

# netlab report

罗思佳2021201679

## PART A 套接字编程

### 一、实验要求

编写2对 TCP 客户端和服务端程序，用于通过 Internet发送和接收文本消息。一对客户端/服务器必须用 C 编写。另一对可以用Python 或 Go 编写，需要从Python 和 Go 中选择一种作为第二对客户端/服务器的实现语言。

具体要求见README.pdf

### 二、实验过程

#### (一) C语言实现

##### 1.服务端

server函数的逻辑是：

先调用 `socket` 函数创建套接字，设置 `server_addr` 的 `sin_family` 为 `AF_INET`，`sin_port` 用 `htons` 将端口号转换成大端，`sin_addr.sin_addr` 设为 `INADDR_ANY`，然后调用 `bind` 函数将 `lfd` 和本地地址绑定在一起，然后调用 `listen` 函数进行监听，注意函数的第二个参数设为最大连接数 `QUEUE_LENGTH`。（使用 `perror` 处理异常）

因为服务端要按顺序处理多个客户端的请求，所以用一个 `while` 循环来循环处理多个客户端的请求，在循环中调用 `accept` 函数接受客户端连接。在成功与当前客户端连接后，因为一个客户端可能会发多次消息（长消息分批发送），所以又要用到一个 `while` 循环来处理一个客户端的多次发送的消息。

在第二个 `while` 循环中，定义接受消息的缓冲字符串 `recv_buffer` 并初始化，然后调用 `read` 函数接受 `RECV_BUFFER_SIZE` 个字节的消息，用 `fwrite` 函数打印在屏幕上，打印后还要调用 `fflush` 函数清空缓冲区。

每次处理完一个客户端的请求就 `close` 掉 `cfd`，然后处理下一个。

最后调用 `close` 函数关闭 `lfd`。

函数代码：

```
int server(char *server_port)
{
    int lfd;
    int port = atoi(server_port);
    struct sockaddr_in server_addr;
    socklen_t addr_size;
    // Create socket
    if ((lfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket error");
        exit(0);
    }
}
```

```

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port);
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
// Bind socket to address and port
if (bind(lfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1)
{
    perror("bind error");
    exit(0);
}
// Listen for incoming connections
if (listen(lfd, QUEUE_LENGTH) == -1)
{
    perror("listen error");
    exit(0);
}
// printf("Server started. Listening on port %s\n", server_port);
// Accept and handle client connections
while (1)
{
    // Accept connection
    int cfd;
    struct sockaddr_in client_addr;
    addr_size = sizeof client_addr;
    cfd = accept(lfd, (struct sockaddr *)&client_addr, &addr_size);
    if (cfd == -1)
    {
        perror("accept error");
        exit(0);
    }
    // Receive and print message
    ssize_t num_bytes;
    while (1)
    {
        char recv_buffer[RECV_BUFFER_SIZE + 1]; // 注意这里要加一
        memset(recv_buffer, 0, sizeof(recv_buffer));
        num_bytes = read(cfd, recv_buffer, RECV_BUFFER_SIZE * sizeof(char));
        if (num_bytes > 0)
        {
            // printf("%s", recv_buffer);
            fwrite(recv_buffer, sizeof(char), num_bytes, stdout);
            fflush(stdout);
        }
        else if (num_bytes == 0)
        {
            // printf("The client close the connection.\n");
            break;
        }
        else
        {
            perror("recv error");
            break;
        }
    }
    if (num_bytes == -1)
    {
        perror("recv error");
    }
    // Close the connection

```

```

        close(cfd);
    }
    close(lfd);
    return 0;
}

```

## 2.客户端

client函数的逻辑是：

调用 `socket` 函数创建套接字，返回给 `sockfd`，将函数的参数 `server_port` 用 `atoi` 函数转换成int型，然后设置客户端地址 `addr` 的 `sin_family`、`sin_port`（同样进行大小端转换），调用 `inet_pton` 函数将ip地址转换为大端。然后调用 `connect` 函数与服务器建立连接。

根据client规范，“每个客户端必须能够通过循环迭代来读取和发送消息块来处理比较大的消息，而不是直接先将整个消息读入内存”，因此对于长消息要分批发送，所以我用了一个while循环，在每次循环中，定义用于存输入数据的字符串 `send_buffer` 并初始化，然后调用 `read` 函数从标准输入设备 `STDIN_FILENO` 中读取 `SEND_BUFFER_SIZE` 字节的数据到 `send_buffer` 中，然后再用 `write` 函数发送读到的数据。

所有消息发送完成后关闭 `sockfd`。

客户端函数代码：

```

int client(char *server_ip, char *server_port)
{
    int sockfd, port;
    port = atoi(server_port);

    // Create socket
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket error");
        exit(0);
    }
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    inet_pton(AF_INET, server_ip, &addr.sin_addr.s_addr);
    // Connect to server
    if (connect(sockfd, (struct sockaddr *)&addr, sizeof(addr)) == -1)
    {
        perror("connect error");
        exit(0);
    }
    // printf("Connected to server successfully at %s:%s\n", server_ip,
server_port);
    // Send message from stdin
    while (1)
    {
        ssize_t num_bytes;
        // printf("Please enter the msg:\n");
        char send_buffer[SEND_BUFFER_SIZE + 1]; // 注意要加1
        memset(send_buffer, 0, sizeof(send_buffer));

```

```

        num_bytes = read(STDIN_FILENO, send_buffer, SEND_BUFFER_SIZE *
sizeof(char));
        if (num_bytes == -1)
        {
            perror("read error");
            break;
        }
        if (num_bytes == 0)
            break; // Reached EOF

        ssize_t bytes_sent = write(sockfd, send_buffer, num_bytes);
        // printf("Send: %s\n", send_buffer);
        // fflush(stdout);
        if (bytes_sent == -1)
        {
            perror("send error");
            break;
        }
    }

    // Close the socket
    close(sockfd);

    return 0;
}

```

附上自己测试时的效果：（加上了提示词）

```

vagrant@netruc: /vagrant/client_server
vagrant@netruc:/vagrant/client_server$ ./server-c 12345
Server started. Listening on port 12345
Received: hello
Received: abcdefg
Received: 1234567
Received: 12345

vagrant@netruc: /vagrant/client_server
vagrant@netruc:/vagrant/client_server$ ./client-c 127.0.0.1 12345
connected to server successfully at 127.0.0.1:12345
Please enter the msg:
hello
Please enter the msg:
abcdefg
Please enter the msg:
1234567
Please enter the msg:
12345
Please enter the msg:

```

## （二）Python实现

### 1.服务端

python的server函数思路和c的基本一样，因为python的语法比较简单，所以只需要十几行代码就能完成。

有一点不同是，客户端可能送文本消息也可能发送二进制消息，不过是哪种消息，服务端接收到的都是字节数据，如果是文本消息需要先解码再输出，如果是二进制消息就直接输出。因此这里需要加一个判断，可以用 `try/except` 来实现，先将其当作文本消息并尝试解码输出，如果发生异常（解码失败）说明是二进制消息，进入到 `except` 分支里，调用函数 `sys.stdout.buffer.write(recv_buffer)` 直接打印二进制数据。注意每次输出后要调用 `sys.stdout.flush()`。

```

def server(server_port):
    """TODO: Listen on socket and print received message to sys.stdout"""
    lfd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    lfd.bind(('127.0.0.1', server_port))
    lfd.listen(QUEUE_LENGTH)
    # print("Server started. Listening on port {}".format(server_port))
    # Accept and handle client connections
    while True:
        cfd, client_addr = lfd.accept()
        # Receive and print message
        while True:
            recv_buffer = cfd.recv(RECV_BUFFER_SIZE)
            if len(recv_buffer) > 0:
                try:
                    # 如果是文本消息，先解码再输出
                    recv_text = recv_buffer.decode()
                    sys.stdout.write(recv_text)
                    sys.stdout.flush()
                except UnicodeDecodeError:
                    # 是二进制消息，直接输出
                    sys.stdout.buffer.write(recv_buffer)
                    sys.stdout.buffer.flush()
            else:
                # print "The client closed the connection."
                break

        cfd.close()

    lfd.close()

```

## 2.客户端

python的客户端与c的逻辑也基本一致，有一点需要注意的是从标准输入读取数据时，可能是文本消息也可能是二进制流，但不管是哪种消息发送过去的时候都是字节数据，所以可以直接用 `sys.stdin.buffer.read` 函数读取原始的字节数据然后发送出去。

注意： `sys.stdin.buffer.read()` 是 Python 中用于从标准输入（stdin）的二进制流中读取字节数据的方法。它可以读取指定数量的字节数据，并返回一个包含所读取字节的字节对象（bytes object）。与 `sys.stdin.read()` 方法不同， `sys.stdin.buffer.read()` 不会进行文本编码转换或解码操作，而是直接读取原始的字节数据。

```

def client(server_ip, server_port):
    """TODO: Open socket and send message from sys.stdin"""
    sockfd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Connect to the server
    sockfd.connect((server_ip, server_port))
    # print("Connected to server successfully")

    # Send message from stdin
    while True:
        send_data = sys.stdin.buffer.read(SEND_BUFFER_SIZE)

        if not send_data:
            break

```

```
# Send data to the server
sockfd.send(send_data)

# Close the socket
sockfd.close()
```

### 三、实验结果

20个测试点都能通过

```
=====
16. TEST SHORT MESSAGE
SUCCESS: Message received matches message sent!
-----
17. TEST RANDOM ALPHANUMERIC MESSAGE
SUCCESS: Message received matches message sent!
-----
18. TEST RANDOM BINARY MESSAGE
SUCCESS: Message received matches message sent!
-----
19. TEST SERVER INFINITE LOOP (multiple sequential clients to same server)
SUCCESS: Message received matches message sent!
-----
20. TEST SERVER QUEUE (overlapping clients to same server)
SUCCESS: Message received matches message sent!
=====
TESTS PASSED: 20/20
vagrant@netruc:/vagrant/client_server$
```

### 四、实验中遇到的问题

c语言版本：

#### 1.字符串长度的问题

因为一次最多发送和接受2048字节的数据，所以我最开始设的 `send_buffer` 和 `recv_buffer` 的大小都是2048，测试的时候发现短信可以通过测试，而长消息不能，于是我就对长消息进行单独测试，发现输入长消息后，服务端打印出来的每一段末尾都会显示不正确（有一个小方格），我就明白了是我没有给字符串末尾的'\0'留一个位置，而导致出错。之后我把两个字符串的长度都加了1之后就能正常打印了。

```
char send_buffer[SEND_BUFFER_SIZE + 1]; // 注意要加1
```

```
char recv_buffer[RECV_BUFFER_SIZE + 1]; // 注意这里要加1
```

#### 2.服务端打印的问题

最开始我用的是 `printf` 函数进行打印输出，发现字母数字消息可以正常打印，但二进制消息不能，中间折腾了半天，想要用if-else分开处理，但还是通不过，总是报以下错误：

```

8. TEST RANDOM BINARY MESSAGE
Traceback (most recent call last):
  File "../client-python.py", line 45, in <module>
    main()
  File "../client-python.py", line 41, in main
    client(server_ip, server_port)
  File "../client-python.py", line 22, in client
    send_data = sys.stdin.read(SEND_BUFFER_SIZE)
  File "/usr/lib/python3.4/codecs.py", line 319, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, fina
1)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xa2 in position
3: invalid start byte
)
FAILURE: Message received doesn't match message sent.

```

我就思考是不是 `printf` 的原因，上网查了一下，找到一篇博客([117条消息\)整理fprintf\(\)、sprintf\(\)、printf\(\)、fwrite\(\)函数的用法与区别](#) [Echo-Young的博客-CSDN博客](#)里面整理了几个打印函数的区别，知道了 `printf()` 函数是按照格式化字符串的规则将数据转换为可打印的文本，而二进制消息往往不是以文本形式存储的，相比之下，`fwrite()` 函数将数据以原始字节的形式写入文件流（在这里是标准输出流），没有格式化的过程，能够准确地输出二进制消息。我将 `printf` 函数换成 `fwrite` 函数之后，果然就能通过那几个测试点了。

## python版本

### 1.客户端读取和发送的问题

和c版本遇到的问题比较类似，我一开始是照着网上的最常见的python版本写的，就是客户端先 `encode()`，服务端 `decode()` 再输出，这样也是能处理字母数字消息而不能处理二进制消息。去查了一下 `encode` 的含义，是以指定的编码格式编码字符串，默认编码为 'utf-8'，而二进制流本身不需要编码，所以再用 `encode` 编码的话就会出错。这里我又是学到了一个读取的函数，叫做 `sys.stdin.buffer.read`，是 Python 中用于从标准输入（stdin）的二进制流中读取字节数据的方法。它可以读取指定数量的字节数据，与 `sys.stdin.read()` 方法不同，`sys.stdin.buffer.read()` 不会进行文本编码转换或解码操作，而是直接读取原始的字节数据。而我最开始就是用的 `sys.stdin.read`！

明白了其中的原理之后，我就把读取的函数改成了 `sys.stdin.buffer.read`，把 `encode` 去掉了，这样不论是文本消息还是二进制消息，读进来和发送的都是二进制流数据。

### 2.服务端打印的问题

发送成功了之后，还要能正确打印才算成功。我一开始也是直接用 `print` 函数，后来改成了 `sys.stdout.write`，因为不会自动换行，但处理二进制消息还是会出错，于是我根据之前该客户端代码的经验，找到了 `sys.stdout.buffer.write` 这个函数，该函数接受字节串（bytes）类型的参数，直接写入到标准输出，不进行字符编码转换。

为了能够同时处理文本消息和二进制消息，我利用了 `try-except` 这对关键字，先对接收到的字符串进行解码，然后使用 `sys.stdout.write` 输出，如果发生异常，说明是二进制消息，就进入到 `except` 分支，用 `sys.stdout.buffer.write` 直接输出。

这些都改了之后，终于能通过所有测试点了，还是需要注意细节啊。

## 五、实验心得

在PART A中，我理解了socket编程的基本流程和原理，包括套接字的创建、绑定和监听、客户端和服务端端的交互等；亲身体验了网络通信的过程，通过编写客户端和服务端端的代码，我可以模拟客户端与服务端之间的交互，并观察数据的发送和接收过程，这让我更好地理解了网络通信的实际运行机制；初步理解了数据是如何在网络中进行传输的，也学会了使用调试工具和技巧来排查问题，还是很有成就感的。

## PART B TCP 拥塞控制和 Bufferbloat(选做)

因为做完PART A还有良两天才到ddl，所以我也做了一下PART B。

### 一、实验要求

使用Mininet来模拟一个小型网络，并收集与TCP拥塞控制和Bufferbloat相关的各种性能统计数据，完成bufferbloat.ipynb。

### 二、实验过程

采用方式一：jupyter进行实现。

按照README上的指示，进入虚拟机上的Jupyter Notebook，填写里面的bufferbloat.ipynb。

首先是创建自定义网络拓扑部分，创建了两个主机 h1 和 h2。接下来，使用 addLink 方法添加了两个连接，分别连接 h1 和 s0 以及 h2 和 s0。对于这两个连接，设置了带宽（bw）、延迟（delay）和队列大小（max\_queue\_size）参数。

```
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)
        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='20ms', max_queue_size=queue_size)
        self.addLink(h2, 's0', bw=1.5, delay='20ms')

        return
```

然后是完成 start\_tcp 函数，这个函数会在 h1 上启动一个TCP客户端，并在 h2 上启动一个TCP服务器。客户端和服务端进程会在 experiment\_time 秒后被终止，并打印出任何输出。



```

from time import sleep
def start_tcp(net, experiment_time):
    # Start a TCP server on host 'h2'
    print "Starting server"
    h2 = net.get('h2')
    server = h2.popen("../client_server/server-c 12345", shell=True)

    # TODO: Start an TCP client on host 'h1'.
    #       Ensure that the client runs for experiment_time seconds
    print "Starting client"
    h1 = net.get('h1')
    client_cmd = "../client_server/client-c " + h2.IP() + " 12345"
    client = h1.popen(client_cmd, shell=True)

    # Wait for experiment_time seconds
    print "Running experiment for", experiment_time, "seconds"
    sleep(experiment_time) # 等待client完成, 即等待整个通信用程完成

    # Clean up
    server.terminate()
    #client.terminate()

```

`start_ping` 函数在 `h1` 上启动一个 ping train, 从 `h1` 到 `h2` 发送 ping, 每隔 0.1 秒发送一次, 并将 stdout 重定向到指定的 `outfile` 文件中。ping train 会运行 2 秒钟, 然后被停止。

```

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_cmd = "ping -i 0.1 " + h2.IP() + " > " + outfile
    h1.popen(ping_cmd, shell=True)

```

然后是完成 `bufferbloat` 函数, 主要是调之前写好的函数接口

```

from mininet.node import CPULimitedHost, OVSTController
from mininet.link import TCLink
from mininet.net import Mininet
from mininet.log import lg, info
from mininet.util import dumpNodeConnections

from time import time
import os
from subprocess import call

def bufferbloat(queue_size, experiment_time, experiment_name):
    # Don't forget to use the arguments!

    # Set the cwnd control algorithm to "reno" (half cwnd on 3 duplicate acks)
    # Modern Linux uses CUBIC-TCP by default that doesn't have the usual
    # sawtooth
    # behaviour. For those who are curious, replace reno with cubic
    # see what happens...

```

```

os.system("sysctl -w net.ipv4.tcp_congestion_control=reno")

# create the topology and network
topo = BBTopo(queue_size)
net = Mininet(topo=topo, host=CPULimitedHost, link=TCLink,
              controller= OVSController)
net.start()

# Print the network topology
dumpNodeConnections(net.hosts)

# Performs a basic all pairs ping test to ensure the network set up properly
net.pingAll()

# Start monitoring TCP cwnd size
outfile = "{}_cwnd.txt".format(experiment_name)
start_tcpprobe(outfile)

# TODO: Start monitoring the queue sizes with the start_qmon() function.
#       Fill in the iface argument with "s0-eth2" if the link from s0 to h2
#       is added second in BBTopo or "s0-eth1" if the link from s0 to h2
#       is added first in BBTopo. This is because we want to measure the
#       number of packets in the outgoing queue from s0 to h2.
iface = "s0-eth2" if "s0-eth2" in net.get('s0').intfNames() else "s0-eth1"
outfile = "{}_qsize.txt".format(experiment_name)
qmon = start_qmon(iface=iface, outfile=outfile)

# TODO: Start the long lived TCP connections with the start_tcp() function
start_tcp(net, experiment_time)

# TODO: Start pings with the start_ping() function
outfile = "{}_pings.txt".format(experiment_name)
start_ping(net, outfile=outfile)

# TODO: Start the webserver with the start_webserver() function
webserver_procs = start_webserver(net)

# TODO: Measure and print website download times with the fetch_webserver()
function
fetch_webserver(net, experiment_time)

# Stop probing
stop_tcpprobe()
qmon.terminate()
net.stop()

# Ensure that all processes you create within Mininet are killed.
Popen("pgrep -f webserver.py | xargs kill -9", shell=True).wait()
call(["mn", "-c"])

```

再使用bufferbloat()函数运行两次实验，一次队列大小为20个包，一次队列大小为100个包，分别运行300s

```

from subprocess import call
call(["mn", "-c"])

# TODO: call the bufferbloat function twice, once with queue size of 20 packets
and once with a queue size of 100.
bufferbloat(queue_size=20, experiment_time=300, experiment_name="experiment_20")
bufferbloat(queue_size=100, experiment_time=300,
experiment_name="experiment_100")

```

```

h1 h1-eth0:s0-eth1
h2 h2-eth0:s0-eth2
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)

```

```

Starting server
Starting client
Running experiment for 300 seconds
Starting ping train
Download time: 0.216, 297.0s left...
Download time: 0.217, 293.7s left...
Download time: 0.213, 290.5s left...
Download time: 0.214, 287.3s left...
Download time: 0.213, 284.0s left...
Download time: 0.215, 280.8s left...
Download time: 0.213, 277.5s left...
Download time: 0.216, 274.2s left...
Download time: 0.215, 271.0s left...
Download time: 0.216, 267.8s left...
Download time: 0.215, 264.5s left...
Download time: 0.213, 261.3s left...
Download time: 0.215, 258.1s left...
Download time: 0.215, 254.8s left...
Download time: 0.215, 251.6s left...
Download time: 0.215, 248.4s left...

```

最后是结果可视化

```

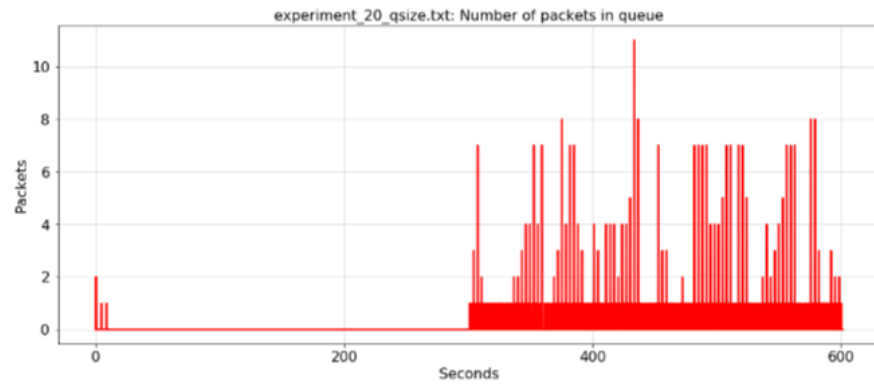
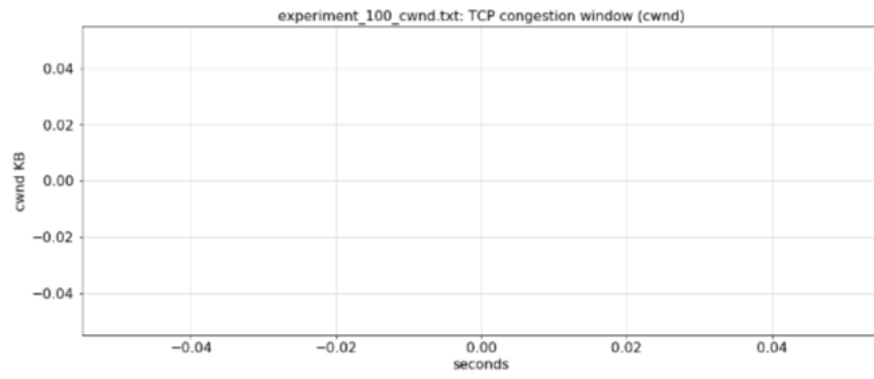
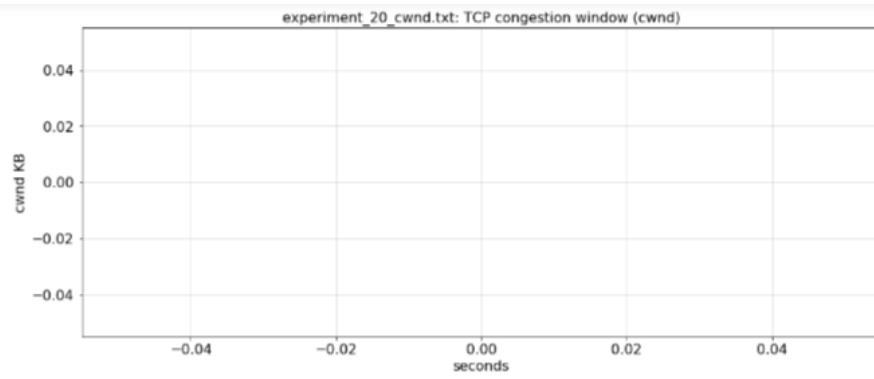
#TODO: Call plot_measurements() to plot your results
# Create a list of experiment names
experiment_name_list = ["experiment_20", "experiment_100"]

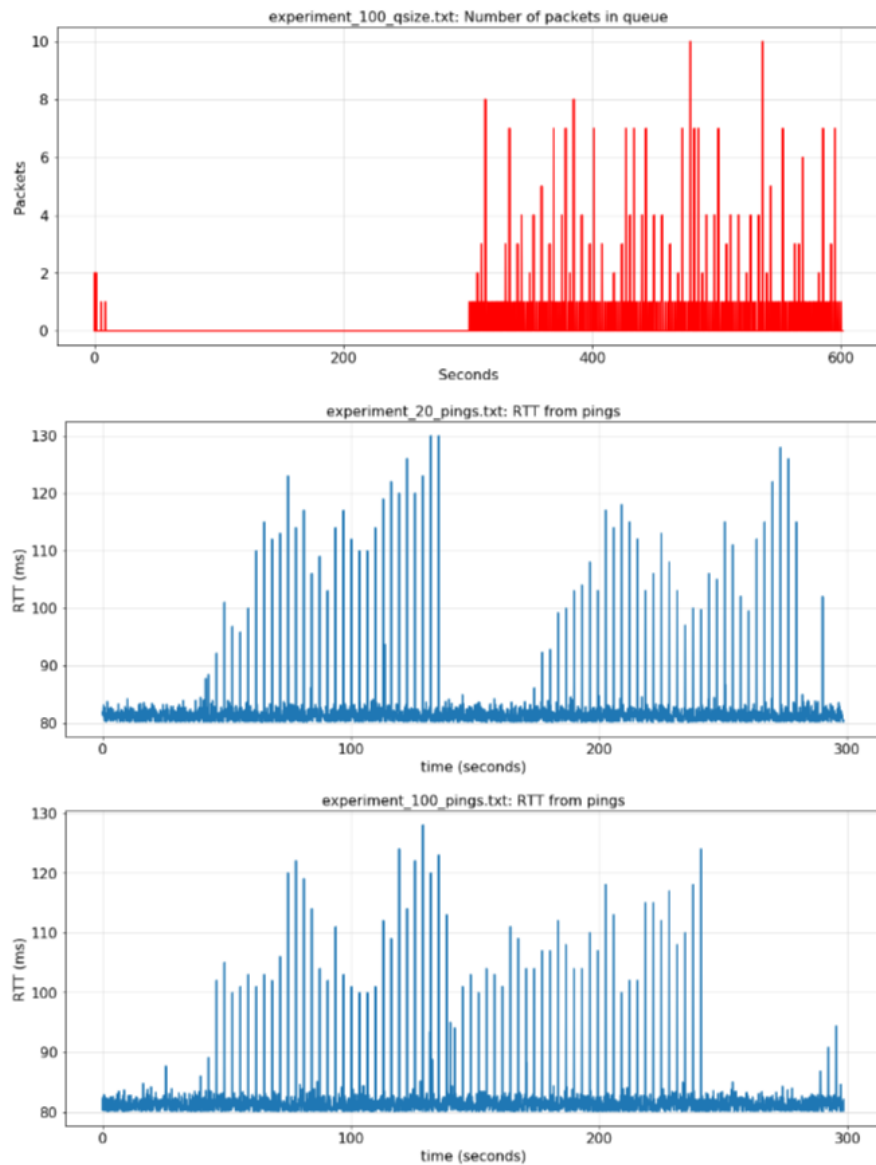
# Call plot_measurements() to plot the results
plot_measurements(experiment_name_list, cwnd_histogram=False)

```

### 三、实验结果

不幸的是六个图中前两个cwnd的变化图未能成功画出，后面四个图是正常的，可以观察到抖动情况。





## 四、实验中遇到的困难

PART B遇到的最大的困难就是未能获取cwnd的变化情况了，一开始像无头苍蝇一样到处找问题，之后发现是 `start_tcp` 函数里 `popen` 的两行代码运行的命令没写正确，主要是因为自己对 `popen` 函数不了解。

但改过来之后依然不能捕捉到cwnd变化的数据，即那两个 `cwnd.txt` 文件里是空白的，双端还是不能正常通信，我就在网上找有没有什么能检测服务端客户端运行状态的方法，然后学到了一个命令，`ps aux | grep xxx`，能对进程进行监测和控制，我用这个命令看服务端和客户端的状态发现，服务端是正常运行的，而客户端是僵死（defunct）的，运行了好几次都是这样，如下图

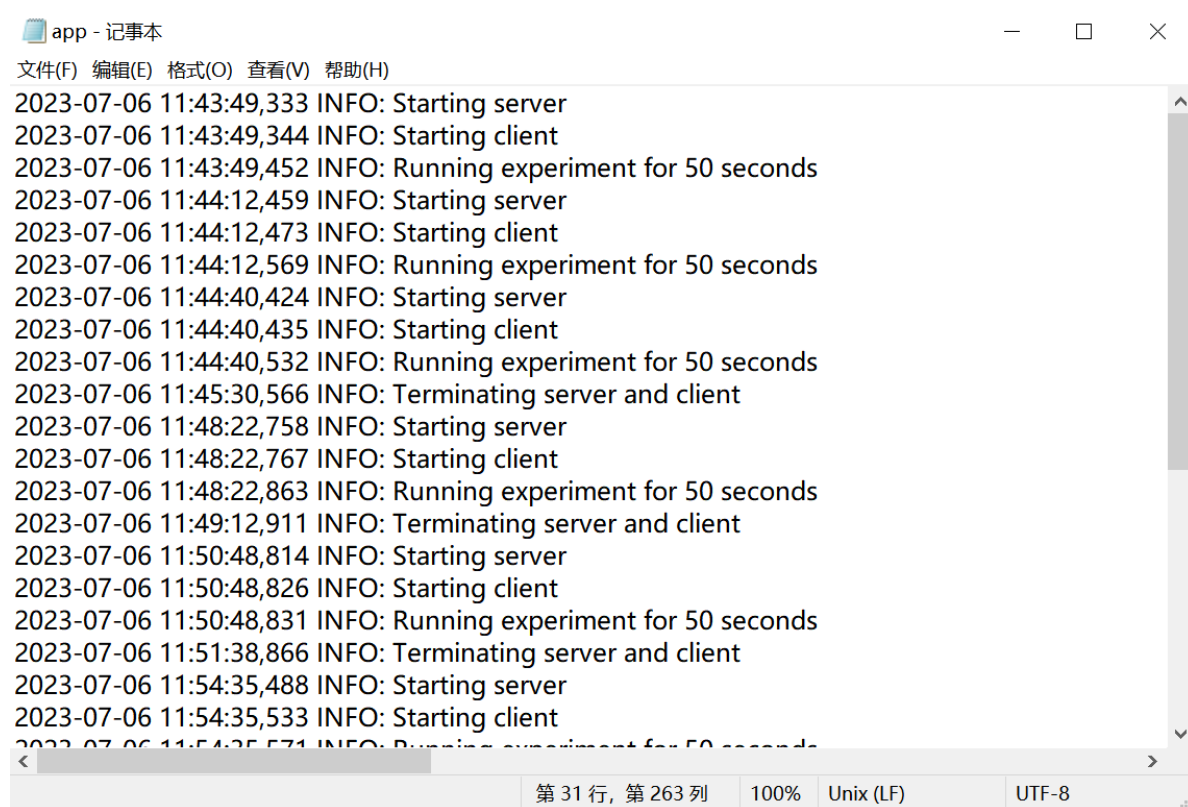
```

vagrant@netruc:/vagrant/client_server$ ps aux | grep server-c
root      10815  0.0  0.0   4192   348 ?        ss      11:09   0:00
./client_server/server-c 12346
vagrant   11162  0.0  0.0  10468   940 pts/1    s+      11:09   0:00
grep --color=auto server-c
vagrant@netruc:/vagrant/client_server$

vagrant@netruc:/vagrant/client_server$ ps aux | grep client-c
root      10817  0.0  0.0      0      0 ?        Zs      11:09   0:00
[client-c] <defunct>
vagrant   10872  0.0  0.0  10468   940 pts/1    s+      11:09   0:00
grep --color=auto client-c
vagrant@netruc:/vagrant/client_server$

```

问题是找到了，但是想解决还是挺难的，我的server和client在PART A是能正常运行的，也注意到了h2的ip地址是10.0.0.2，代码里没有直接写之前的127.0.0.1，所以不知道是哪里出了问题。询问了助教后，我也尝试了打印日志（如下图），但还是看不出问题来



```
app - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
2023-07-06 11:43:49,333 INFO: Starting server
2023-07-06 11:43:49,344 INFO: Starting client
2023-07-06 11:43:49,452 INFO: Running experiment for 50 seconds
2023-07-06 11:44:12,459 INFO: Starting server
2023-07-06 11:44:12,473 INFO: Starting client
2023-07-06 11:44:12,569 INFO: Running experiment for 50 seconds
2023-07-06 11:44:40,424 INFO: Starting server
2023-07-06 11:44:40,435 INFO: Starting client
2023-07-06 11:44:40,532 INFO: Running experiment for 50 seconds
2023-07-06 11:45:30,566 INFO: Terminating server and client
2023-07-06 11:48:22,758 INFO: Starting server
2023-07-06 11:48:22,767 INFO: Starting client
2023-07-06 11:48:22,863 INFO: Running experiment for 50 seconds
2023-07-06 11:49:12,911 INFO: Terminating server and client
2023-07-06 11:50:48,814 INFO: Starting server
2023-07-06 11:50:48,826 INFO: Starting client
2023-07-06 11:50:48,831 INFO: Running experiment for 50 seconds
2023-07-06 11:51:38,866 INFO: Terminating server and client
2023-07-06 11:54:35,488 INFO: Starting server
2023-07-06 11:54:35,533 INFO: Starting client
2023-07-06 11:54:35,571 INFO: Running experiment for 50 seconds
第 31 行, 第 263 列 100% Unix (LF) UTF-8
```

我也尝试了一些其他的方法，比如 `client.communicate()` 用获取其输出和错误信息，但打印出来是空字符串。。。

在尝试了上述以及其他的一些方法后，最终还是没能解决问题，眼看着就要到截止时间了，于是只好无奈放弃。

## 五、实验心得

虽然没能顺利地完PART B，但我的收获是很大的，了解到了一些拥塞控制原理，如何观察和分析TCP行为，如何使用Mininet构建网络拓扑、设置流量控制参数、监测网络状态和性能指标，并使用工具和绘图技术对实验结果进行可视化和分析，还有就是在解决困难时如何用一些命令和技巧逐步排查错误，比如我学会了用ps命令查看进程状态，从而确定了出错原因是客户端僵死，虽然最终没能解决它，但我学到的东西和收获的经验是大于这一个bug的，也非常非常感谢助教对我提问的及时解答和耐心指导🙏

Anyway，终于完成了为期一年的ICS的最后一个lab🎉 完结撒花🎉