Delhi Technological University

Department of Computer Engineering

Subject: **Computer Networks**

Topic of the project: **Implementation of a chat-based file transfer system using socket programming**

B. TECH (SEMESTER VI)

SUBMITTED BY:

Prithish Kumar Rath (2K18/CO/262)

Nilesh Nishant (2K18/CO/235)

Introduction

Objectives:

- To create a chatroom based file transfer application, where peers can come together in private or public chatrooms to pass messages to each other as well as enquire about and transfer files.
- A client connects to a server and logs in to an existing chat room or create a new one. Server facilitates all message passing, and informs the client about peers already inside the room, and existing clients get information about other clients who join or leave a chatroom.
- Clients can use messages starting with @<username> of peers in chatroom to unicast messages, or '@all' to broadcast to all peers in room.
- To implement chat based file transfer where:-
    a) A client can send an '@all' message to all peers asking 'whohas' a particular file.
    b) Clients that have the requested file respond back so that the requester can select a peer from the responders and get the file directly.
    c) File transfer is a UDP based peer-to-peer transfer that doesn't involve the server for the sake of performance.
    d) UDP transfers are implemented with Go-Back-N protocol for reliable transfer where N is configurable.
- A peer can listen to user inputs and server messages at the same time over and above sending/receiving multiple files at the same time.
- A peer can refuse to send a file when it reaches its max parallel uploads value.
- A peer can disable sharing individual files or disable sharing altogether, all via chat. Share - disabled files will 'not be found'.
- Chat rooms can be public or password-protected private rooms.
- A peer can ask the server about the peers in the chatroom, the different chat rooms that are available, and view the password for the chat room.
- A user input console for the server can broadcast messages to all clients, or kill all clients to exit the system.

Background Information & Working:

Firstly, we will have a server that is always running and listening on a pre-assigned set of TCP ports. Multiple clients connect to the server which handles all clients in parallel. The user is able to create or join an existing chat room on the server. Inside the chat room, the client will be able to chat with fellow peers, or the server (typically to ask the server some details). They can also broadcast messages to all connected peers. For file transfer, each client has a folder in his home directory called folder. This represents the file stash of a particular client. When a client wants a particular file, he/she sends out a broadcast message with the key word 'whohasfile' and the name of the required file. Different fields in a message need to be delimited using a '|' escape character. Clients that get this message will check if he has that file in his local folder and if sharing is enabled for that file. If yes, the client will reply. So the requester will now receive a list of all the peers who have this file and are willing to share. The user manually chooses one of the peers and sends a unicast-ed message with 'getfile' command. The peer will establish a UDP connection and send the file to the requestor. We have tested and verified that we are able to transfer .txt, .pdf, .mp3, .jpg etc.

Software & Testing:

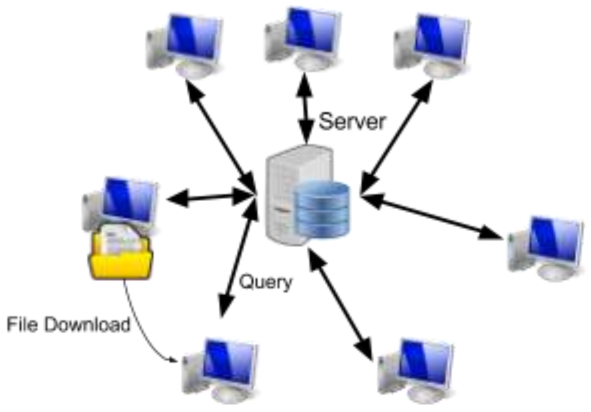


Fig 1. System Functionality

- Check throughput by varying N in Go Back N protocol to see effect of window sizes.
- Test all possible Murphy's law scenarios, like bad message formats, malicious users such that the system doesn't fail due to one client.

Design

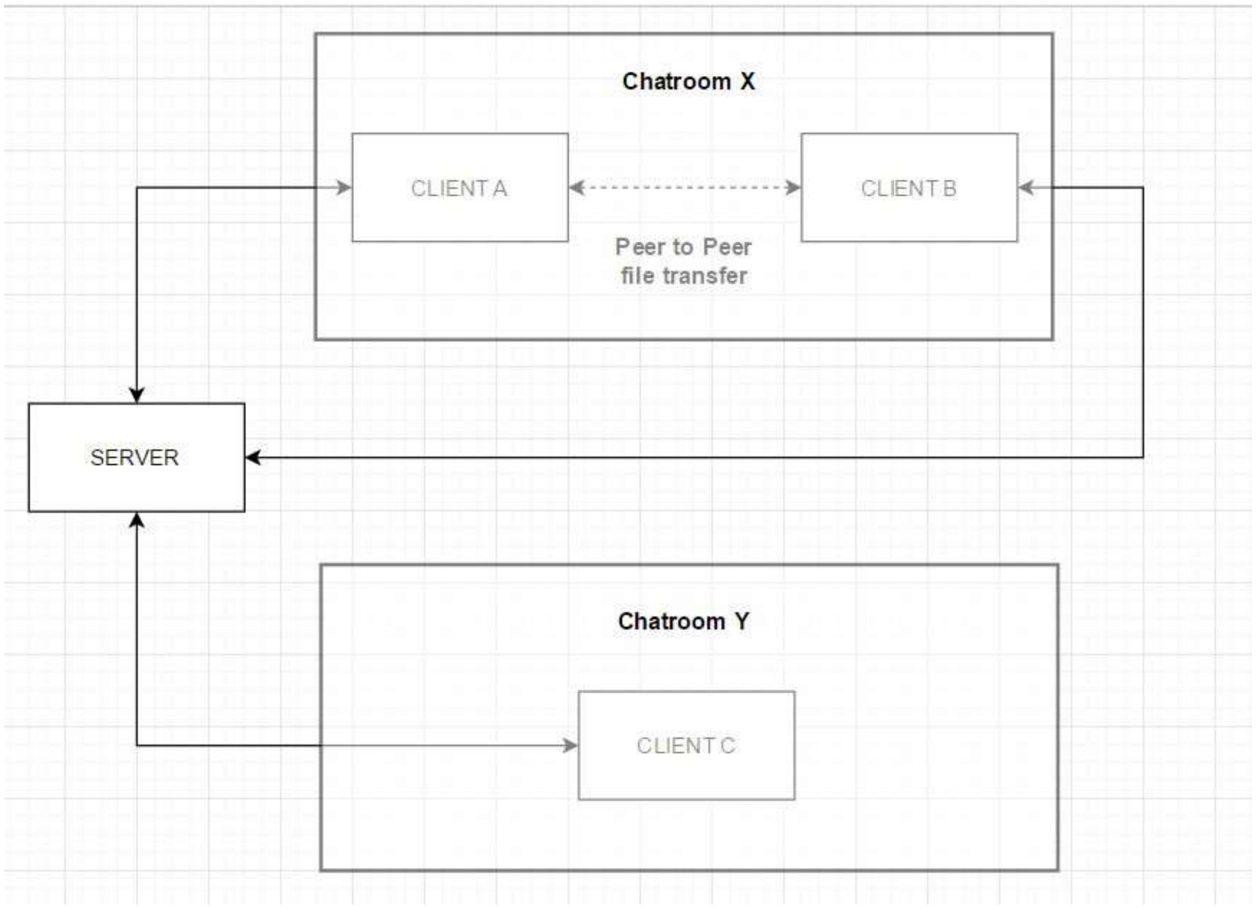


Fig 1. Block diagram of two Chatrooms X & Y. Clients A & B belong to Chatroom X & Client C belongs to Chatroom Y. Clients A & Client B will be able to exchange files with each other. They will also be able to talk to each other as a part of the chat room. Chatroom Y has one client, which can talk to the server.

Server to Client Algorithm:

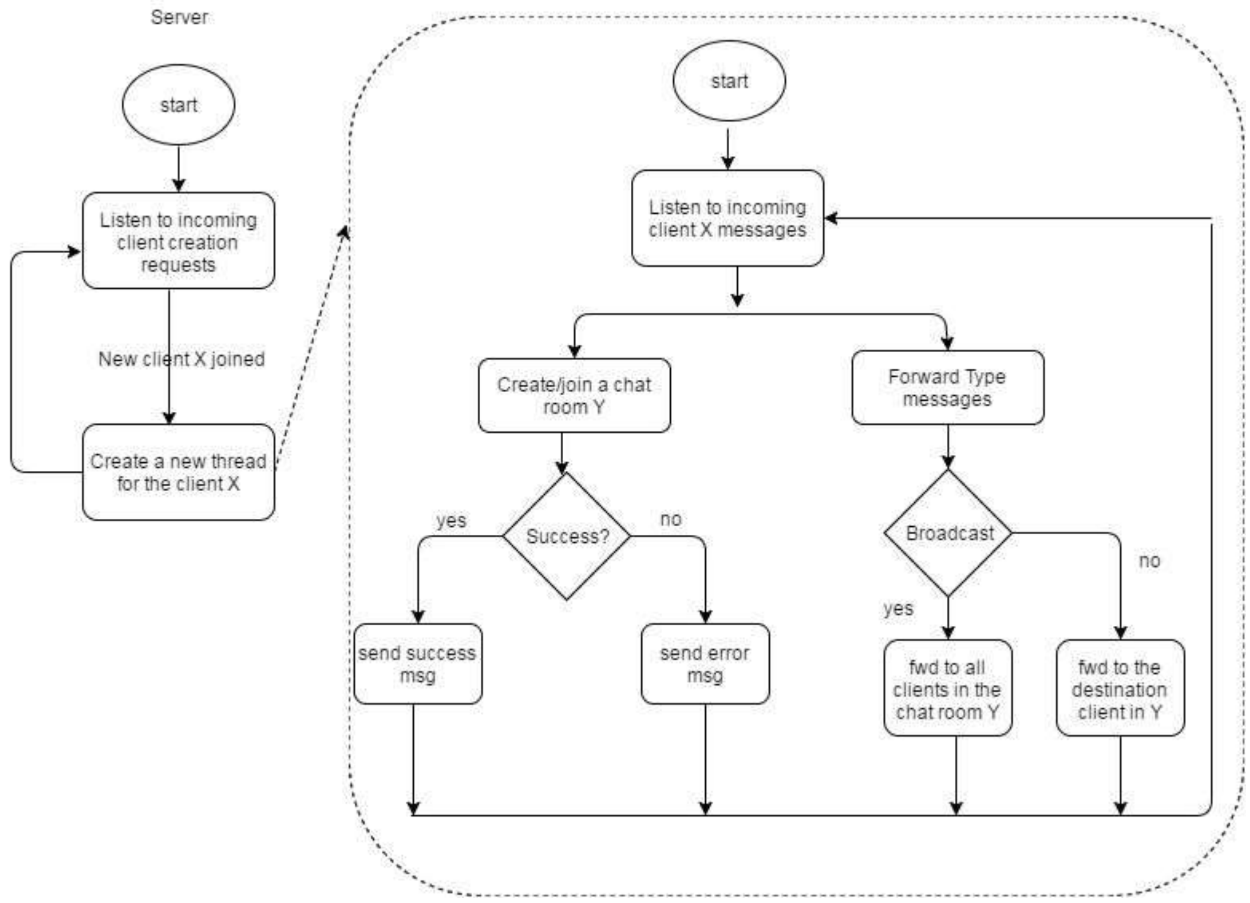
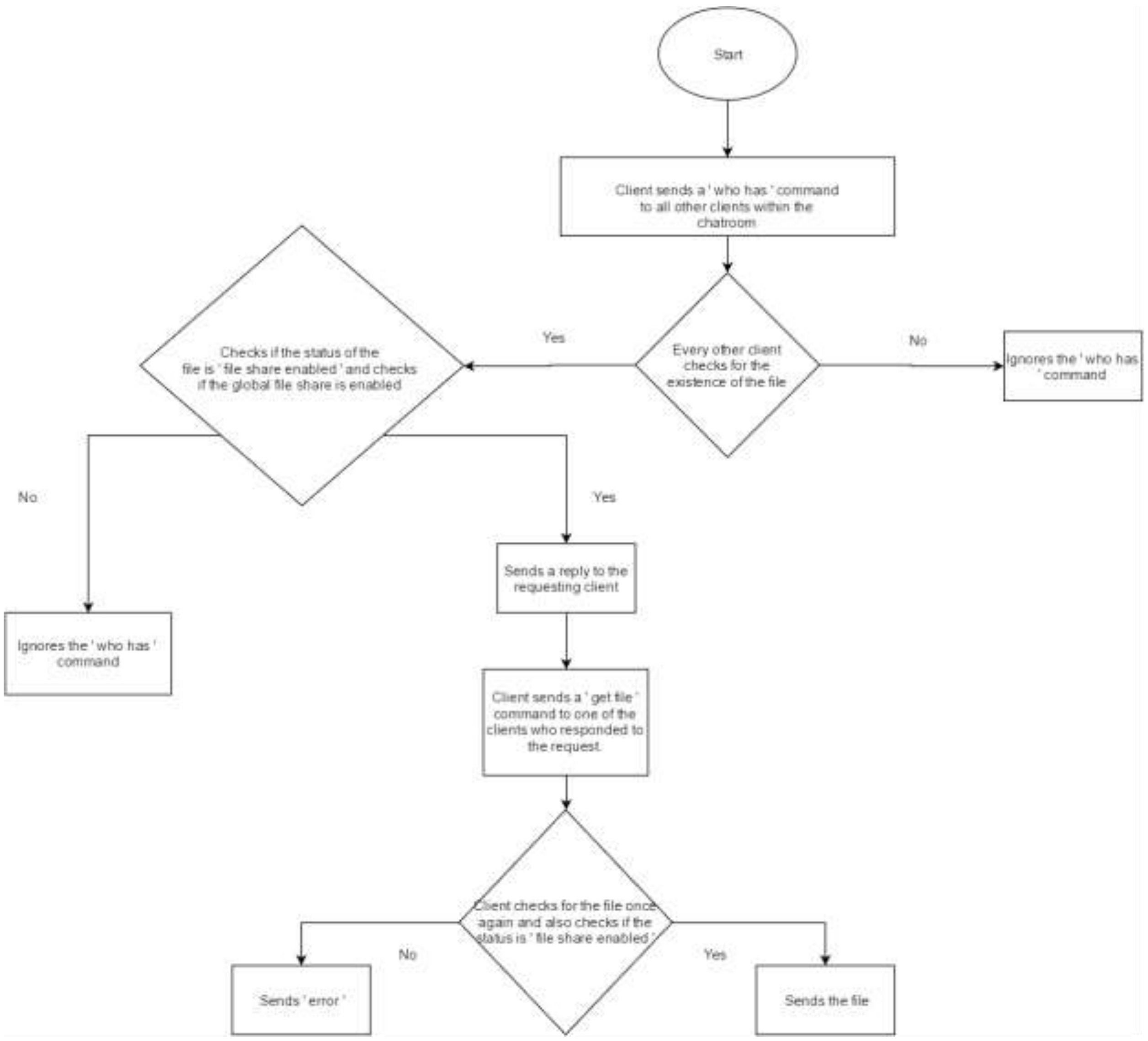

Server
start
Listen to incoming client creation requests
New client X joined
Create a new thread for the client X
start
Listen to incoming client X messages
Create/join a chat room Y
Forward Type messages
yes
Success?
no
Broadcast
no
yes
send success msg
send error msg
fwd to all clients in the chat room Y
fwd to the destination client in Y

Fig 2. Flow Chart depicting the interaction between the Server and the Client in a general scenario. The client has several options available to him, explained in more detail in the code. He interacts with the server and fellow peers using these functions.

Peer to Peer File Transfer Algorithm:



Start
Client sends a ' who has ' command to all other clients within the chatroom
Every other client checks for the existence of the file
Yes
No
Ignores the ' who has ' command
Checks if the status of the file is ' file share enabled ' and checks if the global file share is enabled
No
Yes
Ignores the ' who has ' command
Sends a reply to the requesting client
Client sends a ' get file ' command to one of the clients who responded to the request.
Client checks for the file once again and also checks if the status is ' file share enabled '
No
Yes
Sends ' error '
Sends the file

Fig. 3 Flow Chart depicting the detailed conversation between two clients in a chatroom. Client sends a probe to find out who has the file. Client requests a file and then the requestor will transfer this file via UDP connection. This is peer-to-peer.

Implementation:

Programming language used: Python 2.7.11
Setting up the Application:

1) Install Python 2.7.* in all the participating devices (both the server and the clients).
2) Copy all the python files in the given .zip file into the same directory of all the devices we want to implement file transfer on.
3) Next, make sure that all the devices are connected to the same local network.
4) Check the device we wish to configure as the server. Obtain the IP address of this device using the ifconfig command.
5) Start the server application on this device by running the server.py python program.
6) Next, run the client as follows.
7) sudo ./client.py -h --display help
8) sudo ./client.py -w 16 --run with window size 16
9) The options made available to us are:-

Argument	Description
-h --help	Print help options
-s --share	0/1 to clear/set global file share (default 1)
-p --parallel	Int argument to define parallel connections (default 2)
--ip	Server IP (tries 0.0.0.0, 127.0.0.1 by default)
--port	Server Port (50000-50009 by default)
-w --window	Window Size for Go-Back-N (default is 16)

IP Configuration:

Configuring IP addresses can be done by following the instructions provided in more detail in the zip file.

Chat options

Chat commands made available to the user:-

Command	Description
`@username	chat`
`@all	chat`
`@server	chat`
`@all	whohas
`@user	getfile
`@server	get_rooms`
`@server	get_peers`
`@server	exit`
`@me	setwindowsize
`@me	setshare
`@me	clrshare
`@me	setglobalshare`
`@me	clrglobalshare`
`@me	getsharestatus`

Commands made available to the server:-

Command	Description
`@username	chat`
`@all	chat`
`@server	chat`
exit	Kill all client connections and exit application gracefully

Contents of the Source Code File:

Our source code contains the following files (we have included only a brief description of the code. In detail functional explanation is provided inside the code as inline comments): -

server.py:

server.py is from where our program starts. It implements the TCP server to which clients connect in order to create the chatroom and start operations. It

has one class defined, the Server class. This class represents all server related info.
It has a list of client objects who are instantiated by the server. On deleting, objects are not deleted. Instead, we suspend the thread attribute for that particular object, so that they will be able to reconnect at some point in time.

It uses the dictionary data structure, because of its convenient use of key-value pairs, which is well suited to our requirements. It contains the functions to remove clients, list chat room names, binds a port number and listen on a TCP socket, broadcasts and gets an input from a user.

execute() is the main thread of server. It goes online and starts listening for an incoming TCP connection from a client. When a new client connects, it starts execution with a new thread. So each client connects to a new thread instance of
the server and hence, we can have parallel clients. We implement this multithreading feature to improve performance of the server.

One additional point of note is how the server binds ports . The server starts trying from port number 50000 if we are able to bind. If the bind to 50000 fails (because its already open or in use by another connection), we have the option of doing 10 iterations of this, i.e. we can have port numbers from 50000 to 50010.

client.py:

Client.py file implements the peer to peer file transfer part in the client side of the program. It runs on the device where we wish to implement the peer-to-peer file transfer.

It has one class, called the Client, which functionally stores all the information associated with a peer. It has functions to check a folder for a file, listen for file request queries and the main thread, execute() , which checks for the keyword 'getfile' in the chats a peer sends (henceforth referred to as the sender) to the receiver.

If such a keyword is present (considered to be the stimulus to initiate file transfer), we initiate a UDP connection by calling the udpclient.py function.

clientNode.py:

clientNode.py is the Python file that holds client related information. It has a clientNode class that holds client related information. It's initialized with the IP address of the client and the socket at the client used to communicate with the client. It has functions to connect to a server, create or join a chatroom, optional authentication functions and send messages to the server. It displays a central menu to the user where he can select various options made available to the client, like help, share, parallel, ip, port and window. These options enable the user to customize the implementation configurations itself. What each of them represent is explained in the code.

chatRoom.py:

chatRoom.py holds chat room specific information as a class. A chat room is identified by a chat room ID, randomly generated based on the server's room count history. It's initiated by the client that created the room, and by default, will have the client that created that room as a member.

One point of importance is that the first client, the one who creates the chatroom, can specify using the password functions, whether he wants the chat room to have a password.

It has functions to remove clients, get client list, get
peer info and broadcast message.

library.py:

It contains general functions which can be used across the scope of the project like client_send, client_recieve, send_data, send_ok, send_err etc. Implementing these functions as a common library enables us to reuse the code across the application and avoids the need to have separate functions for the client and the server. Function calls inside specific include files are explained in detail as inline comments to avoid confusing the user.

udpclient.py:

It contains the functions used by the requester, to initiate and respond to the UDP file transfer connection, that comes in from the udpserver. It has three functions: - udp_send, udp_recv and execute. The functions upd_send and udp_recv are common to udpclient.py and udpserver.py.

udpserver.py:

It contains the functions which are utilized by the peer sender to initiate an UDP connection to the requestor. It has nine functions udp_send, udp_recv, connect, execute, send_file, get_index, rec_ack, transfer and send_pkt.

Additional Features:

Here is a comprehensive list of additional features we have added.

- *Server Console:* Server has the ability to act as an admin console, which controls and regulates clients (server is able to exit smoothly, kill misbehaving clients etc). Server is also able to broadcast to all connected peers (server does not have chatroom dependence).

- Optional password encrypted connections to existing chat rooms. The creator of the chat room can decide whether the chat room should have a password or not. This is an attempt to introduce security into the application.

- Another security feature we try to implement in the security context is sharing restrictions. Despite being a file sharing application, some users may not want to share all or some files. Such users have the ability to choose if they want to disable file transfer for specific files, or disable file transfer globally.

- Ability to configure variable window sizes while executing the Go Back N feature of reliable data transfer.

- Control over the number of threads, parallel connections etc. the application spawns while functioning.

Future Work:

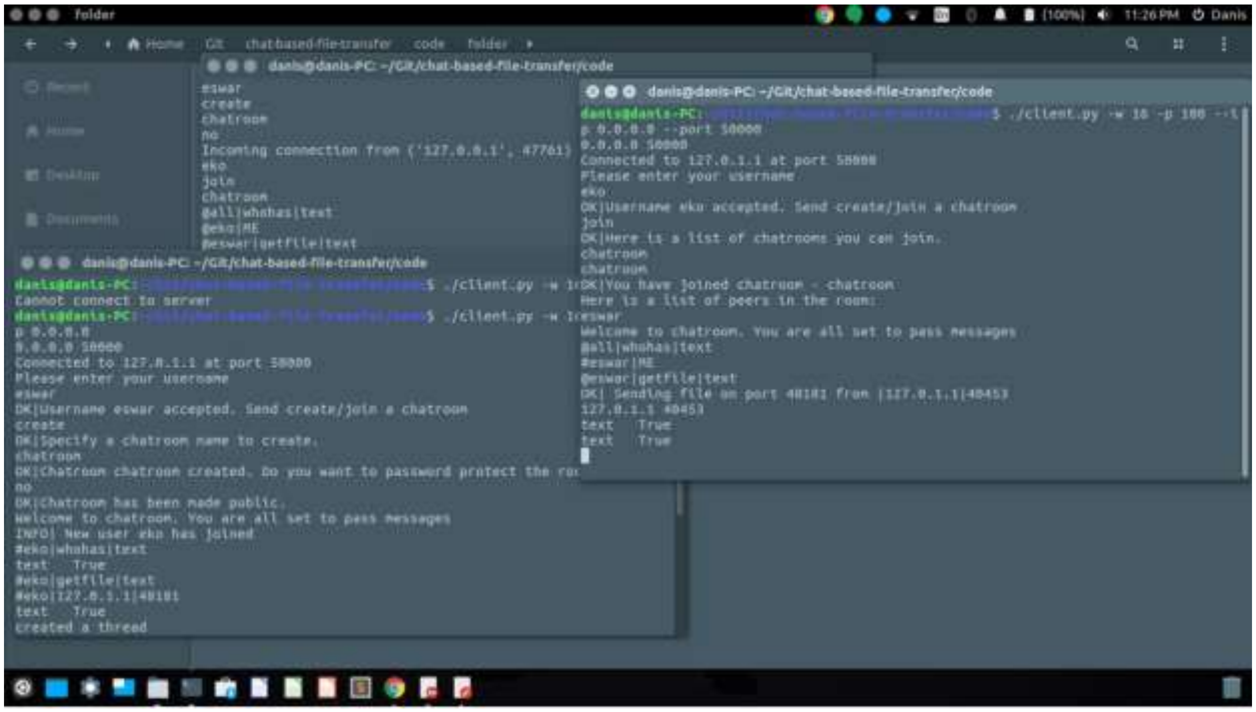
This is a simple, albeit complete implementation of a chat based file transfer application. It has so much potential for enhancements and integration with additional features. SOme of these features we expect to add in the near future include: -

NAT Traversal: The ability to detect servers who are connected behind NATs. Our current
application requires every device to be in the same network. We can overcome this limitation by using NAT traversal (universal plug-and-play).

Multi-hop Multi Server: Ability to have 2 peering servers. This gives us access to support more
clients. The servers should be able to communicate with each other, facilitating exchange of messages between devices connected across the servers.

Results

This is the screenshot for a successful implementation of chat-based file transfer. We have two clients and single server. We run server on the server PC using the terminal as sudo ./server.py. Then in the clients, we run the client code as sudo ./client.py -w 16 -p 100. The clients connect to the server. They enter their usernames and choose to connect to the chat room named chatroom chatroom. Once inside, we send some messages and request a file. We verify that the file has been successfully transferred to our folder. For a window size of 16 and a file size of 21.76MB, we got a throughput of 1.046MBps, and when the window size is 32, we get a throughput of 1.121MBps.



danis@danis-PC: ~/Git/chat-based-file-transfer/code

```
eswar
create
chatroom
no
Incoming connection from ('127.0.0.1', 47761)
eko
join
chatroom
@all|whohas|text
@eko|ME
@eswar|getfile|text
```

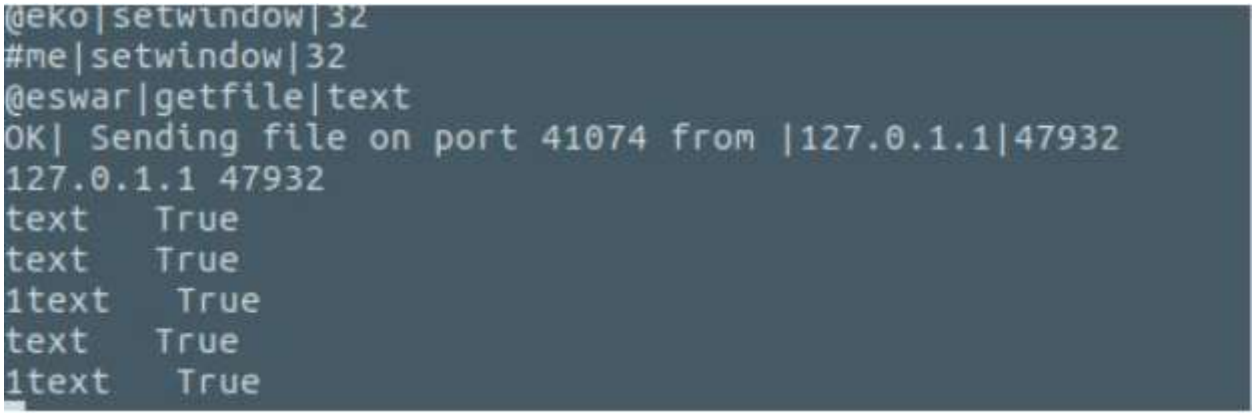
danis@danis-PC: ~/Git/chat-based-file-transfer/code

```
danis@danis-PC:[illegible]$ ./client.py -w 16 -p 100 --i
p 0.0.0.0 --port 50000
0.0.0.0 50000
Connected to 127.0.1.1 at port 50000
Please enter your username
eko
OK|Username eko accepted. Send create/join a chatroom
join
OK|Here is a list of chatrooms you can join.
chatroom
chatroom
OK|You have joined chatroom - chatroom
Here is a list of peers in the room:
eswar
Welcome to chatroom. You are all set to pass messages
@all|whohas|text
#eswar|ME
@eswar|getfile|text
OK| Sending file on port 40181 from |127.0.1.1|40453
127.0.1.1 40453
text   True
text   True
```

danis@danis-PC: ~/Git/chat-based-file-transfer/code

```
danis@danis-PC:[illegible]$ ./client.py -w 1
Cannot connect to server
danis@danis-PC:[illegible]$ ./client.py -w 1
p 0.0.0.0
0.0.0.0 50000
Connected to 127.0.1.1 at port 50000
Please enter your username
eswar
OK|Username eswar accepted. Send create/join a chatroom
create
OK|Specify a chatroom name to create.
chatroom
OK|Chatroom chatroom created. Do you want to password protect the ro
no
OK|Chatroom has been made public.
Welcome to chatroom. You are all set to pass messages
INFO| New user eko has joined
#eko|whohas|text
text   True
#eko|getfile|text
#eko|127.0.1.1|40181
text   True
created a thread
```

This is the screenshot for the set window functionality. We had started our client with a window size of 16. Now we try updating it to 16. Again, we verify that file transfer has successfully taken place by requesting a file named text and verifying that the client is able to receive it from its peer.



```
@eko|setwindow|32
#me|setwindow|32
@eswar|getfile|text
OK| Sending file on port 41074 from |127.0.1.1|47932
127.0.1.1 47932
text   True
text   True
1text   True
text   True
1text   True
```

We have implemented private chatrooms, server console to send messages or kill client connections, support to control file sharing using a global and individual file disables, variable window length for Go-Back-N protocol, among others.

Here is a test result we performed to check the effect of varying window size on throughput. We run server on the server PC using the terminal as sudo ./server.py. Then in the clients, we run the client code as sudo ./client.py -w 16 -p 100. The clients connect to the server. They enter their usernames and choose to connect to the chat room named chatroom chatroom. Once inside, we send some messages and request a file. We verify that the file has been successfully transferred to our folder.

Window Size	**File Size**	**Time Taken**	**Throughput**
16	21.76 MB	20.92 sec	1.04 MBps
32	21.76 MB	19.42 sec	1.12 MBps

References

- Stackoverflow www.stackoverflow.com
- Python 2.7 Official Documentation https://docs.python.org/2.7/reference/
- Beej's Blog on Socket Programming Using Python http://beej.us/blog/