission] 4480 - 5001 [PSH, ACK] Seq=1001 Ack=1 Win=64240 Len=1000
13	0.164887	192.168.0.100	192.168.0.102	TCP	1514	4480 - 5001 [PSH, ACK] Seq=6841 Ack=1 Win=64240 Len=1460
14	0.166002	192.168.0.102	192.168.0.100	TCP	66	5001 - 4480 [ACK] Seq=1 Ack=2001 Win=17520 Len=0 SLE=2461 SRE=8301
15	0.167082	192.168.0.100	192.168.0.102	TCP	1514	4480 - 5001 [PSH, ACK] Seq=8301 Ack=1 Win=64240 Len=1460
16	0.168193	192.168.0.102	192.168.0.100	TCP	66	[TCP Dup ACK 14#1] 5001 - 4480 [ACK] Seq=1 Ack=2001 Win=17520 Len=0 SLE=2461 SRE=9761
17	0.173292	192.168.0.100	192.168.0.102	TCP	1514	4480 - 5001 [PSH, ACK] Seq=9761 Ack=1 Win=64240 Len=1460
18	0.174465	192.168.0.102	192.168.0.100	TCP	66	[TCP Dup ACK 14#2] 5001 - 4480 [ACK] Seq=1 Ack=2001 Win=17520 Len=0 SLE=2461 SRE=11221
19	0.177841	192.168.0.100	192.168.0.102	TCP	514	[TCP Fast Retransmission] 4480 - 5001 [ACK] Seq=2001 Ack=1 Win=64240 Len=460
20	0.178275	192.168.0.102	192.168.0.100	TCP	54	5001 - 4480 [ACK] Seq=1 Ack=11221 Win=17520 Len=0
21	0.183524	192.168.0.100	192.168.0.102	TCP	1514	[TCP Previous segment not captured] 4480 - 5001 [PSH, ACK] Seq=12001 Ack=1 Win=64240 Len=1460
22	0.184657	192.168.0.102	192.168.0.100	TCP	66	[TCP Dup ACK 20#1] 5001 - 4480 [ACK] Seq=1 Ack=11221 Win=17520 Len=0 SLE=12001 SRE=13461
23	0.185143	192.168.0.100	192.168.0.102	TCP	594	4480 - 5001 [PSH, ACK] Seq=13461 Ack=1 Win=64240 Len=540

Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), SACK

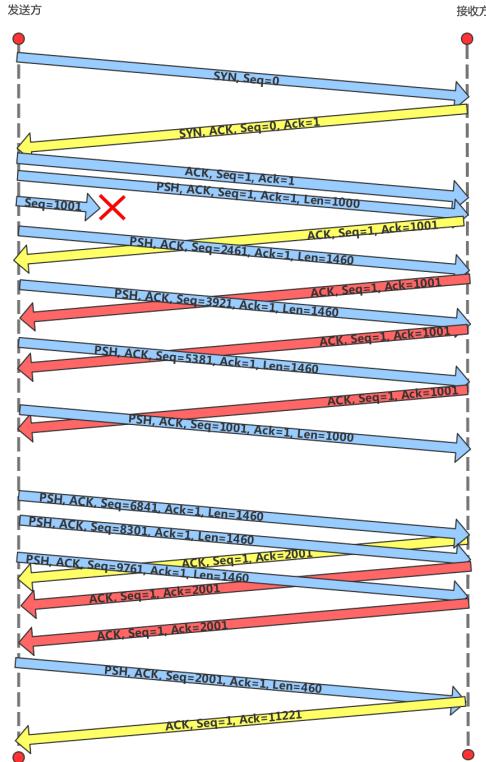
- ▶ TCP Option - No-Operation (NOP)
- ▶ TCP Option - No-Operation (NOP)
- ▼ TCP Option - SACK 2461-3921
  - Kind: SACK (5)
  - Length: 10
  - left edge = 2461 (relative)
  - right edge = 3921 (relative)
- ▶ [TCP SACK Count: 1]
- ▶ [SEQ/ACK analysis]

TCP 收到乱序数据后，会将其放入缓存中，然后发送重复 ACK 给对方，其中包含了失序分组所含数据的边界（SLE 和 SRE），告诉对方收到了哪些数据。对方如果收到多个重复的 ACK，认为发生丢包，TCP 会重传最后确认的包开始的后续包，而不是全部重发。使用 SACK 选项可以告知发包方收到了哪些数据，发包方收到这些信息后就会知道哪些数据丢失，然后立即重传丢失的部分。只有收到失序的分组时才会可能会发送 SACK。

SACK 中含有 kind(5), left edge 和 right edge, 这些选项参数告诉对方已经接收到并缓存的不连续的数据块, 注意都是已经接收的, 发送方可根据此信息检查究竟是哪个块丢失, 从而发送相应的数据块, 完成选择确认。

1.2.7 查看并说明, 一个失序丢失分组导致的后续多个连续失序分组, 在选择确认时的确认号的变化, 以及当丢失分组接收到之后, 确认号的变化情况。

仍以1.2.3为例:



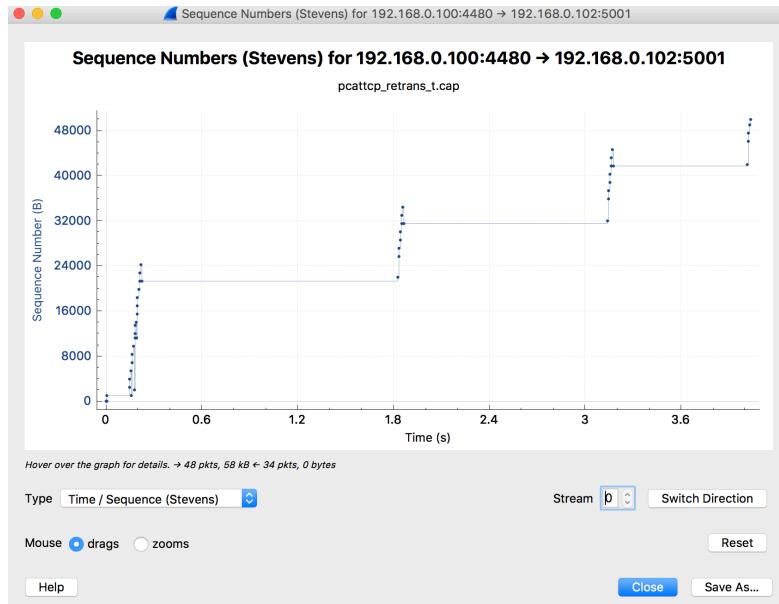
这里看第二次快速重传。发送方Seq=1001分组的丢失引发了第一次快速重传; 重传的时候没有完整的重传丢失的分组引起第二次快速重传。Seq=1001失序丢失分组引起了Seq=2461, 3921, 5381等分组的失序; 重传后的分组引起了Seq=6841, 8301, 9761等分组的失序。失序体现在接收方接收到了想要的分组后面的分组。

可以看出, 一个失序丢失分组导致的后续多个连续失序分组, 在选择确认时的确认号是接收方期望接收的最初失序分组的序号, 而不是后续多个连续失序分组对应的确认号。这与快速重传的机制有关。

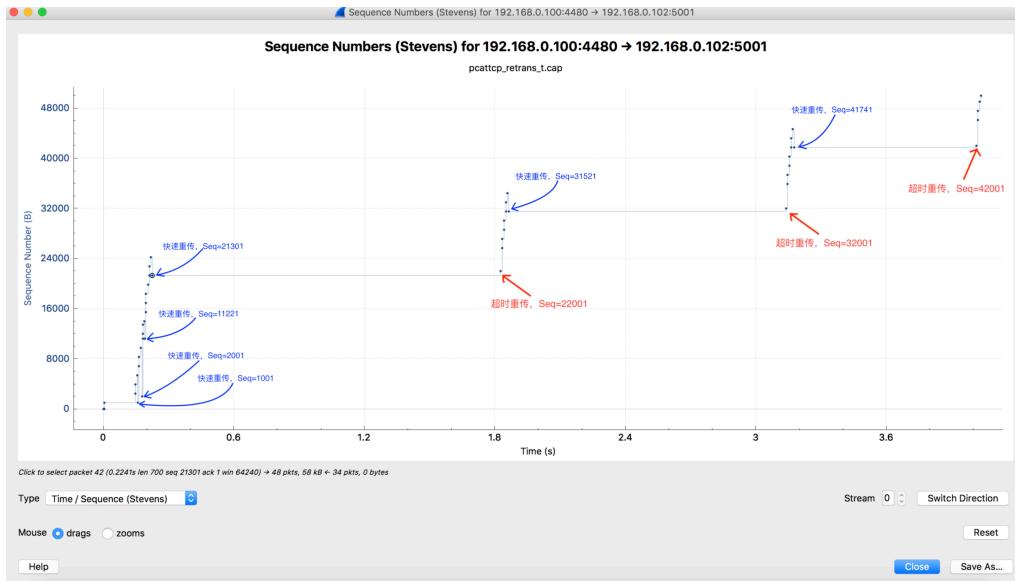
由于有选择重传, 接收方将后续失序的分组放在缓存中, 等失序分组到达后一起进行确认, 将最终的包对应的下一个包的首字节序号作为确认号。如第二次快速重传中, 接收方发送的冗余Ack一直是2001, 然而当失序分组Seq=2001, Len=460到达后, 接收方进行选择确认, 结合缓存中的数据, Ack直接变为11221进行确认, 说明并没有丢弃前面接收到的失序分组。

### 1.2.8 通过查看 统计/TCP流图形/时间序列 (Steven's) 可以看到分组的重传情况 (发送方)。

情况如下：



具体的分析如下：



在理想状况下，随着数据的匀速传输，Sequence 应该逐渐增加，从而形成一根较直的斜线。然而，我们看见，这里出现了很明显的断层，中间是平的，表明序列号没有增加。

出现这样的情况的原因是存在快速重传和重传。在图中，序列号一直上升后突然下降一部分，是因为快速重传：由于重传的分组的序号比前一个分组的序号小，并且快速重传所需的时间很短，所以在图上显示为突然下降。

在图中，水平的部分是因为重传：重传的分组的序号与前一个分组的序号相邻，然而由于重传，计时器的等待消耗了时间，两者在时间上差距较远，所以以时间为横坐标时会存在几段平行于横轴的线。

1.2.9 在获取的两个文件中，是否存在确认信息丢失的分组？如果有，他们是哪些分组，你是怎么知道的？

存在。丢失的、引起快速重传的分组，如：

```
1514 [TCP Previous segment not captured] 4480 → 5001 [PSH, ACK] Seq=22761 Ack=1
```

这里显示没有捕捉到相应的分组，并且接收方发送三次冗余ACK引起了快速重传，证明有信息丢失。

ACK 分组也存在可能丢失的情况，并且 ACK 分组不需要新的 ACK 分组确认，因此 ACK 丢失的话不会重传。ACK 可以看作发送端的时钟：如果连续的 ACK 丢失了，就会出现一个 ACK 确认了大块数据（序号迅速上升）的场景，由于前面 ACK 连续丢失，发送端久久未收到时钟反馈导致数据不能发送，但其实这段时间内需要发送但未发送的数据可能被积累了。如果收到一个确认很多数据的 ACK，表明上一个 ACK 很有可能丢失了。

### 1.3 TCP 与 UDP 的比较

实验步骤：

产生两个 TCP 流，一个有接收方，一个没有；产生两个 UDP 流，一个有接收方，一个没有。获得四个抓包文件。主机为发送方，IP 地址 10.221.121.48，服务器为接收方，IP 地址 10.141.208.237。

网络进程无法全部关闭，包内数据比较混乱，需要使用 Wireshark 提供的过滤器查看相应数据。

实验分析：

有接收方的 TCP 流：

发送方：

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -l 1000 -n 5 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
TCP Transmit Test
    Transmit : TCP -> 10.141.208.237:5001
    Buffer Size : 1000; Alignment: 16384/0
    TCP_NODELAY : DISABLED (0)
    Connect : Connected to 10.141.208.237:5001
    Send Mode : Send Pattern; Number of Buffers: 5
    Statistics : TCP -> 10.141.208.237:5001
    5000 bytes in 0.00 real seconds = 1.#J KB/sec +++
    numCalls: 5; msec/call: 0.00; calls/sec: 1.#J
PS C:\Users\wanga\Desktop>
```

接收方：

```
PS D:\wangao> .\PCATTCP.exe -r -l 1000
PCAUSA Test TCP Utility V2.01.01.08
TCP Receive Test
    Local Host : Gandhi
    ****
    Listening...: On port 5001

    Accept : TCP <- 10.221.121.48:64455
    Buffer Size : 1000; Alignment: 16384/0
    Receive Mode: Sinking (discarding) Data
    Statistics : TCP <- 10.221.121.48:64455
    5000 bytes in 0.00 real seconds = 1.#J KB/sec +++
    numCalls: 8; msec/call: 0.00; calls/sec: 1.#J
PS D:\wangao> -
```

有接收方的 UDP 流：

发送方：

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -l 1000 -n 5 -u 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
UDP Transmit Test
  Transmit : UDP -> 10.141.208.237:5001
  Buffer Size : 1000; Alignment: 16384/0
  Send Mode : Send Pattern; Number of Buffers: 5
  Statistics : UDP -> 10.141.208.237:5001
  5000 bytes in 0.00 real seconds = 1.#J KB/sec +++
  numCalls: 7; msec/call: 0.00; calls/sec: 1.#J
PS C:\Users\wanga\Desktop>
```

接收方：

```
PS D:\wangao> .\PCATTCP.exe -r -l 1000 -u
PCAUSA Test TCP Utility V2.01.01.08
UDP Receive Test
  Protocol : UDP
  Port : 5001
  Buffer Size : 1000; Alignment: 16384/0
  recvfrom : UDP <- 10.221.121.48:51169
  Statistics : UDP <- 10.221.121.48:51169
  5000 bytes in 0.00 real seconds = 1.#J KB/sec +++
  numCalls: 6; msec/call: 0.00; calls/sec: 1.#J
PS D:\wangao>
```

无接收方的 TCP 流：

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -l 1000 -n 5 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
TCP Transmit Test
  Transmit : TCP -> 10.141.208.237:5001
  Buffer Size : 1000; Alignment: 16384/0
  TCP_NODELAY : DISABLED (0)
*** Winsock Error: connect Failed; Error: 10060 (0x0000274C)
```

无接收方的 UDP 流：

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -l 1000 -n 5 -u 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
UDP Transmit Test
  Transmit : UDP -> 10.141.208.237:5001
  Buffer Size : 1000; Alignment: 16384/0
  Send Mode : Send Pattern; Number of Buffers: 5
  Statistics : UDP -> 10.141.208.237:5001
  5000 bytes in 0.01 real seconds = 325.52 KB/sec +++
  numCalls: 7; msec/call: 2.19; calls/sec: 466.67
```

可以看出，TCP发送方返回应用层的信息是连接失败；而UDP不返回任何信息，与有接收方的结果一样。TCP还进行了多次连接尝试：

tcp.port==5001							
No.	Time	Source	Destination	Protocol	Length	Info	
80	2.616331	10.221.121.48	10.141.208.237	TCP	78	64484 -> 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013476461 TScrn=0 SACK_PERM=1	
135	3.621586	10.221.121.48	10.141.208.237	TCP	78	[TCP Retransmission] 64484 -> 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013477461 TS...	
177	4.626895	10.221.121.48	10.141.208.237	TCP	78	[TCP Retransmission] 64484 -> 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013478461 TS...	
226	5.633652	10.221.121.48	10.141.208.237	TCP	78	[TCP Retransmission] 64484 -> 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013479461 TS...	
304	6.639328	10.221.121.48	10.141.208.237	TCP	78	[TCP Retransmission] 64484 -> 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013480461 TS...	
333	7.640534	10.221.121.48	10.141.208.237	TCP	78	[TCP Retransmission] 64484 -> 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013481461 TS...	
415	9.643119	10.221.121.48	10.141.208.237	TCP	78	[TCP Retransmission] 64484 -> 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013483461 TS...	
649	13.656410	10.221.121.48	10.141.208.237	TCP	78	[TCP Retransmission] 64484 -> 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013487461 TS...	
10...	21.679545	10.221.121.48	10.141.208.237	TCP	78	[TCP Retransmission] 64484 -> 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013495461 TS...	

TCP发送方发送SYN后无法收到SYN+ACK报文，超时后重传SYN报文，仍然失败，直到最后放弃连接。

而UDP则直接发送，不与接收方建立连接：

udp.port==5001							
No.	Time	Source	Destination	Protocol	Length	Info	
68	1.786279	10.221.121.48	10.141.208.237	UDP	46	54655 -> 5001 Len=4	
69	1.787242	10.221.121.48	10.141.208.237	UDP	1042	54655 -> 5001 Len=1000	
70	1.787243	10.221.121.48	10.141.208.237	UDP	1042	54655 -> 5001 Len=1000	
71	1.787243	10.221.121.48	10.141.208.237	UDP	1042	54655 -> 5001 Len=1000	
72	1.787244	10.221.121.48	10.141.208.237	UDP	1042	54655 -> 5001 Len=1000	
73	1.787286	10.221.121.48	10.141.208.237	UDP	1042	54655 -> 5001 Len=1000	
74	1.787384	10.221.121.48	10.141.208.237	UDP	46	54655 -> 5001 Len=4	
119	2.805107	10.221.121.48	10.141.208.237	UDP	46	54655 -> 5001 Len=4	
120	2.806049	10.221.121.48	10.141.208.237	UDP	46	54655 -> 5001 Len=4	
121	2.806050	10.221.121.48	10.141.208.237	UDP	46	54655 -> 5001 Len=4	
122	2.806050	10.221.121.48	10.141.208.237	UDP	46	54655 -> 5001 Len=4	

1.3.1 在UDP分组中，观察UDP首部。长度字段是包括首部和数据还是只包括数据？为什么需要这个子选项？

在udp\_trans\_RT.pcapng包中使用 `udp.port == 5001` 过滤规则查看捕捉的包：

No.	Time	Source	Destination	Protocol	Length	Info
1...	2.587269	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
104	2.588203	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
105	2.588203	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
106	2.588204	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
107	2.588204	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
108	2.588205	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
109	2.588205	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
168	3.600717	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
169	3.601111	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
170	3.601356	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
171	3.601576	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4

由于发送时选择发送了5个缓冲区长度为1000字节的数据，所以这里选择包含数据的UDP报文查看。选择第一个Len=1000的报文：

```
► Internet Protocol Version 4, Src: 10.221.121.48, Dst: 10.141.208.237
▼ User Datagram Protocol, Src Port: 51169, Dst Port: 5001
  Source Port: 51169
  Destination Port: 5001
  Length: 1008
  Checksum: 0xd3c4 [unverified]
  [Checksum Status: Unverified]
  [Stream index: 43]
▼ Data (1000 bytes)
  Data: 5043415553412050434154544350205061747465726e2021...
  [Length: 1000]
```

UDP协议报文段的首部由8个字节，64bits组成。这64bits分成4个16bits：第一个16bits，记录发送方的端口号；第二个16bits，记录接收方的端口号；第三个16bits，记录UDP报文长度（Length）；第四个16bits，记录校验和。这里校验和用法与TCP不同，即使校验和不同表明传输过程中发生错误，也不会引起UDP重传；这会交给应用层程序决定如何处理。

这里长度字段Length为1008，很明显除数据外还包含了首部。

长度字段可以用于确认包含首部在内的数据的长度。除此之外，长度字段还参与校验和的计算：在进行检验和计算时，会添加一个伪首部一起进行运算。伪首部（占用12个字节）为：4个字节的源IP地址、4个字节的目的IP地址、1个字节的0、一个字节的数字17、以及占用2个字节UDP长度。接收端进行的校验和与UDP报文中的校验和相与，如果无差错应该全为1。如果有误，则将报文丢弃或者发给应用层，并附上差错警告。

### 1.3.2 分别计算TCP和UDP传输中额外开销是多少，比例是多少？

这里以抓到的有接收方的TCP和UDP流为例。

对于TCP协议，额外的开销是除了数据字节以外的开销，包括握手的信息、首部字节（不一定只有20字节）、应答的ACK信息、挥手信息以及重传的信息。这没有出现重传，不进行考虑。

TCP流列表框如下：

No.	Time	Source	Destination	Protocol	Length	Info
99	2.391167	10.221.121.48	10.141.208.237	TCP	78	64455 → 5001 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1013064444 TSecr=0 SACK_PERM=1
100	2.393950	10.141.208.237	10.221.121.48	TCP	74	5001 → 64455 [SYN, ACK] Seq=1 Ack=1 Win=8192 Len=0 MSS=1260 WS=56 SACK_PERM=1 TSval=433258270 TSecr=433258270
101	2.394015	10.221.121.48	10.141.208.237	TCP	66	64455 → 5001 [ACK] Seq=1 Ack=1 Win=132288 Len=0 TSval=1013064446 TSecr=433258270
102	2.394883	10.221.121.48	10.141.208.237	TCP	1066	64455 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=132288 Len=1000 TSval=1013064446 TSecr=433258270
103	2.395014	10.221.121.48	10.141.208.237	TCP	1314	64455 → 5001 [ACK] Seq=1001 Ack=1 Win=1248 Len=1000 TSval=1013064446 TSecr=433258270
104	2.395015	10.221.121.48	10.141.208.237	TCP	278	64455 → 5001 [PSH, ACK] Seq=2499 Ack=1 Win=212 TSval=1013064446 TSecr=433258270
105	2.395347	10.221.121.48	10.141.208.237	TCP	1314	64455 → 5001 [ACK] Seq=2461 Ack=1 Win=132288 Len=1248 TSval=1013064447 TSecr=433258270
106	2.397941	10.141.208.237	10.221.121.48	TCP	66	5001 → 64455 [ACK] Seq=1 Ack=2249 Win=66048 Len=0 TSval=433258271 TSecr=1013064446
107	2.397998	10.221.121.48	10.141.208.237	TCP	1314	64455 → 5001 [ACK] Seq=3709 Ack=1 Win=132288 Len=1248 TSval=1013064449 TSecr=433258271
108	2.397999	10.221.121.48	10.141.208.237	TCP	110	64455 → 5001 [FIN, PSH, ACK] Seq=4957 Ack=1 Win=132288 Len=44 TSval=1013064449 TSecr=433258271
109	2.398946	10.141.208.237	10.221.121.48	TCP	66	5001 → 64455 [ACK] Seq=1 Ack=3709 Win=66048 Len=0 TSval=433258271 TSecr=1013064446
110	2.401159	10.141.208.237	10.221.121.48	TCP	66	5001 → 64455 [ACK] Seq=1 Ack=5002 Win=66048 Len=0 TSval=433258271 TSecr=1013064449
117	2.403061	10.141.208.237	10.221.121.48	TCP	66	5001 → 64455 [FIN, ACK] Seq=1 Ack=5002 Win=66048 Len=0 TSval=433258271 TSecr=1013064449
118	2.403127	10.221.121.48	10.141.208.237	TCP	66	64455 → 5001 [ACK] Seq=5002 Ack=2 Win=132288 Len=0 TSval=1013064453 TSecr=433258271

由于要分开计算TCP的开销十分麻烦，因此考虑使用图中Length字段。Length是数据帧的长度，除去14字节以太网的头部和20字节的IP协议头部，剩下的都是TCP协议的部分（由于Option，TCP协议头部不止20字节）。

所以，TCP协议总共的数据长度为  $(78 + 74 + 66 + 1066 + 1314 + 278 + 1314 + 66 + 1314 + 110 + 66 + 66 + 66 + 66) - 14 \times (14 + 20) = 5468$ 。其中，有5000个字节的数据。所以，额外的开销是468个字节，额外开销比例为  $468 / 5468 \times 100\% = 8.56\%$ 。

对于UDP协议，额外的开销是首部字节。列表框如下：

No.	Time	Source	Destination	Protocol	Length	Info
1...	2.587269	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
104	2.588203	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
105	2.588203	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
106	2.588204	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
107	2.588204	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
108	2.588205	10.221.121.48	10.141.208.237	UDP	1042	51169 → 5001 Len=1000
109	2.588205	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
168	3.600717	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
169	3.601111	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
170	3.601356	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4
171	3.601576	10.221.121.48	10.141.208.237	UDP	46	51169 → 5001 Len=4

这里可以看见在真正传输数据的UDP连接前后有几个发送了相同的4个字节的数据的UDP连接。并不清楚这是做什么用的，猜测与网络连接有关。

UDP协议总共的数据长度为  $(46 + 1042 + 1042 + 1042 + 1042 + 1042 + 46 + 46 + 46 + 46 + 46) - 11 \times (14 + 20) = 5112$ 。其中，有5000个字节数据。所以，额外的开销是112个字节，额外开销比例为  $112 / 5112 \times 100\% = 2.19\%$ 。

可以看出，UDP的额外开销比例比TCP小很多。

### 1.3.3 比较TCP和UDP的优缺点。

TCP协议提供有序、可靠的数据传输，并且面向连接，利用三次握手建立连接，利用四次挥手断开连接。TCP通过重传提供可靠的数据传输，并且提供了流量控制、拥塞控制等相应的机制来减少网络拥堵。系统提供的API应用上，针对不同的连接分配独立缓冲区，进一步减少可能的丢包问题（由于缓冲区满造成数据无法接受引起数据包丢失问题）。然而，TCP的实现非常复杂，耗时长，并且流量控制、拥塞控制等机制限制了传输速度。此外，TCP的额外开销也比UDP高（1.3.2）。

UDP 协议面向无连接通信，没有三次握手的需求，网络设备在通信时，UDP 数据拥有比 TCP 数据更高的优先通信权利，这点使得 UDP 数据在通信上更加的快速，但是当网络拥堵时，通信设备同样会优先丢弃 UDP 数据，这种设计方式会导致 UDP 数据出现丢失的问题。同时，协议没有规定相应的流控制方式，不会保证数据的顺序到达。在没有流控制的情况下，UDP 数据可能出现丢失、乱序到达的问题，同样由于没有流控制，如果开发控制不当，可能导致数据风暴问题。在系统驱动实现上，由于所有的 UDP 数据接收时，使用共享的数据缓冲区，设置不当大小的缓冲区，可能会增加数据包的丢失问题。

## 1.4 讨论与研究

### 1.4.1 在 SYN, SYN+ACK 和 ACK 分组中有没有数据被发送？可不可以把这些分组中发送数据？

从目前抓到的分组来看，三次握手中 SYN, SYN+ACK, ACK 报文中均不包含数据。一般发送方第四个分组的 Seq 和 Ack 与第三次握手的 ACK 分组相同，只不过其中包含了数据。

可以。根据 RFC 793:

September 1981

Transmission Control Protocol  
Functional Specification

SEGMENT ARRIVES

are acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to SEG.SEQ. SND.UNA should be advanced to equal SEG.ACK (if there is an ACK), and any segments on the retransmission queue which are thereby acknowledged should be removed.

If SND.UNA > ISS (our SYN has been ACKed), change the connection state to ESTABLISHED, form an ACK segment

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

and send it. Data or controls which were queued for transmission may be included. If there are other controls or text in the segment then continue processing at the sixth step below where the URG bit is checked, otherwise return.

Otherwise enter SYN-RECEIVED, form a SYN,ACK segment

<SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

and send it. If there are other controls or text in the segment, queue them for processing after the ESTABLISHED state has been reached, return.

fifth, if neither of the SYN or RST bits is set then drop the segment and return.

[Page 68]

数据可以放在第三次握手的 ACK 分组里的。至于 SYN 和 SYN+ACK 报文，应该不可以。

### 1.4.2 在关闭连接时，LEN=0，为什么 ACK 比 SEQ 大 1？

如图：

66 5001 → 54374 [FIN, ACK] Seq=1 Ack=8194 Win=66048 Len=0 T
66 54374 → 5001 [ACK] Seq=8194 Ack=2 Win=132288 Len=0 TSval

关闭连接时，接收方发送的第二个 FIN 报文，Seq 为 1, Len=0, 下一个 Ack 为 2，比 Seq 大 1。这是要与三次握手的 ACK 区别。因为如果 ACK 为 1 的话，接收方连接本来在发送 FIN 后断开，高并发状态下可能会重新建立新的连接。此时，若发送方最后的 Ack 为 1，会与三次握手中的 ACK 混淆，产生错误。

## 2. TCP流和UDP流的竞争

实验步骤：

查看两个TCP流、TCP流与UDP流、UDP与UDP流的竞争。使用端口号为5001和5002。主机IP地址为10.221.85.96（与之前不同），服务器IP地址为10.141.208.237。以主机为发送方，服务器为接收方。

在TCP流的竞争中，由于使用原始命令 `pcattcp -t -l 1000 -p 5001 10.141.208.237` 传输时间不到一秒，来不及执行第二次传输：

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -l 1000 -p 5001 10.141.208.237
PCUSA Test TCP Utility V2.01.01.08
TCP Transmit Test
Transmit : TCP -> 10.141.208.237:5001
Buffer Size : 1000; Alignment: 16384/0
TCP_NODELAY : DISABLED (0)
Connect : Connected to 10.141.208.237:5001
Send Mode : Send Pattern; Number of Buffers: 2048
Statistics : TCP -> 10.141.208.237:5001
2048000 bytes in 0.47 real seconds = 4264.39 KB/sec +++
numCalls: 2048; msec/call: 0.23; calls/sec: 4366.74
PS C:\Users\wanga\Desktop>
```

所以将传输的缓冲区数量从默认的2048变为32768，增加执行时间，便于查看。

在UDP流的竞争中，将注入网络的缓冲区的个数设为4096。

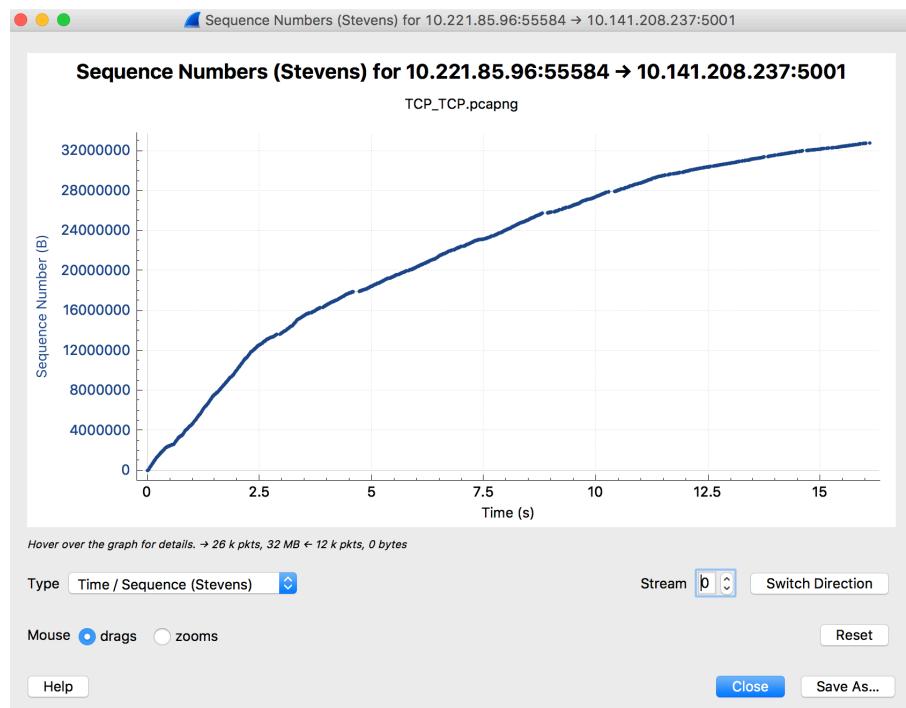
使用抓包过滤器 `src port 5001 or src port 5002 or dst port 5001 or dst port 5002`。

实验分析：

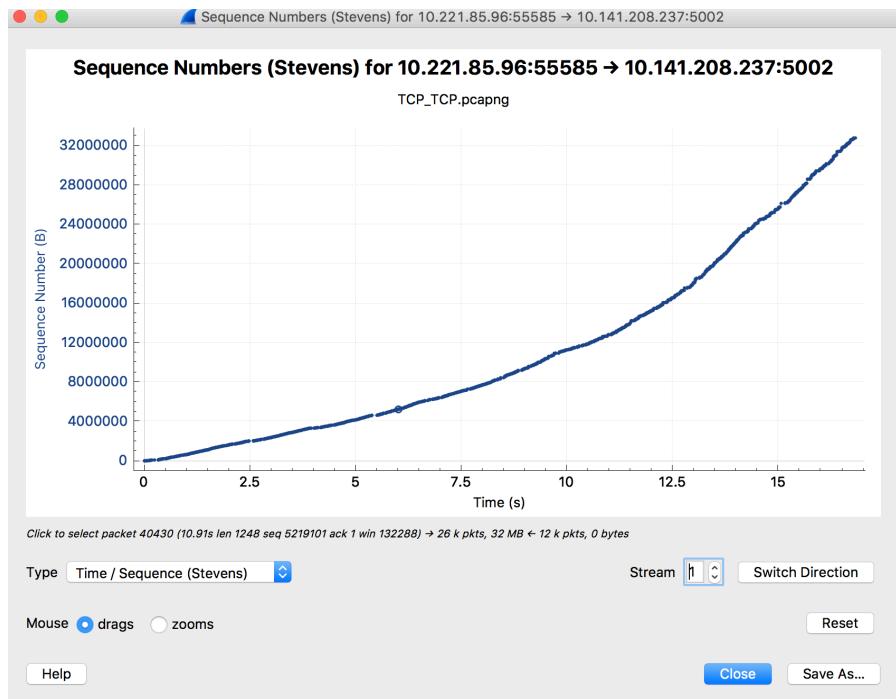
**2.1 使用 TCP流图形 来查看TCP\_TCP.cap文件中两个流的情况，截图。说明产生这样的原因。也可通过流量图来说明情况。**

流图形结果如下：

5001端口：

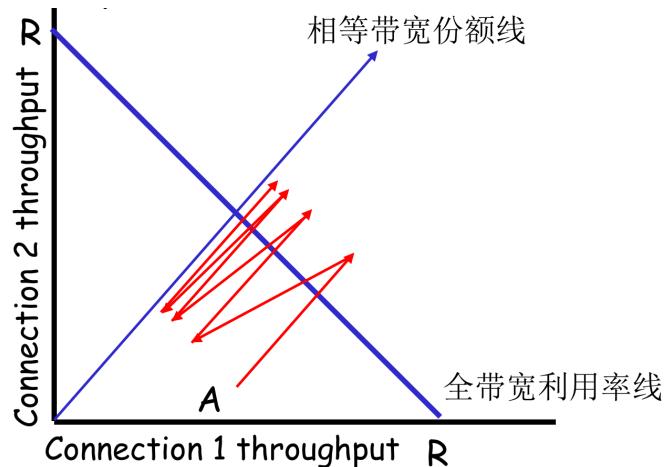


5002端口：



出现这样情况的原因是因为TCP的拥塞控制是公平的。拥塞窗口是加性增、乘性减的。首先，拥塞窗口是随着每个确认而呈指数增长的。这个指数增加增长阶段被称为慢启动。慢启动不一定特别慢，但是它比立即发送整个接收通告窗口慢一些。即使没有包丢失，这种乘法增长方式也不会一直持续下去。达到一个自适应确定的阈值上，TCP切换到一个拥塞窗口的线性加法增长。这个阶段被称为拥塞避免。当感受到网络拥塞时，阈值就降低为拥塞窗口的一半，拥塞窗口降为初始值。

两个拥塞的TCP连接，无论这两个连接在二维空间的何处，他们都会汇聚到这样的状态：两个连接实现的带宽将沿着相同带宽份额的线波动：

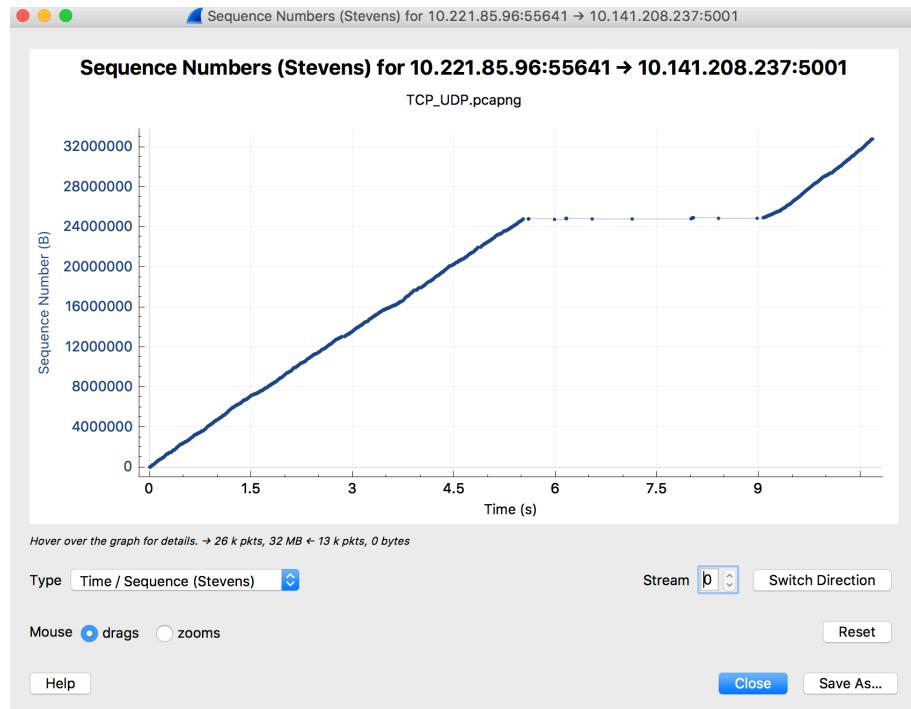


所以我们可以看见，先启动的TCP连接序号增长比较快，然而第二个连接启动后，网络拥塞，第一个TCP连接的序号增长速度就开始下降了，而第二个TCP连接的序号增长速度不断上升，最终二者达到一个平衡的状态。

态，平分带宽。这是TCP的拥塞控制机制实现的，保证了公平性。

## 2.2 使用TCP流图形 来查看TCP\_UDP.cap文件中TCP流的情况

如图：



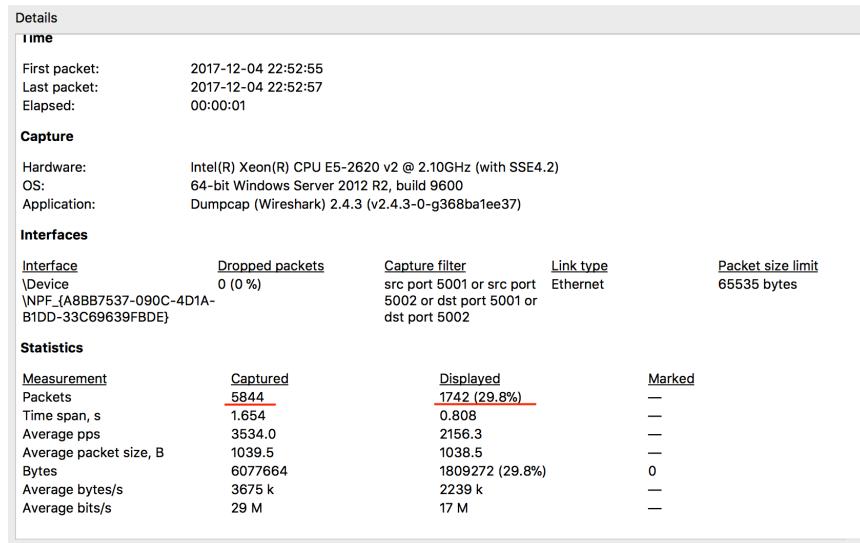
TCP流中间水平直线的原因是TCP与UDP的竞争造成的。UDP没有流量和拥塞控制。TCP察觉到网络拥塞时会触发拥塞控制，降低对带宽的占用，UDP就会增加对带宽的占用；反复进行，直至TCP不占据任何带宽，完全被UDP占据，直到UDP文件传输完毕。

## 2.3 在UDP\_UDP.cap文件中，我们使用 统计-> 捕获文件属性 分别来查看UDP两流的接受情况，计算丢包率。

5001端口处接收到的包：

Details				
Time				
First packet:	2017-12-04 22:52:55			
Last packet:	2017-12-04 22:52:57			
Elapsed:	00:00:01			
Capture				
Hardware:	Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz (with SSE4.2)			
OS:	64-bit Windows Server 2012 R2, build 9600			
Application:	Dumpcap (Wireshark) 2.4.3 (v2.4.3-0-g368ba1ee37)			
Interfaces				
Interface	Dropped packets	Capture filter	Link type	Packet size limit
\Device\NPF_{A8BB7537-090C-4D1A-B1DD-33C69639FBD8}	0 (0 %)	src port 5001 or src port 5002 or dst port 5001 or dst port 5002	Ethernet	65535 bytes
Statistics				
Measurement	Captured	Displayed	Marked	
Packets	5844	4102 (70.2%)	—	
Time span, s	1.654	1.403	—	
Average pps	3534.0	2923.4	—	
Average packet size, B	1039.5	1040.5	—	
Bytes	6077664	4268392 (70.2%)	0	
Average bytes/s	3675 k	3041 k	—	
Average bits/s	29 M	24 M	—	

5002端口处接收到的包：



设定的发送缓冲区的数目是4096，理想情况下每个端口应收到4096个包，可见UDP存在竞争的情况下丢包情况十分严重。

5001端口的丢包率：

一共收到 4102 个包，排除 6 个长度为 4 的非数据包，共收到 4096 个包。所以丢包率为  $(4096 - 4096) / 4096 \times 100\% = 0$

5002 端口的丢包率：

一共收到 1742 个包，排除 6 个长度为 4 的非数据包，共收到 1736 个包。所以丢包率为  $(4096 - 1736) / 4096 \times 100\% = 57.62\%$

很明显，这体现了 UDP 的工作模式：当 UDP 发生竞争时，尽可能保证其中一个 UDP 不丢包。

## 2.4 给出上述三组实验的发送方和接收方的cmd窗口截图。

TCP\_TCP 5001端口发送方：

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -l 1000 -p 5001 -n 32768 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
TCP Transmit Test
    Transmit : TCP -> 10.141.208.237:5001
    Buffer Size : 1000; Alignment: 16384/0
    TCP_NODELAY : DISABLED (0)
    Connect : Connected to 10.141.208.237:5001
    Send Mode : Send Pattern; Number of Buffers: 32768
    Statistics : TCP -> 10.141.208.237:5001
32768000 bytes in 15.73 real seconds = 2033.81 KB/sec +++
numCalls: 32768; msec/call: 0.49; calls/sec: 2082.62
PS C:\Users\wanga\Desktop>
```

TCP\_TCP 5002端口发送方：

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -l 1000 -p 5002 -n 32768 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
TCP Transmit Test
    Transmit : TCP -> 10.141.208.237:5002
    Buffer Size : 1000; Alignment: 16384/0
    TCP_NODELAY : DISABLED (0)
    Connect : Connected to 10.141.208.237:5002
    Send Mode : Send Pattern; Number of Buffers: 32768
    Statistics : TCP -> 10.141.208.237:5002
32768000 bytes in 16.69 real seconds = 1917.66 KB/sec +++
numCalls: 32768; msec/call: 0.52; calls/sec: 1963.68
PS C:\Users\wanga\Desktop>
```

TCP\_TCP 5001端口接收方:

```
PS D:\wangao> .\PCATTCP.exe -r -l 1000 -p 5001
PCAUSA Test TCP Utility v2.01.01.08
TCP Receive Test
Local Host : Gandhi
*****
Listening...: On port 5001

Accept      : TCP <- 10.221.85.96:55584
Buffer Size : 1000; Alignment: 16384/0
Receive Mode: Sinking (discarding) Data
Statistics   : TCP <- 10.221.85.96:55584
32768000 bytes in 16.03 real seconds = 1996.13 KB/sec +++
numCalls: 51384; msec/call: 0.32; calls/sec: 3205.29
PS D:\wangao>
```

TCP\_TCP 5002端口接收方:

```
PS D:\wangao> .\PCATTCP.exe -r -l 1000 -p 5002
PCAUSA Test TCP Utility v2.01.01.08
TCP Receive Test
Local Host : Gandhi
*****
Listening...: On port 5002

Accept      : TCP <- 10.221.85.96:55585
Buffer Size : 1000; Alignment: 16384/0
Receive Mode: Sinking (discarding) Data
Statistics   : TCP <- 10.221.85.96:55585
32768000 bytes in 16.78 real seconds = 1906.92 KB/sec +++
numCalls: 52059; msec/call: 0.33; calls/sec: 3102.26
PS D:\wangao>
```

TCP\_UDP 5001端口发送方:

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -l 1000 -p 5001 -n 32768 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
TCP Transmit Test
Transmit     : TCP -> 10.141.208.237:5001
Buffer Size : 1000; Alignment: 16384/0
TCP_NODELAY : DISABLED (0)
Connect      : Connected to 10.141.208.237:5001
Send Mode   : Send Pattern; Number of Buffers: 32768
Statistics   : TCP -> 10.141.208.237:5001
32768000 bytes in 10.48 real seconds = 3051.98 KB/sec +++
numCalls: 32768; msec/call: 0.33; calls/sec: 3125.23
PS C:\Users\wanga\Desktop>
```

TCP\_UDP 5002端口发送方:

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -u -l 1000 -p 5002 -n 65536 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
UDP Transmit Test
Transmit     : UDP -> 10.141.208.237:5002
Buffer Size : 1000; Alignment: 16384/0
Send Mode   : Send Pattern; Number of Buffers: 65536
Statistics   : UDP -> 10.141.208.237:5002
65536000 bytes in 2.91 real seconds = 22023.40 KB/sec +++
numCalls: 65538; msec/call: 0.05; calls/sec: 22552.65
PS C:\Users\wanga\Desktop>
```

TCP\_UDP 5001端口接收方:

```
PS D:\wangao> .\PCATTCP.exe -r -l 1000 -p 5001
PCAUSA Test TCP Utility v2.01.01.08
TCP Receive Test
Local Host : Gandhi
*****
Listening...: On port 5001

Accept      : TCP <- 10.221.85.96:55641
Buffer Size : 1000; Alignment: 16384/0
Receive Mode: Sinking (discarding) Data
Statistics   : TCP <- 10.221.85.96:55641
32768000 bytes in 10.67 real seconds = 2998.50 KB/sec +++
numCalls: 51099; msec/call: 0.21; calls/sec: 4788.14
PS D:\wangao>
```

TCP\_UDP 5002端口接收方:

```
PS D:\wangao> .\PCATTCP.exe -r -u -l 1000 -p 5002
PCAUSA Test TCP Utility V2.01.01.08
UDP Receive Test
Protocol : UDP
Port      : 5002
Buffer Size : 1000; Alignment: 16384/0
recvfrom   : UDP <- 10.221.85.96:62864
Statistics : UDP <- 10.221.85.96:62864
22315000 bytes in 3.50 real seconds = 6226.28 KB/sec +++
numCalls: 22316; msec/call: 0.16; calls/sec: 6376.00
PS D:\wangao>
```

UDP\_UDP 5001端口发送方:

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -u -l 1000 -p 5001 -n 4096 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
UDP Transmit Test
Transmit   : UDP -> 10.141.208.237:5001
Buffer Size : 1000; Alignment: 16384/0
Send Mode   : Send Pattern; Number of Buffers: 4096
Statistics   : UDP -> 10.141.208.237:5001
4096000 bytes in 0.20 real seconds = 19704.43 KB/sec +++
numCalls: 4098; msec/call: 0.05; calls/sec: 20187.19
PS C:\Users\wanga\Desktop>
```

UDP\_UDP 5002端口发送方:

```
PS C:\Users\wanga\Desktop> .\PCATTCP.exe -t -u -l 1000 -p 5002 -n 4096 10.141.208.237
PCAUSA Test TCP Utility V2.01.01.08
UDP Transmit Test
Transmit   : UDP -> 10.141.208.237:5002
Buffer Size : 1000; Alignment: 16384/0
Send Mode   : Send Pattern; Number of Buffers: 4096
Statistics   : UDP -> 10.141.208.237:5002
4096000 bytes in 0.20 real seconds = 19704.43 KB/sec +++
numCalls: 4098; msec/call: 0.05; calls/sec: 20187.19
PS C:\Users\wanga\Desktop>
```

UDP\_UDP 5001端口接收方:

```
PS D:\wangao> .\PCATTCP.exe -r -u -l 1000 -p 5001
PCAUSA Test TCP Utility V2.01.01.08
UDP Receive Test
Protocol : UDP
Port      : 5001
Buffer Size : 1000; Alignment: 16384/0
recvfrom   : UDP <- 10.221.85.96:50284
Statistics : UDP <- 10.221.85.96:50284
4096000 bytes in 1.20 real seconds = 3325.02 KB/sec +++
numCalls: 4097; msec/call: 0.30; calls/sec: 3405.65
PS D:\wangao>
```

UDP\_UDP 5002端口接收方:

```
PS D:\wangao> .\PCATTCP.exe -r -u -l 1000 -p 5002
PCAUSA Test TCP Utility V2.01.01.08
UDP Receive Test
Protocol : UDP
Port      : 5002
Buffer Size : 1000; Alignment: 16384/0
recvfrom   : UDP <- 10.221.85.96:54513
Statistics : UDP <- 10.221.85.96:54513
1736000 bytes in 0.55 real seconds = 3099.29 KB/sec +++
numCalls: 1737; msec/call: 0.32; calls/sec: 3175.50
PS D:\wangao>
```

2.5 讨论与研究：“TCP友好”的UDP应用程序能对其自身实施拥塞控制算法。探讨术语“TCP友好”的定义。

由于UDP协议是“尽力而为”的服务，会占据所有的带宽，导致带宽分配的严重不公；而“TCP友好”的UDP应用程序能对其自身实施拥塞控制算法。“TCP友好”，指的是使用基于“尽力而为”协议的数据流

必须与同等条件下TCP流的吞吐量相似，保证带宽分配公平，不会出现UDP与TCP竞争一样出现完全占据带宽的情况，其自身能实现拥塞控制。

**3. 参考资料：**

1. 《计算机网络：自顶向下方法》第六版
2. 课堂PPT
3. CSDN技术博客