计算机网络期中项目 -- LFTP实验报告

姓名：陈明亮

学号：16340023

一、实验需求

- 实验要求采用`C、C++、Java、Python`中的一种编程语言，实现用于网络中两台主机相互通信，传输大文件的网络应用 -- `LFTP`
- 实验要求采用`UDP`作为底层传输协议，但需要手动实现与`TCP`100%相同的可依赖性，同时需要实现协议的流控制，拥塞控制机制
- 实验架构要求为`Server-Client`结构，同时客户端能够使用不同命令上传或下载相应文件到服务端，服务端则需要提供相应的服务接口，更需要同时可以服务多个处于连接状态的客户端，同时满足它们的需求
- 该应用需要在运行过程中提供有用的`Debug`信息，利于观察应用所处状态，以及客户端，服务端分别的运行状态，当前连接的建立，当前文件的传输进度等等信息

二、实验架构

1. 总体结构

- 实验代码采用`Java`语言，严格遵循`客户端-服务端`机制，将两者功能划分开，分别进行不同的逻辑代码编写。
- 对于前后端代码，均采用了功能分层的写法，将对应端上的程序划分为`应用层-会话层-传输层`，同时服务端实现可用于多个客户端同时连接的功能，传输层实现多线程传输服务功能。
- 客户端程序的基本执行逻辑：
    1. 应用层负责展示各项连接、传输状态信息，以及使用界面信息，接收用户输入命令，进而将命令转交给会话层。
    2. 会话层负责结合相应命令，进行对目标服务端主机的3次握手请求，等待握手完成，进行传输层进行文件上传 or 下载过程。
    3. 传输层只负责接收对应的需要传输的文件信息，进行文件上传 or 等待下载完成即可，传输过程结束时将成功信息返回到会话层。
    4. 会话层接收到对应的传输完成信息，进行与目标服务端主机的4次挥手断开连接，进而将之前过程中的全部有用信息传输给应用层进行展示。
- 服务端程序基本执行逻辑：
    1. 应用层同样负责展示服务端相关状态信息，包括客户端连接信息，以及不同客户端发来的请求服务信息。
    2. 会话层负责与发起连接请求的客户端进行3次握手过程，完成连接之后将对应连接客户端主机信息放入连接池，服务端则会根据服务类型为每个客户端分配上传 or 下载线程，进行对应文件传输。
    3. 传输层为面向不同服务端主机的文件传输线程，独立于主线程而并行进行数据传送 or 接收，执行完成之后会发出结束标志。

4. 会话层等待结束请求与挥手报文，从连接池中删除对应主机信息，应用层打印相关文件传输成功讯息，以及主机挥手成功信息。

- 同时服务端部分结合流控制机制，设置滑动窗口(缓冲区)用于发送数据包；拥塞控制采取控制拥塞窗口进行数据包发送时的限制，采取`慢启动、拥塞避免、快重传、快恢复`机制。

2. 客户端具体实现逻辑

- 应用层：`LFTP_Client.java`，`ProcessBar.java`

  客户端应用层实现命令行界面，接收用户输入命令，如:`LFTP lsend myServer largeFile`等命令进行解析，分析结果转给相应的会话层函数进行执行。

```
public static void main(String[] args){
    // Init the client
    client = new UDPClient();
    // Scan user's input
    Scanner input = new Scanner(System.in);
    String tip = "-------------- Welcome to LFTP --------------\nEnter commands to
start using LFTP Client, \"help\" for details\nLFTP-Client$: ";
    System.out.printf(tip);
    String command = input.nextLine();
    while(!command.equals("exit")){
        if(!Parse(command)) break;
        System.out.printf("LFTP-Client$: ");
        command = input.nextLine();
    }
    client.close();
    System.out.println("Exit.");
}
```

  分析：此处定义的`UDPClient`为客户端会话层类对象，`Parse()`函数为相应的命令解析函数，返回`bool`表示解析命令合法与否，核心执行代码如下：

```
try {
    switch(sp[1]){
        case "lsend":
            System.out.println("[Info] Client is ready to send file to server....");
            client.handleSendRequest(sp[3], sp[2]);
            res = "";
            break;
        case "lget":
            System.out.println("[Info] Client is ready to get file from server....");
            client.handleGetRequest(sp[3], sp[2]);
            res = "";
    }
} catch (Exception e){
    e.printStackTrace();
    return false;
}
```

分析：`Parse`函数解析参数，将合法命令对应的参数值传递给会话层的启动函数，处理用户输入的请求。

```
public class ProcessBar{
    private long startPoint;  // start point percent
    private long barSize;  // file size (bytes)
    private int barLength;  // bar length
    private String fileName;
    private String barInfo;
    private final int MAX_LENGTH = 50 * 1024;

    public ProcessBar(int start, long barSize, String fileName){
        this.startPoint = start;
        this.barSize = barSize;
        this.fileName = fileName;
        this.barLength = 30;
    }

    public void initBar(boolean flag){
        ....
    }

    public String showBar(int sign){
        ....
    }

    public void updateBar(int currPoint){
        ....
    }

};
```

分析：上述代码为应用层的传输进度条实现，总体类定义较长此处只放出基本架构。本类主要打印文件传输过程的动态进度条，显示传输进度，其中类函数由传输层进行进度变化，已达到应用层更加美观易于理解。

- 会话层：`UDPClient.java`

  客户端会话层实现客户端主体类定义：`UDPClient`，该类包含会话层的执行逻辑，以及传输层的对应函数。首先在该类的构造函数中，需要实现相应的初始类私有变量，包括用于网络数据传输的`DatagramSocket`对象，实际上就是底层`UDP`的传输套接字，用于传输数据包的最基本对象。

```
// Startup constructed method
public UDPClient(){
    try {
        socket = new DatagramSocket(PORT_NUM);
        // Default server host -- localhost
        serverHost = InetAddress.getByName("127.0.0.1");
        packetQueue = new LinkedList<>();
    } catch(SocketException e) {
        e.printStackTrace();
    } catch(IOException e) {
        e.printStackTrace();
    }
}
```

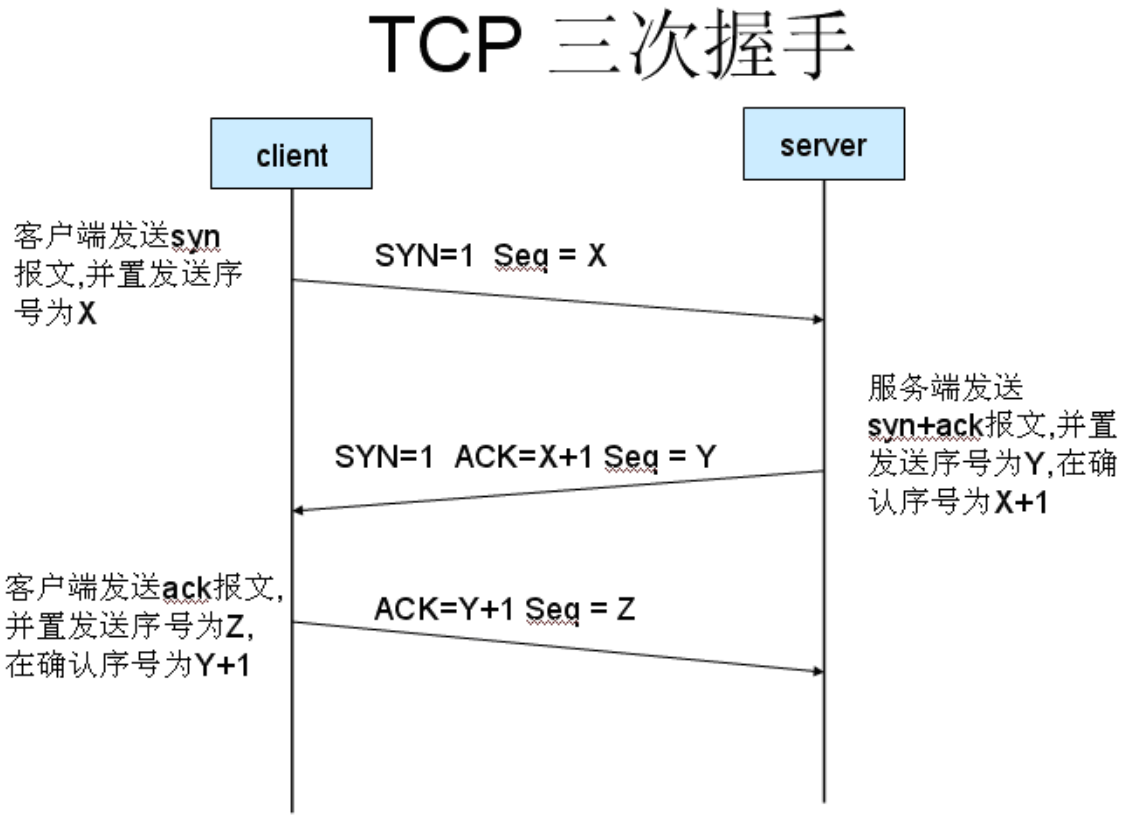
分析：此处默认服务端为本地`localhost`，但在之后的会话层3次握手中会结合应用层传入服务器IP地址，进行类内部服务端主机地址变换。

```
private boolean ShakeHandsWithServer(String serverPath) throws IOException{
    String[] address = serverPath.split(":");
    serverHost = InetAddress.getByName(address[0]);
    serverPort = Integer.parseInt(address[1]);
    int errTime = 0;
    while(true){
        socket.setSoTimeout(2500);
        // Step 1. send Pakcet_SYN_1 to server
        UDPPacket syn_packet_1 = new UDPPacket(this.seq++);
        syn_packet_1.setSYN(this.syn++);
        syn_packet_1.calculateSum();
        sendDataPacket(syn_packet_1);
        // Step 2. wait server return packet and check
        UDPPacket syn_packet_2 = receivePacket();
        if(syn_packet_2 != null && syn_packet_2.isACK() && syn_packet_2.isSYN()
            && syn_packet_2.getACK() == this.syn && syn_packet_2.checkSum()){
            // Step 3. send ack to server to establish it
            UDPPacket ack_packet = new UDPPacket(this.seq++);
            ack_packet.setACK(syn_packet_2.getSYN() + 1);
            ack_packet.calculateSum();
            sendDataPacket(ack_packet);
            System.out.printf("[Info] Successfully establish connections with Sever: %s,
serverPort: %s\n", address[0], address[1]);
        return true;
        }else {
            System.out.println("[Error] Shake hands packet time out....");
            errTime++;
        }
        if(errTime > 5){
            System.out.println("[Error] Shake hands with server failed! Too many tansfer
errors!");
            break;
        }
    }
    return false;
```

```
}
```

分析：

1. 上述代码为客户端3次握手执行函数，严格按照`TCP`的标准三次握手机制：客户端发送`SYN`报文，置发送序号为`X`，然后等待服务端回应；服务端确认无误之后，返回`SYN+ACK`报文，置确认序号为`X+1`；客户端接收确认报文，发送确认报文，置确认序号为`Y+1`。一般三次握手完成之后，相应的上传或下载服务也相应会启动。
2. 还可以看到上述握手过程复现`丢包重传`机制，设定套接字的接收握手报文的时限为`1秒`，若未收到对应的确认`ACK`报文则认为丢包发生，产生重传过程。
3. 此处的`UDPPacket`为自定义的可序列化数据包类结构，具体用于存储与客户端-服务端之间的信息传输，定义内容将在下文提到。



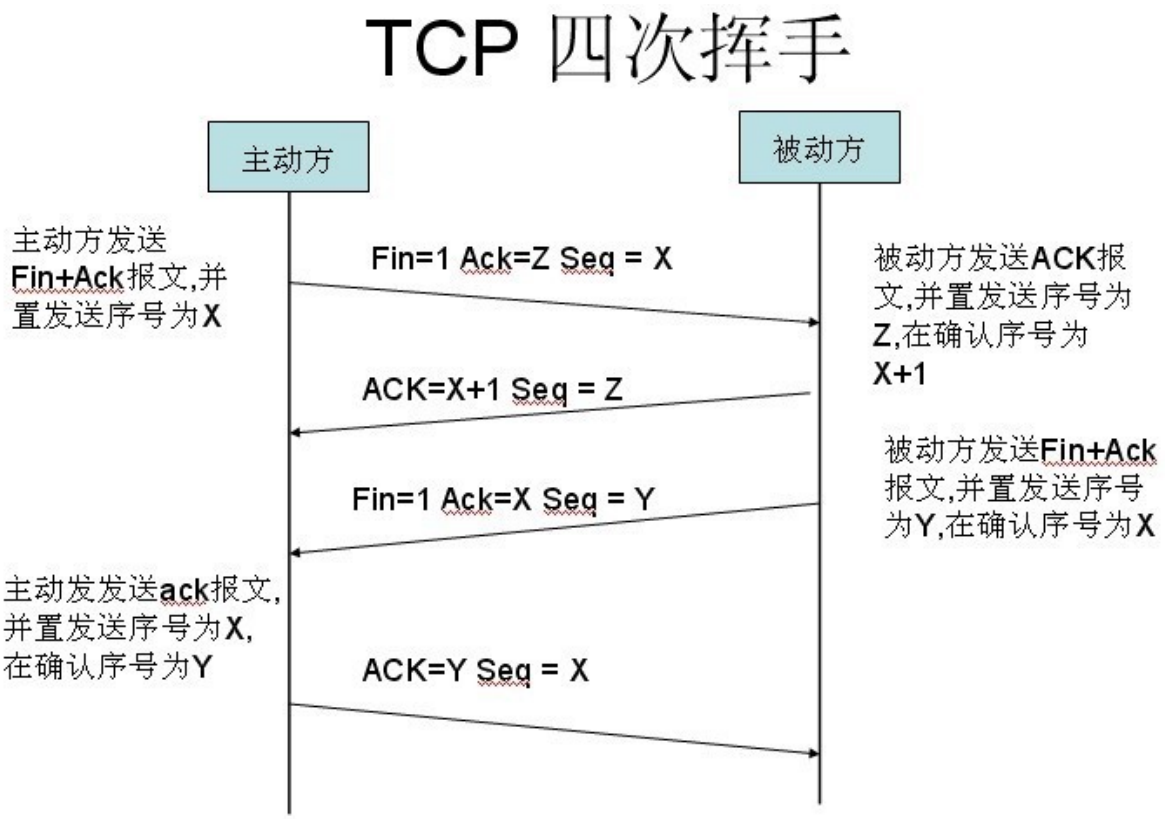

```
private void WaveHandsWithServer() throws IOException{
    // Init: time out
    int errTime = 0;
    while(true){
        socket.setSoTimeout(2500);
        // Step 1. client sends init fin packet to server
        UDPPacket fin_packet_1 = new UDPPacket(this.seq++);
        fin_packet_1.setFIN(this.fin++);
        fin_packet_1.calculateSum();
        sendDataPacket(fin_packet_1);
        // Step 2. wait server's response
        UDPPacket ack_packet;
        while((ack_packet = receivePacket()) == null){}
        if(ack_packet != null && ack_packet.isACK() && ack_packet.checkSum()){

            // Step 3. wait server's fin packet
```

```
            UDPPacket fin_packet_2 = receivePacket();
            if(fin_packet_2 != null && fin_packet_2.isFIN() && fin_packet_2.checkSum()){
            // Step 4. send final ack packet to server
            UDPPacket ack_packet_fin = new UDPPacket(this.seq++);
            ack_packet_fin.setACK(fin_packet_2.getSeq() + 1);
            ack_packet_fin.calculateSum();
            sendDataPacket(ack_packet_fin);
            System.out.println("[Info] Successfully quit the connection with target
server....");
            break;
            }else {
                System.out.println("[Error] Wave hands packet time out....");
                errTime++;
            }
        }else {
            System.out.println("[Error] Wave hands packet time out....");
            errTime++;
        }
        if(errTime > 5){
            System.out.println("[Error] Wave hands with server failed! Too many tansfer
errors!");
            return;
        }
    }
}
```

分析：

1. 上述代码为四次挥手机制，此处规定每次传输服务结束之后由客户端作为主动方断开连接，发送给服务端`FIN`报文，服务端接受到对应挥手报文后返回确认报文`ACK`，紧接着也发送一次`FIN`，最终客户端返回最终`ACK`报文，两者之间的连接完全断开，服务端关闭相应服务，除去连接池内的对应客户端信息。



介绍完会话层的握手，挥手机制之后，接下来回到应用层调用的会话层操作启动函数，包括：`handleSendRequest`与`handleGetRequest`函数。

下述代码为客户端发送相应文件请求处理部分函数，可以看到首先利用`Java.IO`检查目标文件的存在与否，检查无误之后即开始握手函数的调用。

```
public void handleSendRequest(String path, String serverPath) throws IOException{
    System.out.printf("[Info] Ready to send file : %s\n", path);
    // Check path validation
    File file = new File(clientDir + path);
    // Exist check
    if(!file.exists()){
        System.out.printf("[Error] %s does not exist! Please check the path of the file you
want to send!\n", path);
        return;
    } else if(!file.isFile()){
        // File or folder check
        System.out.printf("[Error] %s is not a file!\n", path);
        return;
    }
    // File readability check
    try{
        InputStream stream = new FileInputStream(file);
    } catch(Exception e){
        System.out.printf("[Error] Cannot read file: %s!\n", path);
        return;
    }

    // Shake hands with target server, and then begin to send files
```

```
    if(!ShakeHandsWithServer(serverPath)){
        return;
    }
    // Transform and packets sending and receiving
    ....
    WaveHandsWithServer();
}
```

分析：握手过程结束之后，客户端根据相应的上传或下载请求，调用传输层的相关函数分别进行本地文件传输给服务端，或者本地等待接收服务端文件的函数执行。

- 传输层：`UDPClient.java`

  客户端传输层负责实现本应用最为关键的通信过程，对于客户端而言包括：

  1. 发送给服务端特定文件，首先结合自定义包标识:`SEND`发送初始服务启动报文，让服务端清楚请求类型为上传类型。将本地预备发送给服务端的文件分为多个数据包，依次加入缓冲区，采取流控制、拥塞控制机制进行缓冲区数据包的填充、发送给服务端，接收服务端对的确认报文`ACK`控制数据包发送约束，直到传输结束，给会话层返回成功标识。
  2. 等待接收服务端的特定文件，首先结合自定义包标识:`GET`发送初始服务启动报文，让服务端清楚请求类型为下载服务。启动本地循环接收文件数据包并存储，返回对应的`ACK`报文，直到传输结束，接收到报文`SUCCESS`，结束文件传输存储到本地，给会话层返回成功标识。

  客户端传输层上传文件代码解析：(部分代码)

```
// Init receive buffer
this.buffSize = file.length() / MAX_LENGTH + 1;
this.swnd = this.buffSize / 2;
// Begin sendind file, send begin flag(Feature)
UDPPacket packet0 = new UDPPacket(this.seq++);
String begin = "SEND " + path;
packet0.setBytes(begin.getBytes());
// Make packet checksum
packet0.calculateSum();
// Make window size deliver to server
packet0.setWinSize(this.swnd);
this.packetQueue.add(packet0);
```

分析：建立相应的缓冲区大小，同时给客户端发送上传服务启动报文，也可以看到客户端流控制的窗口大小约定`swnd`，伴随启动报文传递初始流控制窗口大小，随之将该报文加入缓冲区队列`packetQueue`中，等待传输。

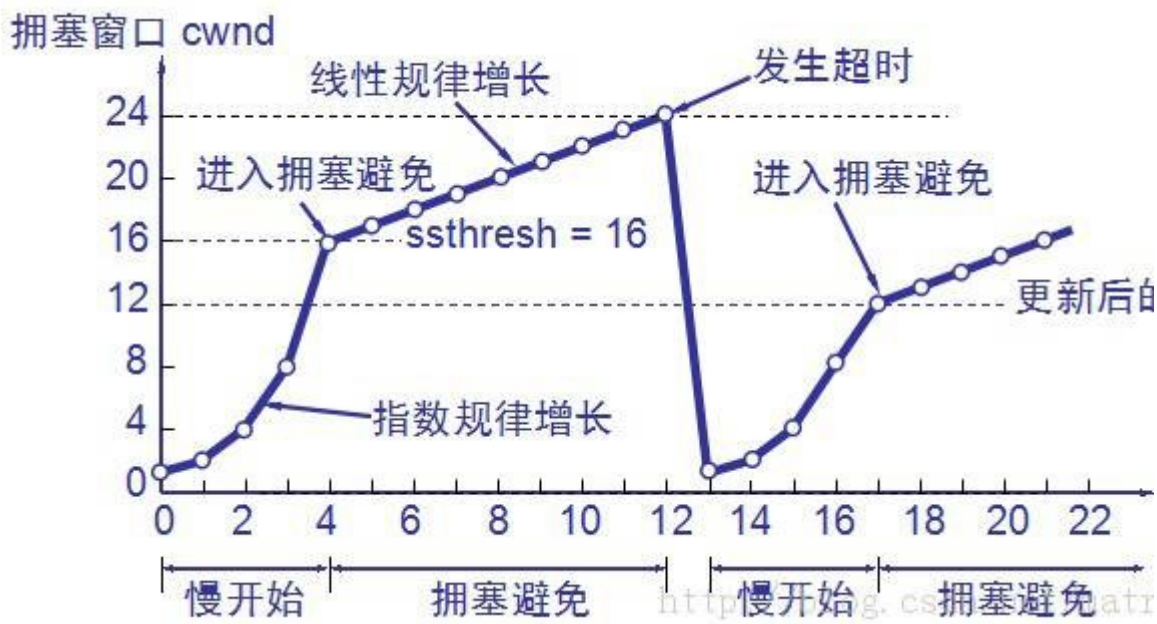
```
private void sendPacket() throws IOException{
    // Get Busy
    this.status = true;
    socket.setSoTimeout(2500);
    // Send all the packets in queue
    while(packetQueue.size() > 0){
        int errTime = 0;  // Record onr packet send fail times
        int sendPacketNum = 0;  // Record how many packets have client sent
        int currCongestNum = 1;  // Congestion control -- slow start's init number
```

```
        UDPPacket top = packetQueue.poll();
        // Dead loop to send one packet (detect failure)
        while(true){
            // Check floe control -- client stop sign
            try{
                if(sendPacketNum == swnd){
                Thread.sleep(2500);
            }
            } catch(Exception e){
                e.printStackTrace();
            }
            byte[] byteData = top.getPacketBytes();
            DatagramPacket outPacket = new DatagramPacket(byteData, byteData.length,
serverHost, serverPort);
            socket.send(outPacket);
            // Check return ack packet
            UDPPacket backPacket = receivePacket();
            if(backPacket != null && backPacket.isACK() && backPacket.checkSum()){
                sendPacketNum++;
                currCongestNum++;
                // Check flow control window size change sign
                if(backPacket.getWinSize() != 0){
                    sendPacketNum = 0;
                    this.swnd = backPacket.getWinSize();
                }
            // Check congestion control, whether slow start needs to incrase cwnd
            if(currCongestNum < this.ssthresh && currCongestNum % 2 == 0){
                // Keep in slow start
                this.cwnd = currCongestNum;
            }else {
                // Change into congestion avoidance
                this.cwnd++;
            }
            bar.updateBar(backPacket.getACK());
            break;
            }else {
                // Meet transfer error
                errTime++;
                // Congestion fast recovery
                this.ssthresh = this.cwnd / 2;
                this.cwnd = 1;
            }
            // Too many errors
            if(errTime > 5){
                System.out.println("[Error] Too many transfer errors, System shut down.");
                System.exit(2);
            }
        }
    }
    // Get Free
    this.status = false;
}
```

分析：上述代码为发送缓冲区内部的发送逻辑(部分)，其中结合了流控制与拥塞控制机制，同时该缓冲区不允许并行访问，设置忙碌状态 `status` 。缓冲区内部依据 `swnd` 与 `cwnd` 进行窗口数据包发送限制，以及丢包重传机制，每次往缓冲区队列里面拿取一个数据包进行发送，结合滑动窗口机制进行确认报文 `ACK` 的等待接收，以及拥塞控制机制的慢启动(cwnd初始为1)，拥塞避免(线性增长)，以及快恢复(阈值将为当前cwnd容量的一半，cwnd降为1)。

`Flow-Control`：

`Congestion-Control`：



客户端传输层下载文件代码解析：(部分代码)

```
while(true){
    // Receive packet file
    UDPPacket packet = receivePacket();
    if(packet == null) continue;
    // Check end or not

    byte[] data = packet.getBytes();
```

```
    if(data != null){
        String mess = new String(data);
        String[] res = mess.split(" ");
        if(res[0].equals("SUCCESS")){
        break;
    }
    bos.write(data, 0, data.length);
    bos.flush();
    // Send ACK
    UDPPacket ackPacket = new UDPPacket(this.seq++);
    ackPacket.setACK(packet.getSeq());
    ackPacket.calculateSum();
    // Increase buffer's size, check whether it's full
    receivePacketNum++;
    if(receivePacketNum == this.rwnd){
        receivePacketNum = 0;
        if(this.rwnd >= 1000){
            this.rwnd /= 2;
        }
        ackPacket.setWinSize(this.rwnd);
    }
    sendDataPacket(ackPacket);
    // Update bar
    bar.updateBar(this.seq);
}
```

分析：客户端下载文件的部分代码，此处建立循环接收服务端发来的数据包，返回对应的`ACK`报文，同时检查之前服务端发来的`swnd`作为自己的`rwnd`，每当接收窗口满足上限之后结合流控制将窗口大小减半，伴随对应的报文返回窗口缩小标识。数据包接收结束之后，进行新文件的创建。

3. 服务端具体实现逻辑

- 应用层：`LFTP_Server.java`

  由于服务端不需要与用户进行直接交互，所以服务端应用层不需要进行繁杂的界面设计，只需要交由会话层启动函数进行对远程主机连接服务的监听即可。

```
public static void main(String[] args){
    // Start the server
    server = new UDPServer();
    // Start server port listen
    System.out.println("[Info] LFTP Server start up at 127.0.0.1:8080...");
    try{
        server.listen();
    } catch(IOException e){
        e.printStackTrace();
    }
}
```

- 会话层：`UDPServer.java`

服务端会话层初始化函数，关键在于基于UDP的`DatagramSocket`对象，它负责将客户端的数据包进行转发，同时也用于接收客户端发送的数据包。

```
// Startup construct method
public UDPServer(){
    try {
    /* Init members */
    this.socket = new DatagramSocket(PORT_NUM);
    this.status = false;
    this.seq = 0;
    this.packetQueue = new LinkedList<>();
    this.map = new HashMap<String, Thread>();
    this.connectPool = new ArrayList<String>();
    this.connectNum = 0;
    } catch(SocketException e) {
    e.printStackTrace();
    }
}
```

服务端会话层通过`listen()`函数开启监听，等待远程客户端主机的访问，函数整体篇幅较长，此处分部分介绍逻辑。

```
// Loop to listen to client, use callback function to handle sending file to client
function
while(true){
    UDPPacket packet = receivePacket();
    if(packet != null){
        System.out.println("[Info]LFTP-Server receives packet from LFTP-Client....");
        // Read init mess
        this.clientIP = packet.getPacket().getAddress();
        this.clientPort = packet.getPacket().getPort();
        // First consider Shake hand and Wave hand packets
    if(packet.isSYN()){
        // Invoke shake hands function to process
        shakeHandswithClient(packet.getSYN());
        continue;
    }else if(packet.isFIN()){
        // Invoke wave hands function to process
        waveHandswithClient(packet.getSeq());
        continue;
    }
```

分析：监听函数开启循环，首先处理客户端的握手请求、挥手请求，通过调用基于多线程的握手、挥手函数进行处理，分别为`shakeHandswithClient()`与`waveHandswithClient()`。若成功握手建立连接，则把当前客户端的主机信息添加到服务端连接池中，方便之后的传输认证与信息获取。

```
// Open mupltiThread to deal with orders
switch(res[0]){
    case "BEGIN":

        // Open upload thread for server to receive files from server
```

```
            tt = new UploadThread(packet, res[1], this.seq, packet.getSeq(),
packet.getWinSize());
            map.put(clientIP.toString() + clientPort, tt);
            tt.start();
            st = (UploadThread) tt;
            st.startThread();
            break;
        case "SUCCESS":
            // Client upload success, end upload thread
            tt = map.get(clientIP.toString() + clientPort);
            st = (UploadThread) tt;
            st.endThread(this.seq, packet.getSeq());
            break;
        case "GET":
            // Check client download request, check file existion
            serverFile = new File(serverDir + res[1]);
            if(!serverFile.exists()){
                // Client bad request
                sendBadResponse(res[1]);
            }else {
                // Open download thread for server to send files to client
                tt = new DownloadThread(packet, res[1], this.seq, packet.getSeq());
                map.put(clientIP.toString() + clientPort, tt);
                dt = (DownloadThread) tt;
                dt.startThread();
                dt.start();
            }
            break;
        default:
            // Client's uploading packets, handle and send to the speical thread
            tt = map.get(clientIP.toString() + clientPort);
            st = (UploadThread) tt;
            st.receivePacket(packet.getBytes(), this.seq, packet.getSeq());
            st.run();
    }
```

分析：会话层接收不同的报文信息，转交给传输层的对应传输进程，此处的传输进程与当前发送数据包的主机关联关系，此处采用`Java Map`数据结构进行映射，用于传输层多线程的正常运行。

- 传输层：`UDPPacket.java`，`UploadThread.java`，`DownloadThread.java`

  此处介绍本应用中使用范围最广的类`UDPPacket`，为自定义的数据包基本单位类，重点在于其可序列化，不仅可以传递多于普通`Java DatagramPacket`的数据含量(包括多处用到的确认标识符，发送序号，数据包内容等等)，还能够正常序列化，保证双端能够正常解读包内容。

```
public class UDPPacket implements Serializable{
    // Define a udp packet
    private transient DatagramPacket packet;
    private String packetName;  // Name of file this packet belongs to
    private int seq;  // Sequence number
    private int ack;  // Acknowledge number
    private int syn;  // Shake hands number

    private int fin;  // Wave hands number
```

```
    private long winSize; // Transfer window size between client and server, make flow
control
    private boolean synFlag;  // Syn flag bit
    private boolean ackFlag;  // Ack flag bit
    private boolean finFlag;  // Fin flag bit
    private byte[] data;  // store packet content
    private String checksum; // check sum
```

数据包传输过程中有可能存在错误，为了保证底层信道的可靠性，此处定义了`checkSum`的计算方法，在每次数据包发送之前进行校验和的计算，每当对方拿到了数据包，首先就检验校验和是否正确。

```
// Calculate the UDPPacket's checksum with its header infos and data
private String getSum(){
    String resultHex;
    int result = 0;
    // Step 1. Calculate header's sum
    result += seq + ack + syn + fin;
    // Step 2. Calculate data's sum
    if(data != null){
        for(int i=0; i<data.length; i++){
            // Make data byte signed integer
            int value = data[i] & 0xff;
            result += value;
        }
    }
    // Step 3. Make Hex string
    return Integer.toHexString(result);
}
```

除此之外，基本数据包类`UDPPacket`内部仍有很多基本变量`get()、set()`函数，利于对包内信息的获取检验。

服务端传输层的核心在于对上传文件线程与下载文件线程的逻辑实现，此处的线程内部文件发送与接收同样基于流控制与拥塞控制机制，核心代码如下：(部分代码)

```
// UploadThread.java
if(this.content != null){
    // Write content
    bos.write(content, 0, content.length);
    bos.flush();
    content = null;
    receivePacketNum++;
    /* Start flow control, notify client not to send anymore,
    *   Cause it reaches the window size, and half divide the window size
    *   to make speed lower,
    */
    if(receivePacketNum == rwnd){
        receivePacketNum = 0;
        if(rwnd >= 1000){

            rwnd = rwnd / 2;
```

```
            }
            sendACKPacket(this.seq, this.ack, rwnd);
        }else {
            // Else stay normal receiving state
            sendACKPacket(this.seq, this.ack, 0);
    }
```

分析：用于接收客户端发送文件的传输线程`UploadThread`，此处结合了流控制窗口，动态改变其滑动窗口的大小，并且每次都从会话层收取到的包缓冲区取出相对应的主机发来的数据包，同时发送确认报文`ACK`。

```
readPos = accessFile.read(buffer, 0, buffer.length);
if(readPos != -1){
    packet.setBytes(buffer);
}else {
    String end = "SUCCESS";
    packet.setBytes(end.getBytes());
    System.out.printf("[Info]LFTP-Server successfully sent large file: %s\n", fileName);
    accessFile.close();
}
packet.calculateSum();
byte[] data = packet.getPacketBytes();
DatagramPacket outPacket = new DatagramPacket(data, data.length, this.clientIP,
this.clientPort);
socket.send(outPacket);
sendPacketNum++;
```

分析：用于发送给客户端对应文件的传输线程`DownloadThread`，分别根据客户端流控制窗口进行发送与`ACK`确认报文接收，依次移动当前文件指针进行分段传输数据包。

三、实验程序用法

1. 如要进行实验程序的测试，则进入到代码文件夹`src`内部，如要启用客户端程序，则在命令行下执行命令：

```
$ javac LFTP_Client.java
$ java  LFTP_Client
```

如要启用服务端程序，则执行：

```
$ javac LFTP_Server.java
$ java  LFTP_Server.java
```

2. 本应用的用于测试的文件存放在`src/lftp/Files`路径文件夹内，内部包含`ClientFiles`与`ServerFiles`，分别存放当前客户端文件与服务端文件。若想查看是否传输成功，不仅可以查看客户端的打印信息，还可以进入专门存放对应端文件夹查看文件完整性。

3. 客户端程序默认运行在测试主机的`8082`端口，服务端程序默认允许在测试主机的`8080`端口，若需要进行客户端`多连接`的传输验证，则需要使用者手动更改另一个客户端程序的监听端口，防止端口占用情况发生。

四、实验应用测试

1. 本地测试(单客户端测试)
   - 首先进行客户端、服务端程序的执行

```
PS E:\Github\Computer_Network\LFTP\16340023+陈明亮\src> java LFTP_Client
-------------- Welcome to LFTP --------------
Enter commands to start using LFTP Client, "help" for details
LFTP-Client$: help
Available commands:
        LFTP lsend 127.0.0.1:8080 "fileName" -- send file to server
        LFTP lget 127.0.0.1:8080 "fileName" -- get file from server
        exit -- leave LFTP-Client
        help -- get help details
LFTP-Client$:
```

     客户端可以执行上传、下载命令，同时也能查看帮助，也能按下`exit`退出程序。

```
Lenovo@LAPTOP-BBTT6KDL MINGW64 /e/Github/Computer_Network/LFTP/16340023+陈明亮/src (master)
$ java LFTP_Server
[Info] LFTP Server start up at 127.0.0.1:8080...
```

   - 客户端输入发送请求命令，与服务端进行握手连接，同时开始文件传输

```
LFTP-Client$: LFTP lsend 127.0.0.1:8080 gakki1.mp4
[Info] Client is ready to send file to server....
[Info] Ready to send file : gakki1.mp4
[Info] Successfully establish connections with Sever: 127.0.0.1, serverPort: 8080
[Info]Sending file to LFTP-Server: [--------->                    ] 30%
```

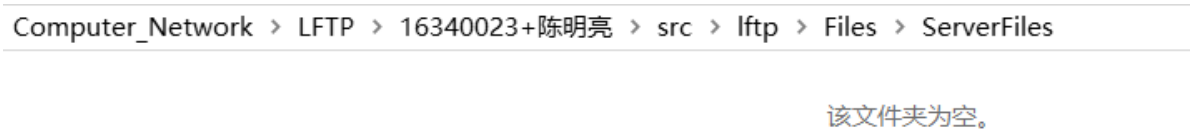
   - 客户端发送文件结束，查看当前打印信息

```
LFTP-Client$: LFTP lsend 127.0.0.1:8080 gakki1.mp4
[Info] Client is ready to send file to server....
[Info] Ready to send file : gakki1.mp4
[Info] Successfully establish connections with Sever: 127.0.0.1, serverPort: 8080
[Info]Sending file to LFTP-Server: [---------------------------->] 100%
[Info]LFTP-Client successfully sends large file: gakki1.mp4, file Size: 541.02 MB, send Time: 22.37 sec
[Info] Successfully quit the connection with target server....
LFTP-Client$:
```

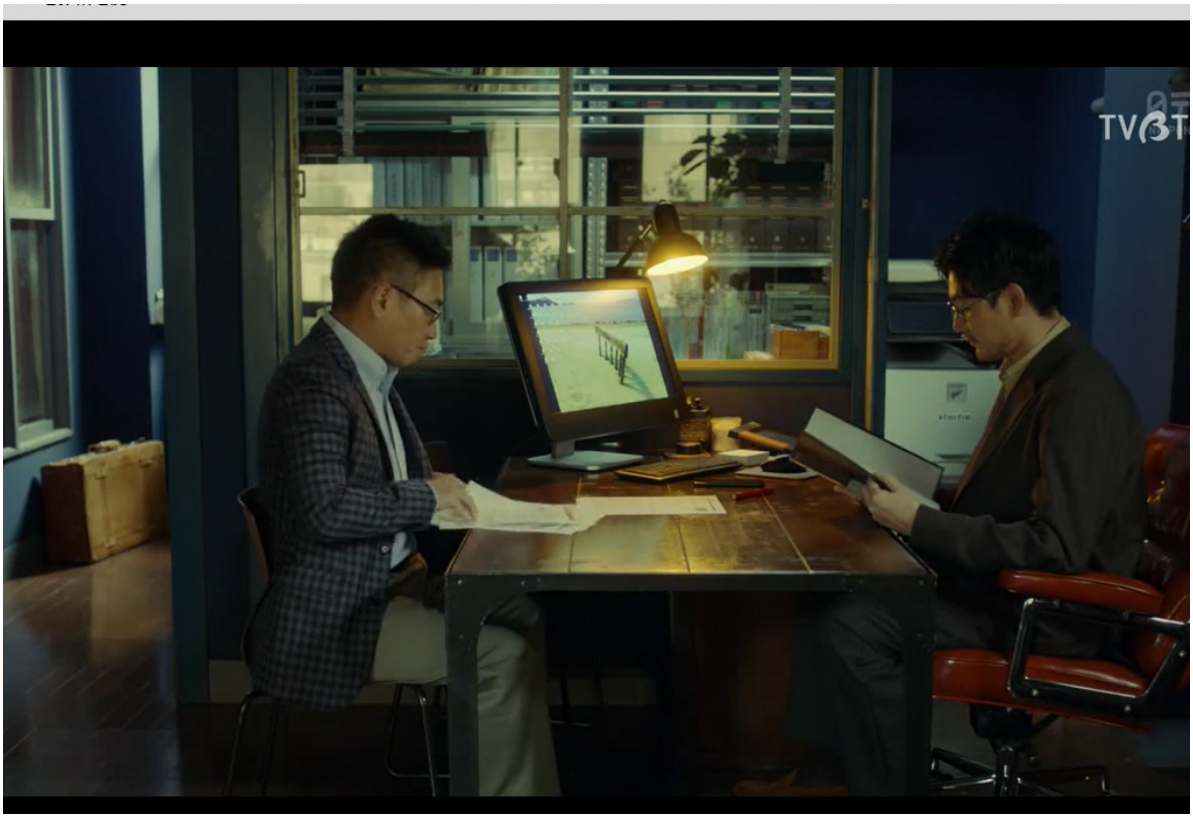
```
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server successfully received large file: gakki1.mp4
[Info]LFTP-Server receives packet from LFTP-Client....
[Info] Server successfully delete connection with Client: /127.0.0.1:8082
```

     可以看到打印信息说明传输成功，挥手过程成功。

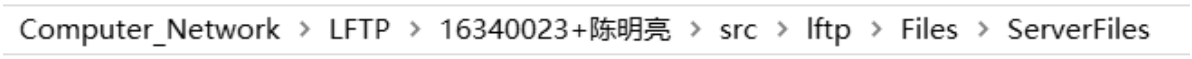
   - 比较传输前后服务端文件夹内部的文件创建是否成功

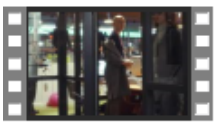


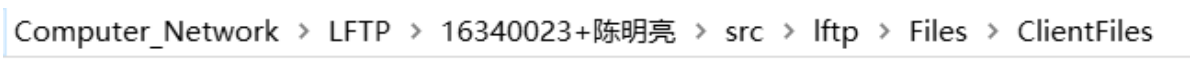
- 查看传输文件完整性



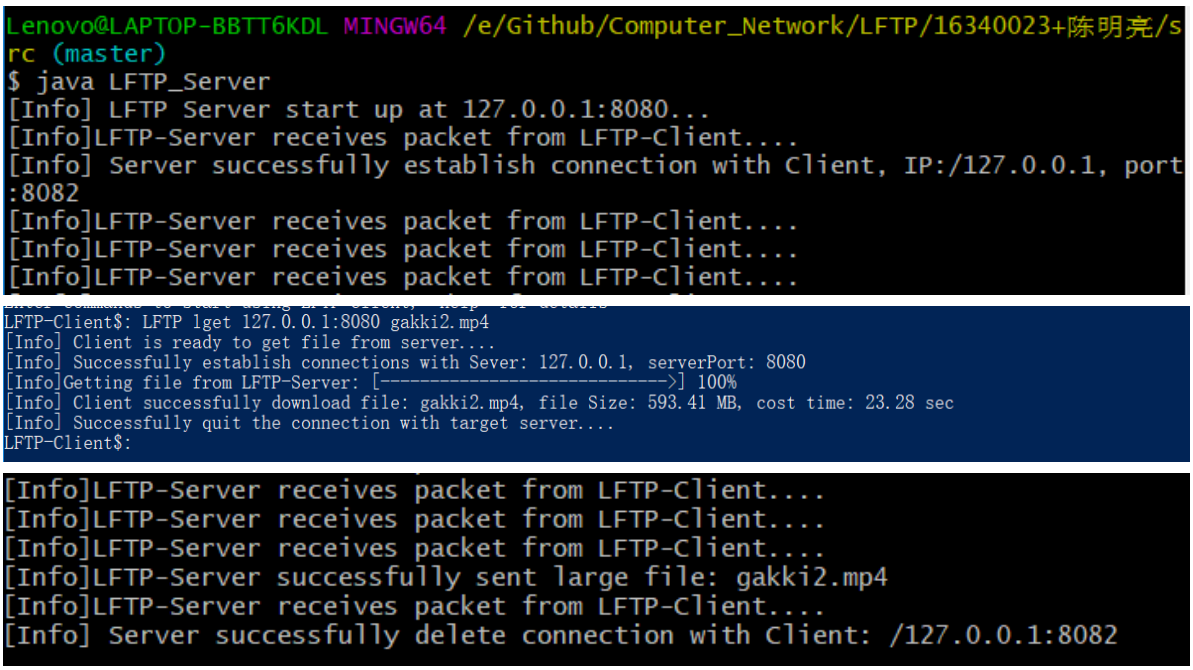
- 紧接着进行客户端从服务端下载对应文件的命令测试，首先把新文件`gakki2.mp4`手动放置到服务端文件夹内部，查看当前客户端服务端文件夹内容



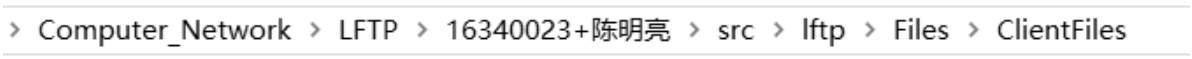




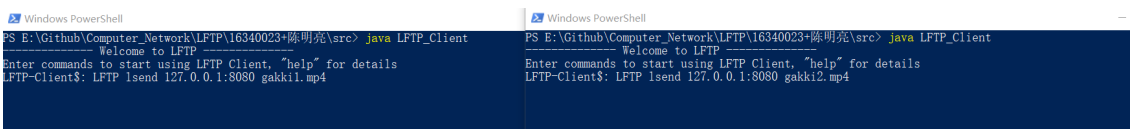
- 客户端发出下载请求，完成下载，查看此时双端的打印信息。

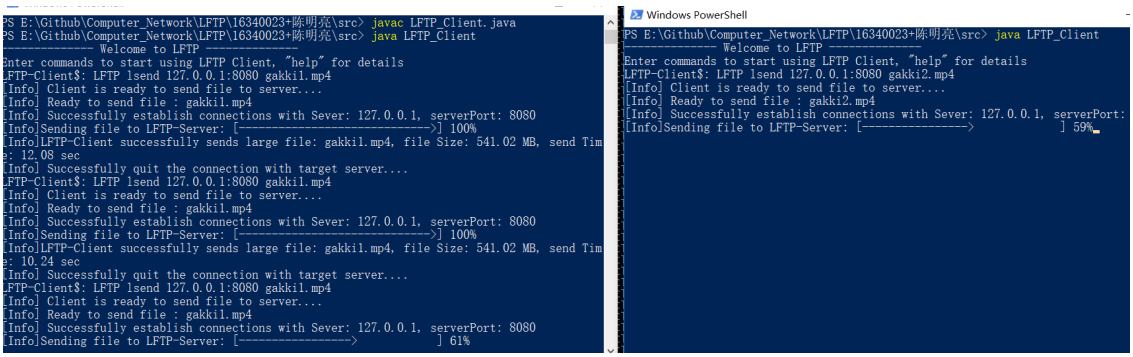


- 上述打印信息可以看出客户端下载`gakki2.mp4`成功，同时握手、挥手过程均正常进行，接下来验证传输文件的完整性。

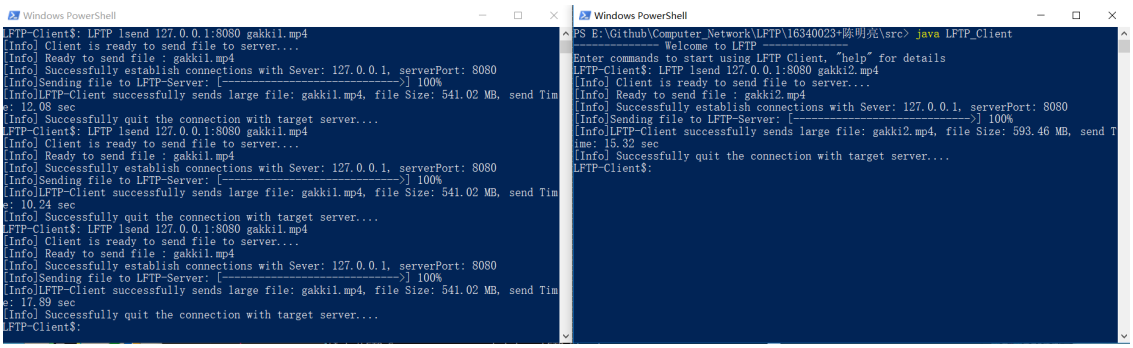


2. 本地测试(多客户端测试)
    - 由于单客户端测试了对应的上传、下载功能均正常，此处对多客户端测试的重点就放在服务端能够同时服务多个客户端的能力验证上
        1. 两个客户端同时进行文件上传测试，可以看到对应进度条

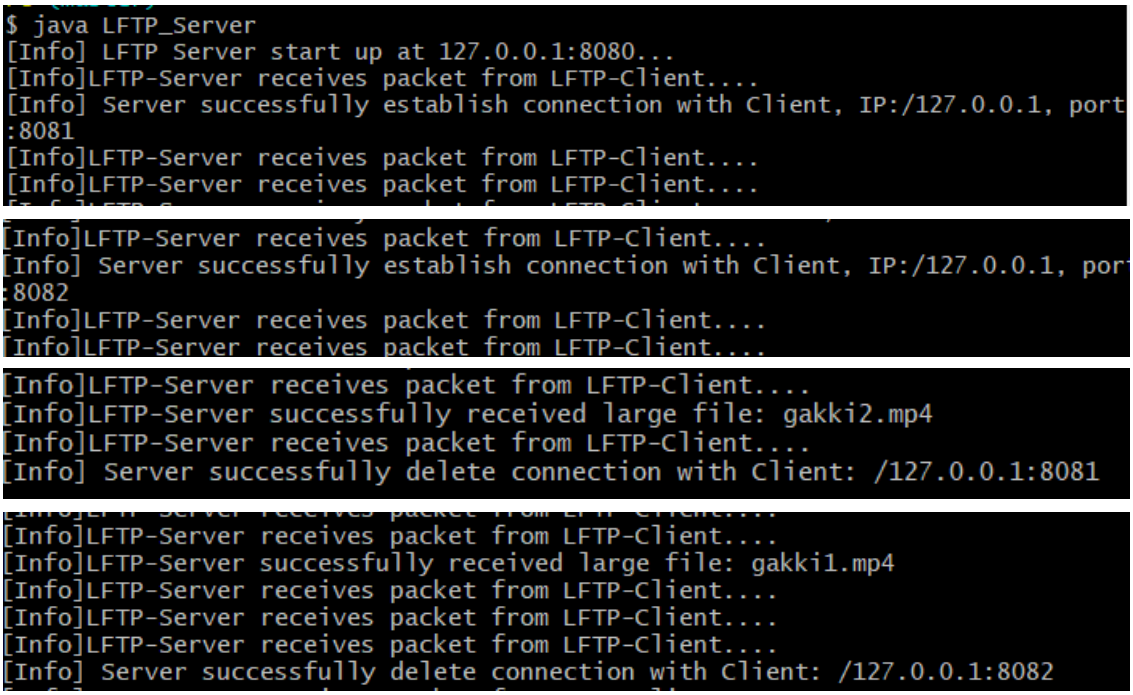




2. 两个客户端成功上传文件之后信息



3. 服务端分别于两个客户端建立连接，传输文件成功，挥手过程打印信息



3. 远程主机之间进行传输测试

   - 使用同学的PC进行远程主机之间的传输测试，断开校园网有线连接，采用无线连接`SYSU-SECURE`更加能够体现差网络传输测试
   - 首先分别打印客户端与服务端`IP`地址，并且分别进行初始化连接。

     客户端IP:

```
无线局域网适配器 WLAN:

   连接特定的 DNS 后缀 . . . . . : lan
   本地链接 IPv6 地址. . . . . : fe80::9c12:267:285b:2e01%15
   IPv4 地址 . . . . . : 192.168.199.119
   子网掩码 . . . . . : 255.255.255.0
   默认网关. . . . . : 192.168.199.1
```

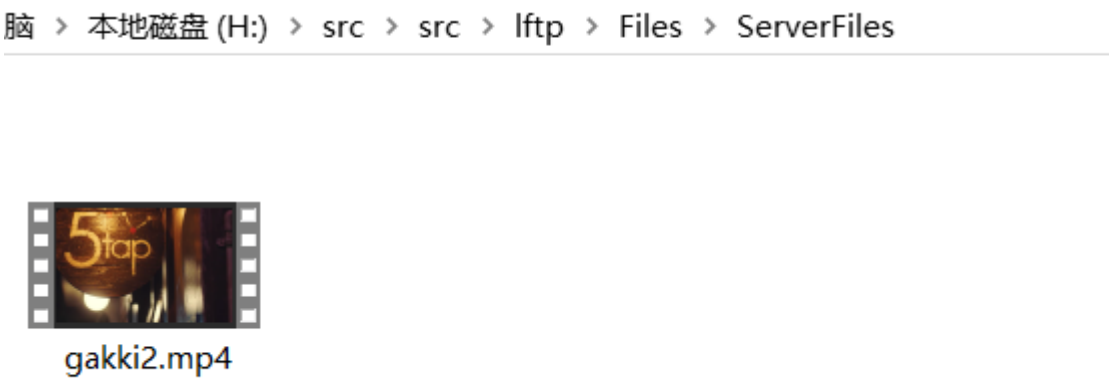
服务端IP:

```
无线局域网适配器 WLAN:

   连接特定的 DNS 后缀 . . . . . : lan
   本地链接 IPv6 地址. . . . . : fe80::f5b0:6889:23c7:1c40%22
   IPv4 地址 . . . . . : 192.168.199.118
   子网掩码 . . . . . : 255.255.255.0
   默认网关. . . . . : 192.168.199.1
```

当前客户端文件夹内容:

当前服务端文件夹内容:



客户端发起连接请求，并且开始传输文件:

```
-------------- Welcome to LFTP --------------
Enter commands to start using LFTP Client, "help" for details
LFTP-Client$: LFTP lsend 192.168.199.118:8080 gakki1.mp4
[Info] Client is ready to send file to server....
[Info] Ready to send file : gakki1.mp4
[Info] Successfully establish connections with Sever: 192.168.199.118, serverPort: 8080
[Info]Sending file to LFTP-Server: [-->                              ] 10%
```

服务端收到握手请求建立连接:

```
$ java LFTP_Server
[Info] LFTP Server start up at 127.0.0.1:8080...
[Info]LFTP-Server receives packet from LFTP-Client....
[Info] Server successfully establish connection with Client, IP:/192.168.199.119
, port:8082
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
```

客户端传输过程中：

```
--------------- Welcome to LFTP ---------------
Enter commands to start using LFTP Client, "help" for details
LFTP-Client$: LFTP lsend 192.168.199.118:8080 gakki1.mp4
[Info] Client is ready to send file to server....
[Info] Ready to send file : gakki1.mp4
[Info] Successfully establish connections with Sever: 192.168.199.118, serverPort: 8080
[Info]Sending file to LFTP-Server: [--------------------------->  ] 95%
```

客户端传输完成：

```
LFTP-Client$: LFTP lsend 192.168.199.118:8080 gakki1.mp4
[Info] Client is ready to send file to server....
[Info] Ready to send file : gakki1.mp4
[Info] Successfully establish connections with Sever: 192.168.199.118, serverPort: 8080
[Info]Sending file to LFTP-Server: [------------------------------>] 100%
[Info]LFTP-Client successfully sends large file: gakki1.mp4, file Size: 541.02 MB, send Time: 230.62 sec
[Info] Successfully quit the connection with target server....
```

服务端打印成功接收信息，以及挥手信息：

```
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server receives packet from LFTP-Client....
[Info]LFTP-Server successfully received large file: gakki1.mp4
[Info]LFTP-Server receives packet from LFTP-Client....
[Info] Server successfully delete connection with Client: /192.168.199.119:8082
```

五、实验感想

1. 本次实验为计算机网络理论课程其中项目，是要求学生结合课本所学，结合实际编程能力完成一款基于底层`UDP`，学生自己实现传输协议必须与`TCP`100%相同的大文件传输应用，其中需要熟练掌握`TCP`的工作原理，更需要有很好的编程水平，成功实现传输大文件，打印相关信息的一款网络通信应用。
2. 总体来说本次实验的难度还是挺大的(本人为单人一组)，不论是客户端、服务端的分层代码编写，还是之后的测试环节，需要花费的精力与时间挺多，当然收获到的知识与经验也随之增加。