

计算机网络期中项目

——lftp实验报告

实验简介

编写一个网络应用程序LFTP，以支持两台计算机在网络中的大文件传输。

实验要求

1. 使用C, C++, Java或Python实现;
2. 使用客户端-服务器服务模型;
3. 必须包含一个客户端程序和一个服务端程序, 客户端不仅能上传大文件, 还要能下载文件,
 - 上传文件格式:
LFTP lsend myserver mylargefile
 - 下载文件格式:
LFTP lget myserver mylargefile
4. 使用udp传输层协议;
5. 实现类似TCP的100%可靠传输;
6. 实现类似TCP的流量控制;
7. 实现类似TCP的拥塞控制;
8. 服务端可以同时支持多个用户;
9. 提供有效的debug信息。

源代码

Github: <https://github.com/9ayhub/LFTP>

client-thread.py 和 server-thread.py为最终代码, 其余文件为进行项目时的主要功

能实现

队员信息

姓名	何坤宁	黄灿铭
学号	16340073	16340079
贡献	50%	50%

项目设计

DESIGN

文件结构

```
client-thread.py
----def resend()
----def receive()
----def transmit()
----main()
server-thread.py
----def write()
----def server()
----main()
```

语言

python2.7 64位

模型

客户端-服务器服务模型

传输层协议

udp

功能

1. 传送大文件

transmit large files

发送方

创建udp socket，打开要发送的文件，将大文件进行分片传输，最后关闭文件和socket。

```
1 // 简化版
2 client_socket = socket(AF_INET, SOCK_DGRAM) // SOCK_DGRAM意味着它是一个UDP套接字
3 f = open(filePath.decode('UTF-8'), 'rb')
4 data = f.read(data_size)
5 while True:
6     if data:
7         packet = pickle.dumps(c_pkt(nextseq, data))
8         client_socket.sendto(packet, server_addr)
9     else:
10        client_socket.sendto("exit " + str(nextseq), server_addr)
11        break
12 f.close()
13 client_socket.close()
```

接收方

创建udp socket，打开要写入的文件，等待来自发送方的消息，收到消息后，如果对方想结束传送，且文件接收完毕，则停止接收，否则将收到的文件片段写入接收文件。

```
1 // 简化版
2 server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
3 server_socket.bind(server_addr)
```

```

4 f = open(filePath.decode('utf-8'), 'wb')
5 while True:
6     request, client_addr = server_socket.recvfrom(packet_size)
7     ...
8     if request == "exit " + str((ack + 1) % seq_limit):
9         ...
10        break
11    else:
12        client_pkt = pickle.loads(request)
13        ...
14    //def write(p, LastByteRead, windows, done, f, server_socket):
15        f.write(windows.get())
16        server_socket.sendto(pickle.dumps(s_pkt(client_pkt.base, rwnd)), client_addr)
17    f.close()
18    server_socket.close()

```

2.100%可靠传输

100% reliability as TCP

确认

接收方

接收方不缓存失序的分组，当接收方收到一个失序的分组，便把它丢弃。当接收方收到一个正确顺序的分组时，便回应该分组的序列号给发送方，表明该分组及其之前的分组已经收到。

```

1 if (client_pkt.base == (ack + 1) % seq_limit
2     ack = client_pkt.base
3     server_socket.sendto(pickle.dumps(s_pkt(client_pkt.base, rwnd)), client_addr)
4 //server.py: def server()
5 //ack表示接收方当前已收到的最后一个分组
6 //当接收方收到一个分组的序列号是ack的下一个序列号时，就表明顺序正确
7 //接收方收取这个包，并把序列号发送回去

```

发送方

在不可靠的网络中，接收方回应的ack是可能失序甚至丢失的，所以发送方应该判断接受哪些ack

因为每一个ack都表示接收方已正确收取（序列号 \leq ack）的数据包，所以凡是（ack \geq base）的都可以接受，base为发送方已发送但是未确认的第一个包。

```
1 if (int(server_pkt.ack) >= base and int(server_pkt.ack) - base < 50) or  
   (base - int(server_pkt.ack) > 800) :  
2     base = (int(server_pkt.ack) + 1) % seq_limit  
3 //client.py: def receive()  
4 //因为base和ack都是循环增长的，即999的下一个序列号为0，所以边界条件的判断很重要  
5 //(ack-base<50) 是为了防止ack==999&&base==0等边界情况被接受  
6 //(base-ack>800)是为了防止ack==0&&base==999等边界情况被遗漏
```

序号

用于为从发送方流向接收方的数据分组按顺序编号。所接收分组的序号间的空隙可使接收方检测出丢失的分组。具有相同序号的分组可使接收方检测出一个分组的冗余副本。

```
1 packet = pickle.dumps(c_pkt(nextseq, data))  
2 client_socket.sendto(packet, server_addr)  
3 nextseq = (nextseq + 1) % seq_limit  
4 //client.py: def transmit()  
5 //nextseq表示最后一个已发送的分组的下一个分组  
6 //也就是当前准备发送的分组
```

超时重传

使用定时器来恢复丢失的数据或确认分组。如果出现超时，发送方重传所有已发送但未被确认过的分组。（回退N步）

首先维护一个重传队列，在每一个数据包发送之后，将其放入队列中，等待超时重传

```
1 packet = pickle.dumps(c_pkt(nextseq, data))
2 client_socket.sendto(packet, server_addr)
3 nextseq = (nextseq + 1) % seq_limit
4 windows.put(packet)
5 //windows为重传队列
```

接着设置定时器，定时器启动后，直到接收方收到一个ack，重新启动定时器，并将已确认的分组从重传队列中出列

```
1 timer = threading.Timer(time_limit, resend)
2 timer.start()
3 response, _ = client_socket.recvfrom(packet_size)
4 timer.cancel()
5 //client.py def receive()
6 //定时器的启动和重启
```

如果接收方一直没有收到ack，那么定时器便循环启动，执行重传函数

重传函数将重传队列中的分组全部重传（回退N步）

```
1 //client.py
2 def resend():
3     win_size = windows.qsize()
4     for _ in range(0, win_size):
5         packet = windows.get()
6         client_socket.sendto(packet, server_addr)
7         windows.put(packet)
8         time_count = time_count + 1
9         timer.cancel()
10        timer = threading.Timer(time_limit + time_count, resend)
11        timer.start()
```

另外，超时重传表明网络拥塞，所以超时间隔可随重传次数加长，避免冗余重传

差错检查

使用检验和来检测在一个分组中的比特错误，受损的报文段将被丢弃。

UDP协议自带差错检查，因此不需要实现

流水线

不使用停等方式运行，允许发送方发送多个分组而无需等待确认。

流水线需要维护两个变量：

base（已发送但未确认的第一个分组）

nextseq（已发送的最后一个分组的下一个分组）

它们之间的差不能超过给定的窗口大小

当超过时，就说明未确认的分组数量过多，因此需要停止发送分组来等待确认

```
1 while True:
2     if (nextseq - base + seq_limit) % seq_limit < min(rwnd, cwnd):
3         data = f.read(data_size)
4         if data:
5             packet = pickle.dumps(c_pkt(nextseq, data))
6             client_socket.sendto(packet, server_addr)
7             windows.put(packet)
8             nextseq = (nextseq + 1) % seq_limit
9 //client.py: def transmit()
10 //min(rwnd, cwnd)为给定的窗口大小
11 //rwnd为流量控制维护的窗口大小，cwnd为拥塞避免维护的窗口大小
12 //二者取较小值，有效地控制发送频率，为下面即将阐述的机制
```

3.流量控制

flow control function similar as TCP

提供流量控制服务以消除发送方使接收方缓存溢出的可能性。

接收方

接收方具有接收缓存，并用RcvBuffer来表示其大小，接收方不时地从该缓存中读取数据。我们定义以下变量：

- LastByteRead：接受方的应用进程从缓存读出的数据流的最后一个字节的编号。
- LastByteRcvd：从网络中到达的并且已放入主机B接收缓存中的数据流的最后一个字节的编号。

为避免缓存溢出，下式必须成立：

$$\text{LastByteRcvd} - \text{LastByteRead} \leq \text{RcvBuffer}$$

接收窗口用rwnd表示，根据缓存可用空间的数量来设置：

$$\text{rwnd} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

当rwnd等于0时，接收方不再将报文放入缓冲区，只发送报文告诉发送方现在的rwnd大小。如果rwnd不等于零，且收到有序报文，接收方将其放入缓冲区，重新计算rwnd，并发送给发送方。

```
1 rwnd = RcvBuffer - (LastByteRecv - LastByteRead[0] + seq_limit) % seq_limit
2 // rwnd 等于 0， 或者收到报文为"No Buffer"时
3 if rwnd == 0 or request == "No Buffer":
4     server_socket.sendto(pickle.dumps(s_pkt(ack, rwnd)), client_addr)
5     continue
6 ...
7 else:
8     client_pkt = pickle.loads(request)
```



```

9 // 报文有序且窗口仍有空间
10 if (client_pkt.base == (ack + 1) % seq_limit and
11     (LastByteRecv - LastByteRead[0] + seq_limit) % seq_limit <= RcvBuffer):
12     server_socket.sendto(pickle.dumps(s_pkt(client_pkt.base, rwnd)), client
13                           _addr)
14 else:
15     server_socket.sendto(pickle.dumps(s_pkt(ack, rwnd)), client_addr)
16 // 序号范围为[0, seq_limit]

```

发送方

发送方轮流跟踪两个变量，LastByteSent和LastByteAked。发送方在发送文件的过程中须保证：

$$\text{LastByteSent} - \text{LastByteAked} \leq \text{rwnd}$$

当接收窗口为0时，发送方继续发送只有一个字节数据的报文段，以确认接收缓存是否已经有新的空间。

```

1 rwnd = 100 // rwnd初始值为100
2 //def transmit():
3 //已发送但未被确认的报文数不能超过rwnd
4 if (nextseq - base + seq_limit) % seq_limit < min(rwnd, cwnd):
5     data = f.read(data_size)
6     ...
7 //当rwnd等于零时，发送方继续发送报文段（为便于测试，这里我们发送“No Buffer”）
8 if rwnd != 0:
9     rwnd_count = 0
10 elif rwnd == 0:
11     if rwnd_count % 10000 == 0:
12         client_socket.sendto("No Buffer", server_addr)
13         rwnd_count = (rwnd_count + 1) % 10000
14
15 //def receive():
16 //rwnd不等于0时，继续进行超时重传
17 if rwnd != 0:
18     timer = threading.Timer(time_limit, resend)

```

4. 拥塞控制

congestion control function similar as TCP

防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制的几种方法：慢启动、拥塞避免、快速恢复。

我们使用cwnd作为拥塞控制维护的窗口，sssthresh为网络拥塞程度的标准

当发送方接收到一个可接受的ack时：

若 $cwnd < sssthresh$ ，则 $cwnd = cwnd * 2$ ，此为慢启动阶段

若 $cwnd \geq sssthresh$ ，则 $cwnd = cwnd + 1$ ，此为拥塞避免阶段

当定时器超时，则 $sssthresh = cwnd / 2$ ， $cwnd = 1$

```

1  cwnd = 1
2  //cwnd初始值为1
3
4  if (int(server_pkt.ack) >= base and int(server_pkt.ack) - base < 50) or
    (base - int(server_pkt.ack) > 800) :
5      base = (int(server_pkt.ack) + 1) % seq_limit
6      if cwnd >= sssthresh:
7          cwnd = cwnd + 1
8      else:
9          cwnd = cwnd * 2
10 //client.py def receive()
11 //当发送方接收到可接受的ack时，cwnd的变化
12
13 def resend():
14     sssthresh = cwnd / 2
15     cwnd = 1
16 //client.py def resend()
17 //当重传函数执行时，cwnd的变化

```

18

19 流水线窗口大小为: `min(rwnd, cwnd)`

5.支持多用户

support multiple clients at the same time

server-thread.py使用一个TCP SOCKET在端口9999监听client-thread.py发送的上传请求后

分配一个线程和特定端口, 供双方在该端口上实现LFTP功能

```
1 //server-thread.py
2 if __name__ == '__main__':
3     tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4     tcp.bind(('127.0.0.1', 9999))
5     tcp.listen(3)
6     print('Waiting for connection...')
7
8     while True:
9         sock, addr = tcp.accept()
10        port = port + 1000
11        sock.send(str(port))
12        data = sock.recv(1024)
13        file_recv = pickle.loads(data)
14        sock.close()
15        print("The uploaded port is: %d" % port)
16        print("The fileName is: %s" % file_recv.name)
17        file_size_Mb = file_recv.size / 1024.0 / 1024
18        print("The fileSize is: %.1fMb" % file_size_Mb)
19
20        filePath = filePathPre + file_recv.name
21        t = threading.Thread(target= server, args= (port, filePath))
22        t.start()
23
24
25 //client-thread.py
```

```

26 ip = sys.argv[1]
27 fileName = sys.argv[2]
28 filePath = filePathPre + fileName
29 file_size = os.path.getsize(filePath)
30 file_sent = pickle.dumps(file_info(fileName, file_size))
31
32 tcp = socket(AF_INET, SOCK_STREAM)
33 tcp.connect((ip, 9999))
34 tcp.send(file_sent)
35 port = tcp.recv(1024)
36 tcp.close()
37 print("port: %d" % int(port))
38 server_addr = (ip, int(port))

```

其它

1.显示输出

output information

发送方

传送开始后，显示发送方端口号，开始传送的时间，进度条，以及传输速度（接收方每秒收到的ack了的文件大小），传输结束后，显示结束时间：

```

PS D:\> python client-thread.py 192.168.43.181 sejie.mkv
port: 13111
start time: Mon Dec 3 17:26:14 2018
[#####] 86% 460.0Kbps

```

```

PS D:\> python client-thread.py 192.168.43.181 sejie.mkv
port: 13111
start time: Mon Dec 3 17:26:14 2018
[#####] 100% done!
end time: Mon Dec 3 18:28:50 2018

```

接收方

服务器启动时，输出 “Waiting for connection...”：

```
1 PS D:\代码\lftp\LFTP> python server-thread.py
2 Waiting for connection...
3
```

收到客户端发送的文件时，输出客户端端口号，发送的文件名，文件大小，以及开始传送的时间，然后输出 “the server is ready to receive”， “[12111]” 表示客户端端口号：

```
1 [12111] The uploaded port is: 12111
2 [12111] The fileName is: maogai.zip
3 [12111] The fileSize is: 32.2Mb
4 [12111] start time: Mon Dec 3 15:37:36 2018
5 [12111] the server is ready to receive
6
```

传送完毕，输出 “empty and done”， 关闭socket， 输出 “closed”， 最后输出传输结束的时间：

```
1 [12111] empty and done
2 [12111] closed
3 [12111] end time: Mon Dec 3 15:38:30 2018
4
```

测试文档

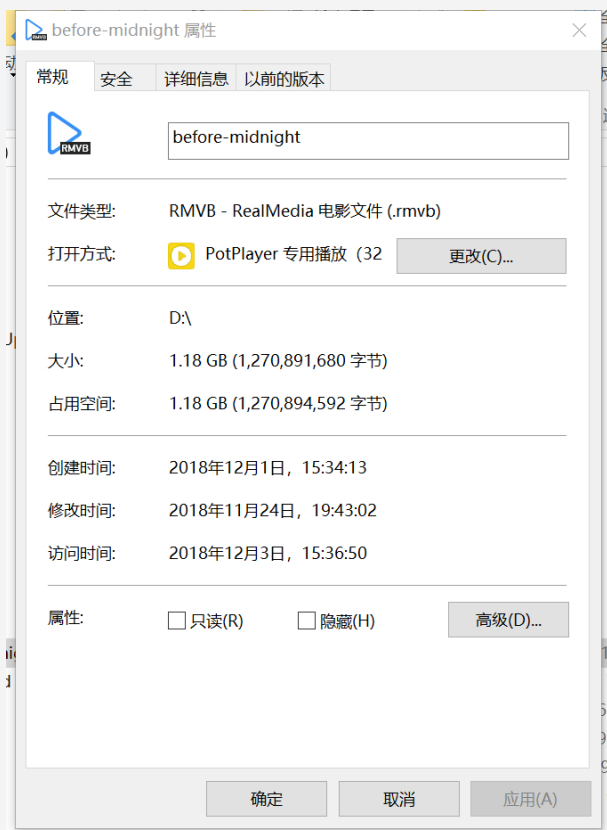
TEST DOCUMENTATION

TEST 1

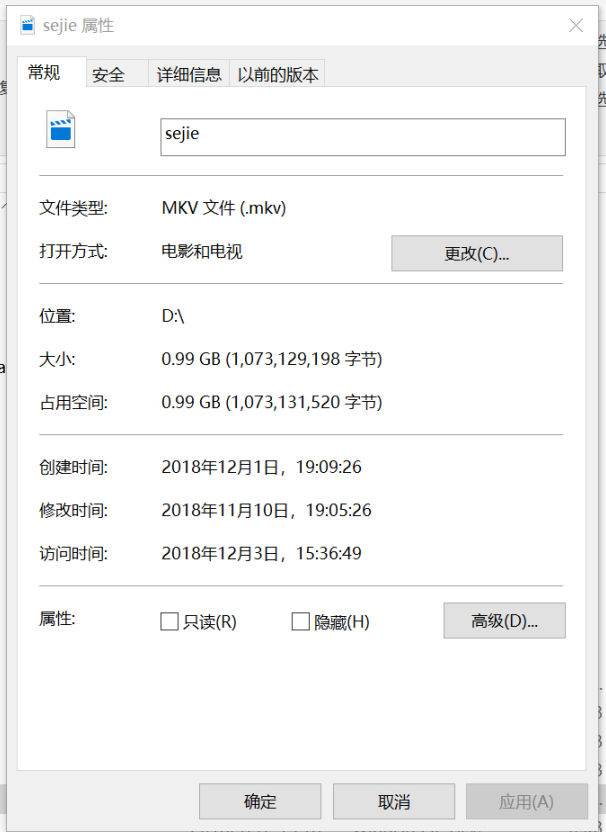
测试用例

两个客户端同时向服务端发送大文件，其中，一个文件是大小为 0.99GB(1,073,129,198字节)的MVK文件(.mvk)，文件名为sejie.mvk，另一个文件是大小为 1.18GB(1,270,891,680 字节)的RMVB文件(.rmvb)。文件名为 before-midnight.rmvb。

before-midnight.rmvb属性：



sejie.mvk属性：



预期效果

服务端能够正确且完整地接收两个文件，服务端接收到的文件的大小、类型属性应与客户端的完全相同。

测试过程

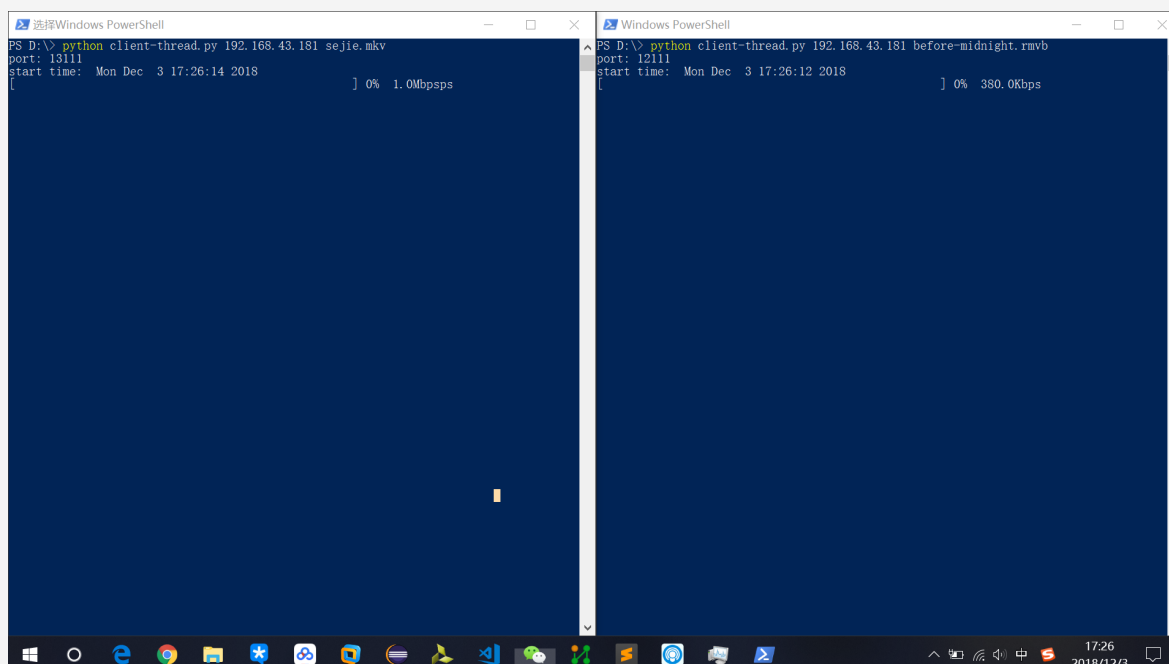
1. 运行server-thread.py

```
python server-thread.py
```

```
PS D:\代码\lftp\LFTP> python server-thread.py
Waiting for connection...
```

2. 运行client-thread.py

python client-thread.py



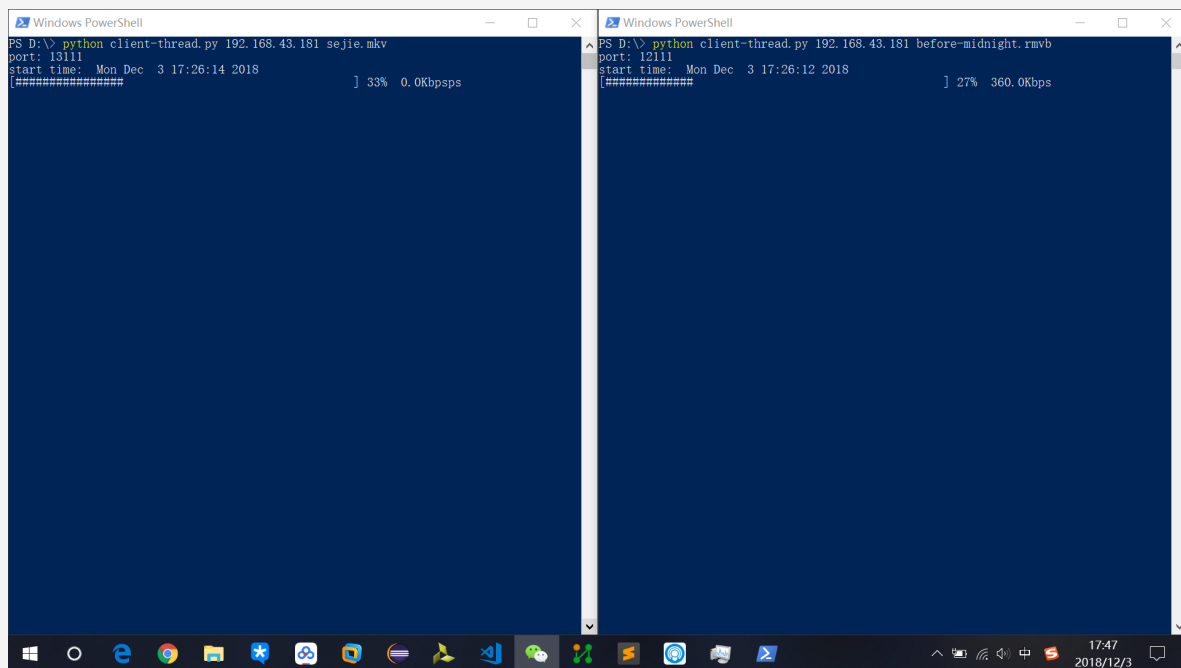
可以看到，开始传送的时间是17:26。

再看下服务端，嗯，服务端成功接收到了文件相关信息，已经准备好接收文件啦：

```
PS D:\代码\lftp\LFTP> python server-thread.py
Waiting for connection...
[12111] The uploaded port is: 12111
[12111] The fileName is: before-midnight.rmvb
[12111] The fileSize is: 1212.0Mb
[12111] start time: Mon Dec 3 17:26:12 2018
[12111] the server is ready to receive
[13111] The uploaded port is: 13111
[13111] The fileName is: sejie.mkv
[13111] The fileSize is: 1023.4Mb
[13111] start time: Mon Dec 3 17:26:14 2018
[13111] the server is ready to receive
```

3. 传送中

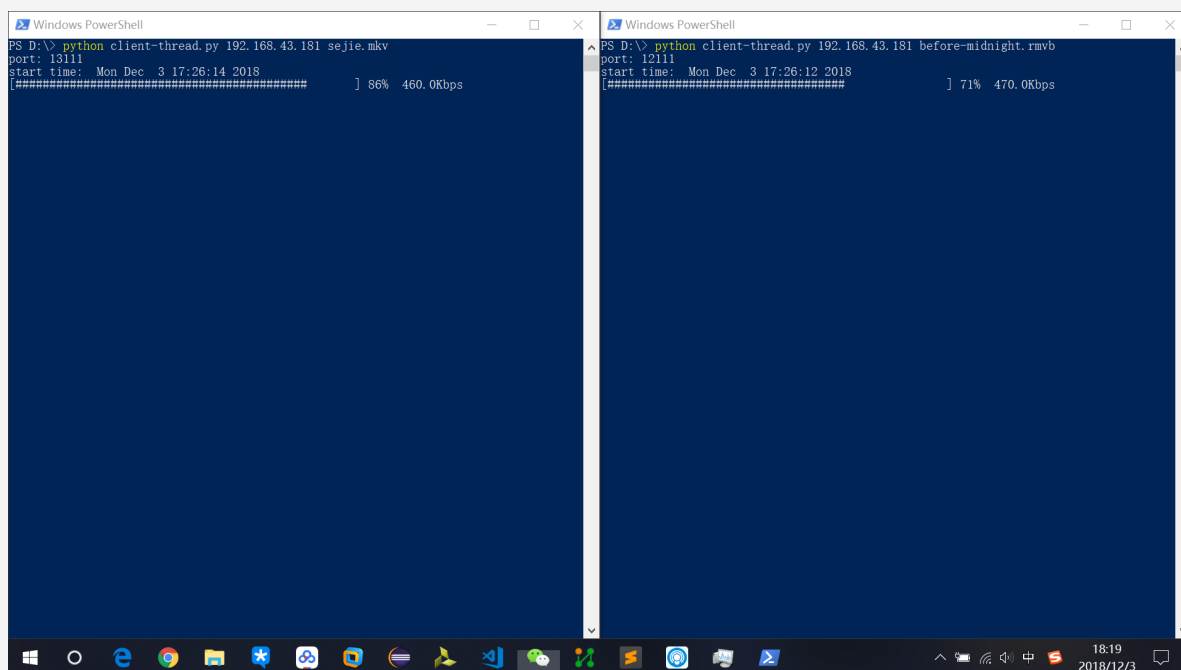
在传送过程中，可以看到传送进度，以及传输速度。20分钟过去了， before-midnight.rmvb已经传送了27%， sejie.mvk传送了33%：



```
PS D:\> python client-thread.py 192.168.43.181 sejie.mvk
port: 13111
start time: Mon Dec 3 17:26:14 2018
[#####] 33% 0.0Kbpsps

PS D:\> python client-thread.py 192.168.43.181 before-midnight.rmvb
port: 12111
start time: Mon Dec 3 17:26:12 2018
[#####] 27% 360.0Kbps
```

又过了30分钟， before-midnight.rmvb传送了71%， sejie.mvk传送了80%， 加油呀！：



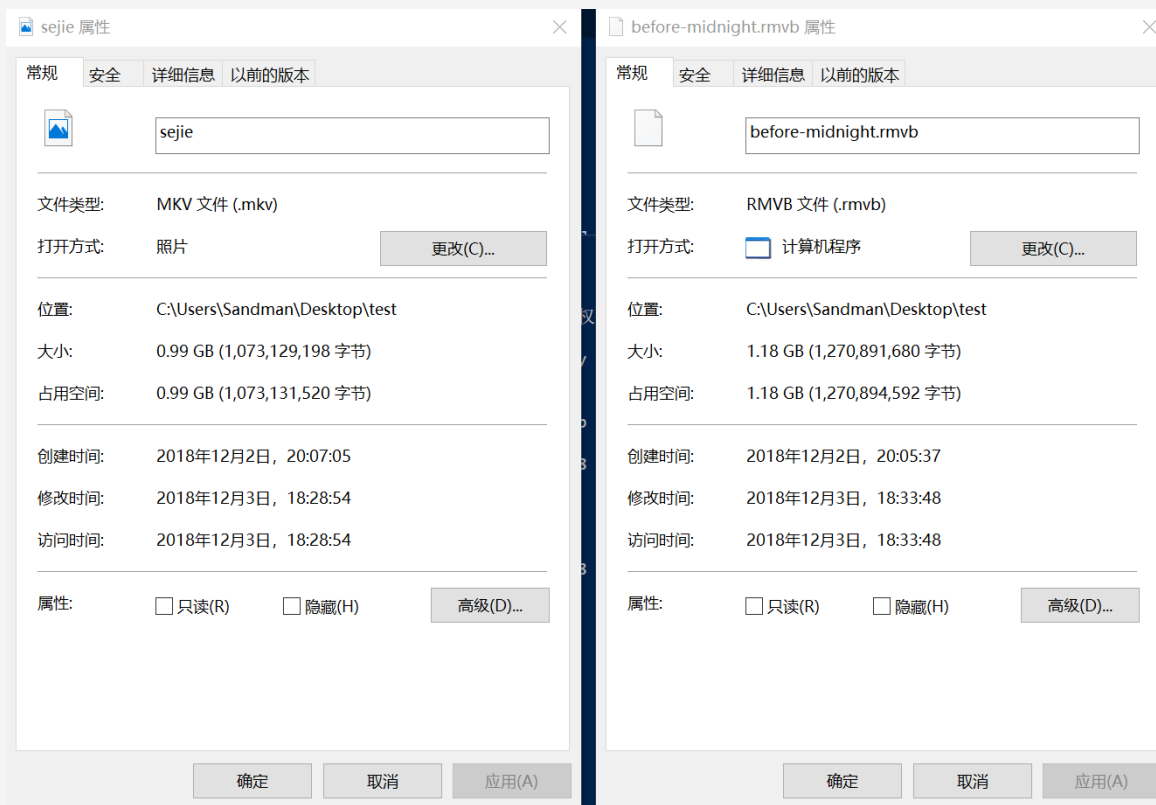
```
PS D:\> python client-thread.py 192.168.43.181 sejie.mvk
port: 13111
start time: Mon Dec 3 17:26:14 2018
[#####] 86% 460.0Kbps

PS D:\> python client-thread.py 192.168.43.181 before-midnight.rmvb
port: 12111
start time: Mon Dec 3 17:26:12 2018
[#####] 71% 470.0Kbps
```

18:28， sejie.mvk成功送到！


```
PS D:\代码\lftp\LFTP> python server-thread.py
Waiting for connection...
[12111] The uploaded port is: 12111
[12111] The fileName is: before-midnight.rmvb
[12111] The fileSize is: 1212.0Mb
[12111] start time: Mon Dec 3 17:26:12 2018
[12111] the server is ready to receive
[13111] The uploaded port is: 13111
[13111] The fileName is: sejie.mkv
[13111] The fileSize is: 1023.4Mb
[13111] start time: Mon Dec 3 17:26:14 2018
[13111] the server is ready to receive
[13111] empty and done
[13111] closed
[13111] end time: Mon Dec 3 18:28:54 2018
[12111] empty and done
[12111] closed
[12111] end time: Mon Dec 3 18:33:48 2018
```

服务端收到的文件属性：



对比客户端，大小、类型完全相同，测试成功！

TEST 2

测试用例

由于时间关系，我们传送一张图片，并显示可靠传输、流量控制、拥塞控制等相关信息。

预期效果

正确进行可靠传输、流量控制、拥塞控制。

测试过程

基本步骤与test1相同，服务端成功收到文件后，我们获得了传送过程的相关信息。下面给出部分信息，完整信息见test.txt。

此时文件正常传输：

```
2018-12-03 19:49:43,095 - DEBUG: ack: 0, base 1, rwnd: 29, cwnd: 2
2018-12-03 19:49:43,127 - DEBUG: ack: 1, base 2, rwnd: 28, cwnd: 3
2018-12-03 19:49:43,127 - DEBUG: ack: 2, base 3, rwnd: 27, cwnd: 4
2018-12-03 19:49:43,142 - DEBUG: ack: 3, base 4, rwnd: 26, cwnd: 5
2018-12-03 19:49:43,157 - DEBUG: ack: 4, base 5, rwnd: 25, cwnd: 6
2018-12-03 19:49:43,157 - DEBUG: ack: 5, base 6, rwnd: 24, cwnd: 7
2018-12-03 19:49:43,190 - DEBUG: ack: 6, base 7, rwnd: 23, cwnd: 8
2018-12-03 19:49:43,190 - DEBUG: ack: 7, base 8, rwnd: 22, cwnd: 9
2018-12-03 19:49:43,190 - DEBUG: ack: 8, base 9, rwnd: 21, cwnd: 10
2018-12-03 19:49:43,190 - DEBUG: ack: 9, base 10, rwnd: 20, cwnd: 11
2018-12-03 19:49:43,190 - DEBUG: ack: 10, base 11, rwnd: 19, cwnd: 12
2018-12-03 19:49:43,190 - DEBUG: ack: 11, base 12, rwnd: 18, cwnd: 13
2018-12-03 19:49:43,204 - DEBUG: ack: 12, base 13, rwnd: 17, cwnd: 14
```

rwnd = 0，进行流量控制：

2018-12-03 19:49:52,838 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,838 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,855 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,885 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,901 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,917 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,933 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,948 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,963 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,980 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:52,996 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97
2018-12-03 19:49:53,010 - DEBUG: ack: 270, base 271, rwnd: 0, cwnd: 97

超时重传，拥塞窗口减小：

2018-12-03 19:49:43,815 - DEBUG: ack: 39, base 40, rwnd: 20, cwnd: 40
2018-12-03 19:49:44,815 - DEBUG: resend 40
2018-12-03 19:49:44,815 - DEBUG: resend 41
2018-12-03 19:49:44,815 - DEBUG: resend 42
2018-12-03 19:49:44,815 - DEBUG: resend 43
2018-12-03 19:49:44,815 - DEBUG: resend 44
2018-12-03 19:49:44,815 - DEBUG: resend 45
2018-12-03 19:49:44,815 - DEBUG: resend 46
2018-12-03 19:49:44,815 - DEBUG: resend 47
2018-12-03 19:49:44,815 - DEBUG: resend 48
2018-12-03 19:49:44,815 - DEBUG: resend 49
2018-12-03 19:49:44,815 - DEBUG: resend 50
2018-12-03 19:49:44,815 - DEBUG: resend 51
2018-12-03 19:49:44,815 - DEBUG: resend 52
2018-12-03 19:49:44,832 - DEBUG: resend 53
2018-12-03 19:49:44,832 - DEBUG: resend 54
2018-12-03 19:49:44,832 - DEBUG: resend 55
2018-12-03 19:49:44,832 - DEBUG: resend 56
2018-12-03 19:49:44,832 - DEBUG: resend 57
2018-12-03 19:49:44,832 - DEBUG: resend 58
2018-12-03 19:49:44,832 - DEBUG: resend 59
2018-12-03 19:49:44,878 - DEBUG: ack: 40, base 41, rwnd: 29, cwnd: 2

