# 计算机网络期中项目设计&测试文档

## 项目说明：

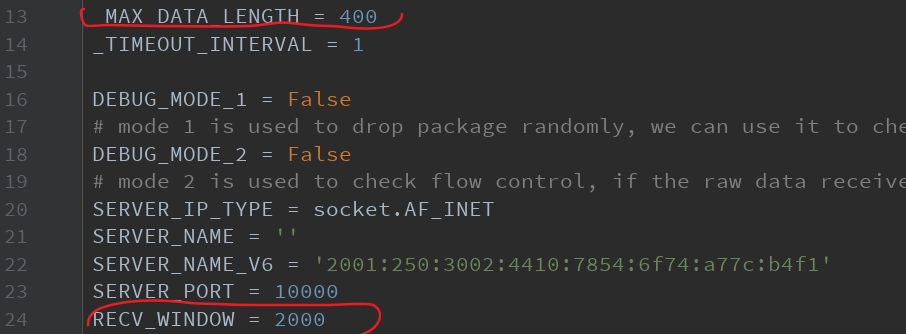
1. 本项目使用Python进行实现
2. 采用Client-Server架构，源代码有两个文件client.py和server.py，两个文件需要放在同一目录下才能运行客户端/服务端功能，支持大文件传输（下载/上传）

服务端监听端口限定为10000，传输端口在10001-10009之间，监听端口用于与用户建立连接，传输端口用于与用户进行文件数据的传输，最大支持9个线程同时工作（此处可以修改，以更改端口号或者最大连接数）

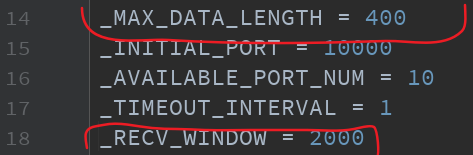
服务端使用方法：命令行输入 **python server.py**即可；客户端使用方法：命令行输入**python client.py lget/lsend ipv6\_hostname filename**

1. 使用UDP作为传输层，使用IPV6作为网络层，所以传输速度比较快
2. 实现了100%可靠传输
3. 实现类似TCP的流式控制功能
4. 实现类似TCP的拥塞控制功能
5. 服务端支持多用户同时下载/上传
6. **关于参数改进：如果使用的时候发现大量丢包，需要更改下面的常量值，以此优化传输：**

**Client.py中：**



**Server.py中：**



**网络越拥堵，MAX\_DATA\_LENGTH、RECV\_WINDOW两个常量值越应该设置的小一点：**

如果网络状况非常拥堵，这两个参数推荐（client.py和server.py都需要更改这两个参数）：

将MAX\_DATA\_LENGTH设置为300（每个UDP数据包大小）

将RECV\_WINDOW设置为400（接收方拥塞窗口大小）

## 结构设计：

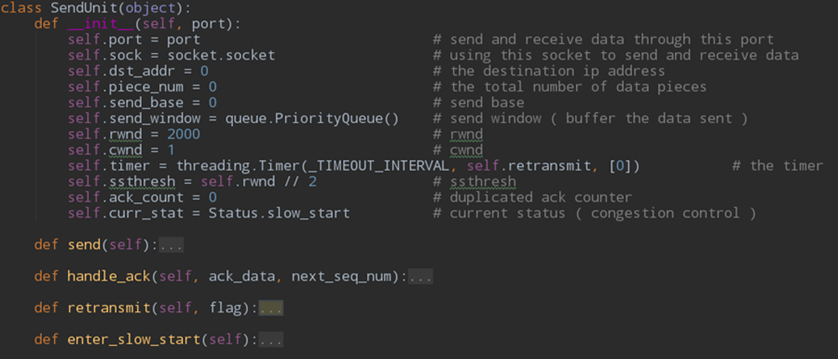
客户端具有从服务器下载和上传的功能，因此我们分别设计Recv\_Unit和Send\_Unit两个类分别完成下载和上传功能

客户端下载文件时实例化Recv\_Unit并调用该类的recv\_file方法接收服务端发来的文件，同时服务端实例化Send\_Unit并调用send\_file向用户发送文件

客户端上传文件时则相反，由客户端实例化Send\_Unit并调用该类的send\_file方法向服务器发送文件，同时服务端实例化Recv\_Unit并调用recv\_file方法接收用户发来的文件

因此主要设计包含：class Send\_Unit 上传类、class Recv\_Unit 下载类、class Fragmentation分片类，下面分别介绍：

**class Send\_Unit 上传类：**

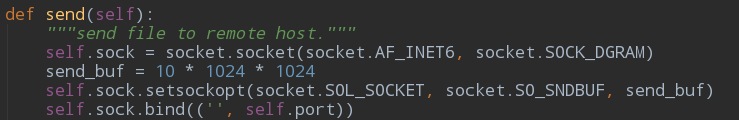


该类用于向远程主机发送数据，其实例变量与各自的说明如上图注释中所示，有端口、套接字、目的IP地址、总数据段数量、用于模拟TCP的send base、send window、rwnd、cwnd、ssthresh、计时器、冗余ACK计数和表示拥塞控制中三个状态的变量等。

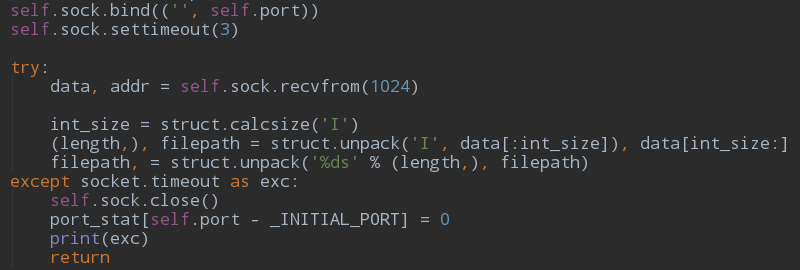
下面简要说明该类的4种方法。

* send方法

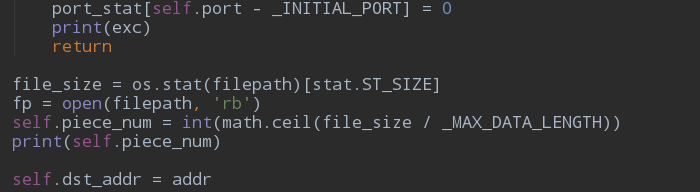
调用该方法向远程主机发送文件。

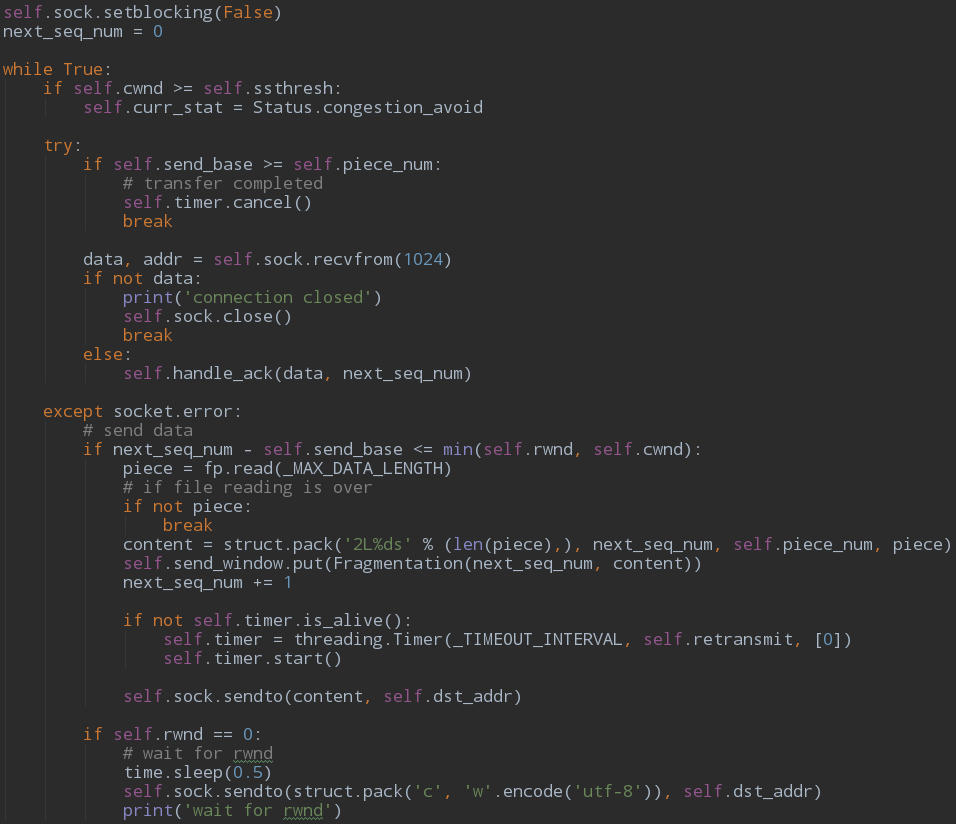


首先创建套接字，并绑定端口（这里还另外调整了UDP的发送缓存）



这里获取远程主机想要下载的文件（文件路径），并设计超时，若3s内对方没有发送这个信息到发送方，则假定连接未建立，此时关闭套接字，切断连接。

计算要传输的文件大小与总数据段数量，获取远程主机IP地址。

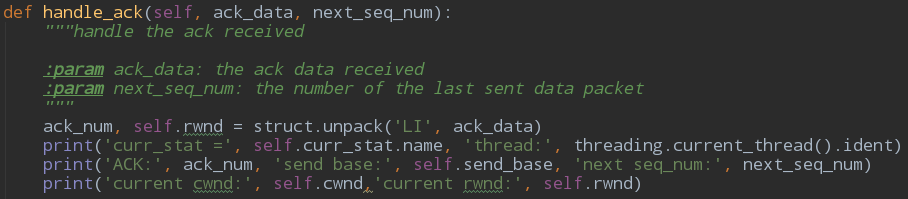
接下来开始正式传输文件：

这里单线程模仿流水线实现传输，为此，首先将套接字设为**非阻塞模式**。

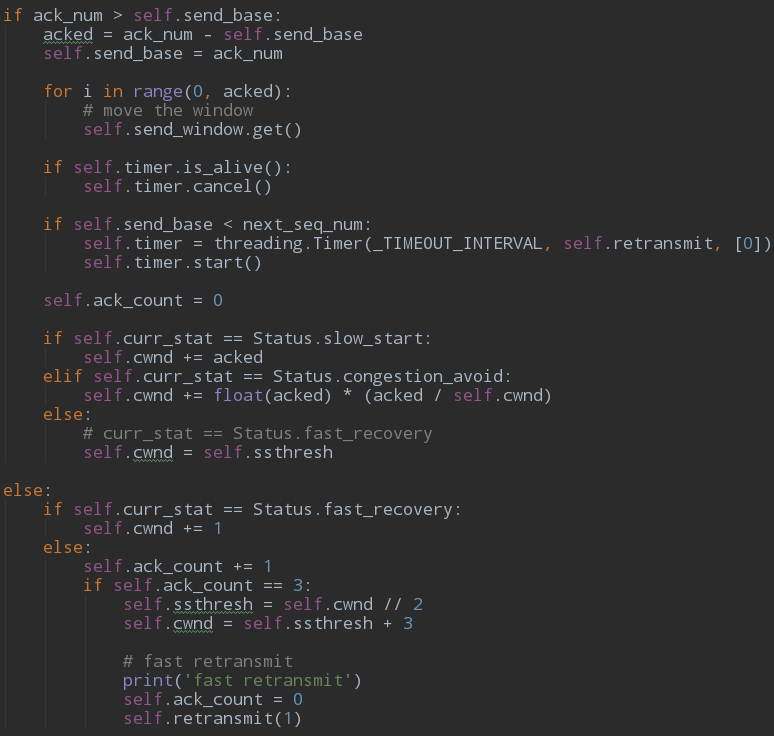
传输文件时发送方接收文件只会接收到接收方发送的ACK，所以我们首先从UDP缓存读取数据（ACK），若发现无法读取到任何数据，则转到异常处理部分；若能够读取到数据，说明发送方这边收到了ACK，则转入ACK处理函数处理ACK。

* + **异常处理部分：**该部分用于发送数据。由于ACK还没有传到，故继续传输数据给接收方，此时由于**流控制**和**阻塞控制**，我们应保证发送到**连接中但未被确认的数据量**（最后发送的数据段序号 next\_seq\_num – 最早未被确认的数据段序号 send\_base）**小于等于接收窗口rwnd和拥塞窗口cwnd的两者的最小值**。若条件满足，我们就读取读取要发送文件的一段发送给接收方，并将这段数据放入send\_window(重传时使用)。当计数器未启动时，启动计时器。另外，若rwnd为0，则根据教材所说方案，发送方这边继续发送只有一个字节数据的报文段到接收方以维持两方的通信。
  + **接收到ACK：**进入ACK处理函数。
* handle\_ack方法

调用该方法处理接收到的ACK。



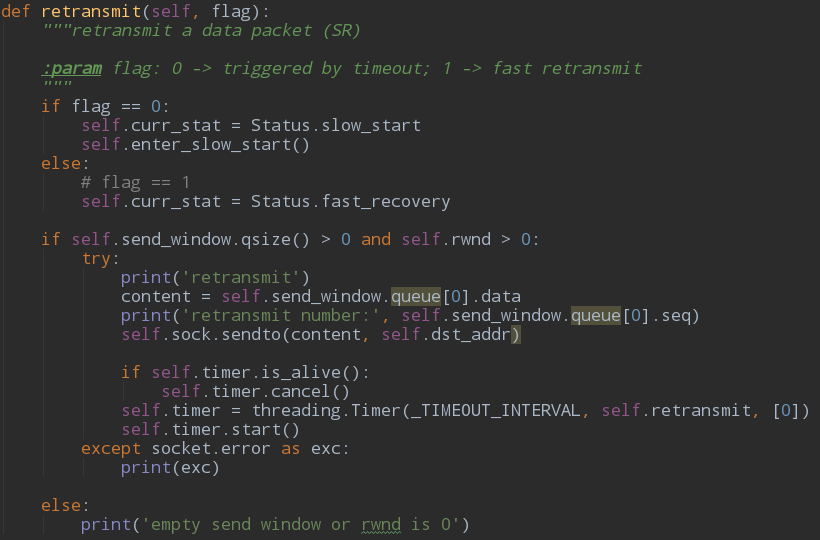
首先解析收到的ACK数据，更新rwnd，并打印一些相关信息。



正式处理ACK，通过判断ACK和send\_base的大小关系来判断是否是冗余ACK。

* + 不是冗余ACK：更新send\_base的值，移动send\_window，若当前还有未确认的数据段，则重启定时器。重置冗余ACK计数。根据当前所处状态（拥塞控制），相应更新cwnd。
  + 是冗余ACK：根据当前所处状态相应更新cwnd。若当前为慢启动或拥塞避免状态，则增加冗余ACK计数，当该计数等于3时触发快速重传，调用重传函数。
* retransmit方法

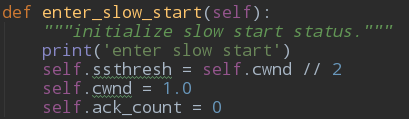
调用该方法重传数据段（选择重传）



首先通过传入的参数判断这次重传是超时触发的还是快速重传，据此更新当前状态。

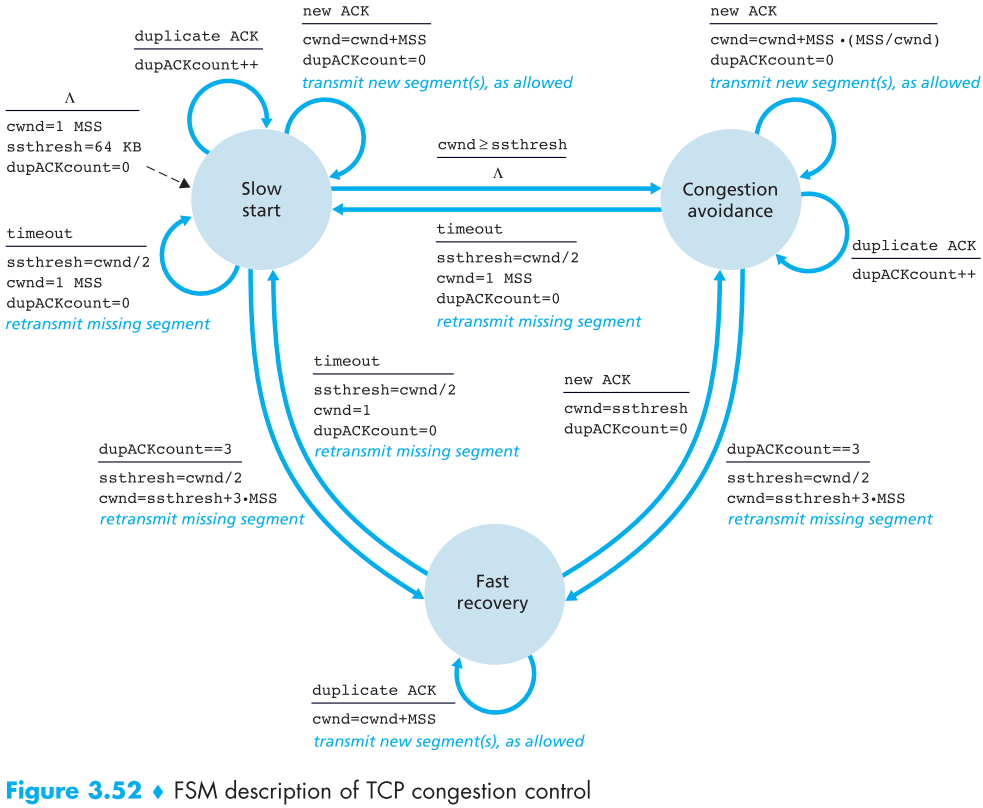
然后判断send\_window是否为空，rwnd是否为0，若是则可以进行重传。进行重传时取出send\_window中的第一个元素，该元素的data成员即为最早发送未确认的数据段，再次向接收方发送该数据段。

* enter\_slow\_start方法



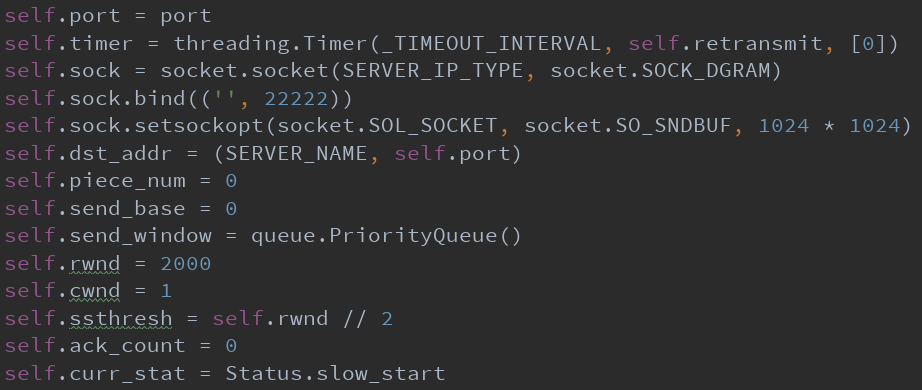
调用该方法初始化进入慢启动状态时的各变量。

阻塞控制主要是根据教材中对TCP拥塞控制的FSM描述设计，即下图：



**class Recv\_Unit 下载类：**

**成员变量：**



（1）self.ack\_send\_interval ：接收方并不会收到数据包就立即发送，而是使用RFC [5681]的建议：当接收到按序的报文，将最多等待500ms发送一次ack给发送方 ，self.ack\_send\_interval设置为不超过500ms；

如果收到乱序报文则马上回复一个ack给发送方，以示按序报文仍未收到，三次这样的冗余ack将引起快速重传  
（2）self.client\_socket： 该类使用的套接字，使用ipv6作为网络层，并使用setsockopt设置UDP套接字缓存大小为1M  
（3）self.port = port：使用传入的port进行初始化，表示后续接收数据通过该端口进行  
（4）self.top：当前还未收到的报文的最小序号，小于该序号的报文段都已收到并写入磁盘  
（5）self.last\_byte\_received ：当前已经收到的报文数目

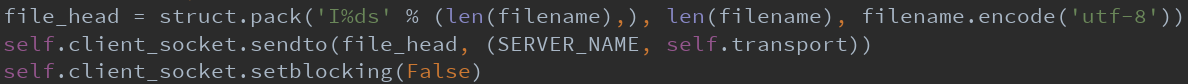
（6）self.clock\_using = False：用于指示计时器是否在使用，该计时器即按照（1）中变量计时，每间隔 ack\_send\_interval发送一次ack

（7）self.send\_ack\_operation ：ack\_send\_interval的计时器

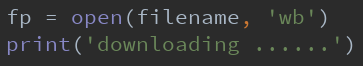
（8）self.package\_num：指示整个文件大小   
（9）self.have\_received ：是一个bool数组，指示某个数据包是否已经收到过，如果是则会丢弃该数据包，不进行分析/写入文件操作

**recv\_file(self, filename)方法：**

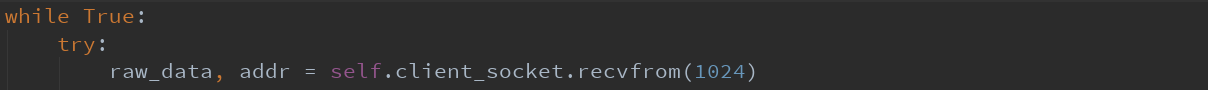
（1）该方法首先向发送方发送文件名字信息，并设置套接字为非阻塞：

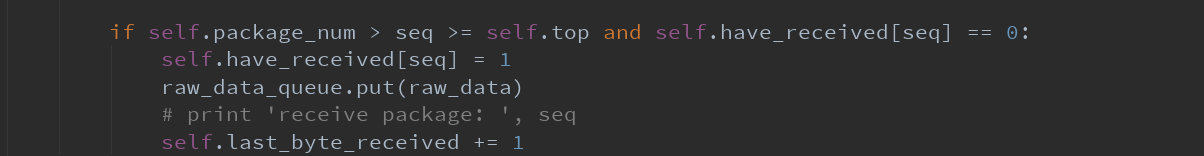


（2）并打开文件向文件写入收到的信息：



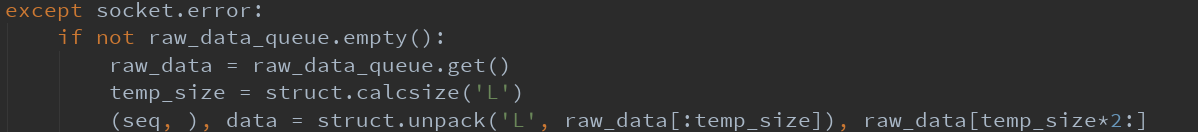
（3）如果能够从套接字缓冲区收到信息，并且该数据包之前未收到过（通过have\_received[seq]是否为1判断，是1则收到过），则将其放入内存，等待后续处理（这样能够及时从缓冲区获取数据防止套接字过长时间未使用而端口连接），同时更新变量：last\_byte\_received++，have\_received[seq]设为0：





（4）如果暂时不能从套接字缓冲区接收到数据（数据还在网络传输中），则对上一步存入内存的数据进行处理：

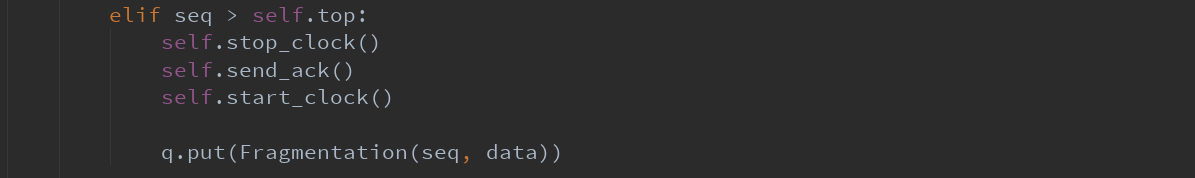
首先进行解包得到该**数据包的序号**和**原始文件中的数据**：



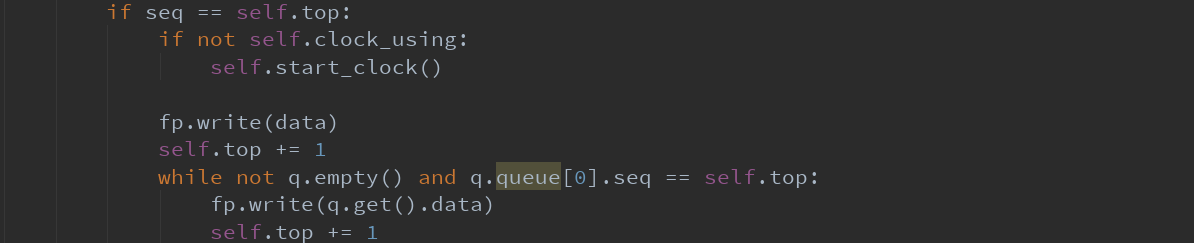
然后判断该数据是否是期望的数据（具有期望的按序的序号，即seq==self.top，表示该数据包的序号就是期望的数据序号）：



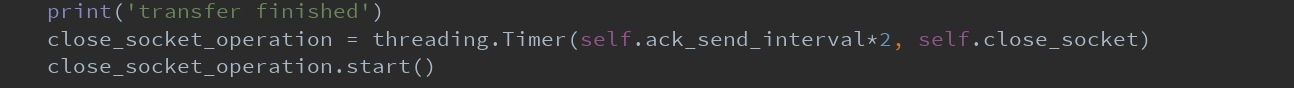
如果不是就将该序号和数据组成一个Fragmentation分片类的实例，放入优先队列q，该优先队列按照序号排序，**并且立刻发送ack（快速重传）：**



如果是就直接将解包的原始文件数据写入磁盘文件，并且如果队列q中有序分片的最小序号（即q.top）等于期望的序号，则将q.top为首连续分配全部写入磁盘文件，并且每写入一段数据都更新self.top = self.top+1，即下一个期望收到的序号，**由于此时收到的报文按序，所以调用计时器，间隔ack\_send\_interval才发送一次ack：**



循环操作（3）、（4）步骤，直到文件传输并处理完毕，发送最后一个ack并关闭连接：



close\_socke函数：

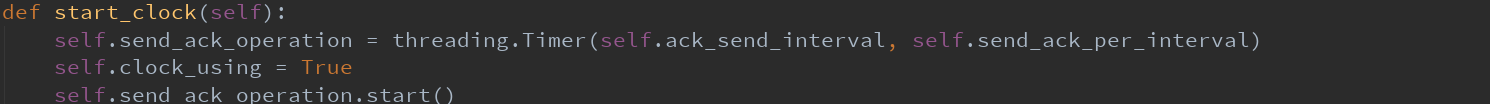


判断传输是否结束方法：



**start\_clock函数：**

设置并开始定时器，间隔一定时间发送一次ack：



**stop\_clock函数：**

停止计时器：



**send\_ack\_per\_interval函数:**

发送一个ack并且设置clock\_using为false表示时钟未在使用

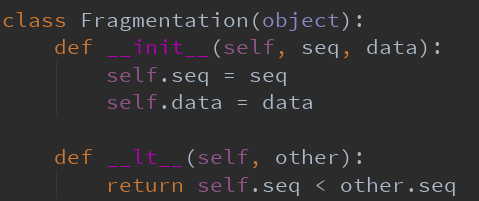
**send\_ack函数:**

计算并发送ack序号和接收窗口信息





**class Fragmentation分片类：**



用于将数据序号和数据包打包，通过序号判断大小顺序，从而形成按序的优先队列

**客户端主函数逻辑：**

（1）判断用户输入是否正确，必须输入正确的command（lget/lsend）、正确的ipv6地址、正确的文件名，否则输出正确使用方法，然后退出程序

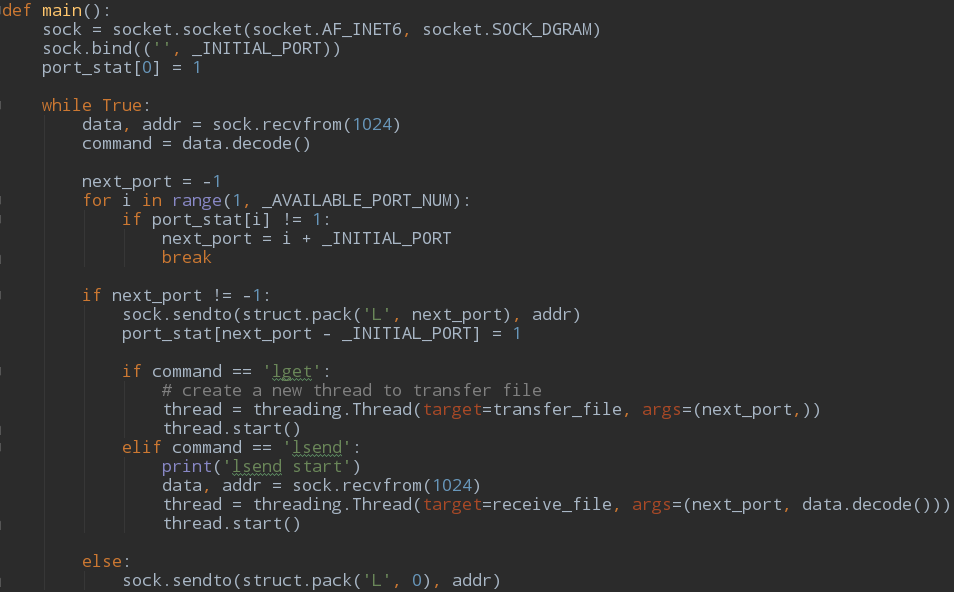
（2）与给定的服务端地址（由用户输入）+端口（10000）进行连接建立，（3）、（4）步骤均在此端口进行

（3）向服务端发送命令，通知其服务类型（lget对应下载服务，lsend对应上传服务）

（4）从服务端接受进行数据传输的端口号（如果收到-1，说明服务器进行传输的端口被占满，需要等待其它用户完成服务才能从服务的接受服务），然后使用该端口号实例化Recv\_Unit/ Send\_Unit类，调用send\_file/recv\_file从该端口进行数据收/发

如果（4）步骤3s之内未受到服务端反馈信息，连接断开，输出超时信息，需要重试

**服务器端主函数设计如下：**

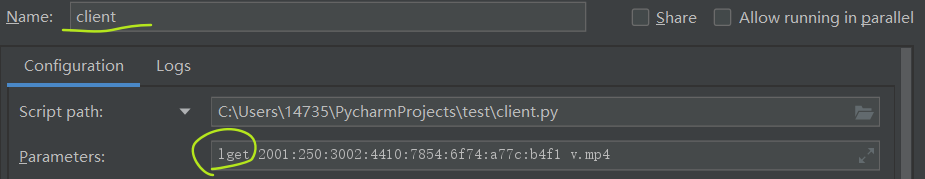


在主函数中，创建一个专门用于接收传输请求的套接字，当有请求从远程发来时，寻找可用的端口号，若有可用的端口号，则新建一个线程，利用该端口号创建套接字进行相应的传输任务；若无可用的端口号，则发送含有数字0的报文段，通知客户端不能进行传输。

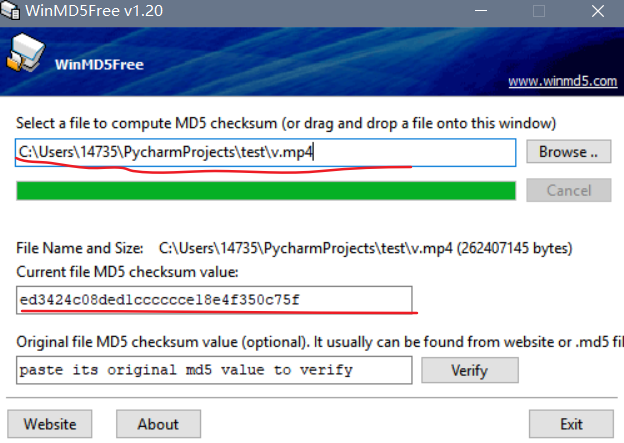
## 测试

（1）客户端下载测试（lget功能，可传输大文件，下面用于下载一个视频）：

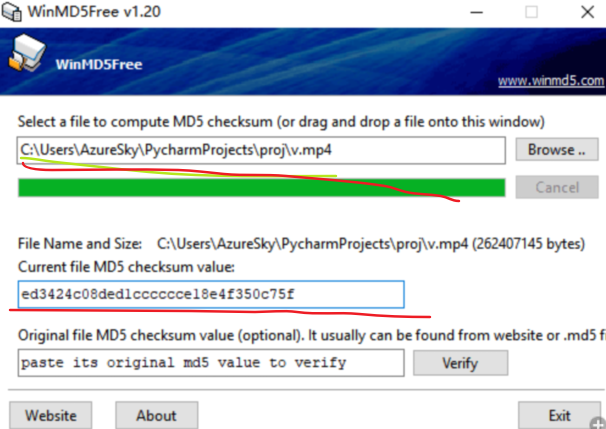
客户端使用lget进行文件下载，输入参数：**lget+服务器地址+文件名（路径）**



客户端收到文件的md5码：



服务器原文件的md5码：



对比md5码完全相同，说明成功实现100%可靠下载，当然作为视频也能在客户端完整播放：

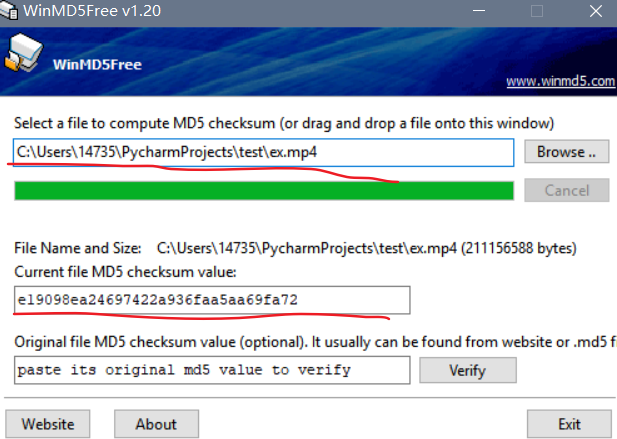


（2）客户端上传测试：（lsend功能，可传输大文件，下面用于上传一个视频）

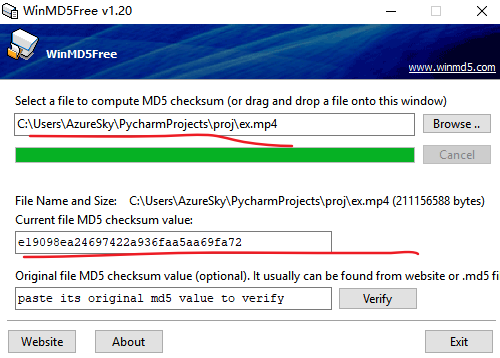
客户端使用lsend进行文件上传，输入参数：**lsend+服务器地址+文件名（路径）**

E:\QQ相关文件\1473595322\Image\C2C\~8`G(TG03HLSF889R%YBGD0.png

客户端原文件的md5码：



服务器收到文件的md5码：

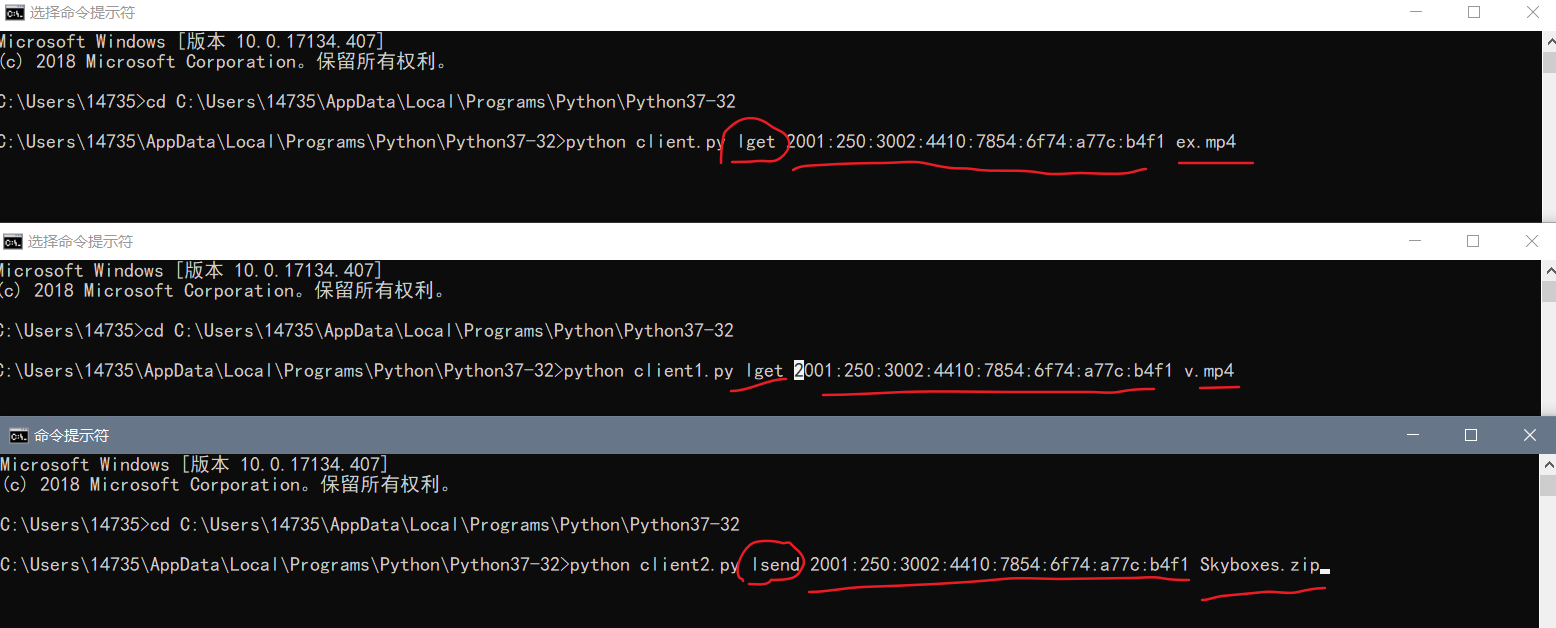


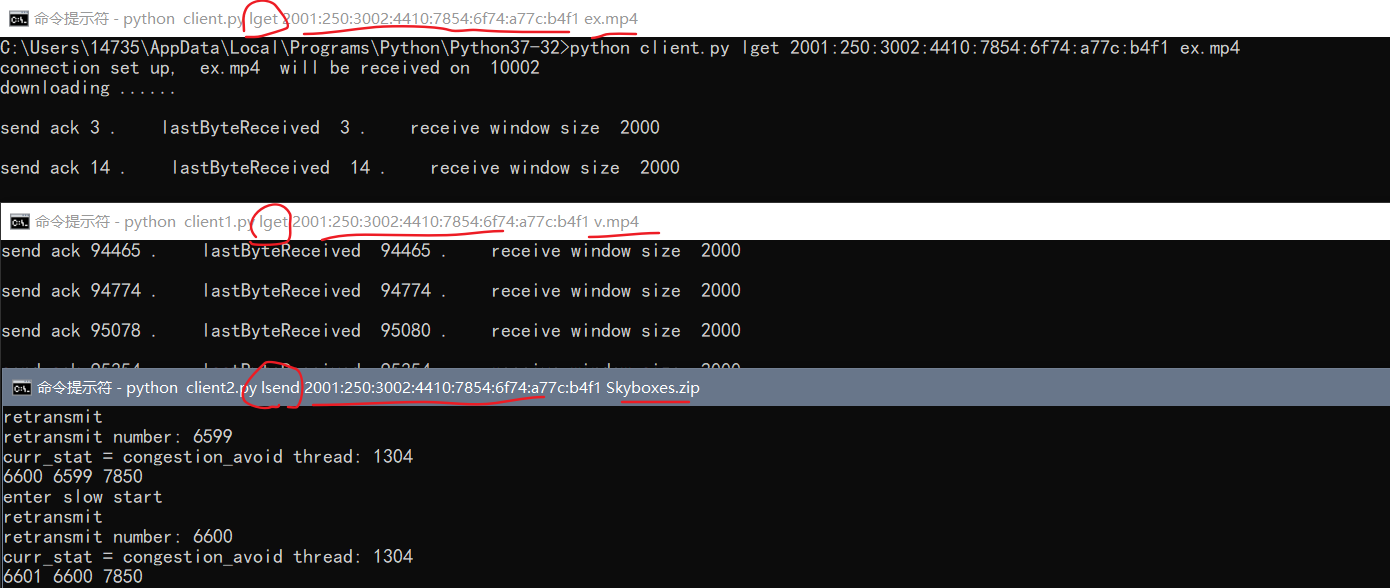
对比md5码完全相同，说明成功实现100%可靠上传，

当然作为视频也能在服务端完整播放：



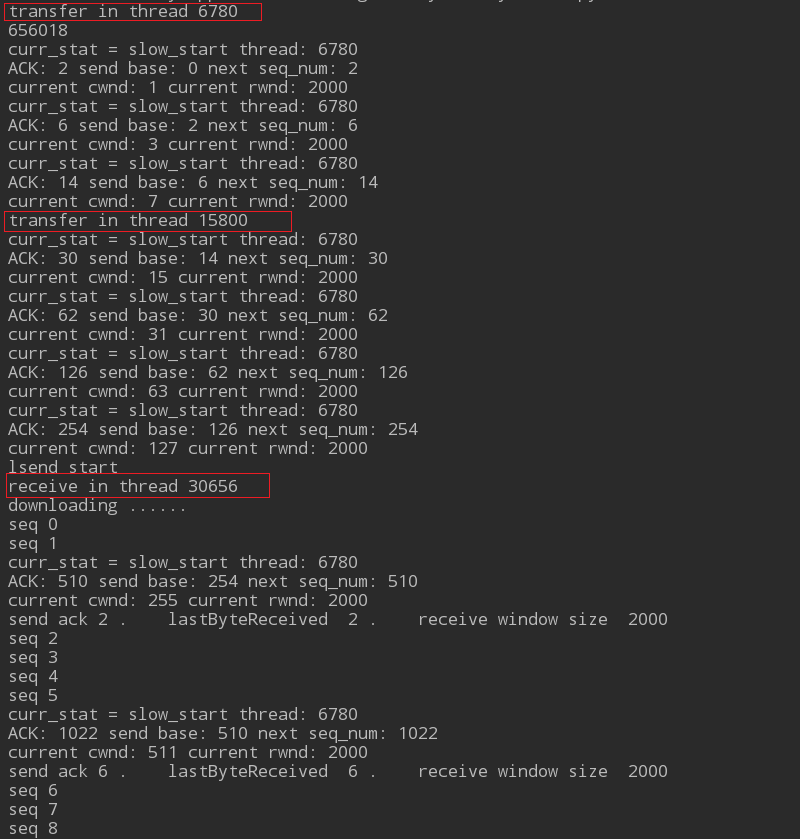
（3）服务端多线程工作：（下面三个客户端同时向服务端发起下载/上传请求，服务端则通过多个线程同时为它们提供服务）





从上面看到，两个客户端client、client1正在进行下载操作、一个客户端client2正在进行上传操作

下面是服务端的输出，从输出中我们也能看到现在有三个线程同时运行：



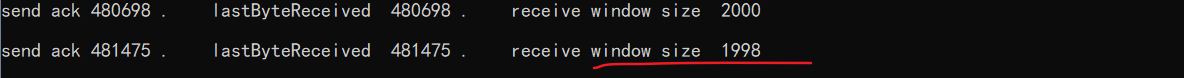
（4）类似TCP的流式控制功能实现

流式控制指的是一种机制，在该机制的作用下发送方的发送数据速度能够与接收方处理数据的速度匹配，防止发送方发送过快而接收方处理速度跟不上，接收方缓存剩余空间不断变小，最终溢出。通过流式控制，发送方在接收方处理好数据之前将不会继续大量发送数据，从而留给接收方足够的数据处理时间。

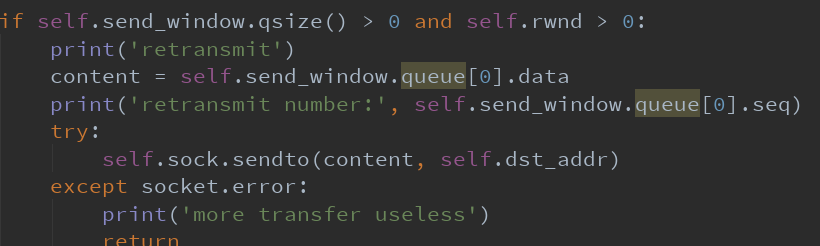
下面通过客户端的一次lget作为例子说明我们的流式控制实现：

随着数据包接收到缓冲区（实际上是内存模拟的缓冲区），缓冲区中的数据可能来不及处理，从而使得接收方的缓冲区剩下的空间变小，也就是接收窗口变小，下面图中（接收方输出）我们可以看到接收窗口从2000变为了1998，即缓存区中还有两个数据等待处理：

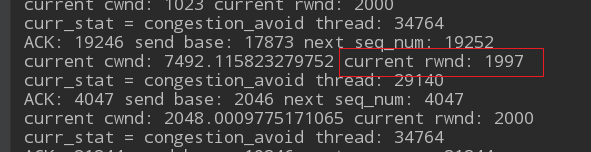




然后接收方将这个缓冲区剩余空间变小的信息通过“接收窗口“来告诉发送方，发送方收到这个信息之后将会限制**已发送的数据<接收窗口**：



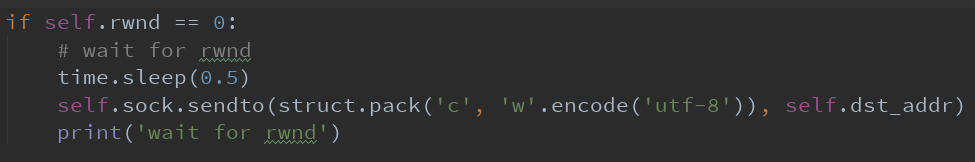
下面是发送方的输出中我们看到发送方已经发送的数据量但是未被接收方收到/确认19252-17873 = 1379 < 1997，即被限制在接收方的接收窗口之内：



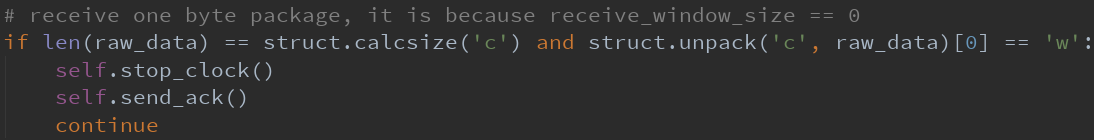
从而能够使得接收方有足够的时间处理，不至于缓存溢出。

但是当接收方的接收窗口变为0的时候，发送方将无法再向接收方发送数据，这时候需要让发送方继续发送大小1字节的报文给接收方，接收方收到之后会回复包含接收窗口信息的报文，等到接收方处理数据一段时间之后，接收窗口将不再为0，从而发送方能够正常发送数据：

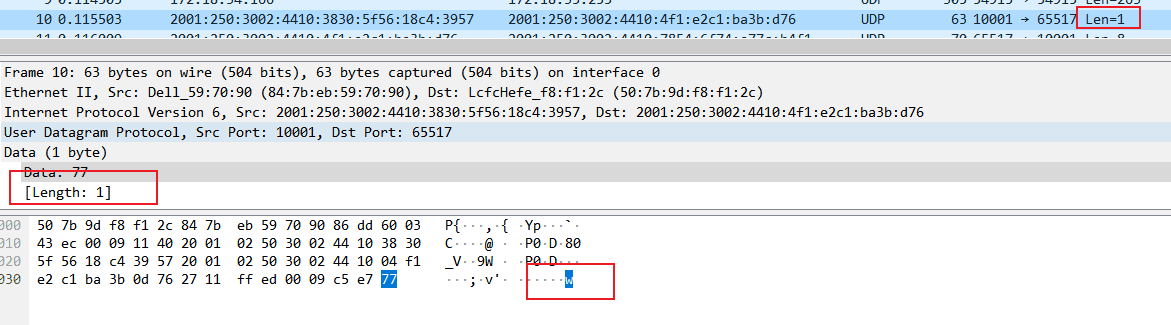
发送方在接收方接收窗口变为0的时候继续发送一个大小1字节的报文给接收方代码：



接收方收到发送方发送的大小1字节的报文时，接收方回复接收窗口代码：



下面时接收窗口为0的时候用wireshark抓到的大小1字节的数据包：



，并且与此同时接收方不停的回复包含receive window的报文，用来通知发送方自己当前的接收窗口：



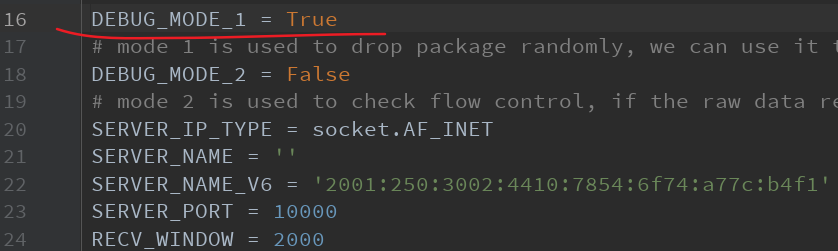
一段时间之后接收方的接收窗口为2，又能够进行正常数据传输了：



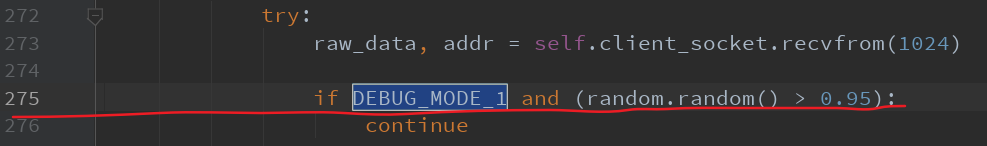
以上说明流式控制功能完整实现

（5）类似TCP的拥塞控制功能实现测试：

首先将client.py中的DEBUG\_MODE\_1常量设置为True，表示检测拥塞控制功能：

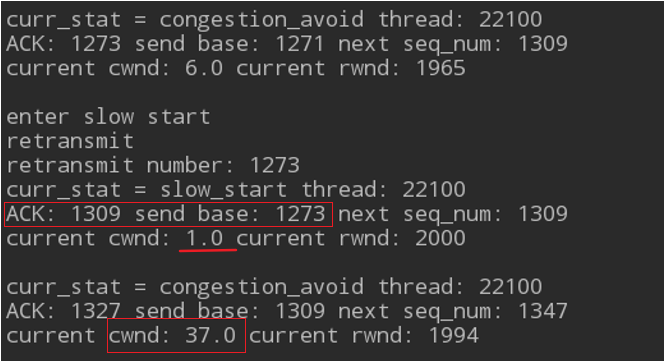


然后设置随即丢包率，从而通过代码主动丢包：



这里random.random()>0.95则表示丢包率5%

然后得到如下结果：



上图中，状态首先从拥塞避免状态进入慢启动阶段，这是因为接收方主动丢包导致了发送方超时，然后进入慢启动

除此之外，从上图可以看出，从慢启动阶段到拥塞避免阶段，接收方收到了1309 – 1273 = 36 个有效数据包，所以在下一次收到ACK打印信息时，cwnd从1增加到37，进入了拥塞避免状态（因为增加后cwnd大于了ssthresh，判断进入拥塞避免状态）。

由于状态转移过于复杂，上面仅仅列出有限的例子进行状态变化说明，更多信息见代码&实际运行状况

以上是本次实验报告全部内容