实验报告--Socket 编程

17307130115 吴毓文

一 实验目的

理解Socket API，并使用标准的Socket API编程实现一个简单的文件传输系统，具有基本功能，并且增加了一些用户体验优化的设计，以及一些错误处理机制。

完成情况：一个人完成

二 实验环境

Windows 10, Python 3.6

三 实验内容

1 库函数调用

本次实验调用的python库为 socket、threading、time和os

```
import socket
from threading import Thread
import time
import os
```

2 头部字段

头部字段长度为5个字节，由数字0和1构成，表现形式为字符串（例:00100），在程序中对应着HEADER变量，一共定义了12种头部字段，其余未定义的头部可用于扩展，具体如下所示。

```
_HEADER_NAME_ = "00000"  # user send name of the client
_HEADER_UMSG_ = "00001"  # user's get files request
_HEADER_GFile_ = "00010"  # user's get files content
_HEADER_PMSG_ = "00011"  # user's put files request
_HEADER_ChangePath_ = "00100"  # user change the server's file path
_HEADER_SearchPath_ = "10100"  # user search the server's file path
_HEADER_UCLO_ = "00111"  # user close connection
_HEADER_OMSG_ = "01000"  # other users' message
_HEADER_UMSGB_ = "01001"  # user message back
_HEADER_ENTER_ = "01010"  # user enter the file system
_HEADER_EXIT_ = "01011"   # user leave the file system
_HEADER_USERS_ = "01100"  # users in the file system
_HEADER_TIME_ = "01101"  # time of the file system
_HEADER_SCLO_ = "01111"  # server close connection
```

头部字段

文件系统使用流程：

1 开启服务端，监听端口的socket连接请求

2 用户打开客户端，输入用户名，点击“登入”按钮登录系统

3 用户根据显示的服务器端的文件目录来进行下载，下载方式：请求输入端输入文件名+点击“下载”按钮，额外可以在文件名后加上“@”+自定义存放地址来存放下载的文件至指定目录

4用户根据显示的客户端的文件目录来进行上传，上传方式：请求输入端输入文件名+点击“上传”按钮

5用户点击“查询”按钮来获取服务器存放文件的目录

6用户点击“刷新”按钮来更新服务器端的文件目录

7用户点击离开“登出”按钮文件系统

---

3 服务器端

服务器端的代码文件为**server.py**，改文件首先定义了一系列全局变量：

- ·服务器的IP（127.0.0.1）和端口号(9999)；
- ·服务器的socket信息: server_socket；
- ·list类型变量socket_list，用于管理socket连接；
- ·dict类型变量CLIENTS（文件描述符，用户名），用于标识管理用户，文件描述符可通过socket的fileno()调用获得，服务器通过描述符来唯一标识用户；
- 服务器文件目录 filepath，初始化为"e:/server/"
- ·头部字段
- 最大允许用户连接的数目*MAX_CONN*，设置为1000

在主函数**main**中，清空socket_list和CLIENT，此时还没有用户连接；创建能监听键盘输入的进程并启动，最后调用`startServer()`启动服务器。

`getKeyboard()`调用input()来监听键盘输入。如果输入为q，则调用`closeServer()`关闭服务器；如果输入为端口号，则断开相应的用户连接，否则不进行任何操作。

startServer():

- 创建套接字：面向网络、面向连接

```
server_socket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

- 解决端口冲突问题：设置socket选项，参数socket.SO_REUSEADDR实现端口重用，解决端口冲突问题：

```
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

- 绑定端口、启动监听

```
# 绑定端口
server_socket.bind((IP, PORT))
# 监听连接请求
server_socket.listen(_MAX_CONN_)
```

然后进入for循环，当接受一个连接请求时，保存并输出连接的socket信息并添加到`socket_lists`当中，然后创建一个新的线程，绑定`recvMsg`并运行，以接受用户发来的请求消息。

startServer()对应的是closeSever()，用于关闭服务器，调用方法是socket.close()。在关闭服务器之前，会向所有用户发送提示，并且所有的连接都会被中断。

`recvMsg（）`用于服务器处理用户发来的各种报文，参数为一个socket连接`socket_connection`和文件描述符`fileno()`。主体是while循环，流程如下：

报文发送统一格式: `HEADER+msg`

通过`msg = socket_connection.recv(1024)`阻塞接收用户发来的请求消息，最长只能接收1024字节请求消息，如果接受失败，则可能是服务端或者客户端断开了连接，跳出循环，否则继续；将`msg`解码，并保证长度至少为5，因为头部字段为5字节，然后将头部`HEADER`与数据分离，接下来根据`HEADER`类型来执行下一步操作：

- `HEADER == _HEADER_NAME_`，则是新用户发来的是自己的用户名，并且保证用户名不能为空，否则断开连接。输出新用户用户名。来保证每次发送的数据长度不长于用户端接收的长度限制。最后**将一个时间戳和新用户的用户名发送给所有用户**，来告知新用户加入文件系统，以便实时更新服务器文件改变动态。
- `HEADER == _HEADER_UMSG_`，则是用户发来的文件下载请求消息，然后发送需要发送文件的大小加上用户发送的文件名（或者加上自定义路径），等用户发送准备接收的报文段后，接收确认。**实现三次握手**。开始传输文件，等等传输完成再发消息通知所有用户。具体代码如下：

```
 global filepath#服务器存储文件路径
            filename = None
            sendMsg = None
            if len(msg.split(".")[1])<5:#用户没有自定义文件存储位置
                filename = filepath+str(msg)
                sendMsg = "Client "+CLIENTS[fileno] + " has downloaded " +
msg + " ... \n"
            else:
                filename=filepath+msg.split("@")[0]#用户指定了文件存储位置
                sendMsg= "Client "+CLIENTS[fileno] + " has downloaded " +
msg.split("@")[0] + " ... \n"

            if os.path.isfile(filename):  # 判断文件存在
                # 1.先发送文件大小，让客户端准备接收
                size = os.stat(filename).st_size  # 获取文件大小
                sendToOne(fileno, str(size)+"#"+str(msg), _HEADER_GFile_)#发
送数据长度和文件名和可能的自定义文件路径
                sendMsg = sendMsg+" filesize = "+str(size)+" bytes \n"

                socket_connection.recv(1024)#接收确认

                f = open(filename, "rb")
                for line in f:
                    socket_connection.send(line)  # 发送数据
                f.close()

            #将文件传输信息通知所有用户
            sendToAll(getLocalTime(), _HEADER_TIME_)
            sendToOne(fileno, sendMsg, _HEADER_UMSGB_)
            sendToOthers(fileno, sendMsg, _HEADER_OMSG_)
```

- `HEADER == _HEADER_UCLO_`，则是客户端断开连接，跳出循环即可。在*while*循环结束后，会调用`closeSocket()`来关闭与该客户端的socket连接并清除该用户信息。事实上，只有在这里调用了`closeSocket()`，因为该线程阻塞接收用户发来的消息，如果监听接收失败，那么无疑是连接已断开，所以在其他地方的关闭socket连接操作都是多余的。`closeSocket()`为关闭单一socket连接的方式。
- `HEADER == _HEADER_PMSG_`，则是用户发来的**文件上传**请求消息，然后发送需要发送文件的大小加上用户发送的文件名（或者加上自定义路径），等用户发送准备接收的报文段后，接收确认，开始传输文件，等等传输完成再发消息通知所有用户。具体代码如下：

```
elif HEADER == _HEADER_PMSG_:
            sendToAll(getLocalTime(), _HEADER_TIME_)
            #接收长度
            client_response=socket_connection.recv(1024)
            file_size=int(client_response.decode("utf-8"))

            putMsg = None #将要输出到系统消息界面的信息

            filename = None #将要保存到本地的文件名

            if len(msg.split(".")[1])<5:
               putMsg= "Client "+CLIENTS[fileno] + " has uploaded " + msg +
" ... \n"
               #创建要从客户端下载的文件名
               filename =str(filepath)+msg

            else:
               putMsg= "Client "+CLIENTS[fileno] + " has uploaded " +
msg.split("@")[0] + " ... \n"
               filename =str(filepath)+msg.split("@")[0]

            #准备好接收
            socket_connection.send("1".encode("utf-8"))

            f = open(filename, "wb")
            received_size = 0

            #发送文件
            while received_size < file_size:
                size = 0  # 准确接收数据大小，解决粘包
                if file_size - received_size > 8192: # 多次接收
                    size = 8192
                else:  # 最后一次接收完毕
                    size = file_size - received_size

                data = socket_connection.recv(size)  # 多次接收内容，接收大数据
                data_len = len(data)
                received_size += data_len

                f.write(data)

            f.close()

            sendToOne(fileno, putMsg, _HEADER_UMSGB_)
            sendToOthers(fileno, putMsg, _HEADER_OMSG_)
```

在用`recvMsg()`处理接收到的报文之外，服务器需要根据对应的请求发送报文给客户端。这里使用了三种发送报文的方法：

- 发送给单个客户端`sendToOne()`，来相应客户端的请求。
- 发送给所有客户端`sendToAll()`，来通知所有客户端服务器进行的文件操作，实时更新文件系统信息。
- 发送给除某个文件描述符对应的客户端之外所有的客户端`sendToOthers()`，来通知其他用户某一用户对文件系统的操作。

以上就是服务端的基本实现。

---

4 客户端

客户端用tkinter(缩写为tk)实现了一个文件管理系统，如图3.1所示，具体实现就不详细介绍了，主要逻辑为创建一个画布，将各个组件依次放在画布上，并且将相应的功能函数与对应的按键绑定，用户需要进行的主要操作为输入需要上传或者下载的文件名或者文件名加上自定义的路径，然后点击对应的功能按键即可。其他窗口显示了系统和本地客户端的文件目录，右侧的消息界面主要用来实时更新客户对文件系统的改动，方便客户了解文件系统的最新文件信息。

图 3.1 文件管理系统

客户端代码文件为`client.py`。这里定义了一系列全局变量：

- 服务器的IP地址(127.0.0.1)和端口号PORT(9999)；
- 客户端的socket信息`client_socket`；
- list型变量`UERS`，用来保存用户列表；
- 字符串`GROUPNAME`所表示的系统名称；
- 头部字段；
- 界面用到的颜色RGB值。

在主函数**main**中，程序首先生成一个tk窗口，并定义标题，大小和位置，然后初始化界面，如果初始化发生错误，则调用`popup()`的弹窗功能，一共三类：错误(error)，警告(warning)和提示(info)。

六个功能按钮

如图3.1所示，界面一共有6个功能按钮，左侧三个按钮依次为"登入"、"刷新"和"登出"，右下角三个按钮依次为"上传"、"下载"和"查询"

- 登入：在右侧输入框输入用户名，点击登入或者按下回车键即可登入文件系统；

  绑定的函数为`connectServer()`，流程如下：

  1 检查当前是否存在连接，如果已经连接，则不连接，否则弹窗警告；

  2 判断用户名输入格式，不能为空，不能包含""，当格式正确的时候，才进行下一步，并自动剔除前　　面输入的空格，取有效输入用户名的前20字节；

  3 调用`socket.socket()`，和服务端的操作一样，尝试创建套接字，并保存在client_socket，创建失败则弹窗提示错误并返回；

  4 调用`socket.connect()`，尝试连接服务端，参数为服务器IP地址和端口号，连接失败则弹窗提示错误并且提示错误并返回；

  5 调用`sendToServer()`向服务端发送用户名，如果发送失败则返回；

  6 最后创建一个新的线程，绑定`recvMsg()`并运行，如果发送失败，则返回。
- 刷新：刷新服务器的文件目录，以便获取最新的文件信息；
- 登出：登出文件系统；

  1 首先检查当前连接是否存在，如果存在，则调用`sendToServer`向服务器发送“断开连接”的报文，然后调用`socket.close()`断开连接，如果断开失败，则说明已经断开了，pass即可。然后将`client_socket`设为空值。

  2 如果有必要的话，调用`clearAll()`清除数据并更新显示，主要为清除系统消息界面。
- 上传：在上方的输入框输入本地客户端的文件名，不加自定义路径默认为上传客户默认文件中的文件，点击按钮即可实现上传；
- 下载：在上方的输入框输入服务器的文件名或者加上自定义的文件存放路径，点击按钮即可下载对应的文件；
- 查询 ： 点击可查询得服务器存放文件的路径。

接下来介绍发送报文方法`sendToService()`和接收保温方法`recvMsg()`。

`sendToService(msg, HEADER, needPopup = True)`的三个参数分别为要发送的数据、头部字段和是否弹窗提醒。其流程如下：

1. 首先检查当前连接是否存在，如果不存在，则弹窗提醒并返回*False*，否则进入下一步；
2. 然后将数据和头部字段合并为报文；
3. 最后调用`socket.send()`发送报文，如果发送失败，可能是连接断开/服务器已关闭或者其他原因造成的，则弹窗提醒错误，并且返回*False*，否则返回*True*。这里没有使用`socket.sendall()`来发送报文，一是因为这样可能会将数据分多次发送，后面的数据都没有加上我自定义的头部字段，服务端无法识别；二是限制了发送数据大小，远远小于服务端设定的一次接收1024字节大小，所以一次就能发送所有报文。

`recvMsg()`主要功能为识别并处理服务器端发来的各种报文，主体为一个 while循环，其主要流程如下：

- 通过msg=client_socket.recv(1024)阻塞接收服务端发来的消息，如果接收失败，则可能是服务器自己断开了连接，跳出循环，否则继续；
- 将msg解码，因为头部字段长度为5所以保证长度至少为5，然后将头部HEADER与数据分离，接下来根据HEADER进行下一步操作(封装在recvMsgHandle里)：
  - `HEADER==_HEADER_OMSG_` ,其他用户执行的文件操作信息，显示字体为蓝色；
  - `HEADER==_HEADER_UMSGB_` ，用户自己的操作信息，显示字体为绿色；
  - `HEADER==_HEADER_ENTER_`,任意用户进入文件系统的提示；
  - `HEADER==_HEADER_EXIST_`，任意用户退出文件系统的提示，显示字体为红色；
  - `HEADER == _HEADER_TIME_`，则是服务端发来的时间戳，对应文件操作的时间；
  - `HEADER == _HEADER_SCLO_`，则是服务端断开连接，在系统消息框输出信息即可。

以上就是客户端的基本实现。

五 实验测试

目前文件系统主要实现的功能有:

1监听键盘输入，能够主动关闭服务器，将指定用户断开连接

2接收并处理不同类型的用户请求

3向相应的单一用户传送/接收文件数据

4自定义文件下载路径

5查询服务器存储文件的目录

启动服务端，如图5.1所示

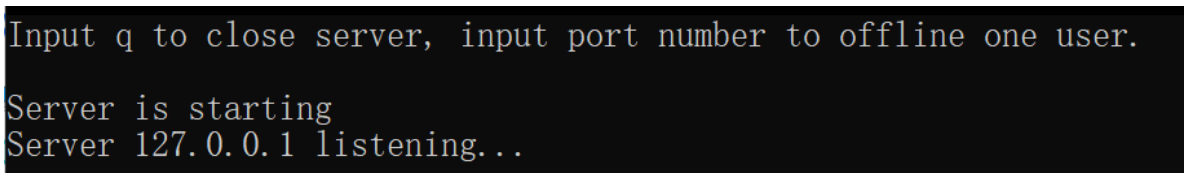

图5.1

启动两个客户端，并进入文件系统，然后执行一系列文件请求。

连接两个客户端，名称分别为student1和student2，对应服务器和客户端显示如图5.2所示：

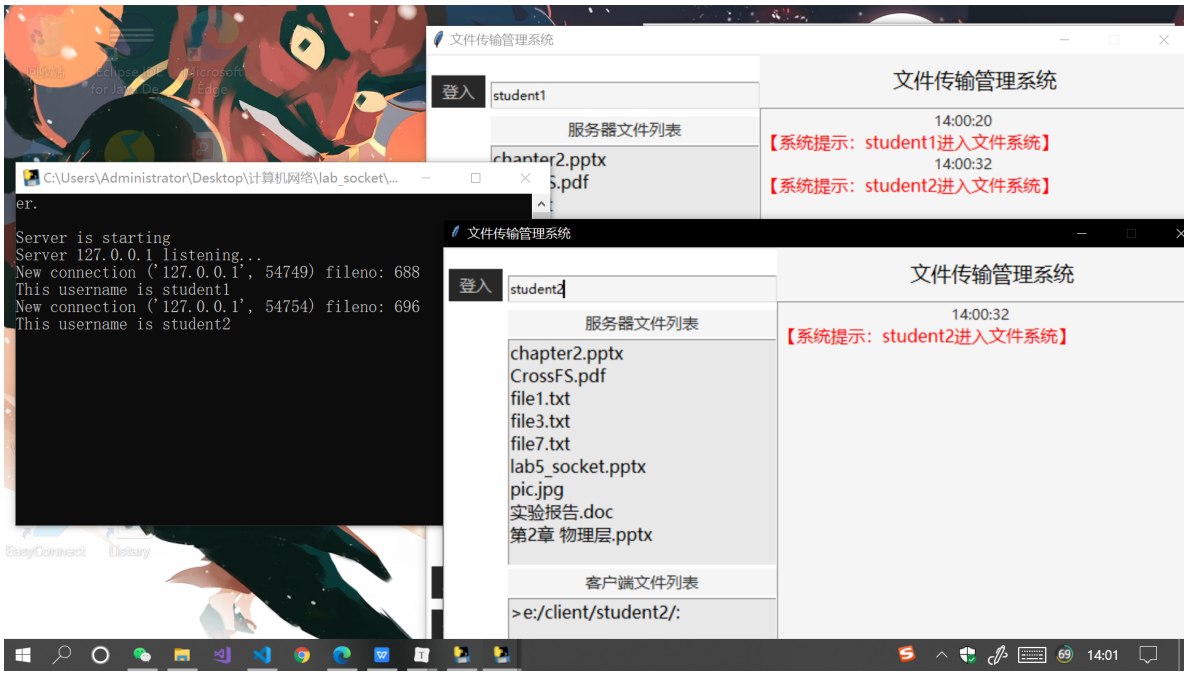

图 5.2

可以看到左边服务器的界面显示了刚登陆进来的客户 名以及他们的端口和文件描述符。

执行操作：客户student1下载文件chapter2.pptx，对应界面显示如图5.3所示：

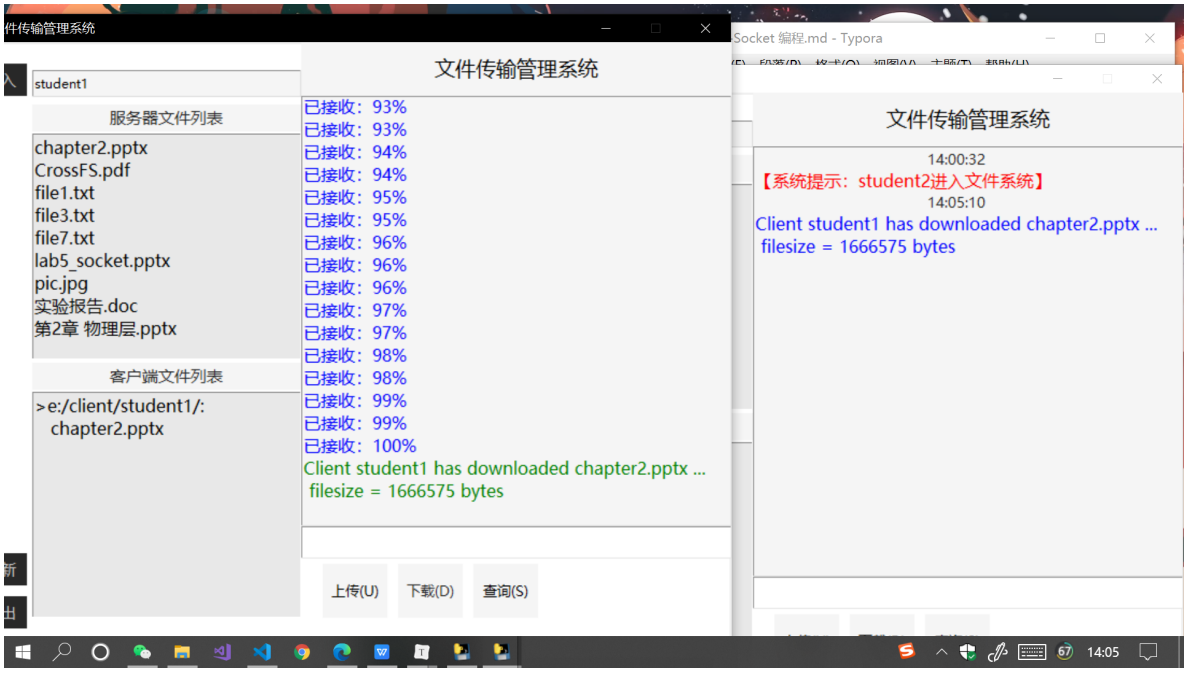

图5.3

执行操作：客户"student2"下载file7.txt后上传file7.txt至服务器，对应界面如图5.4所示：

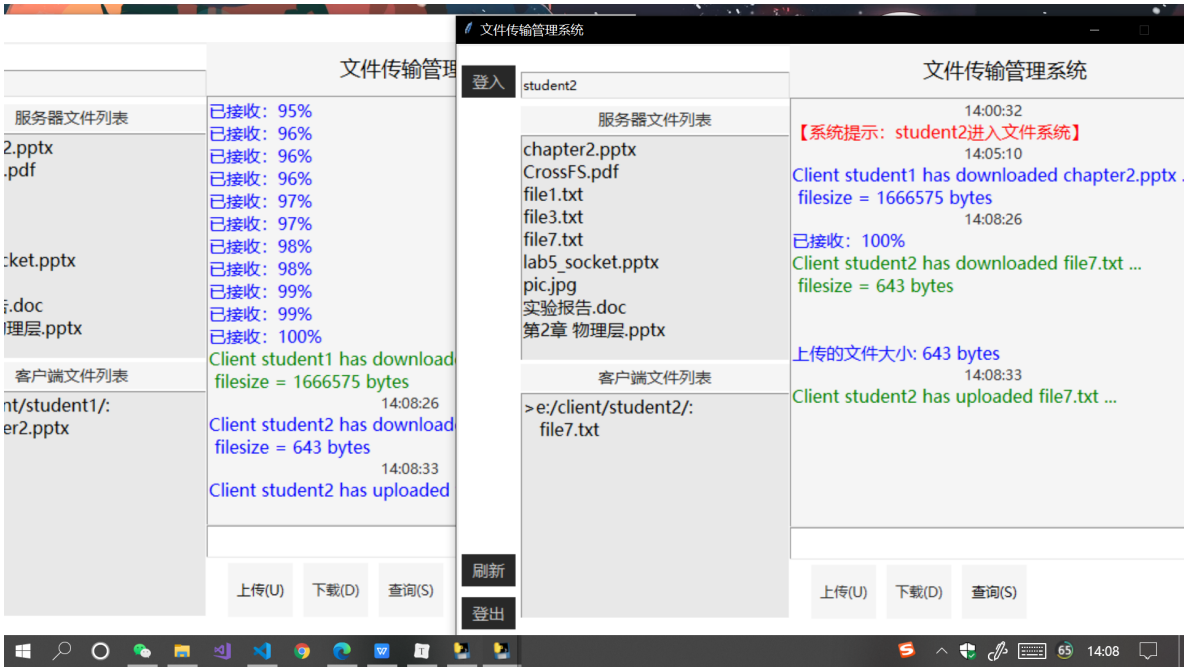

图5.4

执行操作：客户"student1"下载实验报告.doc至自定义目录e:/client/student1/subfile/，对应界面如图5.5所示：

图5.5

执行操作：客户"student1"查询服务器存储文件目录的位置，对应界面如图5.6所示：

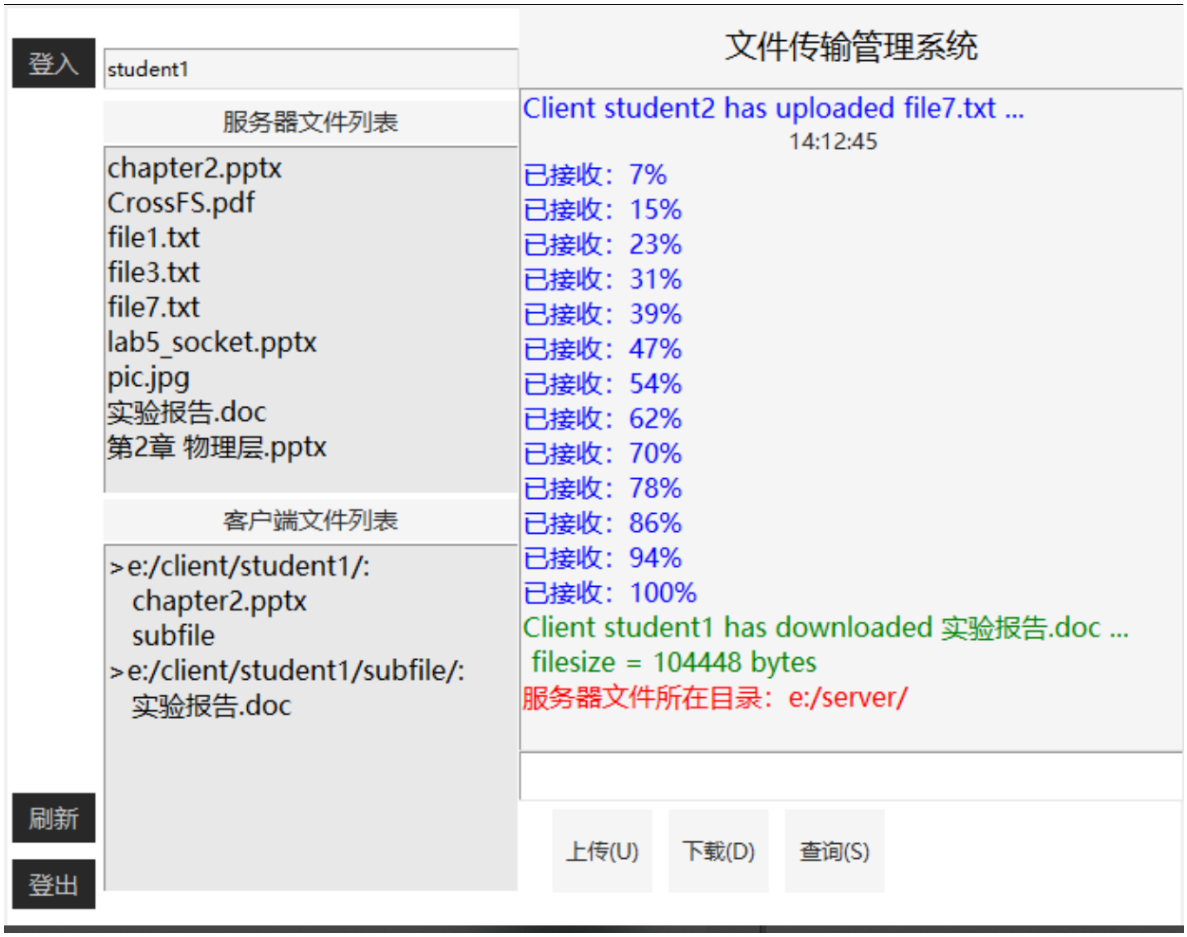

图5.6

执行操作：客户"student1"和客户"student2"点击“登出”按钮来退出系统，对应界面如图5.7所示：

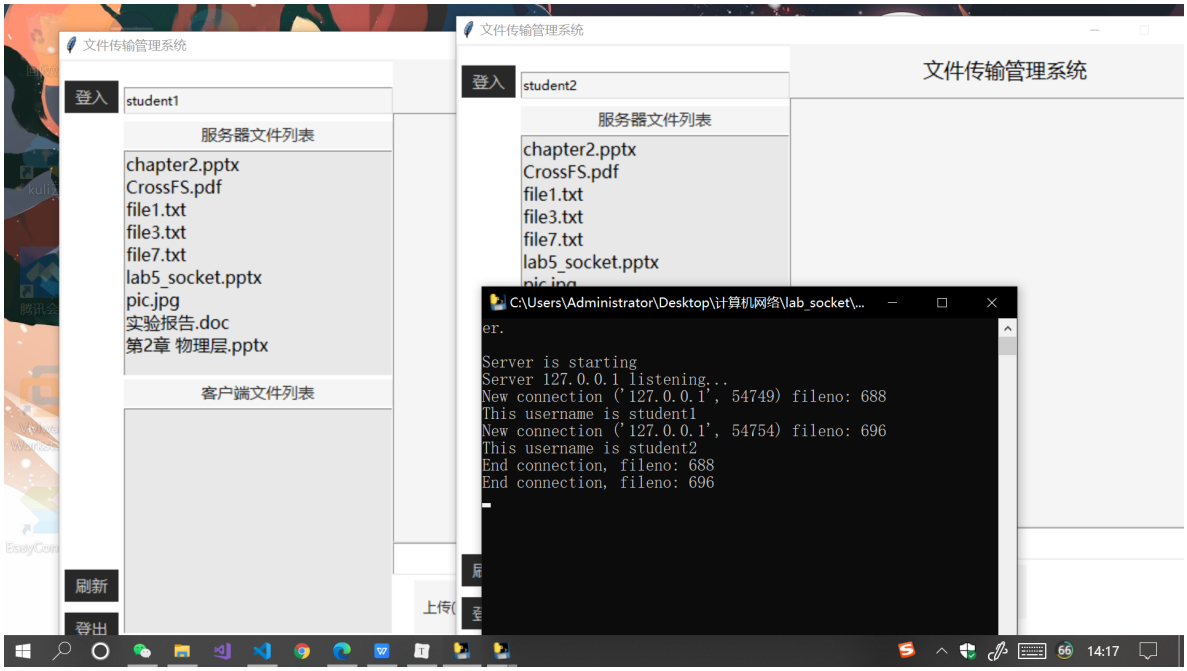

图5.7

这里服务器端显示了断开连接的客户的socket文件的文件描述符。

六 实验总结

这次实验让我悉知了socketAPI在python端的用法以及实现，对三次握手等计算机网络知识理解的更加深入，并处理了一系列服务器与客户的交互。实验过程中主要处理的问题为解决文件传输过程中的粘包，这里我参考了网上的文件传输方法，每传递一次文件内容都要判断传输的部分是否为文件末尾，来解决粘包问题。实验过程遇到的错误很多，说明编程能力薄弱，需要继续努力。