CSC/ECE 573 – Internet Protocols
Course Project
Fall 2015

Peer-to-peer system with Centralized Index based on UDP

Submitted on 12/2/2015

Team members:

Anuja Supekar

Arjun Karat

Devan Harikumar

Sreekanth Ramakrishnan

1. INTRODUCTION

Objectives:

- To design a peer-to-peer system with centralized index based on UDP.
- To configure TCP and UDP socket interfaces.
- Create a server that can listen from and communicate with multiple clients simultaneously using TCP connection and act as a centralized index CI, that stores information about all peers in linked lists.
- To implement a client-server system based on TCP and client-client system based on UDP.
- Computing checksum, creating packets and sending them on transport layer using UDP.
- To transfer RFCs between clients using Simple File Transfer Protocol.
- Implement Go Back N protocol at the sending and receiving sides of the peers to make the transfer reliable.
- To implement error generation to test the working of Go-Back N protocol.

Background Information

- The first step was to design a server that will listen to multiple clients simultaneously. The server port is always active and every client that connects to the server on its port is spawned a new thread to talk to the server with. The server listens to its upload port and then assigns a new thread to each new joining peer which handles all the communication with that peer.
- The challenge was management of multiple processes. For instance a server such as TCP server is destined to receive multiple requests from different peers at the same time. A UDP server, on the other hand should be able to handle multiple data and ACK packets at the same time.
- The data format of the files such as .pdf and .txt were all tested. While .txt was comparatively easy to implement, .pdf was a bit difficult to work with on account of its difficulty to repeatedly edit. It was overcome using a different encoding in the source code.

SOFTWARE DESIGN

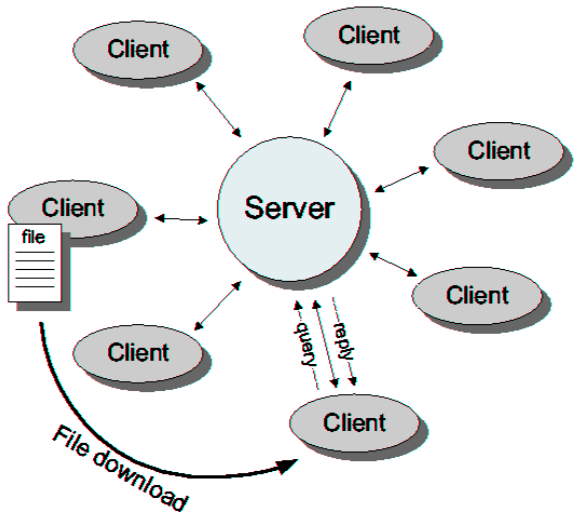


Fig. 1 System Functionality

The File sharing application is developed in python 3.5 and is tested to be compatible with all 3.x versions. As of this moment, the software deals with pdf files. Expanding its compatibility to other extensions is a very small step away. It is not implemented as it does not fall under the purview of this particular project under consideration.

Like any classic Peer to Peer FTP application, this application works by contacting a central server and getting information about the location of a desire file. Rather than downloading it from the said server, a connection is made to the peer that holds the file and subsequently it is downloaded from that file server (peer).

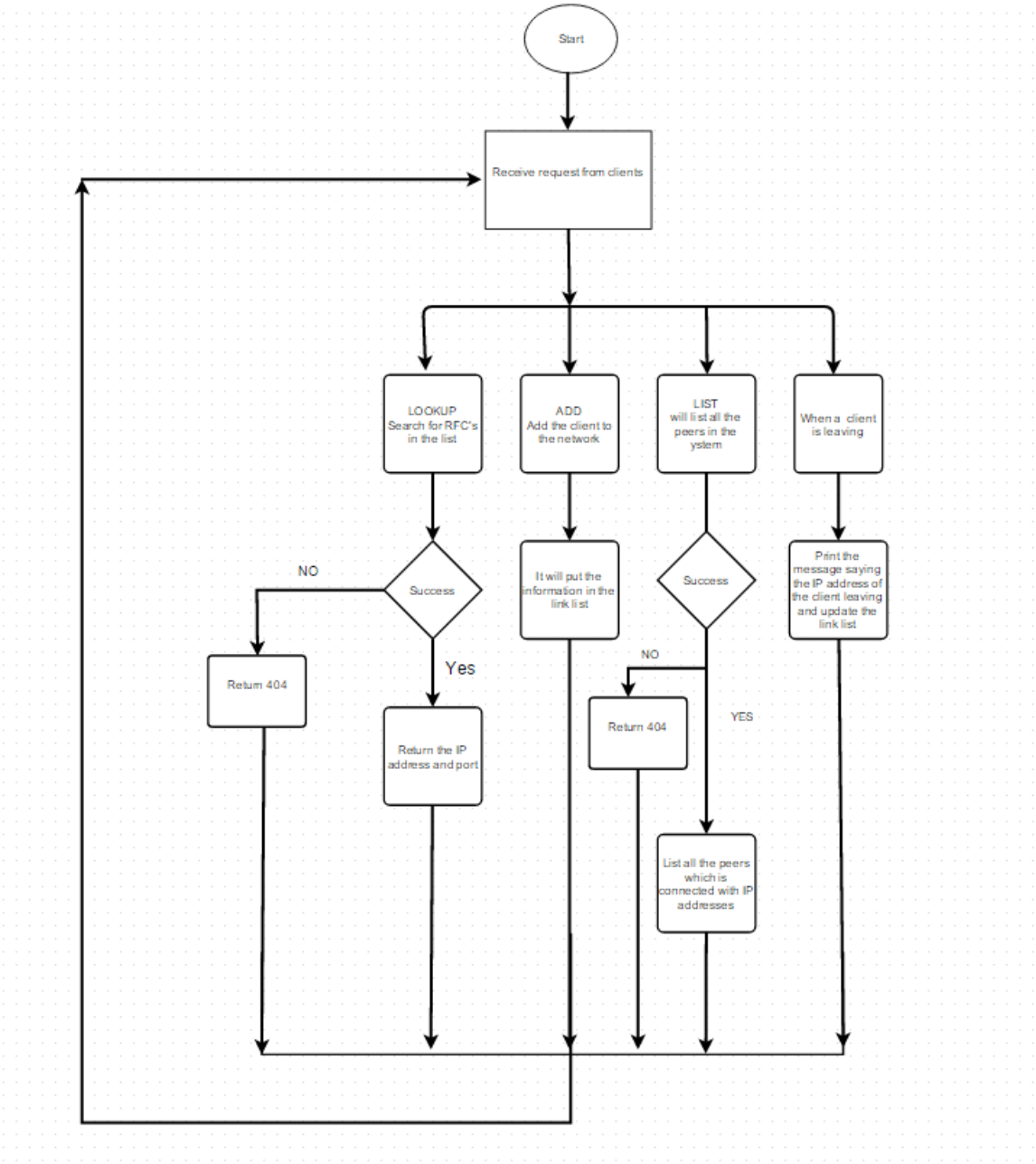
The robustness of the application can be tested by varying factors such as probability, window size to extreme (and even impractical in the real – life scenarios) values. This particular application is tested for file sizes up to tens of MBs, with window sizes varying from 1 to 100, RTT varying from a few milli seconds to 1 second, MSS varying from 100 to 2000 bytes and probability threshold for packet drop as high as 0.9

Contribution Breakdown

Task	Anuja Supekar	Arjun Karat	Devan Harikumar	Sreekanth Ramakrishnan
Algorithm Design	17.5	17.5	35	30
Coding	20	20	35	25
Debugging	20	25	25	30
Report preparation	35	30	17.5	17.5
Requirement Analysis	30	30	20	20

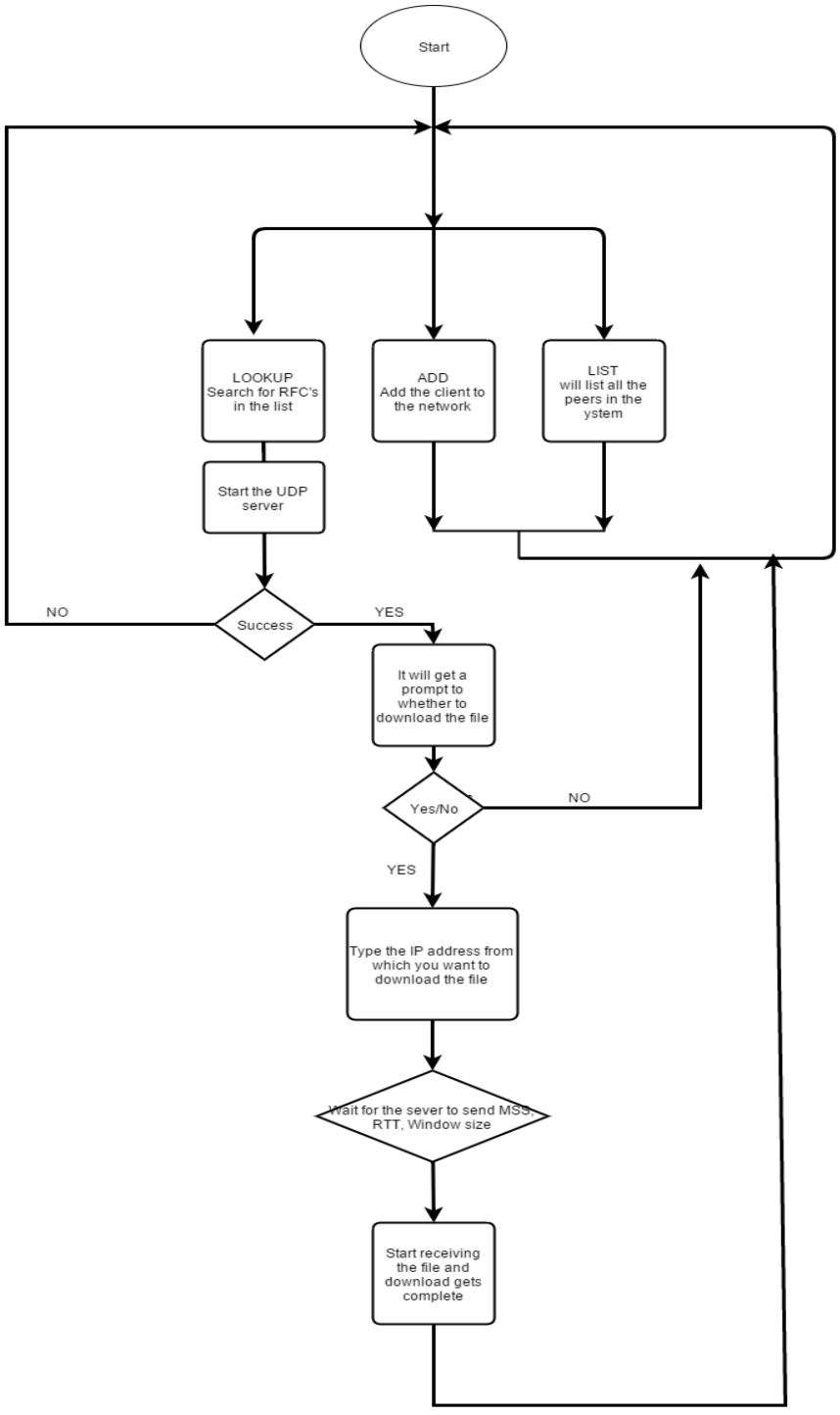
FLOWCHARTS

1. TCPServer



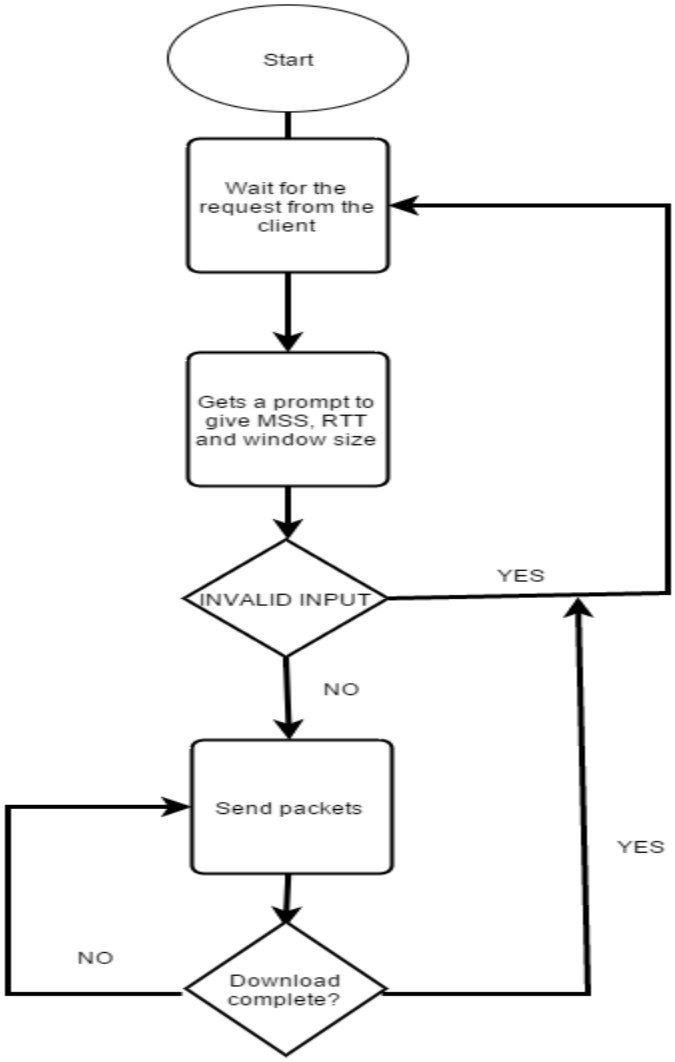
This flowchart gives the flow of TCPServer. It handles the ADD, LOOKUP and LIST commands from the peers and maintains the records of peers and RFCs in linked lists.

2. Client



This flowchart gives the working of Client.py which handles the TCP connection between TCPServer-client and UDP connection between client-client. It handles receiving side of Go-Back N protocol. The user-controllable parameters are request for download, ip address and loss probability threshold. If the user says Y, it has to input ip address from the list of ip address and loss probability between 0.0-0.99.

3. UDPServer



This flowchart gives the flow of UDPServer. It handles the transfer of RFC to client peer and the sending side of the Go-Back N protocol. The user controllable parameters are MSS,RTT and window size. Enter a higher value for MSS for a higher file size. RTT is in milliseconds and optimum values can be 100ms or above. Very low RTT values give more timeouts and should not be used.

2. IMPLEMENTATION

Programming language used: Python

Setting up the application

1. Install python 3.* in all of the participating workstations.
2. Copy the .py files into the same folder in all the workstations. TCPServer.py, UDPServer.py, Client.py, Lists.py, RFCLists.py and P2PUtils.py
3. Connect all of the workstations to the **same** network.
4. From ipconfig, find and note the ip addresses of all the workstations.
5. One of the above work stations is the CI TCP server. Open TCPServer.py in that work station and update its IP address to what you found in step 4.
6. Run TCPServer.py there.
7. All the remaining workstations are clients. Every client is also a UDP File server. So open, Client.py and UDPServer.py in each one of them and edit the IP address of CI server to that of step 5 and edit the IP address of client socket and file server socket to that of this machine found in step 4.
8. Run UDPServer.py and Client.py in each of the work stations except the server.

IP configuration

Please refer to the readme file attached along with the project zip file.

Content of source code files

The entire code is divided into 5 files- TCPServer.py, Client.py, UDPClient.py, P2PUtils.py and Lists.py

1. TCPServer.py

This file implements the central server. It is the starting point for the system. It is the Centralized Index CI which contains information about all peers.The server holds 2 linked lists.

First list "list_of_peers" contains the information of the peers. Each item of the linked list of peers contains two elements:
1. IP address of the peer, and
2. Port number (of type integer) to which the upload server of this peer is listening.

Second linked list "list_of_rfcs" contains the information about the RFCs. Each item of the linked list representing the index of RFCs contains these elements:
• the RFC number (of type integer),
• the title of the RFC (of type string), and
• the IP address of the peer containing the RFC (of type string).

The server is listening for connection requests from clients on its default port 7734, Fig. 2. The server implements **multithreading**. It starts a new thread for each new peer that joins the system and associates all the future communication with the peer with the newly created thread. The communication between server and the client is based on TCP.
First, the server initializes the 2 empty linked lists. The server is listening continuously on its default port for any incoming connection requests.

After connection is established, the server waits for an ADD, LOOKUP or LIST command from the client.

As soon as it receives a request, it checks if the request is valid or else it displays undefined. If the request is a incomplete ADD request, the server displays "check the command". If the request is valid ADD request, Fig. 3, then the server checks whether the requesting peer is already added in the linked list list_of_peers. If the peer is already added, then it ignores the request. If the peer is not present in the linked list, then it adds the new peer information and sends 200 OK response, Fig. 3.
It also checks whether the particular RFC added already has an entry in the linked list list_of_rfcs. If the RFC is not present for that particular peer, then it stores the new RFC information in the linked list list_of_rfcs.

When the server receives a LOOKUP request, Fig. 5, it first checks whether there are any elements in the list_of_rfcs. It the list is empty, it displays response 404 Not found. If the list is not empty, then it searches for the requested client, matches the ip address and displays the information about the RFC.

If the server receives a LIST request, the server checks whether the list_of_rfcs is empty. If not empty, it displays all the elements in the list_of_rfcs.

Following is the brief description of the method used in the file,

1. **threadPerClient(peersocket,addr)**

- The method is called in a new thread for each incoming client. This remains alive until the client does not leave the system.
- It handles the ADD, LOOKUP, LIST requests by the client. The linked list

manipulations for the client joining , leaving,
- The ADD of the index of a file and query for a particular file by the client is handled

2. Client.py

This file implements the client side of the client-server communication and receiver peer in case of SFTP. It implements the receiving side of the Go Back N protocol and error generation for testing the Go-Back N protocol.

The client establishes a TCP connection with the server and requests an ADD, LOOKUP or LIST. Depending upon the request, the TCPServer responds.

The incoming new peer adds its information by an ADD request, Fig 2.

The client who wants to receive a particular RFC, gets the information about the peer having the RFC by a LOOKUP request, Fig. 5

The server then asks whether the client wants to request the file. If yes, the GET request is generated internally and sent to the server peer. If there are multiple peers having the same RFC, the user is asked to choose a particular IP address, Fig 6.

After receiving the IP address of the server peer from the TCPServer, the client peer binds to the upload port of server peer and establishes a UDP connection. The input for the packet loss probability is taken from the client peer from the command line, Fig. 7

The client peer then sends the GET request internally to the server peer, and is now ready to receive the file. In order to implement error generation we have introduced packet loss probability. While receiving the file in bytes, the client peer generates a random number r between 0.0-1.0. Also the checksum in each received segment is checked for any errors.

If the random number generated is less than the packet loss probability and the checksum at sender and receiver is not equal, then the current packet is discarded by the client peer and a message indicating packet loss and sequence number of lost packet is displayed in the console, Fig. 7

If random number generated is greater than packet loss probability, then the packet is accepted and an acknowledgement is sent and the window is slided to the next expected sequence number. The ACK sent consists of the following 3 fields,

• the 32-bit sequence number that is being ACKed,
• a 16-bit field that is all zeroes, and
• a 16-bit field that has the value 1010101010101010, indicating that this is an ACK packet.

The packet loss probability value is received from the command line.

Following is a brief description of the methods used in the file,

1. create_request()
The method reads a request from the console and formats it by replacing newlines with '|'

2. get_ip_addr(request)
Extracts the IP address of the peer from the request

3. receive_file()
The method is called once the file request is send to another peer. The client listens at a fixed port for the packets. When a packet is received it checks for type and if its in correct sequence and without any checksum error an acknowledgement is sent. Method runs till the download is complete.

4. send_ack()
The method creates an ack packet with expected sequence number and sends it to the file server.

Following steps are to be performed while running the application:

1. Adding a file to the server's index. This is done by the clients as an ADD request shown below. Please note this is a sample.

   ADD RFC 2386 P2P-CI/1.0
   Host: thishost.csc.ncsu.edu
   Port: 5678
   Title: A Proferred Official ICP

This command has to be typed in the client console.

2. Listing all the RFC's the server knows of, is as follows. Please note this is a sample.

   LIST ALL P2P-CI/1.0
   Host: thishost.csc.ncsu.edu
   Port: 5678

This command has to be typed in the client console.

3. Looking up and subsequently downloading a specific RFC can be done as follows. Please note this is a sample.

LOOKUP RFC 2386 P2P-CI/1.0
Host: thishost.csc.ncsu.edu
Port: 5678
Title: Requirements for IPsec Remote Access Scenarios

This command has to be typed in the client console. This command prompts for the following things.

Do you wish to download the RFC? (Y/N)

Enter Y or N. If you enter N, there is no further action. If you enter Y, you get the following prompt

Choose an IP address<Client console>

Enter any IP address from the list received after doing LOOKUP. After this, following prompt is received.

Probability Threshold Value <Client console>

Enter a value between 0 and 1

Good practice: Enter values from 0.0 to 0.99 but for better results and quicker response, lower the p value the better. Also the entered value should be a float, else it displays invalid input. Since this relies a lot on random number generators, we cannot be cent percent sure to state that the performance will be better for lower values of p. This is to check the retransmission policy of the application in case of packet drops. There is an internal random generated value which if below the above said threshold, the packet will be dropped.

3. UDPServer.py

The UDP server is the file handling the server peer. It communicates with other client peer with UDP. The server peer always listens on its upload port any incoming request for an RFC.

As soon as the server receives a GET request from a client peer, the server peer sets the Maximum Segment Size MSS of data bytes in the packet, the round trip time RTT required for transfer and the sender window size. **These 3 parameters- MSS, RTT and sender window size are obtained from the command line,** Fig. 6

The sending side of the Go Back N protocol is implemented at the server peer. The method rdt_send() is used to provide reliability as the unreliable UDP protocol is used.

The RFC is in .pdf format. It is divided into bytes of data. These bytes are merged into datachunks of MSS bytes. The packets are created having following 4 fields,

- a 32-bit sequence number,
- a 16-bit checksum of the data part, and
- a 16-bit field that has the value 0101010101010101, indicating that this is a data packet.
- Data

These packets are sent and the server peer waits for an ACK from the client peer. As soon as it receives the ACK, it slides the window. There is a timer implemented. If the ACK is not received till timeout, there is a message displayed in console <timeout, sequence number: > and all packets in the window are sent again.

After the last packet is sent, there is an End Of File(EOF) packet sent to indicate the client peer that the transfer of file is complete.

Following is a brief description of the methods used in the file,

1. file_sender_thread()
The method extracts the RFC no requested for from the request and calls rdt_send()

2. send_packet(pkts)
Send the pkts to the client using sendto() method.

3. make_packets(fileData)
The method builds packets containing type, checksum and sequence number for each MSS chunk of data. These packets are converted to bytes and an array of the bytes of packet is returned.

4. udt_send(packetbytes)
The method takes in the list of packet in bytes and continuously send the packets to the clients until end of file is reached. Packets are send up to the maximum window size and the thread waits there while the window size is increased by another thread whenever an acknowledgement is received.

5. timer(msecs, seqno)
The method handles the timer for the send packets and whenever there is a timeout the packets are resend.

6. rdt_send(rfcno)
The method is the starting point of file send. A file with the rfcno taken as input to the function is loaded.

The file is divided into chunks of MSS , put into a list and returned. If a file is not found a 404 error is returned.

7. resend_thread(pkts)

The method handles the resend when the timer finds a timeout. If the ack does not have the appropriate sequence number the window is slid back.

If the sequence number reaches total number of packets an EOF is set.

8. ack_thread(pkts)

The method waits for the ack from the client. When an ack is received, the sequence number is checked for order and the window is slid until it reaches the end of file

Following prompts are received in the console when the client peer requests for downloading an RFC.

Maximum segment size <UDP File server console>

Enter a value for the maximum segment size.

Good practice: Enter a higher value for a higher file size. There is no upper bound on allowed value here, other than (trivially) of course the file size.

Round Trip Time<UDP File Server console>

Enter a value for the round trip time.

Good practice: A few hundred milliseconds(100-500). Lower this value, faster the file transfer. Hence we are tempted to give a lower RTT, but the chances of packet drops are more when RTT is low. Of course the application works for very small values and very large values or RTT but the file transfer maybe slower than it has to be.

Sender Window Size <UDP File Server Console>

Enter a value for the sender window size.

Good practice: Higher window size helps in keeping the pipe full but at the same time in case of a packet loss the entire window needs to be resent. One should keep that in mind while trying this application.

4. P2PUtils.py

This file contains the checksum generation and correction methods. This file is used by both the Client.py for correcting the checksum and by UDPServer.py for generating the checksum.

5. Lists.py

This file contains the two linked lists list_of_peers and list_of_rfcs. It is used by the TCPServer to update the information of new peers joining the system.

4. RESULTS AND DISCUSSION

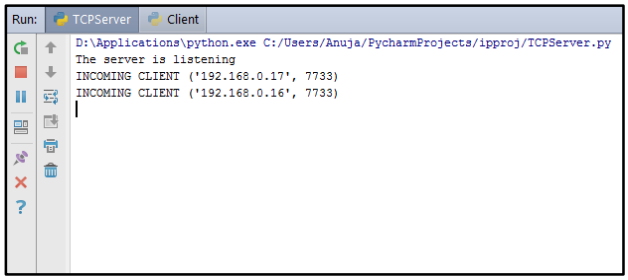


Fig.2 TCPServer console: incoming peer connections

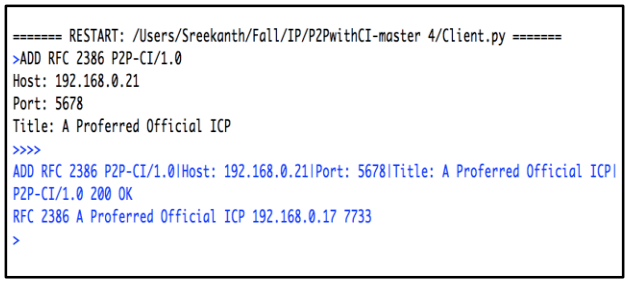


Fig.3 Client console: ADD request and response

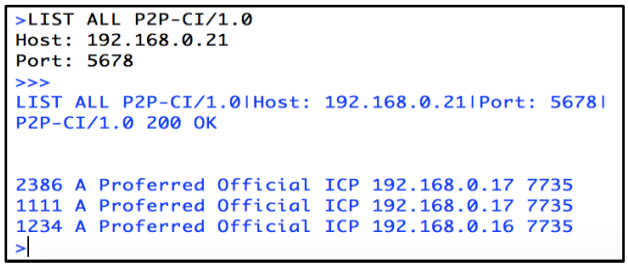


Fig.4 Client console: LIST request and response

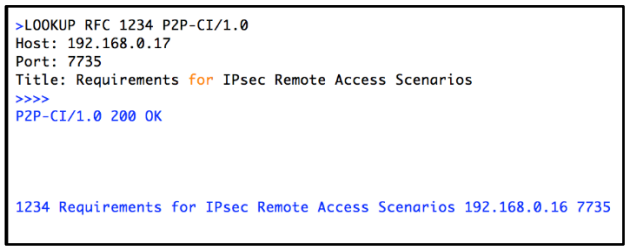


Fig. 5 client console: LOOKUP request and response

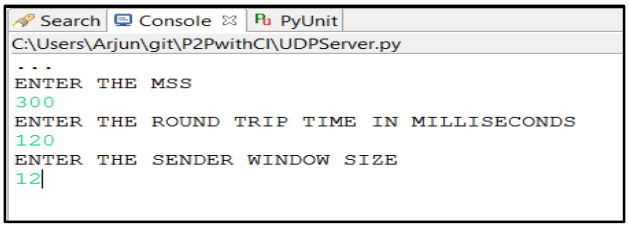


Fig. 6 UDPServer console: input MSS, RTT, Window size from command line

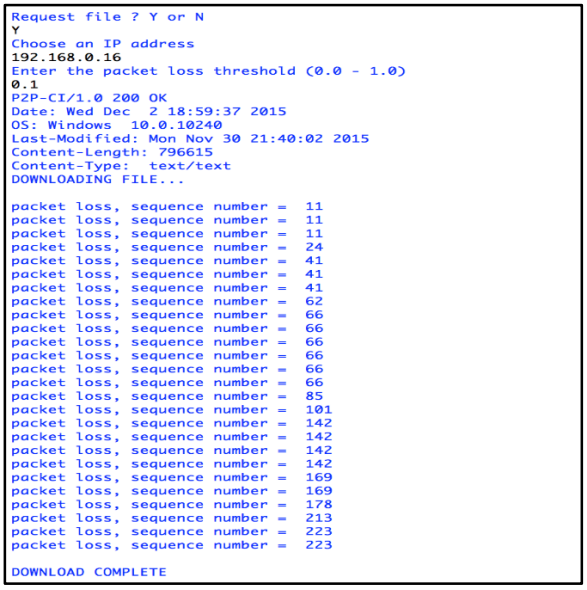


Fig. 7 Client console: Downloading RFC

It is worth noting that the packets lost (due to network errors, checksum errors) at the receiver side are a subset of those shown at the sender side because once sequence number k is found to be lost, the sender sends everything starting from k onwards. The file is received by the requested peer and it is AUTOMATICALLY added to the server's index. Hence if A wants a file from B and C wants it in the next iteration, C can request the file directly from A or B. This is in keeping with the classic peer to peer file sharing architecture of applications such as Bit torrent etc.

5. REFERENCES

1. www.python.org
2. www.tutorialspoint.com
3. www.stackoverflow.com
4. Computer Networking- A Top Down Approach – James Kurose and Keith Ross.
5. Computer Networks (5th Ed.) Tannenbaum et Al.
6. www.dmst.aueb.gr/dds/pubs/jrnl/2004-ACMCS-p2p/html/AS04.html