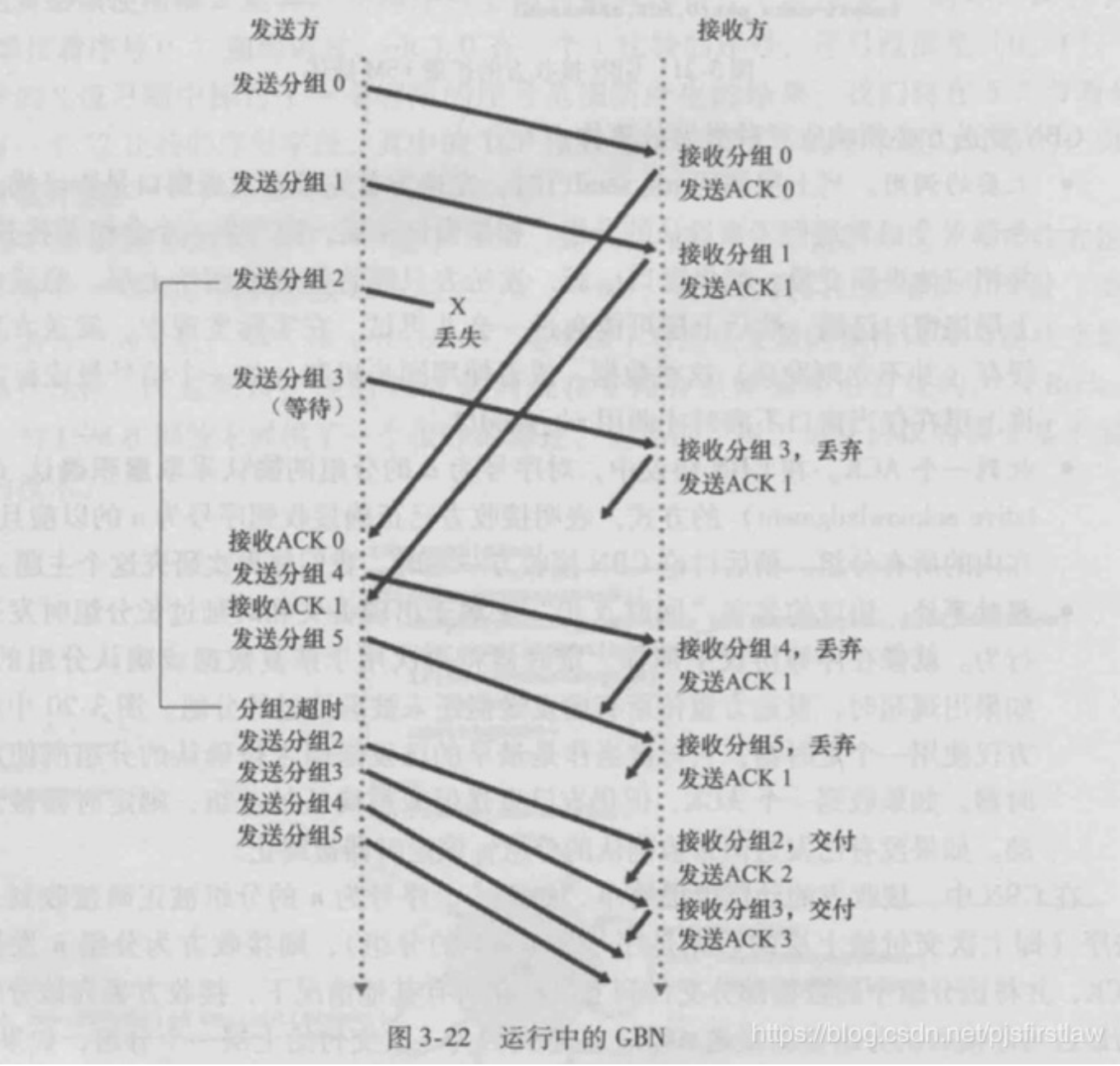
计网期中项目设计文档

写一个网络应用LFTP，该应用支持互联网中的两台计算机进行大文件传输

需求点实现

1. 使用UDP协议传输，但要求像TCP一样完全可靠

使用rdt3.0停等协议效率太慢，因此我们使用回退N步（GBN）的流水线协议，**允许发送方发送多个分组，而不需要等待确认。** UDP是不可靠传输的，它所收到的数据是无序的，也有可能在中途丢包。而在GBN协议中，**接收方丢弃所有失序分组，确保其像TCP一样可靠**。



运行时如上述所示，发送方一直在发送数据包，同时启动一个定时器，**只有当接收到回复的ACK包时，才取消定时器**。接收方如果收到是所期待的包，给发送方回一个ACK包，**ACK的值为当前所收到的包的序列号**。如果

接收的包不是所期待的包，则说明发生了丢包，则给发送方回一个ACK包，**ACK的值为期待的数据包的序列号减1**。由于发送方没有收到ACK包，则**触发超时事件，导致重发**。

也有可能是ACK包丢了，但这没有影响，因为发送方会根据下一个收到的ACK包来调整。比如ACK2丢失了，**但收到了ACK3，说明数据包2没有丢**。

实现方法：

发送方的实现方法参考以下的FSM图。

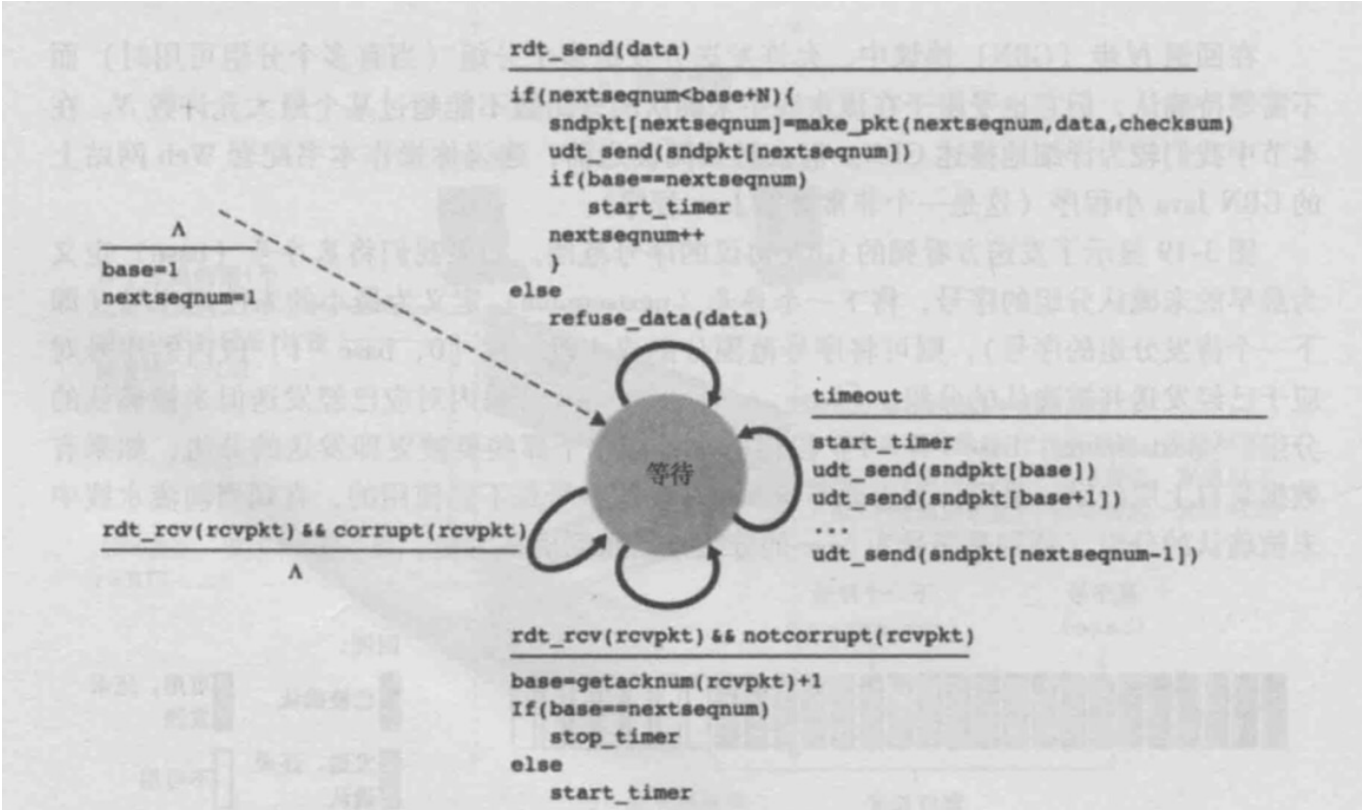


图 3-20　GBN 发送方的扩展 FSM 描述

发送方通过维护一个窗口来控制发送。基序号（base）是**最早未确认分组的序号**，下一个序号（nextseqnum)是**最小的未使用序号**。**N是流水线是最大能允许未确认分组的数量**（我们 在程序里使用的变量名是是**SEND_BUFFER_SIZE**）。通过维护这个窗口，发送端可以控制发送速率，也可以方便地实现重发分组的功能。

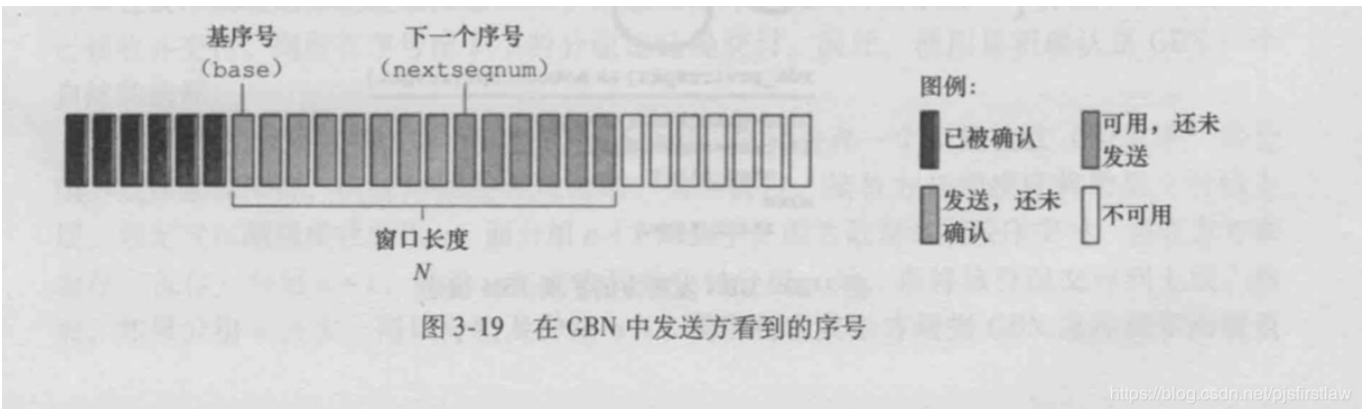


图 3-19　在 GBN 中发送方看到的序号

程序进入一个循环， 发送端在**两个状态**下进行转换， **第一个状态是接受收到接收方发送的ACK包**， 当此时没有收到 ACK包则进入状态2， **状态2是向接收方发送数据**。这样就可以保证不用停等协议， 能够一直在发送状态， 即可以发送多个包。

对于状态1（**接收ACK包**）： 当接收到ACK包, 首先把收到的数据包解包， **得到确认的包号值** ，判断**如果确认了最后一个包，则说明已经传送完了， 跳出循环，** 结束传输,

```
message, client_address = server_socket.recvfrom(BUF_SIZE)

unpacked_message = pkt_struct.unpack(message)

if (newBase == lastSendPacketNum + 1):
    mytimer.cancel()
    break
```

得到最后一个包时通过发送时,用一个变量**lastSendPacketNum**存文件发送的最后一个包号，然后判断收到的ACK包是否是确认最后一个包即可。

```
if str(data) != "b''":  # b''表示文件读完
    end_flag = 0
    #rnwd发送方没用到
    client_socket.sendto(pkt_struct.pack(*(nextseqnum, ack, end_flag, 1, data)),
server_address)
else:
    end_flag = 1  # 发送的结束标志为1，表示文件已发送完毕
    lastSendPacketNum = nextseqnum
    # rnwd发送方没用到
    print ("============================" +  str(lastSendPacketNum) +
"============================ ")
    client_socket.sendto(pkt_struct.pack(*(nextseqnum, ack, end_flag, 1 ,
'end'.encode('utf-8'))), server_address)
    break
```

当收到确认包后，对**确认的包进行删除**， **把序号在base到确认的包序号之间的包**全部从缓冲区删除， 更新base值， 同时会更新cwnd值， 代码在下面的阻塞控制实现可以看到。 如果缓冲区为0， 停止定时器， 否则启动计数器

```
#如果缓冲区为0  停止定时器，  否则启动定时器
if (base == nextseqnum):
    mytimer.cancel()
else:
    mytimer.cancel()
    mytimer = threading.Timer(MAX_TIME_OUT, timeout, [base, nextseqnum,
sendBuffer,client_socket, lastSendPacketNum, server_address])
    mytimer.start()
```

对于状态2：(发送数据包） 当没有收到包， 就会进入发送数据包的状态。。状态实际实现中，将socket的接收方式设置为非阻塞，一旦没有数据，则会抛出一个错误，我们在捕获的错误中实现

注意此时发送方发包不需要先把原来的包确认，才能发送包， 这就实现了允许发送方发送多个分组，而不需要等待确认 当满足以上条件， 表明发送方可以发送分组， 所以发送一个包， 发送前，把发送包加入到缓存区 这里面保存的是发送但未被确认的包。 当**base == nextseqnum** 说明此时缓冲区为空， 这时应该启动定时器， 来查看是否丢包

```
sendBuffer[nextseqnum] = pkt_struct.pack(*(nextseqnum, ack, end_flag, 1, data))
#当base和nextseqnum相等时， 开始计时
if (base == nextseqnum) :
    mytimer = threading.Timer(MAX_TIME_OUT, timeout,
                              [base, nextseqnum, sendBuffer, client_socket,
lastSendPacketNum, server_address])
    mytimer.start()
```

如果发生超时了，则要跳到处理超时的函数。重发**base**到**nextseqnum - 1**范围内的包，即已经发送了但却未确认的包。

```
#重新发送缓冲区里的所有包
print("重新 send packet:" + str(base) + "~" + str(nextseqnum - 1))
for i in range (base, nextseqnum + 1):
    try :
        client_socket.sendto(sendBuffer[i], server_address)
        print ("重新发送packet: " + str(i))
    except:
        #如果缓冲区的包已经发完， 将停止发送
        mytimer.cancel()
```

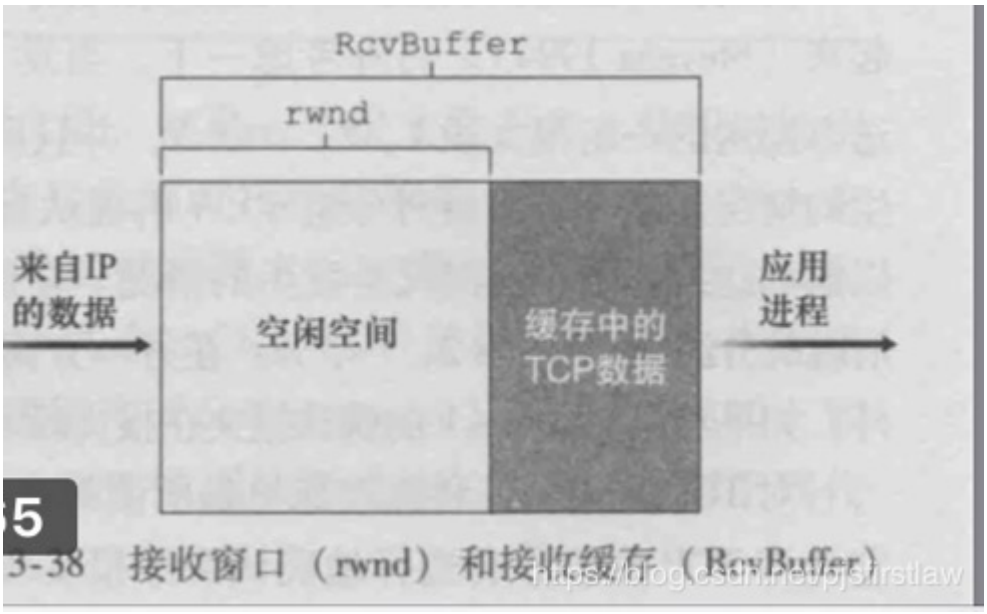
重发的包也有可能丢，因此也要重新设置定时器。

```
#重新启动定时器
mytimer = threading.Timer(MAX_TIME_OUT, timeout, [base, nextseqnum, sendBuffer,
client_socket, lastSendPacketNum, server_address])
mytimer.start()
```

接收方的动作则比较简单，接收方维护一个**expected**值就行了。如果收到的序列号为expected，则更新expected，返回当前序列号的ACK包，否则不更新**expected**，返回值为expected的ACK包。

2. 实现流控制

流控制，是让发送方的发送速率不要太快，让接收方来得及接受。 接收端维护一个接收缓存，每次处理数据之后，接收端把当前缓存的空闲空间的大小（rwnd)返回给发送端。发送端跟踪两个变量，**LastByteSent**和**LastByteAcked**，这两个值的差就是主机A发送到连接中但未被确认的数据量。发送端要将未确认的数据量控制在**rwnd**以内，以确保发送端不会使主机B的接收缓存溢出。



实现方法

我们在接收端使用一个**队列**来模拟接收缓存。因为我们用的python的socket接口，实际上socket.recvfrom(size)这个方法是从UDP接收缓存中获取size个字节的数据，缓存已经是有一个实现了的，**通过sock.getsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF)我们可以知道它的大小为65536**。但是这个应用中我们**忽视这个底层真正的缓存，我们把真正的缓存里的内容当作是仍在网络中传输的数据包**。

在流控制中，接收方还要多做一些工作。**当接收方接收到数据包，先保存在缓存中**，当没有收到数据包了，**就从缓存中取出数据处理数据包**。处理完数据包回复ACK包时要把当前的rwnd值发送给发送方。rwnd值保存在数据包的头部信息中。

```
rwnd = RCV_BUFFER_SIZE - buff.qsize()
```

发送方通过收到ACK包中的rwnd，来限制其发送速度。在发送方发送数据包的状态中，在发送之前加一个判断，**倘若已发送但还未确认的数据量大于rwnd值，则说明此次发送有可能让接收方的接收缓存溢出，因此不允许发送。**

```
  if nextseqnum - base > rwnd:
    continue
```

因此发送方**除了维护窗口之外还要维护rwnd值**。

3. 实现阻塞控制

参照TCP的拥塞控制算法，应用实现了**慢启动**和**拥塞避免**。 通过维护一个变量拥塞窗口（**cwnd**），对发送方的发送流量的速率进行了限制。加上前面的流量控制，发送端必须满足

```
LastByteSent - LastByteAcked <= min {cwnd, rwnd }
```

何时拥塞：当出现丢包的时候就假设出现网络拥塞的情况。

慢启动

设置初始cwnd为1，表示一个RTT内传送一个数据包。开始时应用向接收端发送一个数据包并等待确认，当收到确认包后，将cwnd值翻倍，指数增长。当检测到拥塞时，将慢启动阈值（**ssthresh**)设为**cwnd值的一半**，当到达或超过ssthresh值时，**结束慢启动进行拥塞避免模式。**

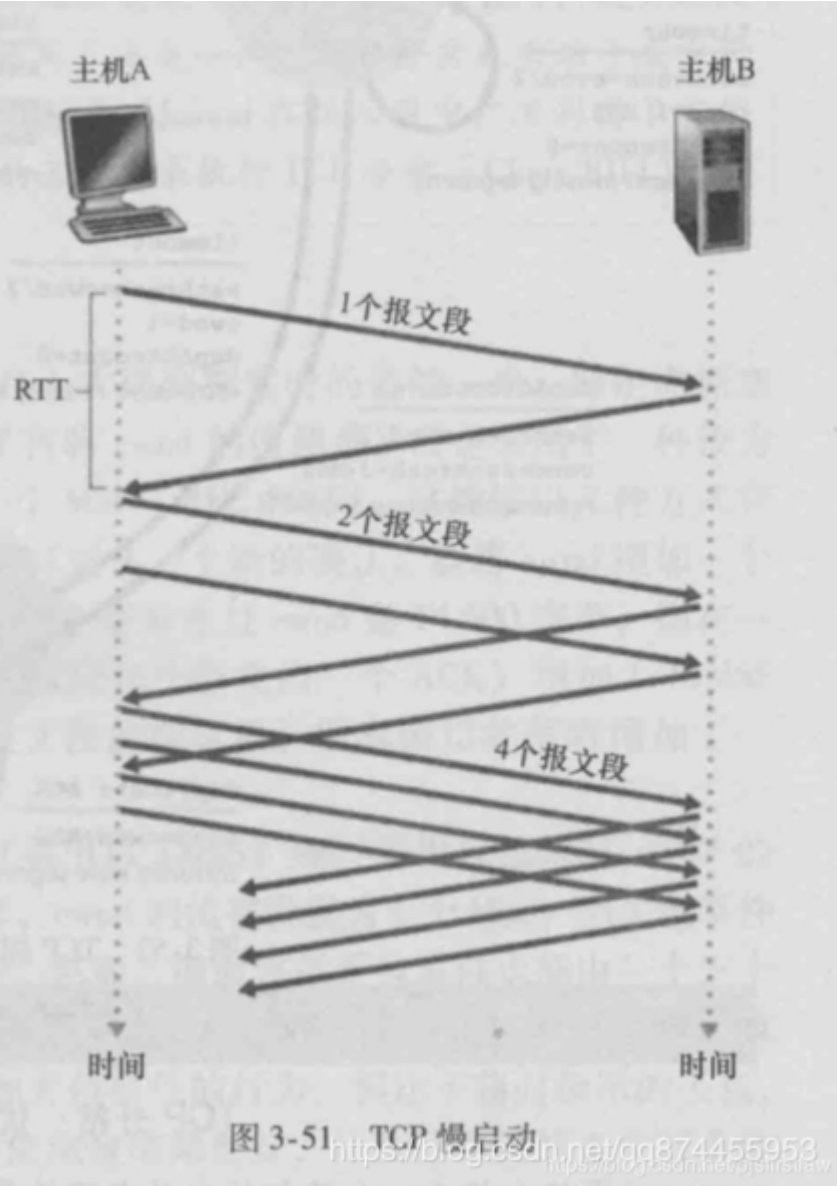


图 3-51 TCP 慢启动

拥塞避免

此时距离拥塞可能并不遥远，因此**不能每过一个RTT将cwnd的值翻番**，而采用每次只增加1、当再次出现拥塞时，ssthresh的值被更新为原来的cwnd值的一半，然后将cwnd设为1，进入慢启动阶段。

实现方法

书上慢启动的图，实际上是一种停等协议，它是一开始先发一个数据包，然后等待确认，再一次性发2个，再等待2个确认全部确认完才开始发4个。**因为我们是流水线地发送**，因此作一些变换：**在慢启动阶段，只要收到一个ACK包，就将cwnd加1，这实际上的效果是将速率翻倍了。**

同样像流控制一样，在发送数据包之前，检测在网络上的包数量是否大于**rwnd**值，如果是的话，就不允许发送。

```
 elif nextseqnum - base > cwnd:
    #print("受拥塞控制限制，发送速率拒绝发送")
    continue
```

当每次收到**ACK**包时，就维护cwnd的值。若cwnd>ssthresh，则说明在拥塞避免状态，这里不能直接将cwnd加1，而是维护一个变量**add_num**，一轮结束了才将cwnd加1。若cwnd<=ssthresh，则说明在慢启动状态，将cwnd加1。

```
if cwnd > ssthresh:
    if add >= cwnd:
        cwnd += 1
        add_num = 0
    else:
        add_num += 1
else:
    cwnd += 1
```

当进入超时函数，说明出现了拥塞状态。此时要改变**ssthresh**为当前**cwnd**值的一半，将cwnd变为1，回到慢启动状态。 实际实现中，因为超时是用另外一个线程来处理的，在主线程中也会修改cwnd值，因此要避免两者冲突，要为变量加互斥锁。

```
 #当进入超时操作时 阻塞控制 使得cwnd为1， ssthresh 为当时的cwnd的一半， 利用互斥锁使得更改ssthresh不会冲突
    if mutex.acquire(1):
        ssthresh = cwnd / 2
        #防止ssthresh变为负数
        if (ssthresh < 1) :
            ssthresh = 1
        cwnd = 1
        print("更新cwnd值为" + str(cwnd), "  更新ssthresh值为" + str(ssthresh))
        mutex.release()
```

这样，发送方要想发送数据包，得满足3个条件：

1. 未确认的分组数不能超过N
2. 未确认的分组数不能超过rwnd
3. 未确认的分组数不能超过cwnd

4. 服务端要支持多个客户端同时在线

使用python的threading包。当检测到有用户连接到服务器时，为用户创建一个进程。

```
# 创建新的线程，处理客户端的请求
new_thread = threading.Thread(target=serve_client, args=(client_address, message))
new_thread.start()
```

这样，对不同的客户端，会使用不同的线程去作相应的处理。

5. 程序要提供必要的出错反馈信息

当get文件不存在时将返回给接收方错误信息

```
if cmd == 'lget':
    # 文件不存在，并告知客户端
    if os.path.exists(SERVER_FOLDER + large_file_name) is False:
        server_socket.sendto('fileNotExists'.encode('utf-8'), client_address)
        # 关闭socket
        server_socket.close()
        return
```

6.相关的细节

1. 数据包的格式

根据以上设计，数据包得保存一个Seq值，一个ACK值，一个结束标记值，一个rwnd值，还有1024个字节的数据。

```
pkt_struct = struct.Struct('IIII1024s')
```

实际的方式采用python的**struct**包，可方便地打包和解包。

```
server_socket.sendto(pkt_struct.pack(*(nextseqnum, ack, end_flag, 1, data)),
client_address)
unpacked_data = pkt_struct.unpack(packed_data)
```

2. 建立连接的方式

模仿TCP连接，在数据传送之前要进行三次握手。对于发送方应该要发送两次数据包， 第一次发送命令字符串给接收方，这是第一次握手，然后接收方发送允许，这是第二次握手， 第三次发送方发送**ACK** 等待接收方的确认，这是第三次握手。

3. 超时时隔的确定

同样参考TCP连接。在建立连接的时候先估计RTT的值EstimatedRTT，显然超时时隔应该大于等于**EstimatedRTT**，否则将造成不必要的重传。 根据以下公式确定超时时隔。

```
TimeoutInterval = EstimatedRTT + 4 * DevRTT
```

DevRTT是偏离值。

```
DevRTT = |SamepleRTT - EstimatedRTT|
```