一、*需求分析*

（1）在 socket 客户端实验的基础上，编写自己的服务器

（3）实现的功能包括：

a.客户端上传文件并自动用 RSA 算法加密 / 服务器接收文件

b.客户端下载文件并自动解密（拥有密钥） / 服务器发送文件

c.客户端获得服务器上的文件列表。

（4）本次实验中数据的传输采用 TCP 协议。

服务端 IP 地址为本机 IP 地址

端口为 10086

（5）考虑服务器并发性，依次尝试阻塞式服务器、并发式服务器、异步式服务器，并给出特征分析

二、*操作环境*

1.操作系统：Mac OS

2.编写语言：Java

3.编译软件： Eclipse

三、*概要设计*

客户端：

1.Client 类的基本操作：

<1>public void put() throws Exception
操作结果：将本地文件发送到服务器。
<2>public String listAll() throws Exception
操作结果：返回服务器保存的文件列表，包括可供下载的文件和用户上传的文件
<3>public int get(String file_name) throws Exception
传入参数：要下载文件的名称
操作结果：得到下载的文件，下载成功返回 1，失败返回 0（文件不存在）
<4>public String pre_list() throws Exception
操作结果：为方便得到服务器上传和下载的文件列表，此函数保留以便今后添加新功能
<5>public Client()
操作结果：构造函数，连接服务器

2.Frame 类

主要为显示客户端主界面，包括各种空间和各种事件处理函数，调用 Client 类的各种方法，实现图形界面

3.ListAll 类

主要为显示服务器上的文件列表，包括可供下载的文件和用户上传的文件，调用 Client 类的各种方法，实现图形界面

4.download 类

主要为显示服务器上可供下载的文件，输入相应文件名后可自动下载，调用 Client 类的各种方法，实现图形界面

RSA 加解密：

5. Encrypt 类

采用静态方法 public static byte[] encrypt(byte[] data,String filename) throws Exception，传入加密文件转换成的 byte 数组和加密文件的文件名，返回得到经 RSA 算法加密得到的 byte 数组（加密后的文件），其中，私钥保存在本地。

6. Decrypt 类

采用静态方法 public static byte[] decrypt(byte[] data,String filename) throws Exception，传入从服务器收到加密后文件转换成的 byte 数组和加密文件的文件名，返回得到经 RSA 算法解密得到的 byte 数组（解密后的文件），其中，自动提取本地私钥，若私钥不存在，则无法解密，返回加密文件。

服务器：

7. Sever 类

包括绑定服务器端口，选择服务器保存文件的目录（以供接受客户端文件），依次测试阻塞式服务器、并发式服务器、异步式服务器。

8. Socket_connect 类

<1>public Socket_connect() throws Exception
操作结果：构造函数，传入连接成功的 socket，进行初始化。
<2>public void choose() throws Exception
操作结果：根据客户端传来指令的类型，自动选择发送文件、接受文件、显示客户端文件列表功能。
<3>public void success_message(String str) throws Exception
传入参数：成功时需要显示的信息
操作结果：服务器成功操作后返回给客户端信息
<4> public void error_message(String str) throws Exception
传入参数：失败时需要显示的信息
操作结果：服务器成功失败后返回给客户端信息
<5> public void send——list() throws Exception
操作结果：服务器发送给客户端文件列表
<6> public void receive_file() throws Exception
操作结果：服务器接受客户端发送过来的文件（已加密）
<7> public void send_file() throws Exception
操作结果：服务器发送给客户端文件，若文件不存在，返回失败信息

四、服务器并发性能分析

目前常用的服务器模式共分为三种：

第一种是阻塞式服务器，是最好实现的服务器，也是问题最多的服务器。客户端发送到服务器的请求，服务器会进行排队，依次处理请求。前一个请求没有处理完成，服务器不会处理后面的请求。也就相当于使用一个 while（true）循环，依次调用 seversocket.accept（）函数，每次只接受一个客户端访问，知道访问完成后才处理下一个客户端的请求。这种服务器很容易进行攻击，只需要向服务器发送一个处理时间很长的请求，就会将其他的请求堵在门外，导致其他请求无法得到处理，所以，这种服务器更多的是作为理论模型，实际应用并不多。

第二种是并发式服务器，这种服务器处理请求时，每接收到一个请求，就启动一个线程处理该请求，这种模式的服务器，好处是不会出现阻塞式服务器请求被拥堵的情况，但是也是存在问题的，服务器启动线程是有一定的开销的，请求数量不多的时候，服务器启动线程没有问题，但是请求过多时，将会导致服务器的资源耗尽。所以，会存在一种方式——建立线程池来处理请求，每当请求到来时，向线程池申请线程进行处理，这样，线程池开放多少线程是固定的，不会导致系统资源耗尽，但是依然会有一些问题，当线程池被占用满时，还是有可能出现请求被阻塞的情况，所以这种方式是一种折中的方式。但是，对于并发请求不是很多的场景来说， 使用这种方式是完全可以的。本次客户端最终版本就是采用这种方法编写的。

第三种方式是异步服务器，使用该种方式，一般要借助于系统的异步 IO 机制，如 select 或 poll，这种方式，当一个请求到达时，我们可以先将请求注册，当有数据可以读取时，会得到通知，这时候我们处理请求，这样，服务器进程没有必要阻塞处理，也不会存在很大的系统开销，因此，目前对于并发量要求比较高的服务器，一般都是采用这种方式。

对于三种服务器，本人都进行了测试：其中最初版本是阻塞服务器，当一个客户端连接请求到达时，其他客户端都无法进行连接，需要依次排队等候，效率比较低；后来我采用多线程方法，实现了并发服务器，然而发现有些进程无法执行完毕，经检查发现是被系统自动回收了，因此创建了 ArryList 来保存每个客户端连接的进程，以防被回收。在尝试第三种服务器时，发现使用的方法与前两种大不相同：前两种用的是 java.net 包，主要是 socket 和 seversocket 完成通讯；异步服务器使用的是 java.nio 包，主要是 socketchanel 和 seversocketchanel 完成通讯。有很多差别，因此在实现上无法做到预期的设想，仅仅是做了简单的字符串传送通信，如果想要实现文件上传下载，还需要阅读相关资料深入学习，尤其是 buff 缓冲区的各种操作。所以最终的客户端选用第二种－并发服务器来完成。对其进行改善，建立线程池来处理请求，每当请求到来时，向线程池申请线程进行处理，这样不会导致系统资源耗尽。这些在代码中都已实现。

对于异步服务器，基本格式如下：

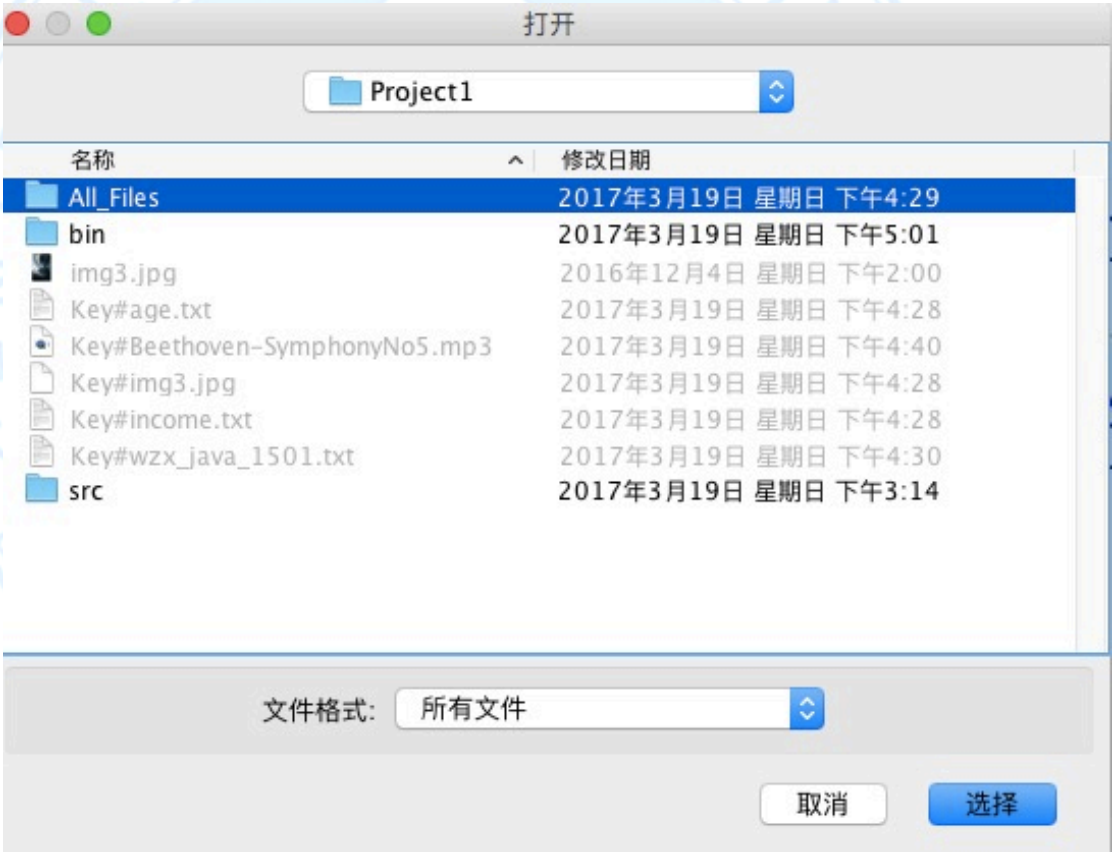
```
public class Server{

String IP = "127.0.0.1";
int PORT = 10086;

public void startServer(String serverIP, int serverPort) throws IOException {
ServerSocketChannel serverChannel = ServerSocketChannel.open();

InetSocketAddress localAddr = new InetSocketAddress(IP, Port);
//服务器绑定地址
serverChannel.bind(localAddr);
//设置为非阻塞
erverChannel.configureBlocking(false);
//注册到 selector，会调用 ServerSocket 的 accept，用 selector 监听 accept 能否返回
        Selector selector = Selector.open();
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);

        while (true) {

            //调用 select，阻塞在这里，直到有注册的 channel 满足条件
            selector.select();
            //可以通过 selector.selectedKeys().iterator()拿到符合条件的迭代器
            Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
            //处理满足条件的 keys
            while (keys.hasNext()) {
                //取出一个 key 并移除
                SelectionKey key = keys.next();
                keys.remove();
                try {
                    if (key.isAcceptable()) {
                        //取得可以操作的 channel
                        ServerSocketChannel server = (ServerSocketChannel) key.channel();
                        SocketChannel channel = server.accept();
                        //注册进 selector，当可读或可写时将得到通知，select 返回
                        channel.register(selector, SelectionKey.OP_READ);
                    }
                    else if (key.isReadable()) {
                        //有 channel 可读,取出可读的 channel
                        SocketChannel channel = (SocketChannel) key.channel();
                        //创建读取缓冲区,一次读取 1024 字节
                        ByteBuffer buffer = ByteBuffer.allocate(1024);
                        channel.read(buffer);
```

```
                    //锁住缓冲区，缓冲区使用的大小将固定
                    buffer.flip();
                    //附加上 buffer，供写出使用
                    key.attach(buffer);
                    key.interestOps(SelectionKey.OP_WRITE);
                }
                else if (key.isWritable()) {
                    //有 channel 可写,取出可写的 channel
                    SocketChannel channel = (SocketChannel) key.channel();
                    //取出可读时设置的缓冲区
                    ByteBuffer buffer = (ByteBuffer) key.attachment();
                    //将缓冲区指针移动到缓冲区开始位置
                    buffer.rewind();
                    //读取为 String
                    String recv = new String(buffer.array());
                    //清空缓冲区
                    buffer.clear();
                    buffer.flip();
                    //写回数据
                    byte[] sendBytes = recv.toUpperCase().getBytes();
                    channel.write(ByteBuffer.wrap(sendBytes));
                }
            } catch (IOException e) {
                //当客户端 Socket 关闭时，会走到这里，清理资源
                key.cancel();
                try {
                    key.channel().close();
                } catch (IOException e1) {
                    e1.printStackTrace();
                }
            }
        }
    }
}
```

此框架的设计经阅读并参考资料编写，对于具体情况需要改变 buff 缓冲区的读写方式，可以成功发送接受信息，完成通信。异步服务器的好处在于，服务器没有工作可做的时候，会等在 select 调用上，不会占用系统资源，而当不同的条件满足时，又可以第一时间被唤醒，执行相应的操作，所以无论从资源的利用上，还是从响应的及时性上都优于前两种。另外，如果 write 和 read 的时间比较长，处理也可以放到线程中处理，这样就结合了并发服务器的优势。

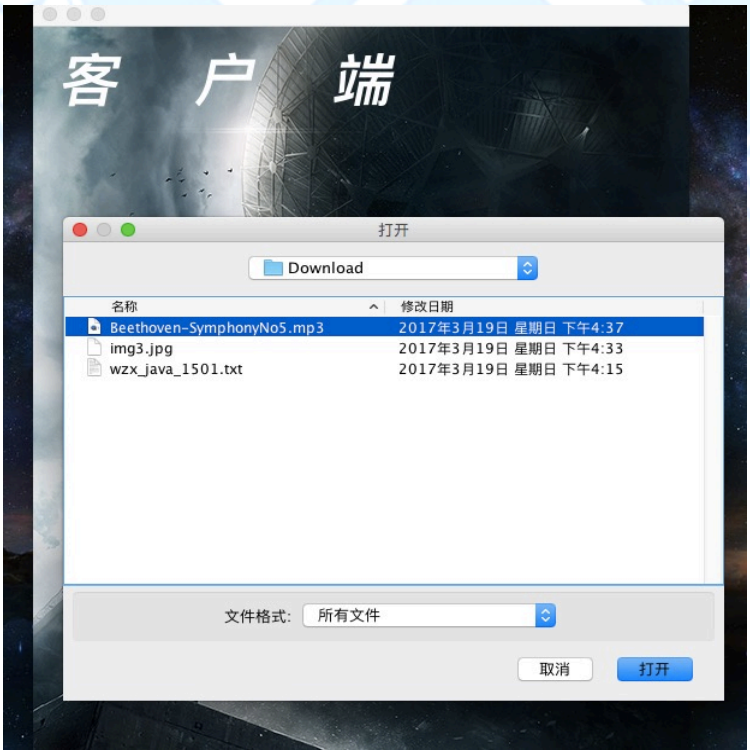
四、调试分析

1.服务器启动，选择存放文件的目录

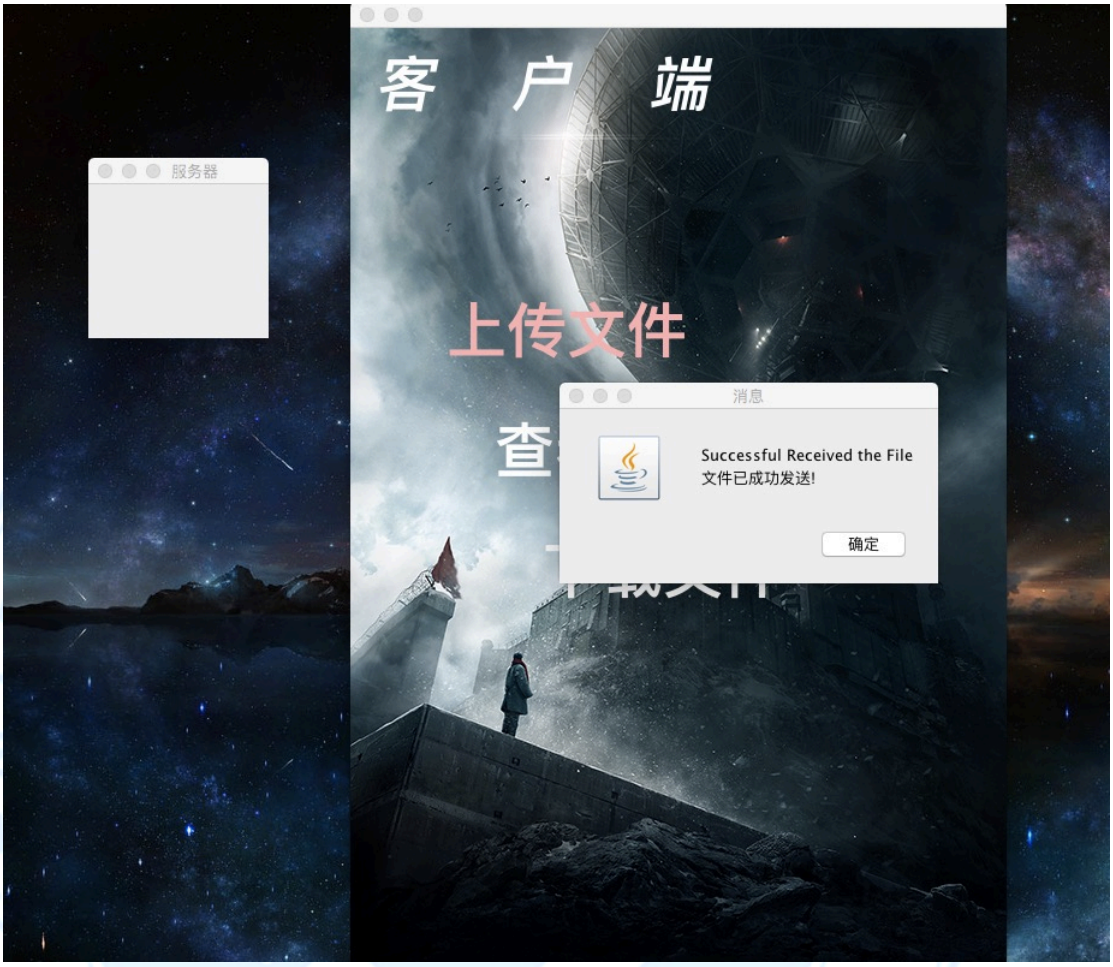


1.客户端运行时的主界面：

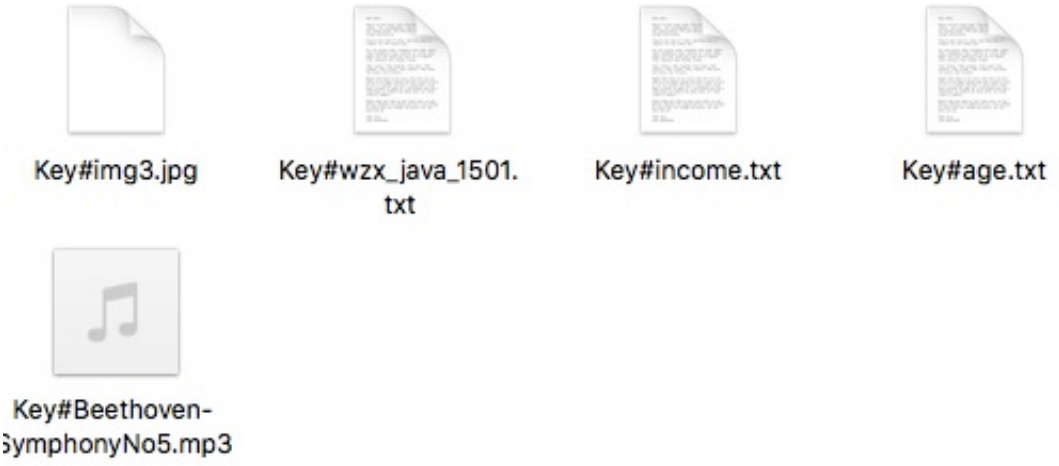
2.点击“上传文件”，显示文件选择器，选取要上传的文件，进行上传



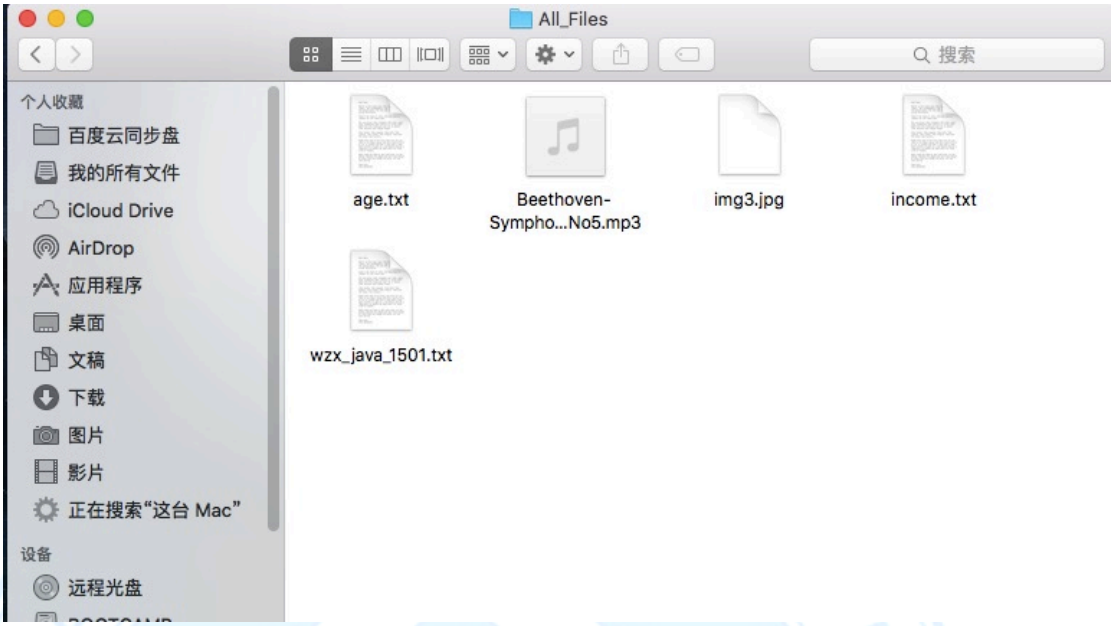
发送后会显示成功上传



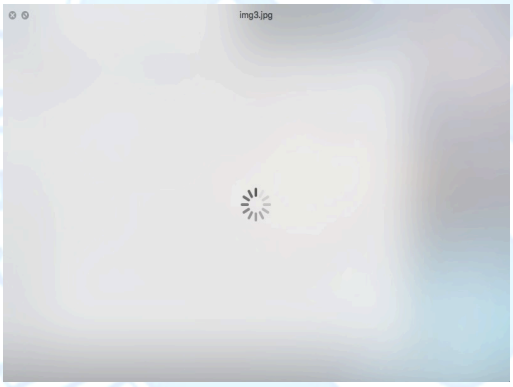
依次上传本地几个文件，生成的私钥会被保存

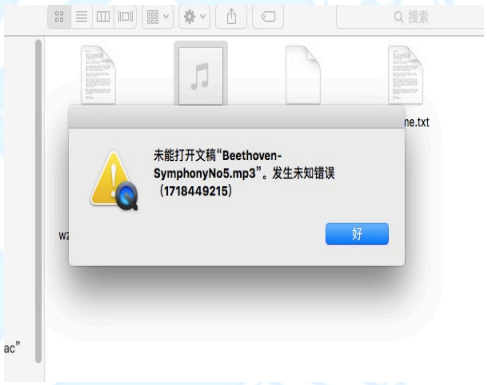


此时服务器上的文件均为加密文件，不可查看

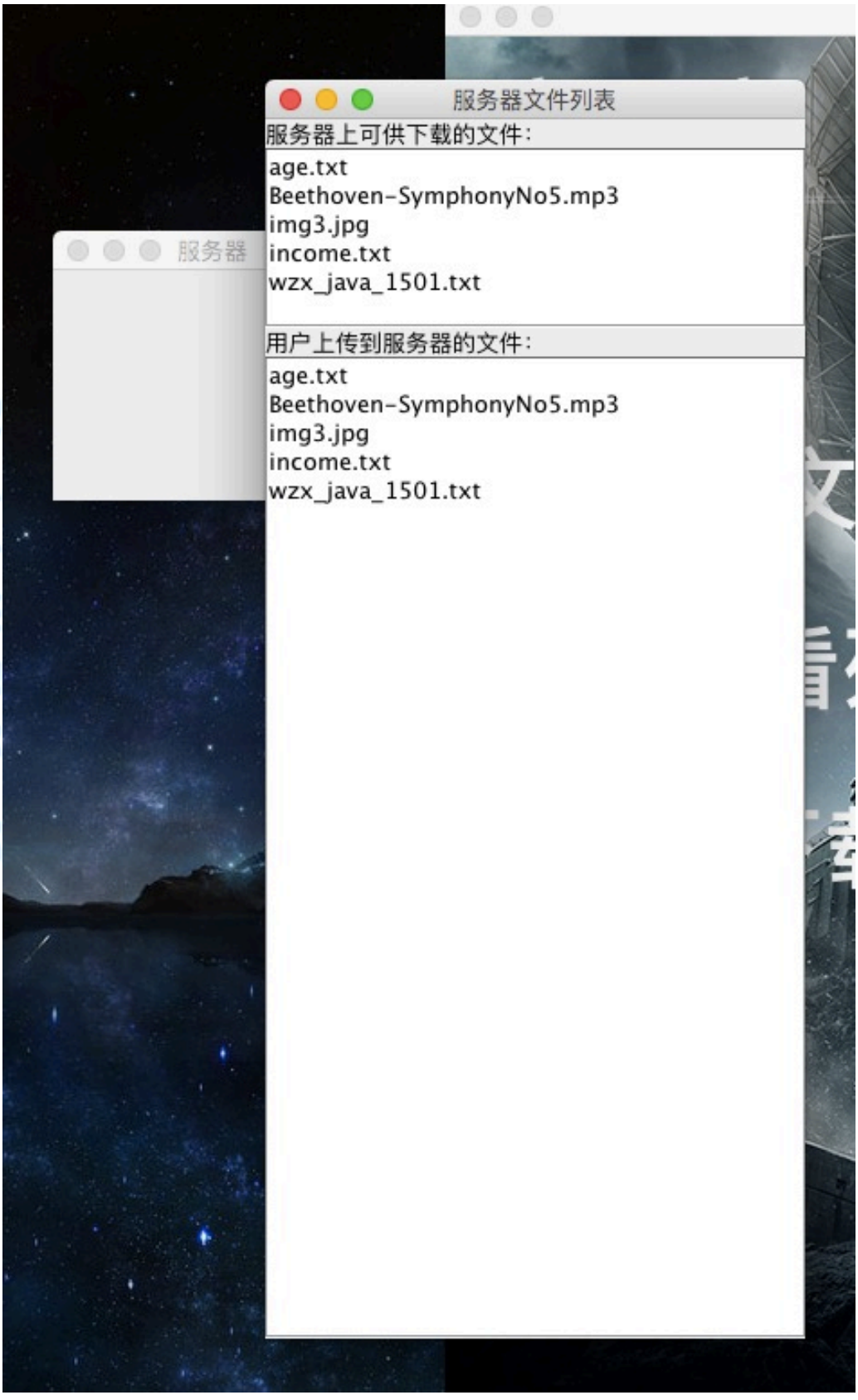


图为服务器上存储的文件

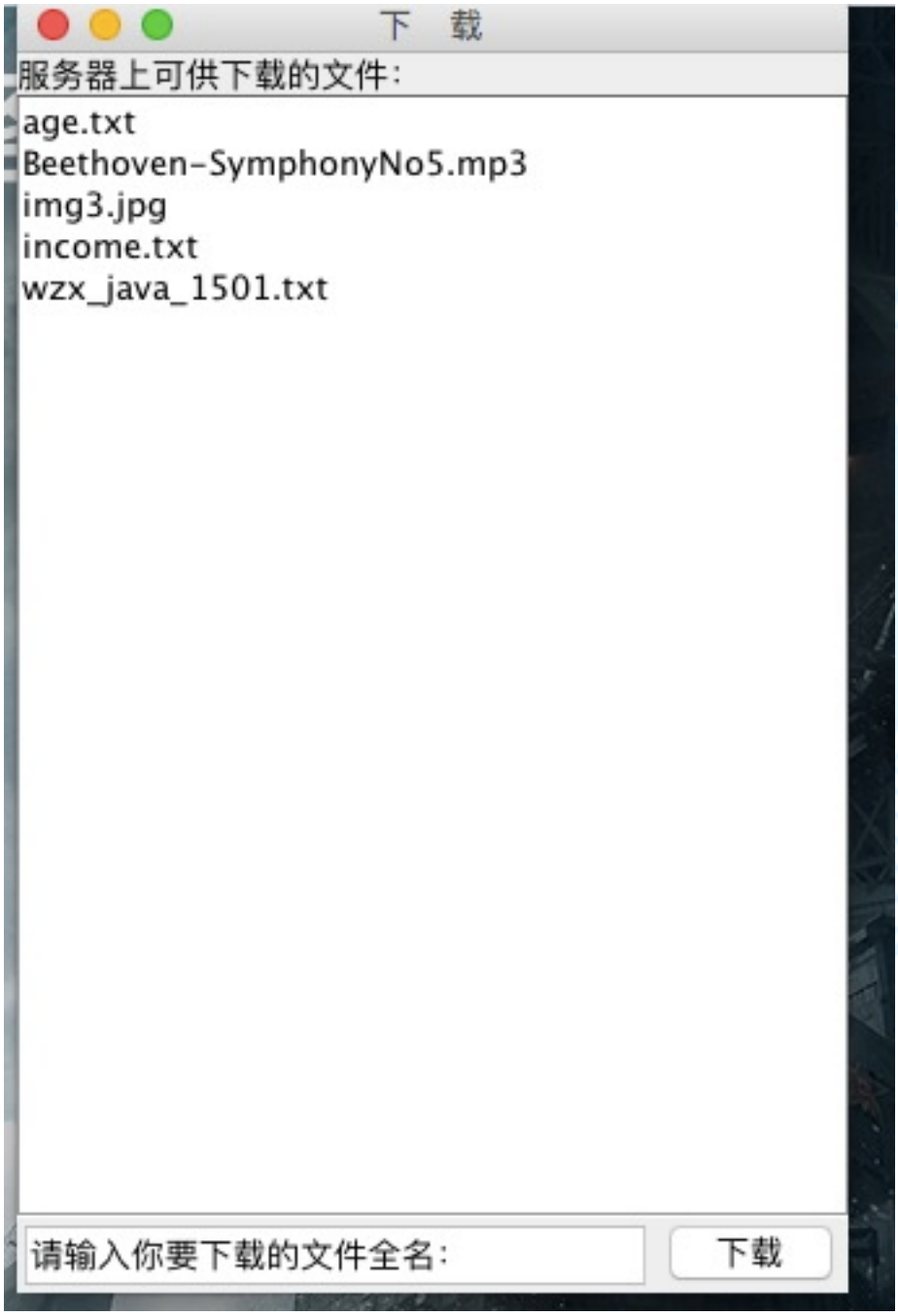




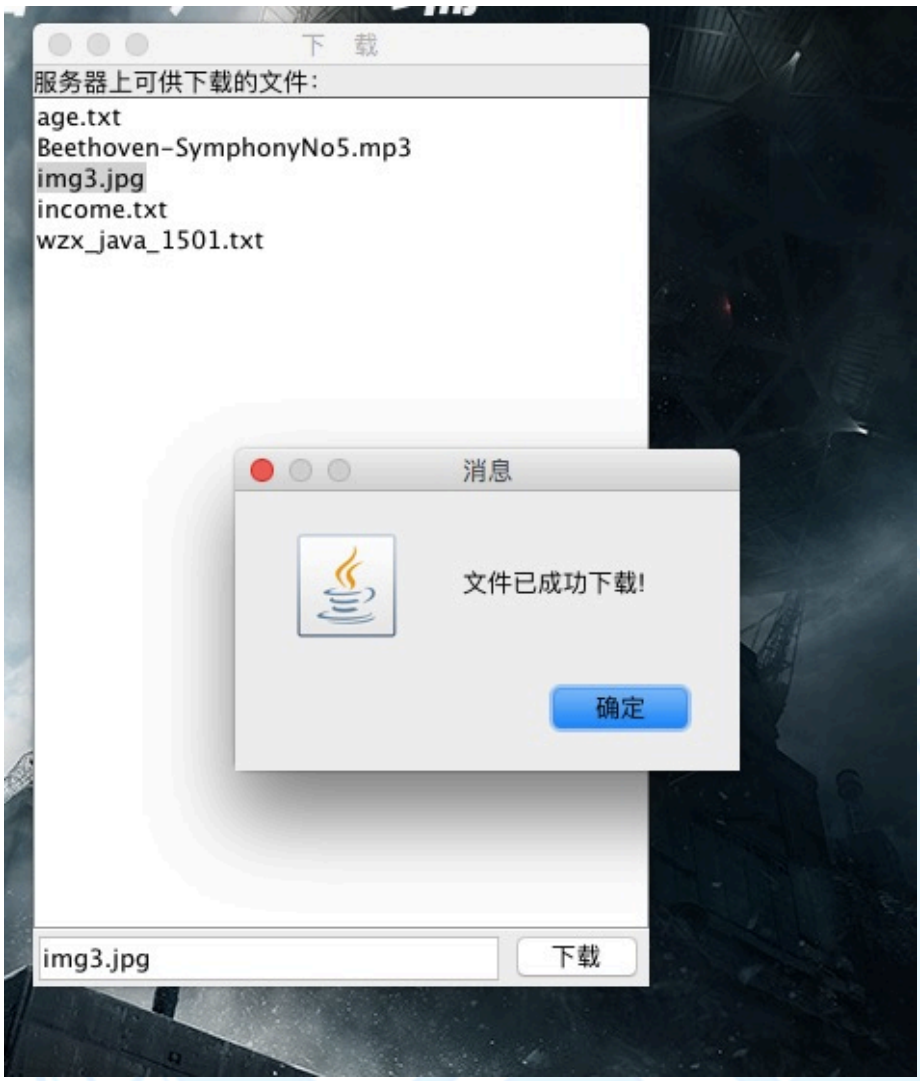
4.点击“查看列表”，会显示服务器上可供下载的文件和服务器上已有的文件，可以看到刚刚上传的文件



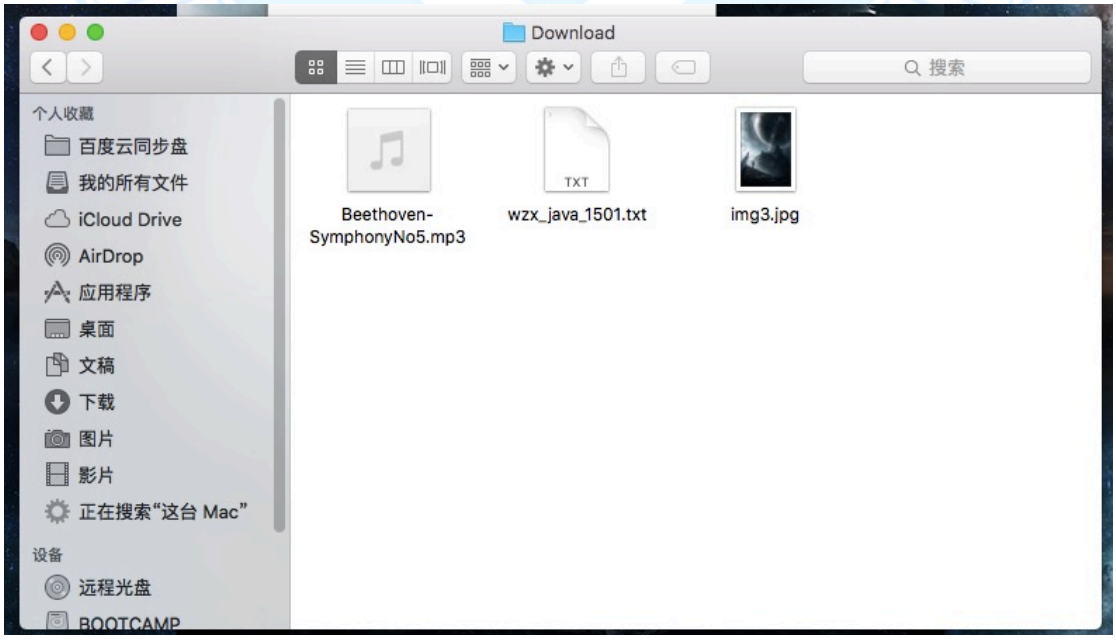
5.点击“下载文件”，会显示服务器上可供下载的文件和服务器上已有的文件，可以看到刚刚上传的文件，输入要下载的文件的全部名称（复制即可），单击下载

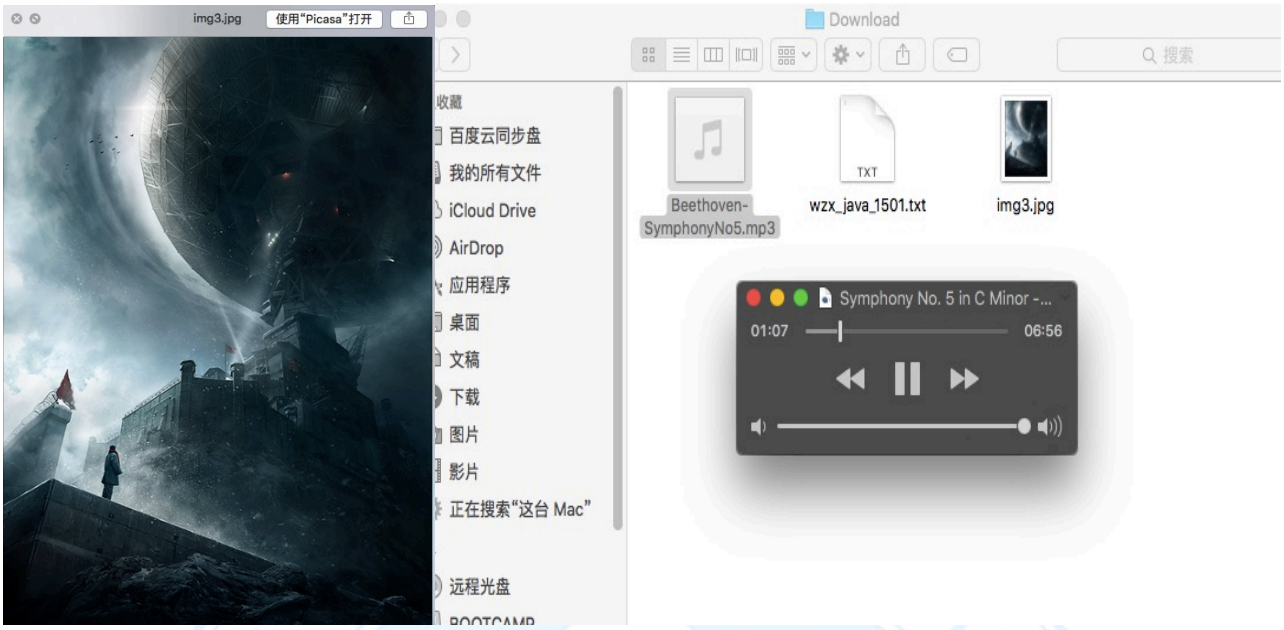


选择文件目录，保存下载文件。下载成功后返回信息。



本地的文件已经自动解密，可以正常使用




img3.jpg
使用“Picasa”打开
Download
搜索
收藏
百度云同步盘
我的所有文件
iCloud Drive
AirDrop
应用程序
桌面
文稿
下载
图片
影片
正在搜索“这台 Mac”
远程光盘
Beethoven-SymphonyNo5.mp3
TXT
wzx_java_1501.txt
img3.jpg
Symphony No. 5 in C Minor -...
01:07
06:56

五、使用说明

＝＝＝＝＝＝＝＝＝＝＝＝＝服务器使用说明＝＝＝＝＝＝＝＝＝＝＝

1.本作品为.jar 文件，需要 java 环境。

2.在客户端连接之前，必须提前打开服务器，否则客户端无法连接。

3.服务器打开时，会指导使用者选择服务器存放文件的目录，此目录一旦选择，在服务器未关闭之前所有从客户端上传的文件都会存放在此目录内。

4.为保证客户端用户隐私，本作品采用了 RSA 加密技术，将文件加密后上传到服务器，所以从服务器无法查看用户上传的文件。

＝＝＝＝＝＝＝＝＝＝＝＝＝客户端使用说明＝＝＝＝＝＝＝＝＝＝＝

<注意>必须用此目录下的客户端软件，此软件与之前单独做的客户端有所不同，请注意！

1.本作品为.jar 文件，需要 java 环境

2.服务器采用图形界面，将各种功能封装，留下“上传文件”、“查看列表”、“下载文件”三个接口。

3.运行程序时，需要在跟目录下保留“img3.jpg”文件，否则会导致主界面无法显示。

4.界面出现后，点击“上传文件”，会弹出文件选择器，由于本人是在mac os下编写的,所以其他操作系统可能会选择不同的默认路径，使用者自己选择要上传到服务器的文件，文件大小不要过大；考虑到用户隐私，上传文件时会将文件自动加密，同时会在本地保留一个“Key＃＋文件名”格式的私钥，请勿删除，否则会导致文件无法解密，请勿轻易传送给陌生人，否则会导致您个人隐私的泄漏，上传结束后会给以提示。

5.点击“查看列表”，会弹出显示服务器上可供下载的文件和服务器中已上传的文件的窗口，由于本服务器采用文件加密技术，所以所有文件公开可供下载,但只有上传者或拥有私钥的用户才可以解密获得原始文件。

6.点击“下载文件”，会弹出下载窗口，需要输入下载文件的全名，如果名称输入错误或输入服务器上不可下载的文件,会给以提示并允许重新输入；如果输入正确则弹出文件选择器，选取目录，文件就会自动下载到指定目录下。如果检测到本地有此文件的私钥，则会自动解密并得到原始文件；如果没有私钥，则会获得加密后的文件，无实

际用处。

7.为方便起见，客户端连接服务器的 IP 地址默认为本机 IP 地址，若要在其他设备上安装服务器，请修改客户端源代码中连接的 ip 地址。