计算机网络安全 PJ

杨乙　　21307130076　　信息安全　智息技术班

注：本项目实现了 GBN 协议、双向传输、SR 协议以及拥塞控制

GBN 实现

在给出的参考代码中删除部分冗余、进行适当改动即可（完整代码见文件 “GBN” ）。程序中类或函数的作用如下：

GBNSender 类：包含发送方的部分操作，以供发送方调用。包含如下函数：

- **初始化函数**：用给定的参数初始化发送方
- **udp_send(self, pkt)**：以 0.2 秒的间隔发送数据包，并根据 `loss_rate` 值模拟一定概率的丢包
- **wait_ack(self)**：描述发送方对以下事件的响应（未要求对 ACK 包进行差错检测）：
  - 收到 ACK：若所有已发送的分组都已经确认（窗口空）则停止计时，否则开始计时
  - 超时事件：回退 N 步，重发所有已发送未确认的分组
- **make_pkt(self, seqNum, data, checksum, stop=False)**：将以下字段和数据段打包
  - seqNum：发送的分组序号
  - flag：传输完成标志
  - checkSum：数据段的检验和
- **analyse_pkt(self, pkt)**：得到 ACK 包序号、期待数据包的序号

Send 函数：读取文件，调用函数构造数据包序列并发送数据包。根据提供的参考资料，发送端先一次性发完一整个窗口内的分组，再调用 `wait_ack()` 函数等待对应序列号的 ACK 包被返回

GBNReceiver 类：包含接收方的部分操作。此类中包含如下函数：

- **初始化函数**：用给定的参数初始化接收方
- **udp_send(self, pkt)**：以 0.2 秒的间隔发送 ACK 包，并根据 `loss_rate` 值模拟一定概率的丢包
- **wait_data(self)**：描述接收方对收到数据包的响应：若数据包无差错且按序到达，则将数据包交付给上层，并发送对应的新的 ACK 包，并将期待数据包的序号加一。否则重传原来的 ACK 包
- **analyse_pkt(self, pkt)**：得到数据包的序号、传输完成标志、检验和以及数据段
- **make_pkt(self, ackSeq, expectSeq)**：将 ACK 包序号和期待数据包序号打包

Receive 函数：接收数据包，将收到的数据包写入指定文件中

getChecksum 函数：计算数据包的检验和

双向传输实现

使用多线程实现双向的同时传输。首先创建两对套接字，两个传输方向上各自的客户端和服务端使用相同的端口号进行绑定：

```
client_send_fp = open(os.path.dirname(__file__) + '/client/client_to_server.jpg', 'rb')
server_send_fp = open(os.path.dirname(__file__) + '/server/server_to_client.jpg', 'rb')
client_receive_fp = open(os.path.dirname(__file__) + '/client/' + str(int(time.time())) + '.jpg', 'ab')
server_receive_fp = open(os.path.dirname(__file__) + '/server/' + str(int(time.time())) + '.jpg', 'ab')

client_send_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_send_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client_receive_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_receive_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

clientSender = GBNSender(client_send_socket, ('127.0.0.1', 8888))
serverSender = GBNSender(server_send_socket, ('127.0.0.1', 6666))
client_receive_socket.bind(('', 6666))
server_receive_socket.bind(('', 8888))
clientReceiver = GBNReceiver(client_receive_socket)
serverReceiver = GBNReceiver(server_receive_socket)
```

再为两个传输方向上各自的客户端和服务端创建两对线程。注意先开启两个接收端的线程，再开启两个发送端的线程：

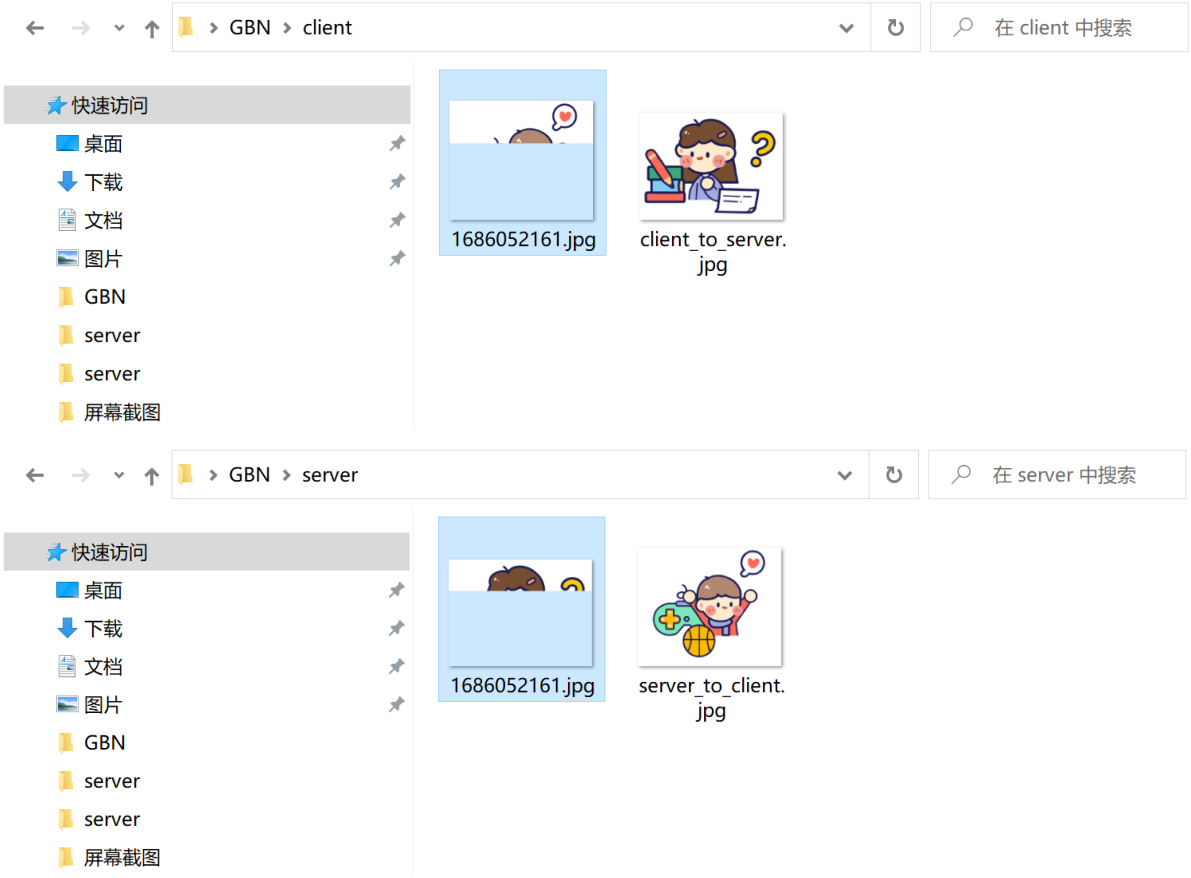
```
ClientSend = threading.Thread(target=Send, args=(clientSender, client_send_fp))
ServerSend = threading.Thread(target=Send, args=(serverSender, server_send_fp))
ClientReceive = threading.Thread(target=Receive, args=(clientReceiver, client_receive_fp))
ServerReceive = threading.Thread(target=Receive, args=(serverReceiver, server_receive_fp))

ClientReceive.start()
ServerReceive.start()
ClientSend.start()
ServerSend.start()
```

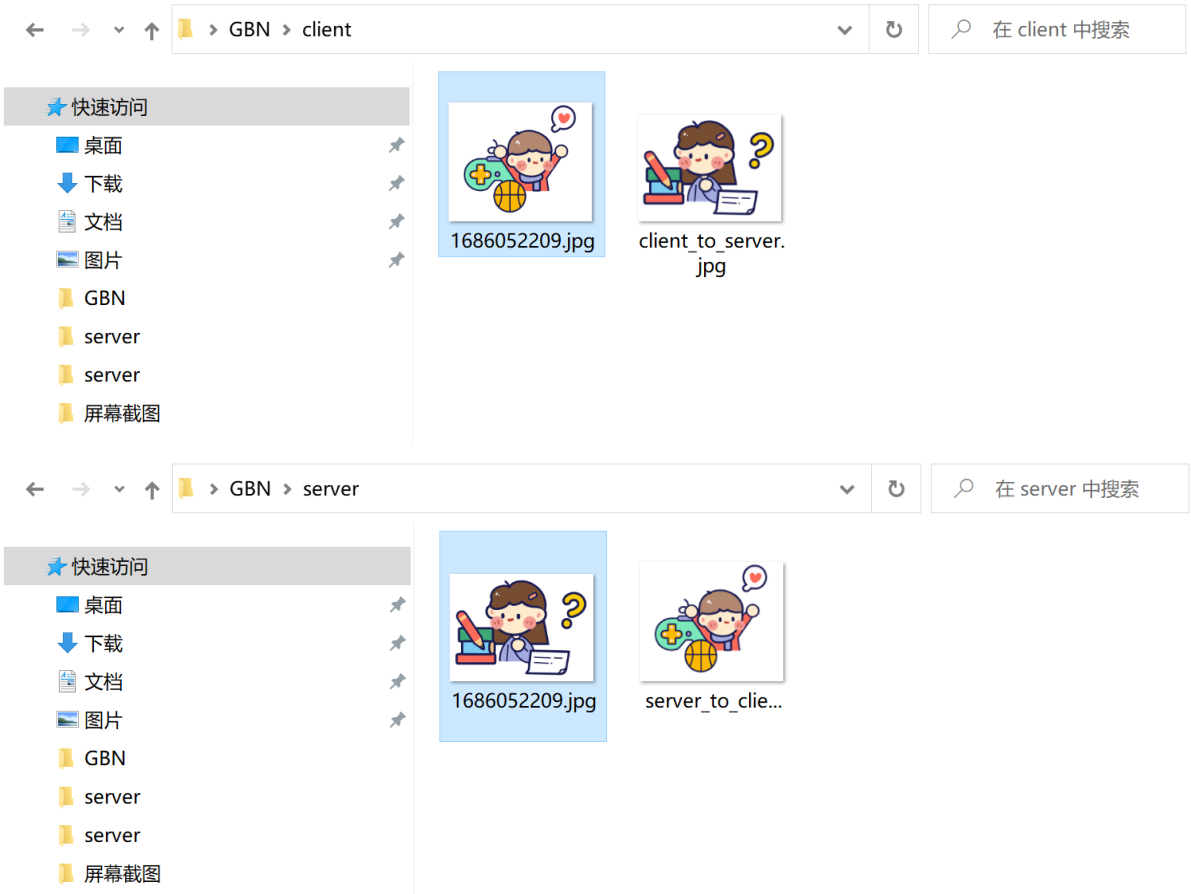
观察输出结果，可以发现实现了双向传输：

```
D:\Desktop\GBN\venv\Scripts\python.exe D:\Desktop\GBN\main.py
The total number of data packets:  74
The total number of data packets:  70
Sender send packet: 0
seq_num:  0 not end
Receiver receive packet: 0
Sender send packet: 0
seq_num:  0 not end
Receiver receive packet: 0
Receiver send ACK: 0
Data length: 2048
Receiver send ACK: 0
Data length: 2048
Sender receive ACK: ack_seq 0 expect_seq 1
SEND WINDOW: 0
Sender send packet: 1
seq_num:  1 not end
Receiver receive packet: 1
```

传输过程中，可以发现客户端、服务器端文件夹下都出现了对方传输过来的图片：



传输完成：



GBN 模拟丢包

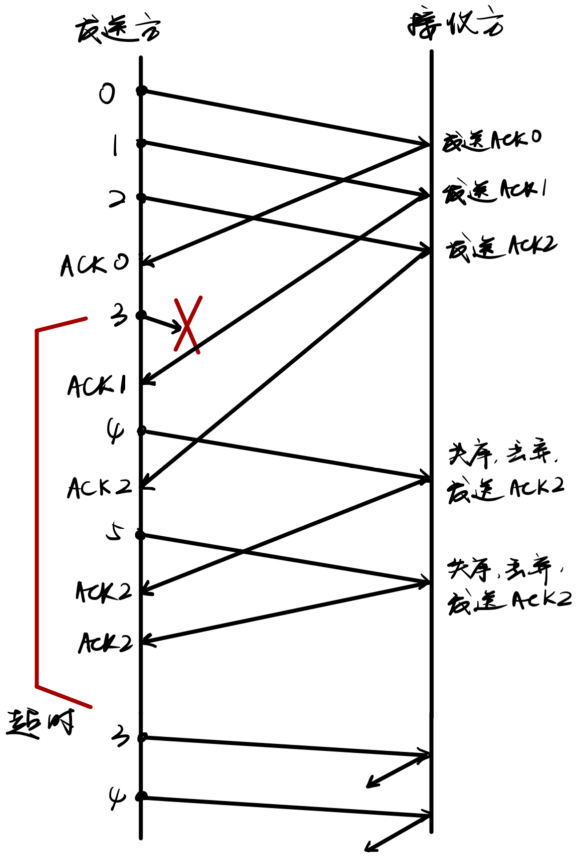
为方便观察丢包现象，仅从客户端向服务器发送数据（代码见文件中注释部分），下图记录了某次运行的输出结果：

```
D:\Desktop\GBN\venv\Scripts\python.exe D:\Desktop\GBN\main.py
The total number of data packets:  70
Sender send packet: 0
seq_num:  0 not end
Receiver receive packet: 0
Receiver send ACK: 0
Data length: 2048
Sender send packet: 1
seq_num:  1 not end
Receiver receive packet: 1
Receiver send ACK: 1
Data length: 2048
Sender send packet: 2
seq_num:  2 not end
Receiver receive packet: 2
Receiver send ACK: 2
Data length: 2048
Sender receive ACK: ack_seq 0 expect_seq 1
SEND WINDOW: 0
Sender send packet: 3
Packet lost.
```

此处数据包 3 丢失

```
Sender receive ACK: ack_seq 1 expect_seq 2
SEND WINDOW: 1
Sender send packet: 4
seq_num:  4 not end
Receiver receive packet: 4
Receiver send ACK: 2
Data length: 0
Sender receive ACK: ack_seq 2 expect_seq 3
SEND WINDOW: 2
Sender send packet: 5
seq_num:  5 not end
Receiver receive packet: 5
Receiver send ACK: 2
Data length: 0
Sender receive ACK: ack_seq 2 expect_seq 3
SEND WINDOW: 2
Sender receive ACK: ack_seq 2 expect_seq 3
SEND WINDOW: 2
Sender wait for ACK timeout.
Sender resend packet: 3
seq_num:  3 not end
Receiver receive packet: 3
Receiver send ACK: 3
Data length: 2048
Sender resend packet: 4
seq_num:  4 not end
Receiver receive packet: 4
Receiver send ACK: 4
```

根据运行结果，下图描述了窗口长度为 3 的 GBN 的运行情况：



可以发现，分组 3 丢失后，分组 4、5 被当作失序分组丢弃，直到分组 3 超时后进行重传，符合 GBN 协议（当窗口大小设置为 1 时，GBN 协议就是停等协议）

SR 实现

SR 的实现可以在 GBN 的代码基础上对以下几处进行更改（完整代码见文件 “SR” ）：

SRSender 类：在 `GBNSender` 类的基础上进行改动：

- 增加已发送包列表 `already_sent`，通过将索引对应元素设为 1 记录已发送包的序号，用于指示收到 ACK 且窗口移动后窗口内的未发送分组
- 增加已确认包列表 `received_ack`，通过将索引对应元素设为 1 记录已确认包的序号，因为窗口移动时需要将基序号移到具有最小序号的未确认分组处
- 修改 `wait_ack` 函数：修改后的 `wait_ack` 函数描述发送方对收到 ACK 和超时事件的响应。核心代码注释如下：

```
    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 = self.analyse_pkt(data)
                print('Sender receive ACK', ack_seq)
                if self.send_base <= ack_seq <= self.send_base +
self.window_size:
                    # 收到ACK，若该分组序号在窗口内，则将被确认的分组标记为已接收
                    self.received_ack[ack_seq] = 1
                if self.send_base == ack_seq:
                    # 若分组序号等于窗口基序号，窗口序号向前移动到具有最小序号的未
确认分组处
                    # 发送分组操作在Send函数中实现
                    while self.received_ack[self.send_base] == 1:
                        self.send_base = (self.send_base + 1) % 256
                        print('SEND WINDOW move to:', self.send_base)
                if self.send_base == self.next_seq:
                    # 所有分组都已经确认，停止计时
                    self.sender_socket.settimeout(None)
                    return

            except socket.timeout:
                # 超时，重发所有已发送但未收到确认的分组
                print('Sender wait for ACK timeout.')
                for i in range(self.send_base, self.send_base +
self.window_size):
                    if self.already_sent[i] == 1 and
self.received_ack[i] == 0:
                        print('Sender resend packet:', i)
                        self.udp_send(self.packets[i])
                self.sender_socket.settimeout(self.timeout)  # reset
timer
                count += 1
```

```
35          return
```

Send 函数：在 GBN 中 `Send` 函数的基础上进行如下更改：

- 分组序号位于发送方窗口内，且在 `already_sent` 列表中发送标记为 0，则打包发送。因此 while 循环条件修改如下：

```
while True:
        while sender.next_seq < (sender.send_base + sender.window_size) \
                and sender.already_sent[sender.next_seq] == 0:
            # ......
```

- 发送后需要在 `already_sent` 列表中标记为已发送。添加如下代码：

```
sender.already_sent[sender.next_seq] = 1
```

SRReceiver 类：在 `GBNReceiver` 类的基础上进行改动：

- 增加接收窗口，基序号为 `self.rcv_base`，大小和发送窗口相同，都是 `windowSize`
- 增加缓存列表，用于缓存接收分组
- 修改 `wait_data` 函数：修改后的 `wait_data` 函数描述接收方对以下事件的响应：
  - 序号在 `[rcv_base, rcv_base + N - 1]` 内的分组被正确接收：

```
if self.rcv_base <= seq_num <= (self.rcv_base + self.window_size - 1) \
    and getChecksum(data) == checksum:
    # 一个选择ACK回送给发送方
    ack_pkt = self.make_pkt(seq_num)
    self.udp_send(ack_pkt)
    if self.received_data[seq_num] == 0:
        # 此分组之前未收到过，缓存此分组
        self.buffer[seq_num] = data
    # 标记已收到
    self.received_data[seq_num] = 1
    if seq_num == self.rcv_base:
        # 此分组序号等于接收窗口的基序号，则此分组以及之前缓存的序号连续的分组交付上层
        self.rcv_base = (self.rcv_base + 1) % 256
        # 已经记录了data，需要将接收窗口滑动一步
        for i in range(self.rcv_base, self.rcv_base + self.window_size):
            if self.received_data[i] == 1:
                data = data + self.buffer[i]
                self.rcv_base = (self.rcv_base + 1) % 256  # 滑动接收窗口
                print('RECEIVE WINDW move to:', self.rcv_base)
            else:
                break
        if flag:
```

```
            return data, True
        else:
            return data, False
    return bytes('', encoding='utf-8'), False
```

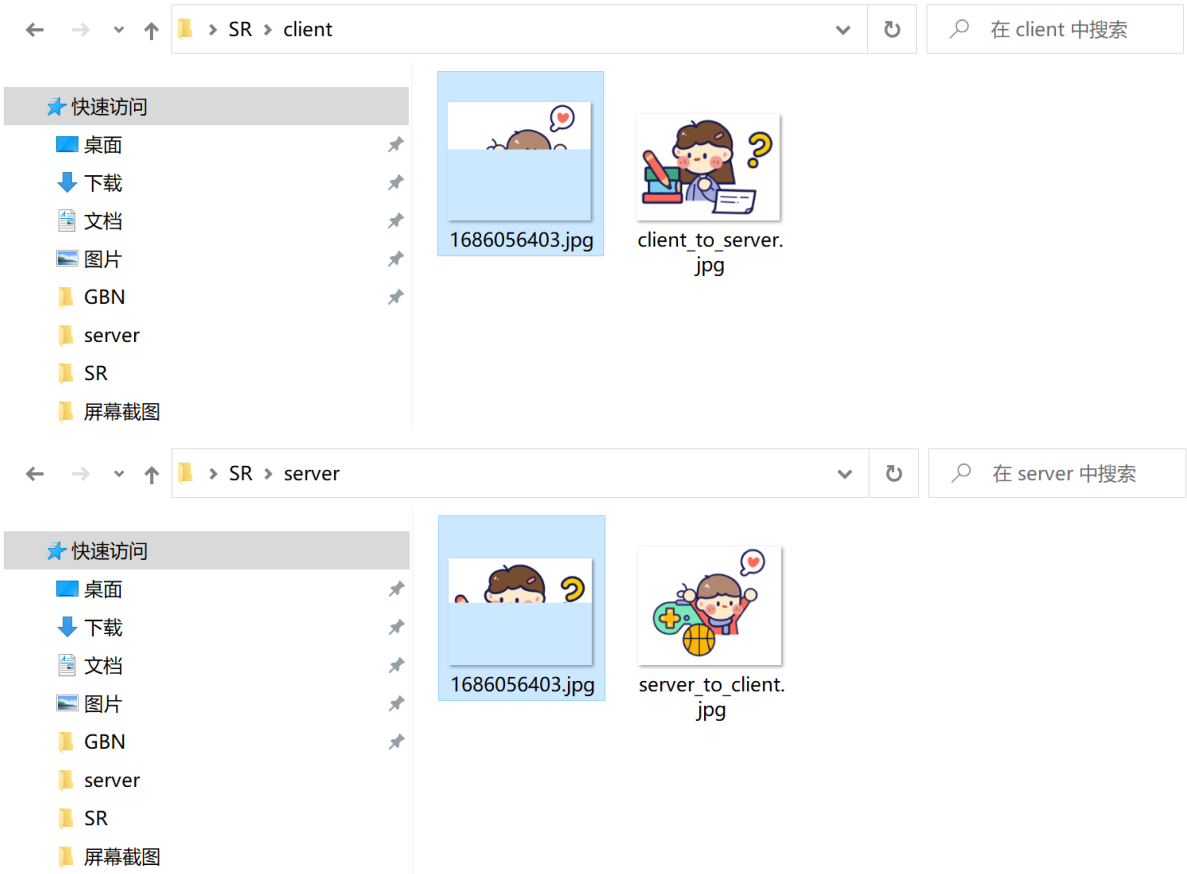
- 序号在 `[rcv_base - N, rcv_base - 1]` 内的分组被正确接收：

```
elif self.rcv_base-self.window_size <= seq_num <= self.rcv_base-1
    and getChecksum(data) == checksum:
    # 产生一个ack，即使该分组是接收方以前确认过的分组
    ack_pkt = self.make_pkt(seq_num)
    self.udp_send(ack_pkt)
    return bytes('', encoding='utf-8'), False
```

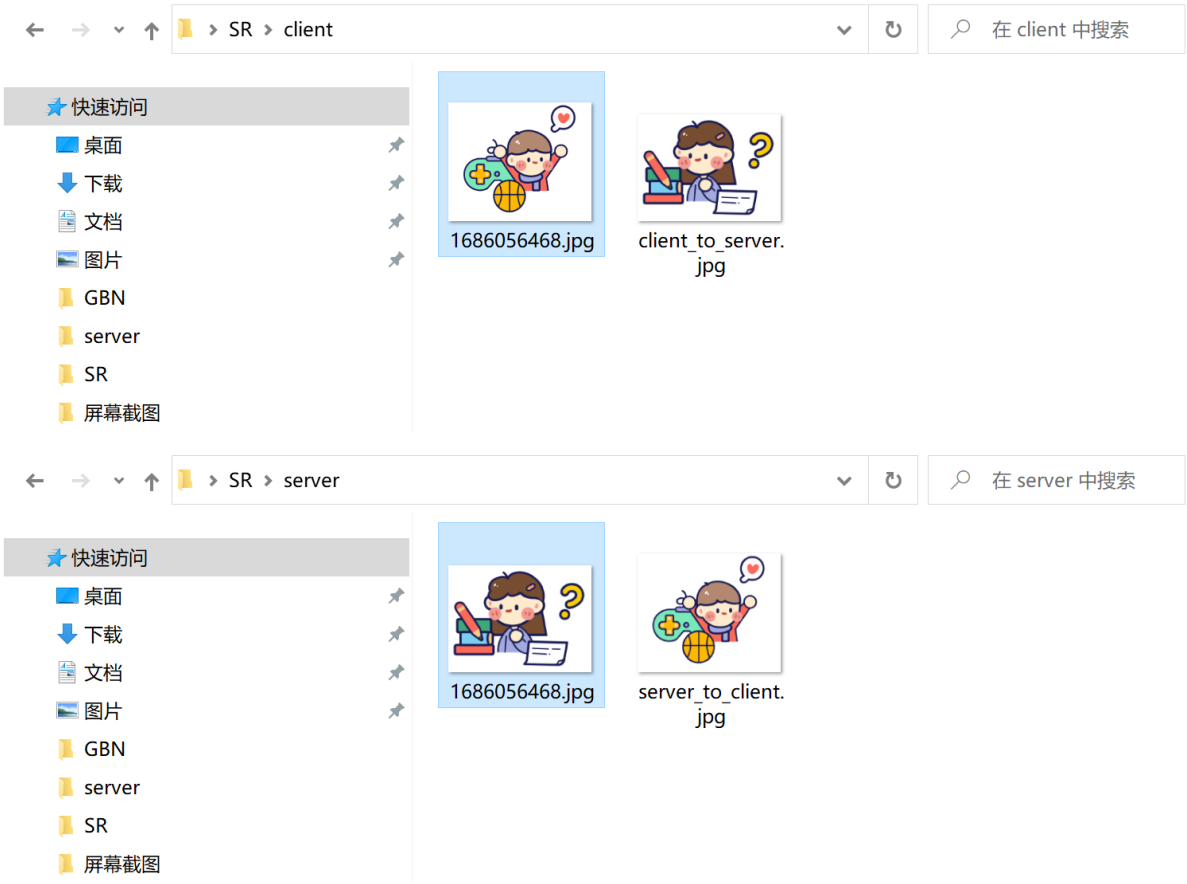
- 其余情况：

```
else:
    # 其他情况，忽略分组
    return bytes('', encoding='utf-8'), False
```

为验证代码准确性，运行程序，执行双向传输。在传输过程中，可以发现客户端、服务器端文件夹下都出现了对方传输过来的图片：



传输完成：

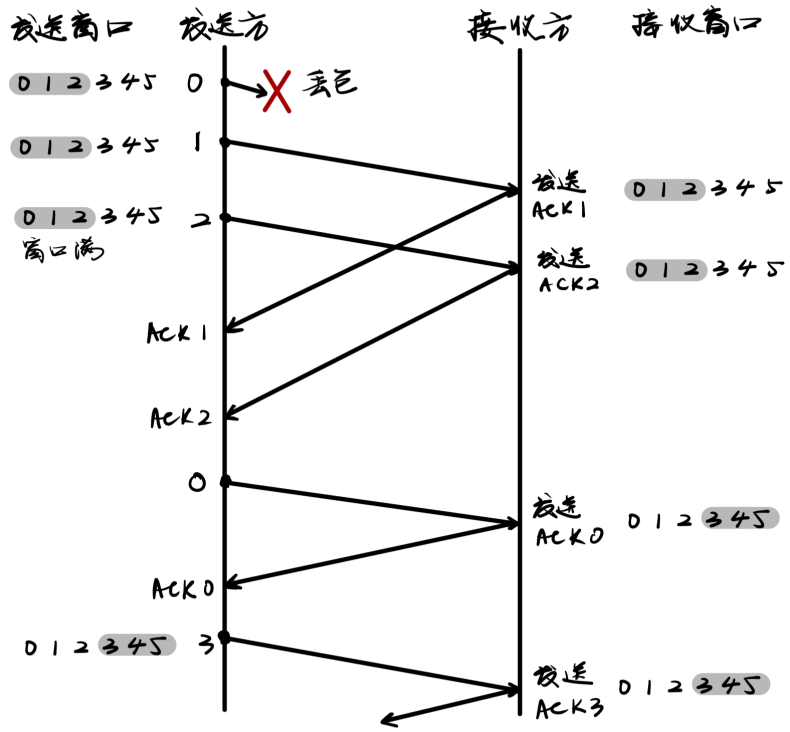


SR 模拟丢包

为方便观察丢包现象，仅从客户端向服务器发送数据，下图记录了某次运行的输出结果（为了体现 SR 协议的特点，输出结果打印了发送窗口和接收窗口的移动情况）：

```
D:\Desktop\GBN\venv\Scripts\python.exe D:\Desktop\SR\main.py
The total number of data packets:  70
Sender send packet: 0
Packet lost.
Sender send packet: 1
seq_num:  1 not end
Receiver receive packet: 1
Receiver send ACK: 1
Data length: 0
Sender send packet: 2
seq_num:  2 not end
Receiver receive packet: 2
Receiver send ACK: 2
Data length: 0
Sender receive ACK 1
Sender receive ACK 2
Data length: 0
Sender wait for ACK timeout.
Sender resend packet: 0
seq_num:  0 not end
Receiver receive packet: 0
Receiver send ACK: 0
RECEIVE WINDW move to: 2
RECEIVE WINDW move to: 3
Data length: 6144
Sender receive ACK 0
SEND WINDOW move to: 1
SEND WINDOW move to: 2
SEND WINDOW move to: 3
Sender send packet: 3
seq_num:  3 not end
Receiver receive packet: 3
Receiver send ACK: 3
Data length: 2048
```

根据运行结果，下图描述了窗口长度为 3 的 SR 的运行情况：



可见 SR 将失序分组缓存直到所有丢失分组都被收到为止，再将一批分组按序交付上层

拥塞控制实现

本项目在 SR 协议的基础上实现了拥塞控制中的慢启动、拥塞避免和快速重传（完整代码见文件 “拥塞控制” ）。

- 慢启动的原理是，新建连接时拥塞窗口 `cwnd` 初始化为 1 个最大报文段（MSS）大小，发送端开始按照拥塞窗口大小发送数据，每当有一个报文段被确认，`cwnd` 就增加 1 个 MSS 大小。使用 `ssthresh` 变量，当 `cwnd` 超过该值后，慢启动过程结束，进入拥塞避免阶段。
- 拥塞避免的原理是加增性，即窗口中所有的报文段都被确认时，`cwnd` 大小加 1，若发生超时，则把 `ssthresh` 降低为 `cwnd` 值的一半，把 `cwnd` 重新设置为 1，重新进入慢启动阶段
- 快速重传的原理是，接收端收到失序报文则重发期待序号的 ACK，接收到连续的 3 个重复冗余ACK（即 4 个同样的ACK）便知晓哪个报文段在传输过程中丢失了，于是重发该报文段，不需要等待超时重传定时器溢出

综上，为实现拥塞控制，需要在现有的 SR 协议代码中进行如下改动：

- 对 `SRSender` 类中的 `wait_ack` 函数进行如下改动：
  - 将 `windowSize` 变量改为 `cwnd` 变量，引入 `ssthresh` 变量，设置为 8
  - 收到 ACK 后在 `received_ack` 列表中进行累加，达到 4 则重传并恢复：

```
if self.send_base <= ack_seq <= self.send_base+self.cwnd:
    self.received_ack[ack_seq] += 1
    # 快速重传
    if self.received_ack[ack_seq] == 4:
        self.udp_send(self.packets[ack_seq])
        self.received_ack[ack_seq] = 1
```

  - 每当有一个报文段被确认，若 `cwnd` 值小于 `ssthresh` 值，`cwnd` 增加 1 个 MSS 大小：

```
ack_seq = self.analyse_pkt(data)
print('Sender receive ACK', ack_seq)
# 慢启动
if self.cwnd <= self.ssthresh:
    self.cwnd *= 2
```

  - 窗口中所有的报文段都被确认时，`cwnd` 大小加 1：

```
if self.send_base == self.next_seq:
    # 所有分组都已经确认，停止计时，cwnd值+1
    self.sender_socket.settimeout(None)
    self.cwnd += 1
```

  - 若发生超时，则把 `ssthresh` 降低为 `cwnd` 值的一半，把 `cwnd` 重新设置为 1，重新进入慢启动阶段：

```
except socket.timeout:
    print('Sender wait for ACK timeout.')
    # 拥塞避免
    self.ssthresh = self.cwnd // 2
    self.cwnd = 1
```

- 为实现快速重传，对 `SRReceiver` 类中的 `wait_data` 函数进行如下改动，若数据包失序到达，则发送冗余 ACK：

```
if self.rcv_base <= seq_num <= self.rcv_base+self.window_size-1
    and getChecksum(data) == checksum:
    # 一个选择ACK回送给发送方
    ack_pkt = self.make_pkt(seq_num)
    self.udp_send(ack_pkt)
    # 快速重传
    if seq_num > self.rcv_base:
        ack_pkt = self.make_pkt(self.rcv_base)
        self.udp_send(ack_pkt)
```