Computer Networks Midterm Project

项目要求：

Please write a network application, LFTP, to support large file transfer between two computers in the Internet.

1. A student can finish the project by himself or herself. Two students may form a group to work on the project too. If two students work together, the tasks to be finished should be assigned to each one by almost 50%;

2. The technical requirements are:

(1) Please choose one of following programing languages: C, C++, Java, Python;

(2) LFTP should use a client-server service model;

(3) LFTP must include a client side program and a server side program; Client side program can not only send a large file to the server but also download a file from the server.

Sending file should use the following format：

*LFTP lsend myserver mylargefile*

Getting file should use the following format

*LFTP lget myserver mylargefile*

The parameter myserver can be a url address or an IP address.

(4) LFTP should use UDP as the transport layer protocol.

(5) LFTP must realize 100% reliability as TCP;

(6) LFTP must implement flow control function similar as TCP;

(7) LFTP must implement congestion control function similar as TCP;

(8) LFTP server side must be able to support multiple clients at the same time;

(9) LFTP should provide meaningful debug information when programs are executed.

**项目设计：**

编程语言：Java

**总体设计简述：**

整个项目分为服务端和客户端两个部分，其中，服务端实现后将放置于云服务器，而客户端则留在本地操作，如果需要，可为客户端设置环境变量，使得其可像一般控制窗口命令般执行。

服务端中，允许多用户访问，上传文件时，通过用户的 IP 地址为每个用户维持一个目录来实现文件之间的互不干扰，但是下载文件时不做此限制。上传文件时，所有用户使用同一个端口与服务器交互；下载文件时，为方便实现可靠数据传输、流量控制与拥塞控制等，不同用户使用不同端口进行交互，每个用户使用一个线程对其进行服务，且每个线程中有两个子线程负责接收数据和给客户端进行反馈。

客户端中，在上传文件的时候，为实现可靠数据传输、流量控制与拥塞控制等功能，使用两个线程，一个线程负责上传数据，另一个线程负责接收服务端发送回来的反馈。而在下载文件的时候，则只需要一个线程，将接收到的报文段放入缓存，将有序的报文段写入硬盘，同时，每次接收一个报文段，对其处理完毕后，向服务端发送反馈信息。

**1. 发送文件时，客户端的设计：**

TimeoutInterval 变量（即预估的超时时长）的计算：

初始的时候，TimeoutInterval 设置为 1 秒，涉及计算的变量初始化如下：

```
private int TimeoutInterval = 1000;
private int EstimatedRTT = TimeoutInterval;
private int SampleRTT = 0;
private int DevRTT;
```

SampleRTT 为样本报文段的往返时间，在任意时刻，只对一个已经发送且尚未被确认的报文段估计 SampleTRR。实现方法为在使用一个 Map 链表，每个报文段第一次发送时，即不包括重传的报文段，将报文段的发送时间及报文段序号存储在 Map 中，在每次接收到非重复的 ACK 报文段的时候，从 Map 中找出对应 ACK 序号回复的报文段，则可计算出该报文段的估计往返时间，并将该报文段的记录中 Map 链表中删除，公式如下：

```
SampleRTT = (int) (dealSendTime(1, LastByteAcked - 1).getTime() - new Date().getTime());
```

dealSendTime 方法为线程同步函数，用于更改 Map 链表。

EstimatedRTT 为估计的下一次往返时间，SampleRTT 发生更新的时候便计算一次，公式为：

```
EstimatedRTT = (int) (0.875 * EstimatedRTT + 0.125 * SampleRTT);
```

DevRTT 则是 RTT 的偏差，计算公式如下：

$$DevRTT = (1-\beta) \cdot DevRTT + \beta \cdot |SampleRTT - EstimatedRTT|$$

本次项目中，$\beta$ 取值为 0.25，故在此次项目中的计算公式如下：

```
DevRTT = (int) (0.75*DevRTT + 0.25*Math.abs(SampleRTT - EstimatedRTT));
```

当 SampleRTT 发生变动时计算一次。

TimeoutInterval 为重传时间间隔，初始值为 1 秒，后面每当 SampleRTT 发生更新时，采用如下公式进行计算更新：

```
TimeoutInterval = EstimatedRTT + 4*DevRTT;
```

并在更新后重新设置超时的计时，如下：

```
if(TimeoutInterval > 0) dataSocket.setSoTimeout(TimeoutInterval);
```

此外，每当发生一次超时的时候，将 TimeoutInterval 设置为原先的两倍：

```
TimeoutInterval = 2 * TimeoutInterval;
```

**可靠数据传输：**

可靠数据传输依靠两个机制实现，一个机制是快速重传机制，当接收端接收到三次重复的 ACK 时，则将最新 ACK 的这个报文段重新发送；另一个机制是超时重传机制，若忽略其他机制时，超时重传机制的时间重置和超时后动作如下：

```
loop (forever) {
    switch(event)

        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment with
                smallest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase=y
                if (there are currently any not-yet-acknowledged segments)
                    start timer
                }
            break;

    } /* forever 循环结束 */
```

当出现超时的时候，将最新的接收端 ACK 的报文段重新发送，即可实现将最小的未被 ACK 的报文段的重传。

**超时重传机制：**

对于每个 ACK 报文段，如果该报文段 ACK 的序号是比上一次 ACK 的序号大，设 ACK 的序号为 y，则可证明在序号 y 之前的报文段均已被正确接收，此时重新启动计时器。对于每个非重传的报文段，发送的时候，重新启动计时器。在超时的时候，根据接收端的发送 ACK 的序号为缺失的最小的失序序号这一机制可知，此时需要重传的即为最新接收到的 ACK 序号，故对该报文段进行重传，并且重新启动计时器。由于计时器需要同时在发送数据的线程和接收数据的线程中使用，且两个线程均需要对计时器进行更改，故计时器的更改使用同步块来处理，如下：

```
private synchronized void setTimer() {
    timer = true;
    beginTimer = new Date();
}
```

其中，beginTimer 为记录的计时开始时的时间，可通过计算当前时间与计时开始时的时间差来确定是否发生超时，如下：

```
if(timer && new Date().getTime() - beginTimer.getTime() >= TimeoutInterval)
```

**快速重传机制：**

当发送端连续接收到三个或以上相同的 ACK 序号的时候，发送端会立刻重发所 ACK 的报文段。通过维护一个 AckTimes 变量来实现这一机制，由于 AckTimes 变量在接收 ACK 的线程与发送数据的线程均需要修改，故使用同步块对其进行操作：

```
//0 to set, 1 to add
private synchronized void addOrSetAT(int mode) {
    if(mode == 0) {
        AckTimes = 1;
    } else if(mode == 1) {
        AckTimes++;
    } else {
        System.err.println("Wrong parameter in addOrSetAT methon.");
    }
}
```

当 AckTimes 的值大于或等于三次时，则马上重发 ACK 的报文段，并重新设置 AckTimes 的值为 1：

```
if(AckTimes >= 3) {
    data = readFrameFile(randomFile, LastByteAcked);
    sendMsg.setId(LastByteAcked);
    sendMsg.setData(data);
    sendMsg.setLen(data.length);
    byte[] sendData = sendMsg.toByte();
    DatagramPacket packet = new DatagramPacket(sendData, sendData.length, address,
    dataSocket.send(packet);

    addOrSetAT(0);
```

而每次接收到一个重复的 ACK 时，则对 AckTimes 变量递增 1，而且，因为发送端这边无需理会接收到的 ACK 序号小于前面接收到的 ACK 序号这一情况，所以，只需当此次接收的 ACK 序号与上一次的 ACK 序号相等时，即可知道是冗余的 ACK，便对 AckTimes 变量增加计数：

```
if(ack.getRecId() == LastByteAcked) {
    addOrSetAT(1);
    if(cwnd_state == 2) {
        setCwnd(1);
    }
```

此处 LastByteAcked 变量为接收端确认的最新有序的 ACK 序号。
每次接收到一个新的 ACK 的时候，则对 AckTimes 重新设置为 1，表示这是第一次接收到该 ACK，而根据 ACK 的传输机制，当最新接收到的 ACK 的序号比 LastByteAcked 大，即可知道该 ACK 序号是第一次传来，如下：

```
else if(ack.getRecId() > LastByteAcked) {
  LastByteAcked = ack.getRecId();
  addOrSetAT(0);
```

**流量控制：**

维护流量控制的变量包括如下几个：

```
private int rwnd;
private int LastByteAcked;
private int LastByteSend;
```

其中，rwnd 为接收窗口长度，该变量记录了接收端的缓冲区中可剩余使用的窗口长度，通过接收端的返回信息中得到：

```
rwnd = ack.getRecWin();
```

LastByteAcked 变量为接收端发送回来的已确认接收有序的序号，由于 ACK 回来的序号为接收端下一个希望接收到的报文段序号，故实际上，LastByteAcked 的数值比接收端所接收到的最后一个有序报文段的值要大一个单位。

LastByteSend 为发送端已发送的最后一个报文段序号。在这次项目中，为了方便维护，实际上，LastByteSend 为已发送的最后一个报文段序号加 1，由此对应 LastByteAcked。当无需重传时，满足下面公式，即可发送新的报文段，发送完后，LastByteSend 自动增加一个单位。

$$\text{LastByteSent} - \text{LastByteAcked} \leqslant \text{rwnd}$$

在项目中，同时考虑拥塞控制窗口 cwnd，则如下：

```
if(LastByteSend - LastByteAcked <= Math.min(rwnd, cwnd)) {
```

**拥塞控制：**

在 TCP 中，需要维护一个拥塞窗口，使得下面式子成立时，才继续发送报文段：

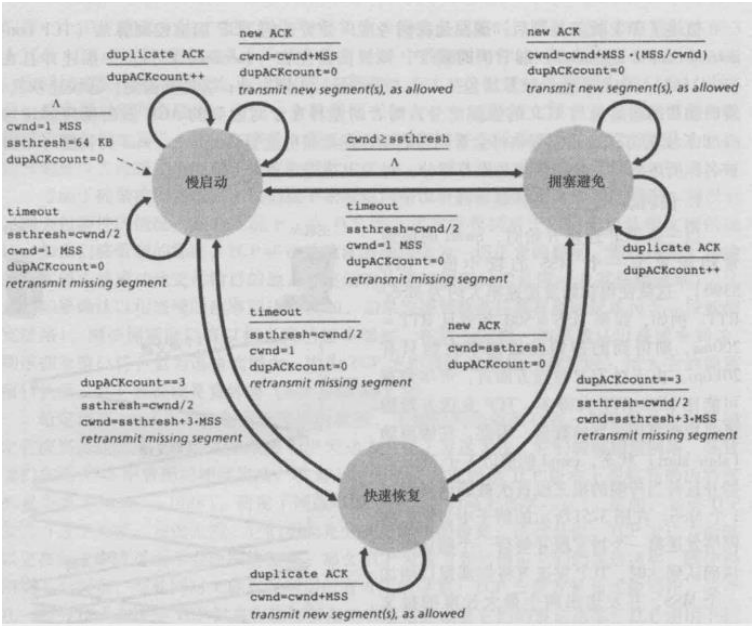
$$\text{LastByteSent} - \text{LastByteAcked} \leqslant \min\{\text{cwnd}, \text{rwnd}\}$$

实现拥塞控制所涉及的变量为：

```
private int TimeoutInterval = 1000;
private int EstimatedRTT = TimeoutInterval;
private int SampleRTT = 0;
private int DevRTT;
private final int MSS = 1024;
private int cwnd = 1;
private int ssthresh = 64;
private int cwnd_state = 0; //0为慢起动阶段，1为拥塞避免阶段，2为快速恢复阶段

private final int WINDOW_SIZE = 20;
```

cwnd 变量为拥塞窗口长度，初始值为 1。cwnd_state 为拥塞控制得状态。具体的状态转换图如下：



为了防止多线程的干扰，对 cwnd 及状态转换时，使用同步块来对两个变量进行维护，如下：

```
//设置状态，0为慢起动阶段，1为拥塞避免阶段，2为快速恢复阶段
private synchronized void setCwndState(int state) {
    cwnd_state = state;
}

/**
 * 0为= 1，1为+= 1，2为*= 2，3为ssthresh + 3，4为= ssthresh
 * @param mode
 */
private synchronized void setCwnd(int mode) {
    switch (mode) {
    case 0:
        cwnd = 1;
        break;
    case 1:
        cwnd += 1;
        break;
    case 2:
        cwnd *= 2;
        break;
    case 3:
        cwnd = ssthresh + 3;
        break;
    case 4:
        cwnd = ssthresh;
        break;
    default:
        break;
    }
}
```

项目具体实现中，根据状态转换图进行设计实现。

在多次 ACK 导致重传的时候，如果状态为慢启动或者拥塞避免时，则将状态改为快速恢复状态，并更改 ssthresh 的值为 cwnd 的一般，同时将 cwnd 的值改为 ssthresh + 3：

```
if(cwnd_state == 0 || cwnd_state == 1) {
    ssthresh = cwnd > 1 ? cwnd / 2 : 1;
    setCwnd(3);
    setCwndState(2);
}
```

在超时重传的时候，如果状态为慢启动，则将 ssthresh 降为当前 cwnd 的一般，然后将 cwnd 的值置 1；如果状态为快速恢复或拥塞避免，则将 ssthresh 降为当前 cwnd 的一般，然后将 cwnd 的值置 1，同时将状态设置为慢启动状态。

```
if(cwnd_state == 0) {
    ssthresh = cwnd > 1 ? cwnd / 2 : 1;
    setCwnd(0);
} else {
    ssthresh = cwnd > 1 ? cwnd / 2 : 1;
    setCwnd(0);
    setCwndState(0);
}
```

当接收到重复冗余的 ACK 时，如果状态为快速恢复，则将 cwnd 的值增加 1，其他状态则不对 cwnd 进行操作。

```
if(cwnd_state == 2) {
    setCwnd(1);
}
```

当接收到一个新的 ACK 时，如果状态为慢启动状态，则将 cwnd 的值设为原来的两倍，而当 cwnd 的值大于或等于 ssthresh 的值时，则进入拥塞避免状态；当状态为拥塞避免得时候，cwnd 的值加 1；当状态为快速恢复状态的时候，直接设置 cwnd 的值为 ssthresh，同时将状态转换到拥塞避免状态。

```
if(cwnd_state == 0) {
    setCwnd(2);
    if(cwnd >= ssthresh) {
        setCwndState(1);
    }
} else if(cwnd_state == 1){
    setCwnd(1);
} else if(cwnd_state == 2) {
    setCwnd(4);
    setCwndState(1);
}
```

**文件读取的设计：**

文件每次只读取一个数据包大小，使用随机读取机制，每次根据要传送的报文段序号来确定读取文件的内容。每次传送一个文件时，报文段序号的初始值设为 0，每个报文段中所含文件信息部分（即包含头部信息）带下为 MSS，此处，MSS 的值设定为 1024Byte，文件读取的方法如下：

```
public byte[] readFrameFile(RandomAccessFile randomFile, int number) throws IOException {
    long fileLength = randomFile.length();
    long beginIndex = number * MSS;
    if(beginIndex >= fileLength)
        return null;
    int byteSize = (int) (fileLength - beginIndex >= MSS ? MSS : fileLength - beginIndex);
    randomFile.seek(beginIndex);
    byte[] bytes = new byte[byteSize];

    if(randomFile.read(bytes) == -1) {
        throw new IOException("Wrong in UDPClient;readFrameFile");
    }
    return bytes;
}
```

传输与接收设计：

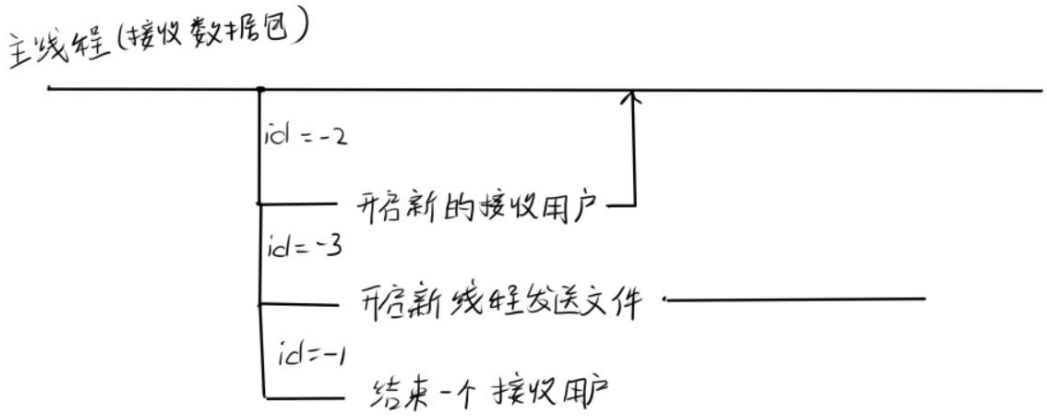
考虑到发送端需要一边发送文件，一边接收从接收端发回来的 ACK 报文段，使用两个线程进行文件传输，SendData 类的线程负责发送报文段，ReceiveACK 类的线程负责接收 ACK 报文段。在发送报文段中，维护一个 isAgain 布尔变量，如果该变量为真，则此时为重传报文段，如果为假，则下一个报文段为第一次发送。同时规定，重传的报文段不受发送限制。为了应对文件已发送完，但是接收端还有些报文段未接受或出现丢失报文段的情况，维护一个 hasAckLastByte 变量，用来记录接收端是否已经确认接收到所有文件。而接收端只能 ACK 下一个其期待收到的序号，故维护一个 EndOfFile 变量，记录该发送文件的最后一个报文段的序号，当接收端 ACK 回来的序号与 EndOfFile 相等时，即可说明接收端已经接收到最后一个文件，则此时发送一个序号为-1 的报文段，告诉接收端文件已发送完毕，并停止发送。在发送完文件且接收端未接收完文件期间，为防止出现报文段丢失，依旧维持发送报文段这一

线程，但是不再主动发送新的报文段，只有当检测到三次冗余 ACK 及出现超时的时候，再进行对最新 ACK 的报文段进行重传。

对于 ReceiveACK 线程，一直接受接收端发送过来的 ACK，并更新 LastByteACK 值，当接收端接收到-1 报文段的时候，接收端会返回一个 ACK -2 报文段，ReceiveACK 线程接收到该报文段的时候，则结束该线程。

**2. 上传文件时，服务端的设计：**

服务端的线程原理大概如同：




都是通过主线程来接受各种消息，有可能是要上传，有可能是要下载，有可能是要结束上传，也有可能就是普通的数据包：

```
while(true) {
    try {
        System.out.println("waiting");
        byte[] message = new byte[MAX_LEN];
        DatagramPacket recPacket = new DatagramPacket(message, message.length);
        mySocket.receive(recPacket);
        SendMsg fromClientMsg = new SendMsg();
        fromClientMsg.toMsg(message);
        int recId = fromClientMsg.getId();
        System.out.println("recId: " + recId);
        InetAddress clientIP = recPacket.getAddress();
        int clientPort = recPacket.getPort();
        switch(recId) {
            case -1:
                endaReceive(clientIP, clientPort);
                continue;
            case -2:
                startNewReceive(clientIP, clientPort, fromClientMsg);
                continue;
            case -3:
                startNewSend(clientIP, clientPort, fromClientMsg);
                continue;
            default:
                if(recId < 0) continue;
                break;
        }
        if(!clients.containsKey(clientIP)) continue;
```

Break 后就是普通的接收数据包。

服务端最主要的功能是维护了多用户的同时操作，这是通过维护很多数组来运行的，这很简单，其中有很关键的两个 map：

```
private Map<InetAddress, Integer> clients = new HashMap<>();
private TreeMap<Integer, InetAddress> clientsByID = new TreeMap<>();
```

第一个 map 是以客户端 ip 为 key，服务端给他分配的 id 为 value，第二个则相反，注意第二个 map 为 TreeMap 可以自动排序，因此服务端通过这个 map 来看哪些 id 是可用的，哪些 id 是已经被用的，从而才可以合理地分配 id。而第一个 map 则是频繁的通过 ip 来获得 id，从而访问其他变量数组的工具，两者相辅相成，密不可分。

服务端另一个功能是关于接收窗口的维护，接收窗口通过接收到的数据包组号和服务端期望的数据包组号来进行接收窗口的调整，有时需要缩小，有时需要释放，有时需要向前移动，逻辑关系比较复杂，主要采用 List 数组的形式来维护多用户的接收窗口：

```
private ArrayList<Integer>[] msgWins = new ArrayList[MAX_CLIENT_COUNT];
```

每一个接收窗口都是一个 list，方便我们进行窗口内数据包组号的调整。

### 3. 总体设计：

客户端总体设计：

在客户端中，分为两个功能，发送文件和接收文件，其中，发送文件的命令为 lftp lsend serverIP filename；下载文件的命令为 lftp lget serverIP filename。serverIP 为服务端的 IP 地址或者域名，filename 为需要上传或下载的文件名。上传时，如文件不存在或服务端已经存在该文件，则上传失败；下载时，若本地存放目录已经存在该文件或服务器中无该文件，则下载失败。且上传文件与下载文件使用同一个端口。

两个命令为在命令窗口中执行，通过获取命令窗口的参数，来得到 serverIP 与 filename 的值，以及进行的是上传操作还是下载操作，如下：

```
D:\f1y\java2>1ftp 1send 139.199.83.68 user2.jpg
```

```
D:\f1y\java2>1ftp 1get 139.199.83.68 user2.jpg
```

客户端涉及的类为四个：UDPClient 类，用于封装客户端的 main 函数，包括解释操作命令以及上传文件时的两个线程的处理；SendMsg 类，用于封装传输文件数据时或其他含有数据的报文段，包括报文段序号、报文段数据以及报文段长度等信息，同时封装了讲这些信息序列化和解序列化的方法；ACKMsg 类，主要用于上传文件时，接收端发送 ACK 报文段的处理，包括 ACK 的序号以及 rwnd 信息，以及序列化和解序列化的方法；Server 类，考虑到下载文件时，客户端的操作实际上与上传文件时服务端的操作类似，于是便仿照服务端接受文件的类，但是由于服务端要维护多线程来处理多用户情况，而客户端这里不需要，故将多线程的处理以及用于维护多用户的那些处理都去掉，其他设计与服务端的 Server 类相似。

**服务端总体设计：**

服务端设计的类为四个：Server 类，维护服务端的全部功能，实现多用户功能，对每个用户增加一个线程来服务，上传文件时，对于每个用户中的线程，维护当前的窗口长度以及收到的数据，当数据有序时，将数据写入硬盘中，并且每次接到一个报文段时，对客户端发送反馈信息，包括下一个所需的报文段以及当前的窗口长度；UDPClient 类，用户要求下载文件时，服务端开启一个线程和新开一个端口，执行 UDPClient 类，从当前的服务器文件中找到用户所需下载的文件，开启两个线程，一个线程负责发送文件，另一个线程负责接收客户端的反馈，实现思想同客户端上传文件时的 UDPClient 类；ACKMsg 类，用于发送和接收 AKC 报文段的处理；SendMsg 类，用于发送和接收带有具体数据的报文段的处理。

用户请求发送文件时，Server 类接收到该用户的请求报文，获取用户的 IP 地址以及端

口信息，为该用户维持一个线程和一个文件夹，接收用户数据及对用户数据进行反馈。当用户的文件传输完成后，将文件保存在服务器中，不再维持用户的 IP 地址和端口信息，即将该用户从当前正在操作的用户队列中删除。

用户请求下载文件时，由于用户发送的命令是在 Server 类接收的，而将文件发送给用户需要另外开一个线程及另外一个发送端口，而用户一般使用的是私有 IP 地址，需要使得用户网关的 NAT 地址表中有服务端中下载文件窗口的信息，为了解决这个问题，Server 先向用户发送一个报文段，里面包含有分配给用户下载文件的端口信息，然后客户端自动向该端口返回一个 ACK 信息，表示愿意用这个端口，同时使得 NAT 地址表中有相应的映射，从而使得服务端接收到 ACK 信息后，便开始发送文件。发送文件的具体流程与客户端上传文件的流程设计类似。

通过 Server 类和 UDPClient 类在服务量和客户端两边的同时调用，可以实现服务端的“上传”和客户端的“下载”