MNC Project

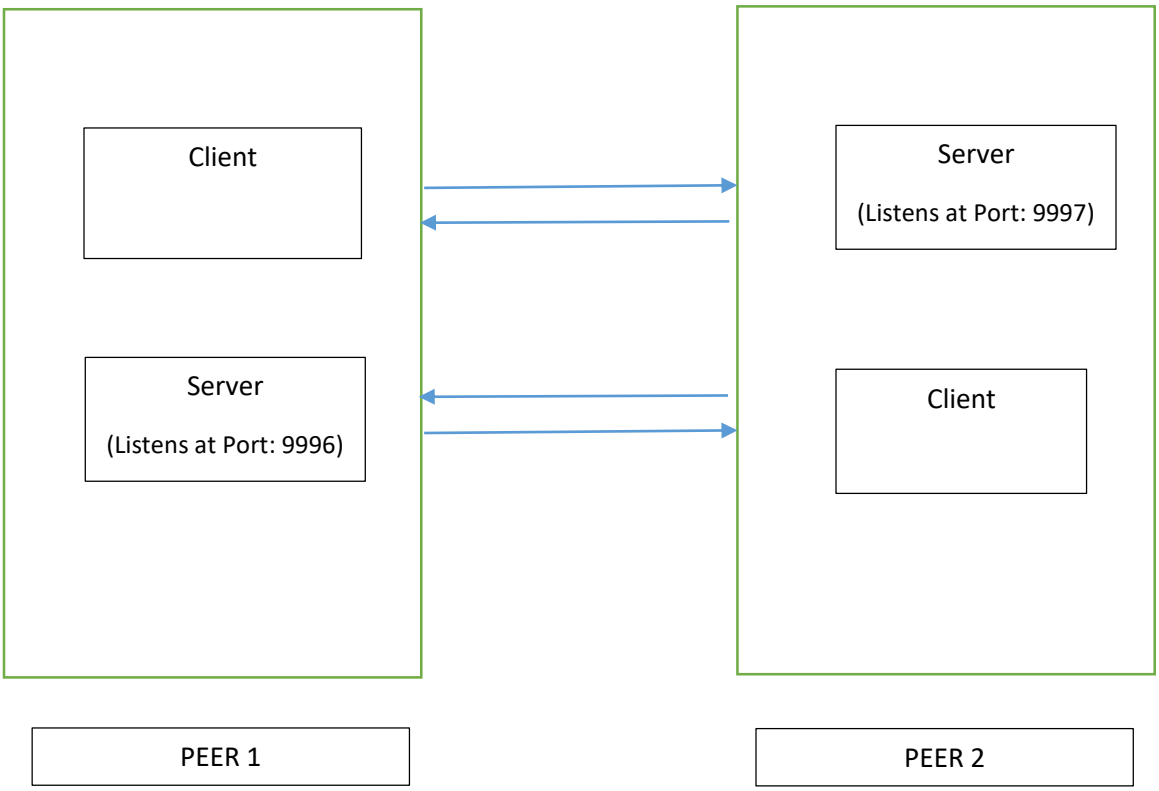
Team : Yash Mali & Chinmay Swami

Chat Application (Part I)

Abstract:

In order to build a peer-to-peer chat application we have used TCP Protocol for communication. The major reason been that TCP protocol guarantees reliability which UDP fails to provide.

Design Diagram:



(Note: As we run the application on Local Host, Peer1 and Peer2 server listens on two different ports. In real applications same ports can be used to listen as those two process would ideally run on two different machines)

Code Explanation:

We have used Java 8 for building the chat application. Our Part A consists of 3 files namely Peer1.java, ServerThread.java and ClientThread.java. Let us look into role of each file in our project.

1) Peer1.java

```
public class Peer1 {


  public static void main(String[] args) {

        Thread Ser = new Thread(new ServerThread(9996));
        Thread cli = new Thread(new ClientThread(9997));
        Ser.start();
        cli.start();


       }


   }
```

Peer1.java file contains the main function which is the beginning point for our project. The above code creates two different threads for Server and Client. Those threads are started using the function start ( ). The “9996” mentioned in the ServerThread is the port number Peer 1 would listen to. “9997” is the port number the client of Peer1 would connect to. The server of Peer 2 would be listening at port “9997”.

2) ServerThread.java

```
public class ServerThread implements Runnable{

        int port;

        public ServerThread(int port)
        {
                this.port = port;

        }


        @Override
        public void run() {

                ServerSocket serSoc = null;
```

```
            try {
                  serSoc = new ServerSocket(port);
            }
            catch (IOException e1) {

                  e1.printStackTrace();
            }

            int b = 0;
            char c;
            String a = "";
            byte [] buffer = new byte[100];

        try {

             while(true) {


            Socket soc = serSoc.accept();
            b = 0;
            InputStream in = soc.getInputStream();

            in.read(buffer);

            a = new String(buffer).trim();

            System.out.println("She: " + a);

            OutputStream os = soc.getOutputStream();
            os.write("ACK".getBytes());

            Arrays.fill(buffer, (byte)0);
            os.flush();
            soc.close();


         }

        }

        catch(Exception e)
        {

            System.out.println(e);

        }


      }

}
```

ServerThread.java contains the logic for the peer server. The ServerThread.java performs the following important tasks

1) Creating a server socket that continuously listens to a mentioned port number.

   ServerSocket soc = new ServerSocket(portnumber);

2) Listen to the socket. Once a client connects provide a different port for connection establishment. This is in fact done to make sure that the server keeps on listening to the mentioned port.

```
Socket soc = serSoc.accept();
```

3) The “soc” is further used to communicate with the client using inputstream and outputstream. The InputStream “in” is used to send data while the OutputStream “os” is used to receive data from other peer.

```
InputStream in = soc.getInputStream();
OutputStream os = soc.getOutputStream();
```

4) A byte array “buffer” is used to read the incoming data. It is necessary to clear the buffer as it may contain previous byte data.

```
byte [] buffer = new byte[100]; // Declaring a buffer
Arrays.fill(buffer, (byte)0);   // Clearing the buffer
```

5) Once the communication is done it is important to close the socket. As a socket is a resource which should be released once it’s use is complete. In complex applications this play an important role.

   soc.close()

6) Finally, it is necessary that the server is always up, So infinite while loop is used to ensure the same.

(Note: A final “ACK” message is sent from server to client after each message is received by the server. This is done to ensure synchronization of the processes.)

3) ClientThread.java

```
public class ClientThread implements Runnable {

	int port;


	public ClientThread(int port)
	{
		this.port = port;
	}

	@Override
	public void run() {


		String msg;
		Scanner scan = new Scanner(System.in);

		System.out.println("Start the conversation");



		while(true)
		{

		    msg = scan.nextLine();
		    System.out.println("You: " + msg);

		 try {
			Socket socket = new Socket("localhost",port);
			OutputStream out = socket.getOutputStream();
			InputStream in = socket.getInputStream();
			byte [] buffer = new byte[50];
			out.write(msg.getBytes());
			in.read(buffer);
			String  a = new String(buffer).trim();

		if(a.equals("ACK") ) {

		    out.flush();
		    out.close();
		    socket.close();

		}


		} catch (IOException e) {

			System.out.println("");
			e.printStackTrace();
```

```
            }

            }

        }
}
```

ClientThread.java contains the logic for the peer server. The ClientThread.java performs the following important tasks

1) Creating a socket that connects to other peer's Server. The Socket constructor takes two arguments. First is the IP Address and the second is the port number.

```
Socket socket = new Socket("localhost",port);
```

2) If it is connected to the server, a socket is created through which it can communicate with the server. Otherwise an exception would be thrown stating "unable to connect to the Server port." So it is necessary that the server is running when the client tries to connect.

3) The "socket" created is further used to communicate with the server using inputstream and outputstream. The InputStream "in" is used to send data while the OutputStream "os" is used to receive data from other peer.

```
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
```

4) A byte array "buffer" is used to read the incoming data. It is necessary to clear the buffer as it may contain previous byte data.

```
byte [] buffer = new byte[50];  // Declaring a buffer
Arrays.fill(buffer, (byte)0);   // Clearing the buffer
```

5) Once the communication is done it is important to close the socket. As a socket is a resource which should be released once it's use is complete. In complex applications this play an important role.

```
socket.close()
```

6) Finally, in order to ensure that the chat application is always active the client code is put into infinite while loop.

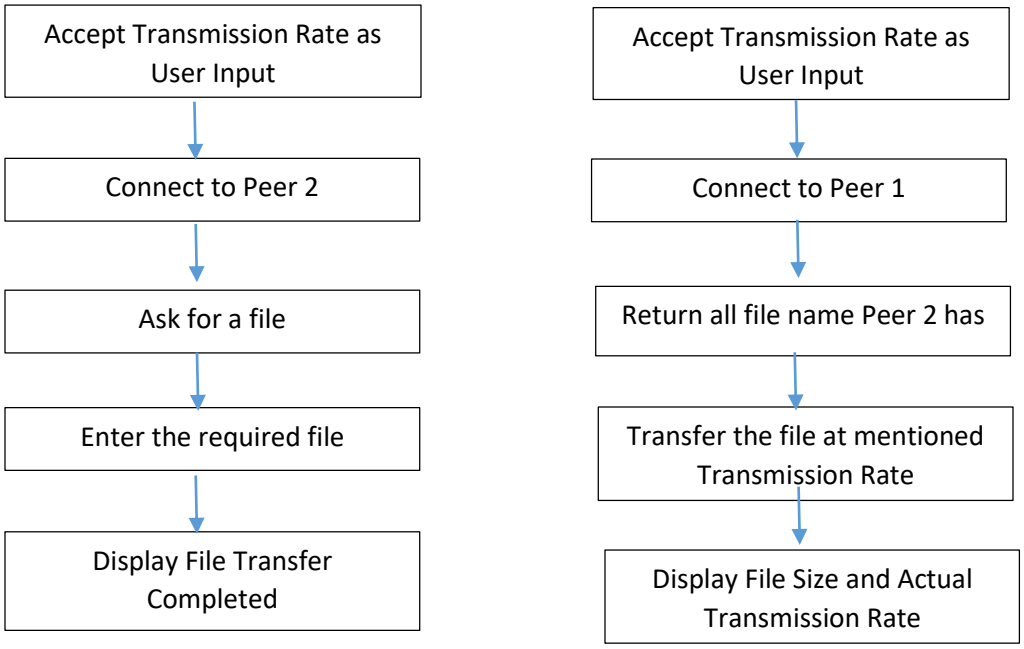
File Transfer Application (Part II)

Abstract:

In order to build a peer-to-peer File Transfer application that needs control over transmission rate it is ideal to use UDP. UDP does not implement Flow or Congestion Control which can help us attain high speed. But it does not ensure reliability due to which we opted for TCP. UDP would be an ideal choice if we could ensure reliability at application level.

Flowchart:

The design diagram for Part II is same as Part I.



PEER 1 PEER 2

(This explains a scenario when PEER 1 asks for a file from PEER 2 and PEER 2 sends the file. The roles would be switched if PEER 2 asks for a file and PEER 1 sends the file)

Transmission Rate Control:

In order to control Transmission Rate we have used Thread.sleep(). Suppose a user mentions 80 Mbps as transmission rate i.e. 8 MB/Second. We keep a track of the number of bytes transmitted. Suppose 8 MB data is transmitted in 0.7 seconds (or 700 milliseconds), we stop transmitting for 0.3 seconds (or 300 milliseconds, that is call Thread.sleep(300 milliseconds). So Eventually the transmission rate is limited to 8 MB/sec or 80 Mbps.

Code Explanation:

We have used Java 8 for building this application. Our Part B consists of 4 files namely Handler.java, Server.java, Client.java, FileDir.java. Let us look into role of each file in our project.

1) Handler.java

This file contains the main function which is the beginning point for our application. The above code creates two different threads for Server and Client. Those threads are started using the function start ( ). The “9996” mentioned in the ServerThread is the port number Peer 1 would listen to. “9997” is the port number the client of Peer1 would connect to. The server of Peer 2 would be listening at port “9997”.

```
public class Handler {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		System.out.println("Please Mention the Transmission Rate in
					Megabits per Second:");
		Scanner scan = new Scanner(System.in);
		Thread Ser = new Thread(new Server(9996,scan.nextInt()));
		Thread cli = new Thread(new Client(9997));
		Ser.start();
		cli.start();
	}

}
```

Transmission Rate is taken as a user input and then passed on to the Server thread.

2) FileDir.java

The code is used to get all the files in a specified directory. These filenames are further passed to the other Peer. So Peer can now choose from a range of file which he requires

```
public class FileDir {


    public File[] finder( String dirName){
        File dir = new File(dirName);

        return dir.listFiles(new FilenameFilter() {
                public boolean accept(File dir, String filename)
                    { return filename.endsWith(".txt"); }
        } );

    }


    public File[] finderpdf( String dirName){
        File dir = new File(dirName);

        return dir.listFiles(new FilenameFilter() {
                public boolean accept(File dir, String filename)
                    { return filename.endsWith(".pdf"); }
        } );

    }



}
```

3) Sever.java

This part of code performs all the activities mentioned in Part A, such as ServerSocket creation, listening to the port and transferring data.

Apart from that it performs the major task of controlling the transmission rate. The logic is discussed earlier in the report. This part of the code takes care of it.

```
        while((c = fin.read(k))!= -1)
{
                os.write(k);
                Arrays.fill(k, (byte)0);
                size = size + 1024;
                 temp = temp + 1024;

            if(temp>transRate)
            {
                midTime = System.currentTimeMillis() - startTime;
                if(1000-midTime>0)
                {
                Thread.sleep(1000 - midTime);
                }
               temp = 0;
               startTime = System.currentTimeMillis();

}
```

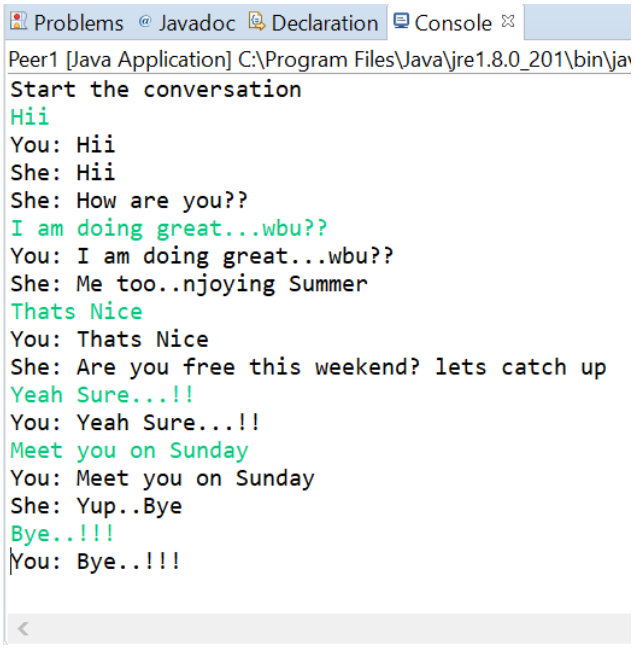
At the end of the file Transfer the Server displays the File Size and the actual rate at which file was transmitted in Mbps.

4) Client.java

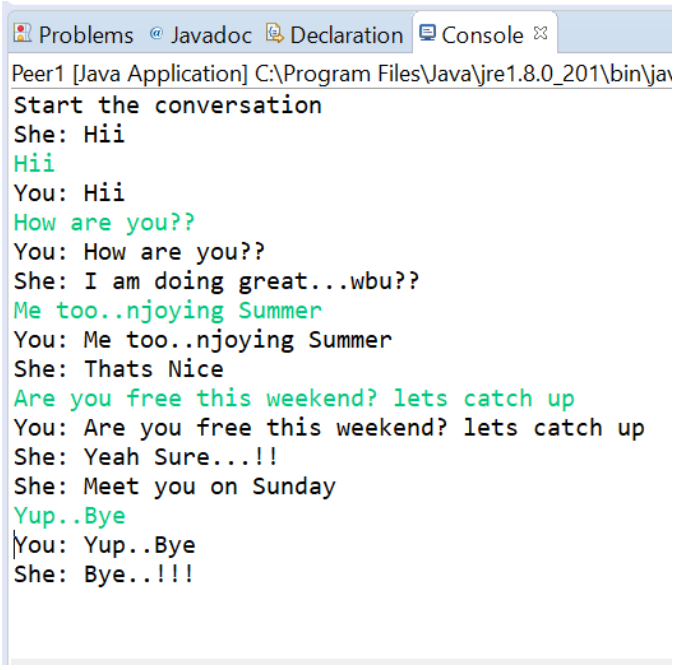
This part of code performs all the activities mentioned in Part A, such as Socket creation, connecting to the other Peer and transferring data. Apart from this, it enables user to receive a list of files available from other Peer and then choose the required file. It also displays a message upon file received.

Result Screenshots:

Part A (Chat Application):



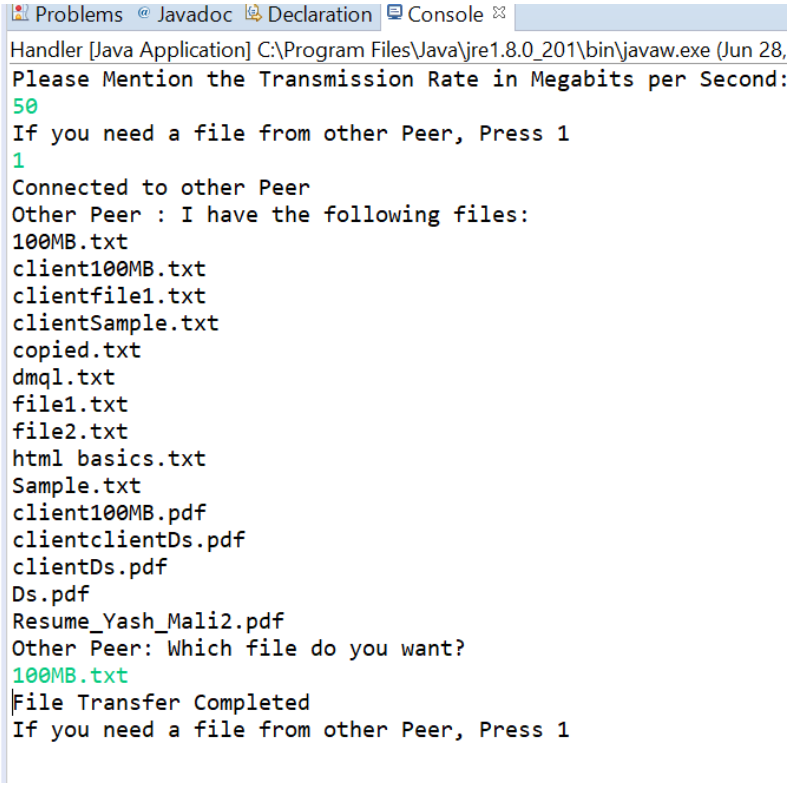
Peer 1



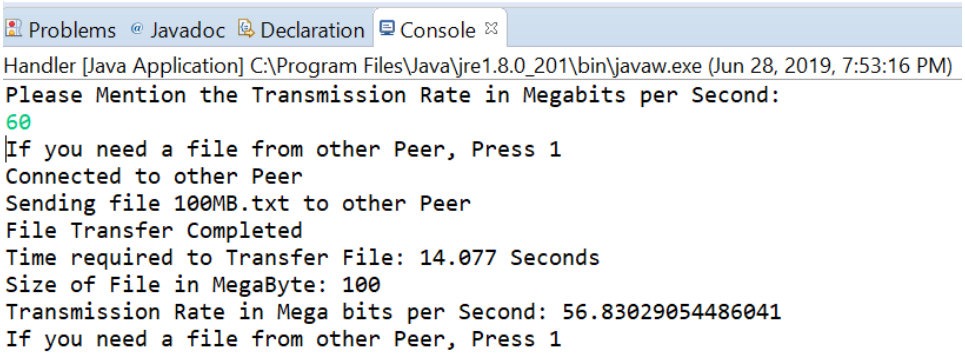
Peer 2

Part B (File Transfer Application) :

1) Peer 1 asking for files from Peer 2



Peer 1 acting as Receiver



Peer 2 acting as Sender (Transmission Rate : 60 Mbps , Actual : 56.8 Mbps)

2) Peer 2 asking for files from Peer 1

```
If you need a file from other Peer, Press 1
Connected to other Peer
Sending file 100MB.txt to other Peer
File Transfer Completed
Time required to Transfer File: 16.209 Seconds
Size of File in MegaByte: 100
Transmission Rate in Mega bits per Second: 49.35529644024925
If you need a file from other Peer, Press 1
```

Peer 1 acting as Sender (Transmission Rate : 50 Mbps , Actual : 49.35 Mbps)

```
If you need a file from other Peer, Press 1
1
Connected to other Peer
Other Peer : I have the following files:
100MB.txt
client100MB.txt
clientfile1.txt
clientSample.txt
copied.txt
dmql.txt
file1.txt
file2.txt
html basics.txt
Sample.txt
client100MB.pdf
clientclientDs.pdf
clientDs.pdf
Ds.pdf
Resume_Yash_Mali2.pdf
Other Peer: Which file do you want?
100MB.txt
File Transfer Completed
```

Peer 2 acting as Receiver

References:

1) Java Socket API : https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html
2) Java I/O Tutorial : https://www.javatpoint.com/java-io
3) James F. F. Kurose and Keith W. Ross, "*Computer Networking: A Top-Down Approach Featuring the Internet*", 7th edition, Addison Wesley, 2017.
4) Peer to Peer Communication : https://techterms.com/definition/p2p