IP Project 1

Submitted by :

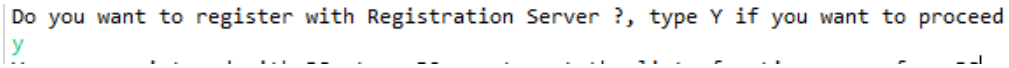
Arpita Sakhuja (asakhuj)

Surbhi Bansal (sbansal3)

Message Formats used in the project

Step 1 : Registration between Peers and Registration server

We have created a user interactive application. Whenever the peer program starts running, we ask the logged in peer to register with the Registration server. The following message is displayed to the user :



If the user enters “Y” , we send a registration message to the Registration Server :

REGISTER

Hostname : <Host_name>

Port number : <Port_number>

where <host_name> and <port_number> are the details of the peer trying to register with the RS.

The Registration server checks it’s database(Linked List in our project) , and if there is no entry of the respective combination of the hostname and the port number, it registers the peer and sends the following message :

REGISTERED

Cookie : <unique_cookie_number>

Where <unique_cookie_number> is unique to every peer.

Moreover, if the peer is already registered with the Registration server (RS), the server won’t register the peer again. Rather, the RS will send the already present cookie for that peer, with the same message format :

REGISTERED

Cookie : <already_present_cookie>

Step 2 : Peer requests for a list of active peers from the Registration server

Whenever peer requests for a list of active peers from the Registration server, we send a PQuery from the peer with the following message format :

```
You are registered with RS, type PQuery to get the list of active peers from RS
```

PQuery

Hostname : <Host_name>

Port number : <Port_number>

Where hostname and the port number are the details of the peer requesting the peer list from the RS.

The response from the RS will be in the following message format if there are active peers present :

PQuery-Response

STATUS 200 OK

<SIZE OF PEER LIST>

<DATA>

Where STATUS CODE 200 enforces that there are active peers present in the system. <SIZE OF THE PEER LIST> is important as the client will understand that how many peers are active in the system and will fetch the respective number of data. <DATA> will contain the port numbers of the active peers in the local system.

The response from the RS will be in the following message format **if there are no active peers present** :

PQuery-Response

STATUS 100 NOT_OK

Where STATUS CODE 100, enforces that there are no active peers present in the system.

Step 3 : If there are active peers present in the network and the peer requests for RFC-Index from the active peers

In our user interactive application, we have asked the user to request for the RFC-Index from the active peers in the network :

```
Type RFC-Index if you need RFC-Index from all the active peers in system
RFC-Index
Your response RFC-Index
```

If the user enters “RFC-Index”, we will send the following message from the logged in peer to all the active peers in the network :

GET RFC-INDEX

Whenever the server of the peer will encounter the GET RFC-INDEX message, it will send the local RFC Index in the following message format :

RFC-INDEX RESPONSE 200 OK

Size : <NUMBER OF FILES>

<DATA>

Where STATUS CODE 200 signifies that the request is successful. “NUMBER OF FILES” signifies the number of rfc files present with the peer server. “DATA” signifies the name of the rfc files present with the peer server.

Step 4 : Once the client peer contains the RFC-Index of all the active peers , it will merge the RFC-Index of the other peers with it’s own, so that it contains the list of all the rfc’s present in the network.

Step 5 : Once the peer contains the complete list of all the RFC’s, we ask the peer if he wants to download the files from the active peers :

```
Do you want to download all files from all the active peers ?, type Y if you want to proceed
y
```

If the user enters Y, we send the following message to the active peer, requesting a rfc :

PROVIDE RFC <RFC_NAME>

The server peer will provide the respective RFC with the following message :

PROVIDING RFC <RFC_NAME>

CONTENT-LENGTH <SIZE OF THE FILE>

<FILE CONTENT/DATA>

Where <RFC_NAME> is the file name requested by the client peer. <SIZE OF THE FILE> denotes the size of the file in bytes. <FILE CONTENT> is the file data in byte array.

Step 6 : Whenever the peer leaves the system , it will send the Leave message to the Registration Peer in the following format :

LEAVING

Hostname : <host_name>

Port number : <port number of the peer>

Note : In our application, we suspect a leave whenever the program encounters System.exit(0) message, which is a graceful turnoff or Ctrl+C in command prompt.

Step 7 : As we know, the peer has to send the Keep-Alive message , to confirm that it is still active, so the following message is sent after every 7200 seconds :

We have created a thread that runs after every 7100 seconds , in order to tell the server that it is alive.

```
TimerTask timerTask = new TimerTask() {

        @Override
        public void run()  {
            // TODO Auto-generated method stub
            try {
                if(Peer0.appActive) {
                    Socket clientSocket = new Socket(rsHostname, 65423);
                    DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream()); //For sending output to RS
                    BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream())); //For gettin
                   //Sending Registration details to RS
                    String RSQuery = "REGISTER";
                    outToServer.writeBytes(RSQuery + '\n');
                    outToServer.writeBytes("Hostname : "+myHostName + '\n');
                    outToServer.writeBytes("Port number : " +myPortNumber + '\n');
                    String registerServerResponse = inFromServer.readLine(); //Registered
                    String receivedCookie = inFromServer.readLine().substring(9);
                    System.out.println(registerServerResponse +" " +receivedCookie);
                    clientSocket.close();
                    outToServer.close();
                    System.out.println("You are registered with the Registration Server");

                }
            }
            catch(Exception e) {}

        }
    };

    Timer timer = new Timer();

    timer.scheduleAtFixedRate(timerTask, 0, 7100*1000); //After every 7100 seconds, register with RS

}
```

REGISTER

Hostname : <host_name>

Port number : <port_number>

The RS will give the following output to the Keep-Alive message

REGISTERED

Cookie : <already_present_cookie>

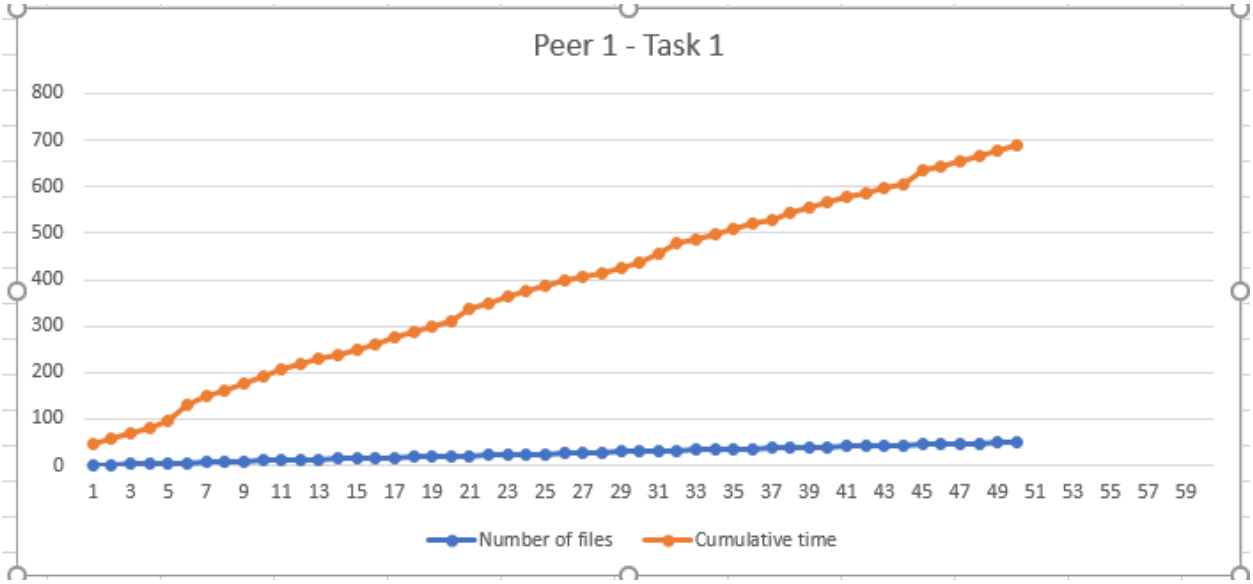
TASK 1 : Centralized Client-Server sharing

NOTE : SINCE P0 CONTAINS ALL THE 60 FILES, WE WILL CALCULATE THE TIME FROM P1 as P0 does not need any additional files. Hence no transfer required.

Peer 1 : 60 files → 1566 millisecond

50 files → 690 millisecond

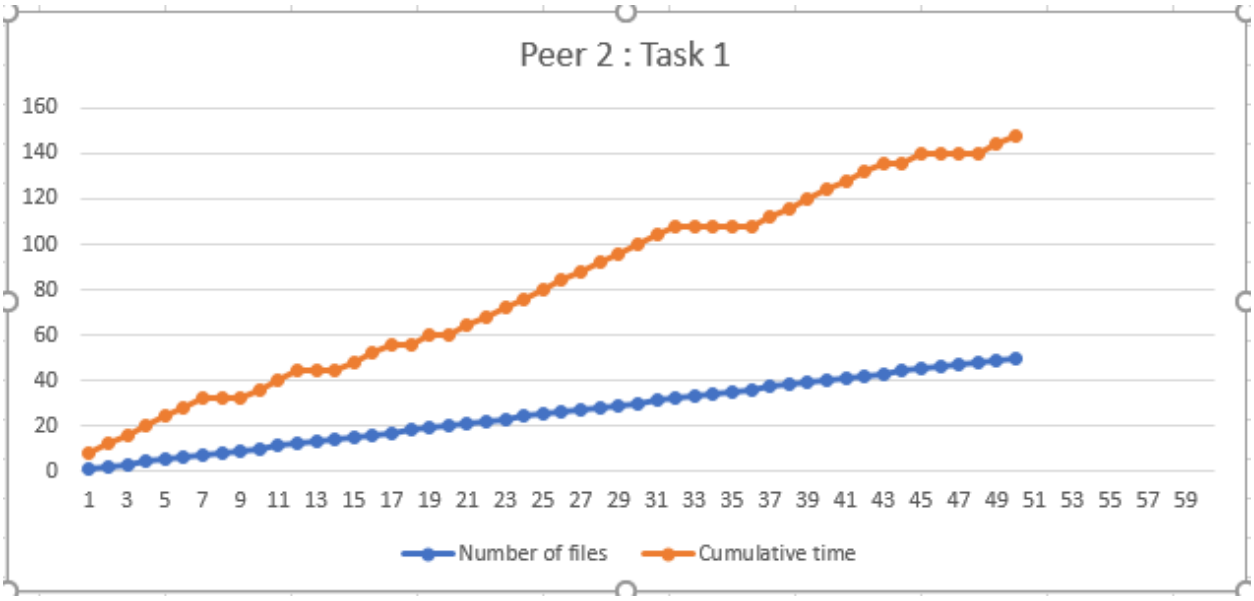
```
File F:/Arpita/IP/Project 1/RFC/Peer1/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 1566
```



Peer 2 : 60 files : 196 millisecond

50 files : 148 millisecond

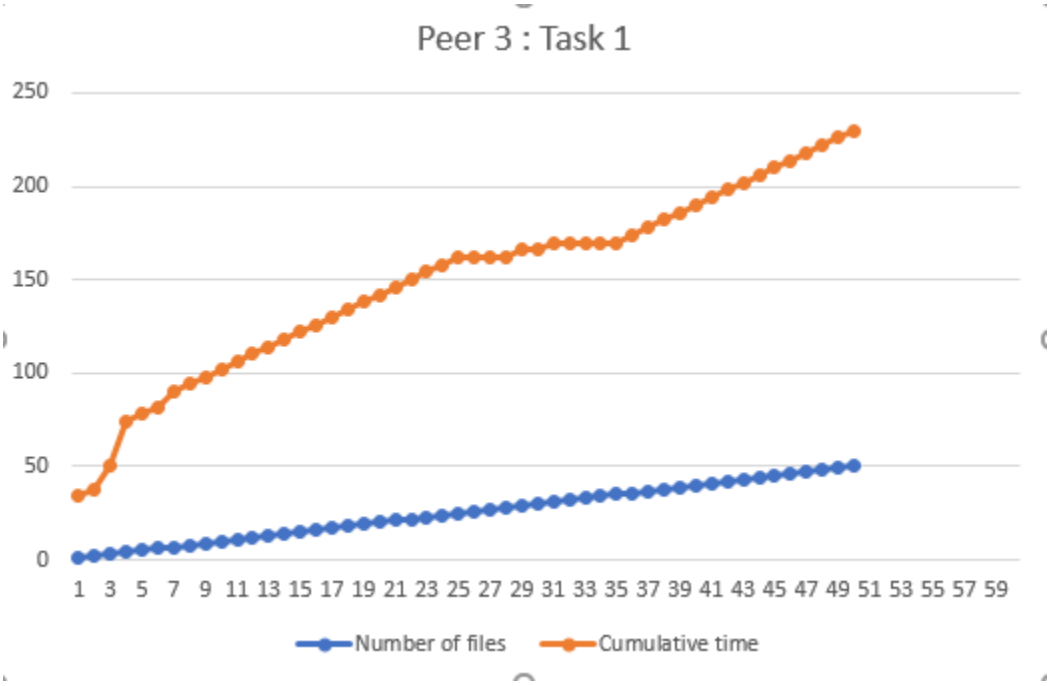
```
File F:/Arpita/IP/Project 1/RFC/Peer2/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 196
```



P3 : 60 files : 278 millisecond

50 files : 230 millisecond

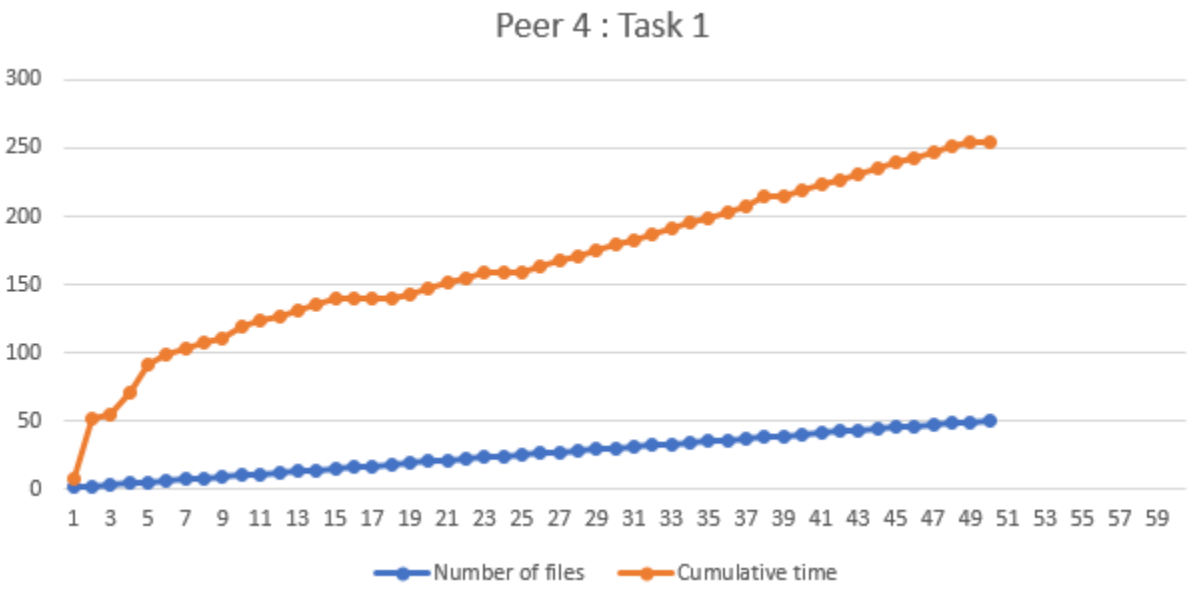
```
File F:/Arpita/IP/Project 1/RFC/Peer3/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 278
```



P4 : 60 files : 303 millisecond

50 files : 255 millisecond

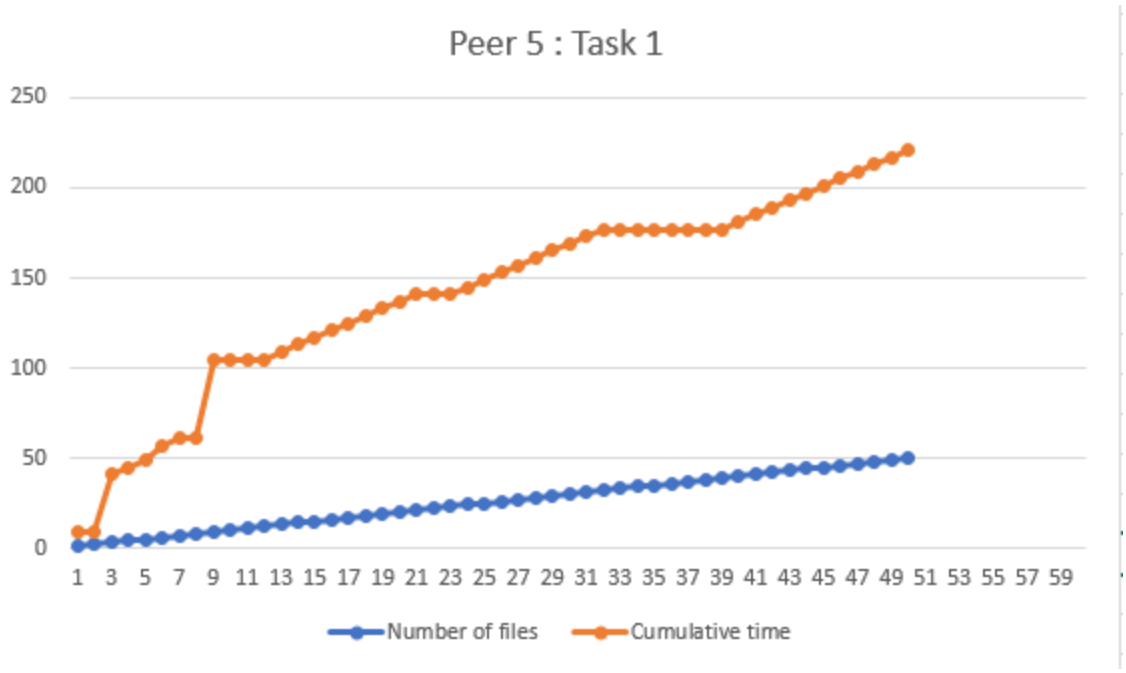
```
File F:/Arpita/IP/Project 1/RFC/Peer4/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 303
```



P5 : 60 files : 269 millisecond

50 files : 221 millisecond

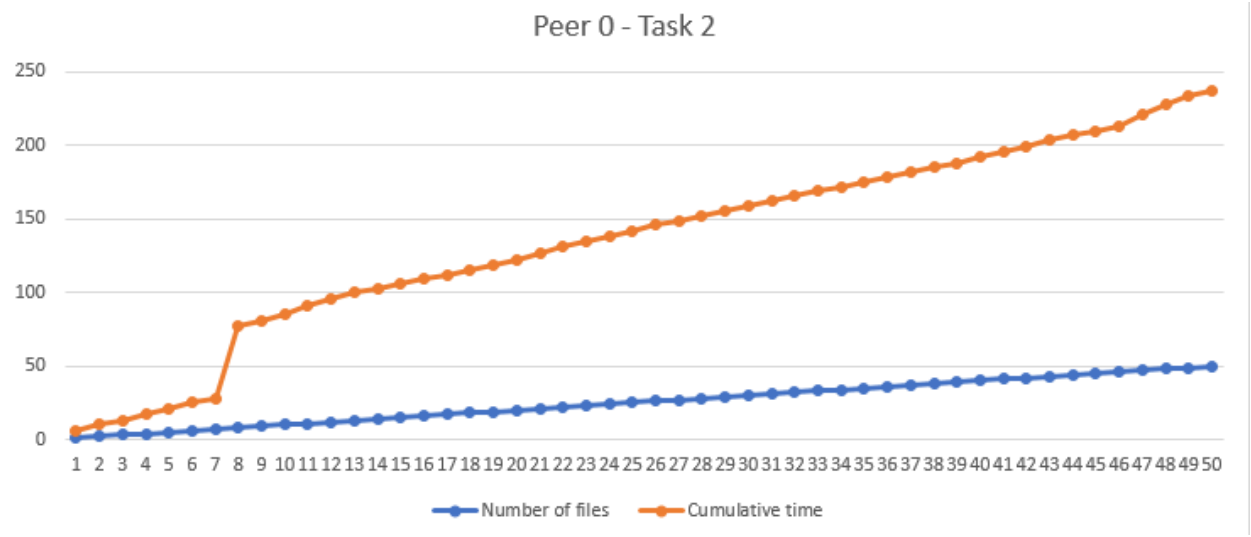
```
File F:/Arpita/IP/Project 1/RFC/Peer5/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 269
```


Peer 5 : Task 1
250
200
150
100
50
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59
Number of files
Cumulative time

TASK 2 : Peer-to-peer file sharing :

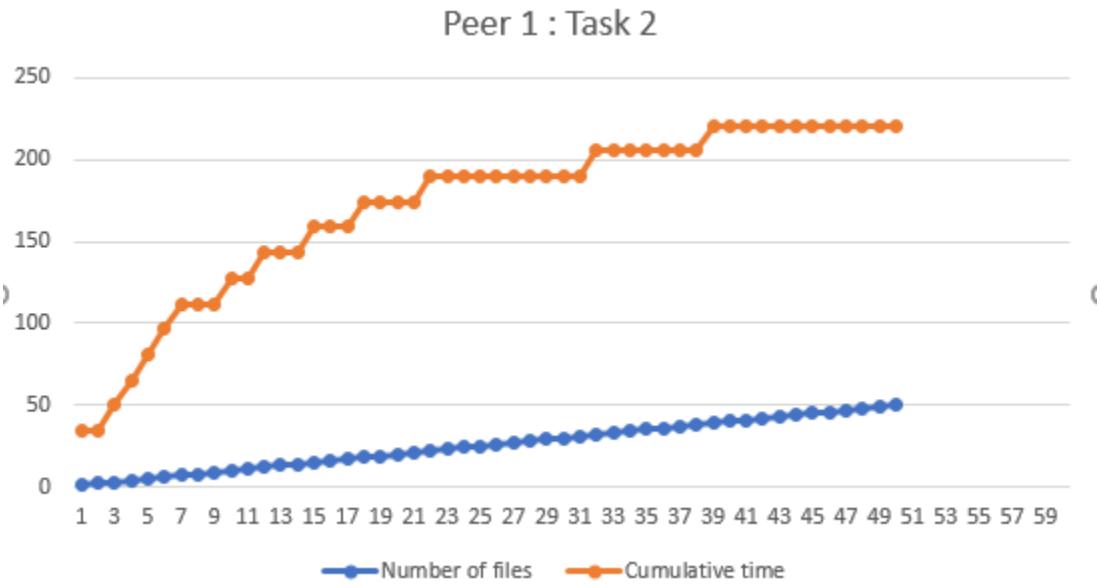
Peer0 : 50 files : 237 milliseconds

```
File C:\Users\Sumit\git\IP_Task2\Task1\Peer0/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 237
```



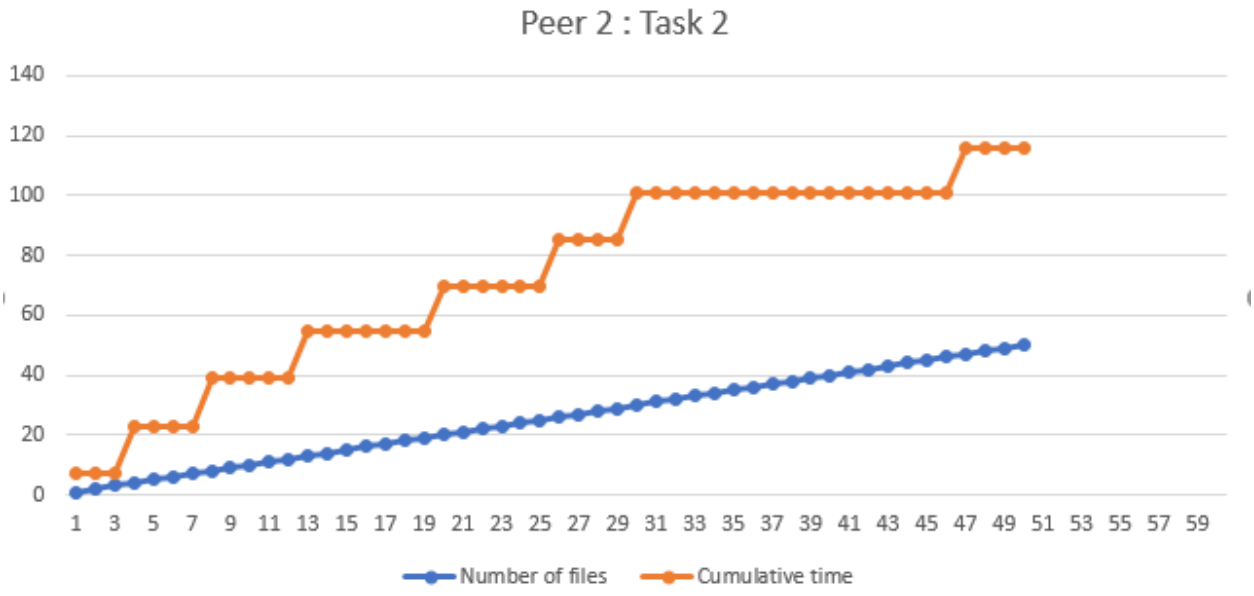
Peer 1 : 221 millisecond

```
File F:/Arpita/IP/Project 1/RFC/Peer1/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 221
```



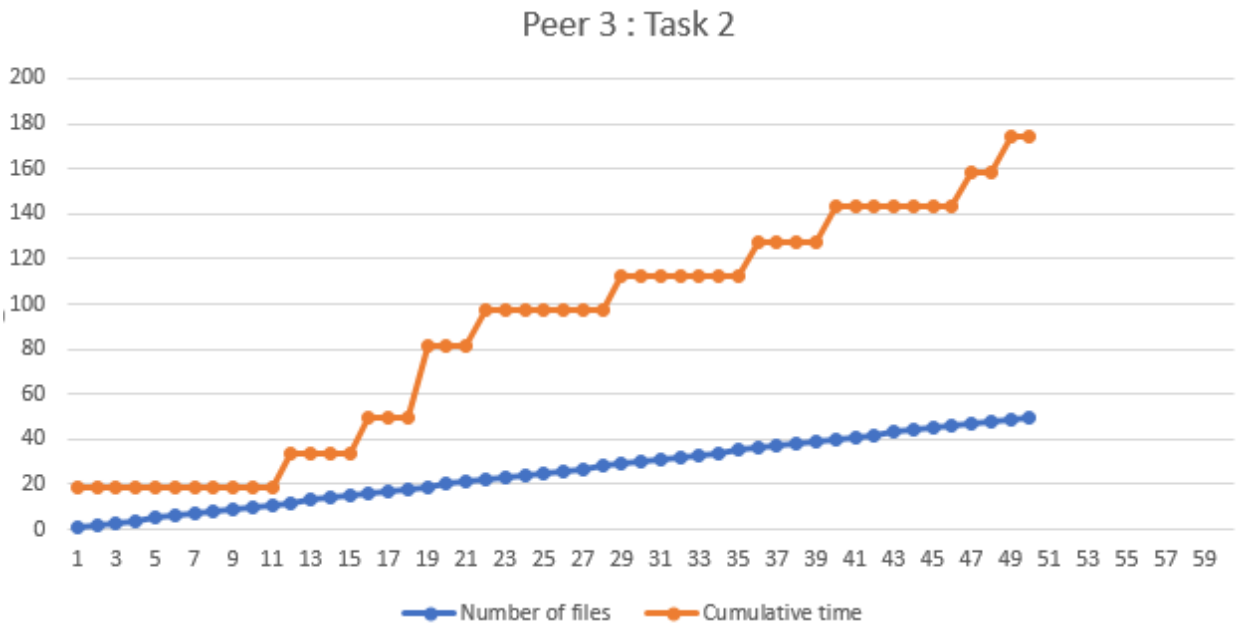
Peer 2 : 116 millisecond

```
26467 Value of C
File F:/Arpita/IP/Project 1/RFC/Peer2/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 116
```



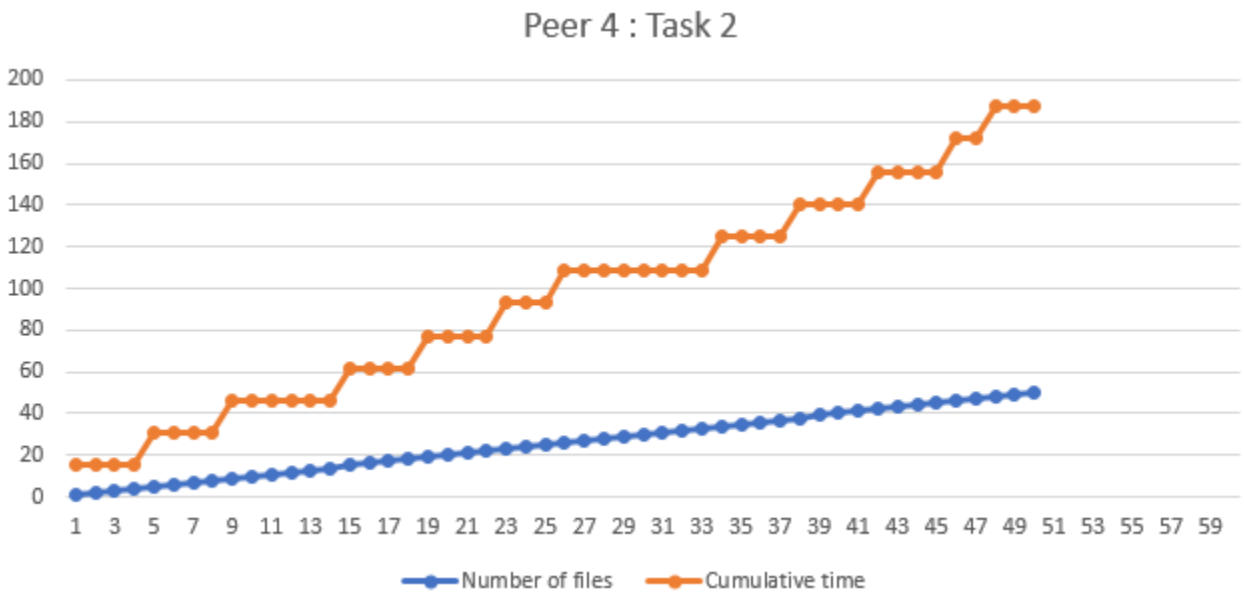
Peer 3 : 50 files →174 millisecond

```
File F:/Arpita/IP/Project 1/RFC/Peer3/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 174
```



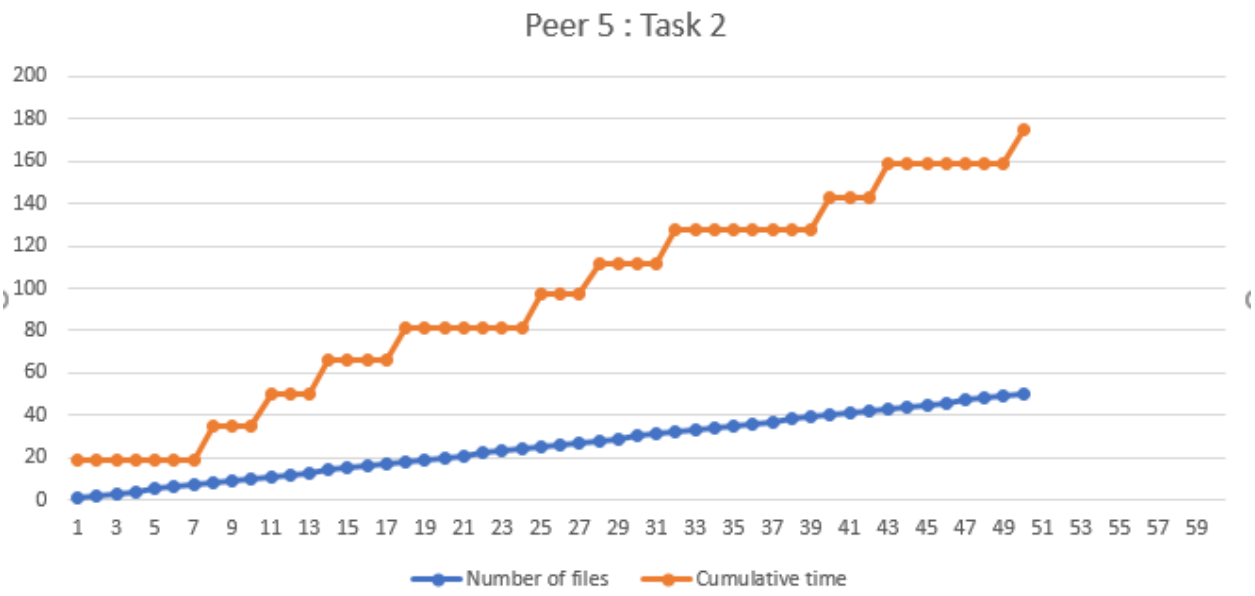
Peer 4 : 187 millisecond

```
File F:/Arpita/IP/Project 1/RFC/Peer4/rfc8269.txt.pdf downloaded (26467 bytes read)
Time taken in milliseconds : 187
```



Peer 5 : 175 millisecond

```
File F:/Arpita/IP/Project 1/RFC/Peer5/rfc8239.txt.pdf downloaded (27488 bytes read)
Time taken in milliseconds : 175
```



Conclusion :

As per the observation, the statistics clearly indicates that the Peer-to-peer file transfer is much more efficient than the centralized server file transfer.

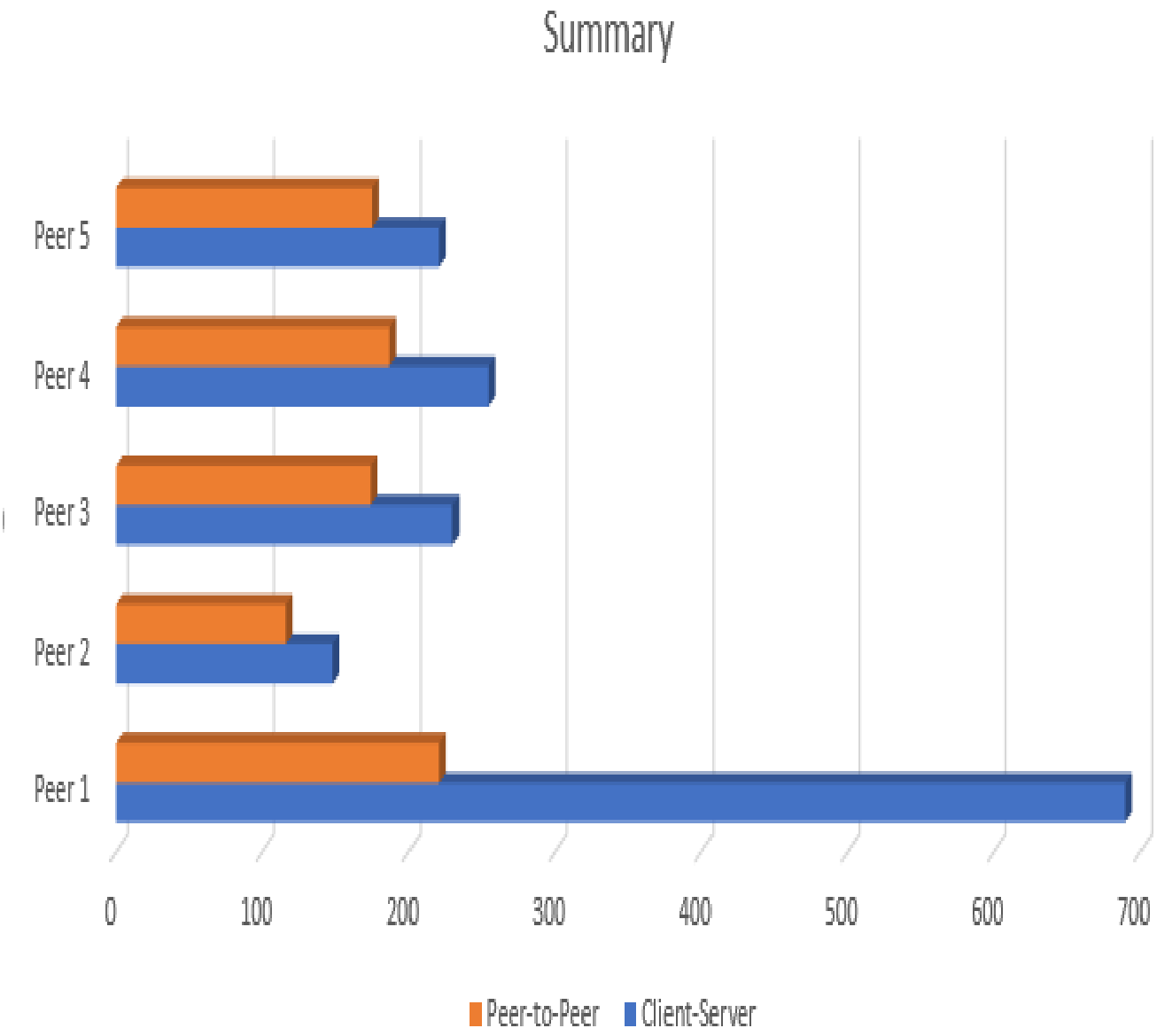
Let us discuss in detail regarding the best case and worst case in both the scenarios :

Centralized server architecture

- ➔ Best case can be considered when all the clients are not trying to access the server at the same time, therefore less time is taken to download 50 files.
- ➔ Worst case will be considered whenever all the peers are trying to access the server at the same time , thereby leading to congestion, hence the download time increases.
- ➔ If we consider OS concepts, 50 threads are spawned in case of Centralized server and the server must ensure scheduling between the mentioned 50 threads, thereby it takes certain amount of time to finish the task of the 50 threads.

Peer-to-peer architecture

- ➔ In the peer-to-peer architecture, best case can be considered when all the files are equally distributed among all the peers and all the peers try to access different peers at the same time. Therefore, total download time will be less than the time taken in Centralized RFC server.
- ➔ The worst case in peer-to-peer architecture is whenever a single peer contains almost all the files(like Task 1) , and all the other peers tries to access the files at the same time. In this case, the system will work as Centralized server architecture.
- ➔ If we consider the OS concept, different child threads are spawned in different machines, at maximum 10 child threads are spawned in Peer0, Peer1,Peer2,Peer3,Peer4,Peer5 respectively. Therefore the scheduling of the threads will be faster in case of different peers running on different systems.



This graph summarizes the total time taken to download the 50 files both in Peer-to-peer and Centralized File server.