**实验三、传输层TCP协议分析实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 组号： |  |  | 向胤兴 | 1-6 |  |
| 姓名： | 徐远帆 | 学号： | 2216113400 | 班级： | 计算机2102 |
| 姓名： | 向胤兴 | 学号： | 2215012469 | 班级： | 计算机2102 |
|  |  |  |  |  |  |

1. 实验目的
2. 理解TCP报文首部格式和字段的作用，TCP连接的建立和释放过程，TCP数据传输中的编号与确认的过程。
3. 理解TCP的错误恢复的工作原理和字节流的传输模式，分析错误恢复机制中TCP双方的交互情况。
4. 理解TCP的流量控制的工作原理，分析流量控制中TCP双方的交互情况。
5. 理解TCP的拥塞控制的工作原理，分析拥塞控制中TCP双方的交互情况。
6. 实验内容
7. 使用基于TCP的应用程序（如浏览器下载文件）传输文件，在**客户端和服务器均要捕获TCP报文**。
8. 分析TCP报文首部信息、TCP连接的建立和释放过程、TCP数据的编号与确认机制。观察几个典型的TCP选项：MSS、SACK、Window Scale、Timestamp等，查资料说明其用途。
9. 观察和估计客户机到服务器的RTT，双方各自的MSS。
10. 观察TCP的流量控制过程，和拥塞控制中的慢启动、快速重传、拥塞避免，快速恢复等过程【观察拥塞控制的难度较大，观察到前两个过程即可】。

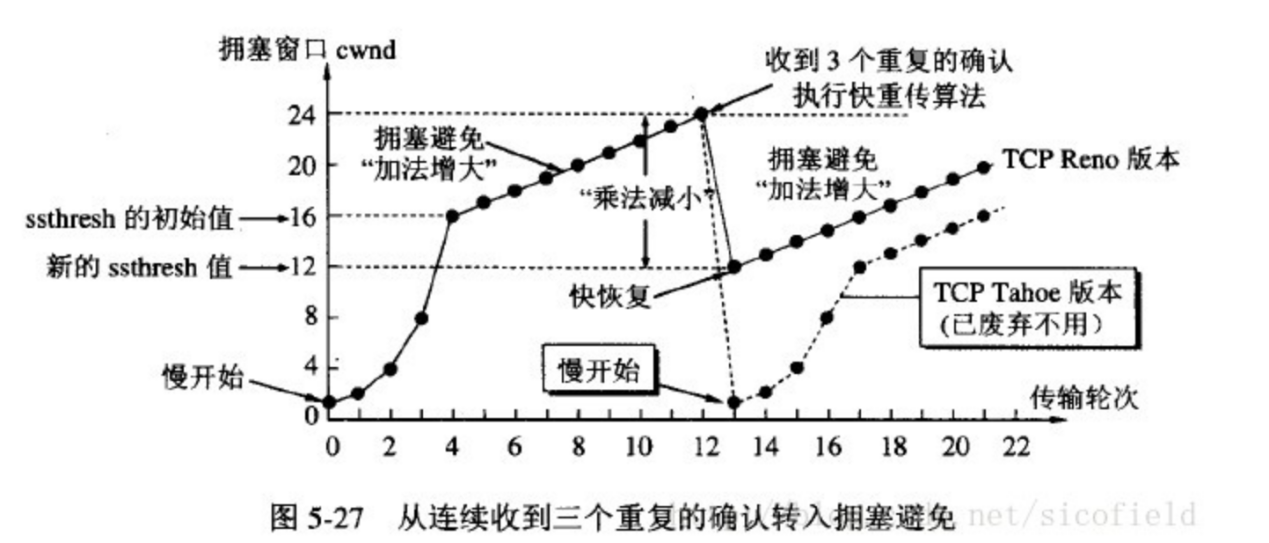


图4-0 典型的TCP 拥塞控制过程图例

1. \*（可选）注意观察初始的cwnd是多少，看看不同的操作系统初始cwnd的差别。

【可以增加题目规定以外的分析】

1. 实验环境与分组

1）云服务器一台，启动Apache2服务（或其他服务器程序）。

2）每2名同学一组，各自在电脑上运行客户端程序（浏览器或其他客户端程序）。

3）使用客户端程序下载数据，运行Wireshark软件捕获报文。【注意：可以关闭Wireshark的HTTP协议分析，专注在TCP协议上，关闭方法是：菜单‘分析’—>‘启用的协议’中，取消‘HTTP’的选择。】

1. 实验组网

下图是本实验的组网图，图中参数请根据实际情况标注。

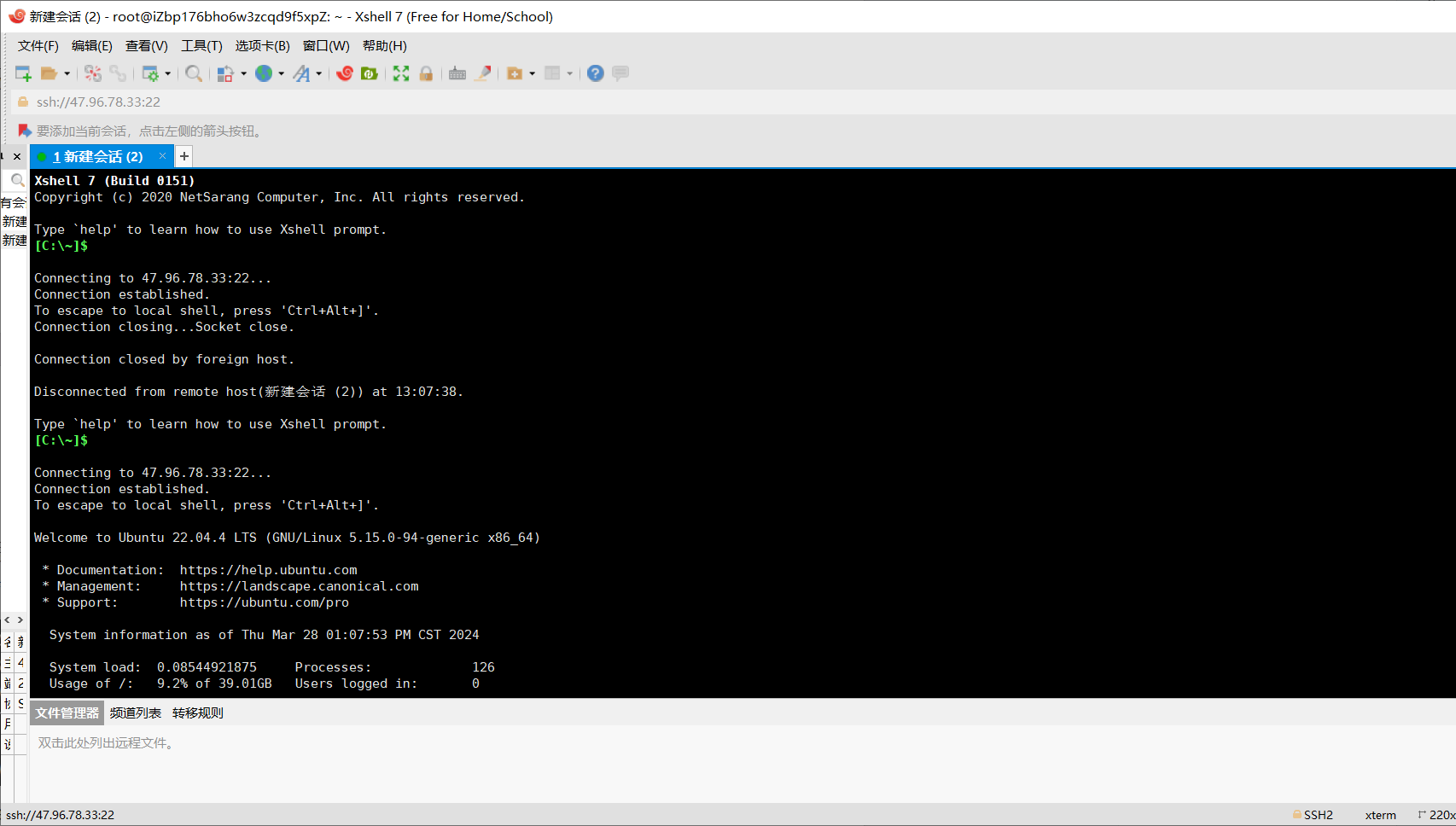


1. 实验过程及结果分析

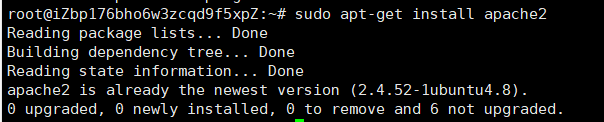
【过程记录应当详尽，截图并加以说明。以下过程和表格仅供参考。】

步骤1：PC2登录到服务器Z上，在云服务器Z上启动web服务器（Apache、IIS等）。

首先通过Xshell连接到云服务器：



本实验使用aliyun配置ubuntu系统，使用Apache2应用程序。

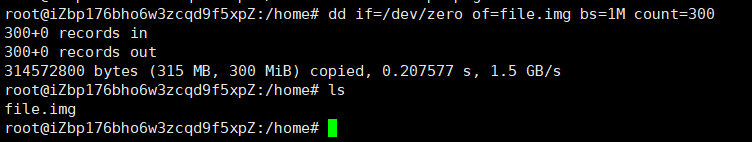


使用sudo apt-get install apache2在云服务器A上安装apache2。

可以看到安装成功（第一次安装忘记截屏了）。

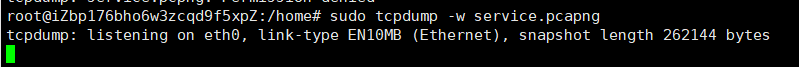
步骤2：在Z上首先创建一个文件，然后在PC1和Z上启动报文捕获软件，开始截获报文（注意抓包一定要包括建立连接的报文）。

首先需要创建一个文件，在shell中执行/home# dd if=/dev/zero of=file.img bs=1M count=500命令，创建一个大小为500M的，格式为.img的文件。创建完成后使用ls命令可以看到当前目录下已经存在file.img项。



服务器端开始抓包：

使用sudo tcpdump -w service.pcapng命令开始抓包并保存为service.pcapng



PC1端打开wireshark抓包即可。

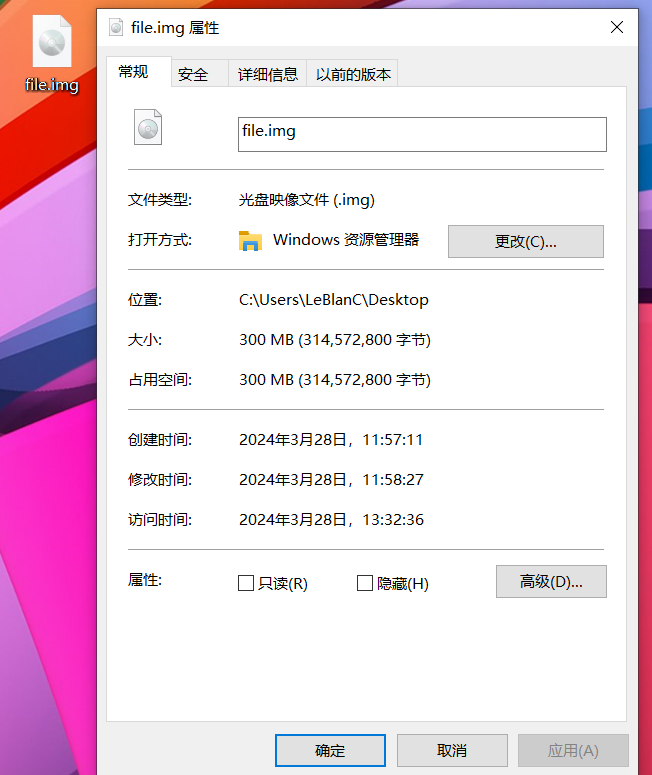
步骤3：在PC1上运行客户端软件，发送和接收一个500MB的file.img。文件传输完成后，停止报文截获。

在PC1使用Powershell下载服务器文件：

Invoke-WebRequest -Uri ‘http://139.9.129.107/file.img’ -OutFile ‘C:\Users\Mercy\Downloads\file.img’



文件下载完成



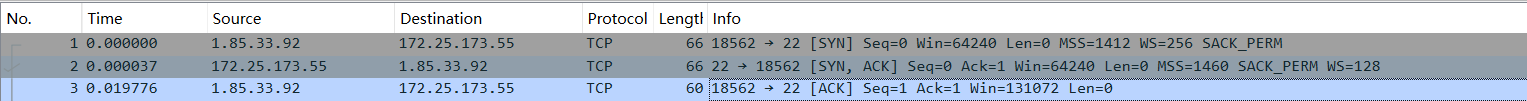
下载完成后云服务器和PC1关闭抓包，在shell中执行ls查看目录：



可以看到服务器端的抓包成功保存，同样从服务器端下载service.pcapng到PC1进行分析。

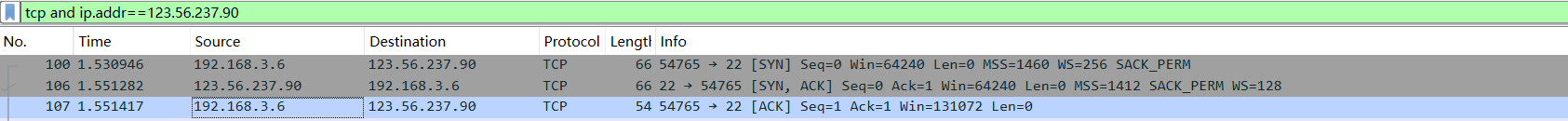
步骤4：对比观察客户端和服务器截获的报文，分析TCP协议的选项：MSS、SACK、Window Scale、Timestamp等，查资料说明其用途。

服务器截获的TCP建立连接的报文：



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 报文序号 | 序号字段Seq Number | ACK flag | 确认号Ack Number | SYN flag | 源IP | 目的IP |
| 1 | 0 | 0 | 0 | 1 | 1.85.33.92 | 172.25.173.55 |
| 2 | 0 | 1 | 1 | 1 | 172.25.173.55 | 1.85.33.92 |
| 3 | 1 | 1 | 1 | 0 | 1.85.33.92 | 172.25.173.55 |

客户端截获的TCP建立连接的报文：



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 报文序号 | 序号字段Seq Number | ACK flag | 确认号Ack Number | SYN flag | 源IP | 目的IP |
| 100 | 0 | 0 | 0 | 1 | 192.168.3.6 | 123.56.237.90 |
| 106 | 0 | 1 | 1 | 1 | 123.56.237.90 | 192.168.3.6 |
| 107 | 1 | 1 | 1 | 0 | 192.168.3.6 | 123.56.237.90 |

可以看到不管是客户端还是服务器端截获的TCP建立连接报文都是经由三次握手：

第一次握手：由客户端发起第一次请求，SYN标志为1，序列号为0；

第二次握手：有服务器向客户端应答，SYN、ACK标志为1，序列号为0，ACK标志置1；

第三次握手：客户端应答服务器，ACK置为1，序列号为1，应答第二次握手中的ACK=1。

可以看到服务器端和客户端截获到的报文基本相同，区别只在于源IP地址和目的IP地址的区别。IP地址区别的原因是一台设备在访问另一网段的设备时，目标地址只能是目标设备的公网地址，但源地址是自身的私有地址。

TCP连接的释放：

本次抓包未能成功抓到四次握手FIN标志，最终是通过RST来强制关闭连接：

RST包是一个特殊的TCP包，用于异常关闭一个现有的连接。当一个TCP端点（客户端或服务器）检测到某些严重的问题，无法或不愿继续与另一个端点进行通信时，它可能会发送一个RST包来强制关闭连接。

服务器端：



客户端：



工作过程：  
 客户端向服务器端发送一个RST置位为1的包，来断开TCP连接。服务器接收到此包后，不再回应，断开连接。

RST包与正常的TCP连接关闭过程（即使用FIN和ACK包）不同。正常的关闭过程是一个有序的四次握手过程，而RST包则是一个突发的、强制性的关闭方法。

这些问题可能包括：

无效的连接：例如，当一个数据包到达一个未被监听的端口时。

连接超时：如果连接在一段时间内没有任何活动，某些系统或应用可能会决定关闭它。

应用层错误：某些应用层协议可能会在检测到错误时发送RST包。

系统资源不足：当系统资源（如内存）不足时，可能会发送RST包来关闭一些连接。

**为了抓获到FIN的报文，重新做了一次实验抓包**，此次在服务器端和客户端都抓到了断开连接的FIN报文：此次PC1，服务器地址如下：

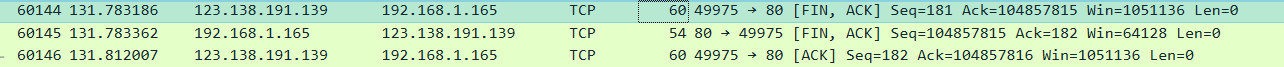
PC1公网IP：123.138.191.139

PC1私网IP：10.169.53.194

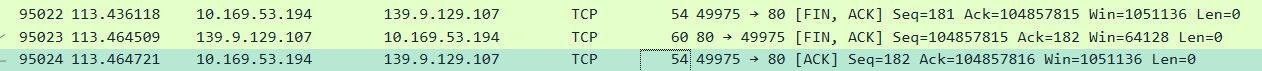
服务器公网IP：139.9.129.107

服务器私网IP：192.168.1.165

服务器抓到的报文：



客户端抓到的报文：



客户端截获的报文列表如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 第一个报文 | 第二个报文 | 第三个报文 | 第四个报文 |
| 序号 | 95022 | 95023 | 95024 | 无 |
| 发送方 | 10.169.53.19 | 139.9.129.107 | 10.169.53.19 |  |
| 接收方 | 139.9.129.107 | 10.169.53.19 | 139.9.129.107 |  |
| Seq# | 181 | 104857815 | 182 |  |
| ACK# | 104857815 | 182 | 104857816 |  |
| ACK Flag | 1 | 1 | 1 |  |
| FIN Flag | 1 | 1 | 0 |  |

分析：一共只抓获了三条报文，原因分析如下：

第一条报文，Seq=181，客户端发送给服务器，FIN标志置1，第一次握手。

第二条报文，Seq=104857815，服务器发送给客户端，ACK=182，表示收到第一次握手的Seq=181报文，起到第二次握手的作用；此外FIN标志置1，起到第三次握手的作用。

第三条报文，ACK=104857816，客户端应答服务器，表示收到第三次握手的Seq=104857815的报文，第四次握手成功。TCP连接成功断开。

关于报文中MSS、SACK、Window Scale、Timestamp字段的分析：

以服务器端第一个报文为例：



MSS (Maximum Segment Size) = 1412：

MSS表示TCP连接中每个报文段所能承载的最大数据长度（不包括TCP头部和IP头部）。这里，MSS的值为1412字节。这意味着在TCP连接建立时，收发双方协商确定，每个报文段的数据部分最大不能超过1412字节。这有助于避免IP层分片，从而提高网络性能。

SACK\_PERM：

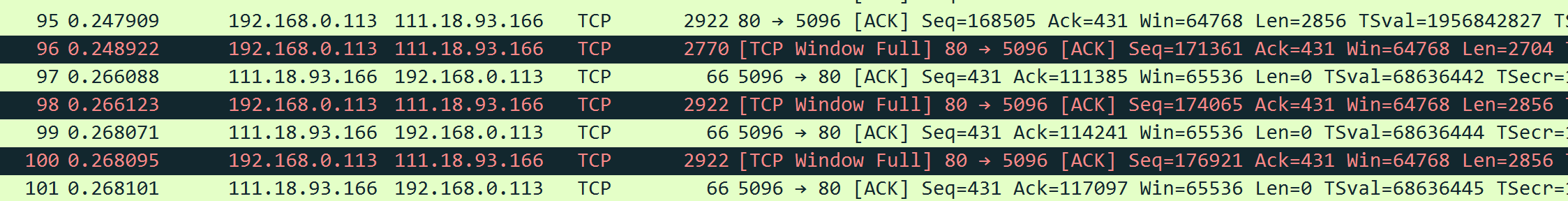
SACK\_PERM标志表示TCP连接支持Selective Acknowledgment（选择性确认）功能。这意味着接收方能够选择性地确认接收到的数据段，而不是仅仅确认最后一个连续收到的数据段。这种机制有助于发送方更精确地确定哪些数据段丢失了，并只重传那些丢失的数据段，从而提高数据传输的效率。

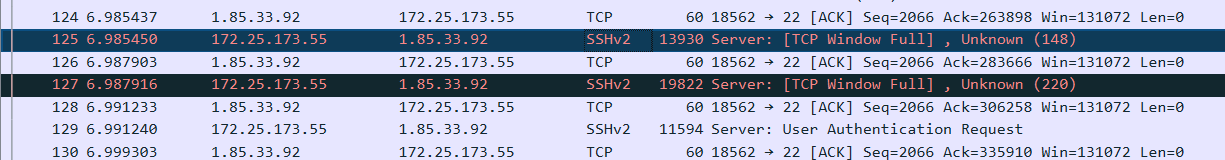
Window Scale = 256：

Window Scale用于放大TCP窗口大小。原始的TCP窗口大小是一个16位的值，最大为65535字节。但在高速网络中，这个大小可能不足以充分利用网络带宽。Window Scale通过一个缩放因子来扩展窗口大小，这里缩放因子为256。这意味着实际的窗口大小是原始窗口大小乘以256。这有助于在高带宽网络中提高TCP的吞吐量。

报文中没有Timestamp，Timestamp（时间戳）在TCP协议中是一个重要的组成部分，用于记录发送和接收数据的时间信息。具体来说，Timestamp的内容包括发送方的时间戳（TSval）和接收方的时间戳回显（TSecr）。

步骤5：分析TCP数据传送阶段的报文，分析其错误恢复和流量控制机制，并填表。【注：出现明显的流量控制的地方，Wireshark会有应答窗口的变化，乃至[TCP Window Full]或[TCP Zero Window]标记的报文出现。如果没有观察到明显的流量控制过程，可以再单独设计实验测试。比如编程设计接收端缓慢接收数据。】





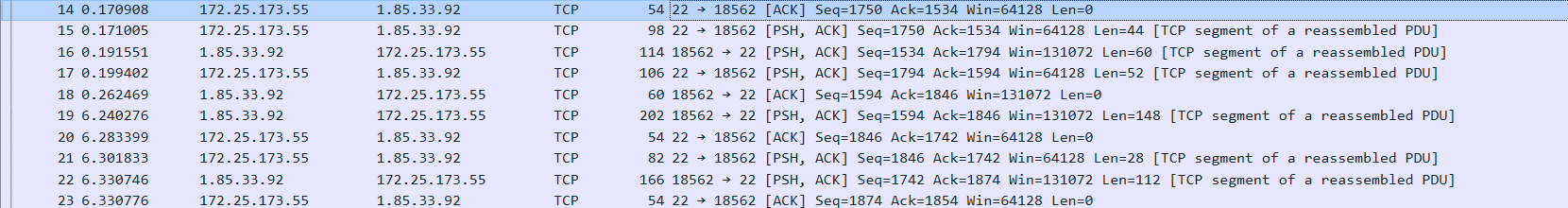


表3-1 正常通信记录TCP数据传送阶段的报文

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 报文序号 | 报文种类 (数据/确认) | 序号字段Seq Number | 确认号Ack Number | 数据长度 | 确认到哪条报文（填序号） | 窗口大小 |
| 14 | 确认 | 1750 | 1534 | 0 | 10 | 64128 |
| 15 | 数据 | 1750 | 1534 | 44 | 10 | 64128 |
| 16 | 数据 | 1534 | 1794 | 60 | 15 | 131072 |
| 17 | 数据 | 1794 | 1594 | 52 | 16 | 64128 |
| 18 | 确认 | 1594 | 1846 | 0 | 17 | 131072 |
| 19 | 数据 | 1594 | 1846 | 148 | 17 | 131072 |
| 20 | 确认 | 1846 | 1742 | 0 | 19 | 64128 |
| 21 | 数据 | 1846 | 1742 | 28 | 19 | 64128 |
| 22 | 数据 | 1742 | 1874 | 112 | 21 | 131072 |
| 23 | 数据 | 1874 | 1854 | 0 | 22 | 64128 |

服务器端在向客户端发送数据时，会提前向客户端发送一个数据长度为0的确认报文，确认可以传下一个报文。

客户端在应答客户端时，原本不应该有数据报文。但由于远程服务器由ssh连接，客户端向服务器端在应答报文中捎带了数据，在计算机网络中把这一现象称为捎带应答。

可以看到序号字段Seq与确认号字段Ack之间的对应关系。

TCP窗口满时TCP数据传送的报文：

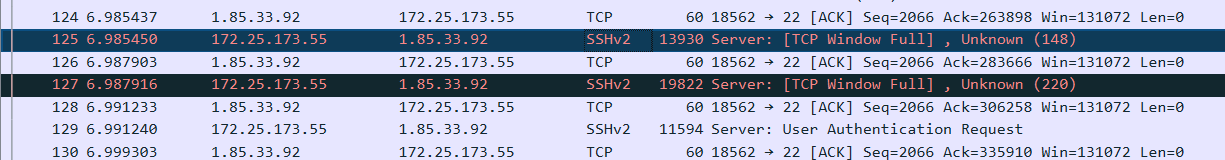


表3-2 TCP窗口满记录TCP数据传送阶段的报文

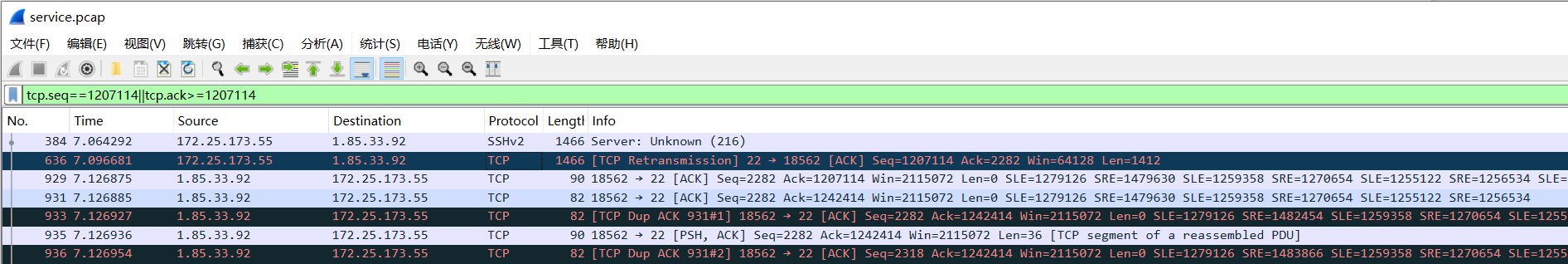
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 报文序号 | 报文种类 (数据/确认) | 序号字段Seq Number | 确认号Ack Number | 数据长度 | 窗口大小 |
| 124 | 确认 | 2066 | 263898 | 0 | 131072 |
| 125 | 确认 | 381094 | 2066 | 0 | 64128 |
| 126 | 确认 | 2066 | 183666 | 0 | 131072 |
| 127 | 确认 | 394970 | 2066 | 0 | 64128 |

错误恢复机制：

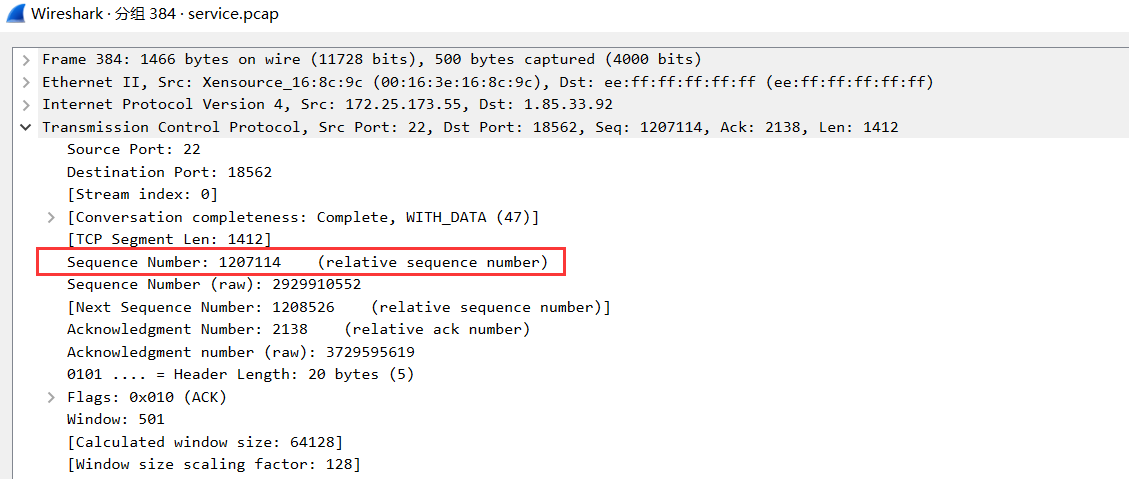
超时重传机制：

当TCP发送端发送一个数据报文段后，它会启动一个定时器，这个定时器被设置为一个初始的超时时间值。如果在定时器超时之前，发送端收到了接收端发送的对于该数据报文段的确认应答（ACK），继续正常工作；如果在定时器超时的时候，发送端仍然没有收到对应的确认应答，那么发送端就会认为该数据报文段在网络中丢失或被丢弃了。此时，发送端会重新发送该数据报文段，并重新启动定时器进行等待。这个过程可能会重复多次，直到收到确认应答或达到最大重传次数为止。

从服务器端截获的针对seq为1207114的报文：



第一套报文因为sshv2协议加密无法直接查看，打开查看详细内容：

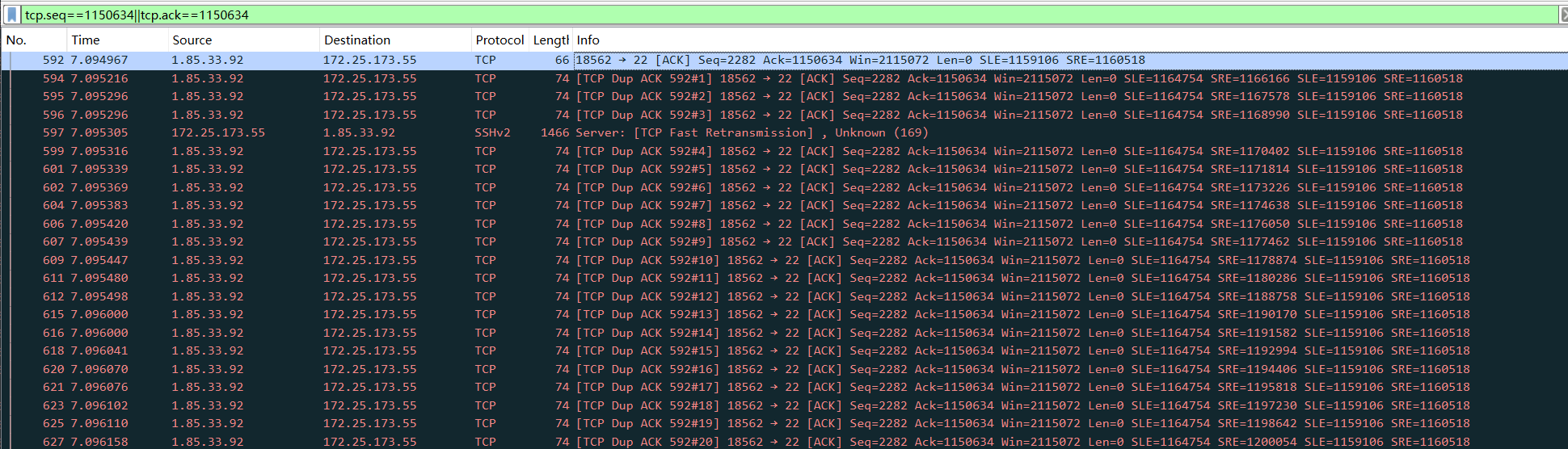


可以看到sequence号为1207114。

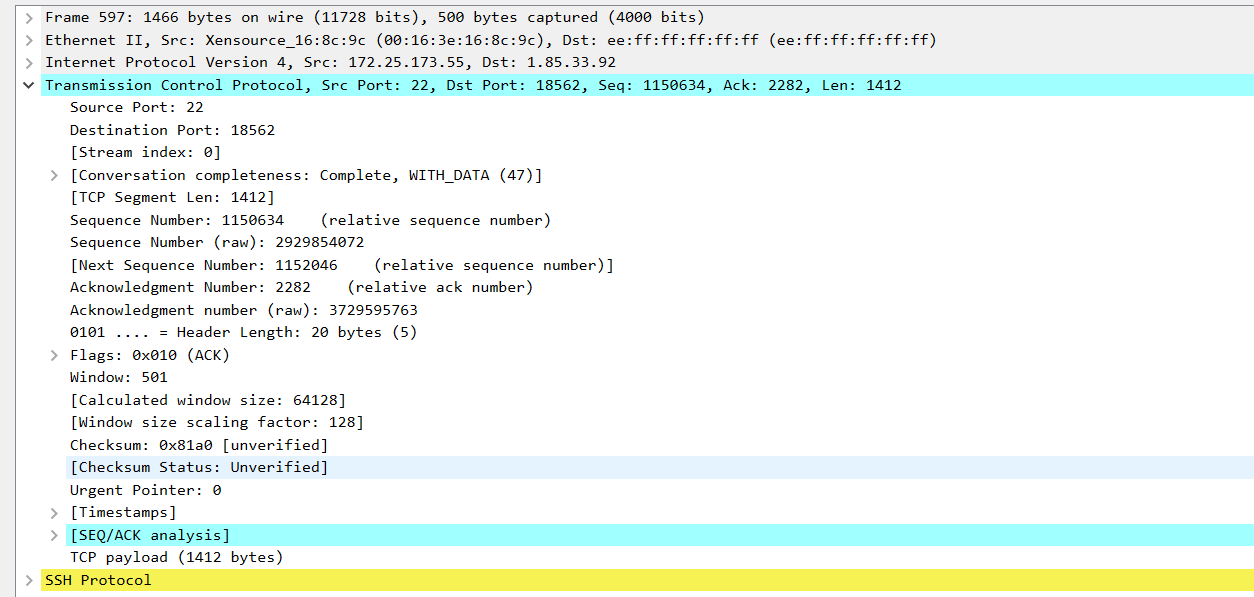
在服务器第一次发送完Sequence号为1207114的包后，等待一段时间后还没有接收到没有收到过ACK号大于1207114的包，于是启动超时重发TCP Retransiation机制重新发送。后续收到ACK号大于1207114的包后，说明sequence为1207114的包被正常接收。

快速重传机制：快速重传的触发条件主要是基于接收端发送的重复确认应答（Duplicate Acknowledgment，简称Dup ACK）。当发送端连续收到三个或更多个重复的ACK时，它会预测某个数据包可能已丢失，并触发快速重传机制。这种情况下，发送端不需要等待超时定时器到期，而是立即重传丢失的数据包。

从客户端截获的针对seq为1150634的报文：



597号包的详细信息：



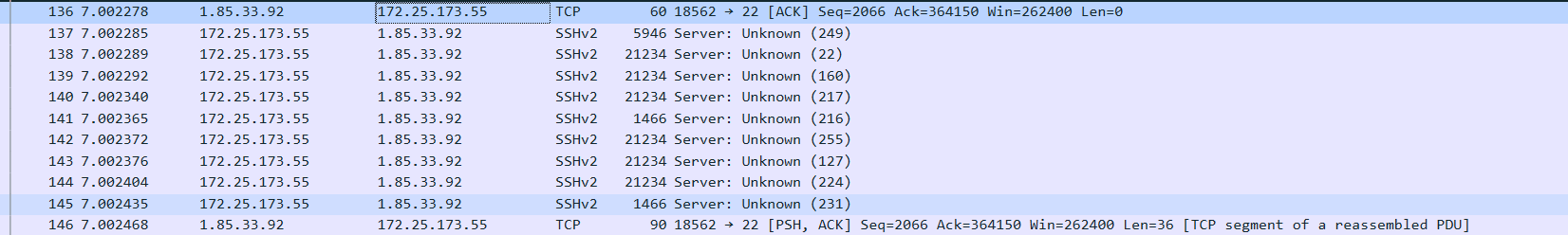
从服务器视角出发，在连续收到3个序列号为1150634的重复请求后，不等超时计时器到达触发值就开启超时重传。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 报文序号 | 报文种类 | 序号字段Seq Number | 确认号Ack Number | 数据长度 | 窗口大小 |
| 592 | 确认 | 2282 | 1150634 | 0 | 2115072 |
| 594 | 确认 | 2282 | 1150634 | 0 | 2115072 |
| 595 | 确认 | 2282 | 1150634 | 0 | 2115072 |
| 596 | 确认 | 2282 | 1150634 | 0 | 2115072 |
| 597 | 数据 | 1150634 | 2282 | 1412 | 64128 |

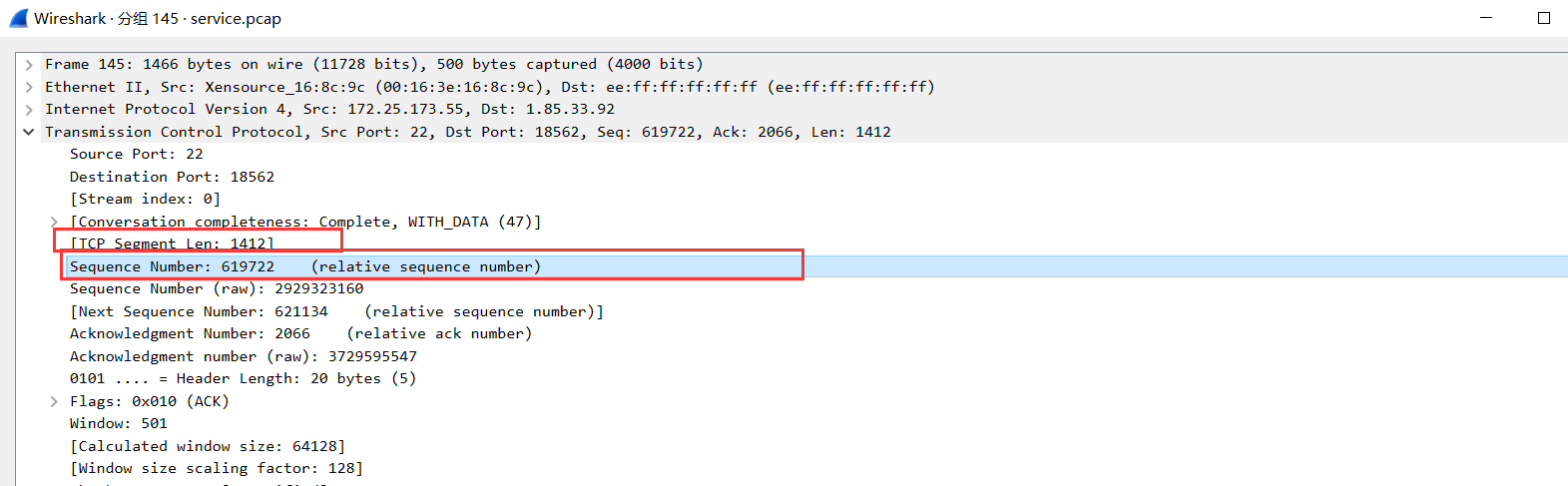
流量控制：

TCP流量控制的主要目的是根据接收方的实际接收能力，来控制发送方的数据发送速率。换句话说，流量控制旨在确保接收方能够按照其处理能力来处理数据，避免数据发送过快而导致接收方缓冲区溢出或数据丢失。

从服务器抓包：



第145号报文的具体内容如下：



客户端最后一次向服务器发送应答的ACK为364150，窗口大小为262400，服务器发送的最后一个报文Sequence值为619722，长度为1412，

已发送未应答的长度：619722+1412-364150=256984

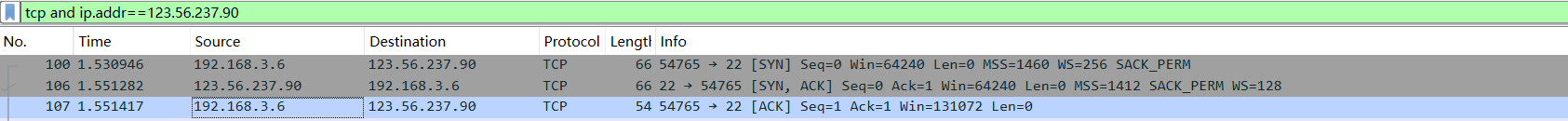
可能剩余的最小剩余缓冲区大小：262400-256984=5416

剩余窗口大小只剩5416，为了保证数据不溢出，服务器直到收到客户端的应答报文后才继续发送。

步骤6、分析客户机和服务器两边各自捕获到的分组，分析整个TCP流，估计双方的RTT。

第一次抓包：

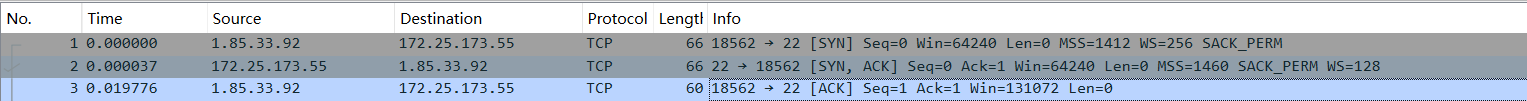
1. Client端



客户端于 1.530946s 发出第一次握手报文，于 1.551282s 收到第二次握手报文，故 RTT1=1.551282s - 1.530946s =20.336ms

由于故障导致的终止重置等原因未抓包到挥手过程，故用握手时的往返时延大致代表整个流传输过程中的RTT，也即RTT=20.336ms.

2. 服务端

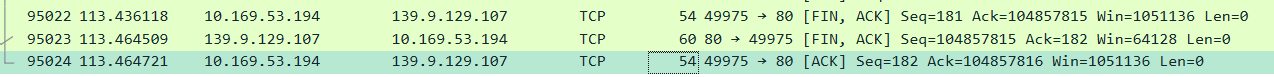


客户端于 0.000037s 发出第二次握手报文，于 0.019776s 收到第二次握手报文，故 RTT1=0.019776s -0.000037s =19.739ms

由于故障导致的终止重置等原因未抓包到挥手过程，故用握手时的往返时延大致代表整个流传输过程中的RTT，也即RTT=20.336ms.

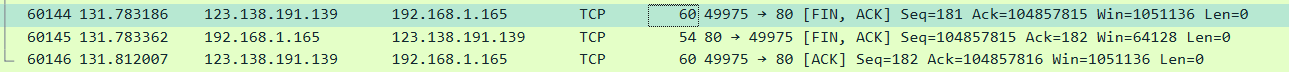
第二次抓包：

1. Client端

客户端于 113.436118s 发出第一次握手报文，于 113.464509s 收到第二次握手报文，故 RTT2=113.464509s - 113.436118s = 28.391ms

故计算得出的客户端往返延时RTT=(RTT1+RTT2)/2=24.3635ms

2. 服务端



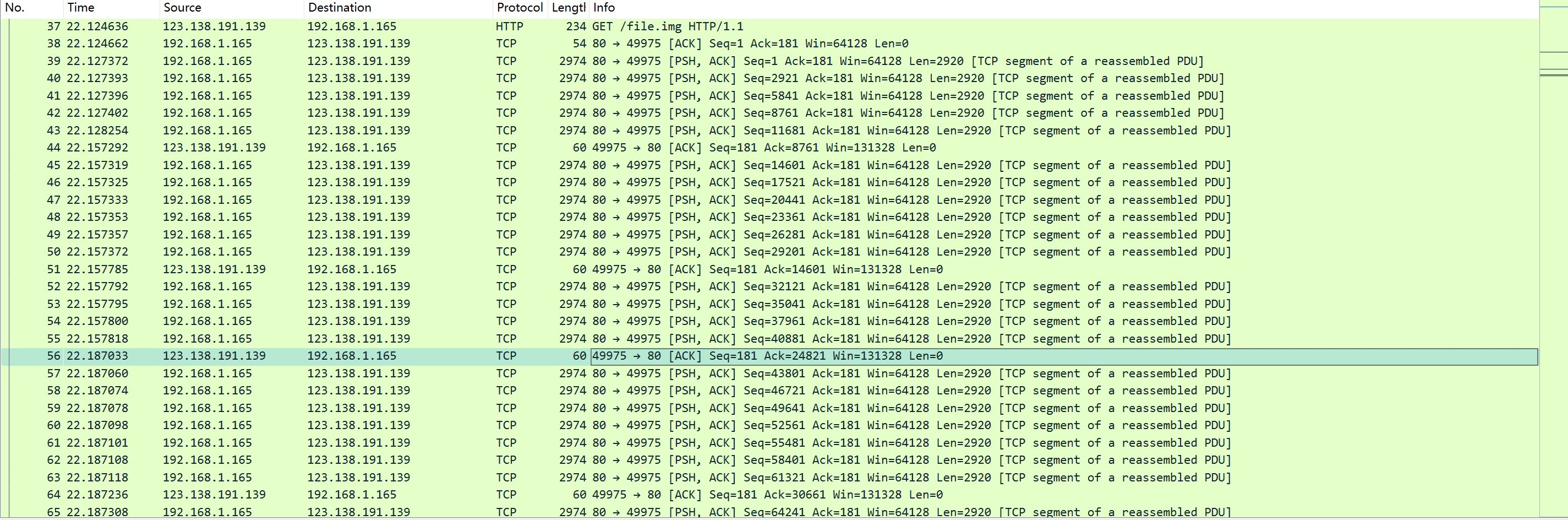
服务器于 131.78.3362s 发出第二/三次挥手报文，于 131.812007s 收到第四挥手报文，故 RTT2=131.812007s -131.783362s =28.645ms。

故计算得出的服务器往返延时RTT=(RTT1+RTT2)/2=24.192ms

步骤7、分析整个TCP流的拥塞控制，找到拥塞控制的几个典型过程（即慢启动、快速重传等）。

**此步骤分析的数据包来自于第二次实验的数据!**

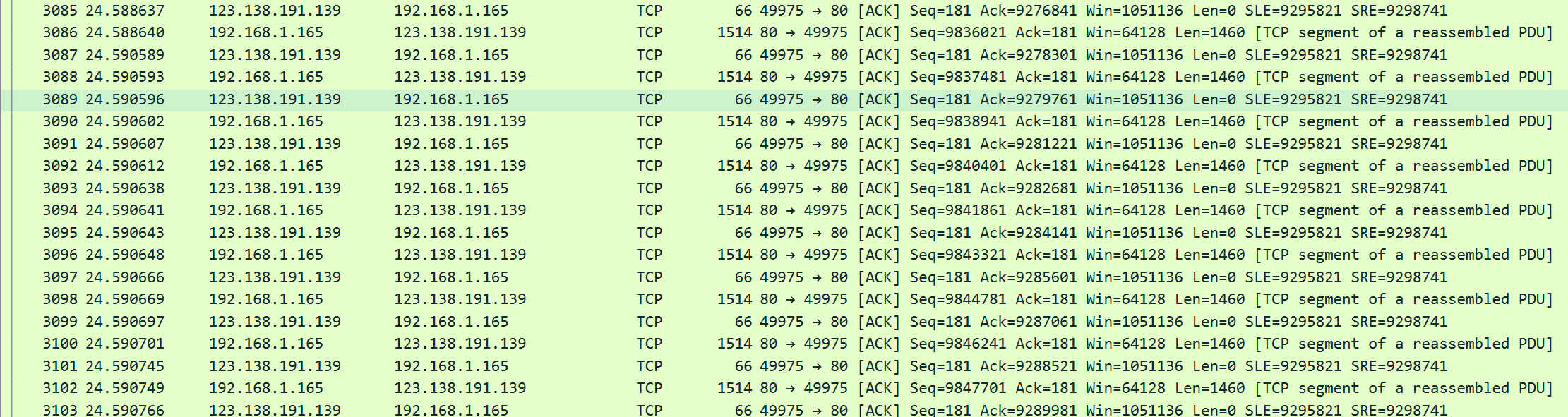
1. 慢启动



在建立连接后，客户端发出HTTP请求，于是初始时服务端发送了10个（初始拥塞窗口大小）MSS大小的数据报文，在第44条确认报文确认接受了6个MSS大小的数据包后，窗口大小变为16，除去尚未确认的10-6=4个报文外，还可发送12个MSS大小的报文，因此在第45-50条，共计发送了12个MSS大小的报文。此后的过程如下表所示：

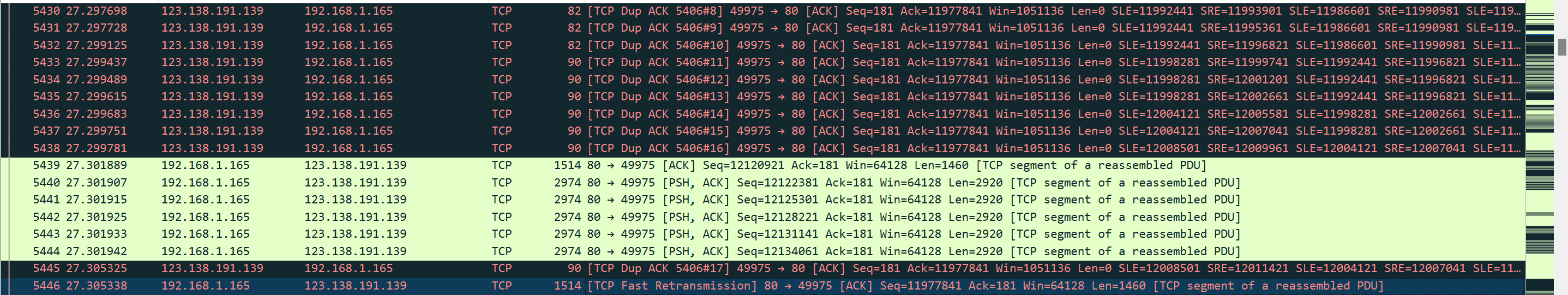
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 服务端 | 报文（单位MSS） | 客户端 | 窗口大小（MSS） |
| 51 |  | 4 ACK（累计10） | ← | 16 |
| 52 - 55 | → | 8 DATA |  | 16 |
| 56 |  | 7 ACK（累计17） | ← | 23 |
| 57 - 63 | → | 14 DATA |  | 23 |

2. 拥塞避免

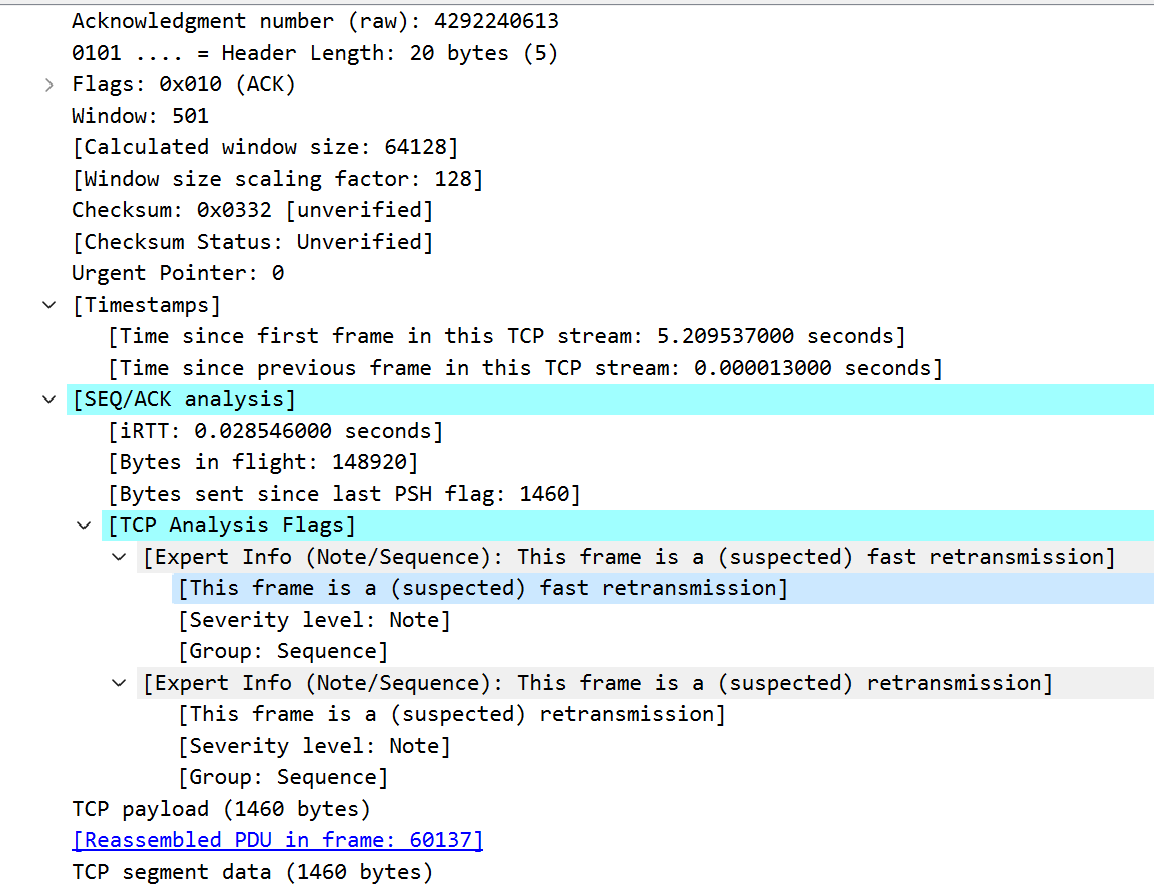


在上面的过程中，发送方每次的发送都只有一个MSS，且没有出现TCP Window Full的提示，这说明此时拥塞窗口不再是在慢启动阶段，而是已经达到了慢启动门限，开始进入拥塞避免阶段，拥塞窗口大小以线性时间（往返时延RTT）增长。

3．快速重传



可以看到在第5446条快速重传前，服务端接收到了客户端连续的七条针对同一字节的确认。因此在5446条没等到重传计时器结束就立即开启了重传。可见此时处于快速重传阶段。



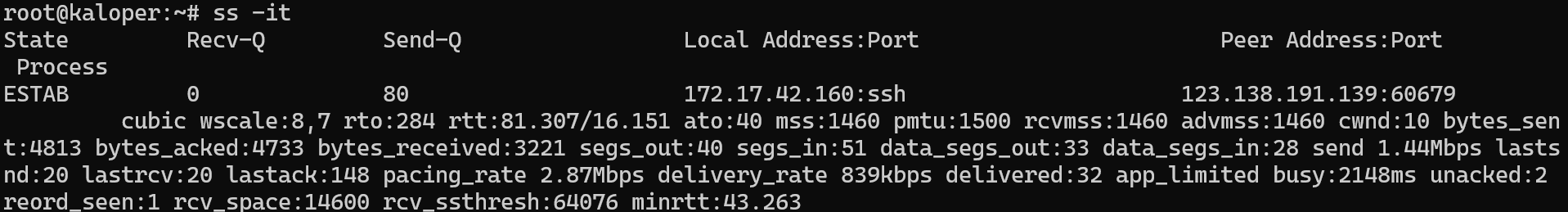
步骤8、如果拥塞控制的相关过程不明显，请设计合适的方法再次测试。

步骤9、完成其他可选的实验步骤。

9.1、在上述的步骤七慢启动和拥塞控制的过程中，拥塞窗口是一个及其重要的参数，其代表能够发送出去的但还没有收到ACK的最大数据报文段，因此他的初始值的选择对于网络拥塞控制显得十分重要，因此我们设计**观察初始的窗口默认值是多少：**

1.Linux – Ubuntu22.04

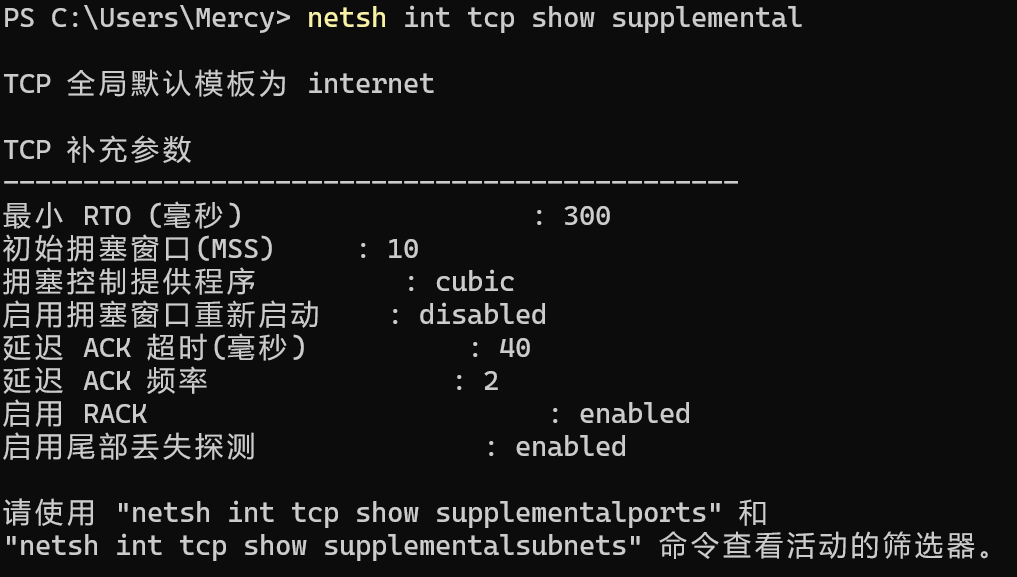
由于在本次实验中我们的服务端选用的是Linux-Ubuntu操作系统，因此在抓包得到的数据中我们已经观察到了服务器最开始传输了10哥MSS数据段给客户端，然后再等待客户端的ACK报文和保温内的windowssize更新新的窗口大小的指针后才继续发送数据，因此我们推测服务端的初始拥塞窗口的大小是10哥MSS，然后在服务端命令行使用ss -it指令可以看到其初始cwnd确实是10哥MSS大小。





2.Windows 11：

客户端使用Windows操作系统，我们使用netsh int tcp show supplemental命令即可观察到所需要的初始拥塞窗口的大小：





可以观察到Windows操作系统的初始拥塞窗口大小也是10个MSS。

1. 互动讨论主题
2. TCP的流量控制和拥塞控制有什么不同？

答：首先我们要明确两个名词面向的主体是谁：前者（流量控制）是面向于接收方和发送方的概念，后者是面向于整个网络传输过程的概念。具体来讲，两个的细节如下：

流量控制解决的是发送方和接收方速率不匹配的问题，有的时候存在发送方发送数据过快，但是接收方的处理速度跟不上，导致来不及接收和处理。具体的解决办法是给发送方添加一个滑动窗口的机制，用来控制的是发送了但未被Ack的包数量。其中的重要参数是窗口大小。

拥塞控制解决的是避免同一时间多个客户端和服务端相互发送数据导致整个网络发生“堵车”的问题，具体的措施包括慢启动、拥塞避免、快速重传和快速恢复等。通过大家自律的采取避让的措施，来避免网络有限资源被耗尽。当出现丢包时，控制发送的速率达到降低网络负载的目的。

两者的联系是：滑动窗口的窗口大小和拥塞控制的过程息息相关，其中涉及到了上面实验所探究的慢启动、拥塞控制等过程。

1. TCP的流量控制是哪一方（接收、发送）来主导的？什么情况下会发生流量控制？

答：TCP的流量控制主要是由接收方来控制。主要是因为在流量控制算法中，窗口大小cwnd严格限制了发送方在未接受到Ack时一次最多能发送的信息量的大小，而这个过程中的关键参数窗口大小又是由接收方的处理速度和发送ACK所决定的（确保不会发送超过接收方能够处理的数据量）。

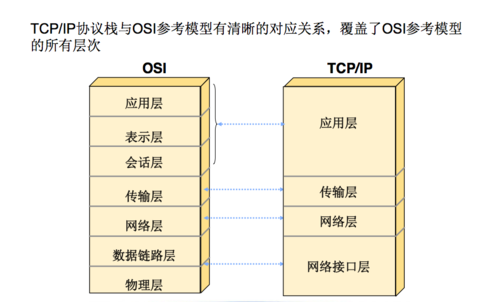
一般来说，比较常见的两种会发生流量控制的情况如下：

1. 接收方的缓冲区空间有限：如果接收方的缓冲区已经满了，它将无法接收更多的数据，因此需要通知发送方减慢发送速率，以免造成数据丢失或缓冲区溢出。（常见的处理手段是告知发送方发送窗口大小为1，从而阻止发送方再次发送数据）

2. 接收方处理能力有限：即使接收方的缓冲区还有空间，但它可能处理数据的能力有限，无法及时处理大量的数据。在这种情况下，接收方同样会通过通告窗口大小来控制发送方的发送速率，以保证自己能够及时处理接收到的数据。（常见的方式是在ACK报文中减小当前的窗口大小，以减缓发送方的发送速率）

3）讨论传输层与其上下相邻层的关系；

答：传输层是TCP/IP体系结构中的第三层，位于网络层之上。



它负责提供端到端的数据传输服务。传输层的主要功能是建立端到端的连接、数据的分段和重组、流量控制和拥塞控制等。传输层与其上下相邻层的关系如下：

1. 与应用层的关系：

传输层为上层应用层提供了可靠的、端到端的数据传输服务，通过简单的接口使的应用层可以轻松的发送、接收数据。传输层为应用层提供了端到端的通信服务，使得不同主机上的应用程序能够通过网络进行数据交换；传输层通过提供不同的传输协议（如 TCP 和 UDP）来满足应用程序的不同需求，如可靠性传输和无连接传输等；应用层和传输层之间的接口包括了端口号，应用程序通过端口号与传输层通信以进行数据传输。

2.与网络层的关系：

传输层和网络层一起构成了 TCP/IP 协议栈中的核心部分；传输层使用网络层提供的服务（如 IP 协议）来实现端到端的数据传输。网络层负责将数据包从源主机传输到目标主机，而传输层则负责在源主机和目标主机之间建立和管理传输连接；传输层通过网络层提供的数据报服务（如 IP 数据报）来传输数据段，实现了网络的透明性，使得网络层的变化对传输层来说是透明的。

4）讨论TCP协议在传输实时语音流方面的优缺点。

TCP显然是可以完成语音流数据传输的任务的，但是工业界常常采用UDP来传输语音、视频信息流主要是因为TCP在这一特定任务上具有如下的缺点：

1. 延迟： TCP 的可靠性带来了一定的延迟，例如在数据包丢失时需要等待重传或接收到全部数据段后才能进行数据交付。对于实时语音通话，即时性是非常重要的，因此 TCP 的延迟可能影响通话的实时性和响应性。

2. 头部开销： TCP 头部较大，包含了序号、确认号、窗口大小等信息，这会增加数据包的大小，导致额外的网络开销。对于实时语音流，传输的数据量通常较小，因此造成了传输的低效率。

不过，TCP在这个任务上也具有一定的优点：

1. 可靠性： TCP 提供了可靠的数据传输机制，减少了语音信息中的断续和失真现象。

3. 有序交付： TCP 保证了数据包的有序交付，即数据包按照发送的顺序在接收端被重新组装，这对于语音信息来说至关重要，因为声音的顺序错乱可能导致内容难以理解。