

计算机网络及安全 PJ

20307130135 李钧 信息安全 2020 级

一、实验目的

理解滑动窗口协议的基本原理，掌握 GBN/SR 的工作原理，掌握基于 UDP 设计并实现一个 GBN 协议的过程与技术，并在此基础上改进实现选择重传协议。

二、实验内容

- 1、基于 UDP 设计一个简单的 GBN 协议，实现双向可靠数据传输
- 2、模拟引入数据包的丢失，验证所设计协议的有效性
- 3、改进 GBN 协议并实现 SR 协议
- 4、扩展：在此基础上可以增加拥塞控制

三、实现 GBN 协议，单项可靠数据传输

1、gbn.py 包

1.1 为了实现 GBN 协议，通信主机发送端需要定义有窗口大小、下一个序列号、待发送的分组个数、超时等待时间，上述内容在 gbn 类中的定义如下：

```
BUFFER_SIZE = 4096
TIMEOUT = 10
WINDOW_SIZE = 3
LOSS_RATE = 0.1
```

1.2 在这个类中，定义了对校验码的计算过程如下图所示：

```
def getChecksum(data):
    """
    char_checksum 按字节计算校验和。每个字节被翻译为无符号整数
    @param data: 字符串
    """
    length = len(str(data))
    checksum = 0
    for i in range(0, length):
        checksum += int.from_bytes(bytes(str(data)[i], encoding='utf-8'), byteorder='little', signed=False)
        checksum &= 0xFF # 强制截断
    return checksum
```

2、gbn_client.py 包

2.1 设置发送主机为本机、端口号 8888，读取将要发送的文件 data.jpg 并将其转化成二进制数组：

```
HOST = '127.0.0.1'
PORT = 8888
ADDR = (HOST, PORT)
CLIENT_DIR = os.path.dirname(__file__) + '/client'

senderSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sender = gbn.GBNSender(senderSocket, ADDR)
```

```

dataList = []
while True:
    data = fp.read(2048)
    if len(data) <= 0:
        break
    dataList.append(data)
print('The total number of data packets: ', len(dataList))

```

2.2 在循环中不断发送分组，如果分组指针还没有到可分数据段的最大位置，则继续打包发送数据包，并调整 **pointer** “指针” 自增 1；如果指到了所有分组的最后一个则设置 **stop** 为真，不再打包发送。在循环中实现了一次性发完整个窗口的内的分组，并且等待对应序列号的 **ACK** 返回。

```

pointer = 0
while True:
    while sender.next_seq < (sender.send_base + sender.window_size):
        if pointer >= len(dataList):
            break
        # 发送窗口未被占满
        data = dataList[pointer]
        checksum = gbn.getChecksum(data)
        if pointer < len(dataList) - 1:
            sender.packets[sender.next_seq] = sender.make_pkt(sender.next_seq, data,
                                                                stop=False)
        else:
            sender.packets[sender.next_seq] = sender.make_pkt(sender.next_seq, data,
                                                                stop=True)
        print('Sender send packet:', pointer)
        sender.udp_send(sender.packets[sender.next_seq])
        sender.next_seq = (sender.next_seq + 1) % 256
        pointer += 1
    flag = sender.wait_ack()
    if pointer >= len(dataList):
        break

```

2.3 发送端根据接收端返回的 **ACK** 调整窗口的 **base**，若到达了窗口的最大值位置则关闭计时器，否则设置等待继续。如果超时，则会重新发送此序列号和之后的所有分组，若连续十次超时则 **count** 计数器会标志使得程序中断：

```

def wait_ack(self):
    self.sender_socket.settimeout(self.timeout)

    count = 0
    while True:

```

```

self.send_base = max(self.send_base, (ack_seq + 1) % 256)
if self.send_base == self.next_seq: # 已发送分组确认完毕

    self.sender_socket.settimeout(None)
    return True

```

```

except socket.timeout:
    # 超时，重发分组。
    print('Sender wait for ACK timeout.')
    for i in range(self.send_base, self.next_seq):
        print('Sender resend packet:', i)
        self.udp_send(self.packets[i])
    self.sender_socket.settimeout(self.timeout) # reset timer
    count += 1

```

3、gbn_server.py 包：主要就是接受发送过来的数据，并写入生成文件，其中的 wait_data() 函数用来接收数据包

在 GBN 接收端存在“期望收到的序列号”，若收到了预期所要的序列号分组，就会留下这个分组并预期序列号自增 1，并返回此序列号的 ACK；否则直接丢弃并返回上一个序列号的 ACK。

```

fp = open(SERVER_DIR + '/' + str(int(time.time())) + '.jpg', 'ab')
reset = False
while True:
    data, reset = receiver.wait_data()
    print('Data length:', len(data))
    fp.write(data)
    if reset:
        receiver.expect_seq = 0
        fp.close()
        break

```

```

def wait_data(self):
    """
    接收方等待接受数据包
    """
    self.receiver_socket.settimeout(self.timeout)

    while True:
        try:
            data, address = self.receiver_socket.recvfrom(BUFFER_SIZE)
            self.target = address

            seq_num, flag, checksum, data = self.analyse_pkt(data)
            print('Receiver receive packet:', seq_num)

```

```
# 收到期望数据包且未出错
if seq_num == self.expect_seq and getChecksum(data) == checksum:
    self.expect_seq = (self.expect_seq + 1) % 256
    ack_pkt = self.make_pkt(seq_num, seq_num)
    self.udp_send(ack_pkt)
    if flag: # 最后一个数据包
        return data, True
    else:
        return data, False
else:
    ack_pkt = self.make_pkt((self.expect_seq - 1) % 256, self.expect_seq)
    self.udp_send(ack_pkt)
    return bytes('', encoding='utf-8'), False
```

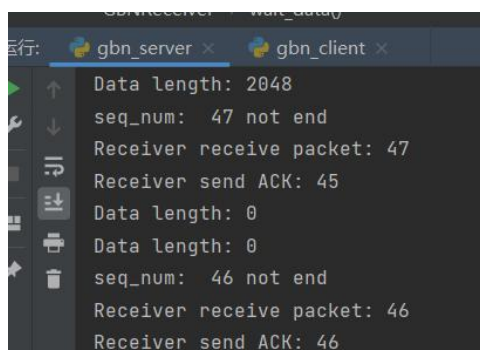
```
except socket.timeout:
    return bytes('', encoding='utf-8'), False
```

4、模拟丢失数据。为了完成这一目标，我们可以在发送端利用一个随机数库来指定不发送特定数据包；当然我们也可以在接收端模拟丢失 ACK：

```
def udp_send(self, pkt):
    if self.loss_rate == 0 or random.randint(0, int(1 / self.loss_rate)) != 1:
        self.sender_socket.sendto(pkt, self.address)
    else:
        print('Packet lost.')
    time.sleep(0.2)
```

```
def udp_send(self, pkt):
    if self.loss_rate == 0 or random.randint(0, 1 / self.loss_rate) != 1:
        self.receiver_socket.sendto(pkt, self.target)
        print('Receiver send ACK:', pkt[0])
    else:
        print('Receiver send ACK:', pkt[0], ', but lost.')
```

5、要实现双向传输，只要同时运行 client 和 server 两个程序即可。模拟丢失数据、发送超时的输出反馈如下：



```
gbn_server x  gbn_client x
Data length: 2048
seq_num: 47 not end
Receiver receive packet: 47
Receiver send ACK: 45
Data length: 0
Data length: 0
seq_num: 46 not end
Receiver receive packet: 46
Receiver send ACK: 46
```

```
gbn_server x gbn_client x
Sender receive ACK:ack_seq 14 expect_seq 14
SEND WINDOW: 14
Sender send packet: 15
Sender send packet: 16
Packet lost.
Sender send packet: 17
Sender receive ACK:ack_seq 15 expect_seq 15
SEND WINDOW: 15
Sender receive ACK:ack_seq 15 expect_seq 16
SEND WINDOW: 15
Sender wait for ACK timeout.
Sender resend packet: 16
Sender resend packet: 17
Sender receive ACK:ack_seq 16 expect_seq 16
SEND WINDOW: 16
Sender receive ACK:ack_seq 17 expect_seq 17
SEND WINDOW: 17
```

四、改造 GBN 实现双向传输和丢包模拟

1、双向传输

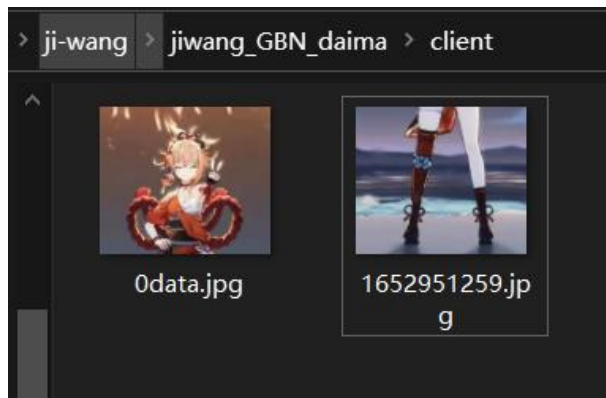
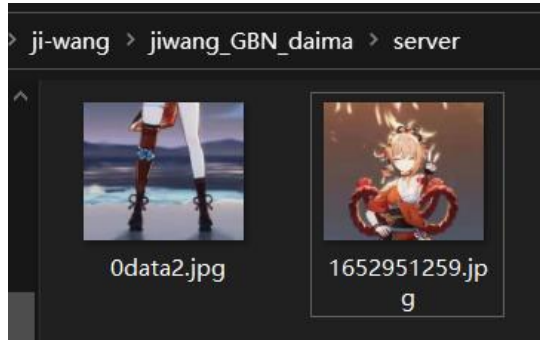
首先整合 client 和 server 到一个文件，根据上面描述的实现逻辑，再采用添加线程的方式实现单向可靠数据传输，如下图是传输成功的样子：



为了实现双向传输，我们可以多添加几个线程来实现 client 和 server 之间的信息交流。刚有这样的想法时，我简单地添加线程之后开始测试，但是发现这会导致传输的不流畅、传输结果十分丑陋，如下图是接收到的图片文件：



我考虑到这会不会是由于多开的几个线程之间会有收发数据的冲突，于是在网上查找了很多避免 python 中线程冲突的方法，但都没有解决我遇到的问题。回过神来才发现，我们模拟的 server/client 传输数据都是在本机运行的，应该是 server/client 设置的端口号一致使得他们接收数据的来源一样，导致了收到的数据信息冗杂、混合、混淆。于是我修改了端口号等信息，这才解决了问题：



```
HOST_Send = '127.0.0.1'
HOST_Receive = ''
PORT_Client = 8888
PORT_Server = 9999
ADDR_Send_C = (HOST_Send, PORT_Client)
ADDR_Send_S = (HOST_Send, PORT_Server)
ADDR_Receive_C = (HOST_Receive, PORT_Client)
ADDR_Receive_S = (HOST_Receive, PORT_Server)
CLIENT_DIR = os.path.dirname(__file__) + '/client'
SERVER_DIR = os.path.dirname(__file__) + '/server'

senderSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client_sender = gbn.GBNSender(senderSocket, ADDR_Send_C)
server_sender = gbn.GBNSender(senderSocket, ADDR_Send_S)

receiverSocket_Client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
receiverSocket_Server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
receiverSocket_Client.bind(ADDR_Receive_C)
receiverSocket_Server.bind(ADDR_Receive_S)
client_receiver = gbn.GBNReceiver(receiverSocket_Server)
server_receiver = gbn.GBNReceiver(receiverSocket_Client)
```

2、模拟数据丢失

为了模拟数据包丢失，我们直接在发送数据包的函数当中添加判断丢包的序列，并且为了和丢包率导致的丢包区别开来，我们添加说明如下，这段代码是添加在 `gbn.py` 这个包当中的。

最终显示的结果如下图所示。为便于观察，丢包模拟的测试前两张图用单向传输丢包进行模拟，双向传输的丢包是后两张，由于线程并行，部分输出比较繁琐复杂不好看。并且会发现在丢包发生时，程序的运行会发生卡顿，这是等待超时之后重新发送的时间。最终双向传输也能够在模拟丢包的情况下实现可靠传输，只不过花费的时间较长，验证了设计的协议的有效性。

双向传输的时候加上“收到重复确认”立即重发，这一部分的代码图如下图最后一张所示。

```
def udp_send(self, pkt):
    if (self.loss_rate == 0 or random.randint(0, int(1 / self.loss_rate)) != 1) and self.next_seq % 3 != 0:
        self.sender_socket.sendto(pkt, self.address)
    else:
        if self.next_seq % 3 == 0:
            print("这是我故意让数据包丢的，丢的Seq是", self.next_seq)
        else:
            print('Packet lost.')
        time.sleep(0.2)
```

```
Client_Send packet: 6
这是我故意让数据包丢的，丢的Seq是 6
Client_Send packet: 7
seq_num: 7 not end
Receiver receive packet: 7
Receiver send ACK: 5
Server_Received Data length: 0
Sender receive ACK:ack_seq 4 expect_seq 4
SEND WINDOW: 4
Sender receive ACK:ack_seq 5 expect_seq 5
SEND WINDOW: 5
Sender receive ACK:ack_seq 5 expect_seq 6
SEND WINDOW: 5
```

```
SEND WINDOW: 5
Server_Received Data length: 0
Sender wait for ACK timeout.
Sender resend packet: 6
seq_num: 6 not end
Receiver receive packet: 6
Receiver send ACK: 6
Server_Received Data length: 2048
Sender resend packet: 7
seq_num: 7 not end
Receiver receive packet: 7
Receiver send ACK: 7
Server_Received Data length: 2048
Sender receive ACK:ack_seq 6 expect_seq 6
SEND WINDOW: 6
```

```
Server_Send packet: 4
这是我故意让数据包丢的，丢的Seq是 4
Client_Send packet: 4
这是我故意让数据包丢的，丢的Seq是 4
Server_Send packet: 5
seq_num: 5 not end
Receiver receive packet: 5
Receiver send ACK: 3
Client_Received Data length: 0
Client_Send packet: 5
seq_num: 5 not end
Receiver receive packet: 5
Receiver send ACK: 3
Server_Received Data length: 0
Sender receive ACK:ack_seq 2 expect_seq 3
SEND WINDOW: 2
```

```
Sender_resend_packet_r: 3
seq_num: 3 not end
Receiver receive packet: 3
Receiver send ACK: 3
Server_Received Data length: 0
Sender_resend_packet_r: 4
seq_num: 4 not end
Receiver receive packet: 4
Receiver send ACK: 4
Client_Received Data length: 2048
Sender_resend_packet_r: 4
seq_num: 4 not end
Receiver receive packet: 4
Sender_resend_packet_r: 5
Receiver send ACK: seq_num: 5 not end
4
Receiver receive packet: Server_Received Data length: 5
2048
Receiver send ACK: 5
```

```

if self.send_base == (ack_seq + 1) % 256:
    # 收到重复确认，此处应当立即重发
    # pass
    print("self.next_seq", self.next_seq)
    print("ack_seq", ack_seq)
    for i in range(self.send_base, self.next_seq):
        print('Sender_resend_packet_r:', i)
        self.udp_send(self.packets[i])

self.send_base = max(self.send_base, (ack_seq + 1) % 256)
print("self.send_base", self.send_base)
if self.send_base == self.next_seq: # 已发送分组确认完毕

    self.sender_socket.settimeout(None)
    return True

```

五、改造成 SR 协议

1、修改代码过程

选择重传协议与 GBN 最大的不同是重新发送的时候发送的不是整个窗口，而是没有正确发送出的部分，所以我们需要知道在窗口中有哪些是已经发送的、哪些还未发送、哪些已经被正确接收。为了解决这个问题，我们通过标记来标识发过的、确认过的等等的序列号，以及用一个 List 来缓存已发送的后面的 data，发送方在模块定义方面的关键代码截图如下：

```

senderSentSet = set()
senderSentLost = set()
senderReceivedACKSet = set()
receiverReceivedSet = set()
data_save = []

```

```

print("fuck:", senderReceivedACKSet.__len__())
# self.send_base = max(self.send_base, (ack_seq + 1) % 256)
if self.send_base == ack_seq:
    while senderReceivedACKSet.__contains__(self.send_base) is True:
        self.send_base = self.send_base + 1

```

```

except socket.timeout:
    # 超时，重发分组。
    # print('Sender wait for ACK 超时.')
    # print("self.next_seq", self.next_seq)
    for i in range(self.send_base, self.next_seq):
        if senderReceivedACKSet.__contains__(i) is False:
            print('Sender 超时重传 packet:', i)
            self.udp_send(self.packets[i])
    self.sender_socket.settimeout(self.timeout) # reset timer
    count += 1

```


下面是接收方模块的部分关键代码：

```
if seq_num == self.expect_seq and getChecksum(data) == checksum:
    if len(data_save) == 0:
        self.expect_seq = (self.expect_seq + 1) % 256
    else:
        self.expect_seq = (self.expect_seq + WINDOW_SIZE) % 256
    ack_pkt = self.make_pkt(seq_num, seq_num)
    self.udp_send(ack_pkt)
    data_S = data
    # print("fffffffffffffffff===== ", len(data_save))
    if len(data_save) != 0:
        for i in range(0, len(data_save)):
            data_S += data_save[i]
        data_save.clear()
    if flag: # 最后一个数据包
        return data_S, True
    else:
        return data_S, False
else:
    data_save.append(data)
    # ack_pkt = self.make_pkt((self.expect_seq - 1) % 256, self.expect_seq)
    ack_pkt = self.make_pkt(seq_num, seq_num)
    self.udp_send(ack_pkt)
    return bytes('', encoding='utf-8'), False
```

为了方便控制丢包、接收 ACK，在 main 函数中我进行了一些修改，重新开了一个线程来进行发送方 wait_ack 函数的执行：

```
print('Client_Send packet:', pointer1)
if client_sender.next_seq_R % 11 != 20:
    client_sender.udp_send(client_sender.packets[client_sender.next_seq_R])
else:
    print("这是我故意让client数据包丢的，丢的Seq是", client_sender.next_seq_R)
client_sender.next_seq_R = (client_sender.next_seq_R + 1) % 256
pointer1 += 1
# client_sender.wait_ack()
```

```
def client_wait_ack():
    while True:
        client_sender.wait_ack()
        if client_sender.next_seq_R >= len(dataList1):
            return

def server_wait_ack():
    while True:
        server_sender.wait_ack()
        if server_sender.next_seq >= len(dataList2):
            return
```

2、修改成 SR 之后的双向传输模拟

2.1 单向传输

单向传输的话，按照 SR 协议，如果发生数据包丢失，则会在传输到窗口最大处之后停下，等到超时传输之后重传丢失的包（此处序号为 2），在 2 后面传输的数据包会缓存到 data_save 中，等到传输结束之后一起交到上一层（main 函数）。如下图所示，可能由于同时开多线程会导致不美观，但无伤大雅从图中可以看出发完 2 之后直接接着发 6，窗口在 2 停一下之后重新开始滑动。

```
Receiver send ACK_R: 5
Server_Received Data length: Sender receive ACK:ack_seq 5 expect_seq0 5
SEND WINDOW: 5
fuck: 5
self.send_base= 2 ack_seq= 5
self.next_seq= 5
Server_Received Data length: 0
Sender 超时重传 packet: 2
seq_num_R: 2 not end
Receiver receive packet_R: 2
Sender receive ACK:ack_seq 2 expect_seq 2
SEND WINDOW: 2
fuck: 6
self.send_base= 6 ack_seq= 2 self.next_seq= 6
Receiver send ACK_R: 2
没想到吧Server_Received Data length: 2048
没想到吧Server_Received Data length: Server_Send packet: 6
2048
没想到吧Server_Received Data length: 2048
没想到吧Server_Received Data length: 2048
seq_num_R: 6 not end
Receiver receive packet_R: 6
Receiver send ACK_R: 6
Server_Received Data length: 2048
```

```
self.send_base= 8 ack_seq= 7 self.next_seq= 7
Server_Send packet: 8
seq_num_R: 8 end
Receiver receive packet_R: 8
Receiver send ACK_R: 8
Server_Received Data length: 998
Sender receive ACK:ack_seq 8 expect_seq 8
SEND WINDOW: 8
fuck: 9
self.send_base= 9 ack_seq= 8 self.next_seq= 8

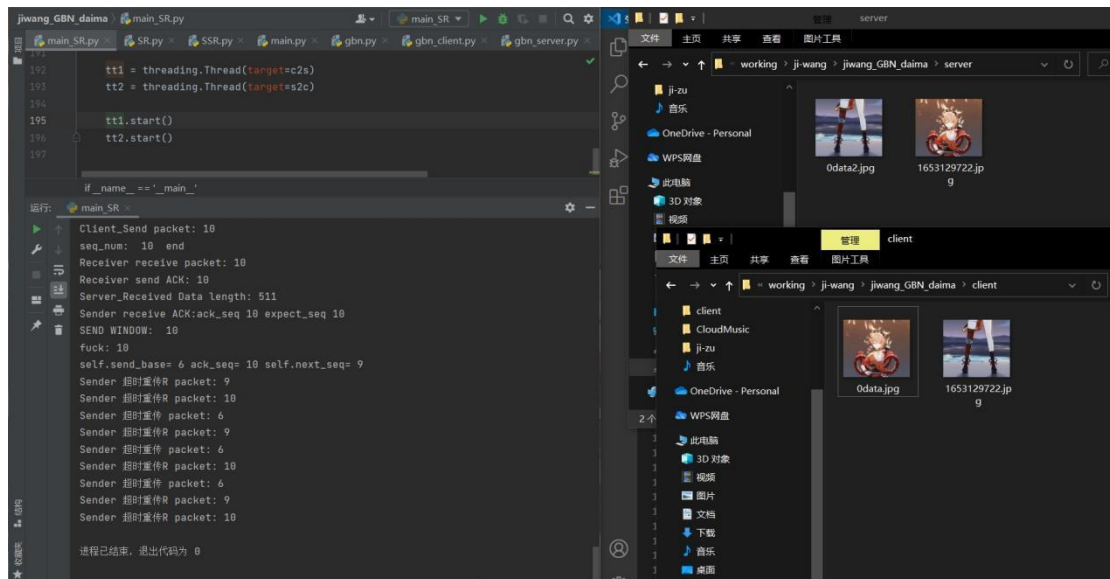
进程已结束，退出代码为 0
```



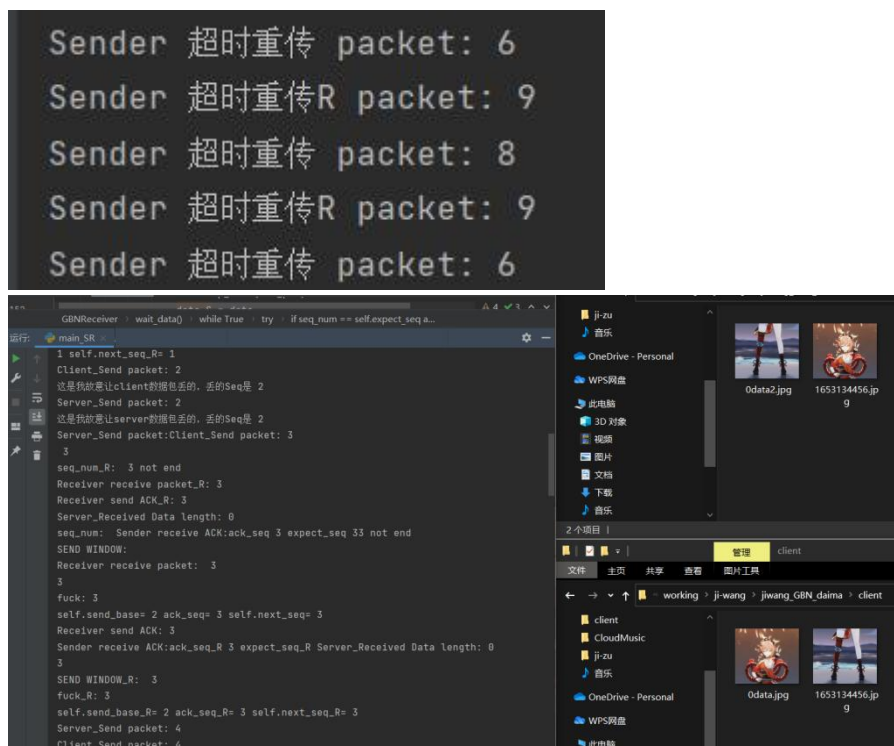
2.2 双向传输

在单向传输的过程中，不论是完全成功传输，还是模拟数据包丢失，都能很好地达到功能。但是直接套到双向传输之后却发生了一些很奇怪的反应，于是我考虑有可能是再写的 SR.py 包对于 client 和 server 来说是同一块地址，对于这里

面缓存数据、执行位置等会有冲突，于是重新再多写了一个 **SSR.py** 来区分。令人发笑的是，这样刚改完仍然不能进行无丢失传输，但是在一顿饭的时间之后他又可以运行了——期间没有更改代码。成功截图如下，图片名采用的是 **time** 时间，证明了同时双向传输。



丢包模拟——同时设置 **client** 和 **server** 丢包的序列号，观察丢包序列号位置的相关信息，到了丢包等待超时之后会发生重传。在下列图中的输出标志可能有些混乱，但是可以看出发送方、接收方都在等待超时之后重传了丢失的数据包，然后加上缓存的数据包统一交给上层。在最后的一连串“超时重传”提醒之后，程序中断、退出，这是在 **SR.py** 中定义了的——**count** 连续超时三次说明接收方已断开，终止程序。



```

Server_Received Data length: 0Sender 超时重传 packet:
  Sender 超时重传R packet: 2
2seq_num:
2 not end
seq_num_R: 2 Receiver receive packet: 2
not end
Receiver receive packet_R: 2Receiver send ACK:Sender receive ACK:ack_seq 2

2 expect_seq 2
SEND WINDOW: 2
fuck: 6
self.send_base= 6 焯Server_Received Data length: 2048
焯Server_Received Data length:ack_seq= 2048
焯Server_Received Data length: 2048Server_Send packet: 6

2焯Server_Received Data length:Receiver send ACK_R: 2
Sender receive ACK:ack_seq_R 2 2048
没想到吧Server_Received Data length: 2048
没想到吧Server_Received Data length: 2048
没想到吧Server_Received Data length: expect_seq_Rself.next_seq= 2
SEND WINDOW_R: 2
fuck_R: 62048
没想到吧Server_Received Data length:
62048self.send_base_R= 6

```

```

⇅  ack_seq_R= 2 self.next_seq_R= 6
🖨 Client_Send packet: 6
🗑 seq_num_R: 6 not end
Receiver receive packet_R: 6
seq_num: 6 not end
Receiver receive packet: 6
Receiver send ACK_R:Sender receive ACK:ack_seq_R 6 expect_seq_R 66
SEND WINDOW_R: 6
fuck_R: 7
self.send_base_R= 7 ack_seq_R= 6 self.next_seq_R= 6

Server_Received Data length: 2048
Server_Send packet: 7
seq_num_R: 7 not end
Receiver receive packet_R: 7
Sender receive ACK:ack_seq_R 6 expect_seq_R 6
SEND WINDOW_R: 6
fuck_R: 7
self.send_base_R= 7 ack_seq_R= 6 self.next_seq_R= 6
Sender receive ACK:ack_seq_R 7 expect_seq_R 7
SEND WINDOW_R: 7
fuck_R: 8
self.send_base_R= 8 ack_seq_R= 7 self.next_seq_R= 6
Receiver send ACK_R: 7

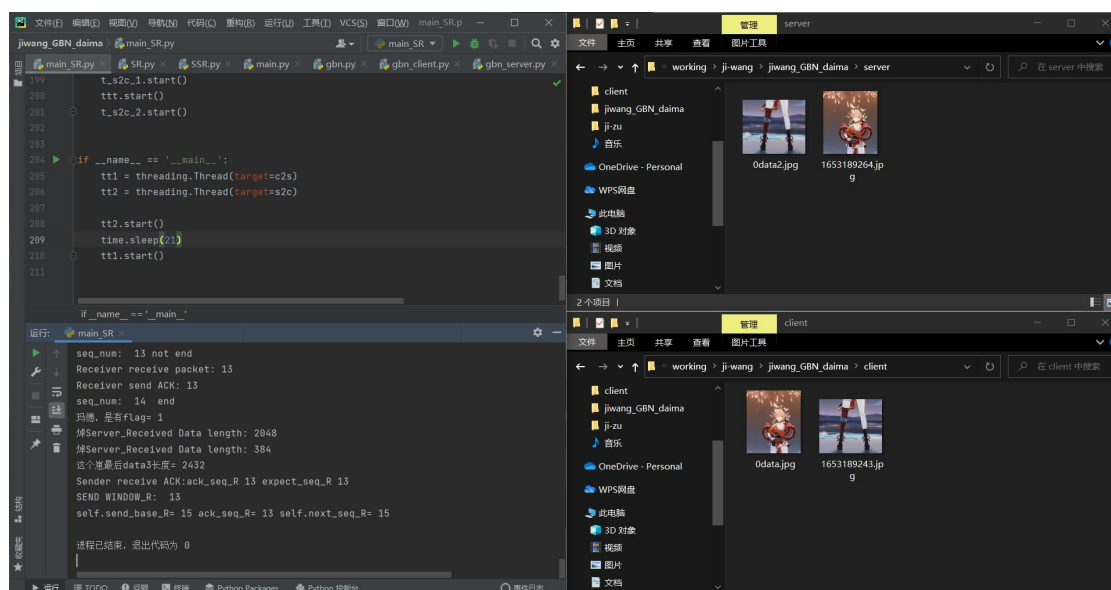
```


2.3 调试与找 BUG

为了使得得到的协议更具有适用性，我尝试了很多种不同的图片数据，发现如果图片太大则会很耗费时间、某些数据最后上交的数据包有缺失等等小问题，于是我通过修改部分代码来克服这些问题，部分代码如下，具体代码还请参照代码文件：

```
def run4_cr():
    client_fp_r = open(CLIENT_DIR + '/' + str(int(time.time())) + '.jpg', 'ab')
    # reset4 = False
    while True:
        data4, reset4 = client_receiver.wait_data()
        if len(data4) / 2048 > 0.0:
            for i in range(0, int(len(data4) / 2048) + 1):
                if int(len(data4) / 2048) > 0 and i <= int(len(data4) / 2048) - 1 != 0:
                    d = data4[i * 2048:(i + 1) * 2048]
                    client_fp_r.write(d)
                else:
                    d = data4[i * 2048:]
                    if len(d) != 0:
                        client_fp_r.write(d)
            print('没想到吧Client_Received Data length:', len(d))
            # client_fp_r.write(d)
        # else:
        #     print('客户端_Received Data length:', len(data4))
        #     client_fp_r.write(data4)
        if reset4:
            print("data4长度=", len(data4))
            client_receiver.expect_seq = 0
            client_fp_r.close()
            break
```

除此之外，我还发现，单向传输的话不论数据如何，这样修改之后都能够很好地完成“同时传输”的任务，但是如果两边同时进行的话偶尔会发生线程冲突，发生这样冲突的原因应该是 python 同时开多个线程导致的。为了规避这个瑕疵，我直接先跑一个线程然后经过一段时间 `time.sleep` 之后再开启另一个线程，效果如下。当然，模拟中的丢包率比日常生活中两主机之间正常通信的丢包率要高很多，传输效率较低，故没有模拟很夸张的丢包率。



六、实验心得

完成本次 PJ 的过程相比于平常的 LAB 较为坎坷，主要是进行了 GBN 协议的模拟、双向传输、丢包模拟，而且还进行了协议的修改——改成 SR 协议，并在此基础上实现单向可靠数据传输、双向传输、丢包模拟。

从刚看到 PJ 的时候不太理解题意，到后面的不断调试，都需要我的耐心、恒心，在传输数据、模拟丢包的时候为了节约时间我截取的图片大小都比较小，否则会耗费相当时间。一步一个脚印，不断尝试——在 GBN 协议、SR 协议的模拟传输过程中，我首先是尝试单向传输能否成功，然后模拟单向传输的丢包，再是双向传输，最后是双向传输的丢包模拟，尽量控制变量一点一点 debug，不断尝试数据以达到尽量完善。

关于多个线程同时开启可能有冲突的情况，我猜测这是因为程序运行用的同一片内存，用到了临时工作栈。可以说，我很大一部分时间花在了调试和找 bug 找问题，同时也对程序员的 debug 工作有了更深的了解。