基于UDP的LFTP文件传输实验报告

一、实验分工

院系	数据科学与计算机学院	班级	计算机应用
16340134	梁毓颖	16340166	马佳
实验分工			
梁毓颖		马佳	
主要完成了Connection、TCPClient、TCPServer这些最基本的用于传输的类和用于与用户交互的test_client，处理并发请求与连接建立逻辑处理。参与其余部分设计与对接。		主要完成了Header类的基本模型、Buffer、ReceiveWindow、SendWindow等类的实现与拥塞控制、流控制、差错恢复接口。参与其余部分的设计与对接。	

二、实验要求

1. 选择C、C++、Java、Python当中的语言实现
2. LFTP使用client-server模式
3. LFTP必须包括客户端与服务端的代码，并且客户端可以从服务器下载文件与上传文件
4. LFTP使用UDP作为传输层工具
5. LFTP必须保证100%可靠性
6. LFTP必须声明与TCP类似的流控制
7. LFTP必须声明与TCP类似的拥塞控制
8. LFTP服务端必须支持多客户端
9. LFTP应当在程序运行时提供有意义的信息

三、实验环境与工具

Pycharm、 Python 3.0+、Windows10开发环境

四、实验原理

使用UDP来实现可靠传输，需要在应用层模仿TCP的传输，所以需要对两者的传输进行对比

	基本区别	编程区别

UDP	①无连接的，及发送数据之前不需要建立连接 ②UDP尽最大努力交付，即不保　证可靠交付 ③UDP是面向报文的 ④UDP的首部开销小，只有8个字节 ⑤不可靠信道	**UDP编程的服务器端**一般步骤是： 1、创建一个socket，用函数socket()； 2、设置socket属性，用函数setsockopt();* 可选 3、绑定IP地址、端口等信息到socket上，用函数bind(); 4、循环接收数据，用函数recvfrom(); 5、关闭网络连接； **UDP编程的客户端**一般步骤是： 1、创建一个socket，用函数socket()； 2、设置socket属性，用函数setsockopt();* 可选 3、绑定IP地址、端口等信息到socket上，用函数bind();* 可选 4、设置对方的IP地址和端口等属性； 5、发送数据，用函数sendto()； 6、关闭网络连接；
TCP	①面向连接的 ②提供可靠的服务，通过TCP连接传送的数据无差错、不丢失、不重复，并且按序到达（有流控制、拥塞控制、差错避免） ③面向字节流的，TCP将报文看成是一连串五结构的字节流 ④每一条TCP连接只能够是点对点的 ⑤TCP首部开销20字节 ⑥TCP的逻辑通信信道是全双工的可靠信道	**TCP编程的服务器端**一般步骤是： 1、创建一个socket，用函数socket()； 2、设置socket属性，用函数setsockopt(); * 可选 3、绑定IP地址、端口等信息到socket上，用函数bind()； 4、开启监听，用函数listen()； 5、接收客户端上来的连接，用函数accept()； 6、收发数据，用函数send()和recv()，或者read()和write()； 7、关闭网络连接； 8、关闭监听； **TCP编程的客户端**一般步骤是： 1、创建一个socket，用函数socket()； 2、设置socket属性，用函数setsockopt();* 可选 3、绑定IP地址、端口等信息到socket上，用函数bind();* 可选 4、设置要连接的对方的IP地址和端口等属性； 5、连接服务器，用函数connect()； 6、收发数据，用函数send()和recv()，或者read()和write()； 7、关闭网络连接；

五、实验设计

报文结构：

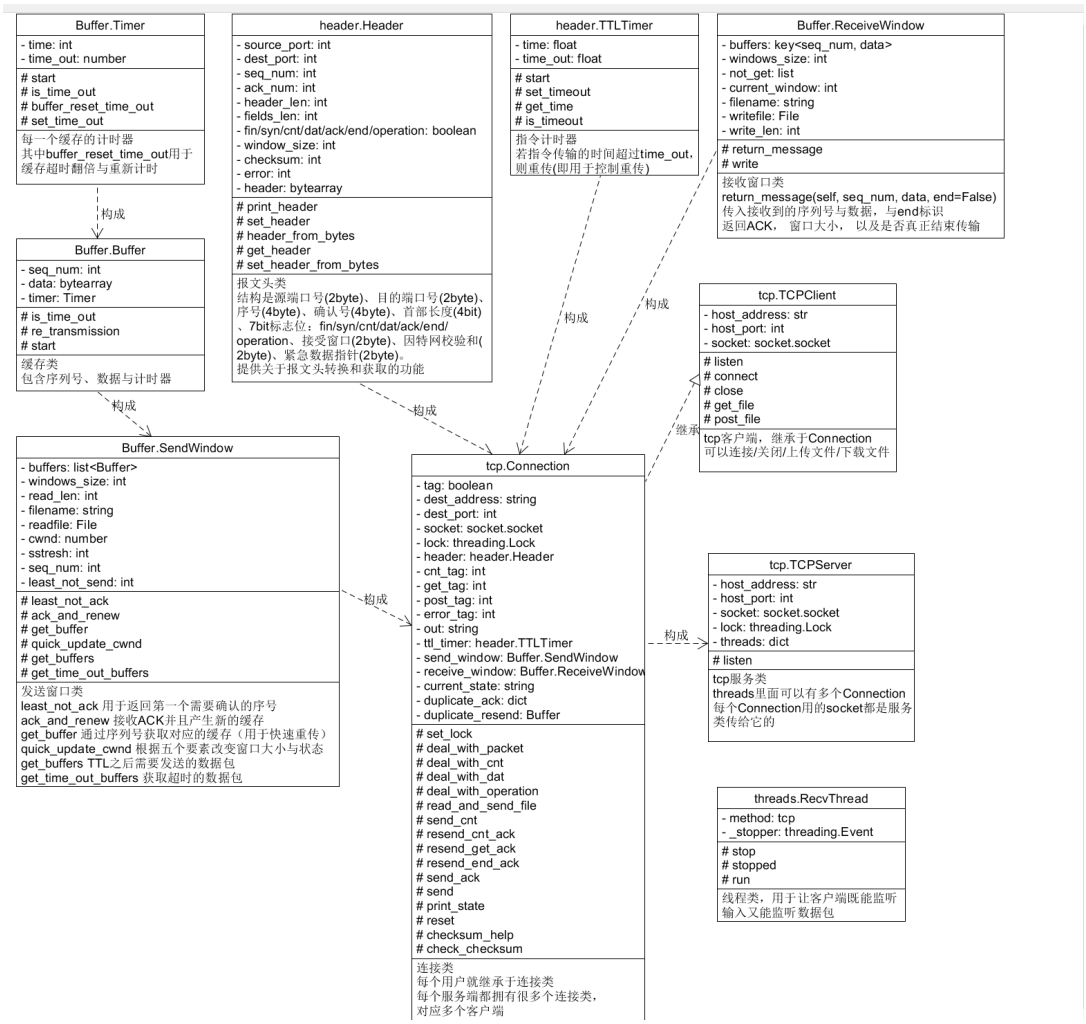
模仿TCP的报文结构并在其基础上改进。每一行仍为32bit

源端口									目的端口
序列号									
确认序列号									
首部长度	-	Op	End	Ack	Dat	Cnt	Syn	Fin	检验和
Error 标志位	窗口大小								

其中：与TCP有差异的只是标志位不同。

标志位	功能
Op(1bit)	get形式的数据/post形式的数据的指令操作
End(1bit)	End报文，确认数据包完整
Ack(1bit)	Ack确认报文
Dat(1bit)	data数据报文
Cnt(1bit)	请求报文（请求数据）
Syn(1bit)	初始化链接
Fin(1bit)	结束连接
Error(1byte)	错误提示功能字节

整体框架



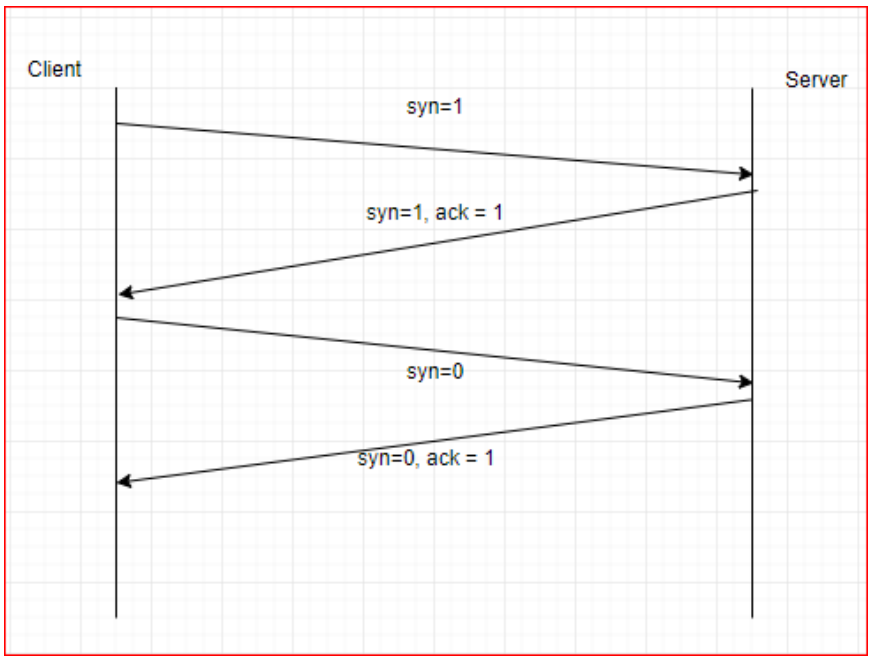
事件监听：

为了能够让一个客户端在监听事件(即监听其他socket发送来的数据)的同时，还能输入指令进而发送指令报文，所以需要将监听这个事件独立出来作为一个线程，主线程则可以用于用户输入。所以要用到RecvThread来监听抵达到这台电脑这个端口的数据包，当然如果服务端也需要在监听事件的同时需要做其他事情，也需要这个方法就行了。
以客户端为例：

```
// 声明线程
class RecvThread(threading.Thread):
  inititalfunc()
  otherfunc()
// 然后是独立出线程
client = TCPClient()
client_listener = threads.RecvThread(client)
client_listener.start()
// client还可以继续进行其他操作，而不用一直监听抵达的数据包了
client.dosomething()
```

三次握手：

在传输文件(上传或下载)之前，客户端与服务端需要先建立连接，即三次握手。客户端先发送一个syn=1的报文，让服务端收到连接建立请求报文，然后服务端返回syn=1,ack=1的syn报文的确认报文，之后客户端又返回syn=0来表明连接已建立，最后服务端又返回syn=0,ack=0的ack报文，让客户端停止重传syn=0的报文(由于担心丢包，syn=1和syn=0的连接建立请求报文都是不断重传的)。



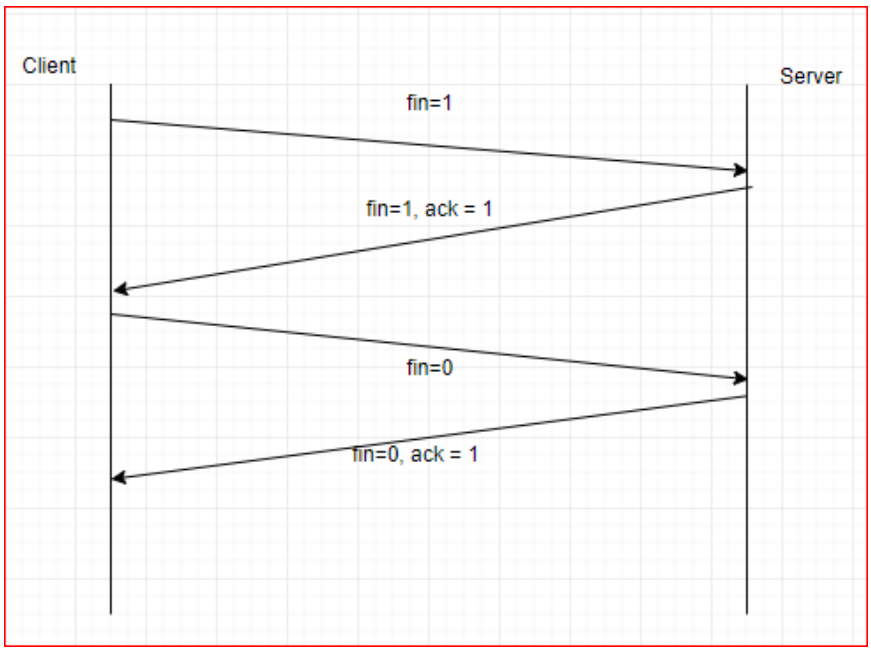
大概实现如下：

```
// client的请求连接函数
func connect():
  // 确认是控制报文
  cnt = 1
  // 三次握手的第一个报文：syn=1
  syn = 1
```

```
  send()
  while not_receice_syn_ack:  // 接收到三次握手的第二个报文：syn=1, ack = 1
    send()
  // 第三个三次握手的报文：syn=0
  syn = 0
  send()
  while not_receive_ack:  // 接收到第三个报文的确认报文：syn=0, ack=0
    send()
  finish()
  cnt = 0
// server的响应控制报文的函数
func deal_with_cnt(some parameter):
  if receive_syn1:
    syn = 1
    send_ack()
  else if receive_syn0:
    syn = 0
    send_ack()
  else:
    do_other_thing()
```

四次挥手：

想要断开连接，只需要像三次握手一样来处理就行了，所以客户端先得发送fin=1报文，然后是服务端的fin=1, ack=1的确认报文，之后就是客户端的fin=0的真正的结束报文，之后服务端会返回fin=0, ack=1的确认报文，让客户端真正断开连接。



大概实现如下：

```
// client的断开连接函数
func close():
  // 确认是控制报文
  cnt = 1
  // 四次挥手的第一个报文：fin = 1
  fin = 1
  send()
```

```
  while not_receive_fin_ack:  // 四次握手的第二个报文：fin=1, ack=1
    send()
  // 四次握手的第三个报文：fin = 0
  fin = 0
  send()
  while not_receive_ack:  // 四次握手的第四个报文：fin=0,ack=1
    send()
  finish()
// server的控制报文处理函数
func deal_with_cnt(some parameter):
  if receive_fin1:
    fin = 1
    send_ack()
  else if receive_fin0:
    fin = 0
    send_ack()
  else:
    do_other_thing()
```

下载文件(get)：

在确认建立连接后，get指令能够让客户端从服务端获取文件，以get filename形式。get的具体流程很简单，就先是客户端向服务端发送get=1的get请求报文，然后服务端向客户端发送get=1, ack=1, error=0/1的报文，告知客户端服务端是否具有此文件，之后客户端就可以根据error标签位来确认是否能够继续请求数据，如果服务端有此文件，则error=0，所以客户端会继续发送get=0的报文来真正的请求数据，这之后服务端就会返回dat=1的真正数据报文。

```
// 客户端的下载文件函数
func get_file(some parameter):
  send_get()
  while not_receive_get_ack():
    send_get()
  send_get0()
  while not_receive_get0_ack():
    send_get0()
// 处理指令报文
func deal_with_operation(some parameter):
  // 如果是get报文
  if header.operation:
    if is_server and reveice_get:
      send_get_ack()
    else if is_server and receive_get0:
      send_get0_ack()
```

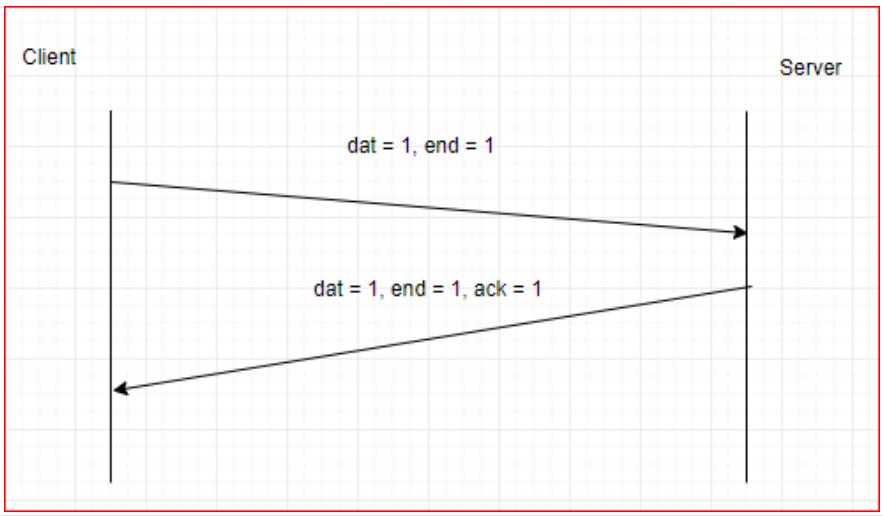
上传文件(post)：

post指令与get指令使用的是同一套机制，都是先是修改其他位，然后传输数据时返回dat报文。只不过post指令将operation设为0，而get是将operation设为1：
大概实现如下：

```
// 客户端的上传文件函数
func post_file(some parameter):
  send_post()
  while not_receive_post_ack():
    send_post()
func deal_with_operation(some parameter):
  // 如果是get报文
  if header.operation:
    if is_server and reveice_post:
      send_post_ack()
    else if is_client and receive_post_ack:
      send_dat()
```

数据处理:

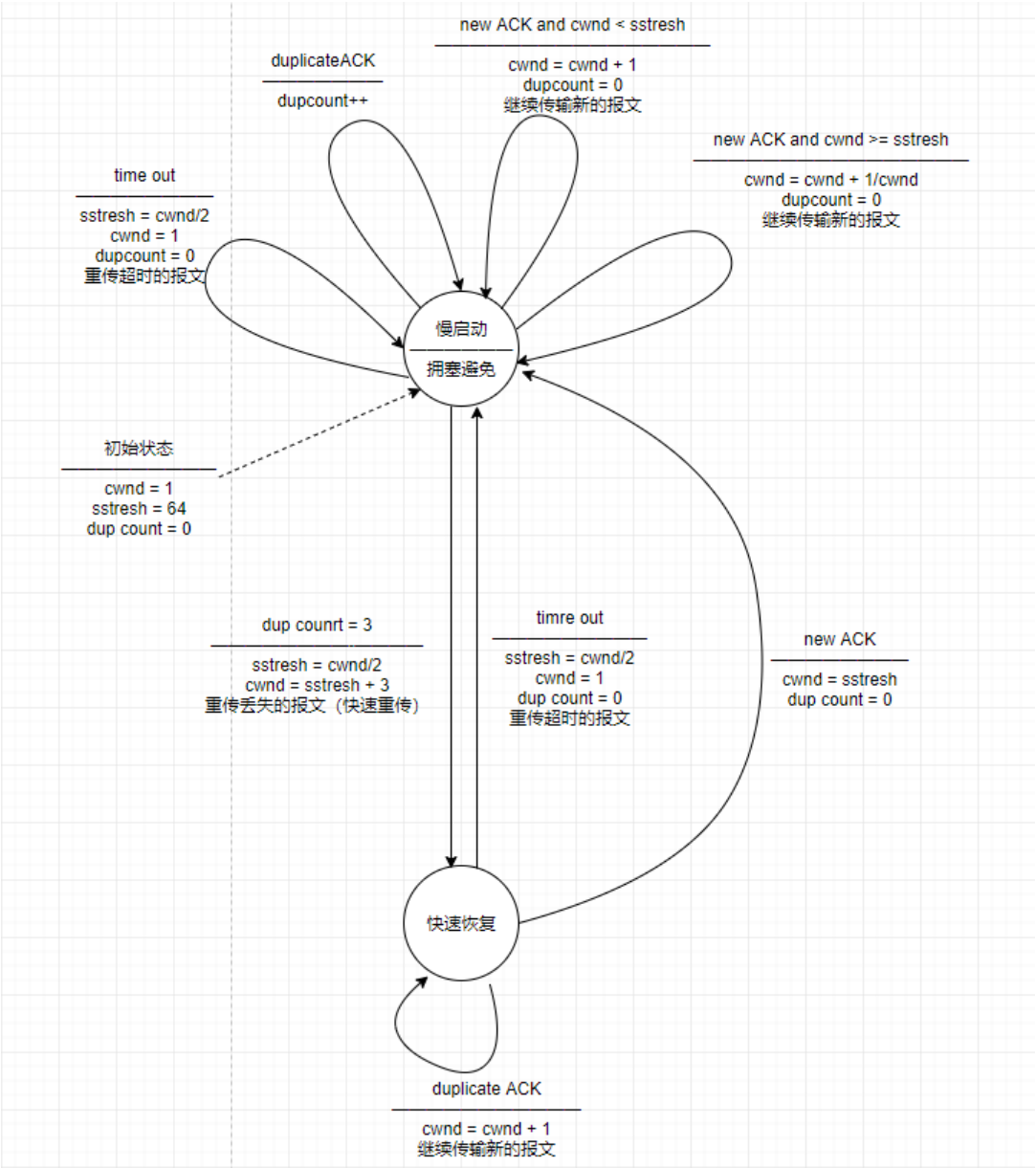
数据处理的状态是从get/post进入的，所以只需要列出结束的过程就行了，以客户端作为发送方为例，只要发送方发出dat end报文，发送方就结束接受，而且返回ack报文。




```
func deal_with_dat(some parameter):
  if reveive_dat_end():
    // 接收方
    finsish()
    return dat_end_ack()
  else if receive_dat_end_ack():
    // 发送方
    finish()
```

发送窗口：拥塞控制/ 发送方状态迁移

在TCP的基础上进行简化，因为TCP窗口中慢启动与拥塞避免状态的区别仅仅在于cwnd的增加速度的降低，所以可以将拥塞控制的三个状态的FSM变化为如下：



也就是进行窗口的大小的变化的时候需要获取的信息有：new_ack，duplicate_ack，dup_count，time_out，current_state，根据上面的五个信息进行判断。所以根据上面的状态转移图进行状态的迁移以及窗口的改变就可以了

```
initial:
      cwnd = 1
      sstresh = 64
      dup_count = 0

current_state = "slow"
  time_out -> sstresh = cwnd/2
              cwnd = 1
              dup_count = 0
              # 重传
  duplicate_ack -> dup_count = 0
                   # 继续
  new ack && cwnd < sstresh -> cwnd = cwnd + 1
                               dup_count = 0
                                # 继续
```

```
  new ack && cwnd >= sstresh -> cwnd = cwnd + 1/cwnd
                                dup_count = 0
                                # 继续
  dup_count = 3 -> sstresh = cwnd/2
                   cwnd = sstresh + 3
                   # 重传丢失的报文
                   current_state = "quick"
current_state = "quick"
  time_out -> sstresh = cwnd/2
              cwnd = 1
              dup_count = 0
              # 重传
  duplicate_ack -> cwnd = cwnd + 1
                   # 继续
  new ack -> cwnd = sstresh
             dup_count = 0
             # 继续
```

快速重传：

每一个发送方维护一个dup_ack字典，当不等于发送方未被确认的最小的ack_num的时候，如果存在于前面维护的字典中的时候，则计数加一，当计数达到3的时候则传入函数冗余的ack_num与count，通过函数迅速获取需要进行重传的分组，然后进行快速重传。优先级最高。

```
def quick_retrans(duplicate_ack, dup_count):
  if dup_count >= 3:
    # 迅速重传
  else:
    # 不返回数据
```

超时重传：超时时间加倍与重传

发送方每一个缓存都有一个计时器，第一次发送的时候打开计时器的计时，而在发生超时的时候调用超时时间翻倍函数并且重新开始计时。然后进行重传。
定时器与缓存类的设计如下：

```
# 计时器
class Timer:
    def __init__(self):
        # 计时开始时间
        self.time = 0
        # 超时时间
        self.time_out = 0.1

    def start(self):
        self.time = time()

    # 判断是否超时
    def is_time_out(self):
```

```
        if time() - self.time > self.time_out:
            return True
        else:
            return False

    # 缓存超时时进行超时翻倍以及重新计时
    def buffer_reset_time_out(self):
        self.time_out = self.time_out * 2
        self.start()

    # 预留
    # 设置超时时间
    def set_time_out(self, new_time_out):
        self.time_out = new_time_out


# 缓存类
class Buffer:
    def __init__(self, seq_num, data):
        # 序列号
        self.seq_num = seq_num
        # 数据
        self.data = data
        # 计时器
        self.timer = Timer()

    # 判断是否超时
    def is_time_out(self):
        return self.timer.is_time_out()

    # 超时时调用
    # 进行重传
    def re_transmission(self):
        self.timer.buffer_reset_time_out()

    # 开启计时
    # 第一次发送的时候使用
    def start(self):
        self.timer.start()
```

发送方通过遍历当前的发送窗口中的缓存来获取超时的缓存然后进行重传
代码如下：用一个发送窗口中的函数进行封装，外部只需要调用即可

```
  # 获取超时分组在外部进行重传
  def get_time_out_buffers(self):
    # 使用try防止两个线程同时操作buffer导致的错误
      try:
        # 存储超时缓存
          resent_buffers = []
          if len(self.buffers) > 0:
              # 获取等待ack的分组
```

```
                wait_buffers = self.buffers[0: self.least_not_send - self.buffers[0].seq_num]
                for i in wait_buffers:
                  # 判断是否超时
                    if i.is_time_out():
                        resent_buffers.append(i)
                        # 超时时间重置
                        i.re_transmission()
                return resent_buffers
        except Exception as e:
            print("Buffer get_time_out_buffers 出错")
        return []
```

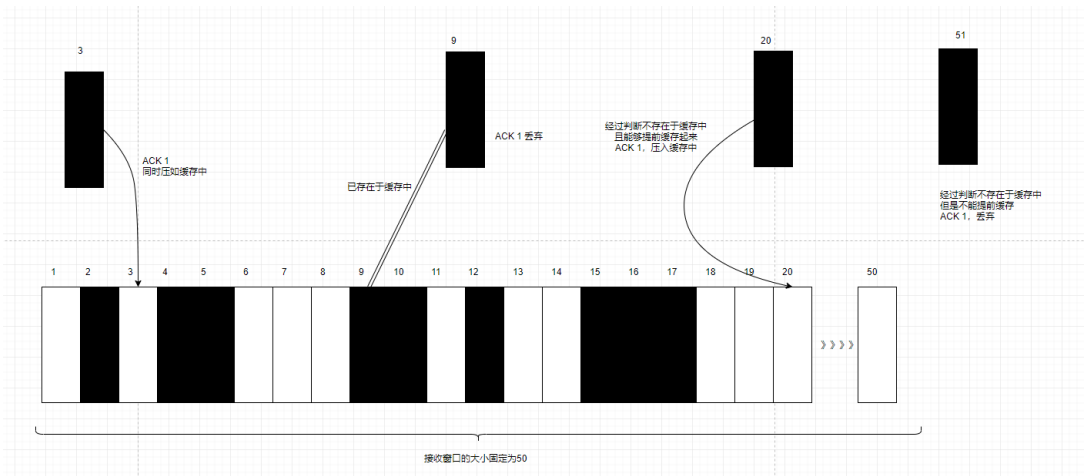
接收窗口：差错恢复

因为TCP的差错恢复机制是通过SR+GBN的机制实现的，也就是TCP的确认是累积式的，正确接收但是失序的报文是不会被接受方逐个确认的。
原来的机制：TCP发送方仅需要维持已发送过但是未被确认的自己的最小序号sendbase和下一个要发送的字节的序号NextSeqNum
强化后的机制：用一个list来存储期望接收到的数据的序列号，用not_get表示，其中not_get[0] – 1表示需要返回给发送方的ACK_num，也就是到目前为止接收到的连续的报文段的最后的序号。然后如果对应的需要的报文到达则更新not_get列表并且进行其余的操作。这样不仅可以做到存储失序的报文，同时容忍的未到达的数字也会更多。
伪代码如下：

```
# 初始化：
# 表示期待第0个序号的报文段到达
not_get = {0}

# 传入接收到的报文中序列号与数据与end标签
# 返回报文中需要的ack_num, left_buffer_szie, whether_true_end（表明文件是否接收完全）

def return_message(seq_num, data, end=False):
    # 条件：序列号等于期待的第一个元素 并且当前的缓存 + not_get的长度小于窗口大小(相当于给not_get预留空间)
    if seq_num == not_get[0] and len(buffers) + len(not_get) < windows_size:
        # 当not_get的长度为1时
        if len(not_get) == 1:
            # 加入
            # 更新not_get
        # 当not_get的长度大于1时
        else:
            # 加入
            # 更新not_get
        # 返回ACK_num, left_buffer_size, 是否真正结束
        return seq_num, windows_size - len(buffers), end == False and len(not_get) == 1

```

```
    # 当小于最小的期待值的时候说明已经存在于缓存中，直接返回
    elif seq_num < not_get[0]:
        # 返回ACK_num, left_buffer_size, 是否真正结束
        return seq_num, windows_size - len(buffers), end == False and
len(not_get) == 1
    # 当大于最小的期待值的时候
    elif seq_num > not_get[0]:
        # 如果在not_get中并且不是最后一个元素(隐含的条件，在not_get数组中间，不包
括头尾)
        if seq_num in not_get and seq_num != not_get[-1]:
            # 加入
            # 更新not_get
        # 等于最后一个元素
        elif seq_num == not_get[-1] and len(not_get) + len(buffers) + 1 <
windows_size:
            # 加入
            # 更新not_get
        # 不在期待的not_get数组中
        # 数值比最后的not_get元素大
        # 有缓存空间可以存储这一个失序的元素或者提前到达的元素
        elif len(not_get) + len(buffers) + seq_num - not_get[-1] + 1 <
windows_size and seq_num > not_get[-1]:
            # 加入
            # 更新not_get
        else:
            # 丢弃
            pass
        # 返回最小期待的元素-1， left_buffer_size, 是否真正结束
        return not_get[0] - 1, windows_size - len(buffers), end == False
and len(not_get) == 1
    else:
        # 丢弃
        # 返回最小期待的元素-1， left_buffer_size, 是否真正结束
        return not_get[0] - 1, windows_size - len(buffers), end == False
and len(not_get) == 1
```

图示如下：



流控制：

窗口大小采用类似操作系统中的分段的机制，也就是每一个缓存的大小是固定的值，不管最终接收到的数组是否为比缓存小。所以反馈的窗口的大小是以一个MSS-header作为单位的，而不是Bytes。流控制这一个比起前面的较简单，因为仅仅需要通过接受方所反馈的窗口大小来做调整，上面接收窗口的重传机制中的函数有反馈窗口的大小，所以在接受方反馈的报文中将窗口的剩余空间添加进报文中进行传输，然后发送方得到报文后从报文中获取剩余的窗口大小判断是否有剩余的空间，如果没有则不发送数据包（有点不同于TCP，TCP发送None，但是觉得None也没有什么意义，而且在前面的机制前提下剩余空间为0的可能性比较小，除非写线程特别慢）
伪代码如下：

```
# 接受方
header.window_size = function(...)
send()

# 发送方
header = recvfrom()
# 剩余的空间为0
if header.window_size == 0:
  # 不发送新的报文
else:
  # 继续发送新的报文

# 继续检测超时与重传
# 继续检测快速重传
```

线程并发：

在这个实验中其实并没有这么使用到多线程，只是每个用户的连接要使用文件传输的时候才建立线程，因为即使有多个用户，每个包还是按顺序来的，每个包只要用单线程就能处理，除了因此带来的IO密集型操作，所以操作文件的时候才需要另起一个线程，线程的并发依靠的是python自带的threading.Thread库，所以大致实现如下：

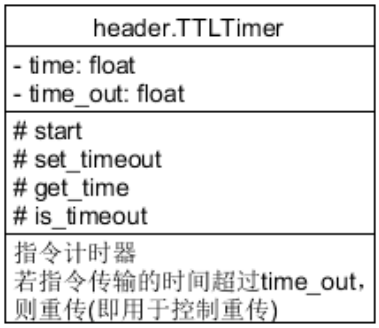
```
// server
if receive_file_operation:
  operate_file_thread = threading.start_new_thread(
    deal_with_file_func, args)
  operate_file_thread.start()
```

而多用户的处理则是依靠它们的IP地址和IP端口。以这两个作为键，建立键值对，而值就是每个用户的连接Connection了，接收到包就传给对应的Connection来处理，当然，这里没有利用到多线程，但是只是因为处理一个指令是很快的，无需线程。

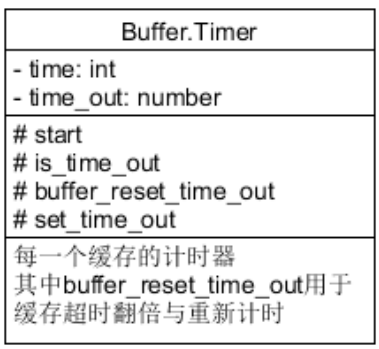
六、实验步骤（代码实现）

1. 前置准备

给出重要的前置模块：

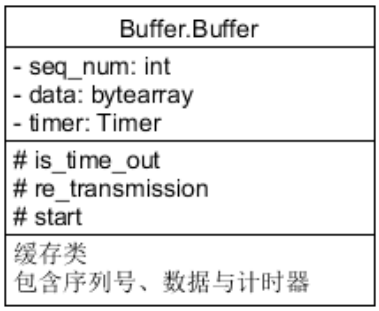



```
class TTLTimer:

    def __init__(self):
        # 计时、默认超时时间
        self.time = 0
        self.time_out = 0.5

    # 开始计时(重新开始计时)
    def start(self):
        self.time = time()

    # 用于进行流控制，根据RTT设置超时时间
    def set_timeout(self, time_out):
        self.time_out = time_out

    def get_time(self):
        return self.time - time()

    # 判断是否超时
    def is_timeout(self):
        return time() - self.time >= self.time_out
```



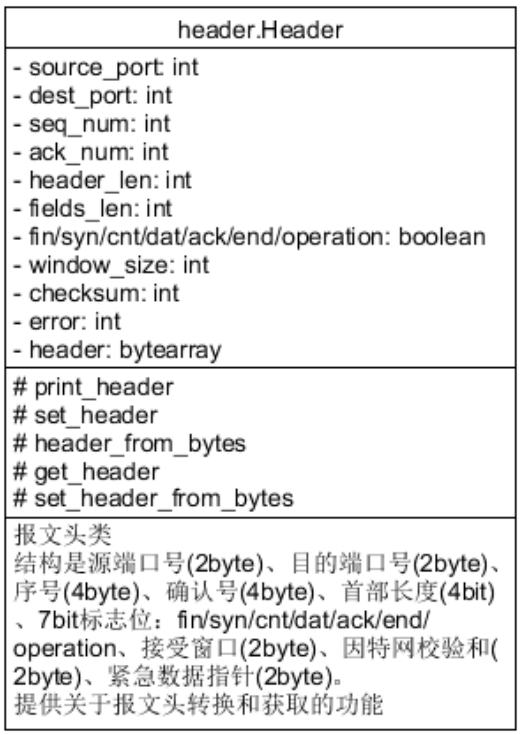

```
# 计时器
# 提供缓存与发送速率
class Timer:
    def __init__(self):
        self.time = 0
        # 定义一个
        self.time_out = 0.1

```

```
    def start(self):
        self.time = time()

    def is_time_out(self):
        if time() - self.time > self.time_out:
            return True
        else:
            return False

    # 缓存超时时进行超时翻倍以及重新计时
    def buffer_reset_time_out(self):
        # 翻倍
        self.time_out = self.time_out * 2
        # 重新开始
        self.start()

    # 预留
    # 设置超时时间
    def set_time_out(self, new_time_out):
        self.time_out = new_time_out
```




```
# 缓存类
class Buffer:
    def __init__(self, seq_num, data):
        # 每一个缓存的序列号
        self.seq_num = seq_num
        # 每一个缓存的数据
        self.data = data
        # 计时器
        self.timer = Timer()

    # 判断是否超时
    def is_time_out(self):
        return self.timer.is_time_out()

    # 超时 时调用
    # 进行重传
    def re_transmission(self):
        self.timer.buffer_reset_time_out()

    # 手动开启时间
    # 第一次发送的时候使用
```

```
    def start(self):
        # 创建的时候顺便开始计时
        self.timer.start()
```




```
class Header:

    # 纯正tcp报文结构：
    #   源端口号(2byte)、目的端口号(2byte)、序号(4byte)、确认号(4byte)
    #   首部长度(4bit)、保留未用(2bit)
    #   6bit标志位：ack、rst、syn、fin、psh、urg
    #   接受窗口(2byte)、因特网校验和(2byte)、紧急数据指针(2byte)
    # 保留未用哪里与6bit标志位作为8个标志位：fin/syn/cnt/dat/ack/end/operation/None(bit顺序从低位到高位)
    # 紧急数据指针被拆分，1byte为error(错误标记)、1byte为接受窗口
    # 然后后面的才是：
    #   选项
    #   数据
    def __init__(self):
        self.source_port = 0
        self.dest_port = 0
        self.seq_num = 0
        self.ack_num = 0
        self.header_len = 20
        self.fields_len = 0
        # 下面八个bool类型的字段都是同一个byte的，只是属于不同的bit，分别代表
        # ack确认报文、syn初始化连接、fin结束连接
        # data数据报文、end报文(确认数据包完整)、cnt请求报文(请求数据)
        # get形式的数据/post形式的数据的指令操作
        self.ack, self.syn, self.fin = False, False, False
        self.dat, self.end, self.cnt = False, False, False
        self.operation = False
```

```
        self.window_size = 0
        self.chekcsum = 0
        self.error = 0
        # 真正存储报文信息的地方，将上面的各个属性变为字节放入byte array中
        self.header = bytearray(20)

    def print_header(self):
        print(self.ack, self.syn, self.fin, self.dat, self.end, self.cnt, self.operation, self.window_size, self.error)
        print(self.source_port, self.dest_port)

    # 将十进制转化为字节流
    def set_header(self):
        # 源端口号与目的端口号
        self.header[0], self.header[1] = (self.source_port >> 8) & 0xFF, self.source_port & 0xFF
        self.header[2], self.header[3] = (self.dest_port >> 8) & 0xFF, self.dest_port & 0xFF
        # 序号
        self.header[4], self.header[5] = (self.seq_num >> 24) & 0xFF, (self.seq_num >> 16) & 0xFF
        self.header[6], self.header[7] = (self.seq_num >> 8) & 0xFF, self.seq_num & 0xFF
        # 确认号
        self.header[8], self.header[9] = (self.ack_num >> 24) & 0xFF, (self.ack_num >> 16) & 0xFF
        self.header[10], self.header[11] = (self.ack_num >> 8) & 0xFF, self.ack_num & 0xFF
        # 首部长度
        self.header[12] = self.header_len & 0xFF
        # 标记字段
        self.header[13] = 0
        if self.fin:
            self.header[13] = 0x1 | self.header[13]
        if self.syn:
            self.header[13] = 0x2 | self.header[13]
        if self.cnt:
            self.header[13] = 0x4 | self.header[13]
        if self.dat:
            self.header[13] = 0x8 | self.header[13]
        if self.ack:
            self.header[13] = 0x10 | self.header[13]
        if self.end:
            self.header[13] = 0x20 | self.header[13]
        if self.operation:
            self.header[13] = 0x40 | self.header[13]
        # 校验和
        self.header[14], self.header[15] = (self.chekcsum >> 24) & 0xFF, (self.chekcsum >> 16) & 0xFF
        # 错误提示
        self.header[16] = self.error & 0xFF
```

```
        # 接受窗口
        self.header[17] = (self.window_size >> 16) & 0xFF
        self.header[18] = (self.window_size >> 8) & 0xFF
        self.header[19] = self.window_size & 0xFF
        return self.header

    # 将字节流转换为十进制
    def header_from_bytes(self, header):
        # 译码过程
        self.source_port = header[0] << 8 | header[1]
        self.dest_port = header[2] << 8 | header[3]
        self.seq_num = header[4] << 24 | header[5] << 16 | header[6] << 8 | header[7]
        self.ack_num = header[8] << 24 | header[9] << 16 | header[10] << 8 | header[11]
        # 功能码翻译
        if header[13] & 0x1 == 0x1:
            self.fin = True
        if header[13] & 0x2 == 0x2:
            self.syn = True
        if header[13] & 0x4 == 0x4:
            self.cnt = True
        if header[13] & 0x8 == 0x8:
            self.dat = True
        if header[13] & 0x10 == 0x10:
            self.ack = True
        if header[13] & 0x20 == 0x20:
            self.end = True
        if header[13] & 0x40 == 0x40:
            self.operation = True
        # 窗口以及检验和
        self.checksum = header[14] << 8 | header[15]
        self.error = header[16]
        self.window_size = header[17] << 16 | header[18] << 8 | header[19]
        self.header = header

    def get_header(self):
        return self.set_header()

    def set_header_from_bytes(self, header):
        self.header = header
```

2. 实现线程一对一模块化与并行

首先服务端的每个Connection与每个客户端是相对应的，这就实现了一对一。

```
# 开始监听客户端
def listen(self, event):
    print("开始接收")
    while True and not event.stopped():
        try:
            data, address = self.socket.recvfrom(1024)
```

```
        except IOError:
            continue
        packet = bytearray(data)
        # 验证校验和
        if Connection.check_checksum(packet):
            # 依据header_len取出header
            header = Header()
            header.header_from_bytes(packet[:packet[12] & 0xFF])
            connection_key = address[0] + "/" + str(address[1])
            if connection_key not in self.threads.keys():
                print("新用户", connection_key)
                connection = Connection(False, address[0], address[1],
self.socket)
                connection.set_lock(self.lock)
                self.threads[connection_key] = connection
            # 处理packet
            self.threads[connection_key].deal_with_packet(packet, header)
        else:
            print(packet)
            print("报文损毁，丢弃")
```

而每个客户端就是一个Connection，只是多出了一些方法，让它能够建立连接、断开连接、上传文件以及下载文件而已，所以能够实现一对一的模型。但要注意由于服务端只有一个socket，所以不能同时调用socket.sendto这个方法，于是要加锁：

```
# 发送包
def send(self, header=None, data=None):
    if self.lock:
        self.lock.acquire()
    try:
        if header is None:
            header = self.header
        datagram = (header.get_header() + data) if data else
header.get_header()
        checksum = Connection.checksum_help(datagram)
        datagram[14], datagram[15] = (checksum >> 8) & 0xFF, checksum &
0xFF
        self.socket.sendto(datagram, (self.dest_address, self.dest_port))
        print("目的地址{0}，目的端口{1}，seq{2}".format(self.dest_address,
self.dest_port, header.seq_num))
    finally:
        if self.lock:
            self.lock.release()
```

threads.RecvThread
- method: tcp - _stopper: threading.Event
stop # stopped # run
线程类，用于让客户端既能监听输入又能监听数据包

```
# 让每个用户端程序能够在监听用户输入的同时能够监听数据包
class RecvThread(threading.Thread):

    def __init__(self, method):
        super(RecvThread, self).__init__()
        self.method = method
        self._stopper = threading.Event()

    def stop(self):
        self._stopper.set()

    def stopped(self):
        return self._stopper.isSet()

    # 运行一个进程来监听数据
    def run(self):
        self.method.listen(self)
```

3. 三次握手四次挥手

```
#--------------- TCPClient -----------
# 开始连接，三次握手中的客户端的两次send
def connect(self):
    print("开始连接")
    self.header.cnt, self.header.syn = True, True
    self.header.seq_num = 0
    self.send_cnt(0, "客户端：建立连接 —— 发送 SYN = 1 报文", "客户端：建立连接 —— 重传 SYN = 1 报文")
    self.header.syn = False
    self.header.seq_num = 1
    self.send_cnt(1, "客户端：建立连接 —— 发送 SYN = 0 报文", "客户端：建立连接 —— 重传 SYN = 0 报文")
    self.header.cnt = False
    self.print_state()
    # print("客户端：连接建立成功", "/" * 100, sep="\n")

# 结束连接，四次挥手中客户端的两次send
def close(self):
    print("结束连接")
    self.header.cnt, self.header.fin = True, True
    self.header.seq_num = 0
    self.send_cnt(2, "客户端：结束连接 —— 发送 FIN = 1 的报文", "客户端：结束连接 —— 重传 FIN = 1 的报文")
    self.header.fin = False
    self.header.seq_num = 1
    self.send_cnt(3, "客户端：结束连接 —— 发送 FIN = 0 的报文", "客户端：结束连接 —— 重传 FIN = 0 的报文")
    # 其实我觉得None可能更好
    self.header = False
    # print("客户端：连接结束成功", "/" * 100, sep="\n")
#--------------- Connection -----------
```

```
# 发送控制(建立连接和结束连接的控制)报文
def send_cnt(self, while_cnt_tag, first_message=None, while_message=None):
    self.send()
    self.ttl_timer.start()
    if first_message:
        print(first_message)
    while self.cnt_tag == while_cnt_tag:
        if self.ttl_timer.is_timeout():
            self.send()
            self.ttl_timer.start()
            if while_message:
                print(while_message)
# 处理控制报文def deal_with_cnt(self, header):
    self.header.ack_num = header.seq_num
    if self.tag:
        if self.cnt_tag == 0 and header.ack and self.header.syn:
            print("客户端：建立连接 —— 接收到 SYN ACK 报文，发送 SYN = 0 报
文")
            self.cnt_tag = 1
            # connect里面会自动send syn=0了
        elif self.cnt_tag == 1 and header.ack:
            print("客户端：建立连接 —— 接收到 ACK，连接建立成功")
            self.cnt_tag = 2
        elif self.cnt_tag == 2 and self.header.fin and header.ack:
            print("客户端：结束连接 —— 接受到 FIN ACK 报文，发送 FIN = 0 报
文")
            self.cnt_tag = 3
            # close里面会自动send fin=0了
        elif self.cnt_tag == 3 and header.ack:
            print("客户端：结束连接 —— 接受到 FIN = 0 ACK 报文，结束连接")
            self.cnt_tag = 0
    else:
        if self.cnt_tag == 0 and header.syn:
            print("服务端：建立连接 —— 接受到 SYN = 1 报文，发送 SYN ACK 报
文")
            self.header.cnt = True
            self.send_ack()
            self.cnt_tag = 1
        elif self.cnt_tag == 1 and header.syn and header.seq_num == 0:
            print("服务端：建立连接 —— SYN ACK重发")
            self.resend_cnt_ack()
        elif self.cnt_tag == 1 and not header.ack and not header.syn:
            print("服务端：建立连接 —— 连接已完成建立")
            self.cnt_tag = 2
            self.send_ack()
            self.header.cnt = False
        elif self.cnt_tag == 2 and header.fin:
            print("服务端：结束连接 —— 接收到 FIN = 1 报文，发送 FIN ACK 报
文")
            self.header.cnt = True
            self.send_ack()
```

```
            self.cnt_tag = 3
        elif self.cnt_tag == 2 and not header.ack and not header.syn:
            print("服务端：建立连接 —— SYN = 0 的ACK重发")
            self.resend_cnt_ack()
        elif self.cnt_tag == 3 and not header.ack and not header.fin:
            print("服务端：结束连接 —— 接受到 FIN = 0 报文，发送 ACK，连接关闭")
            self.cnt_tag = 0
            self.send_ack()
            self.header.cnt = False
        elif self.cnt_tag == 3 and not header.seq_num and header.fin:
            print("服务端：结束连接 —— 重传 FIN ACK")
            self.resend_cnt_ack()
        elif self.cnt_tag == 0 and not header.ack and not header.fin:
            print("服务端：结束连接 —— FIN = 0 的ACK重发")
            self.resend_cnt_ack()
```

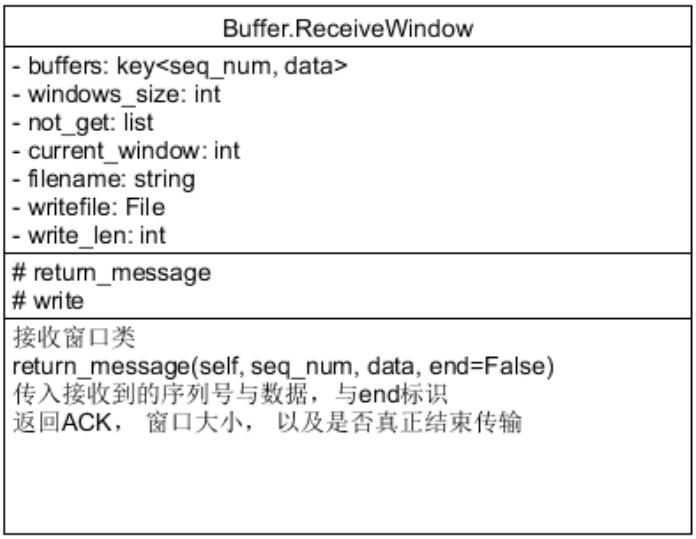
4. 设计接收窗口与发送窗口（包含重传机制接口、拥塞控制接口）


```
# 返回需要确认的序号
# -1表示传输完成
def least_not_ack(self):
    if not len(self.buffers) == 0:
        try:
            return self.buffers[0].seq_num
        except Exception as e:
            return -1
    return -1

# 更新操作
# 三次的重传在上一层实现，不正确的ACK不要进入这里
```

```
# 三次重传调用get_buffer实现
# 小于数字的都弹出，并且压入新的数据
# 返回是否传完
def ack_and_renew(self, seq_num):
    # 删除已经ack的buffer
    while len(self.buffers) > 0:
        #  ??????要不要<=???
        if self.buffers[0].seq_num <= seq_num:
            self.buffers.pop(0)
        else:
            break
    # 插入新的元素
    while len(self.buffers) < self.windows_size:
        data = self.readfile.read(self.read_len)
        if data == b'':
            break
        self.buffers.append(Buffer(self.seq_num, data))
        self.seq_num += 1
    if len(self.buffers) == 0:
        return True
    else:
        return False

# 通过序列号获得对应的buffer
# 返回的是Buffer类型
def get_buffer(self, seq_num):
    # 获取缓存对应的索引
    index = seq_num - self.buffers[0].seq_num
    return self.buffers[index]

# 用于快速重传以及更新cwnd
# current_state:表示的是当前的状态
#           slow 包含congestion
#           quick
# new_ack：
#           1， 表示新的ack
# duplicate_ack:
#           num: 表示序列号
# dupack_count：
#           <3,
#           ==3, 慢启动/拥塞控制->快速重传
#           >3, 继续快速重传
# time_out:
#           0， 表示未超时
#           > 1， 表示超时
#           len(get_time_out_buffers)
# 返回当前需要发送的分组
def quick_update_cwnd(self, current_state, new_ack=-1, duplicate_ack=None,
dupack_count=0, time_out=0):
    # 超时,不返回数组
    # 超时
```

```
    if not time_out == 0:
        if not self.cwnd == 1:
            self.ssthresh = self.cwnd // 2
            self.cwnd = 1
    elif current_state == "slow" and (not duplicate_ack == None) and dupack_count < 3:
        # 外部处理
        pass
    # 返回多次冗余的分组
    elif current_state == "slow" and (not duplicate_ack == None) and dupack_count == 3:
        if self.cwnd >= 2:
            self.ssthresh = self.cwnd // 2
            self.cwnd = self.ssthresh + 3
        else:
            self.ssthresh = self.cwnd
            self.cwnd = self.ssthresh + 3
        print("发送窗口cwnd：", self.cwnd, "发送窗口ssthresh :", self.ssthresh)
        return self.get_buffer(duplicate_ack + 1)
    # 接收到新的ack，重新传输新的分组
    elif current_state == "slow" and not new_ack == -1:
        # 慢启动
        if self.cwnd < self.ssthresh <= self.windows_size:
            self.cwnd += 1
        # 拥塞控制
        elif self.ssthresh <= self.cwnd < self.windows_size:
            self.cwnd += (1 / self.cwnd)
    elif current_state == "quick" and (not duplicate_ack == None):
        # 快速恢复
        if self.cwnd <= self.windows_size:
            self.cwnd += 1
        print("发送窗口cwnd：", self.cwnd, "发送窗口ssthresh :", self.ssthresh)
        return self.get_buffer(duplicate_ack + 1)
    print("发送窗口cwnd：", self.cwnd, "发送窗口ssthresh :", self.ssthresh)

# TTL过后需要进行发送的分组
# 返回None表示发送结束
# 返回[]表示所有的数据都在wait ack
# 返回需要发送的分组
def get_buffers(self):
    if len(self.buffers) == 0:
        return None
    # 表示全部的数据都处于wait ack的状态中
    try:
        if self.least_not_send > self.buffers[-1].seq_num:
            return []
        # 与缓存最后的seq_num比较
        elif self.least_not_send + math.floor(self.cwnd) - 1 > self.buffers[-1].seq_num:
```

```
            # 存储b需要发送的buffer
            buffers = self.buffers[self.least_not_send -
self.buffers[0].seq_num:]
            # 启动计时器
            for buffer in buffers:
                buffer.start()
            self.least_not_send = self.buffers[-1].seq_num + 1
            return buffers
        elif self.least_not_send + math.floor(self.cwnd) - 1 <=
self.buffers[-1].seq_num:
            # 存储需要发送的buffers
            buffers = self.buffers[self.least_not_send -
self.buffers[0].seq_num: self.least_not_send - self.buffers[
                0].seq_num + math.floor(self.cwnd)]
            # 启动计时器
            for buffer in buffers:
                buffer.start()
            self.least_not_send = self.least_not_send +
math.floor(self.cwnd)
            return buffers
    except Exception as e:
        return None

# 获取超时分组在外部进行重传
def get_time_out_buffers(self):
    try:
        resent_buffers = []
        if len(self.buffers) > 0:
            wait_buffers = self.buffers[0: self.least_not_send -
self.buffers[0].seq_num]
            for i in wait_buffers:
                if i.is_time_out():
                    resent_buffers.append(i)
                    # 超时时间重置
                    i.re_transmission()
            return resent_buffers
    except Exception as e:
        print("Buffer get_time_out_buffers 出错")
    return []
```




```python
# 接受窗口
class ReceiveWindow:
def __init__(self, filename):
    self.buffers = {}
    # 缓存字典
    self.windows_size = 30
    # 窗口的大小
    self.not_get = []
    # 序列数组
    self.not_get.append(0)
    # 当前窗口大小
    self.current_window = 0
    # 保存文件名
    self.filename = filename
    # 输出
    self.writefile = open(filename, "wb")
    # 写的大小
    self.write_len = 5

# 传入接收到的序列号与数据
# 返回ACK， 窗口大小， 以及是否结束传输
# 真正传输完成：tag == 1 and len(self.not_get) == 1 成立
# tag表示发送文件是否结束，如果结束则为1
def return_message(self, seq_num, data, tag=0):
    # 是期待的数据的时候
    # 以下是测试的时候用的，用于终止
    # if self.not_get[0] == 111:
    #     self.write(1)
    #     tag = 1
    #     return seq_num, self.windows_size - len(self.buffers), tag == 1
and len(self.not_get) == 1

    if seq_num == self.not_get[0] and len(self.buffers) +
len(self.not_get) < self.windows_size:
        # 一直维护一个最后的元素表示期待的
        if len(self.not_get) == 1:
            self.not_get.pop(0)
```

```
            self.not_get.append(seq_num + 1)
            self.buffers[seq_num] = data
        else:
            self.not_get.pop(0)
            self.buffers[seq_num] = data
        self.write(tag)
        print("接受窗口当前缓存大小：", len(self.buffers), "接受窗口期望数组的大小：", len(self.not_get), "期望0元素", self.not_get[0])
        return seq_num, self.windows_size - len(self.buffers), tag == 1 and len(self.not_get) == 1
    # 小于 最小的期待的时候，直接返回
    elif seq_num < self.not_get[0]:
        self.write(tag)
        print("接受窗口当前缓存大小：", len(self.buffers), "接受窗口期望数组的大小：", len(self.not_get), "期望0元素", self.not_get[0])
        return seq_num, self.windows_size - len(self.buffers), tag == 1 and len(self.not_get) == 1
    elif seq_num > self.not_get[0]:
        # 如果是自己所期待的并且不是最后一个元素
        if seq_num in self.not_get and seq_num != self.not_get[-1]:
            self.not_get.remove(seq_num)
            self.buffers[seq_num] = data
        # 等于最后一个元素
        elif seq_num == self.not_get[-1] and len(self.not_get) + len(self.buffers) + 1 < self.windows_size:
            self.not_get.remove(seq_num)
            self.not_get.append(seq_num + 1)
            self.buffers[seq_num] = data
        # 不在期待的数组中并且数值比一个期待元素大。意味者比最大的元素还大
        elif len(self.not_get) + len(self.buffers) + seq_num - self.not_get[-1] + 1 < self.windows_size and seq_num > self.not_get[-1]:
            for i in range(seq_num - self.not_get[-1] - 1):
                self.not_get.append(self.not_get[-1] + 1)
            self.not_get.append(seq_num + 1)
            self.buffers[seq_num] = data
            # self.windows_size
        else:
            # 丢弃
            pass
        # 调用写方法
        self.write(tag)
        print("接受窗口当前缓存大小：", len(self.buffers), "接受窗口期望数组的大小：", len(self.not_get), "期望0元素", self.not_get[0])
        return self.not_get[0] - 1, self.windows_size - len(self.buffers), tag == 1 and len(self.not_get) == 1
    else:
        self.write(tag)
        print("接受窗口当前缓存大小：", len(self.buffers), "接受窗口期望数组的大小：", len(self.not_get), "期望0元素", self.not_get[0])
        return self.not_get[0] - 1, self.windows_size - len(self.buffers), tag == 1 and len(self.not_get) == 1
```

```
77
78  # tag用于表示是否结束，结束表示为1，立刻写入
79  # 当有效的缓存大于10的时候自动写入
80  def write(self, tag=0):
81      if tag == 1 and len(self.not_get) == 1:
82          if len(self.buffers) > 0:
83              index = min(self.buffers.keys())
84              length = len(self.buffers)
85              for i in range(length):
86                  # self.writefile.write(bytearray("\n序号是" + str(index +
    i) + "\n", encoding="utf-8"))
87                  self.writefile.write(self.buffers[index + i])
88                  del self.buffers[index + i]
89          self.writefile.close()
90      elif len(self.buffers) != 0 and self.not_get[0] -
    min(self.buffers.keys()) > self.write_len :
91          index = min(self.buffers.keys())
92          for i in range(self.write_len):
93              # self.writefile.write(bytearray("\n序号是" + str(index + i) +
    "\n", encoding="utf-8"))
94              self.writefile.write(self.buffers[index + i])
95              self.writefile.flush()
96              del self.buffers[index + i]
97
```

5. get与post的建立过程

```
1   # ---------- TCPClient ----------
2   # get file
3   def get_file(self, filename):
4       role = "客户端"
5       if self.cnt_tag == 2:
6           name_bytes = bytearray(filename, encoding="utf-8")
7           self.out = "./client_files/" + filename
8           self.header.operation = True
9           self.header.seq_num = 0
10          self.send(self.header, name_bytes)
11          self.ttl_timer.start()
12          print(role + ": 发送第一次get请求")
13          while self.get_tag == 0 and self.error_tag == 0:
14              if self.ttl_timer.is_timeout():
15                  self.send(self.header, name_bytes)
16                  self.ttl_timer.start()
17                  print(role + ": 重发了第一次get请求")
18          if self.error_tag != 0:
19              self.error_tag = 0
20              self.header.operation = False
21              self.get_tag = 0
22              return
23          self.header.ack = True
24          self.header.seq_num = 1
```

```
        self.send(self.header, name_bytes)
        self.ttl_timer.start()
        print(role + ": 发送第二次get请求", name_bytes)
        while self.get_tag == 1 and self.error_tag == 0:
            if self.ttl_timer.is_timeout():
                self.send(self.header, name_bytes)
                self.ttl_timer.start()
                print(role + ": 重发了第二次get请求", name_bytes)
        self.header.ack = False
        self.header.seq_num = 0
        self.print_state()
    else:
        print(role + "未连接")

# post file
def post_file(self, filename):
    role = "客户端"
    if self.cnt_tag == 2:
        name_bytes = bytearray(filename, encoding="utf-8")
        self.out = "./client_files/" + filename
        if not os.path.exists(self.out):
            print("客户端没有此文件，请输入正确文件名")
            return
        self.post_tag = 1
        self.header.operation = False
        self.header.seq_num = 0
        self.send(self.header, name_bytes)
        self.ttl_timer.start()
        print(role + "正在发送post请求")
        while self.post_tag == 1:
            if self.ttl_timer.is_timeout():
                self.send(self.header, name_bytes)
                self.ttl_timer.start()
                print(role + "重发post请求")
        self.header.seq_num = 0
    pass
# ---------- Connection ----------
# 处理数据报文
def deal_with_dat(self, header, packet):
    role = "客户端" if self.tag else "服务端"
    if self.get_tag == 1:
        # 接收方
        self.get_tag = 2
        self.header.dat = True
    if self.post_tag == 1:
        # 接收方
        self.post_tag = 2
        self.header.dat = True
    # 准备输入文件
    if header.ack and (self.get_tag == 2 or self.post_tag == 2):
        # 发送方
```

```
        # print(role + ": 接收到ACK为%d的数据包" % header.ack_num)
        if header.ack_num == self.send_window.least_not_ack():
            print(role + ": 接收到ACK为%d的数据包" % header.ack_num)
            self.send_window.ack_and_renew(header.ack_num)
            self.send_window.quick_update_cwnd(self.current_state,
new_ack=1)
            self.current_state = "slow"
            if header.ack_num in self.duplicate_ack.keys():
                del self.duplicate_ack[header.ack_num]
        else:
            if header.ack_num not in self.duplicate_ack.keys():
                self.duplicate_ack[header.ack_num] = 0
            self.duplicate_ack[header.ack_num] += 1
            self.duplicate_resend = self.send_window.quick_update_cwnd(
                self.current_state,
                duplicate_ack=self.send_window.least_not_ack() - 1,
                dupack_count=self.duplicate_ack[header.ack_num]
            )
            if self.duplicate_ack[header.ack_num] >= 3:
                self.current_state = "quick"
                self.duplicate_ack[header.ack_num] = 0
    elif self.get_tag == 2 or self.post_tag == 2:
        # 接收方
        # 是否重复 TODO
        # print(packet[header.header_len:].decode("utf-8"))
        if self.receive_window is None:
            self.receive_window = ReceiveWindow(self.out)
        self.header.ack_num, self.header.window_size, tag =
self.receive_window.return_message(header.seq_num,
packet[header.header_len:], header.end)
        # print(role + ": 接收到数据包，seq_num: ", header.seq_num, ": 数据
是", packet[header.header_len:], "传回dat ack报文")
        print(role + ": 接收到数据包，接收到seq_num: ", header.seq_num, "传回
dat ack报文，期望", self.header.ack_num)
        self.send_ack()
        if tag:
            self.resend_end_ack()
            self.header.dat = False
            self.header.operation = False
            self.get_tag = 0
            self.post_tag = 0
            self.out = None
            self.duplicate_resend = None
            self.duplicate_ack = dict()
            self.current_state = "slow"
            self.receive_window = None
            print(role + ": 接收到 dat end 数据包，结束传输，发送end ack")
    elif (self.get_tag == 2 or self.post_tag == 2) and header.end and not
header.ack:
        pass
```

```
    elif (self.get_tag == 0 or self.post_tag == 0) and header.end and not header.ack:
        # 接收方
        self.resend_end_ack()
        print(role + ": 重传 end ack 数据包")
    elif (self.get_tag == 3 or self.post_tag == 3) and header.end and header.ack:
        # 发送方
        self.header.end = False
        self.header.dat = False
        self.header.operation = False
        self.get_tag = 0
        self.post_tag = 0
        print(role + ": 接收到 end ack，结束连接")

# 处理指令报文
def deal_with_operation(self, header, packet):
    if header.operation:
        if self.tag and self.get_tag == 0 and header.ack:
            print("客户端：接收到get ack，确认服务端存在此文件，开始接收文件")
            self.get_tag = 1
        elif self.tag and self.get_tag == 0 and header.error == 1:
            print("客户端：接收到get error，文件不在服务器")
            self.error_tag = 1
        elif not self.tag and not header.ack and self.get_tag == 0:
            print("服务端：接收到get 报文，开始确认文件是否存在")
            path = "./server_files/" + packet[header.header_len:].decode("utf-8")
            if not os.path.exists(path):
                print("服务端：未找到文件，请输入正确的文件名")
                self.header.error = 1
                self.header.operation = True
                self.send()
                self.header.error = 0
                self.header.operation = False
            else:
                print("服务端：发送get ack，确认文件存在")
                self.get_tag = 1
                self.resend_get_ack()
        elif not self.tag and self.get_tag == 1 and not header.ack:
            print("服务端：重发get ack，确认文件存在")
            self.resend_get_ack()
        elif not self.tag and self.get_tag == 1 and header.ack:
            print("服务端：接收到get ack，开始文件传输")
            self.get_tag = 2
            read_and_send_thread = threading.Thread(
                target=self.read_and_send_file,
                args=("./server_files/" + packet[header.header_len:].decode("utf-8"), "服务端")
            )
            read_and_send_thread.start()
```

```
    else:
        if not self.tag and header.seq_num == 0:
            # 接收方
            self.header.ack_num = 0
            self.header.seq_num = 1
            self.post_tag = 1
            self.out = "./server_files/" +
packet[header.header_len:].decode("utf-8")
            self.send_ack()
            print("服务端：接收到post 报文，开始准备文件")
        elif self.tag and header.seq_num == 1 and header.ack:
            print("客户端：接收到post ack，开始文件传输")
            self.post_tag = 2
            read_and_send_thread =
threading.Thread(target=self.read_and_send_file, args=(self.out, "客户端"))
            read_and_send_thread.start()

# 读取文件和发送
def read_and_send_file(self, filename, role):
    self.send_window = SendWindow(filename)
    self.header.dat = True
    while self.send_window and self.send_window.least_not_ack() != -1:
        if self.duplicate_resend:
            try:
                self.header.seq_num = self.duplicate_resend.seq_num
                self.send(self.header, self.duplicate_resend.data)
                print("快速重传序号是", self.header.seq_num, "DATA IS",
self.duplicate_resend.data)
            except Exception as e:
                print("快速重。。。")
            self.duplicate_resend = None
        time_out_buffers = self.send_window.get_time_out_buffers()
        if len(time_out_buffers) > 0:
            self.send_window.quick_update_cwnd("slow", time_out=1)
            for buffer in time_out_buffers:
                print("重传序号是", buffer.seq_num, "DATA IS", buffer.data)
                self.header.seq_num = buffer.seq_num
                self.send(self.header, buffer.data)
        send_buffers = self.send_window.get_buffers()
        if send_buffers is None or len(send_buffers) == 0:
            continue
        # print(role + "发送数据")
        if not send_buffers == -1 and send_buffers is not None:
            for buffer in send_buffers:
                if random.random() < 0.01:
                    print("序号是", buffer.seq_num, "DATA IS", buffer.data)
                    self.header.seq_num = buffer.seq_num
                    self.send(self.header, buffer.data)
                # print(send_windows.cwnd, send_windows.ssthresh)
    if self.get_tag == 2:
        self.get_tag = 3
```

```
    if self.post_tag == 2:
        self.post_tag = 3
    self.header.end = True
    self.send()
    self.ttl_timer.start()
    print(role + ": 发送 dat end 报文")
    while self.get_tag == 3 or self.post_tag == 3:
        if self.ttl_timer.is_timeout():
            self.send()
            self.ttl_timer.start()
            print(role + ": 重发 dat end 报文")
    print("文件传输结束")
    self.print_state()

```

6. 接口对接与快速重传、重传、拥塞控制

将窗口与Connection结合起来就有重传、快速重传、拥塞控制了，所以在发送文件的时候将修改发送窗口和从发送窗口中获取数据，而在接受到ack后发送窗口也要修改，当然还有内置的超时修改。至于接收窗口，则是只在接收到数据时才会起作用，所以只要在deal_with_dat函数里面处理就行了。

```
# ---------- 使用发送窗口 ----------
# 读取文件和发送
def read_and_send_file(self, filename, role):
    self.send_window = SendWindow(filename)
    self.header.dat = True
    while self.send_window and self.send_window.least_not_ack() != -1:
        if self.duplicate_resend:
            try:
                self.header.seq_num = self.duplicate_resend.seq_num
                self.send(self.header, self.duplicate_resend.data)
                print("快速重传序号是", self.header.seq_num, "DATA IS",
self.duplicate_resend.data)
            except Exception as e:
                print("快速重。。。")
            self.duplicate_resend = None
        time_out_buffers = self.send_window.get_time_out_buffers()
        if len(time_out_buffers) > 0:
            self.send_window.quick_update_cwnd("slow", time_out=1)
            for buffer in time_out_buffers:
                print("重传序号是", buffer.seq_num, "DATA IS", buffer.data)
                self.header.seq_num = buffer.seq_num
                self.send(self.header, buffer.data)
        send_buffers = self.send_window.get_buffers()
        if send_buffers is None or len(send_buffers) == 0:
            continue
        # print(role + "发送数据")
        if not send_buffers == -1 and send_buffers is not None:
            for buffer in send_buffers:
```

```
                if random.random() < 0.01:
                    print("序号是", buffer.seq_num, "DATA IS", buffer.data)
                    self.header.seq_num = buffer.seq_num
                    self.send(self.header, buffer.data)
                # print(send_windows.cwnd, send_windows.ssthresh)
    if self.get_tag == 2:
        self.get_tag = 3
    if self.post_tag == 2:
        self.post_tag = 3
    self.header.end = True
    self.send()
    self.ttl_timer.start()
    print(role + ": 发送 dat end 报文")
    while self.get_tag == 3 or self.post_tag == 3:
        if self.ttl_timer.is_timeout():
            self.send()
            self.ttl_timer.start()
            print(role + ": 重发 dat end 报文")
    print("文件传输结束")
    self.print_state()
def deal_with_dat(self, header, packet):
  ...
  if header.ack and (self.get_tag == 2 or self.post_tag == 2):
      # 发送方
      # print(role + ": 接收到ACK为%d的数据包" % header.ack_num)
      if header.ack_num == self.send_window.least_not_ack():
          print(role + ": 接收到ACK为%d的数据包" % header.ack_num)
          self.send_window.ack_and_renew(header.ack_num)
          self.send_window.quick_update_cwnd(self.current_state,
new_ack=1)
          self.current_state = "slow"
          if header.ack_num in self.duplicate_ack.keys():
              del self.duplicate_ack[header.ack_num]
      else:
          if header.ack_num not in self.duplicate_ack.keys():
              self.duplicate_ack[header.ack_num] = 0
          self.duplicate_ack[header.ack_num] += 1
          self.duplicate_resend = self.send_window.quick_update_cwnd(
              self.current_state,
              duplicate_ack=self.send_window.least_not_ack() - 1,
              dupack_count=self.duplicate_ack[header.ack_num]
          )
          if self.duplicate_ack[header.ack_num] >= 3:
              self.current_state = "quick"
              self.duplicate_ack[header.ack_num] = 0
  ...
# ---------- 使用接收窗口 ----------
def deal_with_dat(self, header, packet):
elif self.get_tag == 2 or self.post_tag == 2:
    # 接收方
    # 是否重复 TODO
```

```
    # print(packet[header.header_len:].decode("utf-8"))
    if self.receive_window is None:
        self.receive_window = ReceiveWindow(self.out)
    self.header.ack_num, self.header.window_size, tag =
self.receive_window.return_message(header.seq_num,
packet[header.header_len:], header.end)
    # print(role + ": 接收到数据包，seq_num：", header.seq_num, "；数据是",
packet[header.header_len:], "传回dat ack报文")
    print(role + ": 接收到数据包，接收到seq_num：", header.seq_num, "传回dat
ack报文，期望", self.header.ack_num)
    self.send_ack()
    if tag:
        self.resend_end_ack()
        self.header.dat = False
        self.header.operation = False
        self.get_tag = 0
        self.post_tag = 0
        self.out = None
        self.duplicate_resend = None
        self.duplicate_ack = dict()
        self.current_state = "slow"
        self.receive_window = None
        print(role + ": 接收到 dat end 数据包，结束传输，发送end ack")
```

7. 修改命令行与一些极端事件避免

修改命令行：
我们与用户交互的命令行界面最开始是不一样的，甚至没有stop——退出功能，只是在最后确认已经完成了其他所有工作后才开始修改(完善)命令行界面的，所以有些截图的命令行根本不一样，在这里我将与用户交互的客户端进程的代码给出：

```
if __name__ == "__main__":
    # 自己地址、自己端口、server address、server port
    mess = ["127.0.0.1", 12001, "127.0.0.1", 10000]
    client = None
    client_listener = None
    while True:
        time.sleep(.500)
        print("type connect 'your ip_address' 'your ip_port' - to
establish connection \n"
              + "     get 'filename' - to download the file from server
\n"
              + "     post 'filename' - to upload the file to server \n"
              + "     disconnect - to close the connection\n"
              + "     stop - to stop the lftp\n"
              + "     instructions - to show the instruction.\n")
        command = input()
        args = command.split()
        args_len = len(args)
        print()
```

```
        if args_len == 3 and args[0] == "connect":
            if client is not None:
                print("The connection has been established.\n")
                continue
            ip_address = args[1]
            ip_port = args[2]
            ip_address_parts = ip_address.split(".")
            if len(ip_address_parts) != 4:
                print("Invalid IP address. Please try again.\n")
                continue
            flag = True
            for ip_address_part in ip_address_parts:
                try:
                    part_num = int(ip_address_part)
                except ValueError as e:
                    flag = False
                    break
                if not (0 <= part_num <= 255):
                    flag = False
                    break
            if not flag:
                print("Invalid IP address. Please try again.\n")
                continue
            try:
                ip_port = int(ip_port)
                if not (2000 <= ip_port <= 65535):
                    raise ValueError()
            except ValueError as e:
                print("Invalid IP port. Please try again.")
                continue
            client = TCPClient(ip_address, ip_port, mess[2], mess[3])
            client_listener = threads.RecvThread(client)
            client_listener.start()
            client.connect()
        elif args_len == 2 and args[0] == "get":
            if client:
                client.get_file(args[1])
            else:
                print("Please connect first")
        elif args_len == 2 and args[0] == "post":
            if client:
                client.post_file(args[1])
            else:
                print("Please connect first")
        elif command == "disconnect":
            if client:
                client.close()
                client_listener.stop()
                client.socket.close()
                client = None
                client_listener = None
```

```
            else:
                print("Please connect first")
        elif command == "stop":
            if client:
                client.close()
                client_listener.stop()
                client.socket.close()
                client = None
                client_listener = None
            break
        elif command == "instructions":
            continue
        else:
            print("The command is not correct")
        print()
    print("The lftp has stopped!")
```

顺便给出用于测试的服务端代码：

```
if __name__ == "__main__":
    # server = TCPServer("172.20.10.14", 10000)
    server = TCPServer("127.0.0.1", 10000)
    server_listener = threads.RecvThread(server)
    server_listener.start()
```

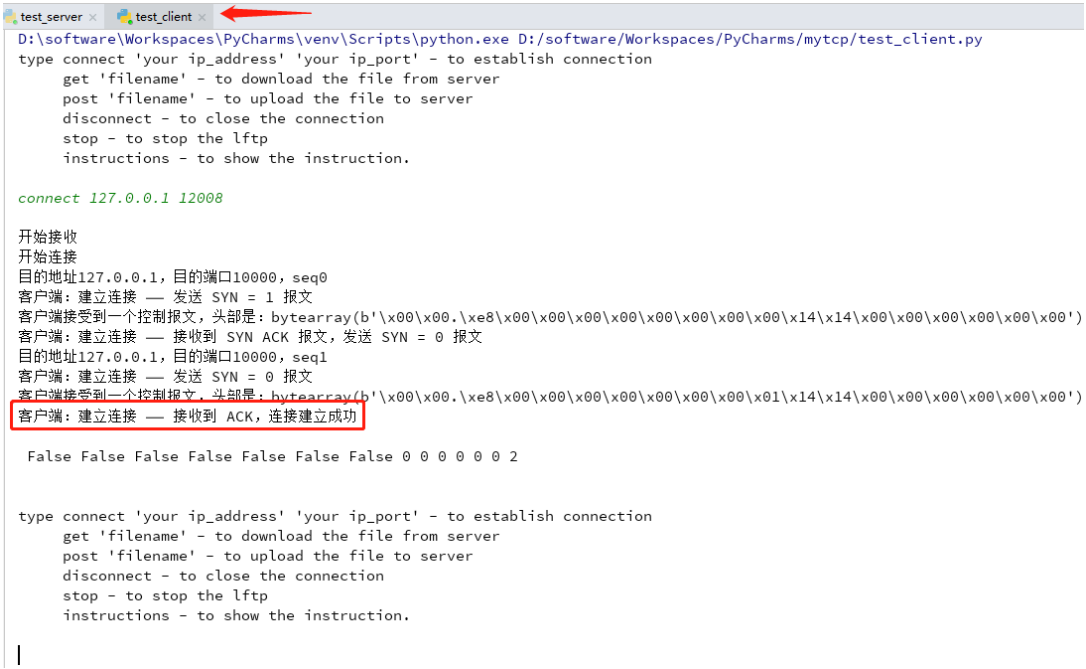
极端事件避免：

```
try:
      # 指令
except Exception as e:
      # 输出错误
```

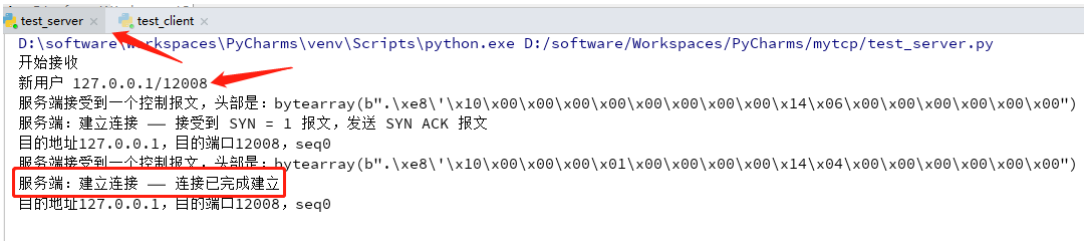
七、实验结果（截图）

三次握手

客户端：

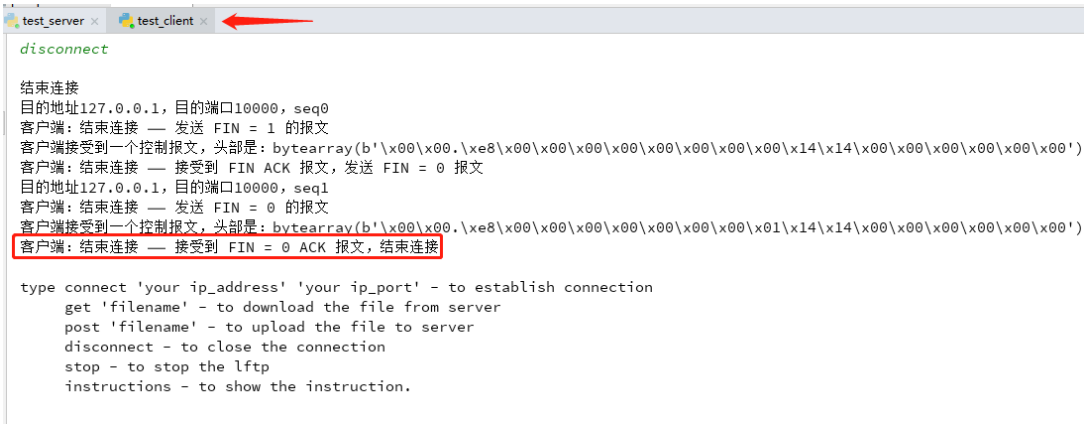


服务端：

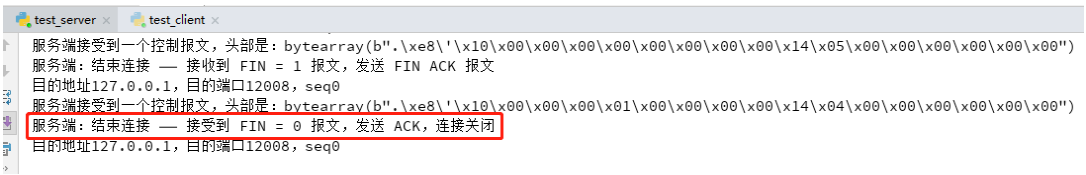


四次挥手

客户端：



服务端：



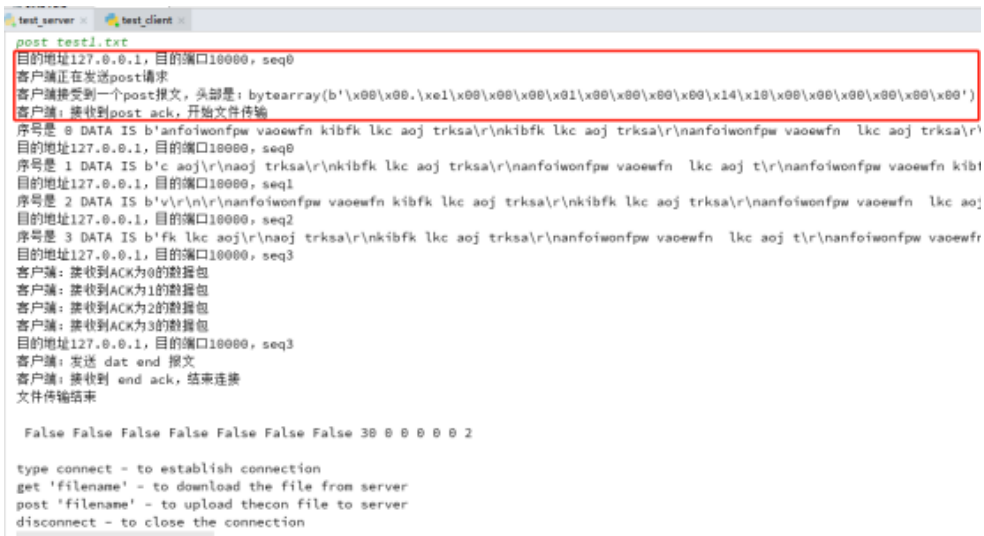
get

客户端：

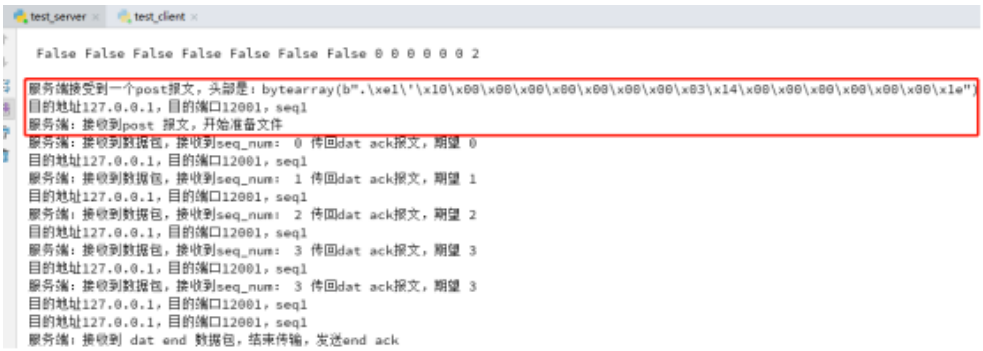
服务端：

post

客户端：

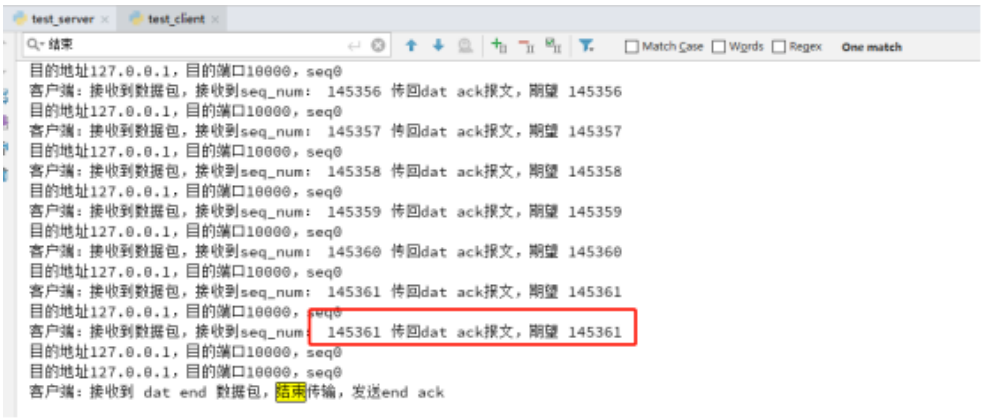


服务端：

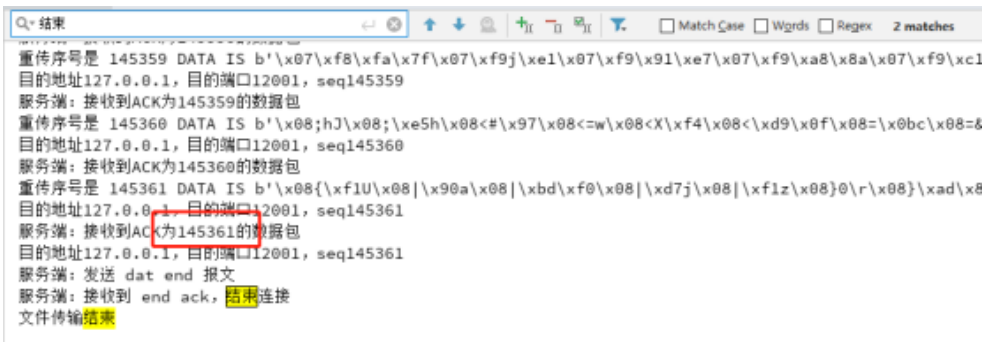


data文件传输结束

客户端：



服务端：



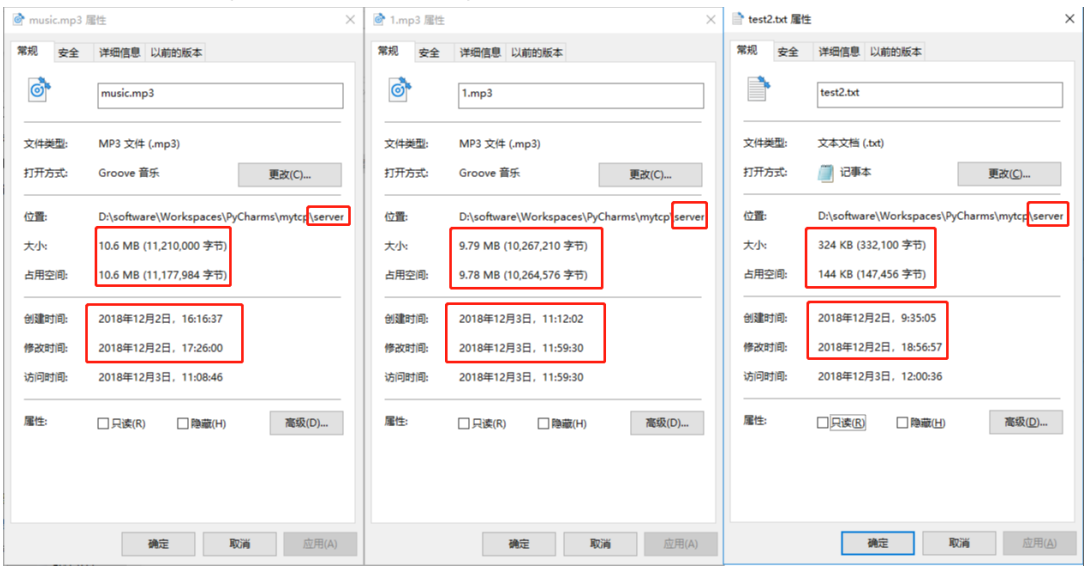
并发测试

一个客户端向服务端发送1.mp3，第二个客户端向服务端请求music.mp3，第三个客户端向服务端请求test2.txt。

服务端的输出报文：

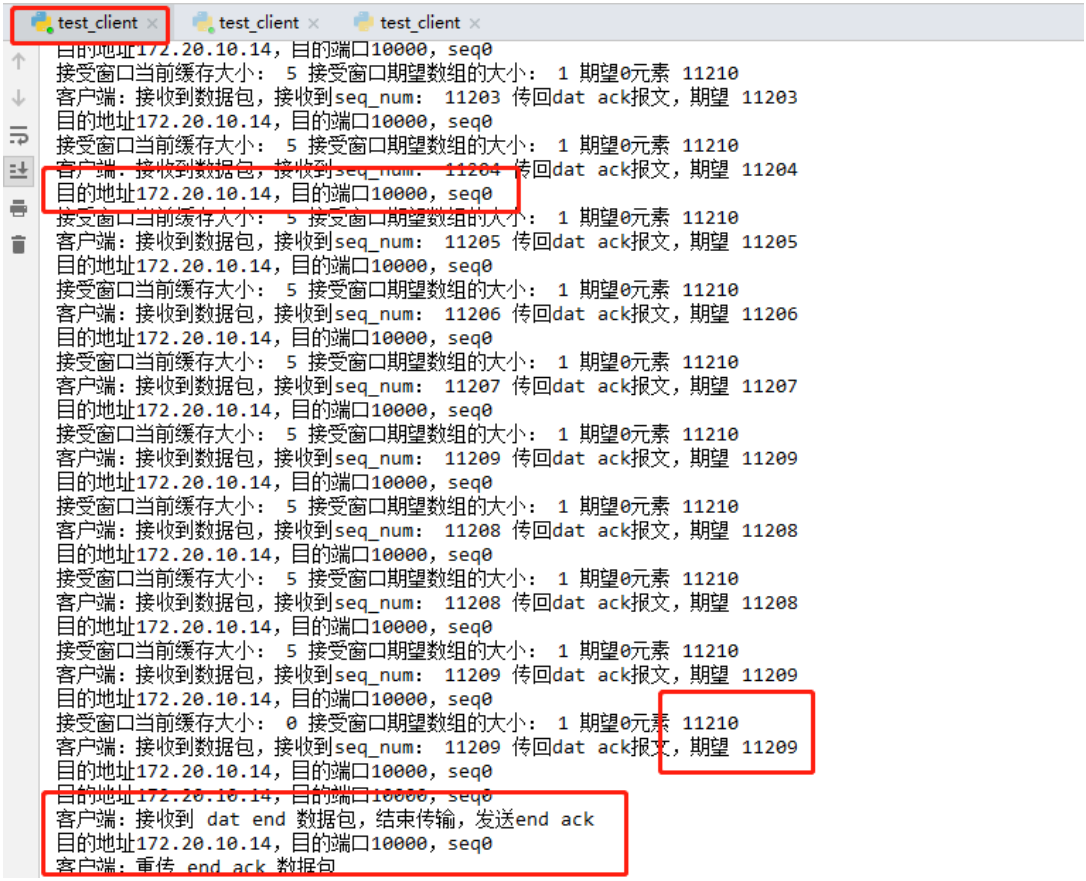

```
接受窗口当前缓存大小： 3 接受窗口期望数组的大小： 1 期望0元素 10268
服务端：接收到数据包，接收到seq_num： 10264 传回dat ack报文，期望 10264
目的地址172.20.10.2，目的端口12008，seq1
接受窗口当前缓存大小： 3 接受窗口期望数组的大小： 1 期望0元素 10268
服务端：接收到数据包，接收到seq_num： 10265 传回dat ack报文，期望 10265
目的地址172.20.10.2，目的端口12008，seq1
接受窗口当前缓存大小： 3 接受窗口期望数组的大小： 1 期望0元素 10268
服务端：接收到数据包，接收到seq_num： 10266 传回dat ack报文，期望 10266
目的地址172.20.10.2，目的端口12008，seq1
接受窗口当前缓存大小： 3 接受窗口期望数组的大小： 1 期望0元素 10268
服务端：接收到数据包，接收到seq_num： 10267 传回dat ack报文，期望 10267
目的地址172.20.10.2，目的端口12008，seq1
接受窗口当前缓存大小： 0 接受窗口期望数组的大小： 1 期望0元素 10268
服务端：接收到数据包，接收到seq_num： 10267 传回dat ack报文，期望 10267
目的地址172.20.10.2，目的端口12008，seq1
目的地址172.20.10.2，目的端口12008，seq1
服务端：接收到 dat end 数据包，结束传输，发送end ack
```

服务端接收到的1.mp3、发送的music.mp3、发送的test2.txt的属性：

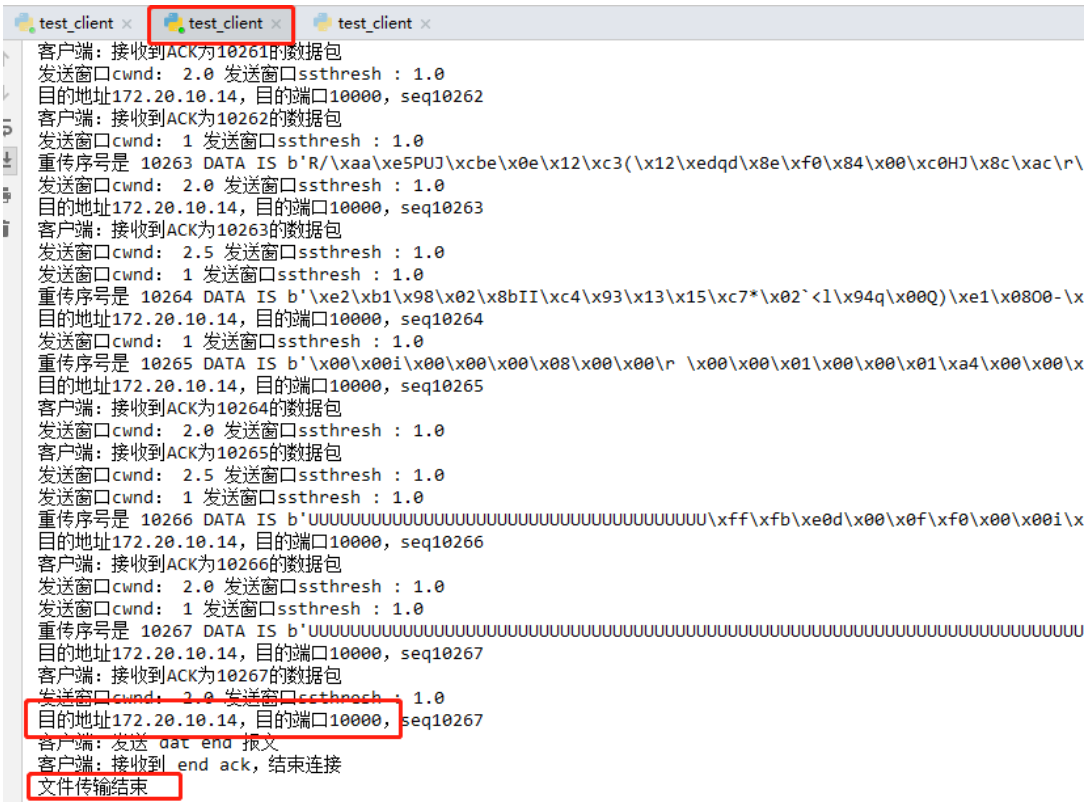


客户端：

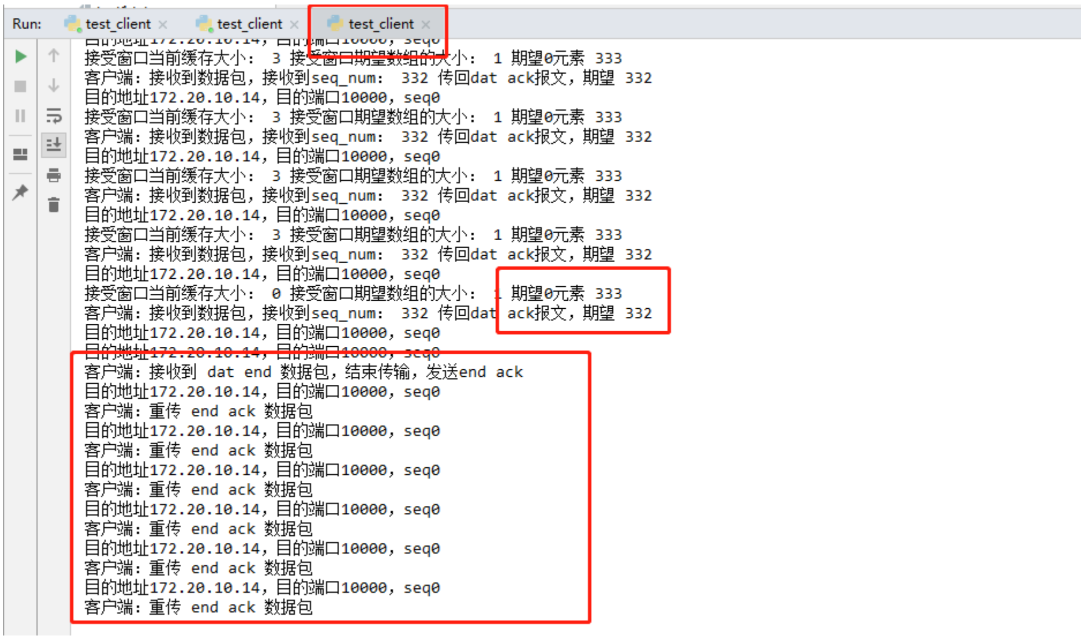
client 1：



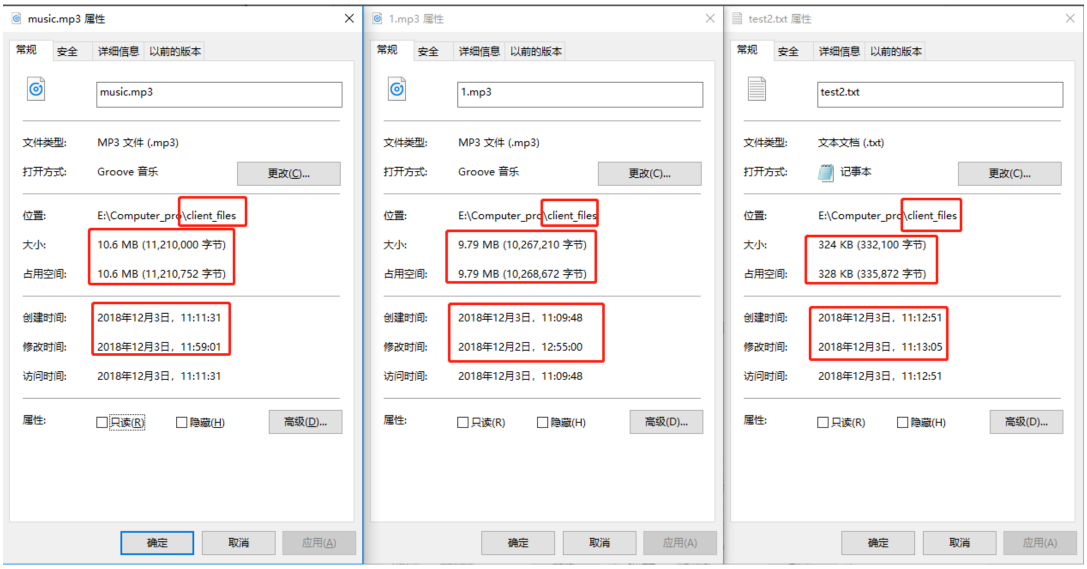
client 2：



client 3：

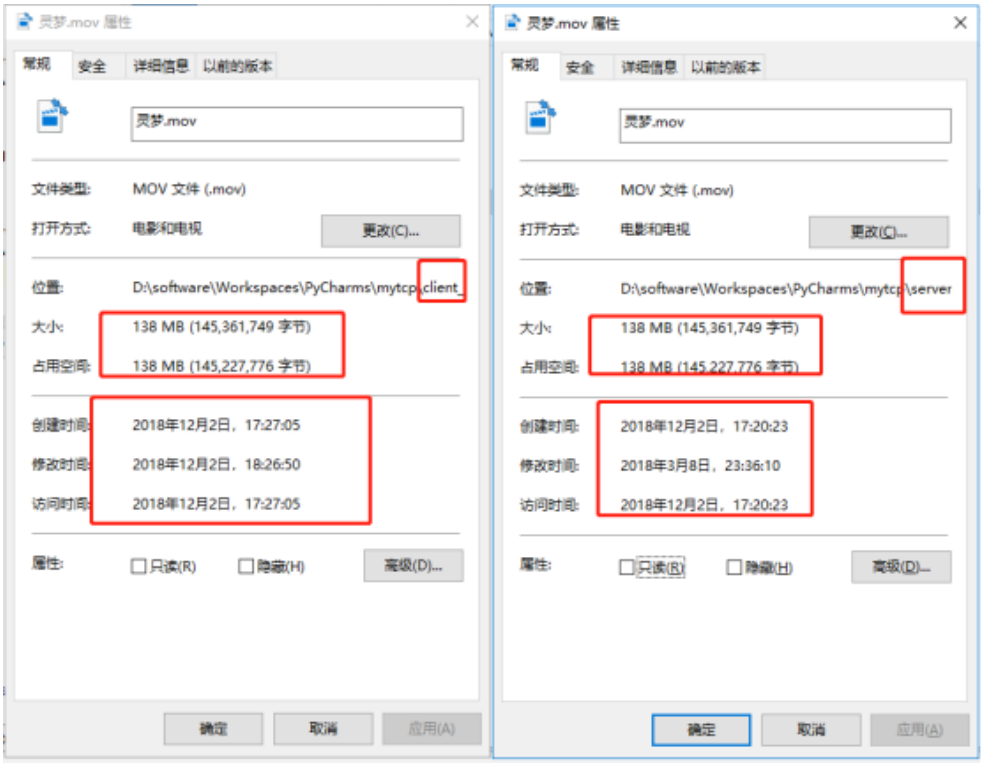


客户端发送的1.mp3、接收到的music.mp3、发送的test2.txt的属性：

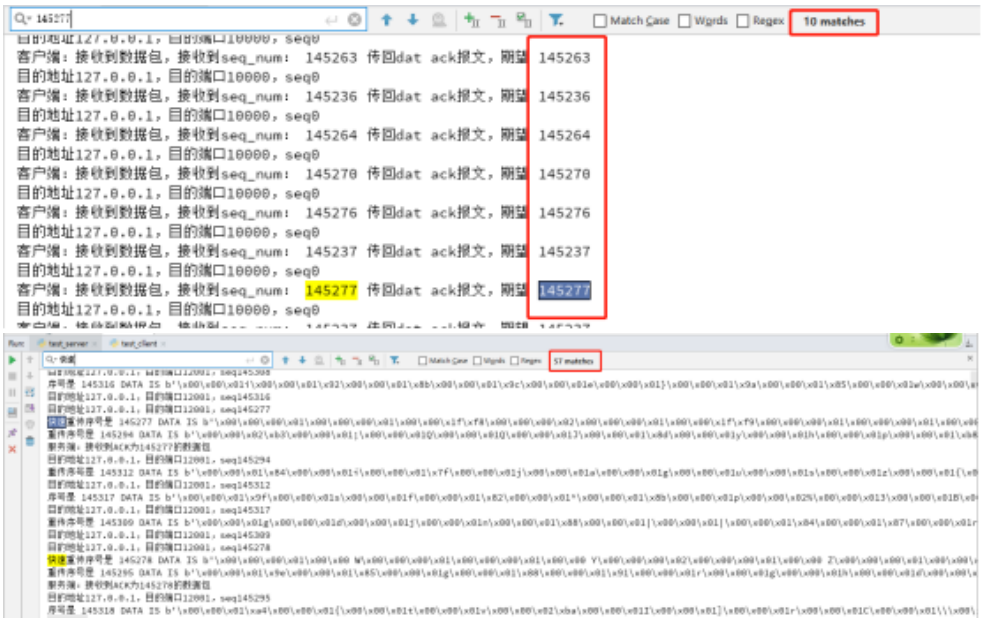


大文件测试

138 MB (145,361,749 字节)的大文件结果：

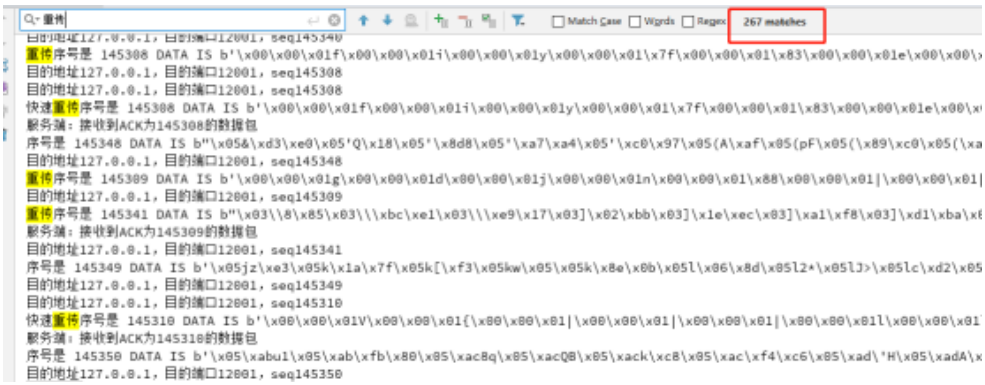


快速重传

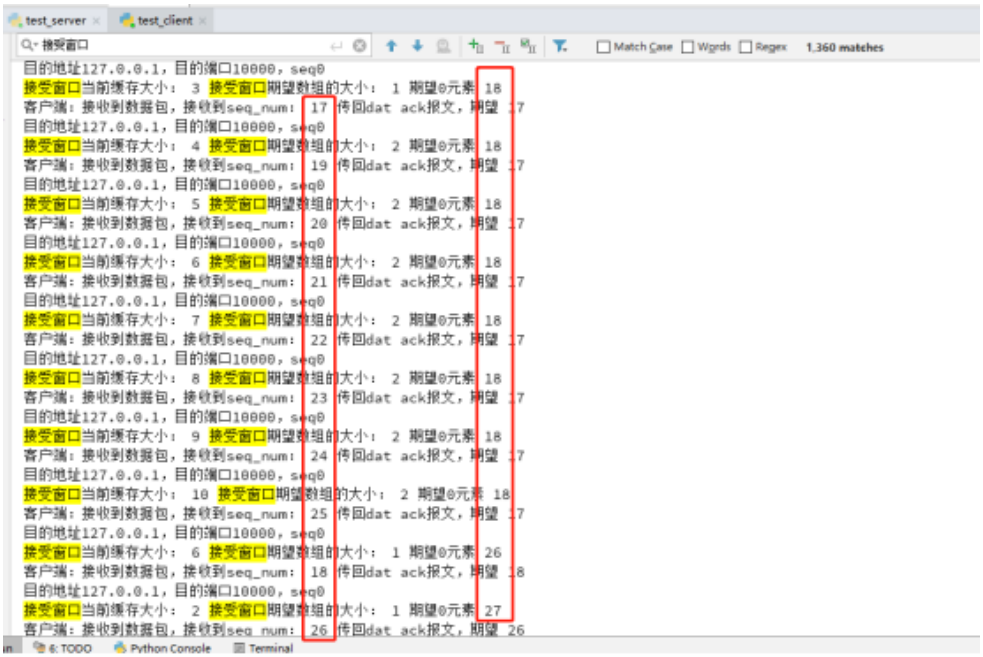


重传

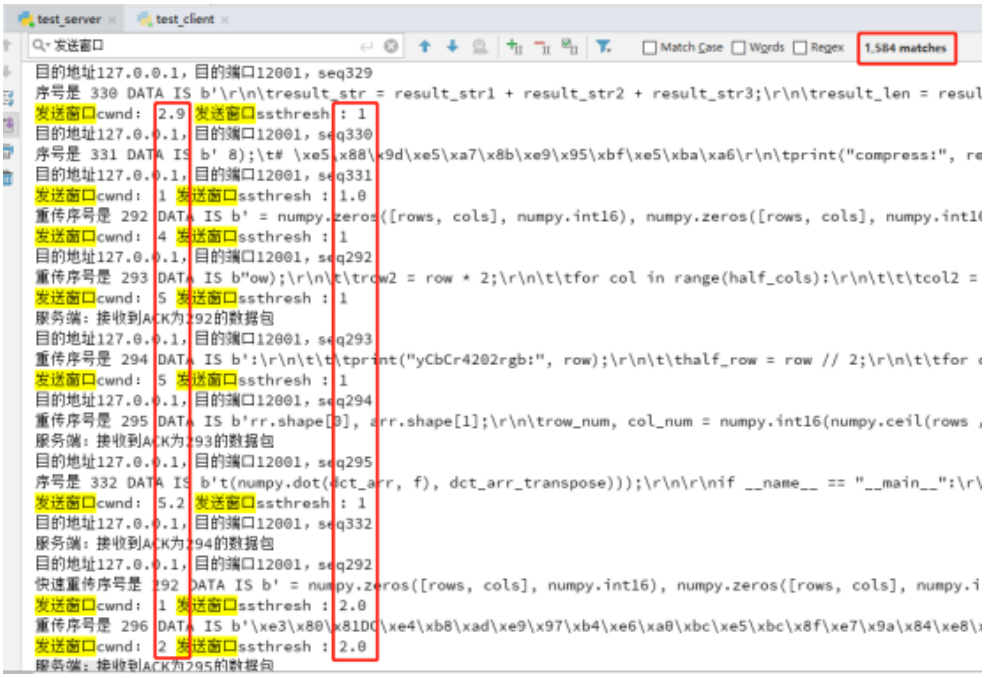
部分输出报文：



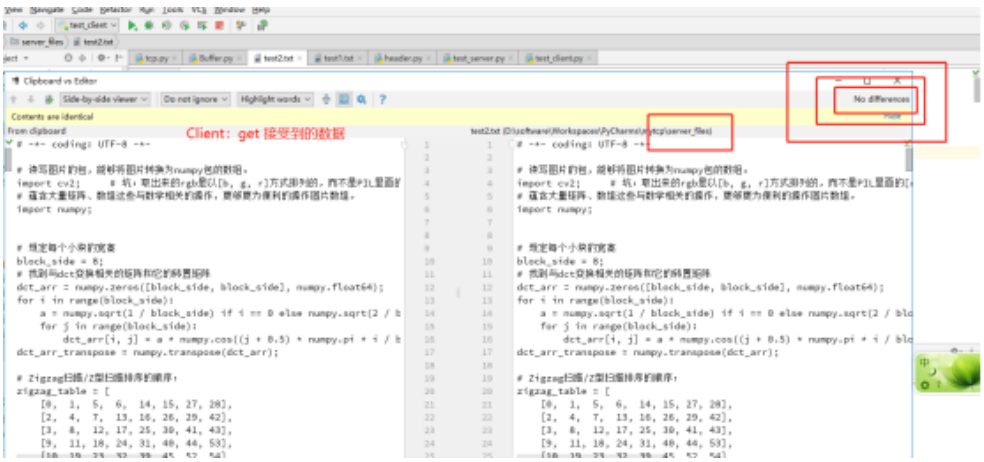
重传机制（SR+GBN）：接收窗口



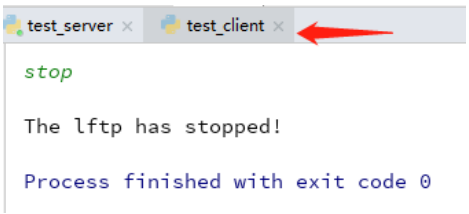
拥塞控制、发送窗口、拥塞窗口：



比对



退出客户端



八、测试指令格式

```
type connect 'your ip_address' 'your ip_port' - to establish connection
     get 'filename' - to download the file from server
     post 'filename' - to upload the file to server
     disconnect - to close the connection
     stop - to stop the lftp
     instructions - to show the instruction.
```

连接指令

connect ’your ip_address’ ’your ip_port’

get指令

get ’filename’

post指令

post ’filename’

断开连接指令

disconnect

结束程序指令

stop

九、实验思考

收获的知识

1. python如何进行多线程
2. python如何进行网路编程
3. 如何确保实现三次握手和四次挥手
4. 如何具体实现流控制、拥塞控制、快速重传等等有关可靠数据传输的具体知识

可能的bug

1. 文件处理线程中有发送窗口，用于操作数据发送，但是主线程在接受到ack时也会更新发送窗口，可能会造成发送窗口的问题，即使我们已经用try:execpt来处理了，但是极端情况下依然可能出现问题
2. 服务端没有真正实现多线程，如果有很多用户的时候可能会造成无限阻塞与丢包。

设想的改进

1. 在文件处理中将发送窗口的处理都放置到一个线程中，即使主线程接收到ack，也只是记录下来，然后在另一个线程中使用这个属性来修改发送窗口，即发送窗口的处理是线性的。
2. 在服务端真正的实现多线程，每个用户的Connection都继承threading.Thread类即可。
3. 在文件接收方的接收窗口中没有真正的将文件写入变为线程，可以考虑用multiprocessing库以及queue类来进行生产者消费者模型修改，另建一个线程来操作文件写入，收到数据包的接收窗口就是生产者，写入线程就是消费者。

实验的感想

梁毓颖

这次实验真的是让我全力以赴了，由于我其他作业太多，我只能在最后几天来完成这个作业，那时心里真的非常慌，感觉是一边慌一边敲代码的。甚至为了尽快完成，尽快入门这个实验，我是熬夜到5点把三次握手和四次挥手给完成的(刚开始敲代码，什么都不懂，书本的知识都没运用到，连丢包都没有考虑到)，然后第二天8点半就又起来敲代码了，这样的日子持续了两三天，现在想来还是心有余悸。当然，成果也是不错的，完整的实现了基于UDP实现的TCP上的FTP文件传输(当然不是完全按照书本来，我的代码能力有限，只能按照自己的实际情况进行修改了)。这几天其实前两天就基本实现了，第三天的bug竟然是因为我和队友的对接不是那么的很好，我完全没有用对他提供给我的接口，于是又拖了一天才完成啊，这还只是两三个函数的使用啊，果然我们还是太缺少项目经验了。至于书本的内容，别的我不敢说看，但是就TCP这一部分我想我是更为了解了，应该很久都不会忘记了。

马佳

这一次实验可能让自己很长时间内都不会忘了TCP与UDP的报文格式以及拥塞控制流控制、拥塞避免等。平时看了很多遍的书都没有自己亲手写一份代码更能够让自己记住。这一次实验过程做出的最惊险的事情就是一开始测试一不小心使用自己的代码作为传输文件结果乱掉了，还好能够及时恢复，第二惊险的就是手动设置丢包率达到99%，但是让人兴奋的是自己的代码没有让自己失望。这一次实验自己没有考虑到多线程的问题，因此在设计窗口的时候忘记了很多并发现象，也就是多个线程同时操纵同个对象，所以也引发了挺多的事故，后来一一进行修复。然后就是对于多线程还是不了解。觉得完成的比较快的原因是因为一开始与队友进行了很长的框架讨论积极配合以及队友的带飞。尽管提供的接口在一开始的使用中会用错，但是还是完成得比较好的，配合还是十分重要的，看书也同样是十分重要的。

十、参考资料

1. Python 网络编程
2. 一行 Python 实现并行化 -- 日常多线程操作的新思路
3. transmission control protocol
4. TCP vs UDP