
# Indian Institute of Technology, Roorkee

A mini project on

**"Multimedia File Sharing Using Socket Programming"**

Under the subject of

**Advanced Computer Networks**

Submitted by: Group 4


| | |
|---|---|
| **Abhishek S Adhikari** | **20535002** |
| **Ankit Daswani** | **20535004** |
| **Anunay Katare** | **20535005** |
| **Ayush Kasara** | **20535009** |
| **Devyanshu Sengal** | **20535012** |
| **Kartik Sharma** | **20535035** |
| **Maj Gaurav Thapliyal** | **20535036** |

# Contribution from members

| Enrollment No | Contribution | Page No. |
|---|---|---|
| 20535002 | Created MultithreadSend class file for implementing the connection establishment of multiple sender threads to receiver and then to send a file in parts to Receiver and some testing to yield results. | 13 - 16 |
| 20535005 | Created StdComm.java file for two way sending of files in a single application using a single port. It implements multiple threads to handle different operations. | 8-10 |
| 20535004 | Explored use of dataOutputStream instead of OutputStream and suggesting its use and running test cases | 8 |
| 20535009 | Created MultithreadReceive class for implementing the logic of receiving the files from sender using multi threading and showing final output | 10 - 12 |
| 20535012 | Finding ip address of receiver dynamically for socket connection. | 7 |
| 20535035 | Code for creating FileChooser & assistance provided for finding errors, creating data flow and control flow structure of the program | 8 |
| 20535036 | Assistance provided in building a rough program structure and plan of code reuse from single thread program to avoid logical errors | 8 |

## Problem Statement

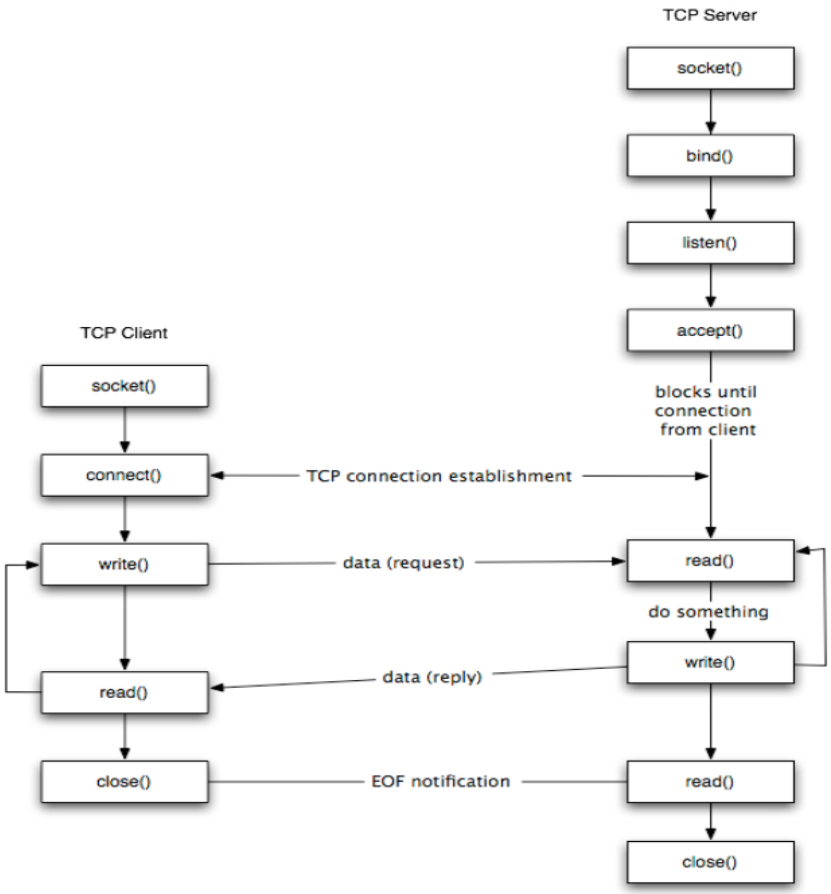
Develop an application which can share large multimedia files between two nodes on the same network using socket programming. Further optimise the application using multithreading to run faster for large files. Show performance gain in multithreading over a single threaded program.

## File Sharing

File sharing, in one form or another, is as old as computer networks. Programs such as uucp (unix to unix copy) and, later, ftp (from the file transfer protocol that it used) constituted the early means through which files were shared among network users.

## Socket Programming

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while the client reaches out to the server.



The steps for establishing a TCP socket on the client side are the following:

- Create a socket using the socket() function;
- Connect the socket to the address of the server using the connect() function;
- Send and receive data by means of the read() and write() functions.

The steps involved in establishing a TCP socket on the server side are as follows:

- Create a socket with the socket() function;
- Bind the socket to an address using the bind() function;
- Listen for connections with the listen() function;
- Accept a connection with the accept() function system call. This call typically blocks until a client connects with the server.
- Send and receive data by means of send() and receive().

| Socket type | Protocol | Description |
|---|---|---|
| SOCK_STREAM | Transmission Control Protocol (TCP) | The stream socket (SOCK_STREAM) interface defines a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent. |
| SOCK_DGRAM | User Datagram Protocol (UDP) | The datagram socket (SOCK_DGRAM) interface defines a connectionless service for datagrams, or messages. Datagrams are sent as independent packets. The reliability is not guaranteed, data can be lost or duplicated, and datagrams can arrive out of order. However, datagram sockets have improved performance capability over stream sockets and are easier to use. |
| SOCK_RAW | IP, ICMP, RAW | The raw socket (SOCK_RAW) interface allows direct access to lower-layer protocols such as Internet Protocol (IP). |

## SOCKET CLASS

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

### Important methods

| Method | Description |
|---|---|
| 1) public InputStream getInputStream() | returns the InputStream attached with this socket. |
| 2) public OutputStream getOutputStream() | returns the OutputStream attached with this socket. |
| 3) public synchronized void close() | closes this socket |

### ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

## Important methods

| Method | Description |
|---|---|
| 1) public Socket accept() | returns the socket and establish a connection between server and client. |

| | |
|---|---|
| 2) public synchronized void close() | closes the server socket. |

## Approach for Sharing files without any multithreading

This is one of the most convenient form of sharing files across network. Since we are dealing with a private LAN, it might offer great performance for file transfer.

There is a Receiver code which will be listening on a specific port eg. 5000, and will look out for any incoming request. If some request is there then it will use the accept() method of java.net.* package for allowing to establish a connection.

Similarly there is a Sender code which will try to connect with the Receiver using the socket class of java. It requests a connection using new Socket(String IP, int port).

Sender and Receiver are now connected with each other over a TCP connection. Now sender can share its data easily by using methods like read(), readUTF() etc., correspondingly, the Receiver will also accept the bytes from its inputstream. As long as sender is sending the file receiver will get some data and as soon as sender stops, receiver will close the connection using close().

## Approach for Sharing files With multithreading

This is another way, much more cumbersome but may provide better performance if used with right hardware.

There is a Receiver code which will listen on a specific port eg. 5000, and will look out for any incoming request. If some request is there then it will use the accept() method of java.net.* package for allowing to establish a connection, creates a handler thread for that socket and then goes on to listen for other incoming requests until its mentioned number of incoming requests are not met with.

The Sender code will try to create the specified number of threads and each thread will request for a socket to the Receiver. After this all threads will send a certain portion of file from their outputstream on the socket.

Each handler thread will accept the data and finally merge all the files. Thereby, completing the sharing process.

## Contribution by Devyanshu Sengal (20535012)

Before jumping to the main idea of the project, i.e multimedia file sharing, the very first crucial thing is to establish a connection between two nodes. And for that we have to get the IP addresses of both the parties. And for fulfilling the purpose we have used the InetAddress class provided by Java.

The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on. This class represents an Internet address as two fields: hostName (a String) and address (an int). hostName contains the name of the host; for example, *www.google.com*. address contains the 32-bit IP address. These fields are not public, so you can't access them directly. It will probably be necessary to change this representation to a byte array when 16-byte IPv6 addresses come into use. However, if you always use the InetAddress class to represent addresses, the changeover should not affect you; the class shields you from the details of how addresses are implemented.

The following code has been devised to get the IP address of the receiver :

### findIPReceiver( ) function:

```
public static String findIPReceiver() throws Exception{
        Process result = Runtime.getRuntime().exec("tracert -h 1 www.google.com");
        System.out.println("Finding IP address of Receiver...");
        // to store the output in some buffer
        BufferedReader output = new BufferedReader(new InputStreamReader(result.getInputStream()));
        String info = null;
        String[] tokens = null;
        int i = 0 ;
        while((info = output.readLine()) != null){
            i++;
            if(i == 5){
                tokens = info.split("\\s{1,}");
                break;
            }
        }

    String IPADDRESS_PATTERN = "(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)";

    Pattern pattern = Pattern.compile(IPADDRESS_PATTERN);
    Matcher matcher = pattern.matcher(tokens[tokens.length-1]);
    if (matcher.find()) {
        String temp = matcher.group();
        System.out.printf("IP address of Receiver --> %s\n", temp);
            System.out.println("-------------------------------------------------------------");
        return temp;
    } else{
        return "0.0.0.0";
    }
}
```

The presented snippet of code uses tracert command to get the IP address of the destination/receiver. After getting the IP address it checks if the IP address received is valid or not.

## Contribution by Ankit(20535004)

Explored use of dataOutputStream instead of OutputStream and suggesting its use as Dataoutputstream is independent of different formats used in different machines.
Running test cases.

## Contribution by Maj Gaurav Thapliyal(20535036)

For assisting in quick development of multithreaded file sharing programs, a general program structure was made, which lists the classes and methods that are going to be used and the data structures required.

Also a plan of code reuse from a single thread program where appropriate methods from the program could be used was laid out, this not only saved time but also reduced the chances of logical errors and reduced the complexity of the program.

## Contribution by Kartik Sharma(20535035)

Before writing the code for the sender and receiver, a general idea of the structure of both these classes is required.
This is to get an idea of what functions and classes are going to be used, what data types are going to be used.
A general idea of the control flow and data flow in the program was also discussed which helps not only in development of the program but also helps in removing logical errors more quickly.

Also wrote the code for creating a file chooser in java with which the user can navigate through directories and can select the file he/she want to send.

## Contribution by Anunay Katare(20535005)

### StdComm.java

Use: To continuously share multimedia files between two computers using single thread model.

Java classes used: ServerSocket, Socket, DataOutputStream, DataInputStream, Exception, Scanner , String , Thread.

The code first of all asks for the IP of the computer which can be seen from the network setting.
It is asked only once in execution.

Here, the main() function is the main() thread. It makes objects of Send and Receive classes then it then creates threads. So they can perform their jobs.
The Send class keeps on watching console if we write “send” on console it asks for the receiver’s IP and the file that is to be transferred;
The send first sends the format and length of file.
Format  so that the receiver does not have to enter file name repeatedly the file gets stored in the Format- “received1.txt”,”received2.mp4” etc.

The receiver comprehends the type of file and creates a filename accordingly. Sender and receiver communicate in blocks of 8KB so that the  buffer does not take a lot of heap space.

Some code snippets are as follows:-

```
public Receive(Socket sock) {

    this.sock=sock;
}
```

The Receive class constructor that takes the sock thread from main() function and assigns it to the socket object in the Receive class.

The following code is used by Receiver Class to get the extension of the file and its length. Accordingly, it also sets the new file name.

```
DataInputStream is=new DataInputStream(sock.getInputStream());

is.read(t,0,t.length);

 String type=new String(t, StandardCharsets.US_ASCII);
 System.out.println(type);
 switch(type)
 {
     case "mp4": file=file+".mp4";
         break;
     case "jpg": file=file+".jpg";
         break;
     case "png": file=file+".png";
         break;
     case "avi": file=file+".avi";
         break;
 }
  System.out.println(file);
  long size;
  size=is.readLong();
  fos = new FileOutputStream(file);
```

Following is used by the sender to send files. It uses bytearray as buffer of 8KB.

```
File myFile = new File (filename);
byte [] bytearray  = new byte [8192];



fis = new FileInputStream(myFile);
bis = new BufferedInputStream(fis);

DataOutputStream os=new DataOutputStream(sock.getOutputStream());

os.write(type.getBytes(StandardCharsets.UTF_8));
os.writeLong(myFile.length());
int count;
while ((count = bis.read(bytearray)) > 0)
  {
   os.write(bytearray, 0, count);
  }
```

## Contribution by Ayush Kasara(20535009)

**MultithreadReceive.java**

Use - for receiving a file over socket using multiple threads.
Java Classes used - ServerSocket, Socket, DataOutputStream, DataInputStream, Exception, Thread.

The main thread between receiver and sender first of all exchanges the file name that is to be shared along with the size of file. The sender creates the threads at his end and those threads will send connection requests to the main thread of the receiver.

The receiver main thread establishes all connections, but to handle the input from the sender side, a thread handler is created. So in this way thread1 at sender may connect with thread3 at receiver side.
We store the thread results in some temporary files. The sender side will send the name of the temporary file. It is helpful in reordering the file. The sender will create the temporary file with the same name received and it will continue to do so until the sender stops sending.

To ensure the receiver will stop receiving after the sender stops sending files, we maintained a busy wait kind of lock structure. The main thread will combine all the temporary files one by one and it will simultaneously delete the non-required files to ensure free space. Lastly one single file will be there, that will be our output.
The receiver thread has a constructor ReceiverThread that will use thread id, socket, and a boolean variable complete and performs the above mentioned work.

Some snippets of these classes are shared in the next pages.The main() function will perform the work for assembling the data coming from the sender. ReceiverThread class will implement the connection and collection of data from sender threads part with sender.

```
6  public class MultithreadReceive{
7
8      private static int port = 5000;
9      private static int BUFFER_SIZE = 16384;
10
11     public static void main(String args[]) throws Exception{
12         int n = 4;
13         boolean[] completed = new boolean[n];
14         ServerSocket server = new ServerSocket(port);
15         ArrayList<ReceiverThread> thread_list = new ArrayList<ReceiverThread>();
16
17         System.out.println("Receiver process started");
18         System.out.println("Waiting for a Sender ...");
19         Socket socket = server.accept();
20         System.out.println("Sender request accepted");
21         System.out.println("----------------------------------------------------------------");
22         long startTime = System.currentTimeMillis();
23         DataInputStream socketInputStream = new DataInputStream(socket.getInputStream());
24         String temp = socketInputStream.readUTF();
25         String[] arrOfStr = temp.split("SEP", 2);
26         //System.out.println(arrOfStr[0] + " --------- "+ arrOfStr[1]);
27
28         String filename = arrOfStr[0];
29         System.out.println("FileName -> "+arrOfStr[0]);
30         System.out.println("FileSize -> "+arrOfStr[1]);
31         System.out.println("----------------------------------------------------------------");
32         socketInputStream.close();
33         socket.close();
34
35         for(int i=0;i<n;i++){
36             Socket sock = server.accept();
37             ReceiverThread obj = new ReceiverThread(i,sock,completed);
38             thread_list.add(obj);
39         }
40
41         for(ReceiverThread t: thread_list){
42             t.start();
43         }
44
45         boolean flag;
46         while(true){
47             flag = true;
48             for(int i=0;i<n;i++){
49                 if (completed[i] == false){
```

```
        System.out.println("------------------------------------------------------------");
        long endTime = System.currentTimeMillis();
        System.out.println("Total sent time: " + Long.toString(endTime - startTime));
        System.out.println("----------------- All files received and now assembling ----------------");

        DataOutputStream os = new DataOutputStream((new FileOutputStream(filename)));
        File file = null;
        byte[] byte_read = new byte[BUFFER_SIZE];
        for(int i=0;i<n;i++){
            DataInputStream is = new DataInputStream((new FileInputStream("temp"+Integer.toString(i))));
            try{

                int length = 0;
                while((length = is.read(byte_read)) !=-1){

                    os.write(byte_read, 0 ,length);
                    os.flush();
                }

            }catch(Exception e){
                System.out.println("while assembling files.");
                System.out.println(e);
            }
            is.close();
            file = new File("temp"+Integer.toString(i));
            Files.deleteIfExists(file.toPath());
        }
        os.close();
        System.out.println("Assembling done!");
        System.out.println("############################################################");
```

```
    public ReceiverThread(int thread_id, Socket socket, boolean[] completed){
        this.socket = socket;
        this.completed = completed;
        this.thread_id = thread_id;
    }

    public void run(){

            try{
                inStream = new DataInputStream(socket.getInputStream());

                String str = inStream.readUTF();
                details = str.split("SEP",2);
                System.out.println(details[0]+" ^^^^^^^^^^ "+details[1]);

                outStream = new DataOutputStream(new FileOutputStream(details[0]));
            }catch(Exception e){
                System.out.println(e);
            }

            try{
                int length = 0;
                while((length = inStream.read(buffer))!= -1){
                    outStream.write(buffer, 0 ,length);
                    outStream.flush();
                }
                System.out.println(" data is received from -> " + Integer.toString(thread_id));

            }catch(Exception e){
                System.out.println(e);
            }

            try{
                inStream.close();
                outStream.close();
                socket.close();

            }catch(Exception e){
                System.out.println(e);
            }
            completed[thread_id] = true;
    }
```

## Contribution by Abhishek Singh Adhikari(20535002)

**MultithreadSend class and SenderThread class:**

Use - for sending a file using multiple threads over sockets.

*Basic idea behind the program is to initially create 'n' number of threads apart from the **main_thread,** create a socket from each thread towards the receiver and make these threads equally responsible for sending certain part of the file towards the receiver.*

Java Classes used: Socket, RandomAccessFile, DataOutputStream, Exception, Thread

The main_thread create 'n' child threads, the code for child thread is present in the class ***SenderThread*** present inside MultithreadSender.java**.** A Parameterized Constructor of **SenderThread** class takes several parameters, like thread_id, receiver_ip, original_filename, original_filesize and a commonly shared Boolean array etc.,

In the run() method of thread class, all socket connection and sending of bytes is taken care of.

- To define which thread will send how much portion of the file and also from which byte number (**start position**) in the file up to which byte number (**end position**), initially we calculated the size of the original file and then equally divided it into each thread i.e., **chunksize.**
- Now starting position of the thread = thread_id*chunksize where thread_id = {0,1,2,…}
- And ending position of the thread = starting pos + chunksize
- So to put the file handle in starting position, we used the **RandomAccessFile** class in JAVA, which allows the file handle to be put at a certain location.

Now each thread in the program will open an *outputstream* of their established socket and place the data(bytes) into it. Every thread will send its share of file in the form of bytes using the method **.read(byte[] buffer)** repeatedly with the help of *while() loop* and the receiver threads will handle them at the receiver side by storing them in some temporary files.

On completion each thread will close the socket and also update the Boolean array named completed on its index = thread_id with TRUE.

The main_thread will wait for these child threads to complete, ensure their completion with the help of join() and then finally exit.

```
public void send(){
    try{
        int n;
        JFileChooser jfc = new JFileChooser();
        int dialog_value = jfc.showOpenDialog(null);
        if(dialog_value == JFileChooser.APPROVE_OPTION){
                File target_file = jfc.getSelectedFile();
                String original_fileName = target_file.getName();

                Scanner in = new Scanner(System.in);
                System.out.print("How many threads you want to create at sender -> ");
                while((n = in.nextInt()) > 128){
                    in.nextLine();
                    System.out.println("Try again with value less than 33");
                }
                in.nextLine();
                //System.out.println("How many buffer size you want to keep -> ");  BUFFER_SIZE = in.nextInt(); in.nextLine();
                String host = null;

                System.out.print("Whether you want to enter reciver ip manually ?(y/n)-> ");
                char ch;    ch = in.next().charAt(0);       in.nextLine();
                //
                if('y' == Character.toLowerCase(ch)){
                    System.out.print("Enter receiver ip -> ");
                    host = in.nextLine();
                }else{
                    host = this.findIPReceiver();
                }

                System.out.println("------------------------------------------------------------");

                long original_fileSize = new File(target_file.getAbsolutePath()).length();
                long chunkSize = (long)(Math.ceil(original_fileSize/n)) + 1;
```

```
DataOutputStream socketOutputStream = new DataOutputStream(socket.getOutputStream());
socketOutputStream.writeUTF(original_fileName + "SEP" + Long.toString(original_fileSize)+"SEP"+Integer.toString(n));
socketOutputStream.close();
socket.close();

for(int i=0;i<n-1;i++){
    SenderThread obj = new SenderThread(host, i, target_file.getAbsolutePath(), chunkSize, chunkSize, completed);
    thread_list.add(obj);
}
SenderThread obj = new SenderThread(host, n-1, target_file.getAbsolutePath(), chunkSize, (long)(original_fileSize - (n-1)*(
thread_list.add(obj);

for(SenderThread t : thread_list){
    t.start();
}

for(SenderThread t: thread_list){
    t.join();
}

long endTime = System.currentTimeMillis();

System.out.println("------------------------------------------------------------");
System.out.println("Total sent time: " + Long.toString(endTime - startTime));
System.out.println("Sender's whole operation is completed");
lse{
System.out.println("Couldn't complete file selection.Try again");

eption e){
.out.println(e);
```

```
public void run() {
    Socket socket = null;
    RandomAccessFile inStream = null;
    DataOutputStream outStream = null;
    byte[] bytes_read = new byte[BUFFER_SIZE];

    try{
        socket = new Socket(host,port);
        System.out.println("Thread "+ Integer.toString(thread_id) + " is connected");
    }catch(Exception e){
        System.out.println(e);
    }

    try{
        inStream = new RandomAccessFile(original_fileName,"r");
        outStream = new DataOutputStream(socket.getOutputStream());
        outStream.writeUTF("temp" + Integer.toString(thread_id) + "SEP" + Long.toString(fileSize));

    }catch(Exception e){
        System.out.println(e);

    }


        Long start = (thread_id*chunkSize);
        Long end = start + fileSize;
        //System.out.println(Long.toString(start) +" to "+ Long.toString(end));
        try{

            inStream.seek(start);
            long count = (fileSize/BUFFER_SIZE);
            int remain = (int)(fileSize - count*BUFFER_SIZE);
```

```
while(count-- != 0){
    //System.out.println(Integer.toString(thread_id)+" : count "+Long.toString(count));
    inStream.read(bytes_read);
    outStream.write(bytes_read);
}

byte[] temp = new byte[remain];
inStream.read(temp);
outStream.write(temp);
```

## Testing

Testing was done in the combined code of MultithreadSend.java and MultithreadReceive.java by connecting two computers through wi-fi and actually shared several files. Timings were Noted down on certain results to calculate a maximum achieved performance gain of 9.767% as compared to single threaded sharing on a file of size 13.5GB.

Multiple combinations of buffer size and number of threads ‘n’ were checked, finally reaching the conclusion that the multithreaded sharing may bring performance gain but it highly depends on the underlying hardware i.e., number of processors. In some testing we allowed a large number of threads to be created on 2 or 4 physical cores of the system, but eventually it led to performance degrade as unnecessary context switching overhead was there.

| File Size | Single thread Send time(msec) | N | Multi-thread Send time(msec) | Performance gain (%) |
|---|---|---|---|---|
| 13.5GB | 2193794 | 4 | 2000237 | 9.677 |
| 3.34GB | 606191 | 4 | 563688 | 7.540 |
| 749MB | 199445 | 4 | 205153 | -2.78 |

Performance gain over single thread model

| File Size | No. of Threads | Transfer time(msec) |
|---|---|---|
| 3.82GB | 2 | 703066 ≈ 11.71 min |
| 3.82GB | 4 | 695917 ≈ 11.59 min |
| 3.82GB | 8 | 685183 ≈ 11.41 min |
| 3.82GB | 64 | 753862 ≈ 12.56 min |
| 3.82GB | 128 | 964423 ≈ 16.07 min |

Variation of transfer time when no. of threads are changed