华东师范大学数据科学与工程学院实验报告

课程名称：计算机网络	**年级：**2024 级	**上机实践成绩：**
指导教师：赵明昊、王春阳	**姓名：**唐一嘉	**学号：**10245101434
上机实践名称：实验 2 TCP 拥塞控制和缓冲区膨胀		**上机实践日期：**2025.10.31
上机实践编号：	**组号：**	**上机实践时间：**8h

一、实验任务

创建自己的网络模拟，以研究 TCP 的动态以及缓冲区队列大小如何对性能产生重大影响。

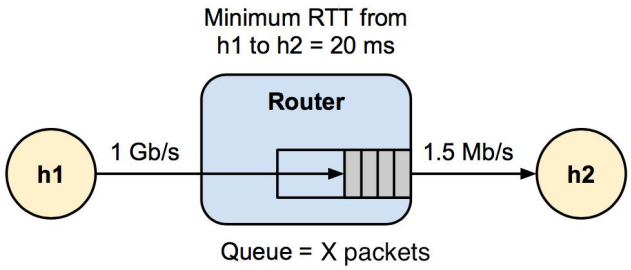
二、使用环境

Linux 系统，使用 Mininet 进行网络仿真。

三、实验过程

（一）、前置准备

1、首先，定义一个名为 BBTopo 的网络拓扑类，用于构建实验所要求的网络结构。要求的结构：



代码实现：

```
from mininet.topo import Topo


class BBTopo(Topo):
    "Simple topology for bufferbloat experiment."

    def __init__(self, queue_size):
        super(BBTopo, self).__init__()

        # Create switch s0 (the router)
        s0 = self.addSwitch("s0")

        # TODO: Create two hosts with names 'h1' and 'h2'
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        # TODO: Add links with appropriate bandwidth, delay, and queue size parameters.
        #       Set the router queue size using the queue_size argument
        #       Set bandwidths/latencies using the bandwidths and minimum RTT given in the network diagram above
        self.addLink(h1, s0, bw=1000, delay="10ms", max_queue_size=queue_size)
        self.addLink(s0, h2, bw=1.5, delay="10ms", max_queue_size=queue_size)

        return
```

2、接下来，添加流量生成函数。整个实验中，一共有三种流量：

(1). 持久的 TCP 会话（使用 iperf 创建）将大量流量从家庭计算机发送到服务器。

```
def start_iperf(net, experiment_time):
    # Start a TCP server on host 'h2' using perf.
    # The -s parameter specifies server mode
    # The -w 16m parameter ensures that the TCP flow is not receiver window limited (not necessary for client)
    print "Starting iperf server"
    h2 = net.get('h2')
    server = h2.popen("iperf -s -w 16m", shell=True)

    # TODO: Start an TCP client on host 'h1' using iperf.
    #       Ensure that the client runs for experiment_time seconds
    print "Starting iperf client"
    h1 = net.get('h1')
    client = h1.popen("iperf -c " + h2.IP() + " -t " + str(experiment_time), shell=True)
```

函数 **start_iperf** 来生成两台主机之间的流量，启动一个长期存在的 TCP 流，该流使用 iperf 将数据从 h1 发送到 h2。

(2). 定期对家庭计算机和服务器进行 ping 和 ping 回复。

```
def start_ping(net, outfile="pings.txt"):
    # TODO: Start a ping train from h1 to h2 with 0.1 seconds between pings, redirecting stdout to outfile
    print "Starting ping train"
    h1 = net.get('h1')
    h2 = net.get('h2')
    ping = h1.popen("ping -i 0.1 " + h2.IP() + " > " + outfile, shell=True)
```

函数 **start_ping** 启动从 h1 到 h2 的连续 ping 序列以测量 RTT，每 0.1 秒发送一次 ping。

(3). 定期尝试从服务器下载网站到家用计算机。

```
from time import sleep

def start_webserver(net):
    h1 = net.get('h1')
    proc = h1.popen("python http/webserver.py", shell=True)
    sleep(1)
    return [proc]
```

函数 **start_webserver** 是一个在 h1 上启动服务器的函数。

```
def fetch_webserver(net, experiment_time):
    h2 = net.get('h2')
    h1 = net.get('h1')
    download_times = []

    start_time = time()
    while True:
        sleep(3)
        now = time()
        if now - start_time > experiment_time:
            break
        fetch = h2.popen("curl -o /dev/null -s -w %{time_total} ", h1.IP(), shell=True)
        download_time, _ = fetch.communicate()
        print "Download time: {0}, {1:.1f}s left...".format(download_time, experiment_time - (now-start_time))
        download_times.append(float(download_time))

    average_time = mean(download_times)
    std_time = std(download_times)
    print "\nDownload Times: {}s average, {}s stddev\n".format(average_time, std_time)
```

函数 **fetch_webserver 是**一个在 h2 上运行的函数，在 experiment_time 内每 3 秒从 h1 获取网站，并打印下载时间的平均值和标准差。

3、然后，添加一系列监测函数，检测并记录拥塞窗口、队列长度和 ping RTT 随时间的变化来分析测量结果用于图表绘制与分析。

(1). 用于测量 TCP 流量的拥塞窗口的函数，其收集的数据可以使我们分析 mininet

网络中 TCP 连接的动态。

```
from subprocess import Popen
import os

def start_tcpprobe(outfile="cwnd.txt"):
    Popen("sudo cat /proc/net/tcpprobe > " + outfile, shell=True)

def stop_tcpprobe():
    Popen("killall -9 cat", shell=True).wait()
```

(2). 用于监视给定接口上的队列长度的函数，其收集的数据可以使我们分析路由器缓冲区队列中的数据包数量如何影响性能。

```
from multiprocessing import Process
from monitor import monitor_qlen

def start_qmon(iface, interval_sec=0.1, outfile="q.txt"):
    monitor = Process(target=monitor_qlen,
                      args=(iface, interval_sec, outfile))
    monitor.start()
    return monitor
```

(3). 用于监测 ping RTT 的代码被集成在了 **start_ping** 函数中

（二）、汇总运行

接下来，我们需要汇总运行所有代码，创建网络，启动所有流量并进行测量。

这一系列操作均由 **bufferbloat** 函数（代码过长不粘贴）完成，其具体完成了以下内容：

1. 创建 BBTopo 网络对象
2. 启动拥塞窗口和队列长度监视函数
3. 使用 iperf 启动一个长 TCP 流
4. 启动 ping 行为
5. 启动 webserver
6. 定期从服务器下载网页并测量下载时长

为作对比，使用 20 和 100 两种不同缓冲区队列大小进行测试：

```
bufferbloat(20, 300, "queue_20")
bufferbloat(100, 300, "queue_100")
```

（三）、绘制图像

```
def plot_measurements(experiment_name_list, cwnd_histogram=False):

    # plot the congestion window over time
    for name in experiment_name_list:
        cwnd_file = "{}_cwnd.txt".format(name)
        plot_congestion_window(cwnd_file, histogram=cwnd_histogram)

    # plot the queue size over time
    for name in experiment_name_list:
        qsize_file = "{}_qsize.txt".format(name)
        plot_queue_length(qsize_file)

    # plot the ping RTT over time
    for name in experiment_name_list:
        ping_file = "{}_pings.txt".format(name)
        plot_ping_rtt(ping_file)
```

这个函数会绘制三个图像，分别是拥塞窗口、队列长度和 ping RTT 随时间的变化。

（四）、实验结果与分析

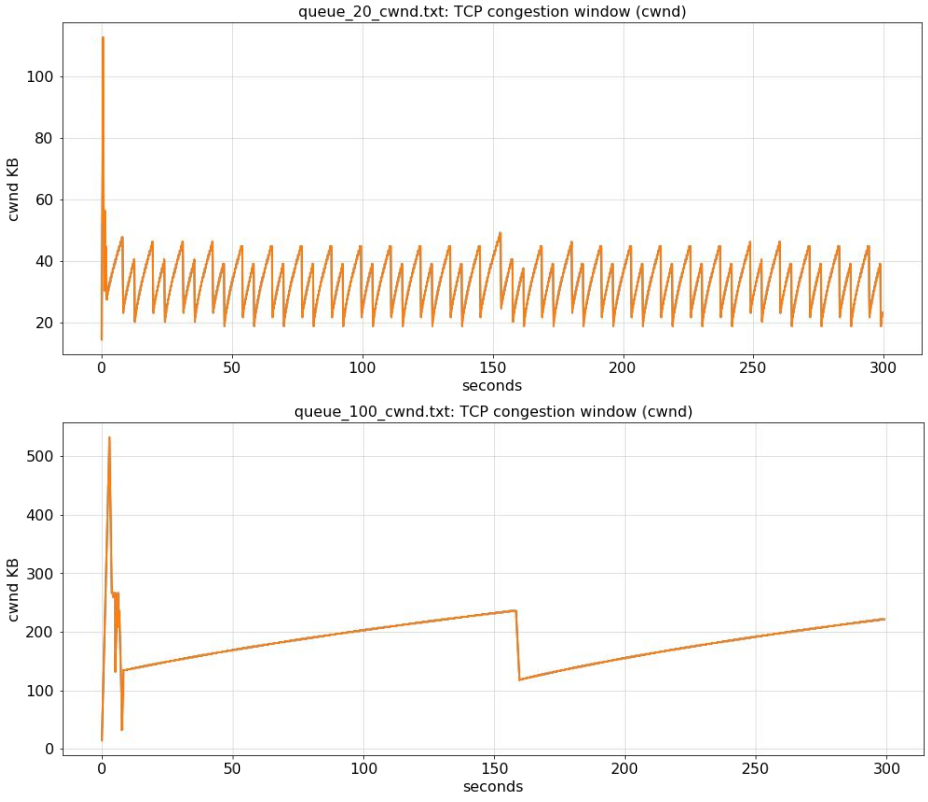
1、下载网页所需时间

queue_size 20 vs queue_size 100：

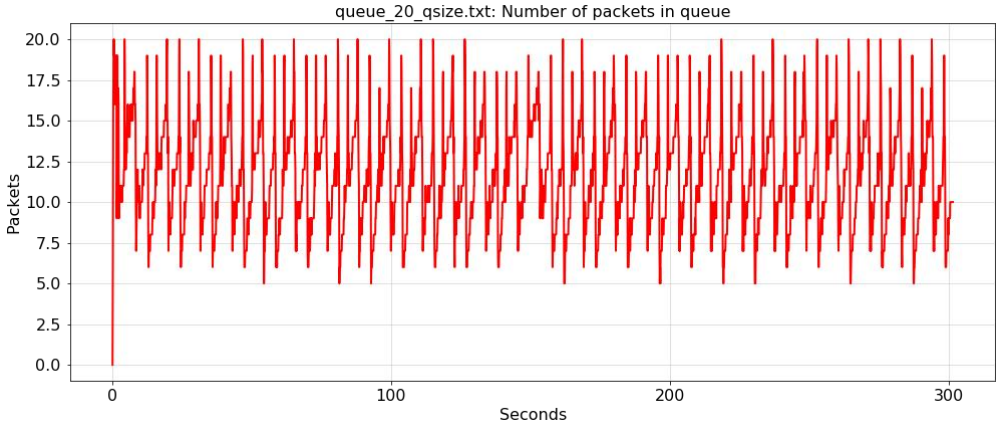
```
Starting iperf server                    Starting iperf server
Starting iperf client                    Starting iperf client
Starting ping train                      Starting ping train
Download time: 0.478, 297.0s left...     Download time: 6.563, 297.0s left...
Download time: 1.556, 293.5s left...     Download time: 1.646, 287.4s left...
Download time: 0.514, 288.9s left...     Download time: 1.674, 282.8s left...
Download time: 0.458, 285.4s left...     Download time: 1.737, 278.1s left...
Download time: 1.487, 281.9s left...     Download time: 1.773, 273.3s left...
Download time: 0.511, 277.4s left...     Download time: 1.817, 268.5s left...
Download time: 0.458, 273.9s left...     Download time: 1.867, 263.7s left...
Download time: 1.472, 270.4s left...     Download time: 1.903, 258.8s left...
Download time: 0.514, 266.0s left...     Download time: 1.960, 253.9s left...
Download time: 0.456, 262.4s left...     Download time: 2.013, 248.9s left...
Download time: 1.407, 259.0s left...     Download time: 2.027, 243.9s left...
Download time: 0.509, 254.5s left...     Download time: 2.074, 238.9s left...
Download time: 0.440, 251.0s left...     Download time: 2.112, 233.8s left...
Download time: 1.389, 247.6s left...     Download time: 2.161, 228.7s left...
Download time: 0.510, 243.2s left...     Download time: 2.192, 223.5s left...
Download time: 0.440, 239.6s left...     Download time: 2.226, 218.3s left...
Download time: 1.409, 236.2s left...     Download time: 2.258, 213.0s left...
Download time: 0.510, 231.8s left...     Download time: 2.308, 207.8s left...
Download time: 0.439, 228.2s left...     Download time: 2.357, 202.4s left...
Download time: 1.390, 224.8s left...     Download time: 2.387, 197.1s left...
Download time: 0.514, 220.4s left...     Download time: 2.420, 191.7s left...
Download time: 0.441, 216.9s left...     Download time: 6.041, 186.2s left...
Download time: 1.391, 213.4s left...     Download time: 2.519, 177.2s left...
                                         Download time: 6.287, 171.7s left...
                                         Download time: 2.614, 162.4s left...
                                         Download time: 2.650, 156.7s left...
```

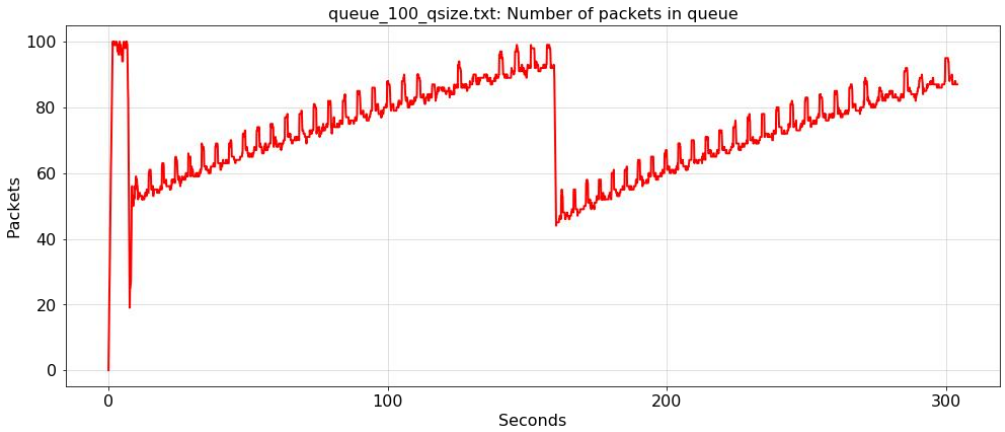
2、绘制的图表：

(1)、拥塞窗口

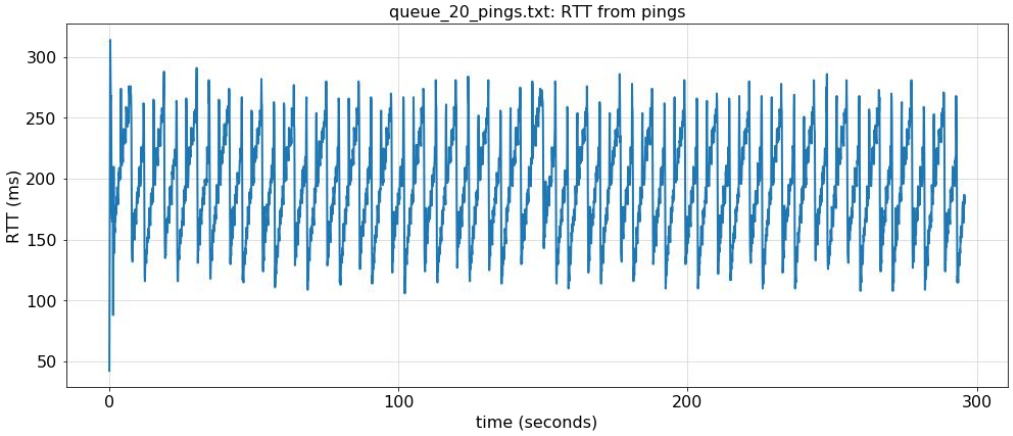


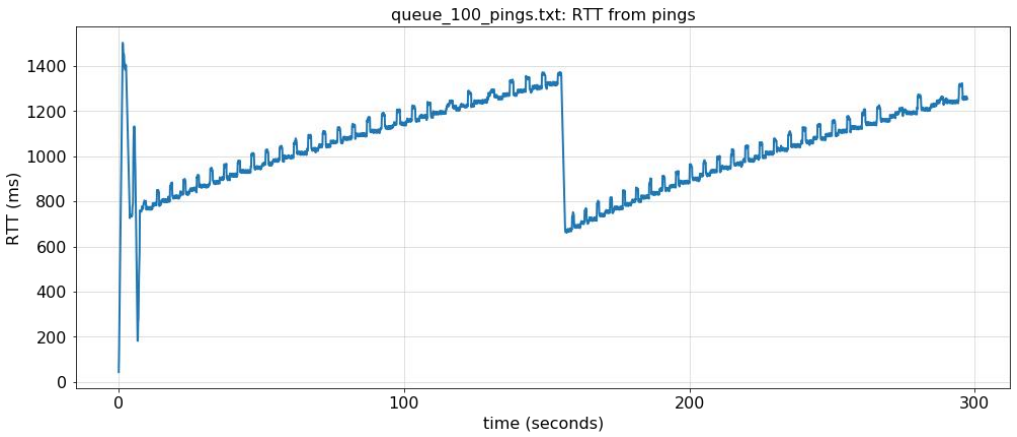
(2)、缓冲区队列中的数据包数量





(3)、ping RTT





3、实验分析

(1). 增加缓冲区大小对 TCP 拥塞控制的影响

在实验中，可以观察到当队列大小从 20 个包增加到 100 个包时，性能显著下降。这可以通过 TCP 的拥塞控制机制来解释：

拥塞窗口（Congestion Window, cwnd）动态：
- TCP 使用拥塞控制算法来调整发送速率
- 当数据包丢失时，TCP 检测到拥塞并减半 cwnd
- 然后线性增加 cwnd，形成典型的锯齿状

缓冲区大小的影响：
- 较大的缓冲区延迟了数据包丢失的检测
- 路由器队列填满之前，TCP 不会检测到拥塞
- 这导致大量数据包在队列中排队，增加了排队延迟

(2). 增加缓冲区大小对 RTT 的影响

从 ping RTT 图中我们可以观察到：

延迟增加：
- 大缓冲区（100 包）情况下，RTT 显著增加
- RTT = 基础传播延迟 + 排队延迟
- 排队延迟 =（队列中的包数 × 包长度）/ 链路带宽

RTT 变化的波动性：
- 大缓冲区导致 RTT 波动更大
- 当队列满时，RTT 达到峰值
- TCP 拥塞控制触发后，队列清空，RTT 下降

(3). 增加缓冲区大小对网页下载时间的影响

从网页下载时间的测量结果可以看出：

平均下载时间增加：
- 大缓冲区情况下，平均下载时间从 0.80 秒增加到 2.44 秒
- 下载时间标准差也从 0.45 增加到 1.23，表明性能不稳定

排队阻塞：
- 大量的 iperf 流量填满队列
- HTTP 流量被延迟，因为需要等待队列中的包被传输

(4). 拥塞控制机制效果下降

缓冲区膨胀的根本问题：

- TCP 依赖数据包丢失作为拥塞信号
- 大缓冲区推迟了拥塞信号的产生
- 这导致 TCP 无法及时调整发送速率

带宽延迟积（BDP）问题：

- 大缓冲区允许更多的数据在传输中
- 但瓶颈链路的实际带宽没有增加
- 结果是延迟增加而吞吐量没有相应改善

(5). 图示验证

从生成的图表中可以清楚地看到：

1. 拥塞窗口图：显示了 TCP 的锯齿状行为，但大缓冲区情况下波动更剧烈
2. 队列长度图：大缓冲区允许队列增长到更高水平，进而导致更多排队延迟
3. Ping RTT 图：直接显示了 RTT 随着缓冲区大小增加而增加的现象

增加缓冲区大小会降低性能的核心是缓冲区大小与 TCP 拥塞控制机制之间的不匹配。当缓冲区过大时，TCP 无法及时检测到拥塞，导致过多的数据包排队，从而显著增加延迟和 RTT 波动，降低网页下载性能。

四、总结

本次实验主要研究 TCP 拥塞控制和缓冲区膨胀(bufferbloat)现象，通过 Mininet 网络模拟器创建了一个简单的网络拓扑来观察不同队列大小对网络性能的影响。

缓冲区膨胀是一个普遍存在的网络问题，不是特定于某种网络或协议的现象。根本原因是缓冲区大小与 TCP 拥塞控制机制之间的不匹配。通过合理的队列管理策略，可以在不减少缓冲区大小的情况下缓解这一问题。

本次实验让我从理论到实践全面理解了 TCP 拥塞控制和缓冲区管理的重要性，为今后处理实际网络性能问题打下了坚实基础。