PROJECT:

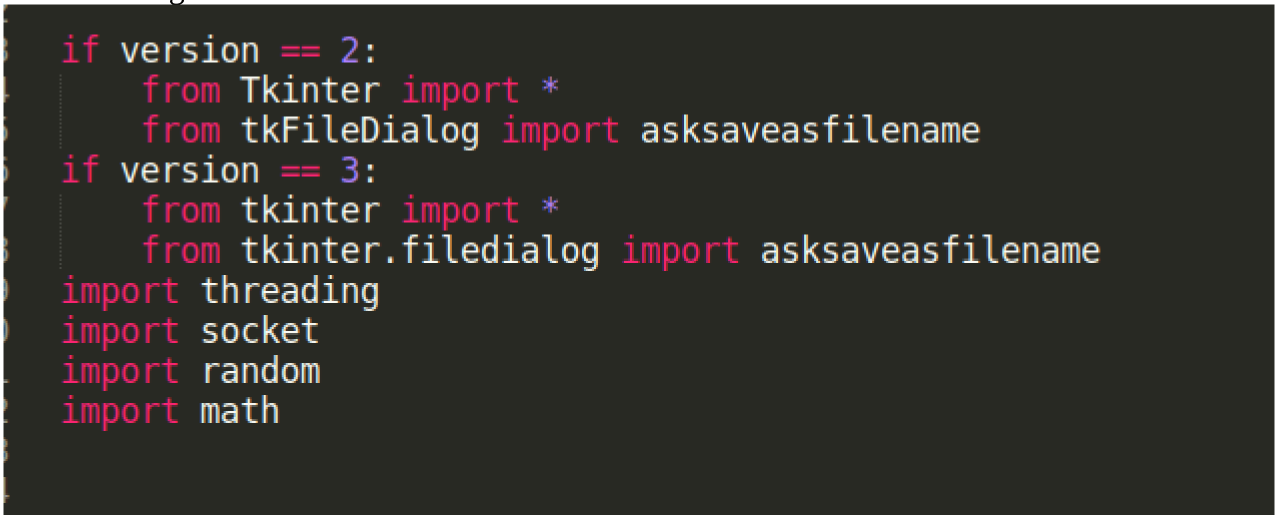
Contribution Analysis:

1. SUJOY DATTA(CED18I063)- Setting up the mainframe serverless architecture through socket programming in a fashion for simultaneous execution of app.
2. MIHIR SHRI(COE18B064)- Tkinter GUI implementation for efectiveness and adding the feature of contact adding in contacts.dat file including multithreading.
3. PRATEEK AGRAWAL(CED18I040)- Setting up runner function to provide additional features like saving chats in a txt file and quick peer communication with subordinate IPs.

Multithreading

In computer architecture, multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system. This approach differs from multiprocessing. In a multithreaded application, the threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer (TLB).

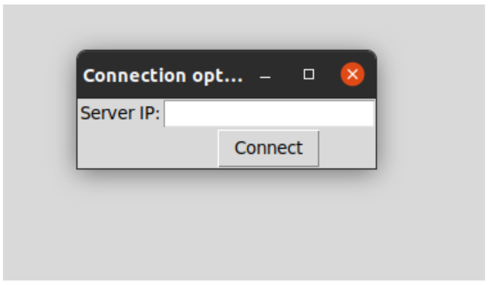
Where multiprocessing systems include multiple complete processing units in one or more cores, multithreading aims to increase utilization of a single core by using thread-level parallelism, as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and with CPUs with multiple multithreading cores.



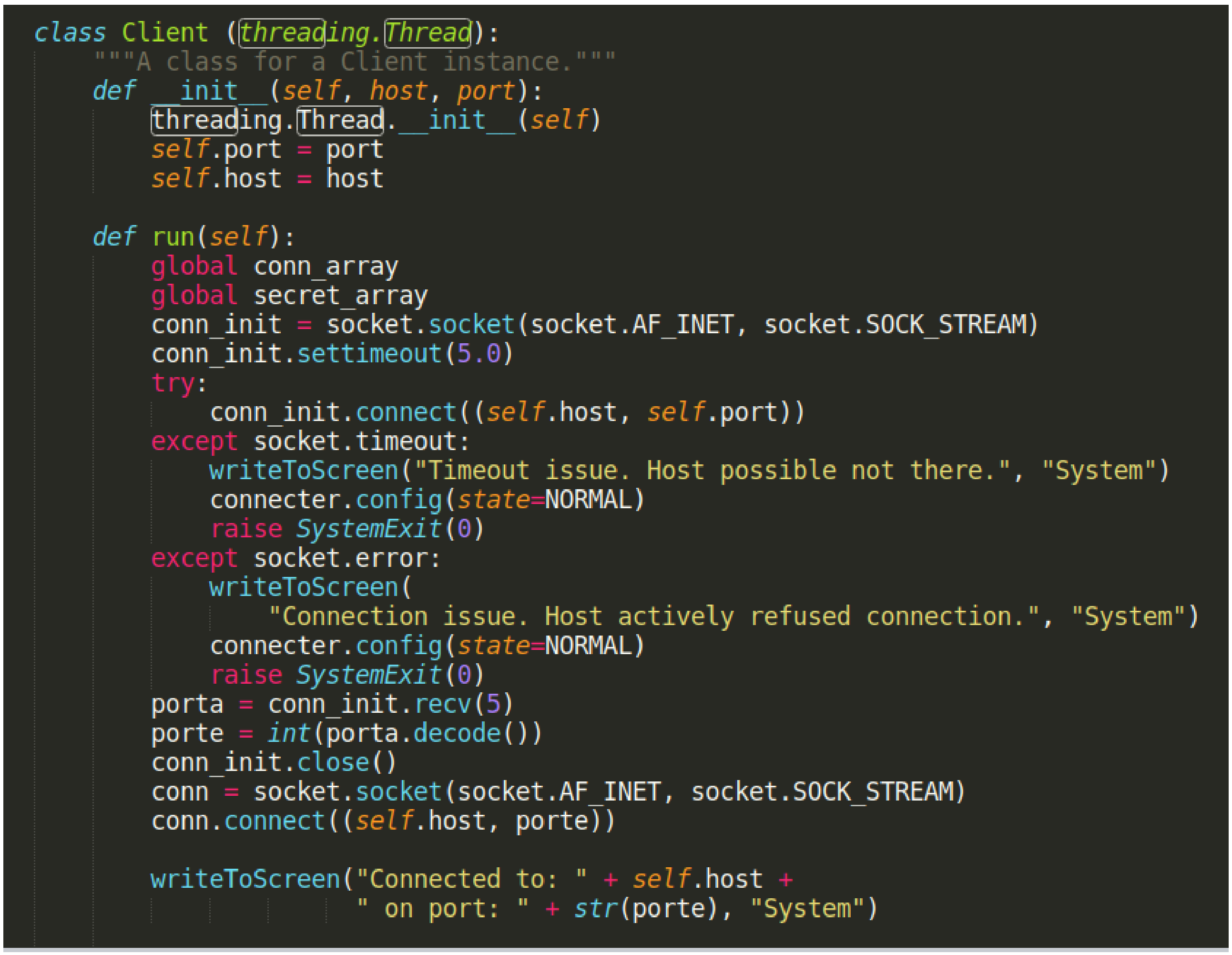
```
if version == 2:
    from Tkinter import *
    from tkFileDialog import asksaveasfilename
if version == 3:
    from tkinter import *
    from tkinter.filedialog import asksaveasfilename
import threading
import socket
import random
import math
```

How is multithreading used?

Every time the user enables an option of opening a dialog box, be it for changing his username or entering IP and port:
For example:
When the app is opened and the user clicks on connect a box opens up, prompting the user to enter their IP and port.



Code Screenshot:


```
class Client (threading.Thread):
    """A class for a Client instance."""
    def __init__(self, host, port):
        threading.Thread.__init__(self)
        self.port = port
        self.host = host

    def run(self):
        global conn_array
        global secret_array
        conn_init = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        conn_init.settimeout(5.0)
        try:
            conn_init.connect((self.host, self.port))
        except socket.timeout:
            writeToScreen("Timeout issue. Host possible not there.", "System")
            connecter.config(state=NORMAL)
            raise SystemExit(0)
        except socket.error:
            writeToScreen(
                "Connection issue. Host actively refused connection.", "System")
            connecter.config(state=NORMAL)
            raise SystemExit(0)
        porta = conn_init.recv(5)
        porte = int(porta.decode())
        conn_init.close()
        conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        conn.connect((self.host, porte))

        writeToScreen("Connected to: " + self.host +
                      " on port: " + str(porte), "System")
```

IP MESSENGER

IP Messaging, variously referred to as "realtime", "native", "in-app", or "over-the-top" messaging, describes the general use of TCP/IP (the Internet protocol) to provide messaging capabilities to mobile applications (typically via GSM / 3G / LTE or WiFi connectivity). In mobile development, IP Messaging is closely associated with instant messaging, geolocation tracking, mobile / push notifications, and IoT automation.

It is the primary form of user-to-user and application-to-user messaging in modern mobile applications and has several distinct advantages over technology like SMS / MMS. These advantages include:

1. Any device with an IP address may send and receive messages to anywhere globally over the Internet. In today's world, it's not difficult to see how far this idea went. Per-message fees do not apply. As such, IP Messaging is technically free apart from the indirect, and often negligible, cost of bandwidth usage. The term, "over-the-top", originates from this fact, as IP Messaging uses public Internet routing and skips over the fees imposed by telecommunication carriers.

2. Developers have complete control over messaging functionality and appearance as it can be embedded within the application UI (i.e. "in-app"). Similarly, users can enjoy the benefit of not

having the leave the app to receive messages or notifications that would otherwise be served via SMS.

3. Any application – whether web, mobile, desktop, or embedded – can make use of IP Messaging, while the specific methods of implementation and technologies used will vary widely between its almost unlimited use cases ranging from basic chat apps to the next Uber. In short: anything the Internet can do, IP Messaging can do for your app, too.

P2P communication

Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes.

Peers make a portion of their resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts. Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model in which the consumption and supply of resources is divided. Emerging collaborative P2P systems are going beyond the era of peers doing similar things while sharing resources, and are looking for diverse peers that can bring in unique resources and capabilities to a virtual community thereby empowering it to engage in greater tasks beyond those that can be accomplished by individual peers, yet that are beneficial to all the peers.

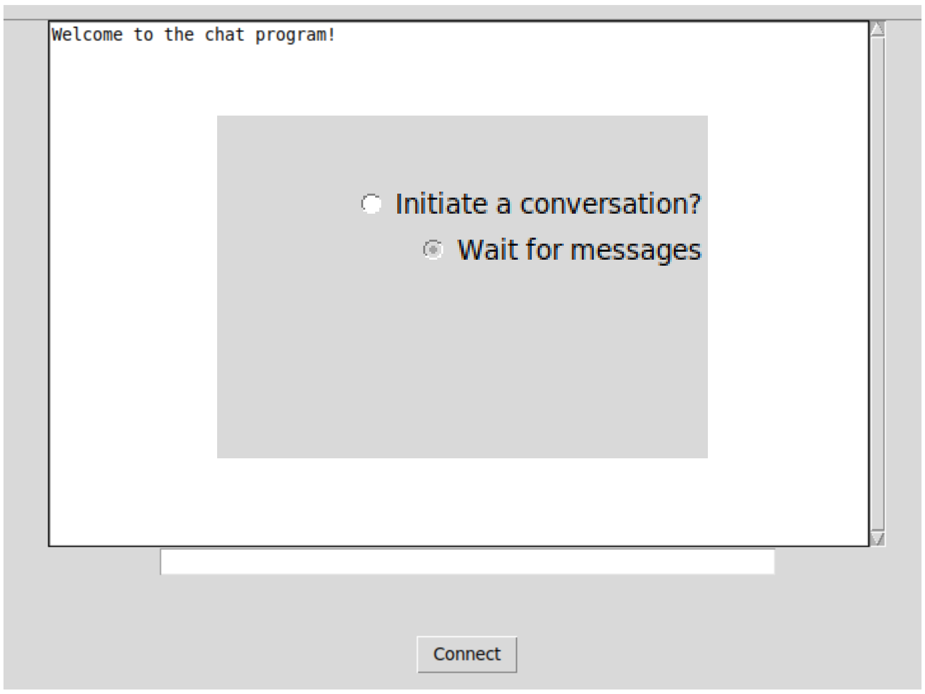
While P2P systems had previously been used in many application domains, the architecture was popularized by the file sharing system Napster, originally released in 1999. The concept has inspired new structures and philosophies in many areas of human interaction.

About ChatApp.py

ChatApp is a simple client server architecture which implements a very minimal structure of IP Messenger.
Features:
1. Multithreaded for multiple window support
2. Username specific messaging
3. Requires port only connection
4. Has a contacts.dat file set up on local storage that stores the IP and port of connections.

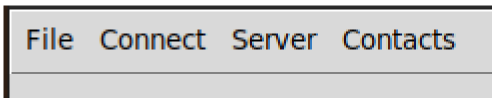
For more features check out the videos in the live demo folder.

Wide ChatBox:

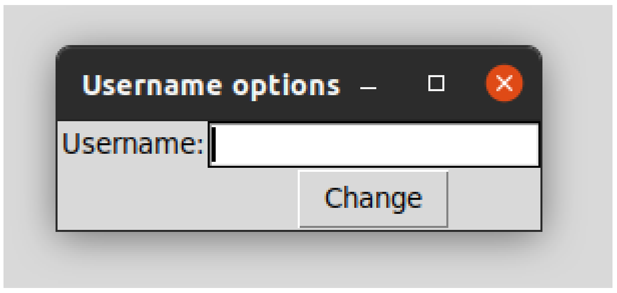


Custom Options:

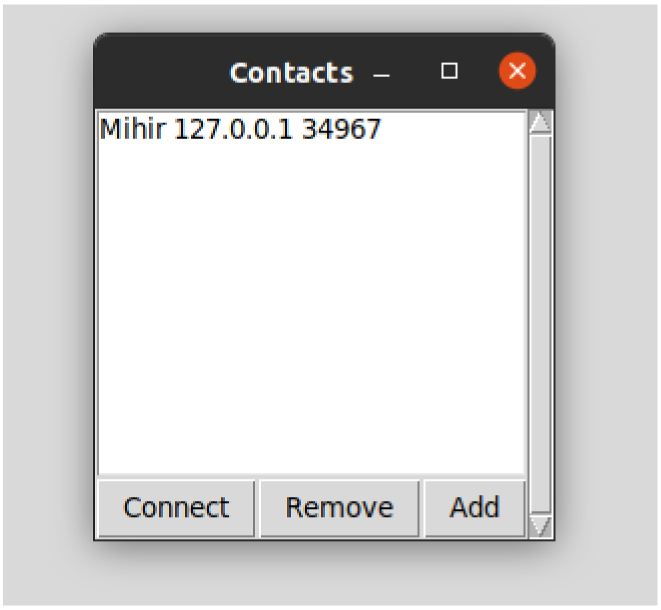
Toolbar:



Change Username:

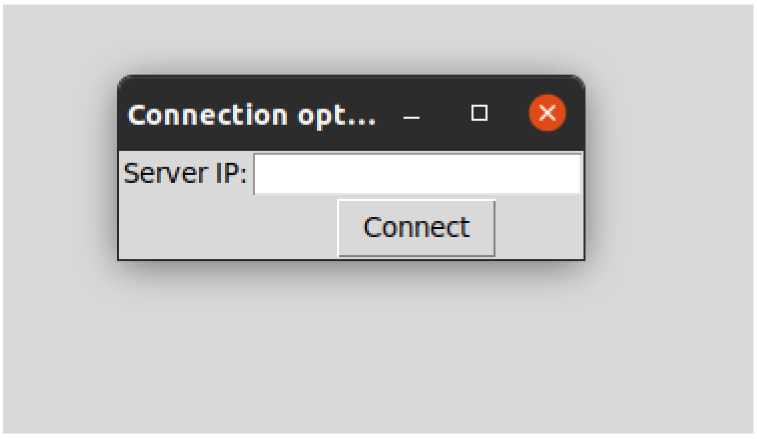


Save Contacts:



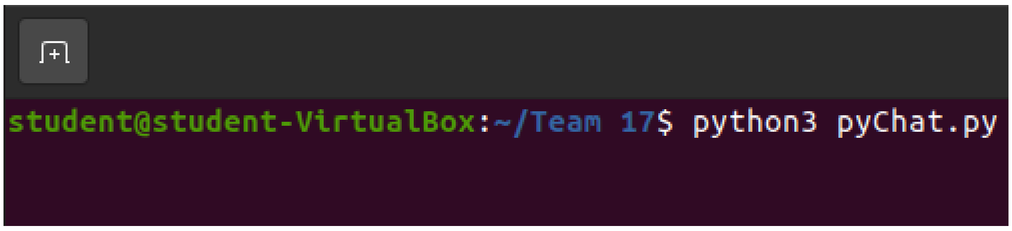
Mihir 127.0.0.1 34967

Quick Connect:



Working Manual

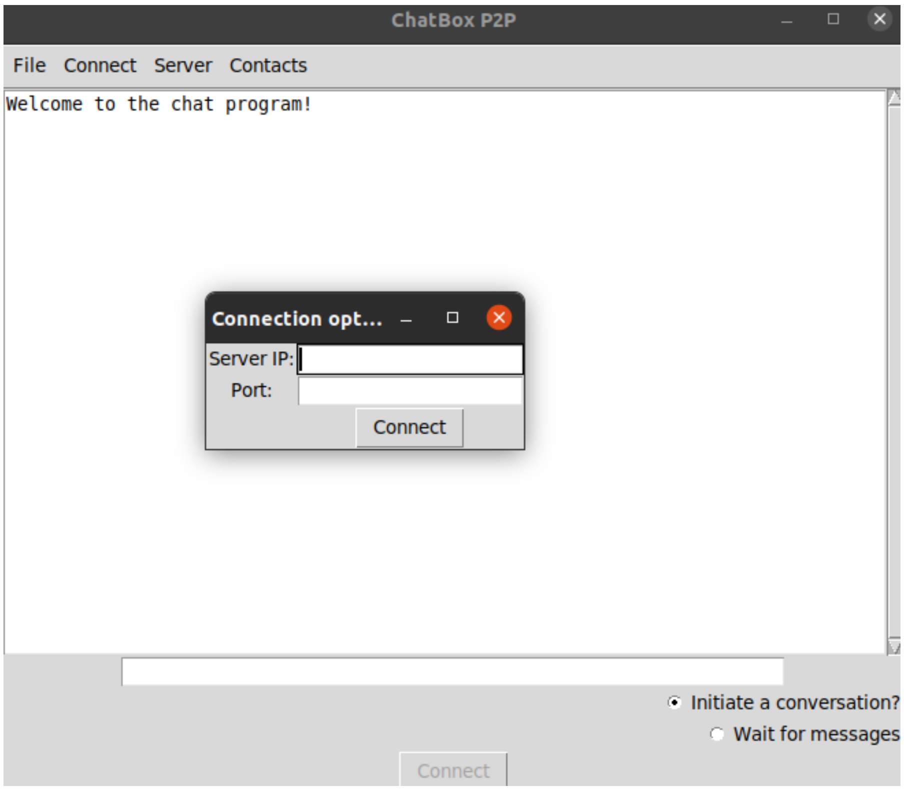
1. Write **python3 pyChat.py** in the project folder. Make sure that your sysetm has python3 installed beforehand.



```
student@student-VirtualBox:~/Team 17$ python3 pyChat.py
```

2. A tkinter window will open and will contain an option of wait for messages or initiate a conversation. This resembles to the WhatsApp feature of new message or receive message only option.

   a. If you click on initiate a conversation, the following dialog prompt opens, where you can enter the server IP line you want to connect to, along with the port number.

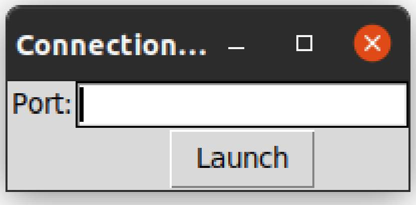


```
Welcome to the chat program!
```

Server IP:
Port:

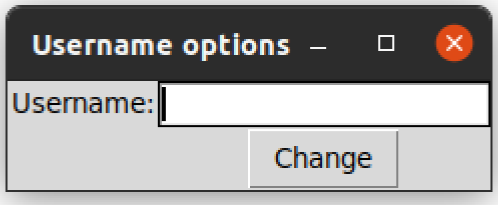
(●) Initiate a conversation?
( ) Wait for messages

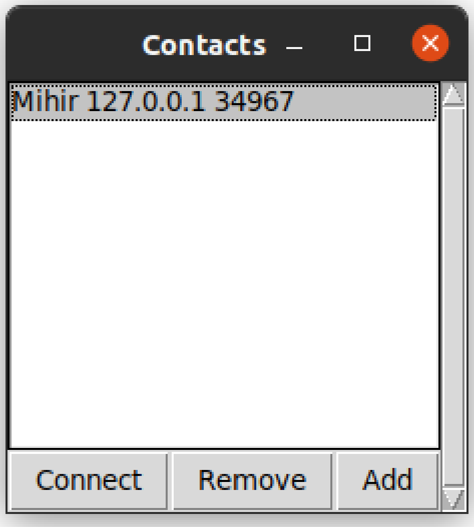
   b. If you wait for messages, you can just enter the port number on the LAN cable.



Port:

3. Once the mainframe connection is setup, you just have to start chatting. You can disconnect at any point. The additional features include changing your username at any point of the chatting algorithm to ensure professionality in freedom of the application. There is also a feature of adding a contact to your contact file so that you don't have to manually connect to a peer everytime you want to chat.





Mihir 127.0.0.1 34967