Design Document

16340296 张子权

Overview

LFTP使用UDP作为传输层协议，支持在互联网上大文件传输，在应用层上实现了流量控制，拥塞控制以及多用户访问，和TCP相似能够提供稳定的数据传输。

Usage

启动服务器：

```
java -jar MyServer.jar [port]
# default port: 8080
# example: java -jar MyServer.jar 8080
```

启动客户端：

Ther port should be the same as the Server's port.

```
Sending File:
java -jar MyClient.jar lsend hostname [port] filename
# default port: 8080
# example: java -jar MyClient.jar lsend localhost 8080 test.mp3
# note: the port is the Server's port, Client's port is random
```

```
get File:
java -jar MyClient.jar lget hostname [port] filename
# default port: 8080
# example: java -jar MyClient.jar lget localhost 8080 test.iso
# note: the port is the Server's port, Client's port is random
```

Design process

基本工作：

采用了UDP作为传输层协议，先使用Client-Server模式，完成服务端与客户端的简单发包和接受包，试验udp的传输方式和方法。

```
DatagramSocket serveSocket = new DatagramSocket(serverPort);
DatagramPacket dataPacket = new DatagramPacket(new byte[allbits], allbits);
```

```
dataPacket.setData("data".getBytes(), 0, 4);
serveSocket.send(dataPacket);
```

文件传输：

开始访问： 客户端对服务端的一个常驻监听端口发送packet（类似于连接请求）使得服务端能够分出一个端口处理客户端的后续消息。此时根据客户端的请求（发送\接收文件）来进行下面的步骤

```
clientAdd = packet.getAddress();
clientPort = packet.getPort();
String result = new String(packet.getData());
filename = result.substring(result.indexOf("file:") + 5).trim();
if (result.contains("lsend")) {
    mode = "lsend";
} else if (result.contains("lget")) {
    mode = "lget";
}
```

发送/接收文件： 使用DatagramPacket进行传输二进制包，客户端接收后能够正确地获取，然后再进行数据获取逐步写入到硬盘上。

结束传输： 发送端发送一个结束的packet给接收端，从而使得接收端停止接收，发送端不再发送数据。结束访问。

```
DatagramSocket serveSocket = new DatagramSocket(serverPort);
DatagramPacket dataPacket = new DatagramPacket(new byte[allbits], allbits);
dataPacket.setData("end".getBytes(), 0, 3);
serveSocket.send(dataPacket);
```

稳定性：

seq 与 ack： 完成基本传输后，在普通的udp的结构上加入序列号以及确认号，使得文件能够有序的被确认，防止因为packet到达时间出错导致文件顺序紊乱,仿造TCP的模式，采用加入确认序号。

校验和： 在发送时产生校验和，加入到包的头部，当传输到客户端时，客户端可进行一个验证，若校验和符合，则接受，发送ack，若不符合，则不发送ack，等待服务端的重传。

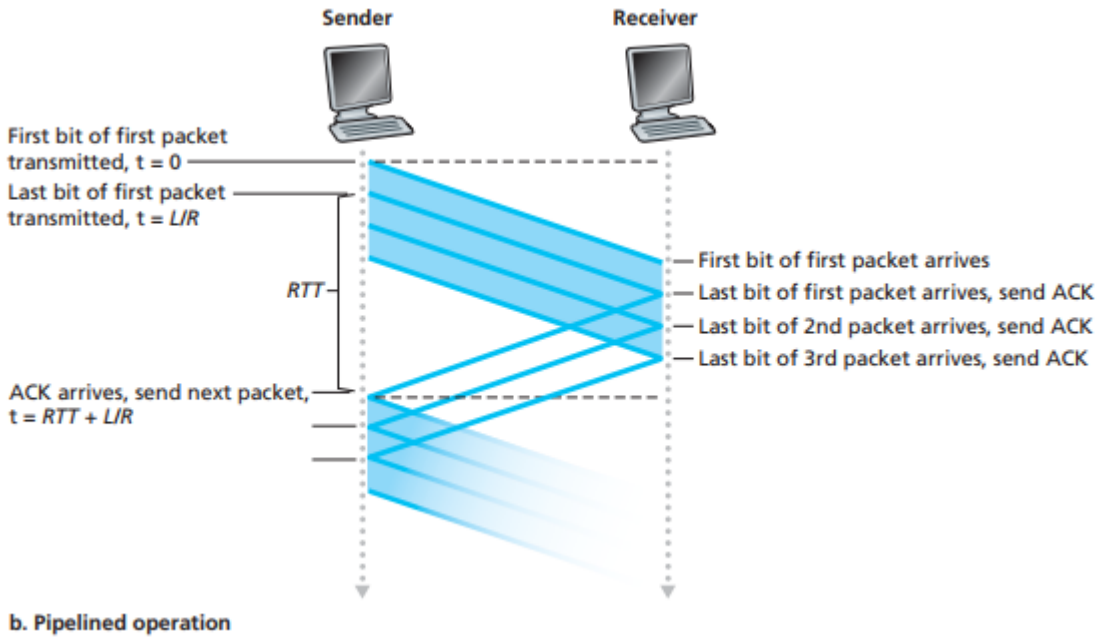
```
// compute the checksum.
byte[] checksum = SumCheck(buffer, checksumbits);

// compute the seq.
System.arraycopy(seq.getBytes(), 0, seqData, 0, seq.getBytes().length);
```

MyUDP Packet

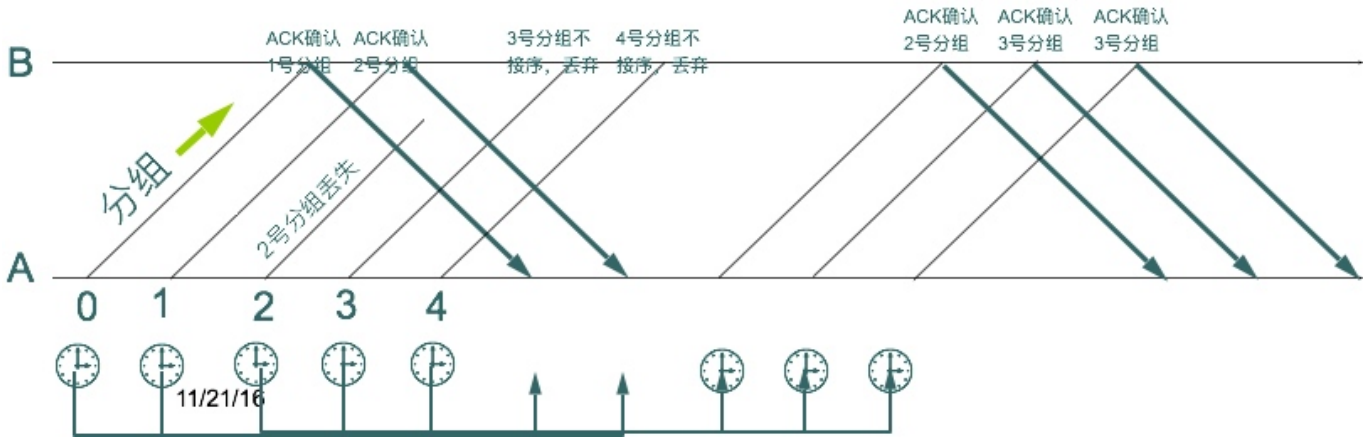
MyUDP Packet		
Sequence Number	Acknowledgement Number	checksum
Data Filed		

停等协议rdt2.0： 按照TCP的rdt2.0在能够完成传输文件的基础上，使用停等的方式，实现接受一个ack，发送一个packet的方式完成最基础的稳定传输，当发生丢包时，进行重传。

GBN回退N步模式： 在rdt2.0的基础上完成流水线的发送操作，即Go Back N模式，采用发送窗口使得一次性能够发送N个包，通过确认使得窗口能够进行移动。




```
serveSocket.send(dataPacket);
Timer newtimer = new Timer(timeout, new DelayActionListener(serveSocket, end));
newtimer.start();
```



流量控制：

GBN流水线： 进行多个包的传输，并且根据缓冲区的剩余空间确定窗口的大小，接收端在返回的ack包中添加入rwnd即窗口大小，从而控制发送端的发送窗口，防止缓冲区溢出。发送端发送的包的数量应当保持

$$LastSendByte - LastAckByte <= rwndsize$$

最后构造的UDP包的结构如下：

MyUDP Packet	
Sequence Number	Acknowledgement Number
checksum	receive windows size
Data Filed	

拥塞控制：

cwnd窗口： 除了通过返回的窗口大小对发送的控制之外，使用拥塞窗口进行控制，根据当前的链路情况决定发送的包的数量，从而降低丢包率。让链路更加通畅。

拥塞控制：

状态	控制
cwnd<=ssthresh	cwnd *= 2
cwnd>ssthresh	cwnd += 1
重复ack	ssthresh = cwnd/2
	>3： cwnd = ssthresh +1
	<=3： cwnd = ssthresh +1
超时	ssthresh = cwnd/2 , cwnd = ssthresh +1

超时加倍： 当发生超时时，超时加倍，避免在重传阶段发送更多的数据包到链路当中，同时也避免过多的不必要重传。在超时之后，在正常连续接受ack后将超时限制每次*0.95，逐步恢复到原来的时间。

多客户端访问：

多线程管理： 在服务端中使用多个线程，当有客户端进行访问时，分配一个未被占用的端口同时新建一个线程负责对那一个客户端进行服务。客户端一开始的访问都是访问监听端口，得到响应的包后就能得到随后的文件传输所用的端口。从而实现了多个客户端可以同时进行访问，文件的传输。

```
Runnable runnable = new Handle(clientAdd, clientPort, filename, mode);
Thread thread = new Thread(runnable);
thread.start();
```

Improvement

构造缓冲区

当客户端写两个线程，一个负责接受服务端发送的包，一个负责将包中的数据写入到磁盘中，从而构造出一个缓冲区，便于窗口的建立，若写入速度过慢，则缓冲区中的数据增多，这时就需要缩小窗口，通知服务端减小

发送窗口。

```
while (tranfer) {
    while (filecache.size() != 0) {
        System.out.println("the size of cache is " + filecache.size());
        byte[] buffer = filecache.get(0);
        filepointer.write(buffer, 0, buffer.length);
        filecache.remove(0);
    }
    Thread.sleep(5);
}
```