復旦大學

PJ：GBN 实现

【设计目的】

进一步深入理解传输层协议：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（SR）协议，最后添加拥塞控制。

【具体要求】

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）在此基础上改进并实现 SR 协议。（选做）
3）模拟引入数据包的丢失，验证所设计协议的有效性。
扩展：在此基础上可以根据自己对于传输层协议的理解增加更多功能——拥塞控制

【代码分析】

1、基础功能实现

检验和计算函数

```python
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: 字节串
    """
    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
```

用于发送方和接收方对于数据的检验和计算,发送方将数据打包前计算出数据的检验和附在包中一同发送,接收方收到数据后用该函数对收到的数据再次计算并将结果和包中的检验和对比从而验证数据是否错误。

- GBNSender 类中:

类的参数：

```
class GBNSender:
    def __init__(self, senderSocket, address, who, timeout=TIMEOUT,
                 windowSize=WINDOW_SIZE, lossRate=LOSS_RATE):
        self.sender_socket = senderSocket
        self.timeout = timeout
        self.address = address
        self.window_size = windowSize
        self.loss_rate = lossRate
        self.send_base = 0
        self.next_seq = 0
        self.packets = [None] * 256
        self.who = who
```

其中 sender_socket 是发送套接字，timeout 是超时等待时间，address 是目标地址，window_size 是发送窗口大小，loss_rate 是丢包率，send_base 是发送窗口的最前端，next_seq 是下一个要发送的数据包的序号，packet 是存储待数据的列表，who 是表明该类的对象的身份标识，根据 who 我们可以在终端区分打印内容是谁发出的。

包发送函数：

```
    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(self.who, ' Packet lost.')
        time.sleep(0.2)
```

用于把数据包发送给制定的接收方套接字，函数里包含接收方的地址，同时，通过生成随机数实现给定概率的丢包模拟，在后续的代码运行过程中我们可以看到丢包模拟的效果。

数据打包函数：

```
    def make_pkt(self, seqNum, data, checksum, stop=False):
        """
        将数据打包
        """
        flag = 1 if stop else 0
        return struct.pack('BBB', seqNum, flag, checksum) + data
```

用于把序号、数据、检验和代表是否为最后一包的 stop 标志打包，用于后续的套接字发送。

包分析函数：

```python
def analyse_pkt(self, pkt):
    """
    分析数据包
    """
    ack_seq = pkt[0]
    expect_seq = pkt[1]
    return ack_seq, expect_seq
```

用于对接受到的来自接收方的 ACK 确认包的解析，获取 ACK 序号和预期序号

- GBNReceiver 类中：

类的参数：

```python
class GBNReceiver:
    def __init__(self, receiverSocket, who,timeout=10, lossRate=0,windowSize = WINDOW_SIZE+7):
        self.receiver_socket = receiverSocket
        self.timeout = timeout
        self.loss_rate = lossRate
        self.window_size = windowSize
        self.expect_seq = 0
        self.target = None
        self.who = who
```

receiver_socket 是接收套接字，windows_size 是接收窗口大小，它在 GBN 实现中不使用，在 SR 实现中使用，expect_seq 是接受方期望收到的数据包的序号，用于告知发送方，target 记录发送方的地址，用于回复 ACK 时作为返回地址使用，who 和 GBNSender 类中的 who 作用相同。

包分析函数：

```python
def analyse_pkt(self, pkt):
    '''
    分析数据包
    '''
    # if len(pkt) < 4:
    # print 'Invalid Packet'
    # return False
    seq_num = pkt[0]
    flag = pkt[1]
    checksum = pkt[2]
    data = pkt[3:]
    if flag == 0:
        print(self.who," seq_num: ",seq_num, "not end ")
    else:
        print(self.who," seq_num: ", seq_num, " end ")
    return seq_num, flag, checksum, data
```

用于对收到的数据包进行分析，获取其中的序列号、检验和、数据、代表是否为最后一包的 flag 标志位。

打包函数:

```python
def make_pkt(self, ackSeq, expectSeq):
    """
    创建ACK确认报文
    """
    return struct.pack('BB', ackSeq, expectSeq)
```

用于对 ACK 序列号和预期序列号进行数据打包，生成 ACK 确认报文。

包发送函数:

```python
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(self.who,' Receiver send ACK:', pkt[0])
    else:
        print(self.who,' Receiver send ACK:', pkt[0], ', but lost.')
```

用于把生成的 ACK 确认包发送给发送方。

2、GBN 实现代码

根据上面的基础功能代码编写实现 GBN 协议可靠数据传输：

- 发送过程函数

```python
def SendProcess(sender,fp,who):
    dataList = []
    while True:                        # 把文件夹下的数据提取出来
        data = fp.read(2048)
        if len(data) <= 0:
            break
        dataList.append(data)
    print(who,' The total number of data packets: ', len(dataList))

    pointer = 0
    while True:
        while sender.next_seq < (sender.send_base + sender.window_size):
            if pointer >= len(dataList):
                break
            #发送窗口为被占满
            data = dataList[pointer]

            checksum = getChecksum(data)
```

```python
            checksum = getChecksum(data)

            if pointer < len(dataList) - 1:
                sender.packets[sender.next_seq] = sender.make_pkt(sender.next_seq, data, checksum,
                                                                  stop=False)
            else:
                sender.packets[sender.next_seq] = sender.make_pkt(sender.next_seq, data, checksum,
                                                                  stop=True)
            print(who,' 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
    fp.close()
```

先定义一个列表 dataList，根据参数 fp 每次读取待发送文件中的 2048 位装入 dataList 中，这样讲要发送数据就分成了大小位 2048 的块，用于后续装入美如数据包。

定义 pointer 用于作为指示发送位置的指针，然后执行循环体，不断对发送窗口内的数据计算检验和并打包发送给接收方，然后调用 sender.wait_ack()进入等待 ACK 回复阶段，若 pointer 指向 dataList 的最后一位，则说明数据发送完，跳出循环，最后关闭打开的文件。

下面分析 sender.wait_ack()：

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

    count = 0
    while True:
        if count >= 10:
            # 连续超时10次，接收方已断开，终止
            break
        try:
            data, address = self.sender_socket.recvfrom(BUFFER_SIZE)

            ack_seq, expect_seq = self.analyse_pkt(data)
            print(self.who,' Sender receive ACK:ack_seq', ack_seq, "expect_seq",expect_seq)
            print(self.who," SEND WINDOW: ",ack_seq)
            if (self.send_base == (ack_seq + 1) % 256):
                # 收到重复确认，此处应当立即重发
                pass
```

```python
                # for i in range(self.send_base, self.next_seq):
                #     print('Sender resend packet:', i)
                #     self.udp_send(self.packets[i])

                self.send_base = max(self.send_base, (ack_seq + 1) % 256) ##窗口滑动
                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.              ##回退N步
                print(self.who,' 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
    return False
```

首先设置一个等待时间，后续如果超时会触发发送方重传数据包。
设置超时计数器 count，若超时次数达到 10 次即认为接收方已断开，会终止等待。
循环体中接收来自接收方发回的 ACK 确认报文，分析数据包获取 ACK 序号和接收方下一步希望收到的序列号，同时将这两个数据打印出来。
若 ACK 序号小于现在发送方窗口下限 send_base，说明是对已被确认过的数据包的确认，pass 忽略即可；否则，用当前 send_base 和 ack_seq+1 中较大者更新 send_base，这一点依赖于 GBN 协议使用的是累计确认，收到 ack_seq 则说明其前面的所有数据包已成功接收；如果 send_base = next_seq，则说明数据包已发送完，此时结束函数。
如果发生超时，则要触发重传机制，GBN 使用的是回退 N 帧，即一旦发生超时，则将 send_base 以后所有发送过的数据包全部再发送一遍，这一步即为**回退N帧**，同时超时计数器 count 加一。

- 接收过程函数：

```
def ReceiveProcess(receiver, fp,who):
    reset = False
    while True:
        data, reset = receiver.wait_data()
        print(who,' Data length:',len(data))
        fp.write(data)
        if reset:
            receiver.expect_seq = 0
            fp.close()
            break
```

接收过程调用 receiver.wait_data()接收发送方发来的数据包，把收到的数据写入打开的文件中，根据 receiver.wait_data()返回的 reset 来判断文件是否完毕，若是则关闭打开的文件并结束函数。

下面分析函数 receiver.wait_data()：

```
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(self.who,' Receiver receive packet:', seq_num)
            # 收到期望数据包且未出错
```

```
            # 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    #向上层递交数据块
                else:
                    return data, False
            else:
                ack_pkt = self.make_pkt((self.expect_seq - 1) % 256, self.expect_seq)   #重复确认，让客户回退N步重传
                self.udp_send(ack_pkt)
                return bytes('', encoding='utf-8'), False

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

首先设置超市等待时间，然后在循环体中等待接收来自发送方的数据包，然后分析数据包获取序号 seq_num、标志位 flag、检验和 checksun 和数据 data。判断数据包序号如果等于希望收到的序列号且数据无误，则 expect_seq 加一并发送 ACK 确认，然后返回数据和布尔值，布尔值有 flag 决定，标志着是否是最后一个数据包；如果数据包序号不等于希望收到的序列号或者数据有误，则将数据包直接丢弃并发送重复确认 ACK 告知发送方。

3、SR 实现代码

基础代码和 GBN 使用的是一样的。

- 发送过程函数（SR）：

```
356 def SendProcess_SR(sender,fp,who):
357     senderSendSet = []
358     senderReceivedACKSet = []
359     dataList = []
360     while True:                         # 把文件夹下的数据读取出来
361         data = fp.read(2048)
362         if len(data) <= 0:
363             break
364         dataList.append(data)
365     print(who,' The total number of data packets: ', len(dataList))
366 
367     # #放到主函数里去做
368     for i in range(0, 100000):
369         senderSendSet.append(0)          # 初始设置已发送记录全为零
370         senderReceivedACKSet.append(0)  # 初始设置已接收ACK记录全为零
371 
372     pointer = 0
373     while True:
374         while sender.next_seq < (sender.send_base + sender.window_size) and senderSendSet[sender.next_seq] == 0:
375             if pointer >= len(dataList):
376                 break
377             # 发送窗口为被占满
378             data = dataList[pointer]
379 
380             checksum = getChecksum(data)
381 
382             if pointer < len(dataList) - 1:
383                 sender.packets[sender.next_seq] = sender.make_pkt(sender.next_seq, data, checksum,
384                                                                   stop=False)
385             else:
386                 sender.packets[sender.next_seq] = sender.make_pkt(sender.next_seq, data, checksum,
387                                                                   stop=True)
388             print(who,' Sender send packet:', pointer)
389             sender.udp_send(sender.packets[sender.next_seq])
390             senderSendSet[sender.next_seq] = 1                          # 记录已发送的分组编号
391             sender.next_seq = (sender.next_seq + 1) % 256
392             pointer += 1
393         flag = sender.wait_ack_SR(senderSendSet, senderReceivedACKSet)
394         if pointer >= len(dataList):
395             break
396     fp.close()
```

SR 和 GBN 的不同在于它是选择重传机制，当发生等待超时时，它不会发送 send_base 之后所有发送过的数据包，而是只发送没有收到 ACK 确认的数据包，这就要求发送方记录发送的数据包序号和收到的 ACK 的序号，所以在上面函数中我设置了 senderSendSet 用于记录已发送数据包的序号，设置 senderReceivedACKKSet 用于记录收到的 ACK 的序号。没发送一个数据包，把 senderSendSet 对应位设置为 1 表示该数据包已发送，然后调用 sender.wait_ack_SR()来等待接收方回复。sender.wait_ack_SR()也是 SR 与 GBN 不同的地方。

下面分析 sender.wait_ack_SR()：

```python
    def wait_ack_SR(self, senderSentSet, senderReceivedACKSet):
        self.sender_socket.settimeout(self.timeout)

        count = 0
        while True:
            if count >= 10:
                # 连续超时10次，接收方已断开，终止
                break
            try:
                data, address = self.sender_socket.recvfrom(BUFFER_SIZE)

                ack_seq, expect_seq = self.analyse_pkt(data)
                print(self.who,' Sender receive ACK:ack_seq', ack_seq, "expect_seq", expect_seq)
                print(self.who," SEND WINDOW: ", ack_seq)
                senderReceivedACKSet[ack_seq] = 1     # 记录收到的ACK
                print(self.who," 收到ACK:",ack_seq,'并记录')
                if (self.send_base == (ack_seq + 1) % 256):
                    # 收到重复确认，此处应当立即重发
                    pass
                    #self.window_size += 1

                if(ack_seq == self.send_base):
                    print(self.who,' 收到ACK:',ack_seq)
                    while(senderReceivedACKSet[self.send_base] == 1):
                        self.send_base = (self.send_base + 1) % 256   # 窗口滑动
                        # self.send_base = max(self.send_base, (ack_seq + 1) % 256)  ##窗口滑动
                    if(self.window_size < 10):
                        self.window_size += 1
                        print(self.who,' 发送窗口+1，现在为：',self.window_size)
                    print(self.who,' 当前窗口 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

            except socket.timeout:
                # 超时，重发分组.              ## 选择重传
                print(self.who,' 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])
                for i in range(self.send_base,self.send_base + self.window_size):
                    if(senderSentSet[i] == 1 and senderReceivedACKSet[i] == 0):
                        print(self.who,' Sender resend packet:', i)
                        self.udp_send(self.packets[i])
                self.window_size = 1
                print(self.who,' 线路拥塞,发送窗口减小为1！')
                self.sender_socket.settimeout(self.timeout)  # reset timer
                count += 1
        return False
```

在该函数参数中有 senderReceivedACKSet，发送方接收来自接收方的确认报文，同时在 senderReceivedACKSet 对应位记录收到的 ACK,然后对 ACK 序号进行判断：如果收到已确认过的报文，直接 pass；如果 ack_seq > send_base 无许操作，因为 senderReceivedACKSet 已经对这个 ACK 进行了记录；如果 ack_seq = send_base 并且数据无误，则采用 while 循环向前移动,因为可能前面收到了为按序到达的 ACK，我们做了记录，现在按序的 ACK 到达之后，我们移动 send_base 应该把前面未按序到达的 ACK 连起来一并移动过去。再判断 send_base 是否等于 next_seq，若是则说明已发送完毕，直接返回结束函数。

如果发生超时，则 SR 要检查窗口内的数据包，把已发送且未收到确认的数据包

重新发送，这一步即为**选择重传**。重传之后重新设置等待时间。

- 接收过程函数（SR）

```
399  def ReceiveProcess_SR(receiver, fp, who):
400      reset = False
401      receiverReceivedSet = []   # 用于记录接收到的分组
402      buffer = []
403      for i in range(0,1000):
404          receiverReceivedSet.append(0)
405          buffer.append('')
406
407      while True:
408          data, reset = receiver.wait_data_SR(receiverReceivedSet,buffer)
409          print(who,' Data length:',len(data))
410          fp.write(data)
411          if reset:
412              receiver.expect_seq = 0
413              fp.close()
414              break
```

在 SR 协议的接受过程函数中，不同于 GBN，收到失序的数据包时不回丢弃，而是先缓存起来，当按序的数据包到达时，再一并上交给上层，所以该函数中有一个数据缓存 buffer，专门存储失序数据包内的数据，同时定义一个 receiverReceivedSet 用来记录收到的 ACK，在函数中我先给 buffer 和 receiverReceivedSet 初始化设置了大小未 1000 的空间，如果要传输的文件很大，这个空间也可以随之调大。
在 while 循环体中，调用 receiver.wait_ack_SR()接收来自发送方的数据包，再把数据写进打开的文件中。SR 接收方与 GBN 接收方的不同主要在于 receiver.wait_ack_SR()的不同。

下面分析 receiver.wait_ack_SR()：

```python
def wait_data_SR(self,receiverReceivedSet,buffer):
    """
    接收方等待接受数据包
    """
    ##下面的写到主函数里
    # receiverReceivedSet = []
    # buffer = []
    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(self.who,' Receiver receive packet:', seq_num)
            receiverReceivedSet[seq_num] = 1
```

```python
            # 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)

                for i in range(self.expect_seq, self.expect_seq + self.window_size):
                    if(receiverReceivedSet[i] == 1):
                        self.expect_seq = (self.expect_seq + 1) % 256
                        data = data + buffer[i]
                    else:
                        break
                if flag:  # 最后一个数据包
                    return data, True  # 向上层递交数据块
                else:
                    return data, False
```

```python
            elif(seq_num > self.expect_seq and seq_num <self.expect_seq+self.window_size and getChecksum(data) == checksum):
                if(flag == 0):
                    receiverReceivedSet[seq_num] = 1         # 记录已经收到
                    buffer[seq_num] = data
                    ack_pkt = self.make_pkt(seq_num, seq_num)
                    self.udp_send(ack_pkt)
                else:
                    receiverReceivedSet[seq_num] = 0         # 若是最后一个包未按序到达，则丢弃并标记为未接受过
                return bytes('', encoding='utf-8'), False
            elif(seq_num < self.expect_seq):
                ack_pkt = self.make_pkt(seq_num, seq_num)
                self.udp_send(ack_pkt)
                return bytes('', encoding='utf-8'), False
            else:
                return bytes('', encoding='utf-8'), False

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

同样先设置等待时间，然后在 while 循环体中接收发送方发来的数据包并解析数据包，与 GBN 不同的是它会记录收到的数据包的序号，即令 receiverReceivedSet 对应位为 1。

接下来对数据包的序号进行判断（SR 接收方使用了接收窗口）：

如果数据包序号等于希望收到的序号且数据无误，则首先对 expect_seq 加一并对该数据包回复 ACK，之后循环检查时候有之前收到的失序数据包可以连上，如果有就对 expect_seq 加一并把之前缓存的数据加到这次收到的数据后面，最后判断是不是最后一个数据包，返回 data 和 flag。

如果如果数据包序号大于希望收到的序号、在接收窗口内且数据无误，则要记录收到了该失序包并返回针对它的 ACK，需要注意的是，在处理之前必须先判断该失序包是不是最后一包，如果不是才执行上述处理，否则直接丢弃，这是因为如果我们收下失序的最后一包的话，当前面的包按序到达后，通过 for 循环把缓存

的失序包数据加到按序数据包的数据之后时，我们已不能再获知最后一包是最后一包了，因为我们缓存的是它的数据部分，并没有保存它的 flag。针对这一问题，解决办法就是如果收到失序的最后一包，就直接丢弃，最后一包只能按序到达，这样我们就可以获得它的 flag 从而得知这是最后一包，文件已传输完毕。
如果收到已确认的数据包，仍要返回 ACK。其他情况忽略该分组。
如果超时器超时，则直接返回结束函数运行。

- 主函数+双向传输：

在主函数内调用上面写好的那些函数并实现双向运行，首先建立两个套接字，利用这两个套接字定义两个 GBNSender 类的对象，作为双方的发送端；再建立两个套接字，利用这两个套接字定义两个 GBNReceiver 类的对象，作为双方的接收端；

```
19  senderSocket_Client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
20  senderSocket_Server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
21  sender_client = gbn.GBNSender(senderSocket_Client,ADDR_Target_C,'Client')
22  sender_server = gbn.GBNSender(senderSocket_Server,ADDR_Target_S,'Server')# 两个套接字用作各自的发送
23
24
25  receiverSocket_Client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
26  receiverSocket_Server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
27  receiverSocket_Client.bind(ADDR_Receive_C)
28  receiverSocket_Server.bind(ADDR_Receive_S)              # 两个接收套接字用于各自的接收
29  receiver_client = gbn.GBNReceiver(receiverSocket_Client,'Client')
30  receiver_server = gbn.GBNReceiver(receiverSocket_Server,'Server')
```

然后定义不同的线程分别调用发送过程函数和接收过程函数（上面时 GBN 版本的实现，下面是 SR 版本的实现）：

```
39  # 开启两个线程
40  Thread_Client_Send = threading.Thread(target=gbn.SendProcess, args=(sender_client, fpS_Client,'Client',))
41  Thread_Server_Send = threading.Thread(target=gbn.SendProcess, args=(sender_server, fpS_Server,'Server',))
42  Thread_Client_Receive = threading.Thread(target=gbn.ReceiveProcess, args=(receiver_client, fpR_Client,'Client',))
43  Thread_Server_Receive = threading.Thread(target=gbn.ReceiveProcess, args=(receiver_server, fpR_Server,'Server',))
44
45  Thread_Client_Send_SR = threading.Thread(target=gbn.SendProcess_SR, args=(sender_client, fpS_Client,'Client',))
46  Thread_Server_Send_SR = threading.Thread(target=gbn.SendProcess_SR, args=(sender_server, fpS_Server,'Server',))
47  Thread_Client_Receive_SR = threading.Thread(target=gbn.ReceiveProcess_SR, args=(receiver_client, fpR_Client,'Client',))
48  Thread_Server_Receive_SR = threading.Thread(target=gbn.ReceiveProcess_SR, args=(receiver_server, fpR_Server,'Server',))
49
50  # Thread_Client_Send.start()
51  # Thread_Server_Send.start()
52  # Thread_Client_Receive.start()
53  # Thread_Server_Receive.start()
54
55  Thread_Client_Send_SR.start()
56  Thread_Server_Send_SR.start()
57  Thread_Client_Receive_SR.start()
58  Thread_Server_Receive_SR.start()
```

4、扩展内容：拥塞控制（在 SR 版本中实现）

拥塞控制用来调整发送窗口的大小，没有拥塞控制时，发送窗口固定，网络通畅时不能多发，拥塞时也不能少发，添加拥塞控制后能让发送方更好地利用网络带宽。

首先我把 GBNReceiver 的接收窗口大小设置为 10（WINDOW_SIZE+7）：

```python
class GBNReceiver:
    def __init__(self, receiverSocket, who,timeout=10, lossRate=0,windowSize = WINDOW_SIZE+7):
        self.receiver_socket = receiverSocket
        self.timeout = timeout
        self.loss_rate = lossRate
        self.window_size = windowSize
        self.expect_seq = 0
        self.target = None
        self.who = who
```

发送方发送窗口初始设置为 3，每当收到按序到达的 ACK，就增大窗口大小，当窗口大小小于 7 时，每次加 2，当大于等于 7 时，每次加 1，最大不超过接收方的接收窗口大小：

```python
            if(ack_seq == self.send_base):
                print(self.who,' 收到ACK:',ack_seq)
                while(senderReceivedACKSet[self.send_base] == 1):
                    self.send_base = (self.send_base + 1) % 256   # 窗口滑动
                    # self.send_base = max(self.send_base, (ack_seq + 1) % 256)  ##窗口滑动
                if(self.window_size < 7):                                       # 拥塞控制
                    self.window_size += 2
                    print(self.who,' 发送窗口+2，现在为：',self.window_size)
                elif(self.window_size >=7 and self.window_size <10):          # 拥塞控制
                    self.window_size += 1
                    print(self.who, ' 发送窗口+1，现在为：', self.window_size)
                print(self.who,' 当前窗口 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
```

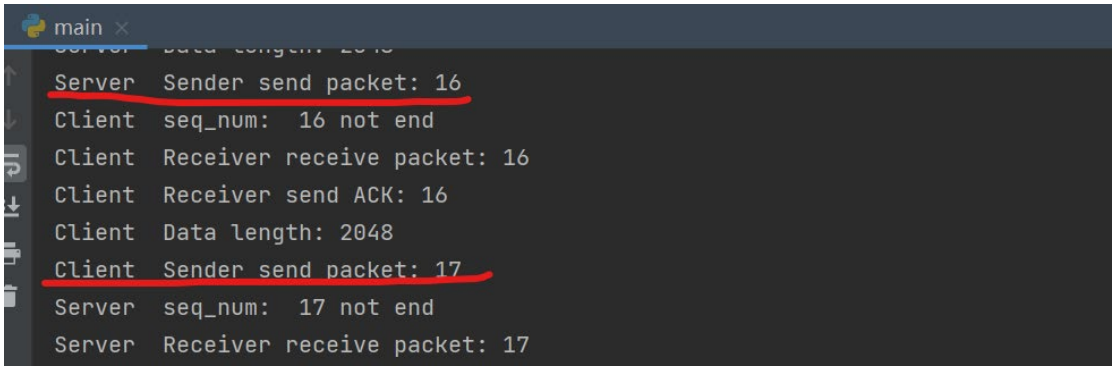
一旦发生超时，发送方选择重传之后，就把发生窗口大小改为 1，避免道路进一步拥塞：

```python
        except socket.timeout:
            # 超时，重发分组.              ## 选择重传
            print(self.who,' 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])
            for i in range(self.send_base,self.send_base + self.window_size):
                if(senderSentSet[i] == 1 and senderReceivedACKSet[i] == 0):
                    print(self.who,' Sender resend packet:', i)
                    self.udp_send(self.packets[i])
            self.window_size = 1
            print(self.who,' 线路拥塞,发送窗口减小为1！')                    #拥塞控制
            self.sender_socket.settimeout(self.timeout)  # reset timer
            count += 1
    return False
```

5、运行结果

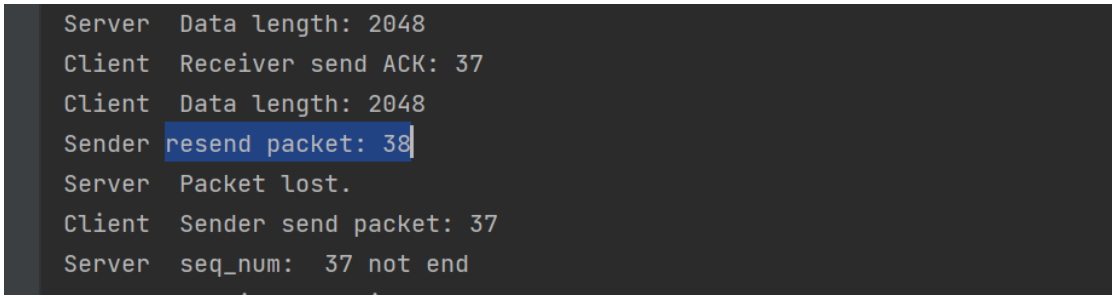
- GBN

双向传输：

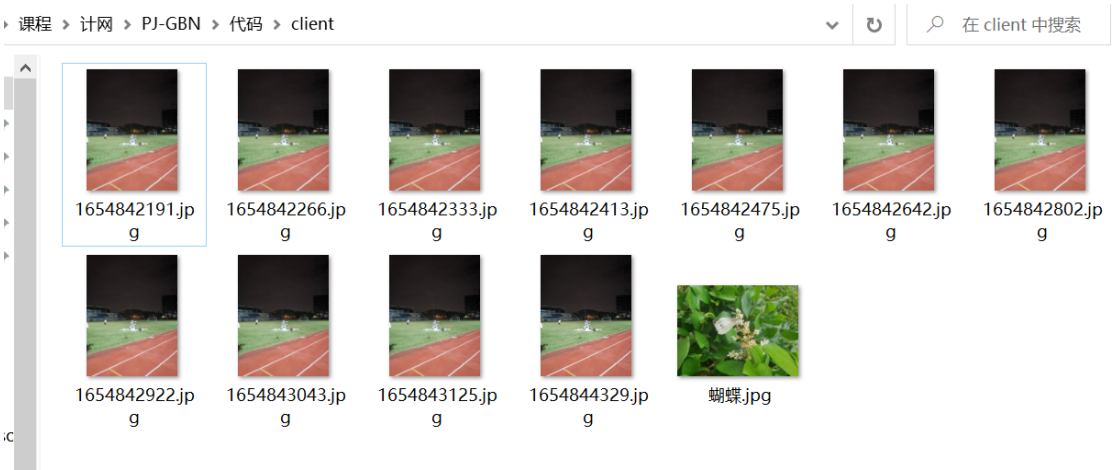


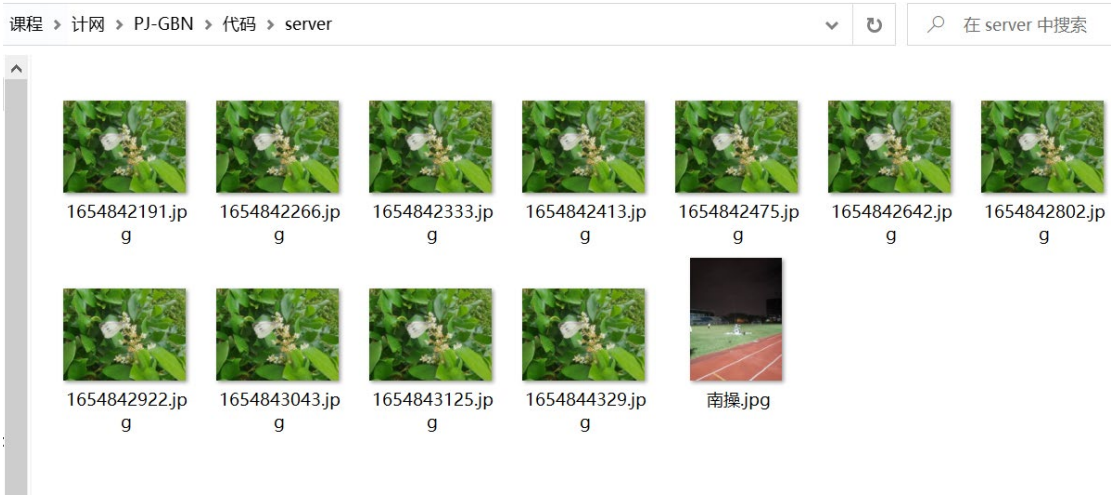
丢包模拟：

回退 N 步（GBN 用的发送窗口为 3 比较小，不太明显，运行时终端打印的内容可以看出）：



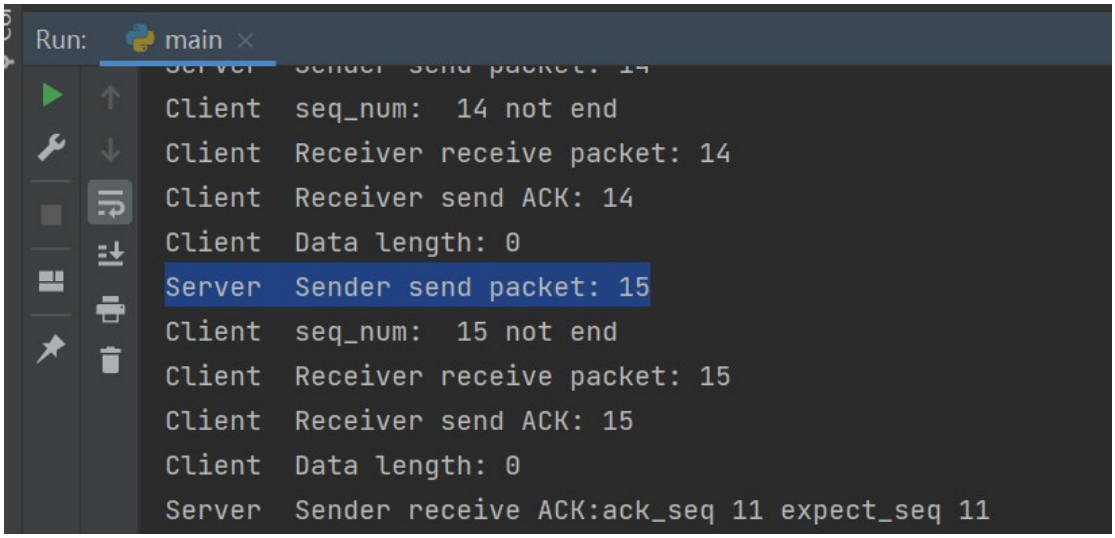
文件传输结果：

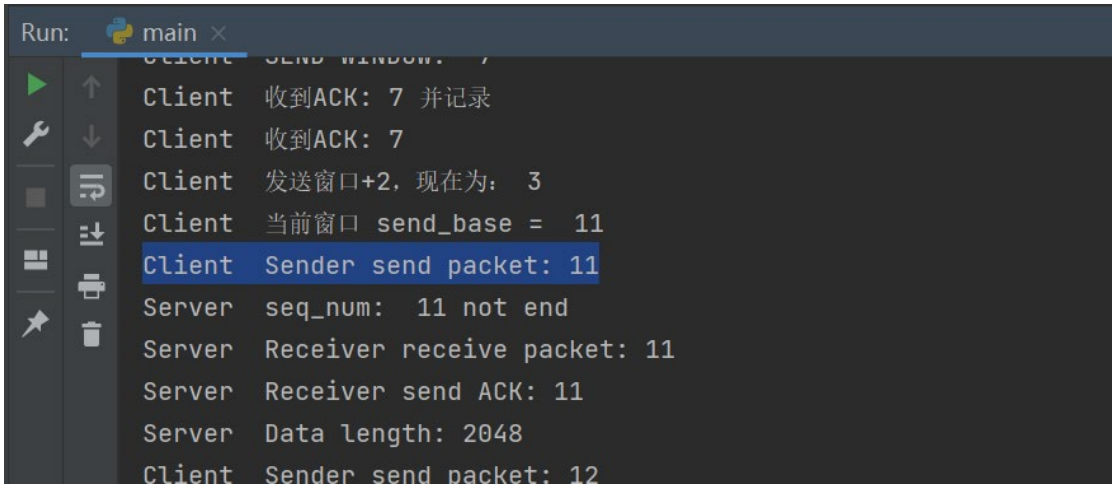




- SR+拥塞控制

双向传输（太长了一下子截不下来），去文件夹看两个正在传输的图片也能看到双向同时传输：





模拟丢包：

```
Client  Data length: 2048
Server  Sender send packet: 74
Server  Packet lost.
Server  Sender send packet: 75
Server  Packet lost.
Client  Sender wait for ACK timeout.
Client  Sender resend packet: 94
```

选择重传：

```
main
Client  seq_num:  67 not end
Client  Receiver receive packet: 67
Client  Receiver send ACK: 67
Client  Data length: 6144
Server  Sender resend packet: 70
Client  seq_num:  70 not end
Client  Receiver receive packet: 70
Client  Receiver send ACK: 70
Client  Data length: 2048
Server  Sender resend packet: 71
Client  seq_num:  71 not end
Client  Receiver receive packet: 71
Client  Receiver send ACK: 71
```

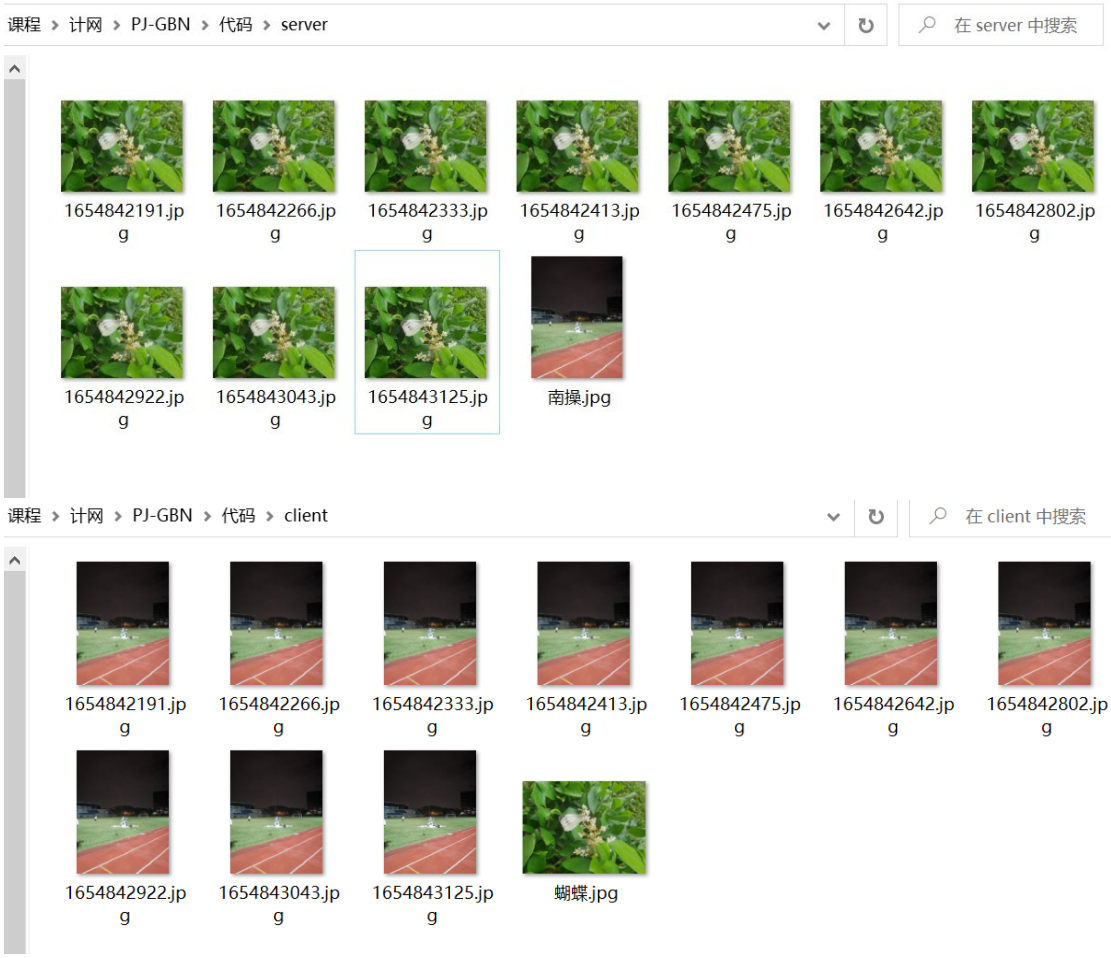
拥塞控制：

```
Server  发送窗口+2，现在为： 5
Server  当前窗口 send_base =  71
Server  Sender receive ACK:ack_seq 71 expect_seq 71
Server  SEND WINDOW:  71
Server  收到ACK: 71 并记录
Server  收到ACK: 71
Server  发送窗口+2，现在为： 7
```

```
Server  SEND WINDOW:  72
Server  收到ACK: 72 并记录
Server  收到ACK: 72
Server  发送窗口+1，现在为： 8
Server  当前窗口 send_base =  73
```

```
Client  seq_num:  71 not end
Client  Receiver receive packet: 71
Client  Receiver send ACK: 71
Client  Data length: 2048
Server  线路拥塞,发送窗口减小为1!
Server  Sender receive ACK:ack_seq 67 expect_seq 67
```

文件传输结果：


课程 › 计网 › PJ-GBN › 代码 › server
在 server 中搜索
1654842191.jpg
1654842266.jpg
1654842333.jpg
1654842413.jpg
1654842475.jpg
1654842642.jpg
1654842802.jpg
1654842922.jpg
1654843043.jpg
1654843125.jpg
南操.jpg
课程 › 计网 › PJ-GBN › 代码 › client
在 client 中搜索
1654842191.jpg
1654842266.jpg
1654842333.jpg
1654842413.jpg
1654842475.jpg
1654842642.jpg
1654842802.jpg
1654842922.jpg
1654843043.jpg
1654843125.jpg
蝴蝶.jpg