# 实验报告

LFTP应用可以支持两台主机之间大文件的传输

## 功能要求

1. CS架构
2. 客户端可以上传或者下载大文件：

   LFTP lsend myserver mylargefile

   LFTP lget myserver mylargefile

   **myserver can be a url address or an IP address.**
3. 使用无连接传输的UDP协议，并达到和TCP一样的100%可靠运输
4. 流控制和拥塞控制
5. 具有多路复用和多路分解的功能，即支持多用户同时上传或者下载
6. 提示有意义的调试信息

## 具体设计

实现语言：python

### 一、套接字socket

使用Python 提供的 socket 模块

套接字格式：socket(family, type[,protocal]) 使用给定的套接族，套接字类型，协议编号（默认为0）来创建套接字

| socket 类型 | 描述 |
| --- | --- |
| socket.AF_UNIX | 用于同一台机器上的进程通信（既本机通信） |
| socket.AF_INET | 用于服务器与服务器之间的网络通信 |
| socket.AF_INET6 | 基于IPV6方式的服务器与服务器之间的网络通信 |
| socket.SOCK_STREAM | 基于TCP的流式socket通信 |
| socket.SOCK_DGRAM | 基于UDP的数据报式socket通信 |
| socket.SOCK_RAW | 原始套接字，普通的套接字无法处理ICMP、IGMP等网络报文，而SOCK_RAW可以；其次SOCK_RAW也可以处理特殊的IPV4报文；此外，利用原始套接字，可以通过IP_HDRINCL套接字选项由用户构造IP头 |
| socket.SOCK_SEQPACKET | 可靠的连续数据包服务 |

创建UDP Socket：

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

发送数据：因为UDP是面向无连接的，每次发送都需要指定发送给谁。

| Socket 函数 | 描述 |
| --- | --- |
| s.sendto(string[, flag], address) | 发送UDP数据，将数据发送到套接字，address形式为tuple(ipaddr, port)，指定远程地址发送，返回值是发送的字节数 |
| s.recvfrom(bufsize[, flag]) | 接受UDP套接字的数据，与recv()类似，但返回值是tuple(data, address)。其中data是包含接受数据的字符串，address是发送数据的套接字地址 |
| s.bind(address) | 将套接字绑定到地址，在AF_INET下，以tuple(host, port)的方式传入，如s.bind((host, port)) |
| s.close() | 关闭套接字 |

# 二、报文首部

由于UDP报文首部只有4个字段，不足以支持字节流传送时的序号和确认号，故根据TCP报文首部添加其他字段。

**将首部信息和数据字段存入字典，并转换为字节码使用UDP套接字进行传送。**

| 首部字段 | 长度 /位 |
| --- | --- |
| 首部长度 | 4 |
| 字节流序号 | 32 |
| 确认号 | 32 |
| 接收窗口大小 | 16 |
| 确认位 | 1 |
| 同步位 | 1 |
| 终止位 | 1 |
| 选项长度（文件名，操作类型） | 8 |

数据字段首先为选项内容，之后为文件具体的数据内容（不超过MSS）。

```
def int2bits(value,length):
    return bytes(bin(value)[2:].zfill(length),encoding='utf-8')

# 字典转二进制码
def dict2bits(dict):
    bitstream = b''

    # 4位首部长度
    if "LENGTH" in dict:
        bitstream += int2bits(dict["LENGTH"],4)
    else:
        bitstream += int2bits(0,4)

    # 字节流序号，32位
    if "SEQ_NUM" in dict:
        bitstream += int2bits(dict["SEQ_NUM"],32)
    else:
        bitstream += int2bits(0,32)

    # 确认号，32位
    if "ACK_NUM" in dict:
        bitstream += int2bits(dict["ACK_NUM"],32)
    else:
        bitstream += int2bits(0,32)

    # 接收窗口大小，16位
    if "recvWindow" in dict:
        bitstream += int2bits(dict["recvWindow"],16)
    else:
        bitstream += int2bits(0,16)
```

```
    # 确认位
    if "ACK" in dict:
        bitstream += dict["ACK"]
    else:
        bitstream += b'0'

    # 同步位
    if "SYN" in dict:
        bitstream += dict["SYN"]
    else:
        bitstream += b'0'

    # 终止位
    if "FIN" in dict:
        bitstream += dict["FIN"]
    else:
        bitstream += b'0'

    # 第88位
    bitstream += b'0'

    # 选项长度，8位
    if "OPT_LEN" in dict:
        bitstream += int2bits(dict["OPT_LEN"],8)
    else:
        bitstream += int2bits(0,8)

    # 选项内容
    if "OPTIONS" in dict:
        bitstream += dict["OPTIONS"]

    # 报文数据字段
    if "DATA" in dict:
        bitstream += dict["DATA"]

    return bitstream
```

```
# 二进制转字典，解析报文
def bits2dict(bitstream):
    dict = {}
    dict["LENGTH"] = int(bitstream[0:4],2)
    dict["SEQ_NUM"] = int(bitstream[4:36],2)
    dict["ACK_NUM"] = int(bitstream[36:68],2)
    dict["recvWindow"] = int(bitstream[68:84],2)
    dict["ACK"] = bytes(str(bitstream[84] - 48),encoding='utf-8')
    dict["SYN"] = bytes(str(bitstream[85] - 48),encoding='utf-8')
    dict["FIN"] = bytes(str(bitstream[86] - 48),encoding='utf-8')
    dict["OPT_LEN"] = int(bitstream[88:96],2)
```

```
    dict["OPTIONS"] = bitstream[96:96+dict["OPT_LEN"]]
    dict["DATA"] = bitstream[96+dict["OPT_LEN"]:]
    return dict
```

## 三、线程管理

由于使用GBN流水线协议，数据接收端使用同一个套接字对收到的报文进行判断并作出ack响应；在数据的发送端使用两个套接字，一个用于接收接收端返回的ack，一个用于给接收端发送数据包。

使用python的threading模块为数据接收端创建一个子线程，数据发送端创建两个子线程。

```
thread = threading.Thread(target=thread_job,)   # 定义线程
thread.start()  # 让线程开始工作
```

- start():启动线程活动。
- **join([time]):** 等待至线程中止。这阻塞调用线程直至线程的join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。

服务端由于要处理多用户同时请求，需要为每次连接请求创建新线程，并且不需要添加join，使其并行执行。

服务端在10000端口监听连接请求，并循环为每次请求返回不同的可用端口：

```
# 服务端监听请求连接的套接字
recv_sock = socket(AF_INET, SOCK_DGRAM)
recv_sock.bind(SER_ADDR)
print("Server start to work on address",SER_ADDR)
serverListend = True

# 支持并发请求
while serverListend:
    # 接收客户端的请求
    data , address = recv_sock.recvfrom(BUFSIZE)
    packet = bits2dict(data)
    print("CLIENT address",address)
    # print("SYN ",packet["SYN"])

    # 判断第一次是否为建立连接
    if packet["SYN"] == b'1':
        # 得到客户端的具体请求选项，如文件名和操作类型
        jsonOptions = packet["OPTIONS"].decode("utf-8")
        jsonOptions = json.loads(jsonOptions)
        # 请求文件名
        FILENAME = jsonOptions["filename"]
        # 请求操作
        OPERATION = jsonOptions["operation"]
```

```
        # 客户端缓存
        RecvBuffer = packet["recvWindow"]
        print("CLIENT
REQUEST:filename,operation,RecvBuffer:\n",FILENAME,OPERATION,RecvBuffer)

        # 返回服务端进行数据传送新的可用端口，不同于第一次连接的端口，每次请求返回的端口
号确保不同
        replyPort = bytes(json.dumps({"replyPort":APP_PORT}),encoding = 'utf-8')
        recv_sock.sendto(replyPort,address)

        # 子线程处理下载请求
        if OPERATION == "lget":
            # 文件发送方共享的ack队列
            transferQueue = queue.Queue()
            # 发送方接收ACK的线程
            recv_thread = threading.Thread(target = TransferReceiver,args =
(APP_PORT,transferQueue,))
            # 发送方发送文件内容的线程
            send_thread = threading.Thread(target = TransferSender,args =
(APP_PORT+1,transferQueue,FILENAME,(address[0],address[1]),RecvBuffer,))

            # 更新新的端口处理其他请求
            APP_PORT += 2
            recv_thread.start()
            send_thread.start()

        # 子线程处理上传请求
        elif OPERATION == "lsend":
            # 文件接收方的线程
            receiver_thread = threading.Thread(target = fileReceiver,args =
(APP_PORT,(address[0],address[1]),FILENAME,RecvBuffer,))
            # 更新新的端口处理其他请求APP_PORT += 1
            receiver_thread.start()

recv_sock.close()
```

客户端首先将具体命令选项和缓存区大小发给服务端并收到服务端返回的可用端口：

```
if __name__ == '__main__':
    # 根据命令行参数进行初始化
    if len(sys.argv)>=4:
        OPERATION = sys.argv[1]
        SER_IP = sys.argv[2]
        FILENAME = sys.argv[3]
    else:
        print("The default parameter is "+OPERATION+" "+SER_IP+" "+FILENAME)
        print('''usage: LFTP
```

```
                        OPERATION [lsend | lget]
                        SER_ADDR 'server_ip_addr'
                        FILENAME 'test.mp4' ''')
        # sys.exit(0)

    SER_ADDR = (SER_IP,SER_PORT)

    # 用户选项
    jsonOptions =
bytes(json.dumps({'filename':FILENAME,"operation":OPERATION}),encoding="utf-8")

    # 建立连接的报文首部信息
    dict = {}
    dict["SYN"] = b'1'
    dict["OPTIONS"] = jsonOptions
    dict["OPT_LEN"] = len(jsonOptions)
    dict["recvWindow"] = RecvBuffer

    # 客户端UDP套接字
    send_sock = socket(AF_INET,SOCK_DGRAM)
    send_sock.bind(('',CLI_PORT))

    # 发送连接请求
    size = send_sock.sendto(dict2bits(dict),SER_ADDR)
    print("Send size: ",size)
    print("Server address: ",SER_ADDR)

    # 等待服务端返回可用端口
    data , address = send_sock.recvfrom(BUFSIZE)
    replyPort = json.loads(data.decode('utf-8'))['replyPort']
    print("The server reply the port avalible at :",replyPort)
    send_sock.close()

    # 处理下载
    if OPERATION == "lget":
        receiver_thread = threading.Thread(target = fileReceiver,args = (CLI_PORT,
(SER_IP,replyPort),FILENAME,RecvBuffer,))
        receiver_thread.start()
        # 阻塞执行
        receiver_thread.join()

    elif OPERATION == "lsend":
        # 发送方共享的ack队列
        transferQueue = queue.Queue()
        # 发送方接收ACK的线程
        recv_thread = threading.Thread(target = TransferReceiver,args =
(CLI_PORT,transferQueue,))
        # 发送方发送文件内容的线程
```

```
        send_thread = threading.Thread(target = TransferSender,args =
(CLI_PORT+1,transferQueue,FILENAME,(address[0],replyPort),RecvBuffer,))
        recv_thread.start()
        send_thread.start()
        recv_thread.join()
        recv_thread.join()
```

## 四、用queue进行通信

Python的Queue模块提供一种适用于多线程编程的FIFO实现。它可用于在生产者(producer)和消费者(consumer)之间线程安全(thread-safe)地传递消息或其它数据，因此多个线程可以共用同一个Queue实例。

基本操作：

1. q.put(10)：在队尾插入一个项目。put()有两个参数，第一个item为必需的，为插入项目的值；第二个block为可选参数，默认为1。如果队列当前为满且block为1，put()方法就使调用线程暂停,直到空出一个数据单元。如果block为0，put方法将引发Full异常。
2. q.get()：从队头删除并返回一个项目。可选参数为block，默认为True。如果队列为空且block为True，get()就使调用线程暂停，直至有项目可用。如果队列为空且block为False，队列将引发Empty异常。
3. q.get([block[, timeout]])：等待时间timeout后获取队头。

可以通过一个空的时间队列控制数据接收端文件写入的速度，将收到的数据首先存入缓存中，每隔一段时间get空的时间队列，抛出empty异常，然后将缓存中的数据写入文件中：

```
# 硬盘每0.5s进行一次写操作
FileWriteInterval = 0.5
# 写文件线程：d为接收数据缓存，timeQueue为空的时间队列
def fileWriter(filename,d,timeQueue,shaVar):

    f = open(filename,"ab+")

    # 文件未写完
    while not shaVar["fileWriterEnd"]:
        try:
            # 每隔FileWriteInterval获取空的队列
            q = timeQueue.get(timeout = FileWriteInterval)
        except queue.Empty:# 抛出异常
            while len(d)>0:
                packet = d.popleft()
                # 更新LastByteRead变量
                shaVar["LastByteRead"] = packet["SEQ_NUM"]
                # 写入数据
                f.write(packet["DATA"])
                # print(LastByteRead)
```

```
                # print(packet["DATA"])
            print("wait to write")
    f.close()
    print("write finish")
```

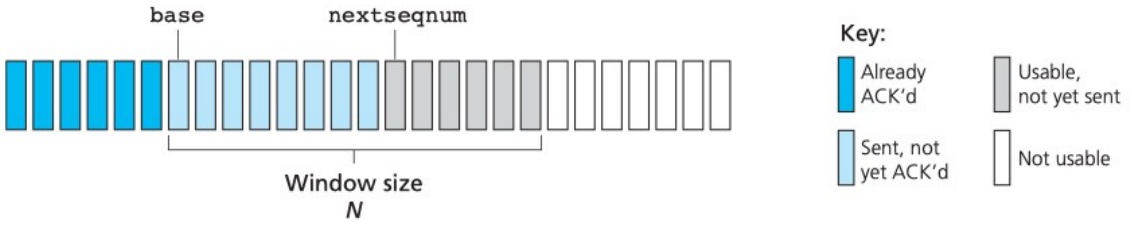
文件接收端的线程fileReceiver：**使用GBN协议解决流水线的差错恢复**（接收方丢弃失序分组）

文件接收端会循环判断接收的seq序号，如果为期望得到的，则加入缓存队列，并发送对应的ack给发送方；否则丢弃（即不加入缓存）并发送上一次得到的seq序号，直到收到FIN==1的数据包终止连接并停止文件写入。

```
def fileReceiver(port,ser_recv_addr,filename,RcvBuffer):

    # 初始化
    shaVar = {}
    shaVar["fileWriterEnd"] = False
    shaVar["LastByteRead"] = 0
    LastByteRcvd = 0

    # 绑定UDP端口
    recv_sock = socket(AF_INET,SOCK_DGRAM)
    recv_sock.bind(('',port))
    print(ser_recv_addr)

    # 期望得到的序号
    expectedSeqValue = 1
    # 计时器用于计算速度
    start_time = time.time()
    # 传输文件大小
    total_length = 0

    # 控制写入文件速度
    # 写文件线程
    # d为共享的接收数据缓存队列
    d = deque()
    timeQueue = queue.Queue()
    fileThread = threading.Thread(target=fileWriter,args=
(filename,d,timeQueue,shaVar,))
    fileThread.start()

    while True:
        data,addr = recv_sock.recvfrom(1024)
        packet = bits2dict(data)
        # 显示收到的seq_num
        print("receive packet with seq",packet["SEQ_NUM"])
        #随机丢包
        '''
```

```
        if random.random()>0.8:
            #print("Drop packet")
            continue
        '''
        # 如果收到FIN包，则终止
        if packet["FIN"] == b'1':
            print("receive FIN, file receiver close.")
            break

        # 按顺序得到
        elif packet["SEQ_NUM"] == expectedSeqValue:
            print("Receive packet with correct seq value:",expectedSeqValue)
            # 更新确认序号
            LastByteRcvd = packet["SEQ_NUM"]
            # 加入缓存用于文件写入
            d.append(packet)
            # print(packet["DATA"])
            # 更新总长度
            total_length += len(packet["DATA"])
            # print(RcvBuffer - (LastByteRcvd-shaVar["LastByteRead"]))
            # 返回ack和接收缓存大小

 recv_sock.sendto(dict2bits({"ACK_NUM":expectedSeqValue,"ACK":b'1',"recvWindow":Rc
vBuffer - (LastByteRcvd-shaVar["LastByteRead"])}),ser_recv_addr)
            # 期望值+1
            expectedSeqValue += 1

        #收到了不对的包，则返回expectedSeqValue-1，表示在这之前的都收到了
        else:
            print("Expect ",expectedSeqValue," while receive",packet["SEQ_NUM"],"
send ACK ",expectedSeqValue-1,"to receiver ",ser_recv_addr)
            recv_sock.sendto(dict2bits({"ACK_NUM":expectedSeqValue-
1,"ACK":b'1',"recvWindow":RcvBuffer - (LastByteRcvd-
shaVar["LastByteRead"])}),ser_recv_addr)


    # 跳出循环
    shaVar["fileWriterEnd"]  = True
    end_time = time.time()
    total_length/=1024
    total_length/=(end_time-start_time)
    print("Transfer speed",total_length,"KB/s")
```

## 五、GBN协议解决流水线的差错恢复




上图中有2个变量和1个固定的 Window size N，它们所表示的含义：

N：pipeline 中最多的 unacknowledged packets 数量

base：sequence number of the oldest unacknowledged packet

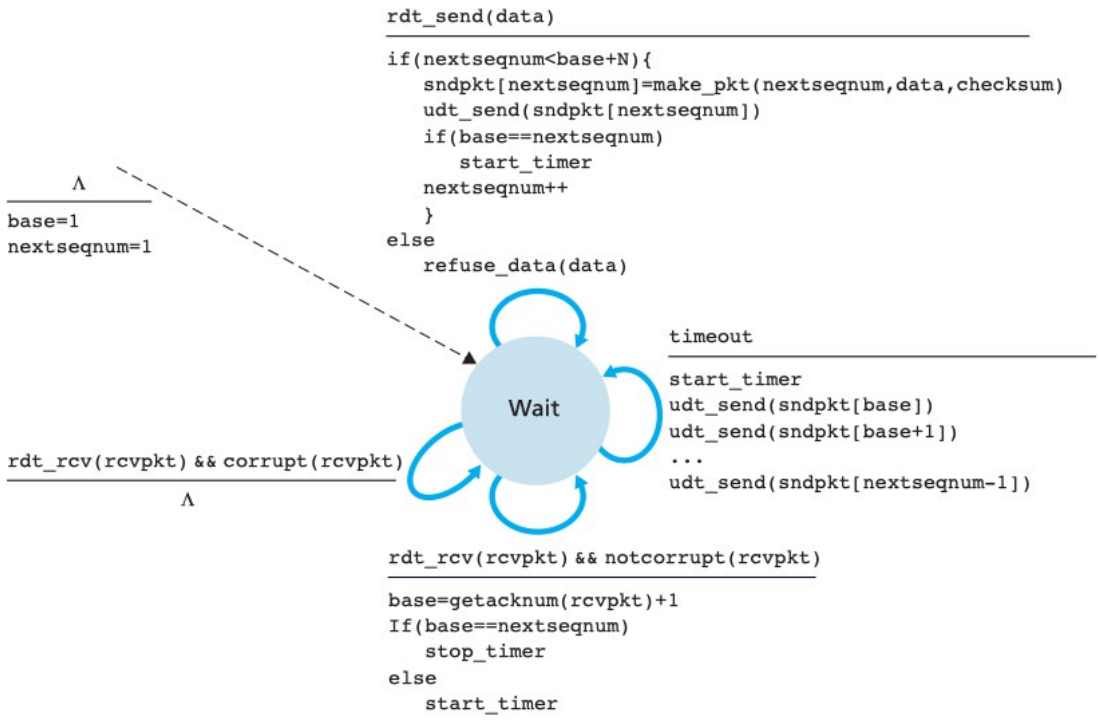
nextseqnum：smallest unused sequence number

**其中N为拥塞窗口cwnd和接收窗口rwnd中的较小者**

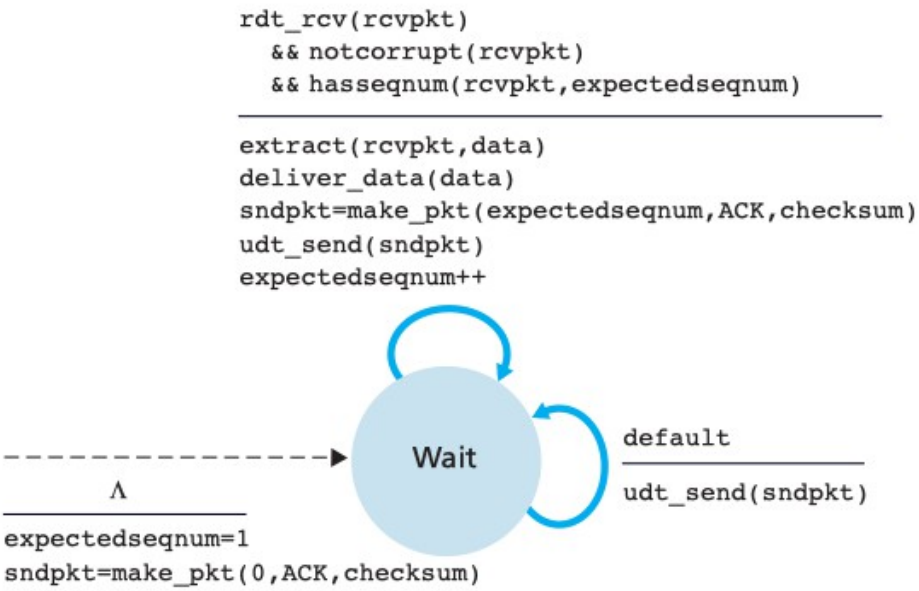
处理三种事件：

1. 判断是否大于窗口长度限制：加入流控制和拥塞控制后，**要小于拥塞窗口cwnd和接收窗口rwnd中的较小者**
2. 累计确认：**只有每次得到的ack为base时才会将base值加1，否则不滑动窗口，即发送缓存队列**
3. 出现超时，重传base到nextseqnum之间的所有数据包。**注意发送方仅使用一个定时器，即当发送base==nextseqnum的数据包开始计时，只要收到一个相应的ack，就会重新计时。**

sender的FSM




receiver的FSM




## 六、流量控制

1. 数据接收方在建立连接时将接收缓存大小，发送给发送方。
2. 如前所述，接收方通过控制文件写入速度，记录当前写入的seq序号为LastByteRead，并根据按序收到的seq序号更新LastByteRcvd，并根据事先设置的接收缓存大小计算rwnd，每次将rwnd随ack序号一起返回给发送方。

## 七、拥塞控制

数据发送方建立两个线程并行运行，并使用队列共享得到的ack信息，其中一个为接收ack的TransferReceiver线程：

```
# 发送方接收ack线程
# port：接收端口
# receiveQueue：共享ack队列
def TransferReceiver(port,receiveQueue):

    receiverSocket = socket(AF_INET,SOCK_DGRAM)
    receiverSocket.bind(('',port))

    while True:
        data,addr = receiverSocket.recvfrom(1024)
        # print(addr)
        packet = bits2dict(data)
        # 加入ack队列
        receiveQueue.put(packet)
        print("receiver receive ack:",packet["ACK_NUM"])
        # print("receiver window size:",packet["recvWindow"])
```

```
    receiverSocket.close()
```

**另一个为发送端发送数据的线程TransferSender**

**注意这里使用的rwnd和cwnd单位都是1MSS而不是书中的字节长度。**

每一轮发送端会发送尽可能多的数据包直到超过min{rwnd,cwnd}，然后开始从receiveQueue中按照GBN的逻辑接受ack，其中有三种情况：

1. 按序到达（ack==base），更新base加1，更新计时器，更新上一个ack的值，如果ack为本轮发送的数据包中最后一个（判断base是否等于nextseqnum），说明此回合未发生冗余ack和超时的情况，判断是慢启动还是拥塞避免阶段，进行乘性或者线性增加cwnd。**注意如果是因为超过rwnd引起的，则在收到本轮全部ack后不增加cwnd值。**
2. 收到3个冗余的ack包，进入快速恢复状态，慢启动阈值设置为cwnd的一半（向下取整数），cwnd为更新后的阈值加3，并进入线性增长阶段，然后进行重传。
3. 如果收到的ack大于当前base，说明之前的包发生丢失，会发生超时情况，此时更新计时器，并进行重传，然后将cwnd置为1，慢启动阈值设置为cwnd的一半（向下取整数），并重新进入慢启动状态。

```
# 发送端发送数据线程
# port：发送端口
# receiveQueue：共享ack队列
# filename：文件名
# cli_addr：接收方地址
# rwnd：接收方缓存
def TransferSender(port,receiveQueue,filename,cli_addr,rwnd):
    send_sock = socket(AF_INET,SOCK_DGRAM)
    send_sock.bind(('',port))
    # print(cli_addr)

    ### 初始化
    # 发送报文数据字段最大字节长度
    MSS = 500
    # 发送方最大等待时延
    senderTimeoutValue = 0.5
    # 拥塞窗口大小
    cwnd = 1
    # 慢启动阈值
    ssthresh = 500
    # 重复ACK计数
    dupACKcount = 0
    # 最早没发送的
    nextseqnum = 1
    # 最早发送没收到的
    base = 1
    # 发送缓存
    cache = {}
```

```
# 已发没收到ACK的包
sendNotAck = 0
# 计时器
GBNtimer = 0
# 是否发完
sendContinue = True
# 是否继续发送
sendAvaliable = True
# 是否为流控制
ClientBlock = False
# 1为指数增长；2为线性增长
congestionState = 1

...
```

发送数据的循环

```
...

f = open(filename,"rb")
while sendContinue:
    # 可以发送数据
    while sendAvaliable:
        # 更新已发没收到ACK的包
        sendNotAck = nextseqnum - base
        # 每一轮开始启动计时器
        if base == nextseqnum:
            GBNtimer = time.time()
        # 如果大于min（rwnd,cwnd）窗口长度，cache则满
        if sendNotAck >= cwnd:
            sendAvaliable = False
            print("已发没收到ACK的包 ",nextseqnum - base)
            print("当前cwnd大小：",cwnd)
        elif sendNotAck >= rwnd:
            sendAvaliable = False
            ClientBlock = True
            print("已发没收到ACK的包 ",nextseqnum - base)
            print("当前rwnd大小：",rwnd)
        else:
            # 每次读取报文中数据的字节长度
            data = f.read(MSS)
            # 文件读入完毕
            if data == b'':
                print("File read end.")
                sendAvaliable = False
                sendContinue = False
                break
            # 发送缓存(base,base+N),用于重传
            cache[nextseqnum] = dict2bits({"SEQ_NUM":nextseqnum,"DATA":data})
```

```
                send_sock.sendto(cache[nextseqnum],cli_addr)
                nextseqnum += 1
```

接收ack的循环，通过共享队列

```
# 等待接收ACK
        receiveACK = False
        # 前一个ACK
        previousACK = 0
        while not receiveACK:
            try:
                # ack队列，最多等待senderTimeoutValue
                receiveData = receiveQueue.get(timeout = senderTimeoutValue)
                # ack序号
                ack = receiveData["ACK_NUM"]
                # 接收窗口大小，用于流量控制
                rwnd = receiveData["recvWindow"]

                # 按顺序收到ACK
                if ack == base:
                    # 更新base
                    base = ack+1
                    # 更新计时器
                    GBNtimer = time.time()
                    # 更新已发未收到ACK的包的数量
                    sendNotAck = nextseqnum - base
                    # 记录上一个ack
                    previousACK = ack
                    dupACKcount = 1
                    # 一次RTT完成，未发生拥塞，根据状态增加cwnd
                    if base == nextseqnum:
                        # 所有上一轮ack确认收到
                        receiveACK = True
                        # 继续下一轮发送
                        sendAvaliable =True
                        if not ClientBlock:
                            # 乘性增长
                            if congestionState == 1:
                                cwnd *= 2
                            else:
                                cwnd += 1
                            # 到达阈值线性增长
                            if cwnd >= ssthresh and congestionState == 1:
                                cwnd == ssthresh
                                congestionState = 2
                        ClientBlock = False
                        break
                        # 开始下一轮发送
```

```
                # 收到重复ACK
                elif ack == previousACK:
                    dupACKcount += 1
                    # 进入快速恢复状态
                    if dupACKcount >= 3:
                        ssthresh = int(cwnd/2)
                        cwnd = ssthresh + 3
                        dupACKcount = 0
                        congestionState = 2
                        print("Three times duplicated ACK",previousACK*500,"
,resend now!")
                        # 进入重传
                        for i in range(base,nextseqnum):
                            packet = cache[i]
                            send_sock.sendto(cache[i],cli_addr)
                            print("Resend packet SEQ:",bits2dict(packet)
["SEQ_NUM"]*500)
                    continue

                # 没收到响应的ack，发生超时
                currentTime = time.time()
                if currentTime - GBNtimer > senderTimeoutValue and not
ClientBlock:
                    print("Time out and output current sequence number",base)
                    # 重启计时器
                    GBNtimer = time.time()
                    # 重传base到nextseqnum
                    for i in range(base,nextseqnum):
                        packet = cache[i]
                        send_sock.sendto(cache[i],cli_addr)
                        print("Resend packet SEQ:",bits2dict(packet)
["SEQ_NUM"]*500)

                    congestionState = 1
                    ssthresh = int(cwnd)/2
                    if ssthresh<=0:
                        ssthresh = 1
                    cwnd = 1

            # 超过rwnd大小引起的超时
            except queue.Empty:
                print("Send empty packet to update flow control value.")
                GBNtimer = time.time()
                # 发送空包等到接收方将更新后的rwnd返回
                send_sock.sendto(dict2bits({}),cli_addr)
                sendNotAck = nextseqnum - base
                # 可以继续发送
                if sendNotAck <= rwnd:
```

```
                    sendAvaliable = True
                    receiveACK = True
```

最后主动向接收端发送终止连接信息

```
    #关闭接受端与客户端
    send_sock.sendto(dict2bits({"FIN":b'1'}),cli_addr)
    send_sock.close()
    f.close()
    print("file sender closes")
```

# 测试结果