



# 实验报告

## 实验五

学 院	信息学院
学 号	22920212204066
姓 名	邓语苏
课程名称	计算机网络
日 期	2023 年 12 月 20 日

# 实验五

## 目录

<b>1 实验目的与要求</b>	<b>1</b>
<b>2 实验内容</b>	<b>1</b>
<b>3 TCP 正常连接观察</b>	<b>1</b>
<b>4 TCP 异常传输观察分析</b>	<b>4</b>
4.1 尝试连接未存活的主机或对未监听端口 . . . . .	4
4.2 客户端发送了第一个 SYN 连接请求，服务器无响应的情景 . . . . .	7
4.3 SYN 洪泛 . . . . .	9
<b>5 TCP 拥塞控制</b>	<b>10</b>
5.1 实验准备 . . . . .	10
5.2 实验结果 . . . . .	10
5.3 TCP 拥塞控制算法对比 . . . . .	11
<b>6 UDP 协议观察</b>	<b>12</b>
6.1 实验准备 . . . . .	12
<b>7 HTTP 协议分析</b>	<b>13</b>
7.1 实验准备 . . . . .	13
7.2 实验结果 . . . . .	14
<b>8 实验总结</b>	<b>14</b>

## 1 实验目的与要求

TCP(Transmission Control Protocol 传输控制协议) 是一种面向连接的、可靠的、基于字节流的传输层通信协议。本实验通过运用 Wireshark 对网络活动进行分析, 观察 TCP 协议报文, 分析通信时序, 理解 TCP 的工作过程, 掌握 TCP 工作原理与实现; 学会运用 Wireshark 分析 TCP 连接管理、流量控制和拥塞控制的过程, 发现 TCP 的性能问题。

## 2 实验内容

启动 Wireshark, 捕捉网络活动中的 TCP 报文并按要求分析。

1. 连接管理: 观察正常 TCP 连接中的三次握手与四次挥手报文, 绘制出时序图, 并标出双方 TCP 状态变化。
2. 异常情况分析: 观察分析 TCP 连接建立过程的异常 (例如, 尝试连接未存活的主机或未监听端口, 客户端发送了第一个 SYN 连接请求而服务端无响应); 观察 SYN 洪泛影响; 观察分析 TCP 通信过程中的各类异常报文 (例如数据超时、乱序), 了解其触发机制与含义。
3. 流量控制 (进阶): 运行一组 TCP 连接客户端/服务器程序, 制造收发不平衡场景, 观察收发报文中通告窗口的变化, 分析与窗口机制相关的类型报文, 了解滑动窗口工作原理。
4. 拥塞控制 (进阶): 改变带宽、时延、丢包率等网络参数, 观察大文件传输过程, 分析识别 TCP 的慢启动、拥塞避免、快速恢复等拥塞控制阶段; 在构建的网络试验环境下运行 iperf3 进行网络性能测试, 比较不同拥塞控制策略的性能表现。

## 3 TCP 正常连接观察

1. 在【PC2】上启动一个建议的 web 服务器

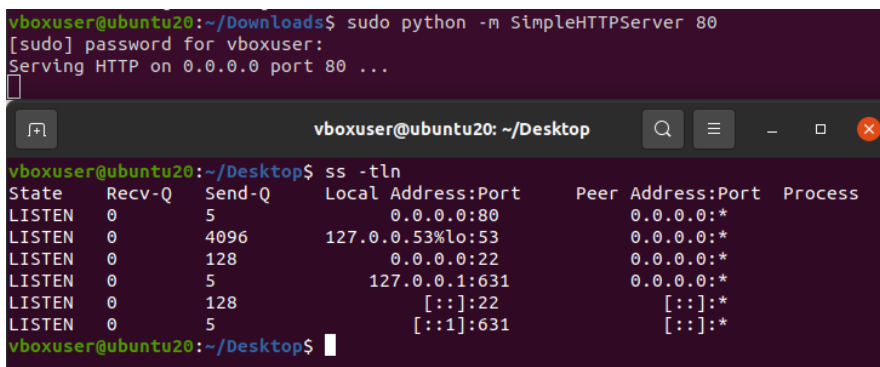
- cd 命令切换到目标路径
- 安装 python 库

```
1 sudo apt-get install python
```

- 在 80 端口开启 web 服务器

```
1 sudo python -m SimpleHTTPServer 80
```

- 确认 80 端口处理监听状态



```
vboxuser@ubuntu20:~/Downloads$ sudo python -m SimpleHTTPServer 80
[sudo] password for vboxuser:
Serving HTTP on 0.0.0.0 port 80 ...

vboxuser@ubuntu20: ~/Desktop
vboxuser@ubuntu20:~/Desktop$ ss -tln
State      Recv-Q    Send-Q    Local Address:Port    Peer Address:Port    Process
LISTEN     0          5         0.0.0.0:80            0.0.0.0:*
LISTEN     0         4096      127.0.0.53%lo:53      0.0.0.0:*
LISTEN     0          128      0.0.0.0:22            0.0.0.0:*
LISTEN     0          5        127.0.0.1:631         0.0.0.0:*
LISTEN     0         128      [::]:22               [::]:*
LISTEN     0          5        [::1]:631             [::]:*
```

2. 在【PC1】上使用 wireshark 抓包

- 打开终端输入 `sudo wireshark` 抓包
- 在 PC2 上使用 `ifconfig` 查看 PC2 的 IP 地址
- 在 PC1 上键入

```
1 curl 192.168.122.132
```

- 使用 wireshark 过滤出 ip 地址为 192.168.122.132 的 TCP 报文

Time	Source	Destination	Protocol	Length	Info
1.0.00000000	10.0.2.15	192.168.122.132	TCP	76	44692 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=248019554 TSecr=0 WS
2.0.00107688	192.168.122.132	10.0.2.15	TCP	62	80 → 44692 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
3.0.00110382	10.0.2.15	192.168.122.132	TCP	56	44692 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0
4.0.001270673	10.0.2.15	192.168.122.132	HTTP	135	GET / HTTP/1.1
5.0.001545351	192.168.122.132	10.0.2.15	TCP	62	80 → 44692 [ACK] Seq=1 Ack=80 Win=65535 Len=0
6.0.002647250	192.168.122.132	10.0.2.15	TCP	2976	80 → 44692 [ACK] Seq=1 Ack=80 Win=65535 Len=2920 [TCP segment of a reassembled PDU]
7.0.002656339	10.0.2.15	192.168.122.132	TCP	56	44692 → 80 [ACK] Seq=80 Ack=2921 Win=62780 Len=0
8.0.003208015	192.168.122.132	10.0.2.15	TCP	4436	80 → 44692 [ACK] Seq=2921 Ack=80 Win=65535 Len=4380 [TCP segment of a reassembled PDU]
9.0.00321357	10.0.2.15	192.168.122.132	TCP	56	44692 → 80 [ACK] Seq=80 Ack=7301 Win=61320 Len=0
10.0.005877939	192.168.122.132	10.0.2.15	HTTP	3929	HTTP/1.1 200 OK (text/html)
11.0.005886316	10.0.2.15	192.168.122.132	TCP	56	44692 → 80 [ACK] Seq=80 Ack=11174 Win=61320 Len=0
12.0.006200213	10.0.2.15	192.168.122.132	TCP	56	44692 → 80 [FIN, ACK] Seq=80 Ack=11174 Win=62780 Len=0
13.0.006501710	192.168.122.132	10.0.2.15	TCP	62	80 → 44692 [ACK] Seq=11174 Ack=81 Win=65535 Len=0
14.0.006914021	192.168.122.132	10.0.2.15	TCP	62	80 → 44692 [FIN, ACK] Seq=11174 Ack=81 Win=65535 Len=0
15.0.006928520	10.0.2.15	192.168.122.132	TCP	56	44692 → 80 [ACK] Seq=81 Ack=11175 Win=62780 Len=0

- 观察 TCP 报首部

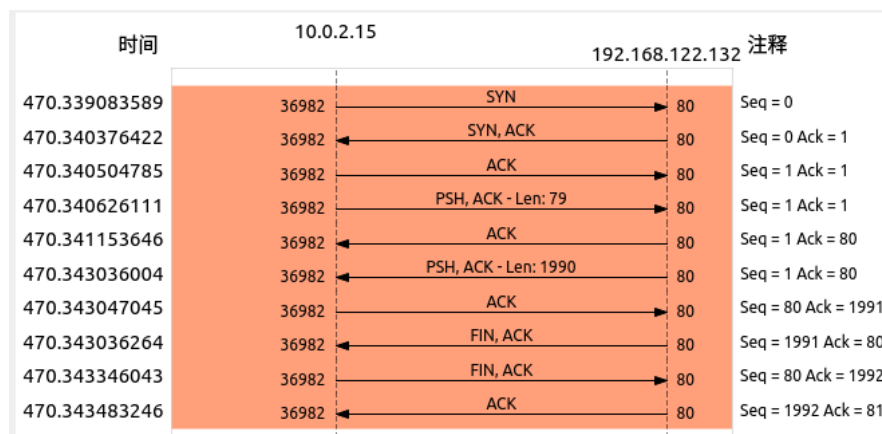
Frame 9524: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on interface any, id 0	
Linux cooked capture v1	
Internet Protocol Version 4, Src: 10.0.2.15, Dst: 192.168.122.132	
Transmission Control Protocol, Src Port: 36982, Dst Port: 80, Seq: 0, Len: 0	
Source Port: 36982	
Destination Port: 80	
[Stream index: 221]	
[Conversation completeness: Complete, WITH_DATA (31)]	
[TCP Segment Len: 0]	
Sequence Number: 0 (relative sequence number)	
Sequence Number (raw): 1111732108	
[Next Sequence Number: 1 (relative sequence number)]	
Acknowledgment Number: 0	
Acknowledgment number (raw): 0	
1010 .... = Header Length: 40 bytes (10)	
Flags: 0x002 (SYN)	
Window: 64240	
[Calculated window size: 64240]	
Checksum: 0x476a [unverified]	
[Checksum Status: Unverified]	
Urgent Pointer: 0	
Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP),...	
TCP Option - Maximum segment size: 1460 bytes	
TCP Option - SACK permitted	
TCP Option - Timestamps	
TCP Option - No-Operation (NOP)	
TCP Option - Window scale: 7 (multiply by 128)	
[Timestamps]	
[Time since first frame in this TCP stream: 0.00000000 seconds]	
[Time since previous frame in this TCP stream: 0.00000000 seconds]	
000	00 04 00 01 00 06 08 00 27 60 2d db 00 00 08 00
010	45 00 00 3c cd 3e 40 00 40 06 26 42 0a 00 02 0f
020	c0 a8 7a 84 90 76 00 50 42 43 af 8c 00 00 00 00
030	a0 02 fa f0 47 6a 00 00 02 04 05 b4 04 02 08 0a
040	31 e6 a3 0a 00 00 00 00 01 03 03 07

- Source Port (源端口): 36982  
表示报文的来源端口号, 用于标识发送方的应用程序或服务。
- Destination Port (目标端口): 80  
表示报文的目标端口号, 指明接收方的应用程序或服务。
- Stream Index (流索引): 2217  
表示流的索引, 可能用于标识数据流的唯一性或顺序性。
- Conversation Completeness (会话完整性): Complete, WITH DATA (31)  
表示这次通信的完整性状态, 包括数据的存在和大小。
- Tcp Segment Len (TCP 段长度): 0  
表示 TCP 段的长度, 这里为 0, 可能表明这是一个控制报文而非携带数据的报文。
- Sequence Number (序列号): 1111732108  
该 TCP 报文段中第一个数据字节的序号, 随机生成
- Next Sequence Number (下一个序列号): 1  
表示下一个期望接收的序列号。

- Acknowledgment Number (确认号): 0  
表示对方期望接收的下一个序列号。
- Flags (标志): 0x002 (SYN)  
表示 TCP 报文的控制标志, 这里是 SYN 标志, 表示发起一个连接请求。
- Window (窗口大小): 64240  
表示接收方可接收的字节数, 用于流量控制。
- Calculated Window Size (计算后的窗口大小): 642407  
表示通过计算得到的窗口大小。
- Checksum (校验和): 0x476a (未验证)  
表示校验和, 用于检测传输过程中是否发生错误, 但这里显示为未验证。
- Checksum Status (校验和状态): Unverified  
表示校验和的验证状态为未验证。
- Urgent Pointer (紧急指针): 0  
表示紧急数据的位置。
- Options (选项): (20 bytes)  
包括 Maximum Segment Size、SACK Permitted、Timestamps、No-Operation (NOP)、Window Scale 等选项。  
表示 TCP 报文的可选字段, 这里包含了多个选项, 如最大段大小、SACK 允许、时间戳、无操作、窗口缩放等。
- Maximum Segment Size (最大段大小): 1460 bytes  
表示 TCP 连接中允许的最大数据段大小。
- SACK Permitted (SACK 允许): 1  
表示选择性应答允许, 用于指示对方可以使用选择性应答选项。
- Timestamps (时间戳): 存在  
表示时间戳选项存在, 用于计算往返时间。
- No-Operation (NOP) (无操作):  
表示无操作选项, 用于填充以保持选项字段的对齐。
- Window Scale (窗口缩放): 7 (乘以 128)  
表示窗口缩放因子, 用于扩大窗口大小的倍数。
- Timestamps (时间戳):  
表示时间戳选项, 用于测量往返时间等。
- First Frame in this TCP Stream (该 TCP 流的第一帧): 0.000 秒  
表示该 TCP 流的第一帧出现的时间。
- Time Since Previous Frame in this TCP Stream (自上一帧以来的时间): 0.000 秒  
表示自上一帧以来的时间。

- 观察 TCP 序号和确认号的工作流程

使用 wireshark 工具中的绘制流功能, Flow Type 选择 TCP flows



### 3. 绘制 TCP 连接时序图

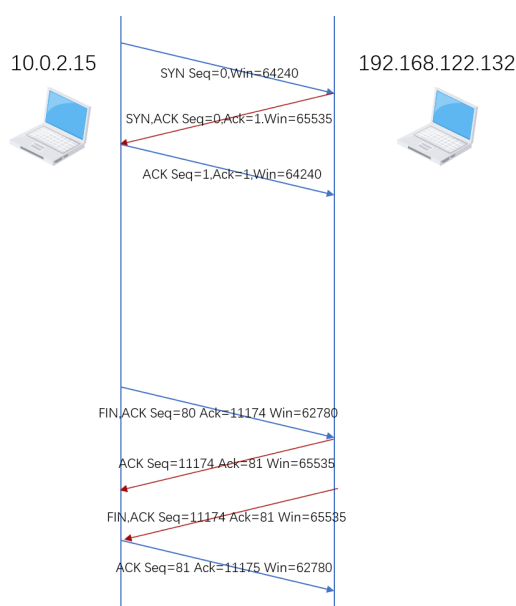


图 1: TCP 连接示意时序-实际

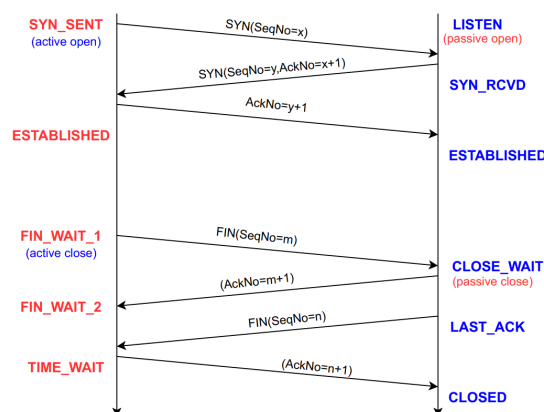


图 2: TCP 连接示意时序-理论

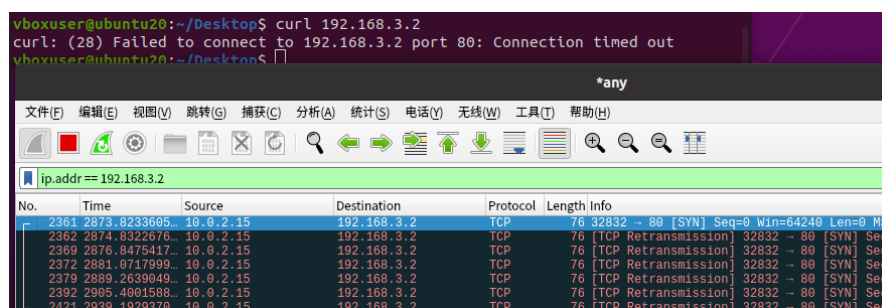
分析更具 wireshark 绘制的实际 TCP 连接示意时序图，可以看出与理论结果相符合

## 4 TCP 异常传输观察分析

### 4.1 尝试连接未存活的主机或对未监听端口

#### 1. 用 curl 访问一个不存在的主机 IP

我使用 curl 访问了不存在的主机 192.168.3.2



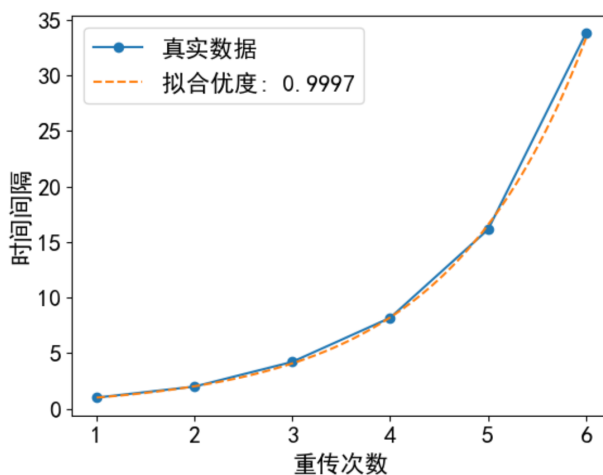
共发送了 7 次 SYN 报文, 其中 6 次为重传

根据 Time 信息计算时间间隔如

1→2	2→3	3→4	4→5	5→6	6→7
1.009	2.000	4.2242	8.1922	16.1362	33.7928

观察间隔时间, 可以发现忽略测量误差, 时间间隔是按  $2^n$  指数增长的。

使用 python 绘制折线图与  $2^n$  拟合曲线图进一步验证结论。



拟合优度为 0.9997, 非常接近 1, 说明拟合良好

2. 查看 Linux 主机的系统的 TCP 参数 SYN 重传设定

```
vboxuser@ubuntu20:~/Desktop$ cat /proc/sys/net/ipv4/tcp_syn_retries
6
```

3. 更改 SYN 重传次数为 3

```
1 sudo echo "3" > /proc/sys/net/ipv4/tcp_syn_retries
```

在进行这一步的时候我出现了权限问题, 即执行此命令的时候会返回 Permission denied。后面发现是 `tcp_syn_retries` 为只读文件, 我就想着把这个文件的权限修改为 777

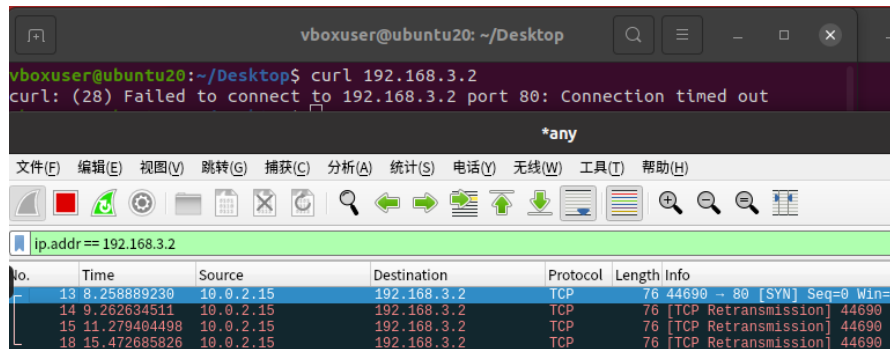
```
1 sudo chmod 777 /proc/sys/net/ipv4/tcp_syn_retries
```

仍然是 Permission denied。最后解决方法为

```
1 sudo su %进入root权限
2 sudo echo "3" > /proc/sys/net/ipv4/tcp_syn_retries
```

```
vboxuser@ubuntu20:~/Desktop$ sudo su
[sudo] password for vboxuser:
root@ubuntu20:/home/vboxuser/Desktop# sudo echo "3" > /proc/sys/net/ipv4/tcp_syn_retries
root@ubuntu20:/home/vboxuser/Desktop# cat /proc/sys/net/ipv4/tcp_syn_retries
3
```

4. 再次 curl 访问, 观察抓包内容。



一共发送了 4 次 SYN 报文，其中 3 次为重传

5. 关闭服务器端的 SimpleHTTPServer(Ctrl+C 中断，或关闭所在终端)，客户端 curl 访问服务器 8000 端口，观察应答报文

注意：我的 PC2 虚拟机总是自动监听 80 端口，不符合要求，因此更换端口为 8000

在 PC2 中键入

```
1 sudo python -m SimpleHTTPServer 8000
```

后使用 Ctrl+C 退出

在 PC1 中键入

```
1 curl 192.168.122.132:8000
```

使用 Wireshark 抓包结果如下

Source	Destination	Protocol	Length	Info
10.0.2.15	192.168.122.132	TCP	76	43030 → 8000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SAC
10.0.2.15	192.168.122.132	TCP	76	[TCP Retransmission] 43030 → 8000 [SYN] Seq=0 Win=642
192.168.122.132	10.0.2.15	TCP	62	8000 → 43030 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

6. 运行 nmap -sS <PC2 的 IP> 扫描服务器，并抓包

主要注意的是，nmap 需要用 root 权限操作

```
1 sudo su
2 nmap -sS 192.168.122.132
```

```
vboxuser@ubuntu20:~/Desktop$ sudo su
root@ubuntu20:/home/vboxuser/Desktop# nmap -sS 192.168.122.132
Starting Nmap 7.80 ( https://nmap.org ) at 2023-12-11 18:30 CST
Nmap scan report for 192.168.122.132
Host is up (0.0053s latency).
Not shown: 995 filtered ports
PORT      STATE SERVICE
7/tcp    open  echo
21/tcp   open  ftp
22/tcp   open  ssh
80/tcp   open  http
81/tcp   open  hosts2-ns
Nmap done: 1 IP address (1 host up) scanned in 4.21 seconds
```

它指出本机的 SSH 端口、FTP 端口等是开放的。



## 7. 解 SYN 扫描原理

用 Wireshark 观察抓到的包, 例如 FTP 端口 21, 如图 3 所示。

可见, 由于 21 端口是开放的, 服务器得到 SYN 请求后会回复 ACK 准备建立连接。此时扫描程序得知了 21 端口是开放的, 但为了防止 IP 黑名单, 就发送 RST 拒绝建立连接, 此时服务器不能建立连接。因此无法记录客户 IP 进行屏蔽。这样就使得扫描能够顺利完成。如图 3 所示。

图 3: 21 端口开放

54 TCP	1720 → 34094 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
58 TCP	34094 → 21 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
58 TCP	21 → 34094 [SYN, ACK] Seq=0 Ack=1 Win=65495 Len=0 MSS=65495
54 TCP	34094 → 21 [RST] Seq=1 Win=0 Len=0

又例如端口 587 未开放。因此服务器就会发送 RST,ACK 表示收到请求但拒绝建立连接。扫描程序收到后, 就会认为端口 587 未开放。如图 4 所示。

图 4: 587 端口未开放

58 TCP	34094 → 587 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
54 TCP	587 → 34094 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

## 4.2 客户端发送了第一个 SYN 连接请求, 服务器无响应的情景

### 1. 服务器安装 ssh 服务

```
1 sudo apt update
2 sudo apt install openssh-server
```

### 2. 服务器启动 ssh 服务

```
1 sudo service ssh start
```

### 3. 客户端连接服务器

```
1 ssh hadoop@192.168.122.132
```

其中 hadoop 为服务器用户名, 192.168.122.132 为服务器 ip 地址

### 4. 查看 TCP 连接状态在客户端、服务器键入 ss -tan 查看 TCP 连接状态

```
hadoop@ubuntu:~$ ss -tan
```

State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port	Process
LISTEN	0	4096	127.0.0.53%lo:53	0.0.0.0:*	
LISTEN	0	128	0.0.0.0:22	0.0.0.0:*	
LISTEN	0	5	127.0.0.1:631	0.0.0.0:*	
ESTAB	0	0	192.168.122.132:22	192.168.122.1:58616	
LISTEN	0	64	*:7	*:*	
LISTEN	0	511	*:80	*:*	
LISTEN	0	511	*:81	*:*	
LISTEN	0	32	*:21	*:*	
LISTEN	0	128	:::22	:::*	
LISTEN	0	5	:::1:631	:::*	

### 5. 在客户端, 利用 iptables 拦截服务器回应的 SYN ACK 包

```
1 sudo iptables -I INPUT -s 192.168.122.132 -p tcp -m tcp --tcp-flags ALL SYN,ACK -j DROP
```

使用 sudo iptables -S 查看是否添加成功。若成功添加, 内容如下所示

- 1 -P INPUT ACCEPT
- 2 -P FORWARD ACCEPT
- 3 -P OUTPUT ACCEPT
- 4 -A INPUT -s 192.168.122.132/32 -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG SYN,ACK -j DROP

6. 将原 TCP 连接断开后重连。使用 ss -tan、wireshark 软件观察 TCP 状态  
使用 tcpkill 将原连接断开后重新连接

- 1 sudo tcpkill -9 host 192.168.122.132 and port 22

在客户端、服务器键入 ss -tan 查看 TCP 连接状态

```
vboxuser@ubuntu20:~/Desktop$ ss -tan
State  Recv-Q  Send-Q  Local Address:Port  Peer Address:Port  Process
LISTEN  0         4096    127.0.0.53:53      0.0.0.0:*          -
LISTEN  0         128     0.0.0.0:22         0.0.0.0:*          -
LISTEN  0         5       127.0.0.1:631      0.0.0.0:*          -
LISTEN  0         128     [::]:22            [::]:*              -
LISTEN  0         5       [::]:631           [::]:*              -
```

Time	Source	Destination	Protocol	Length	Info
78.989106688	10.0.2.15	192.168.122.132	TCP	76	40268 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=2392017196 TSecr=6
79.004874441	192.168.122.132	10.0.2.15	TCP	62	22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
79.013216441	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
79.992783340	10.0.2.15	192.168.122.132	TCP	76	[TCP Retransmission] 40268 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSv
82.009826223	10.0.2.15	192.168.122.132	TCP	76	[TCP Retransmission] 40268 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSv
85.511667718	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
86.107303159	10.0.2.15	192.168.122.132	TCP	76	[TCP Retransmission] 40268 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSv
94.297754615	10.0.2.15	192.168.122.132	TCP	76	[TCP Retransmission] 40268 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSv
97.800631114	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
109.928085363	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
110.425493603	10.0.2.15	192.168.122.132	TCP	76	[TCP Retransmission] 40268 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSv
122.434469235	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
134.447872814	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
143.193860332	10.0.2.15	192.168.122.132	TCP	76	[TCP Retransmission] 40268 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSv
146.697943654	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
158.823022676	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
170.834944149	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
182.845396372	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
194.855353853	192.168.122.132	10.0.2.15	TCP	62	[TCP Retransmission] 22 → 40268 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
205.059313712	192.168.122.132	10.0.2.15	TCP	62	[TCP Keep-Alive] 22 → 40268 [ACK] Seq=0 Ack=1 Win=65535 Len=1
217.070608786	192.168.122.132	10.0.2.15	TCP	62	[TCP Keep-Alive] 22 → 40268 [ACK] Seq=0 Ack=1 Win=65535 Len=1
217.070684640	10.0.2.15	192.168.122.132	TCP	56	40268 → 22 [RST] Seq=1 Win=0 Len=0

图 5: wireshark 抓包结果

7. 服务端的 SYN-RECV 状态何时释放?

服务器收到客户端发送的复位 (RST) 包, 表示客户端不打算建立连接, 那么服务器会释放 SYN-RECV 状态。

8. SYN ACK 重传了几次, 时间间隔有何变化?

SYN ACK 重传了 5 次。

这五次重传的间隔时间如下所示

1->2	2->3	3->4	4->5
12.1251	12.0019	12.0104	12.0100

忽略测量误差, 时间间隔不变

9. 修改服务器 SYN ACK 重传次数后再次观察

- 1 sudo su
- 2 sudo echo "3" > /proc/sys/net/ipv4/tcp\_synack\_retries

10. 清空防火墙规则

- 1 sudo iptables -F

### 4.3 SYN 洪泛

1. 在服务器端禁用 syncookies

注意使用 root 权限

```
1 sudo su
2 sudo echo "0">/proc/sys/net/ipv4/tcp_syncookies
```

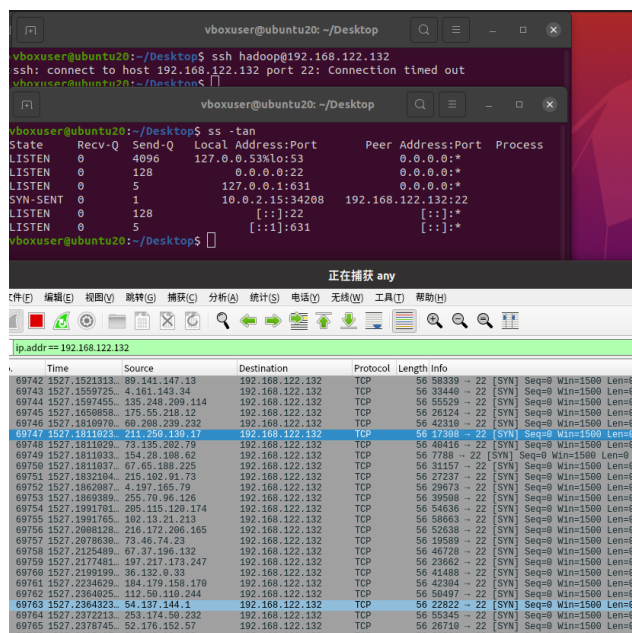
2. 指定服务器所能接受 SYN 同步包的最大客户端数量为 6

```
1 sudo sysctl -w net.ipv4.tcp_max_syn_backlog=6
```

3. 在客户端对服务器监听端口产生大量 SYN 连接请求, 再尝试用 ssh 正常连接

```
1 sudo netwox 76 -i 192.168.122.132 -p 22
2 ssh hadoop@192.168.122.132
```

4. 观察交互情况



使用 ss -tan 查看 TCP 连接情况, 发现一直处于 SYN-SENT 状态。使用 Wireshark 抓包, 发现服务端收到不同源 IP 的 SYN 请求。

5. SYN 洪泛攻击原理与对策

#### 【原理】

攻击者通过伪造源 IP 地址向目标服务器发送大量的 SYN 请求, 但不完成后续的三次握手过程。这导致目标服务器在等待确认的同时, 消耗了资源, 并在连接队列中累积了大量半开放连接。随着连接队列被填满, 服务器的资源如 TCP 端口、内存和处理能力等逐渐被占用。服务器很快达到资源极限, 无法响应正常的连接请求。

#### 【对策】

- SYN Cookies

使用 SYN Cookies 技术, 通过在 TCP 握手过程中的 SYN-ACK 包中包含有关连接的信息, 而不是在服务器端维护连接队列, 从而减轻了攻击的影响。

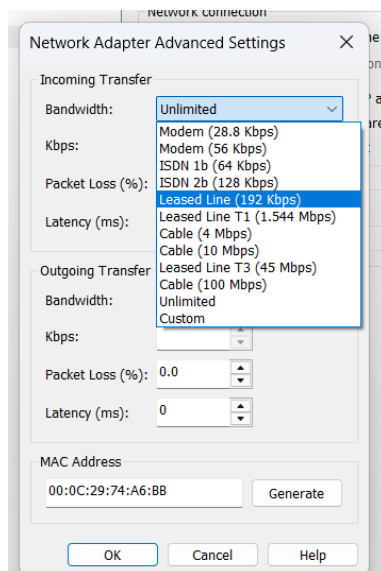
- 限制半开连接数  
设置操作系统的最大半开连接数, 以限制攻击者能够在连接队列中填充的连接数。
- 使用防火墙规则  
在网络层使用防火墙规则, 过滤掉源地址异常的 SYN 请求。

## 5 TCP 拥塞控制

### 5.1 实验准备

首先为了观察普通的拥塞控制算法, 需要对使用的拥塞控制算法进行修改。所以使用命令 `sysctl -w net.ipv4.tcp_congestion_control=reno` 修改拥塞控制算法为 Reno。然后在虚拟机设置中对带宽进行限制。如图 6 所示。

图 6: 对带宽进行限制

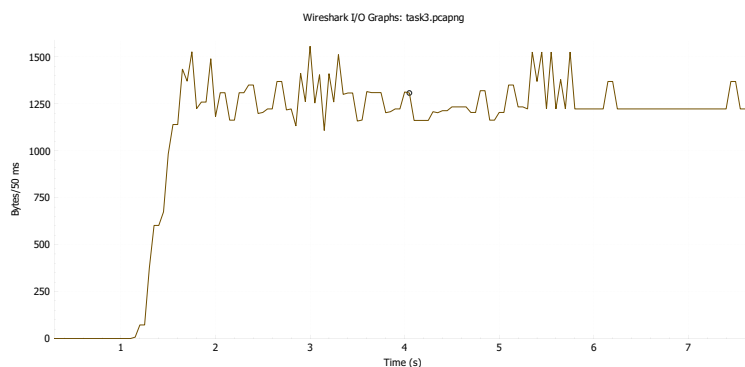


这里为了使得效果明显, 将带宽限制为 192Kbps, 然后下载 <https://mirrors.tuna.tsinghua.edu.cn/ubuntu-cdimage/releases/20.04/release/ubuntu-20.04.5-live-server-arm64.iso> 链接指向的大文件

### 5.2 实验结果

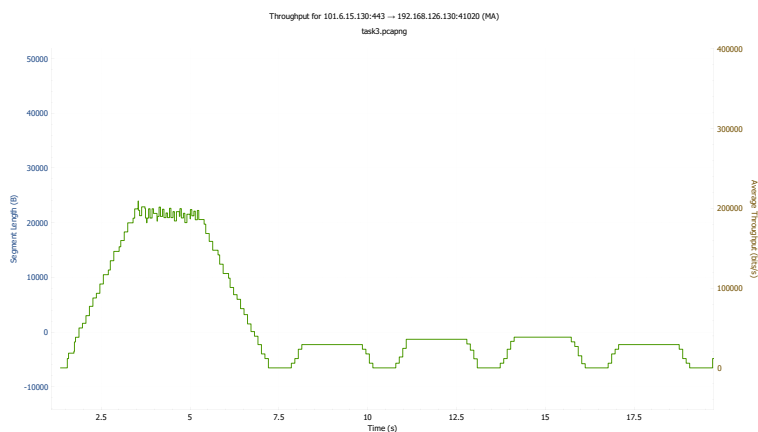
使用 Wireshark 中的 IO Graph 功能做出了如图 7 所示的图表。

图 7: IO Graph



从 I/O 图中可以看出首先, 每秒传播的字节数按照指数级增长, 然后按照线性增长。后来有一些下降的过程, 然后快速地增长, 大致对应了慢启动、拥塞避免和快速重传的过程。此外, 借助 Wireshark 绘制吞吐量图如图 8。

图 8: 吞吐量图



可以发现也遵循了类似的变化过程。

例如 2 秒左右时发生的快速重传, 首先由于链路带宽的问题, 客户端收到了不正常的序列号, 因此启动了快重传机制。如图 9 所示。

图 9: DupACK

192.168.126.130	101.6.15.130	54 TCP	[TCP Dup ACK 109#1] 41020 + 443 [ACK] Seq=726 Ack=92002 Win=65535 Len=0
192.168.126.130	101.6.15.130	54 TCP	[TCP Dup ACK 109#2] 41020 + 443 [ACK] Seq=726 Ack=92002 Win=65535 Len=0
192.168.126.130	101.6.15.130	54 TCP	[TCP Dup ACK 109#3] 41020 + 443 [ACK] Seq=726 Ack=92002 Win=65535 Len=0
192.168.126.130	101.6.15.130	54 TCP	[TCP Dup ACK 109#4] 41020 + 443 [ACK] Seq=726 Ack=92002 Win=65535 Len=0
192.168.126.130	101.6.15.130	54 TCP	[TCP Dup ACK 109#5] 41020 + 443 [ACK] Seq=726 Ack=92002 Win=65535 Len=0
192.168.126.130	101.6.15.130	54 TCP	[TCP Dup ACK 109#6] 41020 + 443 [ACK] Seq=726 Ack=92002 Win=65535 Len=0
192.168.126.130	101.6.15.130	54 TCP	[TCP Dup ACK 109#7] 41020 + 443 [ACK] Seq=726 Ack=92002 Win=65535 Len=0
192.168.126.130	101.6.15.130	54 TCP	[TCP Dup ACK 109#8] 41020 + 443 [ACK] Seq=726 Ack=92002 Win=65535 Len=0
192.168.126.130	101.6.15.130	54 TCP	[TCP Dup ACK 109#9] 41020 + 443 [ACK] Seq=726 Ack=92002 Win=65535 Len=0

由于 Duplicate Ack 的存在, 服务器端重传了相应的报文, 如图 10 所示。

图 10: 重传

101.6.15.130	192.168.126.130	1514 TCP	443 + 41020 [PSH, ACK] Seq=92002 Ack=726 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
101.6.15.130	192.168.126.130	1514 TCP	443 + 41020 [ACK] Seq=93462 Ack=726 Win=64240 Len=1460 [TCP segment of a reassembled PDU]

此时重传的报文还加入了 PSH 标志, 让客户端尽快交付。从图 7 中可以观察到, 虽然由于重传 *cwnd* 减少了, 但是由于快重传机制, *cwnd* 开始快速回复了, 进行了指数级的增长。

### 5.3 TCP 拥塞控制算法对比

#### 【BBR】

BBR 是由 Google 开发的拥塞控制算法, 旨在最大化网络吞吐量并减小传输时延。该算法基于网络路径中的带宽和往返时间 (RTT) 来确定拥塞窗口的大小。BBR 具有较好的带宽利用率和低延迟的优势, 在高速网络和高延迟网络中表现良好。

#### 【Reno】

Reno 是 TCP 的一种变种, 是 TCP 拥塞控制的经典算法之一。使用 AIMD 机制来调整拥塞窗口大小。Reno 主要关注拥塞的存在, 通过减小拥塞窗口来应对拥塞事件。

#### 【CUBIC】

CUBIC 是一种具有拟立方增长特性的 TCP 拥塞控制算法。相较于 Reno, CUBIC 在网络拥塞事件后更快地增加拥塞窗口大小。CUBIC 在高带宽网络中表现较好, 能够更快地适应网络的带宽变化。

**【对比】**

带宽利用率: BBR 通常能够更好地利用可用的带宽, 特别是在高速网络中。延迟: BBR 在一般情况下表现出较低的延迟, 而 CUBIC 在高带宽网络中也能保持相对较低的延迟。拥塞响应: CUBIC 相对于 Reno 来说, 对于网络拥塞的响应更为迅速, 而 BBR 通过更精确的带宽和 RTT 估计来调整拥塞窗口。适用场景: BBR 适用于高速、高延迟网络, CUBIC 适用于高速网络, 而 Reno 在一般网络环境中表现稳定。

## 6 UDP 协议观察

### 6.1 实验准备

在服务器端新建一个 server.py 文件, 内容如下

```
1 import socket
2 # 创建服务器 udp 套接字
3 udp_socket = socket.socket(type=socket.SOCK_DGRAM)
4 # 绑定服务器 ip 和 port
5 udp_socket.bind(('192.168.122.132', 9999))
6 # udp 协议无需建立连接, 直接接收消息 (这里为 1024 字节大小)
7 msg, addr = udp_socket.recvfrom(1024)
8 print(msg.decode('utf-8'))
9 # 给 client 发送消息
10 udp_socket.sendto('hello, I am Server:192.168.122.132'.encode('utf-8'), addr)
11 udp_socket.close()
```

在客户端新建一个 client.py 文件, 内容如下

```
1 import socket
2 # 创建服务器 udp 套接字
3 udp_socket = socket.socket(type=socket.SOCK_DGRAM)
4 # 绑定服务器 ip 和 port
5 udp_socket.bind(('192.168.122.132', 9999))
6 # udp 协议无需建立连接, 直接接收消息 (这里为 1024 字节大小)
7 msg, addr = udp_socket.recvfrom(1024)
8 print(msg.decode('utf-8'))
9 # 给 client 发送消息
10 udp_socket.sendto('hello, I am Server:192.168.122.132'.encode('utf-8'), addr)
11 udp_socket.close()
```

提前开好 wireshark 软件, 在服务器和客户机分别运行 python3 server/client.py

udp and ip.addr==192.168.122.132						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.2.15	192.168.122.132	UDP	73	54713 → 9999 Len=31
2	0.002361453	192.168.122.132	10.0.2.15	UDP	76	9999 → 54713 Len=34

> Frame 1: 73 bytes on wire (584 bits), 73 bytes captured (584 bit		0000	52 54 00 12 35 02 08 00	27 60 2d db 08 00
> Ethernet II, Src: PcsCompu_60:2d:db (08:00:27:60:2d:db), Dst: Re		0010	00 3b a7 e6 40 00 40 11	4b 90 0a 00 02 00
> Internet Protocol Version 4, Src: 10.0.2.15, Dst: 192.168.122.13		0020	7a 84 05 09 27 0f 00 27	47 74 68 69 2c 21
✓ User Datagram Protocol, Src Port: 54713, Dst Port: 9999		0030	61 6d 20 43 6c 69 65 6e	74 3a 31 39 32 21
Source Port: 54713		0040	38 2e 31 32 32 2e 31 33	32
Destination Port: 9999				
Length: 39				
Checksum: 0x4774 [unverified]				
[Checksum Status: Unverified]				
[Stream index: 0]				
> [Timestamps]				
UDP payload (31 bytes)				

udp and ip.addr==192.168.122.132						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.2.15	192.168.122.132	UDP	73	54713 → 9999 Len=31
2	0.002361453	192.168.122.132	10.0.2.15	UDP	76	9999 → 54713 Len=34

> Frame 2: 76 bytes on wire (608 bits), 76 bytes captured (608 bit		0000	08 00 27 60 2d db 52 54	00 12 35 02 08 00
> Ethernet II, Src: RealtekU_12:35:02 (52:54:00:12:35:02), Dst: Pc		0010	00 3e 0f b1 00 00 40 11	23 c3 c0 a8 7a 84
> Internet Protocol Version 4, Src: 192.168.122.132, Dst: 10.0.2.1		0020	02 0f 27 0f d5 b9 00 2a	04 eb 68 65 6c 6c
✓ User Datagram Protocol, Src Port: 9999, Dst Port: 54713		0030	20 49 20 61 6d 20 53 65	72 76 65 72 3a 31
Source Port: 9999		0040	2e 31 36 38 2e 31 32 32	2e 31 33 32
Destination Port: 54713				
Length: 42				
Checksum: 0x04eb [unverified]				
[Checksum Status: Unverified]				
[Stream index: 0]				
> [Timestamps]				
UDP payload (34 bytes)				

- Source Port : 54713 源端口。这是发送方的端口号，表示数据报文的来源。
- Destination Port: 9999 目的端口。这是接收方的端口号，表示数据报文的目的地。
- Length : 39 长度。表示 UDP 报文的总长度，包括 UDP 头和 UDP 数据。在这里，总长度为 39 字节。
- Checksum: 0x4774 [unverified] 校验和。校验和用于检测 UDP 头和数据的错误。0x4774 是校验和的十六进制值。”unverified” 表示校验和状态尚未验证。
- Checksum Status: Unverified 校验和状态。表示校验和状态未经验证。在正常情况下，接收方将验证校验和，以确保数据的完整性。
- Stream Index : 0 流索引。表示流的索引。在这里，索引为 0。
- Timestamps : 时间戳。未提供具体的时间戳数值，但这是 UDP 报文可能包含的可选字段之一。时间戳通常用于测量数据报文的往返时间 (RTT) 等信息。
- UDP Payload : 31 字节 UDP 负载。这是 UDP 数据的实际内容，长度为 31 字节。具体内容的解释需要根据应用层协议或应用程序的协议进行解析。

## 7 HTTP 协议分析

### 7.1 实验准备

http.server 库提供直接使用命令行的方式启动 HTTP 服务器，但是早期版本不方便切换协议，因此先把 python 版本升级到 3.11，如图 11 所示。

这样就可以用 --protocol 参数直接指定协议了，方便了后续的实验。



图 11: 升级 python 版本

By default, the server is conformant to HTTP/1.0. The option `-p/--protocol` specifies the HTTP version to which the server is conformant. For example, the following command runs an HTTP/1.1 conformant server:

```
python -m http.server --protocol HTTP/1.1
```

New in version 3.11: `--protocol` argument was introduced.

## 7.2 实验结果

实验结果如图 12 所示。图 12a 显示了 HTTP1.0 多次请求的结果，由于 HTTP1.0 不支持持久连接，因此每次请求都需要重新建立连接。可以看到，每次请求都先包含了 TCP 的三次握手。在请求结束后都包含了 TCP 的四次挥手。这样多次建立连接，既浪费了几个 RTT 的时间，又浪费了服务器端的资源。

图 12: 不同版本的 HTTP

Length: 200	
74 TCP	3920 → 8000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=627769328 TSecr=0 WS=128
78 TCP	8000 → 39200 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=627769328 TSecr=627769328 WS=128
66 TCP	39200 → 8000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=627769328 TSecr=627769328
568 HTTP	GET /test.txt HTTP/1.1
66 TCP	8000 → 39200 [ACK] Seq=1 Ack=593 Win=65024 Len=0 TSval=627769328 TSecr=627769328
173 HTTP	HTTP/1.0 304 Not Modified
66 TCP	39200 → 8000 [ACK] Seq=563 Ack=108 Win=65536 Len=0 TSval=627769332 TSecr=627769332
66 TCP	8000 → 39200 [FIN, ACK] Seq=108 Ack=563 Win=65536 Len=0 TSval=627769332 TSecr=627769332
66 TCP	39200 → 8000 [FIN, ACK] Seq=568 Ack=109 Win=65536 Len=0 TSval=627769332 TSecr=627769332
66 TCP	8000 → 39200 [ACK] Seq=109 Ack=568 Win=65536 Len=0 TSval=627769332 TSecr=627769332
74 TCP	54022 → 8000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=627771739 TSecr=0 WS=128
78 TCP	8000 → 54022 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=627771739 TSecr=627771739 WS=128
66 TCP	54022 → 8000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=627771739 TSecr=627771739
568 HTTP	GET /test.txt HTTP/1.1
66 TCP	8000 → 54022 [ACK] Seq=1 Ack=593 Win=65024 Len=0 TSval=627771739 TSecr=627771739
173 HTTP	HTTP/1.0 304 Not Modified
66 TCP	54022 → 8000 [ACK] Seq=593 Ack=108 Win=65536 Len=0 TSval=627771740 TSecr=627771740
66 TCP	8000 → 54022 [FIN, ACK] Seq=108 Ack=593 Win=65536 Len=0 TSval=627771741 TSecr=627771741
66 TCP	54022 → 8000 [FIN, ACK] Seq=593 Ack=109 Win=65536 Len=0 TSval=627771741 TSecr=627771741

74 TCP	48794 → 8000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=627779756 TSecr=0 WS=128
78 TCP	8000 → 48794 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=627779756 TSecr=627779756 WS=128
66 TCP	48794 → 8000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=627779756 TSecr=627779756
10 HTTP	GET / HTTP/1.1
66 TCP	8000 → 48794 [ACK] Seq=1 Ack=445 Win=65152 Len=0 TSval=627780182 TSecr=627780182
58 TCP	8000 → 48794 [PSH, ACK] Seq=1 Ack=445 Win=65536 Len=158 TSval=627780183 TSecr=627780183 [TCP segment of a reassembled PSH]
66 TCP	48794 → 8000 [ACK] Seq=445 Ack=159 Win=65488 Len=0 TSval=627780183 TSecr=627780183
10 HTTP	HTTP/1.0 200 OK (text/html)
66 TCP	48794 → 8000 [ACK] Seq=445 Ack=471 Win=65152 Len=0 TSval=627780183 TSecr=627780183
66 TCP	GET /test.txt HTTP/1.1
66 TCP	HTTP/1.1 304 Not Modified
66 TCP	48794 → 8000 [ACK] Seq=987 Ack=578 Win=65536 Len=0 TSval=627802696 TSecr=627802696
66 HTTP	GET /test.txt HTTP/1.1
73 HTTP	HTTP/1.1 304 Not Modified
66 TCP	48794 → 8000 [ACK] Seq=1529 Ack=485 Win=65536 Len=0 TSval=627807624 TSecr=627807624
66 TCP	[TCP Keep-Alive] 48794 → 8000 [ACK] Seq=1528 Ack=485 Win=65536 Len=0 TSval=627817148 TSecr=627807624
66 TCP	[TCP Keep-Alive ACK] 8000 → 48794 [ACK] Seq=485 Ack=1529 Win=65536 Len=0 TSval=627817148 TSecr=627807624
66 TCP	[TCP Keep-Alive] 48794 → 8000 [ACK] Seq=1528 Ack=485 Win=65536 Len=0 TSval=627817148 TSecr=627817148
66 TCP	[TCP Keep-Alive ACK] 8000 → 48794 [ACK] Seq=485 Ack=1529 Win=65536 Len=0 TSval=627823388 TSecr=627807624

(a) HTTP1.0

(b) HTTP1.1

74 TCP	42014 → 8000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=627841533 TSecr=0 WS=128
78 TCP	8000 → 42014 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=627841533 TSecr=627841533
66 TCP	42014 → 8000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=627841533 TSecr=627841533
538 HTTP	GET / HTTP/1.1
66 TCP	8000 → 42014 [ACK] Seq=1 Ack=445 Win=65152 Len=0 TSval=627841670 TSecr=627841670
224 TCP	8000 → 42014 [PSH, ACK] Seq=1 Ack=445 Win=65536 Len=158 TSval=627841671 TSecr=627841670 [TCP segment of a reassembled PSH]
66 TCP	42014 → 8000 [ACK] Seq=445 Ack=159 Win=65488 Len=0 TSval=627841671 TSecr=627841671
378 HTTP	HTTP/2.0 200 OK (text/html)
66 TCP	42014 → 8000 [ACK] Seq=445 Ack=471 Win=65152 Len=0 TSval=627841671 TSecr=627841671
588 HTTP	GET /test.txt HTTP/1.1
173 HTTP	HTTP/2.0 304 Not Modified
66 TCP	42014 → 8000 [ACK] Seq=987 Ack=578 Win=65536 Len=0 TSval=62784568 TSecr=62784568
588 HTTP	GET /test.txt HTTP/1.1
173 HTTP	HTTP/2.0 304 Not Modified
66 TCP	42014 → 8000 [ACK] Seq=1529 Ack=685 Win=65536 Len=0 TSval=627847582 TSecr=627847582
588 HTTP	GET /test.txt HTTP/1.1
173 HTTP	HTTP/2.0 304 Not Modified
66 TCP	42014 → 8000 [ACK] Seq=2071 Ack=792 Win=65536 Len=0 TSval=627848374 TSecr=627848374
588 HTTP	GET /test.txt HTTP/1.1
173 HTTP	HTTP/2.0 304 Not Modified

(c) HTTP2.0

图 12b 显示了 HTTP1.1 多次请求的结果。由于 HTTP1.1 支持持久连接，因此只有一开始有三次握手，后面的请求都是直接使用相同的 TCP 连接发送数据。这样就大大减少了开销。为了让服务器判断是否需要继续保持连接，HTTP1.1 也会发送 Keep-Alive 的请求头，让服务器判断是否需要保持连接。可以看到发送的信息中也有些 TCP Keep-Alive 的信息，这些信息是用来维持 TCP 连接的。

图 12c 显示了 HTTP2 多次请求的结果。由于 HTTP2 是基于 TCP 的，因此也是需要三次握手。HTTP2.0 的多路复用机制，可以让多个请求共享一个 TCP 连接，因此可以看到所有请求都使用了客户机 42014 端口打开的 TCP 连接。这大大节省了服务器的资源，也减少了开销。

## 8 实验总结

本次实验进一步的帮助我们理解了 TCP 首部、TCP 连接原理、超时重传时间的规律、SYN 洪泛攻击的原理等。