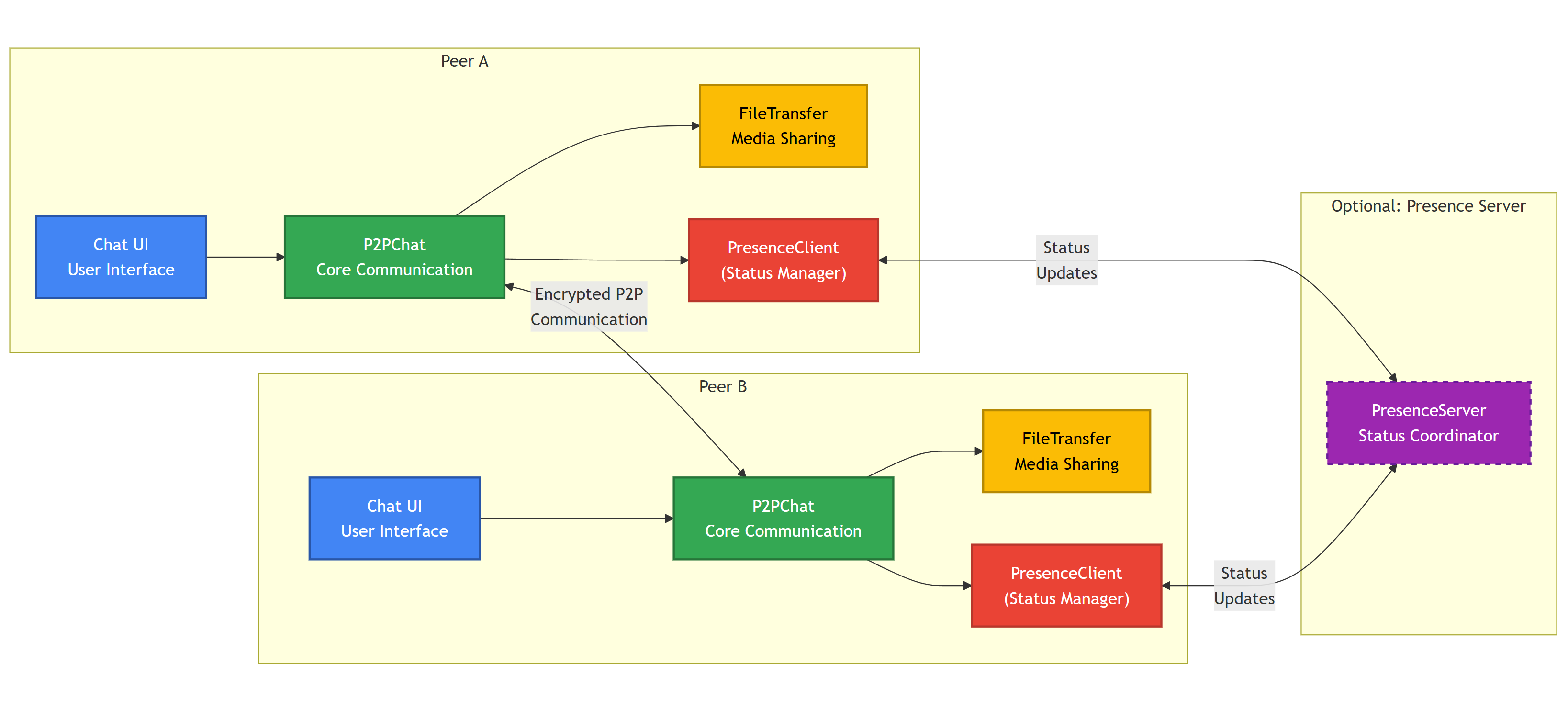
INRODUCTION

ARCHITECTURE

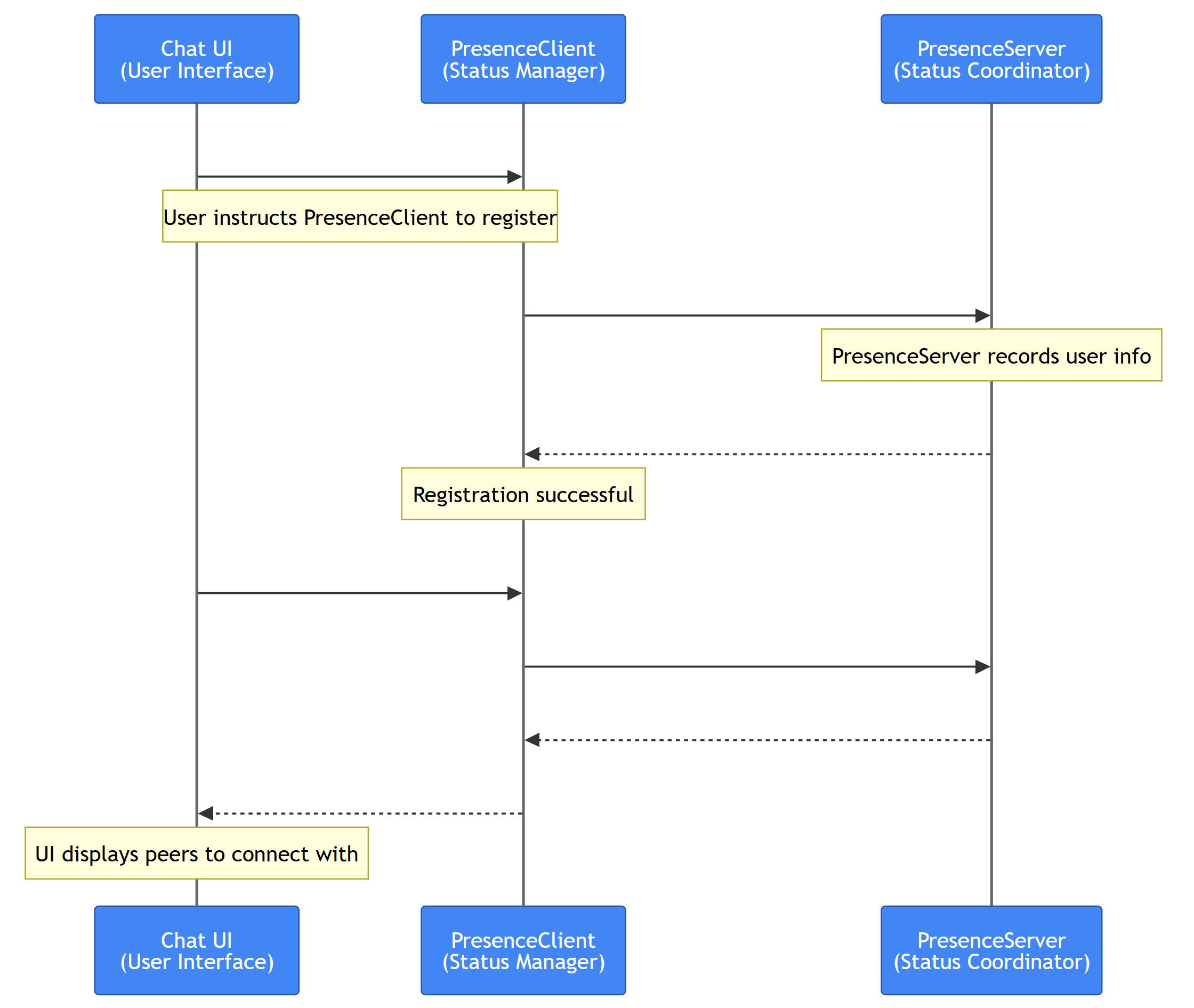
High Level System (Component Diagram)



**Explanation**:

* Each **Peer** encapsulates:
  + A **GUI** (Chat UI) to handle user interaction.
  + The **P2PChat** component for managing peer connections and message routing.
  + A **File\_Transfer** module for chunk-based file sending/receiving.
  + A **Presence Client** that communicates with the **Presence.**
* Lines (<-->) indicate TCP connections.
* Peers can connect directly to each other **and** optionally to the presence server.

Presence-Based Discovery



**Explanation**:

* The **user interface** (U) calls methods on the PresenceClient (PC) to register or query users.
* The **presence server** (PS) responds with success or error messages, and eventually provides a list of online users.
* Once the **UI** receives that list, the user can **initiate a direct P2P connection** to any desired peer’s IP/port.

File Transfer Flow

A screenshot of a computer screen

AI-generated content may be incorrect.

**Explanation**:

* The **Sender’s UI** triggers the send\_file method in the FileTransfer object (SFT).
* FileTransfer reads the file, chunk by chunk, and instructs P2PChat to send each chunk as a JSON message to the **Receiver’s P2PChat**.
* The receiver’s P2PChat calls back into its own FileTransfer module to handle the chunk (file\_chunk\_callback).
* Chunks are buffered until the last chunk is detected, then the receiver writes the file to disk.
* The UI can display progress or completion messages as needed.

IMPLEMENTATION

TESTING

DISCUSSION

CONCLUSION

APPENDIX (Code)

Chat.py

# chat.py

#

# This file implements the main GUI for a peer-to-peer chat application.

# The application allows users to:

# 1. Create a new chat network

# 2. Join existing chat networks directly or through a presence server

# 3. Exchange text messages with connected peers

# 4. Send and receive files

# 5. Track online/offline status of peers

#

# The application uses a combination of threading and socket programming to enable

# simultaneous sending and receiving of messages without blocking the GUI.

import os

import sys

import base64

import socket

import json

import tkinter as tk

from tkinter import (

    Tk,

    Label,

    Entry,

    Button,

    Text,

    Frame,

    Scrollbar,

    PhotoImage,

    messagebox,

    Radiobutton,

    StringVar,

    Toplevel,

)

import threading

from tkinter import filedialog

from p2p\_chat import P2PChat  # Custom P2P chat implementation

import file\_transfer  # Module for handling file transfers

from utils import send\_message  # Utility function for network communication

from presence\_client import PresenceClient  # For user discovery

class ChatUI:

    """

    Main user interface class that handles all GUI elements and interactions for the P2P chat application.

    Manages the different screens (welcome, join, chat) and connects GUI actions to underlying P2P functionality.

    """

    def \_\_init\_\_(*self*, *root*):

        """

        Initialize the ChatUI with the root Tkinter window.

        Args:

            root: The Tkinter root window

        """

*self*.root = *root*

*self*.root.title("P2P Chat")

*self*.root.geometry("360x640")  # Set initial window size

*self*.root.protocol("WM\_DELETE\_WINDOW", *self*.on\_closing)  # Handle window close event

*self*.chat\_instance = None  # Will hold the P2PChat instance when created

*self*.username = ""  # Will store the user's chosen username

*self*.chat\_display = None  # Text widget reference for displaying messages

*self*.peer\_status = {}  # Dictionary to track online/offline status of peers

        # Start with the welcome screen

*self*.setup\_welcome\_screen()

        # Initialize file transfer handler

*self*.file\_transfer = file\_transfer.FileTransfer(

*message\_sender*=*self*.\_send\_message\_to\_peer,  # Provide method to send file chunks

*downloads\_folder*="downloads",  # Where received files will be stored

*ui\_callback*=*self*.update\_chat\_display,  # For showing file transfer status messages

        )

        # Will be initialized when needed to connect to presence service

*self*.presence\_client = None

    def \_send\_message\_to\_peer(*self*, *peer\_addr*, *message*: dict):

        """

        Helper method that sends a message to a specific peer.

        Used by the file transfer system to send file chunks.

        Args:

            peer\_addr: Tuple of (host, port) to connect to

            message: Dictionary to be JSON-encoded and sent

        """

        host, port = *peer\_addr*

        try:

            # Create a new socket for this specific message

            sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

            sock.connect((host, port))

            # Use the utility function to send the message

            send\_message(sock, *message*)

            sock.close()

            print(f"Sent chunk of size {len(*message*.get('data', ''))} to {*peer\_addr*}")

        except Exception as e:

            print(f"Error in \_send\_message\_to\_peer({*peer\_addr*}): {e}")

    def select\_file(*self*):

        """

        Open a file dialog for the user to select a file to send.

        Once selected, transmit the file to all connected peers.

        """

        # Open file selection dialog

        file\_path = filedialog.askopenfilename()

        if file\_path:

            filename = os.path.basename(file\_path)

*self*.update\_chat\_display(f"System: Sending file '{filename}' ...")

            # Broadcast the file to all connected peers

            if *self*.chat\_instance and *self*.chat\_instance.peers:

                for peer\_username, peer\_info in *self*.chat\_instance.peers.items():

                    target\_addr = (peer\_info["address"], peer\_info["port"])

                    # Use the file transfer module to handle splitting the file into chunks

*self*.file\_transfer.send\_file(file\_path, *target\_addr*=target\_addr)

*self*.update\_chat\_display(

                        f"System: File '{filename}' sent to {peer\_username}"

                    )

            else:

*self*.update\_chat\_display("System: No peers to send the file to.")

    def setup\_welcome\_screen(*self*):

        """

        Create and display the welcome screen with username input

        and options to create or join a chat.

        """

        # Clear any existing widgets from the window

        for widget in *self*.root.winfo\_children():

            widget.destroy()

        # Create the welcome header

        welcome\_label = tk.Label(

*self*.root,

*text*="Welcome to P2P Chat",

*font*=("Helvetica", 24, "bold"),

*pady*=20,

        )

        welcome\_label.pack()

        # Add description text

        intro\_text = tk.Label(

*self*.root,

*text*="Connect directly with friends using peer-to-peer technology. No servers, no tracking, just private communication.",

*font*=("Helvetica", 12),

*wraplength*=300,

*justify*="center",

        )

        intro\_text.pack(*pady*=10)

        # Try to load and display app logo image

        try:

            image = PhotoImage(*file*="chat.png")

            image = image.subsample(4, 4)  # Resize image to be smaller

            image\_label = Label(*self*.root, *image*=image)

            image\_label.image = image  # Keep a reference to prevent garbage collection

            image\_label.pack(*pady*=20)

        except Exception as e:

            print(f"Could not load image: {e}")

            # Display a text placeholder if image can't be loaded

            placeholder = Label(*self*.root, *text*="[Chat Icon]", *font*=("Helvetica", 18))

            placeholder.pack(*pady*=20)

        # Username input

        username\_label = tk.Label(

*self*.root, *text*="Please enter your username:", *font*=("Helvetica", 14)

        )

        username\_label.pack(*pady*=10)

*self*.username\_entry = tk.Entry(*self*.root, *font*=("Helvetica", 14))

*self*.username\_entry.pack(*pady*=10)

        # Create navigation buttons

        button\_frame = Frame(*self*.root)

        button\_frame.pack(*pady*=20)

        # "Create Chat" button with blue styling

        create\_button = tk.Button(

            button\_frame,

*text*="Create Chat",

*font*=("Helvetica", 14),

*bg*="#2196F3",

*fg*="white",

*relief*="flat",

*command*=*self*.on\_create\_chat,

        )

        create\_button.pack(*pady*=10, *ipady*=5, *ipadx*=10)

        # "Join Chat" button with outlined styling

        join\_button = tk.Button(

            button\_frame,

*text*="  Join Chat  ",

*font*=("Helvetica", 14),

*bg*="white",

*fg*="#2196F3",

*relief*="flat",

*command*=*self*.show\_join\_screen,

        )

        join\_button.pack(*pady*=10, *ipady*=5, *ipadx*=10)

        join\_button.config(

*highlightthickness*=2,

*highlightbackground*="#2196F3",

*highlightcolor*="#2196F3",

        )

    def on\_create\_chat(*self*):

        """

        Handle the 'Create Chat' button click.

        Creates a new P2P chat network with the user as the first node.

        """

        # Get and validate username

        username = *self*.username\_entry.get().strip()

        if not username:

            messagebox.showerror("Error", "Please enter a username")

            return

        # Store username and set up the chat interface

*self*.username = username

*self*.setup\_chat\_screen()

        # Initialize the P2P chat instance with callbacks for different events

*self*.chat\_instance = P2PChat(

            username,

*ui\_callback*=*self*.update\_chat\_display,  # For displaying chat messages

*file\_chunk\_callback*=*self*.file\_transfer.handle\_incoming\_file\_chunk,  # For handling file transfers

*status\_callback*=*self*.update\_peer\_status,  # For handling peer status updates

        )

        # Show connection information to the user

*self*.update\_chat\_display(f"You are connected as {*self*.username}.")

*self*.update\_chat\_display(

            f"Your chat is running on port {*self*.chat\_instance.port}."

        )

    def show\_join\_screen(*self*):

        """

        Display a screen that lets users choose how they want to join a chat:

        either by direct connection or through the presence service.

        """

        # Get and validate username

        username = *self*.username\_entry.get().strip()

        if not username:

            messagebox.showerror("Error", "Please enter a username")

            return

*self*.username = username

        # Clear existing UI

        for widget in *self*.root.winfo\_children():

            widget.destroy()

        # Create a container frame for the join options

        join\_container = Frame(*self*.root)

        join\_container.pack(*fill*=tk.BOTH, *expand*=True, *padx*=20, *pady*=20)

        # Add a heading

        join\_label = Label(

            join\_container,

*text*="How would you like to connect?",

*font*=("Helvetica", 20, "bold"),

        )

        join\_label.pack(*pady*=20)

        # Create a connection method choice using radio buttons

        connection\_method = StringVar()

        connection\_method.set("direct")  # Default to direct connection

        # Option 1: Direct Connection (IP + Port)

        direct\_frame = Frame(join\_container, *relief*=tk.RIDGE, *borderwidth*=2)

        direct\_frame.pack(*fill*=tk.X, *pady*=10, *ipady*=10)

        direct\_radio = Radiobutton(

            direct\_frame,

*text*="Direct Connection",

*variable*=connection\_method,

*value*="direct",

*font*=("Helvetica", 14),

        )

        direct\_radio.pack(*anchor*=tk.W, *padx*=10, *pady*=(10, 5))

        direct\_desc = Label(

            direct\_frame,

*text*="Connect directly to a peer by entering their IP and port.",

*font*=("Helvetica", 10),

*wraplength*=300,

*justify*=tk.LEFT,

        )

        direct\_desc.pack(*anchor*=tk.W, *padx*=30, *pady*=(0, 10))

        # Option 2: Presence Server (discover online users)

        presence\_frame = Frame(join\_container, *relief*=tk.RIDGE, *borderwidth*=2)

        presence\_frame.pack(*fill*=tk.X, *pady*=10, *ipady*=10)

        presence\_radio = Radiobutton(

            presence\_frame,

*text*="Find Online Users",

*variable*=connection\_method,

*value*="presence",

*font*=("Helvetica", 14),

        )

        presence\_radio.pack(*anchor*=tk.W, *padx*=10, *pady*=(10, 5))

        presence\_desc = Label(

            presence\_frame,

*text*="See who's online and connect to them.",

*font*=("Helvetica", 10),

*wraplength*=300,

*justify*=tk.LEFT,

        )

        presence\_desc.pack(*anchor*=tk.W, *padx*=30, *pady*=(0, 10))

        # Navigation buttons

        next\_btn = Button(

            join\_container,

*text*="Next",

*font*=("Helvetica", 14),

*bg*="#2196F3",

*fg*="white",

*relief*="flat",

*command*=lambda: *self*.\_show\_connection\_screen(connection\_method.get()),

        )

        next\_btn.pack(*pady*=20, *ipady*=5, *ipadx*=20)

        # Back button

        back\_button = Button(

            join\_container,

*text*="Back",

*font*=("Helvetica", 14),

*bg*="white",

*fg*="#2196F3",

*relief*="flat",

*command*=*self*.setup\_welcome\_screen,

        )

        back\_button.pack(*pady*=10, *ipady*=5, *ipadx*=10)

        back\_button.config(

*highlightthickness*=2,

*highlightbackground*="#2196F3",

*highlightcolor*="#2196F3",

        )

    def \_show\_connection\_screen(*self*, *connection\_method*):

        """

        Show the appropriate connection screen based on user's selection.

        Args:

            connection\_method: Either "direct" for IP/port entry or "presence" for online users

        """

        if *connection\_method* == "direct":

*self*.\_show\_direct\_connection\_screen()

        else:  # presence

*self*.\_show\_presence\_connection\_screen()

    def \_show\_direct\_connection\_screen(*self*):

        """

        Show the screen for direct connection by IP address and port.

        """

        for widget in *self*.root.winfo\_children():

            widget.destroy()

        join\_label = Label(

*self*.root,

*text*="Direct Connection",

*font*=("Helvetica", 20, "bold"),

*pady*=20,

        )

        join\_label.pack()

        # Input for host address (IP)

        host\_label = Label(*self*.root, *text*="Host address:", *font*=("Helvetica", 14))

        host\_label.pack(*pady*=5)

*self*.host\_entry = Entry(*self*.root, *font*=("Helvetica", 14))

*self*.host\_entry.insert(0, "127.0.0.1")  # Default to localhost

*self*.host\_entry.pack(*pady*=5)

        # Input for port number

        port\_label = Label(*self*.root, *text*="Port:", *font*=("Helvetica", 14))

        port\_label.pack(*pady*=5)

*self*.port\_entry = Entry(*self*.root, *font*=("Helvetica", 14))

*self*.port\_entry.pack(*pady*=5)

        # Navigation buttons

        button\_frame = Frame(*self*.root)

        button\_frame.pack(*pady*=20)

        join\_button = Button(

            button\_frame,

*text*="Connect",

*font*=("Helvetica", 14),

*bg*="#2196F3",

*fg*="white",

*relief*="flat",

*command*=*self*.on\_join\_chat,

        )

        join\_button.pack(*side*=tk.LEFT, *padx*=10, *ipady*=5, *ipadx*=10)

        back\_button = Button(

            button\_frame,

*text*="Back",

*font*=("Helvetica", 14),

*bg*="white",

*fg*="#2196F3",

*relief*="flat",

*command*=*self*.show\_join\_screen,  # Go back to connection method selection

        )

        back\_button.pack(*side*=tk.LEFT, *padx*=10, *ipady*=5, *ipadx*=10)

        back\_button.config(

*highlightthickness*=2,

*highlightbackground*="#2196F3",

*highlightcolor*="#2196F3",

        )

    def \_show\_presence\_connection\_screen(*self*):

        """

        Set up chat and connect to the presence server to discover online users.

        """

        # Clear the window

        for widget in *self*.root.winfo\_children():

            widget.destroy()

        # Create and set up the chat instance first so we have a listening port

*self*.setup\_chat\_screen()

*self*.chat\_instance = P2PChat(

*self*.username,

*ui\_callback*=*self*.update\_chat\_display,

*file\_chunk\_callback*=*self*.file\_transfer.handle\_incoming\_file\_chunk,

*status\_callback*=*self*.update\_peer\_status,

        )

        # Initialize presence client with our username and allocated port

*self*.presence\_client = PresenceClient(*self*.username, *self*.chat\_instance.port)

        # Try to register with presence server

        if not *self*.presence\_client.register():

            messagebox.showerror(

                "Error",

                "Could not connect to presence server. Please try direct connection.",

            )

*self*.show\_join\_screen()

            return

        # Show the online users dialog to let the user select someone to chat with

*self*.show\_online\_users\_dialog()

    def show\_online\_users\_dialog(*self*):

        """

        Display a dialog showing users who are currently online.

        The user can select someone to connect to from this list.

        """

        # If dialog already exists, destroy it first to refresh the list

        if hasattr(*self*, "online\_users\_dialog") and *self*.online\_users\_dialog:

*self*.online\_users\_dialog.destroy()

        # Create dialog

*self*.online\_users\_dialog = Toplevel(*self*.root)

*self*.online\_users\_dialog.title("Online Users")

*self*.online\_users\_dialog.geometry("300x400")

*self*.online\_users\_dialog.transient(

*self*.root

        )  # Make it appear on top of main window

        # Title

        title\_label = Label(

*self*.online\_users\_dialog,

*text*="Find Online Users",

*font*=("Helvetica", 16, "bold"),

*pady*=10,

        )

        title\_label.pack()

        # Instructions

        instructions = Label(

*self*.online\_users\_dialog,

*text*="Select a user to chat with:",

*font*=("Helvetica", 12),

*wraplength*=280,

        )

        instructions.pack(*pady*=(0, 10))

        # Create a frame for the user list with scrolling capability

        list\_frame = Frame(*self*.online\_users\_dialog)

        list\_frame.pack(*fill*=tk.BOTH, *expand*=True, *padx*=10, *pady*=10)

        # Create a scrollable canvas to hold the users

        canvas = tk.Canvas(list\_frame)

        scrollbar = Scrollbar(list\_frame, *orient*="vertical", *command*=canvas.yview)

        scrollable\_frame = Frame(canvas)

        # Configure the canvas to expand when contents change

        scrollable\_frame.bind(

            "<Configure>", lambda *e*: canvas.configure(*scrollregion*=canvas.bbox("all"))

        )

        canvas.create\_window((0, 0), *window*=scrollable\_frame, *anchor*="nw")

        canvas.configure(*yscrollcommand*=scrollbar.set)

        # Pack the canvas and scrollbar

        canvas.pack(*side*="left", *fill*="both", *expand*=True)

        scrollbar.pack(*side*="right", *fill*="y")

        # Fetch online users from the presence server

        online\_users = *self*.presence\_client.get\_online\_users()

        if not online\_users:

            # Display a message if no users are online

            no\_users\_label = Label(

                scrollable\_frame,

*text*="No online users found",

*font*=("Helvetica", 12),

*fg*="gray",

*pady*=20,

            )

            no\_users\_label.pack()

        else:

            # Display each user with a connect button

            for user in online\_users:

                user\_frame = Frame(scrollable\_frame, *borderwidth*=1, *relief*="groove")

                user\_frame.pack(*fill*=tk.X, *pady*=5, *padx*=3)

                username = user.get("username", "Unknown")

                address = user.get("address", "Unknown")

                port = user.get("port", "Unknown")

                # Username label

                user\_label = Label(

                    user\_frame, *text*=username, *font*=("Helvetica", 12, "bold")

                )

                user\_label.pack(*anchor*=tk.W, *padx*=10, *pady*=(5, 0))

                # Address and port

                addr\_label = Label(

                    user\_frame,

*text*=f"{address}:{port}",

*font*=("Helvetica", 10),

*fg*="gray",

                )

                addr\_label.pack(*anchor*=tk.W, *padx*=10, *pady*=(0, 5))

                # Connect button - uses lambda to pass the specific address and port

                connect\_btn = Button(

                    user\_frame,

*text*="Connect",

*bg*="#4CAF50",

*fg*="white",

*command*=lambda *a*=address, *p*=port: *self*.connect\_to\_presence\_user(

*a*, *p*

                    ),

                )

                connect\_btn.pack(*anchor*=tk.E, *padx*=10, *pady*=5)

        # Button frame at bottom

        button\_frame = Frame(*self*.online\_users\_dialog)

        button\_frame.pack(*fill*=tk.X, *padx*=10, *pady*=10)

        # Refresh button - reloads the dialog with fresh data

        refresh\_btn = Button(

            button\_frame,

*text*="Refresh",

*bg*="#2196F3",

*fg*="white",

*command*=*self*.show\_online\_users\_dialog,

        )

        refresh\_btn.pack(*side*=tk.LEFT, *padx*=5)

        # Cancel button - closes the dialog

        cancel\_btn = Button(

            button\_frame,

*text*="Cancel",

*command*=lambda: *self*.online\_users\_dialog.destroy(),

        )

        cancel\_btn.pack(*side*=tk.RIGHT, *padx*=5)

    def connect\_to\_presence\_user(*self*, *address*, *port*):

        """

        Connect to a user selected from the online users list.

        Args:

            address: IP address of the peer to connect to

            port: Port number of the peer to connect to

        """

        # Close the dialog

        if hasattr(*self*, "online\_users\_dialog") and *self*.online\_users\_dialog:

*self*.online\_users\_dialog.destroy()

        # Connect to the selected user

        try:

            success = *self*.chat\_instance.join\_network(*address*, *port*)

            if success:

                # Show success messages

*self*.update\_chat\_display(f"You are connected as {*self*.username}.")

*self*.update\_chat\_display(

                    f"Your chat is running on port {*self*.chat\_instance.port}."

                )

*self*.update\_chat\_display(f"Successfully connected to {*address*}:{*port*}")

            else:

                # Handle connection failure

                messagebox.showerror("Error", f"Failed to connect to {*address*}:{*port*}")

                # Show the online users dialog again so they can try someone else

*self*.show\_online\_users\_dialog()

        except Exception as e:

            messagebox.showerror("Error", f"Failed to join chat: {e}")

*self*.show\_online\_users\_dialog()

    def on\_join\_chat(*self*):

        """

        Handle the 'Connect' button click on the direct connection screen.

        Attempts to join an existing chat network using the provided host and port.

        """

        try:

            # Get and validate input

            host = *self*.host\_entry.get().strip()

            port = int(*self*.port\_entry.get().strip())

            # Set up chat UI first

*self*.setup\_chat\_screen()

            # Initialize P2P chat instance with necessary callbacks

*self*.chat\_instance = P2PChat(

*self*.username,

*ui\_callback*=*self*.update\_chat\_display,

*file\_chunk\_callback*=*self*.file\_transfer.handle\_incoming\_file\_chunk,

*status\_callback*=*self*.update\_peer\_status,

            )

            # Try to join the network at the specified address

            success = *self*.chat\_instance.join\_network(host, port)

            if success:

                # Show success messages

*self*.update\_chat\_display(f"You are connected as {*self*.username}.")

*self*.update\_chat\_display(

                    f"Your chat is running on port {*self*.chat\_instance.port}."

                )

*self*.update\_chat\_display(f"Successfully connected to {host}:{port}")

            else:

                messagebox.showerror("Error", f"Failed to connect to {host}:{port}")

        except ValueError:

            # Handle invalid port entry (not a number)

            messagebox.showerror("Error", "Port must be a number")

        except Exception as e:

            # Handle any other connection errors

            messagebox.showerror("Error", f"Failed to join chat: {e}")

    def update\_peer\_status(*self*, *peer\_username*, *status*):

        """

        Called by the P2PChat instance when a peer's status changes.

        Updates the UI to reflect peer status changes.

        Args:

            peer\_username: Username of the peer whose status changed

            status: New status ("online" or "offline")

        """

        old\_status = *self*.peer\_status.get(*peer\_username*, "unknown")

*self*.peer\_status[*peer\_username*] = *status*

        # Only notify about status changes, not initial status

        if old\_status != "unknown" and old\_status != *status*:

            if *status* == "online":

*self*.update\_chat\_display(f"System: {*peer\_username*} is now online")

            else:

*self*.update\_chat\_display(f"System: {*peer\_username*} is now offline")

        # Update the UI if the peers dialog is open

        if hasattr(*self*, "peers\_window") and *self*.peers\_window:

*self*.show\_peers()

    def setup\_chat\_screen(*self*):

        """

        Set up the main chat interface with message display, input field, and controls.

        This is shown after successfully creating or joining a chat.

        """

        # Clear any existing widgets

        for widget in *self*.root.winfo\_children():

            widget.destroy()

        # Create header with username and status

        header\_frame = Frame(*self*.root, *bg*="#2196F3")

        header\_frame.pack(*fill*=tk.X)

        username\_label = Label(

            header\_frame,

*text*=f"Username: {*self*.username}",

*font*=("Helvetica", 12),

*bg*="#2196F3",

*fg*="white",

*pady*=10,

        )

        username\_label.pack(*side*=tk.LEFT, *padx*=10)

        # Button to view connected peers

        peers\_button = Button(

            header\_frame,

*text*="Peers",

*font*=("Helvetica", 12),

*bg*="#0d47a1",

*fg*="white",

*relief*="flat",

*command*=*self*.show\_peers,

        )

        peers\_button.pack(*side*=tk.RIGHT, *padx*=10, *pady*=5)

        # Add a status indicator in the header

        status\_label = Label(

            header\_frame,

*text*="Online",

*font*=("Helvetica", 10),

*bg*="#4CAF50",  # Green for online

*fg*="white",

*padx*=8,

*pady*=2,

*borderwidth*=0,

        )

        status\_label.pack(*side*=tk.RIGHT, *padx*=10, *pady*=8)

*self*.status\_indicator = status\_label

        # Create the main chat display area with scrollbar

        chat\_frame = Frame(*self*.root)

        chat\_frame.pack(*fill*=tk.BOTH, *expand*=True)

*self*.chat\_display = Text(chat\_frame, *wrap*=tk.WORD, *state*=tk.DISABLED)

*self*.chat\_display.pack(*side*=tk.LEFT, *fill*=tk.BOTH, *expand*=True)

        scrollbar = Scrollbar(chat\_frame)

        scrollbar.pack(*side*=tk.RIGHT, *fill*=tk.Y)

        # Connect the scrollbar to the text widget

*self*.chat\_display.config(*yscrollcommand*=scrollbar.set)

        scrollbar.config(*command*=*self*.chat\_display.yview)

        # Create the message input area at the bottom

        input\_frame = Frame(*self*.root)

        input\_frame.pack(*fill*=tk.X, *pady*=10)

*self*.message\_entry = Entry(input\_frame, *font*=("Helvetica", 12))

*self*.message\_entry.pack(*side*=tk.LEFT, *fill*=tk.X, *expand*=True, *padx*=(10, 0))

*self*.message\_entry.bind("<Return>", *self*.send\_message)  # Send on Enter key

        # Send message button

        send\_button = Button(

            input\_frame,

*text*="Send",

*font*=("Helvetica", 12),

*bg*="#2196F3",

*fg*="white",

*relief*="flat",

*command*=*self*.send\_message,

        )

        send\_button.pack(*side*=tk.LEFT, *padx*=10)

        # Send file button

        send\_file\_button = Button(

            input\_frame,

*text*="Send File",

*font*=("Helvetica", 12),

*bg*="#2196F3",

*fg*="white",

*relief*="flat",

*command*=*self*.select\_file,

        )

        send\_file\_button.pack(*side*=tk.RIGHT, *padx*=5)

    def send\_message(*self*, *event*=None):

        """

        Send a chat message to all connected peers.

        Called when the Send button is clicked or Enter is pressed in the message entry.

        Args:

            event: Optional keyboard event (when triggered by Enter key)

        """

        message = *self*.message\_entry.get().strip()

        if message:

            # Special case for quit command

            if message.lower() == "/quit":

*self*.on\_closing()

                return

            # Show the message in our own chat display

*self*.update\_chat\_display(f"{*self*.username}: {message}")

            # Broadcast to connected peers if we have a chat instance

            if *self*.chat\_instance:

*self*.chat\_instance.broadcast\_message(message)

            # Clear the input field

*self*.message\_entry.delete(0, tk.END)

    def update\_chat\_display(*self*, *message*):

        """

        Update the chat display with a new message.

        This method can be called from any thread, as it ensures thread-safe UI updates.

        Args:

            message: The message to display in the chat window

        """

        # If chat display hasn't been created yet, just print the message to console

        if not hasattr(*self*, "chat\_display") or *self*.chat\_display is None:

            print(f"Can't display message yet: {*message*}")

            return

        def \_update():

            """Inner function to perform the actual UI update"""

            try:

                # Enable editing of the text widget

*self*.chat\_display.config(*state*=tk.NORMAL)

                # Split the message into username and content

                if ": " in *message*:

                    username, content = *message*.split(": ", 1)

                else:

                    username, content = "System", *message*

                # Apply different styling based on who sent the message

                if username == "System":

                    # Center-align system messages and make them gray

*self*.chat\_display.tag\_configure(

                        "center",

*justify*="center",

*foreground*="gray",

*font*=("Helvetica", 10),

                    )

*self*.chat\_display.insert(tk.END, f"{content}\n\n", "center")

                elif username == *self*.username:

                    # Right-align your messages

*self*.chat\_display.tag\_configure(

                        "right",

*justify*="right",

*foreground*="blue",

*font*=("Helvetica", 12),

                    )

*self*.chat\_display.insert(tk.END, f"{username}:\n", "right")

*self*.chat\_display.insert(tk.END, f"{content}\n\n", "right")

                else:

                    # Left-align others' messages

*self*.chat\_display.tag\_configure(

                        "left",

*justify*="left",

*foreground*="green",

*font*=("Helvetica", 12),

                    )

*self*.chat\_display.insert(tk.END, f"{username}:\n", "left")

*self*.chat\_display.insert(tk.END, f"{content}\n\n", "left")

*self*.chat\_display.see(tk.END)  # Scroll to the end

*self*.chat\_display.config(*state*=tk.DISABLED)

            except tk.TclError as e:

                print(f"TclError in update\_chat\_display: {e}")

        # Ensure thread-safe UI updates

        if threading.current\_thread() is not threading.main\_thread():

*self*.root.after(0, \_update)

        else:

            \_update()

    def show\_peers(*self*):

        """

        Show a dialog with the list of peers and their status.

        """

        if not *self*.chat\_instance:

            return

        # If the peers window is already open, close it and create a new one

        if hasattr(*self*, "peers\_window") and *self*.peers\_window:

*self*.peers\_window.destroy()

        # Create a new Toplevel window for the peers list

*self*.peers\_window = tk.Toplevel(*self*.root)

*self*.peers\_window.title("Connected Peers")

*self*.peers\_window.geometry("300x400")

        # Create a Label for the title

        title\_label = Label(

*self*.peers\_window,

*text*="Connected Peers",

*font*=("Helvetica", 16, "bold"),

*pady*=10,

        )

        title\_label.pack()

        # Create a Frame to hold the peer list

        peers\_frame = Frame(*self*.peers\_window)

        peers\_frame.pack(*fill*=tk.BOTH, *expand*=True, *padx*=10, *pady*=10)

        # Create a scrollable canvas to hold the peers

        canvas = tk.Canvas(peers\_frame)

        scrollbar = Scrollbar(peers\_frame, *orient*="vertical", *command*=canvas.yview)

        scrollable\_frame = Frame(canvas)

        scrollable\_frame.bind(

            "<Configure>", lambda *e*: canvas.configure(*scrollregion*=canvas.bbox("all"))

        )

        canvas.create\_window((0, 0), *window*=scrollable\_frame, *anchor*="nw")

        canvas.configure(*yscrollcommand*=scrollbar.set)

        # Pack the canvas and scrollbar

        canvas.pack(*side*="left", *fill*="both", *expand*=True)

        scrollbar.pack(*side*="right", *fill*="y")

        # Add peer entries

        with *self*.chat\_instance.lock:

            peers\_list = list(*self*.chat\_instance.peers.items())

        if not peers\_list:

            no\_peers\_label = Label(

                scrollable\_frame, *text*="No peers connected", *font*=("Helvetica", 12)

            )

            no\_peers\_label.pack(*pady*=10)

        else:

            # Sort peers by status (online first) then by username

            peers\_list.sort(

*key*=lambda *x*: (*x*[1].get("status", "offline") != "online", *x*[0].lower())

            )

            for peer\_username, peer\_info in peers\_list:

                peer\_frame = Frame(scrollable\_frame, *borderwidth*=1, *relief*="groove")

                peer\_frame.pack(*fill*=tk.X, *pady*=5, *padx*=3)

                # Status indicator (colored circle)

                status = peer\_info.get("status", "offline")

                status\_color = (

                    "#4CAF50" if status == "online" else "#F44336"

                )  # Green if online, red if offline

                status\_indicator = Frame(

                    peer\_frame, *bg*=status\_color, *width*=10, *height*=10

                )

                status\_indicator.pack(*side*=tk.LEFT, *padx*=10, *pady*=10)

                # Username and address

                peer\_label = Label(

                    peer\_frame,

*text*=f"{peer\_username}",

*font*=("Helvetica", 12, "bold"),

                )

                peer\_label.pack(*side*=tk.LEFT, *padx*=5, *pady*=5)

                address\_label = Label(

                    peer\_frame,

*text*=f"({peer\_info['address']}:{peer\_info['port']})",

*font*=("Helvetica", 10),

*fg*="gray",

                )

                address\_label.pack(*side*=tk.LEFT, *padx*=5, *pady*=5)

                # Status text

                status\_text = "Online" if status == "online" else "Offline"

                status\_label = Label(

                    peer\_frame,

*text*=status\_text,

*font*=("Helvetica", 10),

*fg*=status\_color,

                )

                status\_label.pack(*side*=tk.RIGHT, *padx*=10, *pady*=5)

        # Add a close button at the bottom

        close\_button = Button(

*self*.peers\_window,

*text*="Close",

*font*=("Helvetica", 12),

*bg*="#2196F3",

*fg*="white",

*command*=lambda: *self*.peers\_window.destroy(),

        )

        close\_button.pack(*pady*=10)

    def on\_closing(*self*):

        """

        Handle the window close event.

        Clean up resources and disconnect from the chat network.

        """

        # Unregister from presence server if registered

        if (

*self*.presence\_client

            and hasattr(*self*.presence\_client, "registered")

            and *self*.presence\_client.registered

        ):

*self*.presence\_client.unregister()

        # Existing cleanup code

        if *self*.chat\_instance:

            try:

*self*.chat\_instance.disconnect()

            except Exception:

                pass

*self*.root.destroy()

        sys.exit(0)

def main():

    """

    Main entry point for the application.

    Creates the Tkinter root window and starts the ChatUI.

    """

    root = tk.Tk()

    app = ChatUI(root)

    root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

File Transfer.py

# file\_transfer.py

"""

File transfer functionality for P2P Chat.

This module handles sending and receiving files between peers by breaking

them into chunks that can be sent as messages.

"""

import os

import uuid

import base64

import socket

from utils import send\_message

CHUNK\_SIZE = 8192  # 8KB chunk size for file transfers

class FileTransfer:

    """

    Handles file transfer operations between peers.

    Features:

    - Break files into chunks for sending

    - Reassemble received chunks into complete files

    - Track file transfer progress

    - Notify UI about transfer status

    """

    def \_\_init\_\_(*self*, *message\_sender*, *downloads\_folder*="downloads", *ui\_callback*=None):

        """

        Initialize the FileTransfer system.

        Args:

            message\_sender: Function to send messages to peers

            downloads\_folder: Directory to save received files

            ui\_callback: Function to notify UI of transfer events

        """

*self*.message\_sender = *message\_sender*

*self*.downloads\_folder = *downloads\_folder*

*self*.ui\_callback = *ui\_callback*

        # Create downloads directory if it doesn't exist

        os.makedirs(*self*.downloads\_folder, *exist\_ok*=True)

        # Dictionary to track incoming file transfers

        # transfer\_id -> {"filename": str, "data": bytearray(), "sender": str}

*self*.incoming\_transfers = {}

    def send\_file(*self*, *file\_path*, *target\_addr*=None):

        """

        Send a file to a peer by breaking it into chunks.

        Args:

            file\_path: Path to the file to send

            target\_addr: (host, port) of the recipient

        """

        if not os.path.isfile(*file\_path*):

            print(f"[FileTransfer] File not found: {*file\_path*}")

            return

        # Get just the filename (not full path)

        file\_name = os.path.basename(*file\_path*)

        # Generate a unique ID for this transfer

        transfer\_id = str(uuid.uuid4())

        # Read and send file in chunks

        with open(*file\_path*, "rb") as f:

            while True:

                chunk = f.read(CHUNK\_SIZE)

                if not chunk:

                    # Send an empty chunk to indicate end of file

*self*.\_send\_chunk(transfer\_id, file\_name, b"", True, *target\_addr*)

                    break

*self*.\_send\_chunk(transfer\_id, file\_name, chunk, False, *target\_addr*)

    def handle\_incoming\_file\_chunk(*self*, *message*):

        """

        Process an incoming file chunk message and save completed files.

        Args:

            message: Dictionary containing file chunk data with fields:

                - type: "file\_chunk"

                - transfer\_id: Unique ID for this transfer

                - filename: Name of the file

                - data: Base64-encoded chunk data

                - is\_last: Boolean indicating if this is the last chunk

                - sender: Username of sender

        """

        try:

            # Extract message fields

            transfer\_id = *message*.get("transfer\_id")

            filename = *message*.get("filename")

            encoded\_data = *message*.get("data", "")

            is\_last = *message*.get("is\_last", False)

            sender = *message*.get("sender", "Unknown")  # Get sender's username

            # Validate required fields

            if not transfer\_id or not filename:

                print("[FileTransfer] Invalid file chunk message")

                return

            # Decode from base64 if it's a string

            data\_chunk = base64.b64decode(encoded\_data) if encoded\_data else b""

            # If this is the first chunk for this transfer, initialize entry

            if transfer\_id not in *self*.incoming\_transfers:

*self*.incoming\_transfers[transfer\_id] = {

                    "filename": filename,

                    "data": bytearray(),

                    "sender": sender,

                }

                # Notify UI when starting to receive a file

                if *self*.ui\_callback:

*self*.ui\_callback(f"Receiving file '{filename}' from {sender}...")

            # Add this chunk to the accumulated file data

*self*.incoming\_transfers[transfer\_id]["data"].extend(data\_chunk)

            # If this is the last chunk, save the completed file

            if is\_last:

                # Get the complete file data

                file\_data = *self*.incoming\_transfers[transfer\_id]["data"]

                save\_path = os.path.join(*self*.downloads\_folder, filename)

                # Write to disk

                with open(save\_path, "wb") as f:

                    f.write(file\_data)

                # Notify UI that file is complete

                if *self*.ui\_callback:

*self*.ui\_callback(

                        f"File '{filename}' received from {sender} and saved to {save\_path}"

                    )

                # Clean up the transfer

                del *self*.incoming\_transfers[transfer\_id]

                print(f"[FileTransfer] File '{filename}' saved to {save\_path}")

        except Exception as e:

            print(f"[FileTransfer] Error handling file chunk: {e}")

    def \_send\_chunk(

*self*, *transfer\_id*, *filename*, *chunk*, *is\_last\_chunk*, *target\_addr*=None

    ):

        """

        Create and send a file chunk message.

        Args:

            transfer\_id: Unique ID for the transfer

            filename: Name of the file

            chunk: Binary data to send

            is\_last\_chunk: Boolean flag for the last chunk

            target\_addr: (host, port) of recipient

        """

        # Encode binary data as base64 string

        if isinstance(*chunk*, bytes):

*chunk* = base64.b64encode(*chunk*).decode("ascii")

        # Create the message

        message = {

            "type": "file\_chunk",

            "transfer\_id": *transfer\_id*,

            "filename": *filename*,

            "data": *chunk*,

            "is\_last": *is\_last\_chunk*,

            "sender": "You",  # This will be replaced by the receiver

        }

        # Send the message to the specified target

        if *target\_addr*:

*self*.message\_sender(*target\_addr*, message)

        else:

            # Example broadcast logic would go here

            pass

    def \_send\_message\_to\_peer(*self*, *peer\_addr*, *message*: dict):

        """

        Send a message to a specific peer.

        Args:

            peer\_addr: (host, port) tuple of the recipient

            message: Dictionary containing the message

        """

        host, port = *peer\_addr*

        try:

            sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

            sock.connect((host, port))

            # Use the improved send\_message function

            send\_message(sock, *message*)

            sock.close()

            print(f"Sent chunk of size {len(*message*.get('data', ''))} to {*peer\_addr*}")

        except Exception as e:

            print(f"Error in \_send\_message\_to\_peer({*peer\_addr*}): {e}")

message\_handler.py

# message\_handler.py

import time

import socket

import json

from utils import send\_message

class MessageHandler:

    def \_\_init\_\_(*self*, *chat*):

*self*.chat = *chat*

    def handle\_message(

*self*, *client\_socket*: socket.socket, *address*: tuple, *message*: dict

    ):

        message\_type = *message*.get("type")

        sender = *message*.get("username")

        if message\_type == "join":

*self*.handle\_join(*client\_socket*, *address*, *message*, sender)

        elif message\_type == "request\_peers":

*self*.handle\_request\_peers(*client\_socket*)

        elif message\_type == "chat":

            print(f"\n{sender}: {*message*['content']}")  # Clearer display with newline

        elif message\_type == "heartbeat":

*self*.handle\_heartbeat(sender)

        elif message\_type == "leave":

*self*.handle\_leave(sender)

        elif message\_type == "new\_peer":

*self*.handle\_new\_peer(*message*)

        else:

            print(f"Unknown message type received from {sender}: {message\_type}")

    def handle\_join(

*self*, *client\_socket*: socket.socket, *address*: tuple, *message*: dict, *sender*: str

    ):

        with *self*.chat.lock:

*self*.chat.peers[*sender*] = {

                "address": *address*[0],

                "port": *message*["port"],

                "last\_seen": time.time(),  # Initialize last\_seen

            }

        send\_message(

*client\_socket*,

            {"type": "welcome", "username": *self*.chat.username, "port": *self*.chat.port},

        )

        print(f"\n{*sender*} joined the chat!")

        # Inform existing peers about the new peer

*self*.\_broadcast\_new\_peer(*sender*, *address*[0], *message*["port"])

    def handle\_request\_peers(*self*, *client\_socket*: socket.socket):

        # Send the list of known peers back to the new peer

        peers\_list = [

            {"username": peer, "address": info["address"], "port": info["port"]}

            for peer, info in *self*.chat.peers.items()

        ]

        send\_message(*client\_socket*, {"type": "peer\_list", "peers": peers\_list})

    def handle\_heartbeat(*self*, *sender*: str):

        with *self*.chat.lock:

            if *sender* in *self*.chat.peers:

*self*.chat.peers[*sender*]["last\_seen"] = time.time()

    def handle\_leave(*self*, *sender*: str):

        with *self*.chat.lock:

            if *sender* in *self*.chat.peers:

                del *self*.chat.peers[*sender*]

        print(f"\n{*sender*} left the chat.")

    def handle\_new\_peer(*self*, *message*: dict):

        # Handle new peer information received from another peer

        new\_peer\_username = *message*["username"]

        new\_peer\_address = *message*["address"]

        new\_peer\_port = *message*["port"]

        with *self*.chat.lock:

            if new\_peer\_username not in *self*.chat.peers:

*self*.chat.peers[new\_peer\_username] = {

                    "address": new\_peer\_address,

                    "port": new\_peer\_port,

                    "last\_seen": time.time(),

                }

                print(

                    f"\nDiscovered new peer: {new\_peer\_username} ({new\_peer\_address}:{new\_peer\_port})"

                )

    def \_broadcast\_new\_peer(

*self*, *new\_peer\_username*: str, *new\_peer\_address*: str, *new\_peer\_port*: int

    ):

        """Inform existing peers about a new peer that has joined."""

        msg\_data = {

            "type": "new\_peer",

            "username": *new\_peer\_username*,

            "address": *new\_peer\_address*,

            "port": *new\_peer\_port*,

        }

        with *self*.chat.lock:

            peers\_copy = *self*.chat.peers.copy()

        for peer\_username, peer\_info in peers\_copy.items():

            if peer\_username != *new\_peer\_username*:  # Don't send to the new peer itself

                try:

                    peer\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

                    peer\_socket.connect((peer\_info["address"], peer\_info["port"]))

                    send\_message(peer\_socket, msg\_data)

                    peer\_socket.close()

                except Exception as e:

                    print(f"Error sending new peer info to {peer\_username}: {e}")

                    # Consider removing unreachable peer here, or let heartbeat handle it

p2p\_chat.py

# p2p\_chat.py

"""

Core P2P Chat implementation.

This module provides the main functionality for peer-to-peer chat communication,

including connecting to peers, sending/receiving messages, and maintaining

peer connections through heartbeats.

"""

import sys

import socket

import json

import time

import random

import threading

from typing import Dict, Callable

from utils import send\_message, receive\_message

class P2PChat:

    """

    Main P2P Chat class that handles all peer-to-peer communication.

    This class is responsible for:

    - Starting a server to accept incoming peer connections

    - Connecting to existing peers

    - Sending and receiving messages

    - Maintaining peer connections with heartbeats

    - Tracking peer status (online/offline)

    """

    def \_\_init\_\_(

*self*,

*username*: str,

*host*: str = "0.0.0.0",

*port*: int = None,

*ui\_callback*: Callable = None,

*file\_chunk\_callback*: Callable = None,

*status\_callback*: Callable = None,

    ):

        """

        Initialize a new P2P Chat instance.

        Args:

            username: User's display name

            host: Host address to bind the server socket to

            port: Port number to use (random if not specified)

            ui\_callback: Function to call for UI notifications

            file\_chunk\_callback: Function to call when file chunks are received

            status\_callback: Function to call when peer status changes

        """

*self*.username = *username*

*self*.host = *host*

*self*.port = *port* or random.randint(

            49152, 65535

        )  # Use random high port if not specified

*self*.peers: Dict[str, dict] = {}  # Stores info about connected peers

*self*.connected = True

*self*.lock = threading.Lock()  # Thread safety for peer list access

*self*.ui\_callback = *ui\_callback*

*self*.status\_callback = *status\_callback*  # Callback for status changes

*self*.file\_chunk\_callback = *file\_chunk\_callback*

        # Set up server socket to accept incoming connections

*self*.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

*self*.server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

        try:

*self*.server\_socket.bind((*self*.host, *self*.port))

        except socket.error as e:

*self*.\_notify\_ui(f"Error binding socket: {e}. Exiting.")

            sys.exit(1)

*self*.server\_socket.listen(5)

        # Start the server thread to accept connections

*self*.server\_thread = threading.Thread(*target*=*self*.\_listen\_for\_connections)

*self*.server\_thread.daemon = True

*self*.server\_thread.start()

        # Start the heartbeat thread to maintain peer connections

*self*.heartbeat\_thread = threading.Thread(*target*=*self*.\_send\_heartbeat)

*self*.heartbeat\_thread.daemon = True

*self*.heartbeat\_thread.start()

*self*.\_notify\_ui(

            f"P2P Chat started on port {*self*.port}\nYour username: {*username*}"

        )

    def \_notify\_ui(*self*, *message*: str):

        """

        Send a notification message to the UI if a callback is defined.

        Args:

            message: The message to display in the UI

        """

        if *self*.ui\_callback:

*self*.ui\_callback(*message*)

    def \_listen\_for\_connections(*self*):

        """

        Listen for and accept incoming peer connections in a background thread.

        For each connection, start a new client handler thread.

        """

        while *self*.connected:

            try:

                client\_socket, address = *self*.server\_socket.accept()

                client\_handler = threading.Thread(

*target*=*self*.\_handle\_client, *args*=(client\_socket, address)

                )

                client\_handler.daemon = True

                client\_handler.start()

            except socket.error as e:

                if *self*.connected:

*self*.\_notify\_ui(f"Error accepting connection: {e}")

            except Exception as e:

*self*.\_notify\_ui(f"Unexpected error in \_listen\_for\_connections: {e}")

    def \_handle\_client(*self*, *client\_socket*: socket.socket, *address*: tuple):

        """

        Handle communication with a connected client in a dedicated thread.

        Args:

            client\_socket: Socket connection to the client

            address: Client's address as (ip, port) tuple

        """

        try:

            while *self*.connected:

                # Use the receive\_message function to read framed message

                message = receive\_message(*client\_socket*)

                if not message:

                    break  # Connection closed or error

                # Process the received message

*self*.\_handle\_message(*client\_socket*, *address*, message)

        except Exception as e:

*self*.\_notify\_ui(f"Error handling client {*address*}: {e}")

        finally:

*client\_socket*.close()

    def \_handle\_message(

*self*, *client\_socket*: socket.socket, *address*: tuple, *message*: dict

    ):

        """

        Process a message received from a peer.

        Args:

            client\_socket: Socket connection to respond on if needed

            address: Sender's address as (ip, port) tuple

            message: The received message as a dictionary

        """

        # Handle different message types with specific actions

        if *message*["type"] == "join":

            # New peer joining the network

            with *self*.lock:

*self*.peers[*message*["username"]] = {

                    "address": *address*[0],

                    "port": *message*["port"],

                    "last\_seen": time.time(),

                    "status": "online",  # Set initial status

                }

            # Send welcome message back

            send\_message(

*client\_socket*,

                {"type": "welcome", "username": *self*.username, "port": *self*.port},

            )

*self*.\_notify\_ui(f"{*message*['username']} joined the network.")

            # Notify status callback of new online peer

            if *self*.status\_callback:

*self*.status\_callback(*message*["username"], "online")

        elif *message*["type"] == "chat":

            # Regular chat message

*self*.\_notify\_ui(f"{*message*['username']}: {*message*['content']}")

        elif *message*["type"] == "heartbeat":

            # Heartbeat to keep connections alive and detect online peers

            with *self*.lock:

                if *message*["username"] in *self*.peers:

                    peer\_info = *self*.peers[*message*["username"]]

                    old\_status = peer\_info.get("status", "unknown")

                    peer\_info["last\_seen"] = time.time()

                    peer\_info["status"] = "online"

                    # Notify if status changed from offline to online

                    if old\_status != "online" and *self*.status\_callback:

*self*.status\_callback(*message*["username"], "online")

        elif *message*["type"] == "leave":

            # Peer is leaving the network

            with *self*.lock:

                if *message*["username"] in *self*.peers:

                    del *self*.peers[*message*["username"]]

*self*.\_notify\_ui(f"{*message*['username']} left the network.")

        elif *message*["type"] == "request\_peers":

            # Request for a list of known peers

            with *self*.lock:

                peers\_list = [

                    {"username": peer, "address": info["address"], "port": info["port"]}

                    for peer, info in *self*.peers.items()

                ]

            send\_message(*client\_socket*, {"type": "peer\_list", "peers": peers\_list})

        elif *message*["type"] == "file\_chunk":

            # File transfer chunk received

            # Add sender information for the UI

            if "sender" in *message* and *message*["sender"] == "You":

                # Replace "You" with the actual sender's username

*message*["sender"] = *message*.get("username", "Unknown")

            elif "sender" not in *message*:

*message*["sender"] = *message*.get("username", "Unknown")

            # Forward to the file chunk handler

            if *self*.file\_chunk\_callback:

*self*.file\_chunk\_callback(*message*)

        else:

            # Unknown message type

*self*.\_notify\_ui(f"Unknown message type: {*message*['type']} from {*address*}")

    def broadcast\_message(*self*, *message*: str):

        """

        Send a chat message to all connected peers.

        Args:

            message: The text message to send

        """

        msg\_data = {"type": "chat", "username": *self*.username, "content": *message*}

        with *self*.lock:

            peers\_copy = *self*.peers.copy()  # Make a copy to avoid concurrency issues

        # Send to each peer

        for peer\_username, peer\_info in peers\_copy.items():

            try:

                peer\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

                peer\_socket.connect((peer\_info["address"], peer\_info["port"]))

                send\_message(peer\_socket, msg\_data)

                peer\_socket.close()

            except Exception as e:

*self*.\_notify\_ui(f"Error sending message to {peer\_username}: {e}")

                # Remove unreachable peer

                with *self*.lock:

                    if peer\_username in *self*.peers:

                        del *self*.peers[peer\_username]

    def join\_network(*self*, *known\_host*: str, *known\_port*: int):

        """

        Join an existing P2P network via a known peer.

        Args:

            known\_host: IP address of known peer

            known\_port: Port number of known peer

        Returns:

            bool: True if successfully joined, False otherwise

        """

        try:

            # Connect to the known peer

            peer\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

            peer\_socket.connect((*known\_host*, *known\_port*))

            # Send join request with our username and port

            send\_message(

                peer\_socket,

                {"type": "join", "username": *self*.username, "port": *self*.port},

            )

            # Receive welcome message

            message = receive\_message(peer\_socket)

            if message and message.get("type") == "welcome":

                # Add the peer to our list

                with *self*.lock:

*self*.peers[message["username"]] = {

                        "address": *known\_host*,

                        "port": message["port"],

                        "last\_seen": time.time(),

                        "status": "online",

                    }

*self*.\_notify\_ui(

                    f"Successfully joined the network through {message['username']}."

                )

                # Request list of other peers

                send\_message(peer\_socket, {"type": "request\_peers"})

                # Receive and process peer list

                peer\_list\_msg = receive\_message(peer\_socket)

                if peer\_list\_msg and peer\_list\_msg.get("type") == "peer\_list":

                    with *self*.lock:

                        for peer in peer\_list\_msg["peers"]:

                            if peer["username"] != *self*.username:

*self*.peers[peer["username"]] = {

                                    "address": peer["address"],

                                    "port": peer["port"],

                                    "last\_seen": time.time(),

                                    "status": "online",

                                }

*self*.\_notify\_ui(

                        f"Received list of existing peers: {len(peer\_list\_msg['peers'])} peers found."

                    )

                    # Notify status callback of new online peer

                    if *self*.status\_callback:

*self*.status\_callback(message["username"], "online")

            else:

*self*.\_notify\_ui(

                    f"Unexpected or missing response when joining: {message}"

                )

            peer\_socket.close()

            return True

        except Exception as e:

*self*.\_notify\_ui(f"Error joining network: {e}")

            return False

    def \_send\_heartbeat(*self*):

        """

        Periodically send heartbeats to all peers and check their status.

        This background thread maintains the network by:

        1. Sending heartbeats to peers periodically

        2. Detecting when peers go offline

        3. Updating status information

        """

        while *self*.connected:

            current\_time = time.time()

            with *self*.lock:

                peers\_copy = *self*.peers.copy()  # Copy to avoid concurrent modification

            # Check each peer

            for peer\_username, peer\_info in peers\_copy.items():

                try:

                    # Check if peer might be offline (no heartbeat for 15 seconds)

                    if current\_time - peer\_info.get("last\_seen", 0) > 15:

                        old\_status = peer\_info.get("status", "unknown")

                        # Try to connect

                        peer\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

                        peer\_socket.settimeout(5)

                        peer\_socket.connect((peer\_info["address"], peer\_info["port"]))

                        send\_message(

                            peer\_socket,

                            {"type": "heartbeat", "username": *self*.username},

                        )

                        peer\_socket.close()

                        # Update last seen and ensure status is online

                        with *self*.lock:

                            if peer\_username in *self*.peers:

*self*.peers[peer\_username]["last\_seen"] = time.time()

*self*.peers[peer\_username]["status"] = "online"

                                # Notify if status changed

                                if old\_status != "online" and *self*.status\_callback:

*self*.status\_callback(peer\_username, "online")

                except Exception:

                    # Connection failed - mark as offline

                    with *self*.lock:

                        if peer\_username in *self*.peers:

                            old\_status = *self*.peers[peer\_username].get(

                                "status", "unknown"

                            )

*self*.peers[peer\_username]["status"] = "offline"

                            # Notify UI and status callback

                            if old\_status != "offline":

*self*.\_notify\_ui(

                                    f"{peer\_username} appears to be offline."

                                )

                                if *self*.status\_callback:

*self*.status\_callback(peer\_username, "offline")

            time.sleep(10)  # Check every 10 seconds

    def get\_online\_peers(*self*):

        """

        Get dictionary of currently online peers.

        Returns:

            dict: Filtered dictionary containing only online peers

        """

        with *self*.lock:

            return {

                username: info

                for username, info in *self*.peers.items()

                if info.get("status") == "online"

            }

    def disconnect(*self*):

        """

        Disconnect from the network and clean up resources.

        This method:

        1. Notifies all peers that we're leaving

        2. Stops all background threads

        3. Closes the server socket

        """

*self*.connected = False  # Signal threads to stop

        # Notify peers that we're leaving

        with *self*.lock:

            peers\_copy = *self*.peers.copy()

        for peer\_username, peer\_info in peers\_copy.items():

            try:

                peer\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

                peer\_socket.connect((peer\_info["address"], peer\_info["port"]))

                send\_message(peer\_socket, {"type": "leave", "username": *self*.username})

                peer\_socket.close()

            except Exception:

                pass  # Ignore errors on disconnect

        # Close the server socket

        try:

*self*.server\_socket.shutdown(socket.SHUT\_RDWR)

        except OSError:

            pass  # Socket might already be closed

        finally:

*self*.server\_socket.close()

peer\_status\_tracker.py

# peer\_status\_tracker.py

import time

from typing import Dict, Callable, Optional

class PeerStatusTracker:

    """

    Tracks the online/offline status of peers in the P2P network.

    """

    def \_\_init\_\_(*self*, *status\_change\_callback*: Optional[Callable] = None):

        # Dictionary to store peer status: username -> {"status": "online"|"offline", "last\_updated": timestamp}

*self*.peers: Dict[str, dict] = {}

*self*.status\_change\_callback = *status\_change\_callback*

    def update\_peer\_status(*self*, *username*: str, *status*: str):

        """Update a peer's status and invoke callback if it changed"""

        current\_time = time.time()

        # Check if this is a status change

        if *username* in *self*.peers:

            old\_status = *self*.peers[*username*].get("status")

            if old\_status != *status*:

                # Status changed, invoke callback

                if *self*.status\_change\_callback:

*self*.status\_change\_callback(*username*, *status*, old\_status)

        else:

            # New peer

            if *self*.status\_change\_callback:

*self*.status\_change\_callback(*username*, *status*, None)

        # Update the status

*self*.peers[*username*] = {"status": *status*, "last\_updated": current\_time}

    def get\_peer\_status(*self*, *username*: str) -> str:

        """Get a peer's current status"""

        if *username* in *self*.peers:

            return *self*.peers[*username*].get("status", "unknown")

        return "unknown"

    def get\_all\_peers(*self*) -> Dict[str, str]:

        """Get all peers and their statuses"""

        return {username: info["status"] for username, info in *self*.peers.items()}

    def get\_online\_peers(*self*) -> Dict[str, dict]:

        """Get all online peers"""

        return {

            username: info

            for username, info in *self*.peers.items()

            if info.get("status") == "online"

        }

    def get\_offline\_peers(*self*) -> Dict[str, dict]:

        """Get all offline peers"""

        return {

            username: info

            for username, info in *self*.peers.items()

            if info.get("status") == "offline"

        }

Presence\_client.py

# presence\_client.py

import socket

import threading

import time

from utils import send\_message, receive\_message

class PresenceClient:

    """Client for interacting with the presence server to register and discover peers"""

    def \_\_init\_\_(*self*, *username*, *port*, *presence\_server*="127.0.0.1", *presence\_port*=7000):

*self*.username = *username*

*self*.port = *port*  # User's P2P chat port

*self*.presence\_server = *presence\_server*

*self*.presence\_port = *presence\_port*

*self*.registered = False

*self*.heartbeat\_thread = None

*self*.running = False

    def register(*self*):

        """Register with the presence server"""

        try:

            sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

            sock.connect((*self*.presence\_server, *self*.presence\_port))

            # Get the local IP address that can be used by other peers

            # This is a simple approach - in a real-world scenario you might

            # need more sophisticated methods to get the correct external IP

            local\_ip = sock.getsockname()[0]

            if local\_ip == "0.0.0.0":

                local\_ip = "127.0.0.1"  # Fallback to localhost

            send\_message(

                sock,

                {

                    "type": "register",

                    "username": *self*.username,

                    "port": *self*.port,

                    "address": local\_ip,

                },

            )

            response = receive\_message(sock)

            sock.close()

            if response and response.get("success"):

*self*.registered = True

*self*.running = True

                # Start heartbeat thread

*self*.heartbeat\_thread = threading.Thread(*target*=*self*.\_send\_heartbeats)

*self*.heartbeat\_thread.daemon = True

*self*.heartbeat\_thread.start()

                return True

            else:

                reason = (

                    response.get("reason", "Unknown error")

                    if response

                    else "No response"

                )

                print(f"Registration failed: {reason}")

                return False

        except Exception as e:

            print(f"Error registering with presence server: {e}")

            return False

    def unregister(*self*):

        """Unregister from the presence server"""

*self*.running = False

        if not *self*.registered:

            return

        try:

            sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

            sock.connect((*self*.presence\_server, *self*.presence\_port))

            send\_message(sock, {"type": "unregister", "username": *self*.username})

            sock.close()

*self*.registered = False

            print("Unregistered from presence server")

        except Exception as e:

            print(f"Error unregistering from presence server: {e}")

    def get\_online\_users(*self*):

        """Get list of online users from the presence server"""

        try:

            sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

            sock.connect((*self*.presence\_server, *self*.presence\_port))

            send\_message(sock, {"type": "query"})

            response = receive\_message(sock)

            sock.close()

            if response and response.get("type") == "online\_users":

                # Filter out ourselves from the list

                return [

                    user

                    for user in response.get("users", [])

                    if user.get("username") != *self*.username

                ]

            return []

        except Exception as e:

            print(f"Error querying online users: {e}")

            return []

    def \_send\_heartbeats(*self*):

        """Send periodic heartbeats to the presence server"""

        while *self*.running:

            try:

                sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

                sock.connect((*self*.presence\_server, *self*.presence\_port))

                send\_message(sock, {"type": "heartbeat", "username": *self*.username})

                sock.close()

            except Exception as e:

                print(f"Error sending heartbeat: {e}")

            time.sleep(20)  # Send heartbeat every 20 seconds

presence\_server.py

# presence\_server.py

"""

Presence Server for the P2P Chat Application.

This server tracks which users are online and allows users to discover

each other without needing to know IP addresses and ports in advance.

"""

import socket

import threading

import json

import time

from utils import send\_message, receive\_message

class PresenceServer:

    """

    A server that keeps track of online users and their connection details.

    This server allows users to:

    - Register themselves as online

    - Discover other online users

    - Maintain their online status with heartbeats

    """

    def \_\_init\_\_(*self*, *host*="0.0.0.0", *port*=7000):

        """

        Initialize the presence server.

        Args:

            host: Host address to bind to (default: all interfaces)

            port: Port to listen on (default: 7000)

        """

*self*.host = *host*

*self*.port = *port*

*self*.running = False

*self*.server\_socket = None

*self*.online\_users = {}  # username -> {address, port, last\_seen}

*self*.lock = threading.Lock()

    def start(*self*):

        """

        Start the presence server and begin accepting connections.

        This method:

        1. Binds to the specified host and port

        2. Starts a cleanup thread to remove stale users

        3. Accepts and processes client connections

        """

*self*.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

*self*.server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

        try:

*self*.server\_socket.bind((*self*.host, *self*.port))

*self*.server\_socket.listen(5)

*self*.running = True

            print(f"Presence server running on {*self*.host}:{*self*.port}")

            # Start a thread to clean up stale users

            cleanup\_thread = threading.Thread(*target*=*self*.\_cleanup\_stale\_users)

            cleanup\_thread.daemon = True

            cleanup\_thread.start()

            # Accept client connections

            while *self*.running:

                try:

                    client\_socket, address = *self*.server\_socket.accept()

                    client\_thread = threading.Thread(

*target*=*self*.\_handle\_client, *args*=(client\_socket, address)

                    )

                    client\_thread.daemon = True

                    client\_thread.start()

                except Exception as e:

                    if *self*.running:

                        print(f"Error accepting connection: {e}")

        except Exception as e:

            print(f"Error starting presence server: {e}")

        finally:

            if *self*.server\_socket:

*self*.server\_socket.close()

    def stop(*self*):

        """Stop the presence server and clean up resources."""

*self*.running = False

        if *self*.server\_socket:

            try:

*self*.server\_socket.shutdown(socket.SHUT\_RDWR)

*self*.server\_socket.close()

            except Exception:

                pass  # Socket might already be closed

    def \_handle\_client(*self*, *client\_socket*, *address*):

        """

        Handle a client request.

        Args:

            client\_socket: Socket connected to the client

            address: Client's address as (ip, port) tuple

        """

        try:

            message = receive\_message(*client\_socket*)

            if not message:

                return

            msg\_type = message.get("type")

            # Process different request types

            if msg\_type == "register":

*self*.\_register\_user(message, *client\_socket*)

            elif msg\_type == "query":

*self*.\_send\_online\_users(*client\_socket*)

            elif msg\_type == "heartbeat":

*self*.\_update\_user\_heartbeat(message)

            elif msg\_type == "unregister":

*self*.\_unregister\_user(message)

        except Exception as e:

            print(f"Error handling client: {e}")

        finally:

*client\_socket*.close()

    def \_register\_user(*self*, *message*, *client\_socket*):

        """

        Register a new user or update existing user.

        Args:

            message: Dictionary with registration info

            client\_socket: Socket to send response on

        """

        username = *message*.get("username")

        user\_port = *message*.get("port")

        client\_ip = *message*.get("address")

        # Validate required fields

        if not username or not user\_port:

            send\_message(

*client\_socket*,

                {

                    "type": "register\_response",

                    "success": False,

                    "reason": "Missing required fields",

                },

            )

            return

        # Store user information

        with *self*.lock:

*self*.online\_users[username] = {

                "address": client\_ip,

                "port": user\_port,

                "last\_seen": time.time(),

            }

        send\_message(*client\_socket*, {"type": "register\_response", "success": True})

        print(f"Registered user: {username} at {client\_ip}:{user\_port}")

    def \_send\_online\_users(*self*, *client\_socket*):

        """Send list of online users to client"""

        with *self*.lock:

            user\_list = [

                {"username": username, "address": data["address"], "port": data["port"]}

                for username, data in *self*.online\_users.items()

            ]

        send\_message(*client\_socket*, {"type": "online\_users", "users": user\_list})

    def \_update\_user\_heartbeat(*self*, *message*):

        """Update user's last\_seen timestamp"""

        username = *message*.get("username")

        if username:

            with *self*.lock:

                if username in *self*.online\_users:

*self*.online\_users[username]["last\_seen"] = time.time()

    def \_unregister\_user(*self*, *message*):

        """Remove user from online list"""

        username = *message*.get("username")

        if username:

            with *self*.lock:

                if username in *self*.online\_users:

                    del *self*.online\_users[username]

                    print(f"Unregistered user: {username}")

    def \_cleanup\_stale\_users(*self*):

        """Periodically remove users who haven't sent heartbeats"""

        while *self*.running:

            time.sleep(30)  # Check every 30 seconds

            current\_time = time.time()

            stale\_threshold = 60  # Consider users stale after 60 seconds

            to\_remove = []

            with *self*.lock:

                for username, data in *self*.online\_users.items():

                    if current\_time - data["last\_seen"] > stale\_threshold:

                        to\_remove.append(username)

                for username in to\_remove:

                    del *self*.online\_users[username]

                    print(f"Removed stale user: {username}")

# When running as a standalone script, start the presence server

if \_\_name\_\_ == "\_\_main\_\_":

    server = PresenceServer()

    try:

        server.start()

    except KeyboardInterrupt:

        print("Stopping presence server...")

        server.stop()

start\_presence\_server.py

# start\_presence\_server.py

"""

Simple script to start the presence server.

Run this script before trying to use presence-based connections.

"""

from presence\_server import PresenceServer

if \_\_name\_\_ == "\_\_main\_\_":

    print("Starting P2P Chat Presence Server...")

    print("Press Ctrl+C to stop the server.")

    server = PresenceServer()

    try:

        server.start()

    except KeyboardInterrupt:

        print("\nStopping presence server...")

        server.stop()

        print("Server stopped.")

utils.py

# utils.py

"""

Utility functions for network communication in the P2P chat application.

This module provides message framing to ensure complete messages are sent and received.

"""

import socket

import json

import struct

def send\_message(*sock*: socket.socket, *message*: dict):

    """

    Send a message with proper length prefix for framing.

    This function handles the message framing protocol:

    1. Convert message dict to JSON

    2. Add a 4-byte length prefix

    3. Send the complete packet

    Args:

        sock: Socket connection to send message through

        message: Dictionary containing the message data

    """

    try:

        # Convert message to JSON string and encode to bytes

        data = json.dumps(*message*).encode()

        # Prefix with message length (4-byte integer in network byte order)

        length\_prefix = struct.pack("!I", len(data))

        # Send length prefix followed by the data

*sock*.sendall(length\_prefix + data)

    except Exception as e:

        print(f"Error sending message: {e}")

def receive\_message(*sock*: socket.socket) -> dict:

    """

    Receive a message with length prefix framing.

    This function handles the message framing protocol:

    1. Read the 4-byte length prefix

    2. Read the specified number of bytes

    3. Parse the JSON message

    Args:

        sock: Socket connection to receive message from

    Returns:

        Parsed message dictionary or None if an error occurred

    """

    try:

        # Set a timeout to prevent hanging forever

*sock*.settimeout(10.0)  # 10 seconds timeout

        # First read the 4-byte length prefix

        length\_bytes = *sock*.recv(4)

        if not length\_bytes:

            # This is a normal disconnection, no need to print anything

            return None

        if len(length\_bytes) < 4:

            # Only print for incomplete prefixes, not empty ones

            if len(length\_bytes) > 0:

                print(f"Incomplete length prefix received ({len(length\_bytes)} bytes)")

            return None

        # Unpack the length prefix to get the message size

        message\_length = struct.unpack("!I", length\_bytes)[0]

        # Sanity check to avoid allocating too much memory

        if message\_length > 100 \* 1024 \* 1024:  # 100MB limit

            print(f"Message too large: {message\_length} bytes")

            return None

        # Read the message data in chunks to handle large messages

        chunks = []

        bytes\_received = 0

        while bytes\_received < message\_length:

            chunk = *sock*.recv(min(message\_length - bytes\_received, 4096))

            if not chunk:

                print("Connection closed while receiving message data")

                return None  # Connection closed unexpectedly

            chunks.append(chunk)

            bytes\_received += len(chunk)

        # Combine all chunks and decode JSON

        data = b"".join(chunks)

        return json.loads(data.decode())

    except socket.timeout:

        # Socket timeouts are normal during polling, don't print

        return None

    except json.JSONDecodeError as e:

        print(f"JSON decode error: {e}")

        return None

    except Exception as e:

        # Only log actual errors

        print(f"Error receiving message: {e}")

        return None

    finally:

        # Reset timeout to blocking mode

        try:

*sock*.settimeout(None)

        except:

            pass  # Socket might be closed already