RDT Programming Report

File Share P2P for Computer Networking 2016 Spring

Name:ShenJiyuan St.No.:5130309194 E-mail:sheryalyuan@126.com

***Abstract*—This report describes a socket programming about a simple file share application using UDP Socket APIs in detail and implementing RDT3.0 at the application layer. Here, the Java Socket API is chosen as the implementing tool.**

***Index Terms*—Socket, C/S, UDP, TCP, RDT3.0, file share, P2P, client, server.**

I. Recall Problem

Implement unreliable file transfer:

1) transfer a large file to a remote host using UDP;
2) take the number of error bits, and the total delay of transmission from the first bit sending to the last bit received.

Implement reliable file transfer 1:

Using RDT 3.0 protocol at the application layer, based on the implementation on the unreliable file transfer. Compare # of error bits, and the total delay of transmission.

Implement reliable file transfer 2: Using TCP.Compare # of error bits, and the total delay of transmission.

II. Introduction of Basic Concepts

UDP(User Datagram Protocol) uses a simple connectionless transmission model with a minimum of protocol mechanism. It has no handshaking dialogues, and thus exposes the user's program to any unreliability of the underlying network protocol. There is no guarantee of delivery, ordering, or duplicate protection. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram.

TCP(Transmission Control Protocol) originated in the initial network implementation in which it complemented the Internet Protocol(IP), then the entire suite is commonly referred to as TCP/IP. TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating over an IP network.

RDT(Reliable Data Transfer) is important in application, transport and link layer. Characteristics of unreliable channel will determine complexity of reliable data transfer protocol. And RDT 3.0 assumes that underlying channel can also lose packets(data or ACKs, checksum, seq#, ACKs, retransmissions will be of help but not enough). The approach is that sender waits 'reasonable' amount of time for ACK and schedules done by checking the ACKs.

III. On Java API Functions

TCP (same as last project!)

'java.io' includes all functions provided for input and output stream, for UI and file.'java.net' includes that for socket. In the following table, I list all the relating functions in the two packages related to socket programming for file sharing and used in my program TCP.

java.io	
BufferedReader	change 'InputStreamReader' into buffer
readLine()	read the buffer content(better efficiency)
dataInputStream	file data stream, for client read(here)
read()	client read data from stream
dataOutputStream	file data stream, for server write(here)
writeBytes()	server write data to stream
fileInputStream	file data stream, for server read(here)
fileOutputStream	file data stream, for client write(here)
write()	client write data to file
java.net	
Socket	socket for both server and client
connect()	client connect to server by socket
ServerSocket	listening and controls in server
accept()	server listen to a port and get connection

UDP

'java.io' and 'java.net' for UDP are mostly same as the above for TCP. Follows are a list of all API functions I used.

java.io	
File	record for file
getName()	get the file name
FileInputStream	file data stream, for server read(here)
available()	read available byte amount
read()	client read data from stream
FileOutputStream	file data stream, for client write(here)
write()	client write data to file
java.net	
DtagramSocket	create and listen to one port for packet
connect()	connect to a socket
receive()	receive a packet to a defined packet var
getData()	return the packed byte array
getLength()	return the packed byte array length
send()	send a packet out

java.net	
DatagramPacket	data structure for datagram packet
getAddress()	return the IP address
getPort()	return the port
receivePacket()	receive a packet from socket
InetAddress	record for IP address
java.util	
StringTokenizer	split string into tokens
Timer	gauge the time delay

RDT 3.0

Since we have to implement an int ACK and a boolean corrupt flag here for RDT3.0, I packet them in one data structure and use the object I/O stream.(append upon the above table)

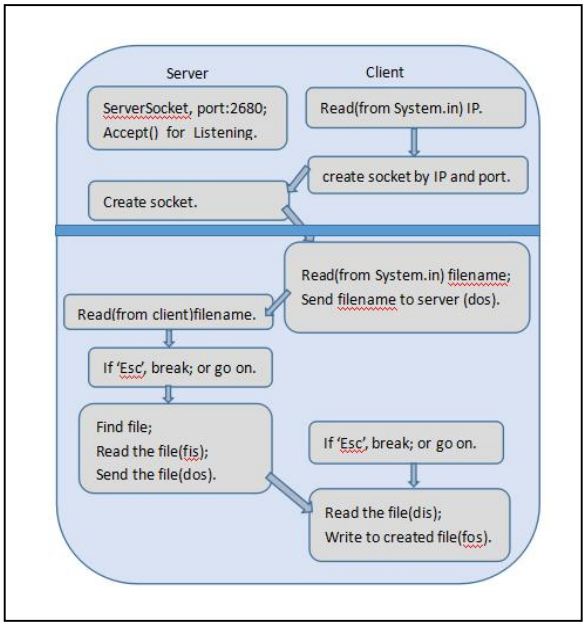
java.io	
ObjectInputStream	serialize with fileinputstream
readObject()	read the object from the file
ObjectOutputStream	serialize with fileoutputstream
writeObject()	write the object to the file
java.util	
Timer	gauge the time
schedule()	Deal with time out trigger

IV. Programming

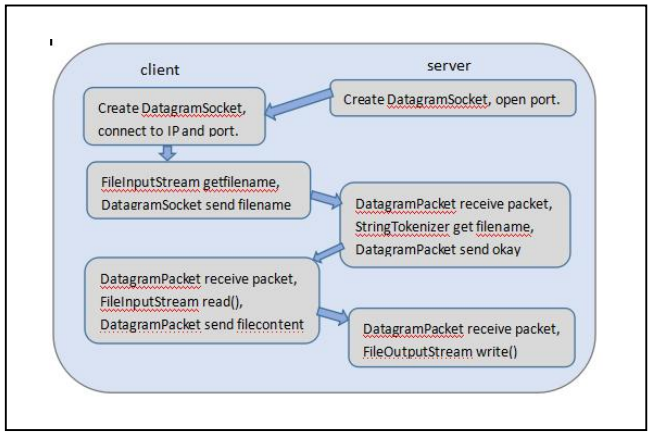
In this part, I will show you the whole programming process and detailed relations between the socket process.

A. OverView（important）

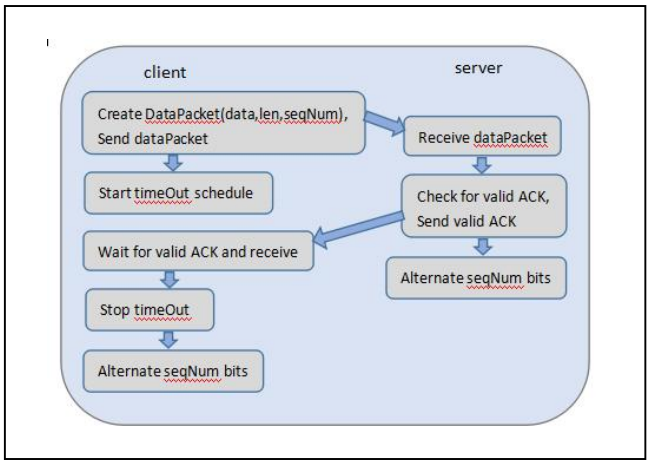
TCP (same as last project!)



UDP



RDT 3.0



For above three step flows, pay attention that some small gadgets like waiting for or checking valid ACK and iscorrupt are simple while-loops.

B. Detailed Description on Key Codes

Environment: Eclipse, JavaSE-1.8

(1) New a Java Program in Eclipse, named 'Project1'.

(2) New 2 classes, named 'FileSender', 'FileReceiver'. (schedule selection between TCP and simple UDP)

TCP (same as last project!)

TCPClient

- Read(from System.in) IP & create socket

```
BufferedReader inFromUser=new
BufferedReader(new
InputStreamReader(System.in));
//输入 ip,如 59.78.29.78
String hostname=inFromUser.readLine();
socket = new Socket();
socket.connect(new    InetSocketAddress
(hostname, 2680),10 * 1000);
```

- Read(from System.in) filename & send to server

```
dos = new DataOutputStream (socket.
getOutputStream ());
String filename=inFromUser.readLine();
dos.writeBytes(filename+"\n");
```

- Read(from DataInputStream) file & Write to file

```
if ((length = dis.read(inputByte, 0,
inputByte.length)) > 0)
{
   System.out.println(length);
   fos.write(inputByte, 0, length);
}
```

TCPServer

- Create ServerSocket & Listening & create socket

```
final ServerSocket server = new
ServerSocket(2680);
try {
Socket socket = server.accept();
```

- Read(from client) filename

```
BufferedReader inFromClient=new
BufferedReader(new InputStreamReader
(socket.getInputStream()));
String
fileName=inFromClient.readLine();
```

- Find file (by filename)

```
File file = new File(fileName);
FileInputStream fis = null;
fis = new FileInputStream(file);
```

- Read the file & Write the file

```
while ((length = fis.read(sendBytes, 0,
sendBytes.length)) > 0) {
dos.write(sendBytes, 0, length);
```

UDP

UDPClient

- Read(from System.in) IP & create socket

```
public UDPSender(InetAddress address,
int port) throws IOException{
    toPort = port;
    toAddress = address;
    msg = new byte[8192];
    buffer = new byte[8192];
    s = new DatagramSocket();
    s.connect(toAddress, toPort);
}
```

- Read(from System.in) filename & send to server

```
fileReader = new
FileInputStream(theFile);
fileLength = fileReader.available();
send((theFile.getName()+"::"+fileLengt
h).getBytes());
```

- If receiver has got the filename, send file content

```
while (currentPos<fileLength){
  bytesRead = fileReader.read(msg);
  send(msg);
  currentPos = currentPos + bytesRead;
}
```

UDPServer

- Open datagramSocket , wait for and get filename

```
s = new DatagramSocket(port);
initPacket = receivePacket();
initString = "Recieved-"+new
String(initPacket.getData(), 0,
initPacket.getLength());
StringTokenizer t = new
StringTokenizer(initString, "::");
filename = t.nextToken();
```

- Send filename-received okay

```
send(initPacket.getAddress(),initPacke
t.getPort(),(newString("OK")).getBytes
());
```

- Receive file content

```
fileWriter=new
FileOutputStream(filename);

while(bytesReceived < bytesToReceive){
receivedPacket = receivePacket();
fileWriter.write(receivedPacket.getDat
a(), 0,  receivedPacket.getLength());
bytesReceived = bytesReceived +
receivedPacket.getLength();}
```

- Send valid ACK and terminate if it is terminating packet

```
assert p.seq == seqNumber;
if(p.data == null) {
    udt.send(new AckPacket(p.seq));
    udt.close();
    return null;
} else {
    udt.send(new AckPacket(p.seq));
  }
seqNumber = alternateBit(seqNumber);
```

RDT 3.0

RDT Client

- Send packet

```
DataPacket p = new DataPacket(data,
length, seqNumber);
udt.send(p);
```

- Start timeOut

```
Timer timeOut = new Timer();
timeOut.schedule(new
senderWithTimer(p),  100, 100);
```

- Receive ACK and Check

```
AckPacket ack = udt.recv();

while(ack.isCorrupted==true          ||
ack.ack!=seqNumber)

{ack = udt.recv();}
```

- With Correct ACK

```
DataPacket p = udt.recv();
while(p.isCorrupted==true ||
p.seq!=seqNumber) {
udt.send(new
AckPacket(alternateBit(seqNumber)));
p = udt.recv();}
```

RDT Server

- Receive and validate received packet

```
assert (ack.isCorrupted==false &&
ack.ack==seqNumber);
timeOut.cancel();

seqNumber = alterateBit(seqNumber);
```

V. FileList and Run

A. FileList and Run

My program is written in Java, and all the codes are all in the 'src' filefold.

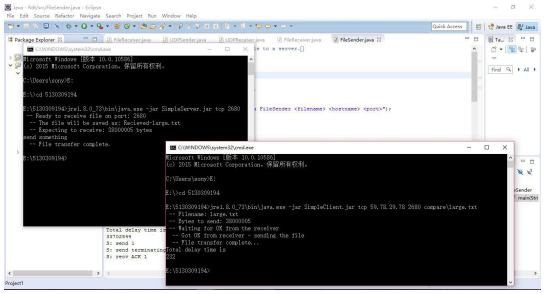
TCP and simple UDP are implemented in Project1. For TCP, the codes included are in files 'TCPSender.java' and 'TCPReceiver.java'. For UDP, the codes included are in files 'UDPSender.java' and 'UDPReceiver.java'.

UDP with RDT3.0 is implemented in Rdt. Here two data structures are used to simplify the original codes.

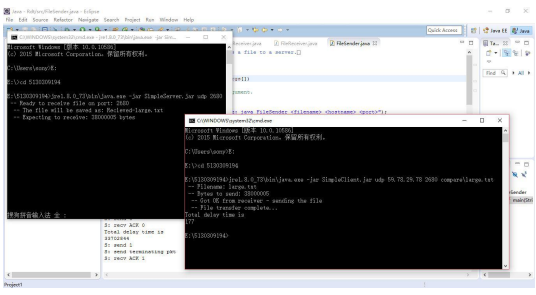
If you run these three apps in Java platform, please set your parameters first in the menu 'run configuration'.

B. ShotScreens when Running jar files

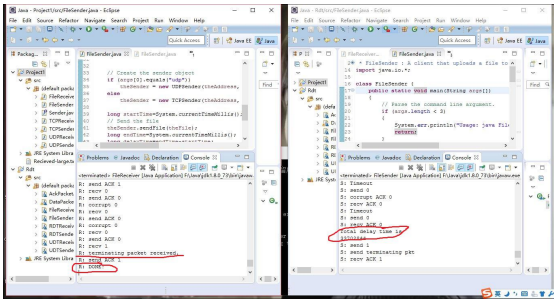
- TCP



- UDP



- UDP with RDT3.0



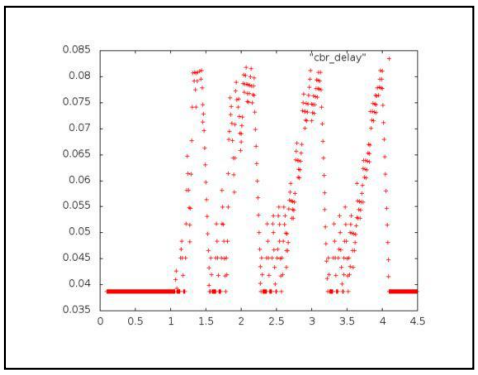
C. Analysis on delay time and # of error bit

File Size	Protocol	Delay Time(ms)	Error Bit
36.2MB (304021513)	tcp	232	null
	udp	177	21468
	udp+rdt3.0	264	null

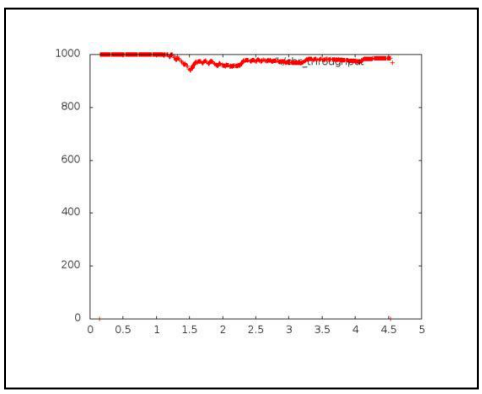
Then we will focus on the delay time by coding cpp file:

```
1. set terminal gif
2. set output "cbr_delay.gif"
3. plot "cbr_delay"
```

where figure over delay is plot:



where figure over miss is plot:



After comparisons, we can draw the conclusion that if the server and client are on a same computer, then the transmission speed would be udp>udp+rdt3.0>tcp. And if the server and the client are on different computers, then the speed would be tcp>udp>udp+3.0. Since udp can easily lose packets during transmission especially on different computers, so we know even if it do not need to greet, retransmission really cost a lot.

VI. Problems and Experience

A. Convert a java program into executable files

Not like C/C++, there are some tricks to convert a Java program into corresponding executable files. And After searching in the Internet, I take the following method:

1. Use the eclipse to transfer the objected main function into file '.jar' by...
   ① Export your program as 'Java-> Runnable Jar File'.
   ② Set your specifications: pay attention to that the library handling should select the 'Package required libraries Into generated JAR'.
2. Use the software 'JSmooth'
   ① Download the 'JSmooth' in http://jsmooth.sourceforge.net/
   ② By using your generated 'jar' to get the executable file.

Notice in this program, we will generate two 'jar' file, for Client and Server; and two executable files finally.

B. More on Two Defined Data Structures

When implementing rdt3.0, three more information including boolean iscorrupted, int sequence number, int ack should be added to the original udp protocol. And there is no need for a whole structure to contain them all, so I simply divide them into two data structures: class DataPacket and class AckPacket. DataPacket: seq, *data, len, isCorrupted; AckPacket: ack, isCorrupted.

Here, the DataPacket is for sending out the file contents to receiver_server; and the AckPacket is for client's receiving ack and server's sending ack.

C. Bubbles on savePath

For tcp and original udp implementing, the savaPath for the resulting copied file is the current directory. For udp with rdt 3.0 imlementing, the resulting copied file can be assigned in the parameters for RdtServer execution.