传输层 TCP 协议实验报告

无04 2019012137 张鸿琳

实验目的

- 理解和掌握 TCP 连接管理中“三次握手”建立连接和拆除连接的过程；
- 理解和掌握 TCP 可靠数据传输的实现原理和方法；
- 理解和掌握 TCP 流量控制的实现原理和方法；
- 理解和掌握 TCP 拥塞控制的实现原理和方法；
- 学习和掌握通过编程和抓包分析工具验证和分析协议运行过程。

实验内容

- 理解实验原理，熟悉 Mininet 网络仿真工具和 Wireshark 抓包分析工具；
- 阅读和运行实验代码，理解和掌握创建网络拓扑、建立 TCP 连接的过程，并抓取 TCP 连接过程中的数据包；
- 基于网络仿真工具和抓包分析工具，查找 TCP 连接管理、可靠传输相关的数据包，结合 TCP 协议原理进行验证和分析；
- 基于网络仿真工具和抓包分析工具，测量 TCP 流量控制和拥塞控制中关键性能指标的变化情况，结合 TCP 协议原理进行验证和分析。

实验过程与结果

5.1 网络仿真环境运行和实验网络搭建

运行 5.1 部分代码，输出如下：

```
构建网络中包含的节点:
['h1', 'h2', 's0']
构建网络中包含的链路:
第1条链路:
('h2', 's0')
第1条链路信息:
{'delay': '10ms', 'bw': 1.5, 'max_queue_size': 10, 'node1': 'h2', 'node2': 's0', 'port2': 2, 'port1': 0}
第2条链路:
('h1', 's0')
第2条链路信息:
{'delay': '10ms', 'bw': 1000, 'max_queue_size': 10, 'node1': 'h1', 'node2': 's0', 'port2': 1, 'port1': 0}
h1 h1-eth0:s0-eth1
h2 h2-eth0:s0-eth2
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)

test all pair pings
packet loss percentage: 0.000000
('h1', '10.0.0.1')
('h2', '10.0.0.2')
```

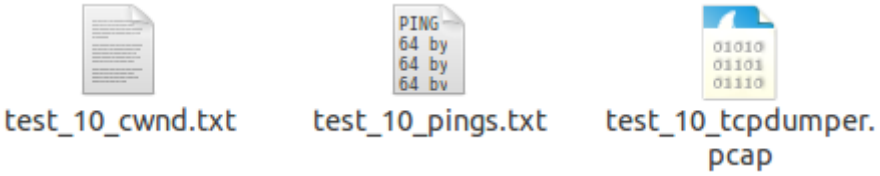
可以看到成功搭建了实验网络，输出表明了 h1 和 h2 的 ip 地址，该网络测试中丢包率为 0 。

5.2 TCP 流量产生和数据包抓取

运行 5.2 部分代码，得到输出如下：

```
Starting iperf server
Starting iperf client
Starting ping...
simulation finished
```

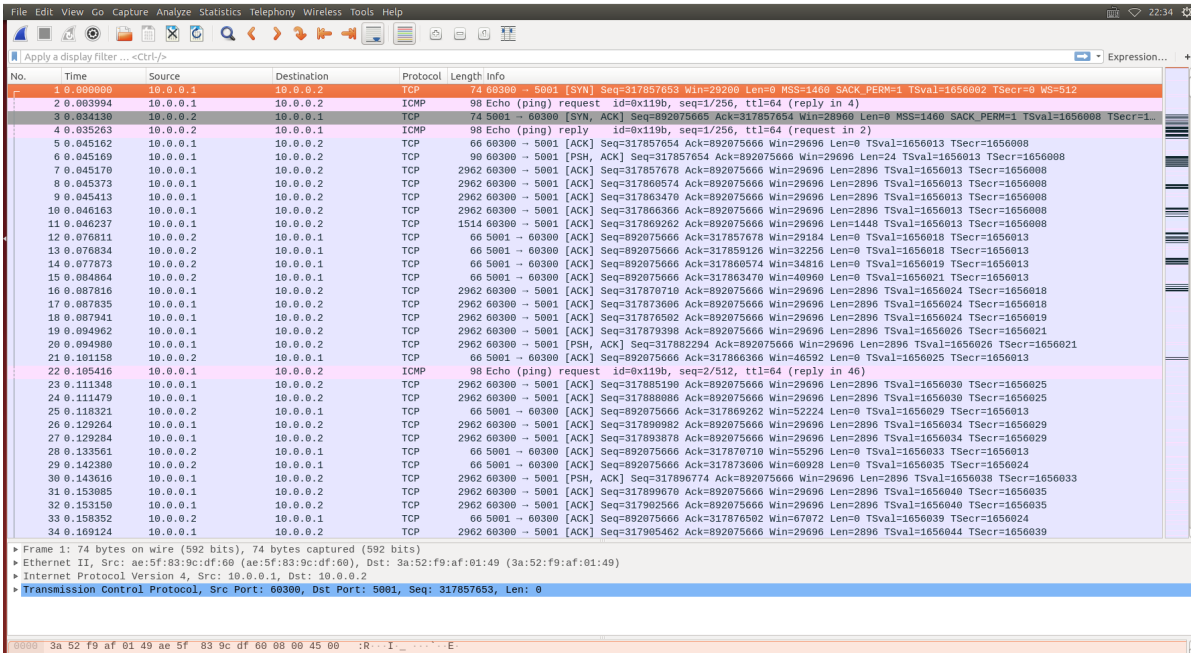
成功在 h2 主机上建立了服务端，在 h1 主机上建立了客户端，且成功进行了样本 RTT 的测试。运行代码后得到的输出文件如下：



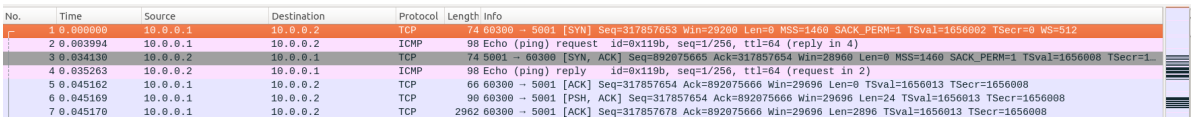
说明成功得到了抓包数据、估计 RTT 和拥塞窗口等指标数据、样本 RTT 数据。

5.3 TCP 连接管理

利用 Wireshark 打开 test_10_tcpdumper.pcap 文件，如下图：



通过 tcp.flags.syn==1 的筛选条件，找到 TCP 连接建立时的三次握手，对应于下图中的第 1 、3 、5 个数据包：



整理数据包并填写下面三个表：

- h1 发送的报文段：

源端口号：60300								目的端口号：5001
序号：317857653								
确认号：								
首部长度	保留位用	URG..	ACK..	PSH..	PST..	SYN..1	FIN..	

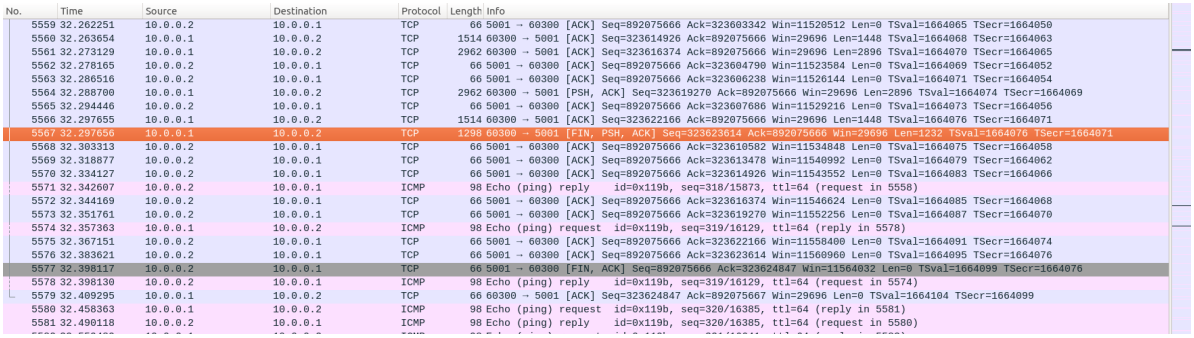
- h2 发送的报文段：

源端口号：5001								目的端口号：60300
序号：892075665								
确认号：317857654								
首部长度	保留位用	URG..	ACK..1	PSH..	PST..	SYN..1	FIN..	

- h1 发送的报文段：

源端口号：60300								目的端口号：5001
序号：317857654								
确认号：892075666								
首部长度	保留位用	URG..	ACK..1	PSH..	PST..	SYN..	FIN..	

再利用 tcp.flags.fin==1 的筛选条件，找到 TCP 连接终止过程中发送的三（四）个数据包，对应于下图中的第 5567 、5577 、5579 个数据包：



No.	Time	Source	Destination	Protocol	Length	Info
5559	32.262251	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323603342 Win=11520512 Len=0 TSval=1664065 TSecr=1664050
5560	32.263654	10.0.0.1	10.0.0.2	TCP	1514	60300 → 5001 [ACK] Seq=323614926 Ack=892075666 Win=29696 Len=1448 TSval=1664068 TSecr=1664063
5561	32.273129	10.0.0.1	10.0.0.2	TCP	2962	60300 → 5001 [ACK] Seq=323616374 Ack=892075666 Win=29696 Len=2896 TSval=1664070 TSecr=1664065
5562	32.278165	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323604790 Win=11523584 Len=0 TSval=1664069 TSecr=1664052
5563	32.286516	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323606238 Win=11526144 Len=0 TSval=1664071 TSecr=1664054
5564	32.288700	10.0.0.1	10.0.0.2	TCP	2962	60300 → 5001 [PSH, ACK] Seq=323619270 Ack=892075666 Win=29696 Len=2896 TSval=1664074 TSecr=1664069
5565	32.294446	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323607686 Win=11529216 Len=0 TSval=1664073 TSecr=1664056
5566	32.297655	10.0.0.1	10.0.0.2	TCP	1514	60300 → 5001 [ACK] Seq=323622166 Ack=892075666 Win=29696 Len=1448 TSval=1664076 TSecr=1664071
5567	32.297656	10.0.0.1	10.0.0.2	TCP	1298	60300 → 5001 [FIN, PSH, ACK] Seq=323623614 Ack=892075666 Win=29696 Len=1232 TSval=1664076 TSecr=1664071
5568	32.303313	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323610582 Win=11534848 Len=0 TSval=1664075 TSecr=1664058
5569	32.318877	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323613478 Win=11540992 Len=0 TSval=1664079 TSecr=1664062
5570	32.334127	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323614926 Win=11543552 Len=0 TSval=1664083 TSecr=1664066
5571	32.342607	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x119b, seq=318/15873, ttl=64 (request in 5558)
5572	32.344169	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323616374 Win=11546624 Len=0 TSval=1664085 TSecr=1664068
5573	32.351761	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323619270 Win=11552256 Len=0 TSval=1664087 TSecr=1664070
5574	32.357363	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x119b, seq=319/16129, ttl=64 (reply in 5578)
5575	32.367151	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323622166 Win=11558400 Len=0 TSval=1664091 TSecr=1664074
5576	32.383621	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [ACK] Seq=892075666 Ack=323623614 Win=11560960 Len=0 TSval=1664095 TSecr=1664076
5577	32.398117	10.0.0.2	10.0.0.1	TCP	66	5001 → 60300 [FIN, ACK] Seq=892075666 Ack=323624847 Win=11564032 Len=0 TSval=1664099 TSecr=1664076
5578	32.398130	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x119b, seq=319/16129, ttl=64 (request in 5574)
5579	32.409295	10.0.0.1	10.0.0.2	TCP	66	60300 → 5001 [ACK] Seq=323624847 Ack=892075667 Win=29696 Len=0 TSval=1664104 TSecr=1664099
5580	32.458363	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x119b, seq=320/16385, ttl=64 (reply in 5581)
5581	32.490118	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x119b, seq=320/16385, ttl=64 (request in 5580)

整理数据包并填写下面三个表：

- h1 发送的报文段：

源端口号：60300								目的端口号：5001
序号：323623614								
确认号：892075666								
首部长度	保留位用	URG..	ACK..1	PSH..1	PST..	SYN..	FIN..1	

- h2 发送的报文段：

源端口号：5001								目的端口号：60300
序号：892075666								
确认号：323624847								
首部长度	保留位用	URG..	ACK..1	PSH..	PST..	SYN..	FIN..1	

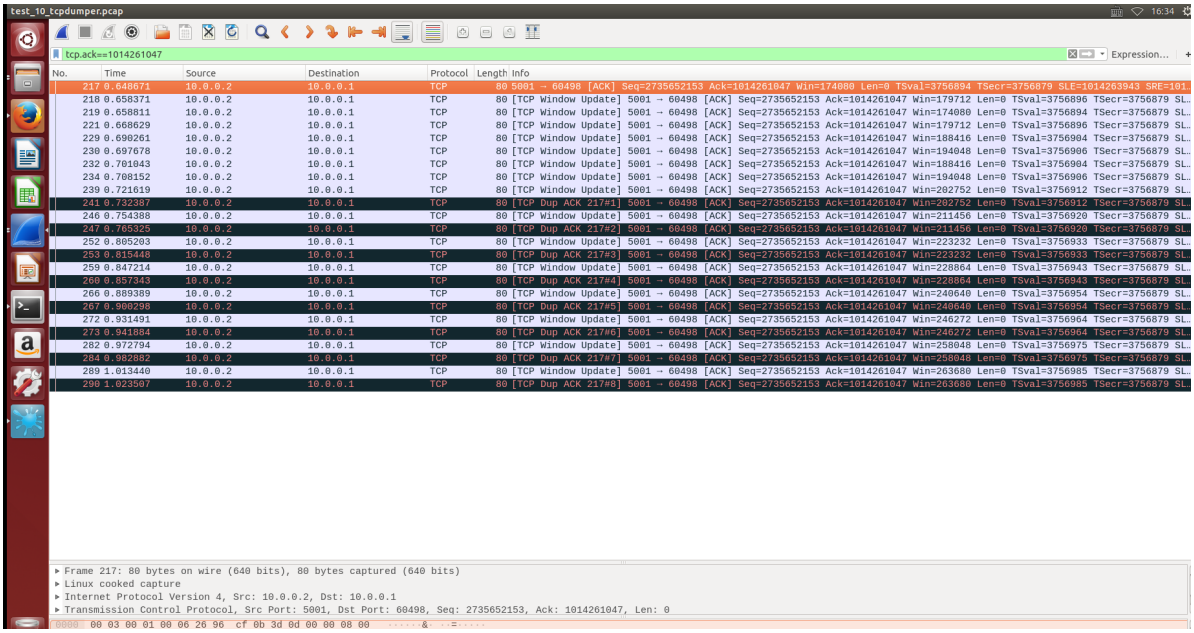
- h1 发送的报文段：

源端口号：60300								目的端口号：5001
序号：323624847								
确认号：892075667								
首部长度	保留位用	URG..	ACK..1	PSH..	PST..	SYN..	FIN..	

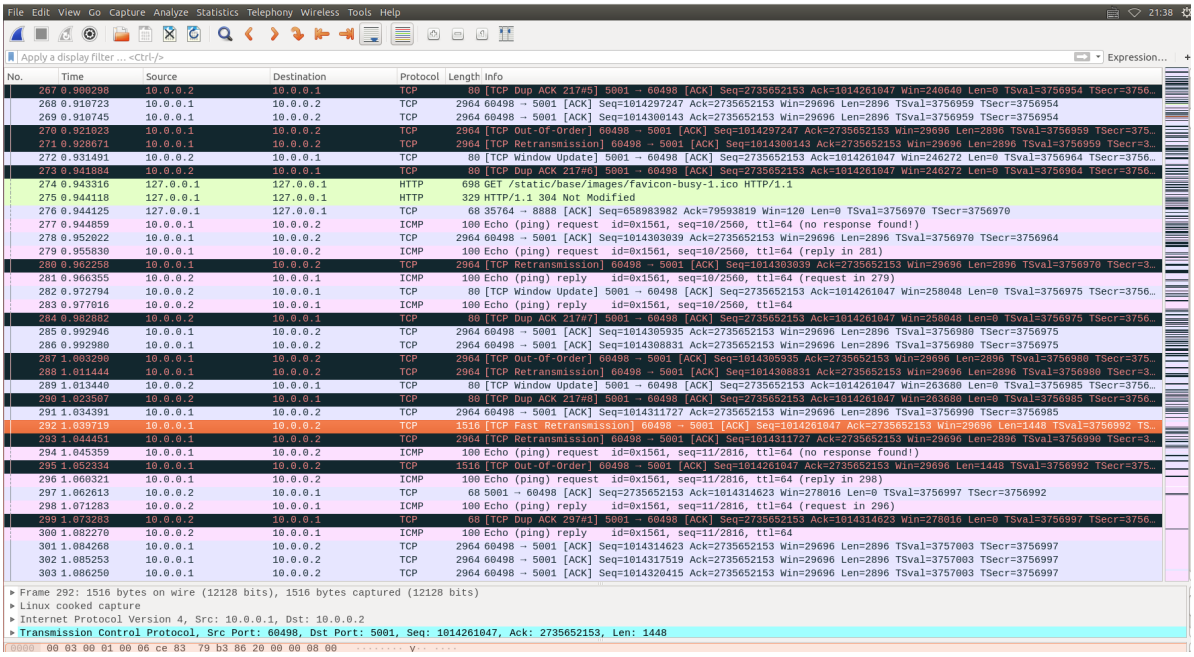
5.4 TCP 可靠传输

先根据实验答疑和补充中的操作，提高丢包率，更改原有 tcpdump 命令，并将实验时间增加为 60 s，然后再进行抓包测试。

通过 tcp.analysis.fast_retransmission 来检索出一些 h1 向 h2 的快速重传，确定重传的序号后，再据之利用 tcp.ack 找到 h2 向 h1 发送的重复 ACK 包，一个例子（数据包信息）如下：

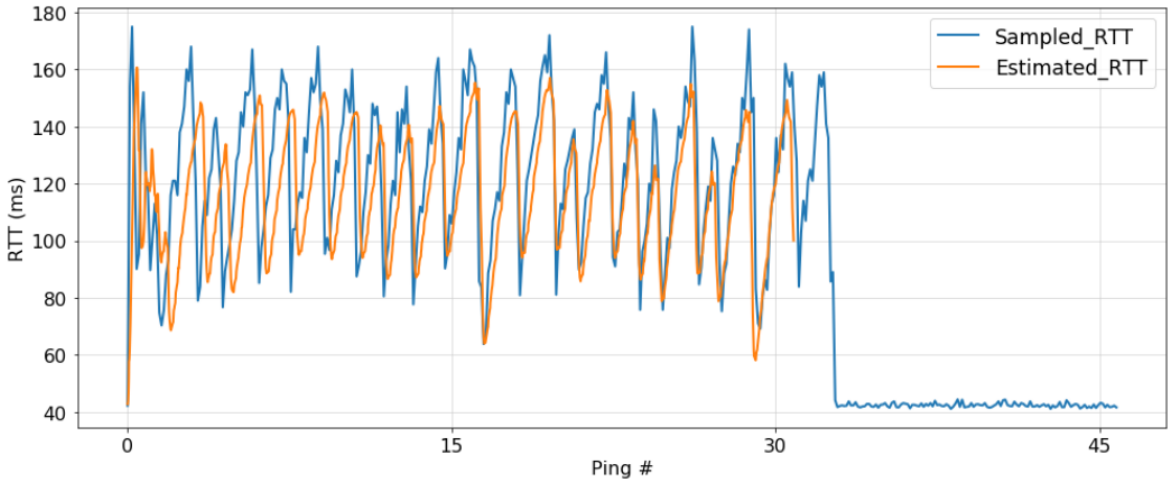


这些重复的 ACK 数据包中的确认号都为 1014261047 ，再去除筛选条件，看 h1 向 h2 的重传，如下图：

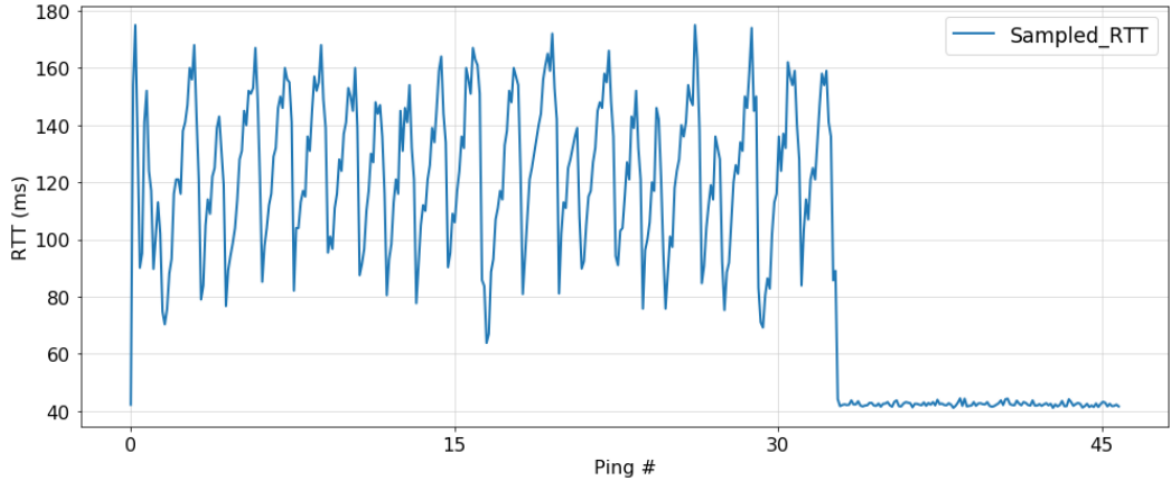


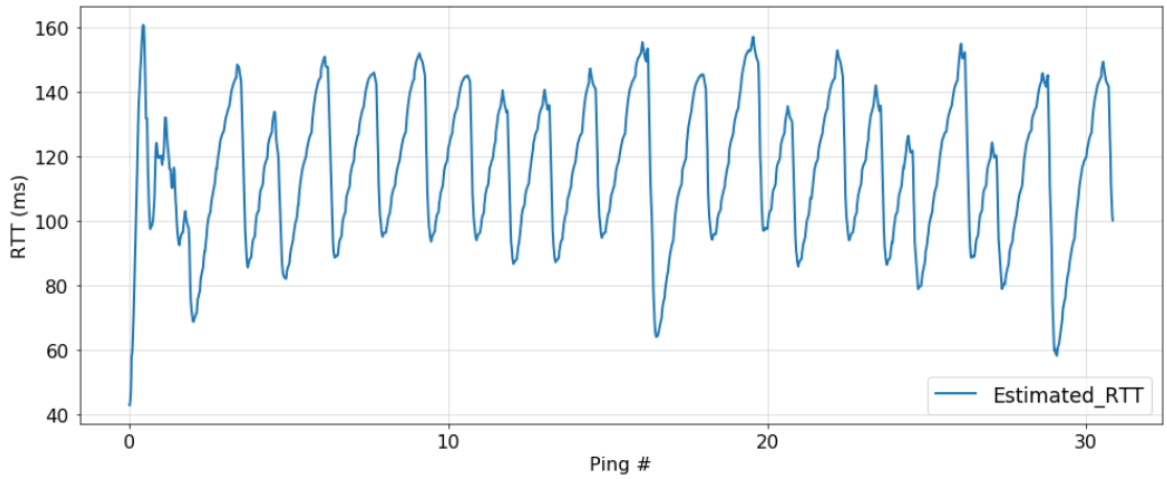
上图中第 292 个数据包为 h1 向 h2 发送的序号为 1014261047 的重传，可以看到重传完毕后， h2 不再重复发送确认号为 1014261047 的 ACK 数据包，而是（第 297 个数据包）开始发送确认号为 1014314623 的 ACK 数据包，证明 h2 成功接收到了 h1 重传的数据包，开始等待新的数据包了，**因此重传原因是序号为 1014261047 的数据包在传输过程中发生丢包。**而且还可以进一步确认的是，在 h1 重传序号为 1014261047 的数据包前，其已经发出的最大序号的数据包为上图中第 291 个数据包，其序号为 1014311727 ，数据包长度为 2896 ， 1014311727 + 2896 = 1014314623 ，这恰是此前提到的 h2 接收到 h1 的重传数据包后的确认号，由此可知在（发生丢包的）序号为 1014261047 的数据包之后发出的数据包均被 h2 正常接收了。

此后，利用代码绘制采样 RTT 和估计 RTT 的曲线（我对原代码做了一些修改，删去了对估计 RTT 的时间刻度的向左平移，以免在 -1 横坐标处就出现估计 RTT 值，事实上在平移之后确实让两个波形在后期更匹配了，但是造成这样的不匹配的原因显然是采样 RTT 对应的时间刻度是序号乘以周期直接得到的，默认的周期可能不准确或者实际周期随时间有变化，所以得到的时间刻度自然也不准确，因此为了让两种波形更加匹配，我将采样 RTT 的时间刻度乘上了 1.05 ，看起来在后期的匹配效果还不错），得到下图：



为了更好地观察细节，可以单独绘制两张波形图如下：





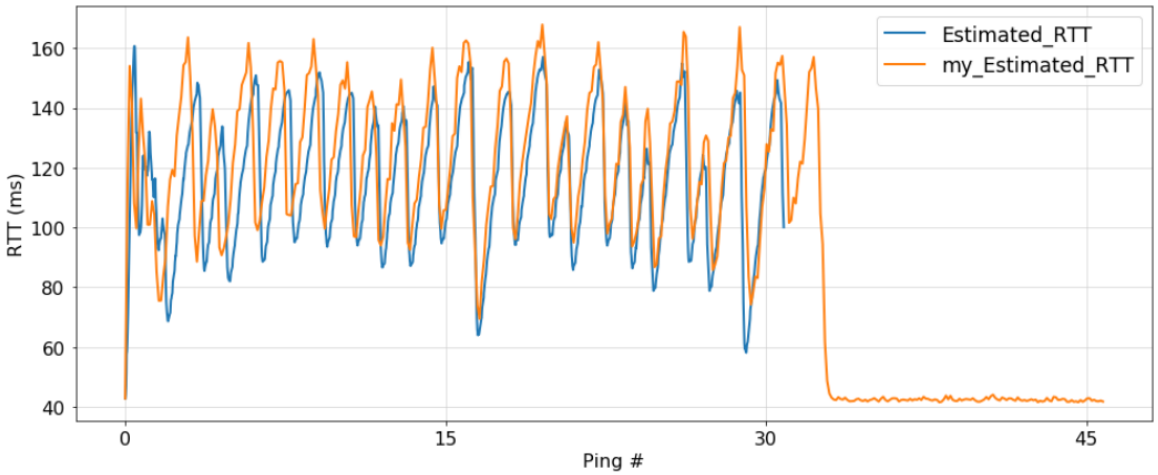
比较容易发现，相较于采样 RTT ，估计 RTT 整体上比较缓和，没有那么多小尖刺，这是因为估计 RTT 采用了指数加权移动平均算法，即 $EstimatedRTT = (1-\alpha) \times EstimatedRTT + \alpha \times SampleRTT$，在$\alpha$选取合理的情况下，当前估计 RTT 值对小波动不敏感，因而估计 RTT 曲线更为平滑。

【选做】自行改写代码，基于采样 RTT 计算估计 RTT 并绘制图像，如下（对采样 RTT 的时间尺度乘上了 1.05 来抵消一点误差，同时令估计 RTT 计算公式中的 α 取 0.65 ）：

```
import matplotlib.pyplot as plt
from plot_ping import parse_ping

def ertt_get(fname):
    ertt = []
```

```
    tt = []
    with open('./'+fname+'_cwnd.txt', 'r') as f:
        for i, line in enumerate(f.readlines()):
        xx = line.strip().split(' ')
        if i == 0:
            tt_0 = float(xx[0])
        if '10.0.0.1' in xx[1]:
            tt.append((float(xx[0])-tt_0-0.0))
            ertt.append(abs(float(xx[-2]))/1e3)
    return tt, ertt

from pylab import figure
import plot_defaults
from matplotlib.ticker import MaxNLocator
from helper import *

def plot_srtt_ertt(fname):
    fig = figure(figsize=(16, 6))
    ax = fig.add_subplot(111)
    f = "{}_pings.txt".format(fname)
    data = parse_ping(f)
    xaxis = map(float, col(0, data))
    freq=10
    start_time = xaxis[0]
    xaxis = map(lambda x: 1.05*(x - start_time) / freq, xaxis)
    qlens = map(float, col(1, data))

    #ax.plot(xaxis, qlens, lw=2, label='Sampled_RTT')
    ax.xaxis.set_major_locator(MaxNLocator(4))

    plt.ylabel("RTT (ms)")
    plt.xlabel("Ping #")
    plt.grid(True)
    tt, ertt = ertt_get(fname)
    plt.plot(tt, ertt, '-', label='Estimated_RTT')

    initial = ertt[0]
    tt2, ertt2 = my_ertt_get(initial, data, start_time, freq)
    plt.plot(tt2, ertt2, '-', label='my_Estimated_RTT')

    plt.legend()
    plt.show()


def my_ertt_get(initial, data, start_time, freq):
    a = 0.65
    b = 1.05
    xaxis = map(float, col(0, data))
    tt = map(lambda x: b*(x - start_time) / freq, xaxis)
    ertt = map(float, col(1, data))
    for i in range(len(ertt)):
        if i == 0:
            ertt[0]=initial
            continue
        ertt[i] = (1-a)*ertt[i-1]+a*ertt[i]
    return tt, ertt
```

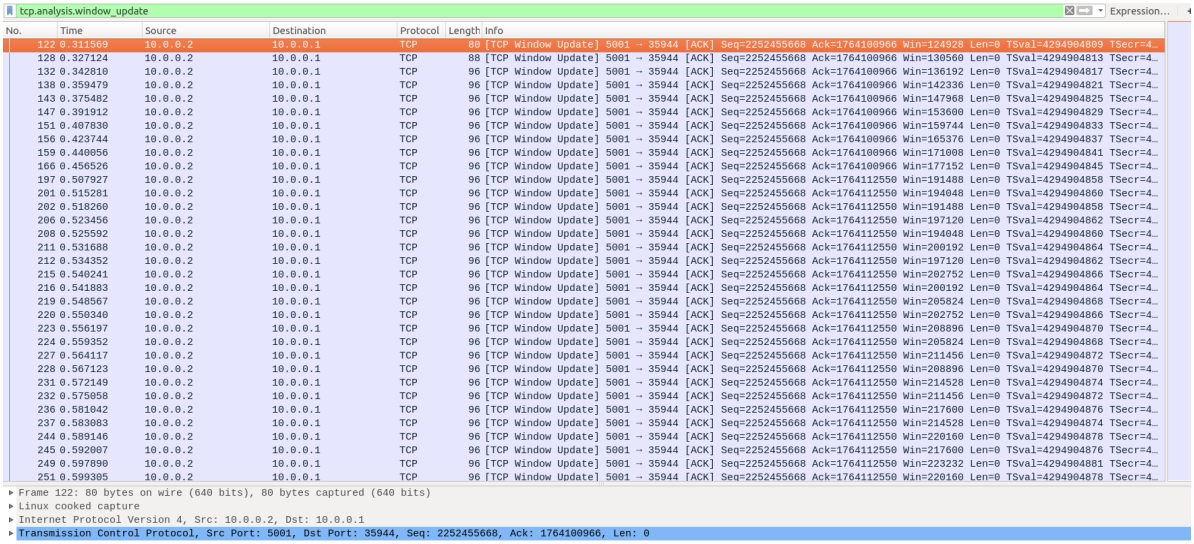
得到的图像如下：



可以看到在α取 0.65 时，我基于采样 RTT 自行计算的估计 RTT 已经和 tcpprobe 得到的估计 RTT 十分吻合了（在前期有些错位应该还是采样 RTT 的周期存在误差造成的），只是自行计算的估计 RTT 相较于实际估计 RTT 偏高一点，这或许是因为实际估计 RTT 在计算时采用的采样 RTT 数据和基于 ping 的采样 RTT 数据有一定差别，实际α值也应该和 0.65 略有差异。

5.5 TCP 流量控制

再运行一遍代码，利用 Wireshark 软件打开 test_10_tcpdumper.pcap 文件，并利用筛选条件 tcp.analysis.window_update ，得到接收窗口变化数据如下：



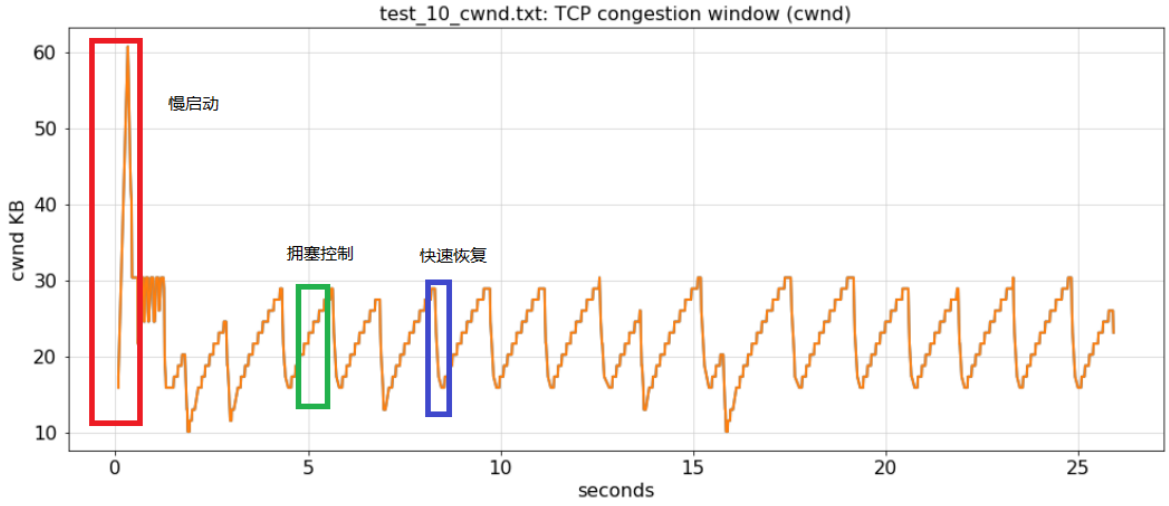
对上面数据进行梳理，填写下表：

时间	0.311569	0.327124	0.342810	0.359479	0.375482	0.391912
接收窗口	124928	130560	136192	142336	147968	153600
时间	0.407830	0.423744	0.440056	0.456526	0.507927	0.515281
接收窗口	159744	165376	171008	177151	191488	194048
时间	0.518260	0.523456	0.525592	0.531688	0.534352	0.540241
接收窗口	191488	197120	194048	200192	197120	202752

（事实上，在不提示 [TCP Window Update] 时，也会出现接收窗口 Win 的变化，不清楚原因）可以看到在上面抽取这一段时间内，接收窗口基本一直在增大，这是因为 iperf 不对接收到的包进行复杂处理，瓶颈较小，因而窗口基本在不断增大，而偶有接收窗口减小现象的发生，则可能是发送端发送数据包较为频繁，短时间发送数据的速率超过了接收端处理数据的速率，产生了短时间的缓存中的 TCP 数据的堆积，使得接收窗口大小略有下降。

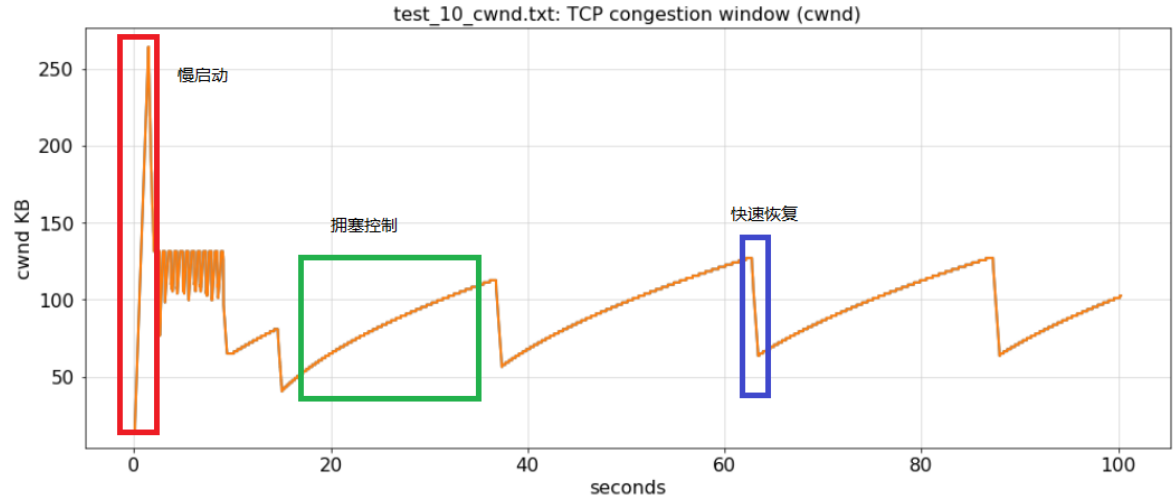
5.6 TCP 拥塞控制

运行 5.6 部分代码（路由器缓存大小为 10 ），得到下图：

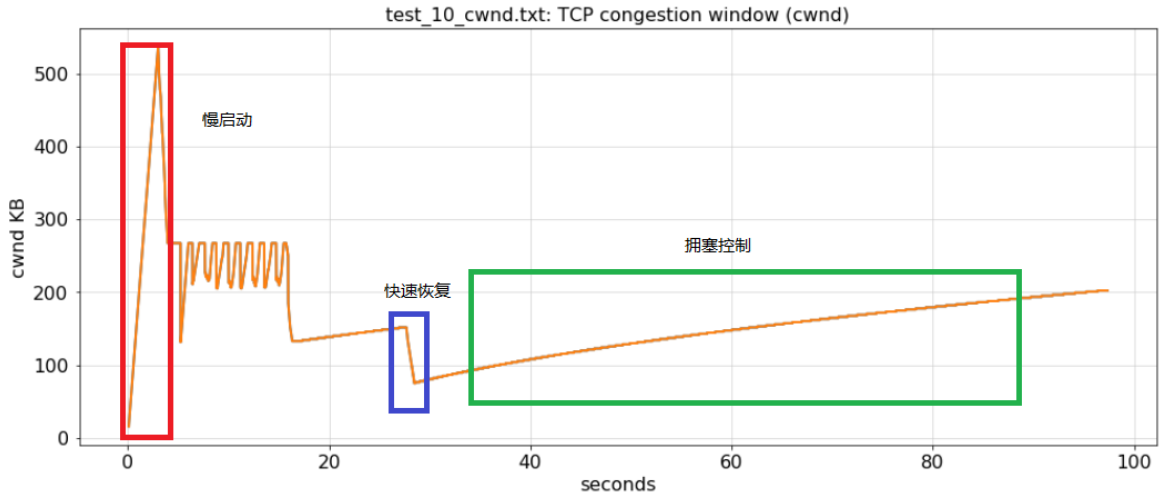


图像中最左侧快速上升的部分为慢启动阶段，在后面稳定后，各个锯齿中近似线性上升的部分都为拥塞避免阶段，而每个锯齿的下降部分及其附近都为快速恢复阶段。

在将路由器缓存大小改为 50 后，运行代码，得到下图（时间范围改为 100 ）：



在将路由器缓存大小改为 100 后，运行代码，得到下图（时间范围改为 100 ）：



路由器缓存大小为 50 、 100 时，其慢启动都为最左侧快速上升的部分，而在后续稳定后，线性上升的部分都为拥塞避免阶段，而每个锯齿的下降部分及其附近都为快速恢复阶段。（在慢启动后面的震荡部分或许也是属于快速恢复阶段）

通过三个图像的对比可以发现，随着路由器缓存大小的增大，平均拥塞窗口大小增大了，因为路由器缓存增大，链路变得越来越宽松，不容易发生拥塞。此外随着路由器缓存大小的增大，拥塞避免阶段、慢启动阶段的上升越发平缓，这是因为路由器缓存增大后，可暂存的数据包变多，但是这也使得往返时间 RTT 变大（在路由器中排队时延增加了），而慢启动阶段拥塞窗口大小是随确认到达而增大的，拥塞避免阶段拥塞窗口大小是按照每过一个 RTT 时间增加一个 MSS 的方式增大的，这两种增长方式都明显受 RTT 影响， RTT 增大自然会导致拥塞窗口增长放缓。

【选做】经测试，当路由器缓存大小为 500 时，最终拥塞窗口大小稳定在 463 左右， RTT 的绝对大小约为 3747 ms ，当路由器缓存大小为 1000 时，最终拥塞窗口大小稳定在 911 左右， RTT 的绝对大小约为 7385 ms 。而路由器缓存大小为 10 时，拥塞窗口大小在 25 左右，RTT 的绝对大小在 120 ms 左右；路由器缓存大小为 50 时，拥塞窗口大小在 100 左右，RTT 的绝对大小在 500 ms 左右；路由器缓存大小为 100 时，拥塞窗口大小在 180 左右，RTT 的绝对大小在 1000 ms 左右。

通过对比可以发现当路由器缓存越来越大，平均拥塞窗口也越来越大，这是因为路由器缓存变大后，丢包现象随之变少（丢包往往发生在缓存充满时），也就是更不容易发生拥塞了，故而信息发送方会认为链路十分宽松，就会不断增大拥塞窗口，使之维持在一个更高的水平，但是路由器处理数据包的能力是有限的，平均拥塞窗口增大后，短时间发送的数据包会变多，在路由器缓存中排队的长度也会变长，因而各个数据包的平均排队时延明显增加，故而往返时间 RTT 显著增大了。

实验思考题

(1) 本实验中，基于 Wireshark 和抓到的数据包，分析 h1 的接收窗口变化情况，请解释产生这样现象的原因。

h1 的接收窗口大小不变，因为接收窗口是根据接收端缓存器空间大小和接收到的数据包动态调整的，当接收到的数据包堆积在缓存中，则接收窗口变小，若缓存中的数据包不断被处理，则接收窗口变大。对于 h1 ，其接收到的数据包来自 h2 ，但是 h2 作为服务端没有向 h1 发送过有实际数据的数据包（基本上都是 ACK 数据包，这种数据包所含数据长度为 0 ，应该基本不占用缓存），因而也就不存在缓存空闲空间的变化，其接收窗口也就不变。

(2) 本实验利用 Wireshark 分析抓包过程中，除了本实验重点分析的 TCP 协议数据包，还存在哪些其他类型数据包？通过进一步 Baidu 或查阅资料确定这些数据包对应于网络哪一层。

还存在如下类型的数据包：

- OpenFlow：数据链路层
- ICMP：网络层
- ICMPv6：网络层
- HTTP：应用层
- MDNS：应用层