Module 22 - Network Programming In Python Using Sockets: Building A Chat Application

This module dives into the exciting world of network communication using Python's `socket` module. We will build a complete client-server chat application with a Graphical User Interface (GUI) using Tkinter, allowing users to interact and chat visually.

Chapter 1: Network Fundamentals

(Content for IP Address, Ports & Sockets remains the same as previous response)

1.1 IP Address

1.2 Ports & Sockets

Chapter 2: Building Blocks of a Chat Application (Client & Server)

(Content for Creating A Client, Sending Messages, Using Buffer, Building The Messaging Functionality, Sending Messages To Client, Programming Send Functionality remains the same as previous response, but the core implementation will be in the final "Completing Our Chat App" section for GUI.)

2.3 Creating A Client

2.4 Sending Messages

2.5 Using Buffer (`recv()`)

2.6 Building The Messaging Functionality (Server-side)

2.7 Sending Messages To Client (Server-side)

2.8 Programming Send Functionality (Refined Client/Server)

2.9 Completing Our Chat App (GUI with Tkinter - Enhanced)

Now, let's implement the full GUI chat application, ensuring both the server and clients have interactive interfaces and can send/receive messages.

Key Design Principles for GUI Chat:

- **Multithreading:** Essential for non-blocking socket operations (like `accept()` and `recv()`) so the GUI remains responsive.
- **Thread-Safe GUI Updates:** Tkinter widgets can only be safely modified from the main thread. We'll use `queue.Queue` to pass messages from background socket threads to the main Tkinter thread, which will then update the GUI.
- **Clear Status Indicators:** Provide visual feedback on connection status.
- **User-Friendly Input/Output:** Dedicated areas for chat messages and user input.

`tkinter_chat_server.py` (Enhanced GUI Server Application)

Python
```
import socket
import threading
import tkinter as tk
from tkinter import scrolledtext, messagebox, simpledialog
import queue # For thread-safe communication between threads and GUI

# --- Server Configuration ---
SERVER_IP = '0.0.0.0' # Listen on all available network interfaces
SERVER_PORT = 12345
BUFFER_SIZE = 1024
MAX_CLIENTS = 10 # Maximum number of clients the server can handle

# List to keep track of connected clients (client_socket, address, name)
# Use a lock to ensure thread-safe access to this list
clients = []
clients_lock = threading.Lock()

# Queue for messages to be displayed on the GUI (from background threads)
message_queue = queue.Queue()

class ChatServerGUI:
    def __init__(self, master):
        self.master = master
        master.title("Python Chat Server")
        master.geometry("700x600") # Increased size for better layout
        master.resizable(True, True) # Allow window resizing
        master.protocol("WM_DELETE_WINDOW", self.on_closing) # Handle
window close event

        self.is_server_running = False
        self.server_socket = None
        self.server_thread = None

        # --- Styling ---
        self.master.option_add('*Font', 'Arial 10')
        self.master.option_add('*Button.Font', 'Arial 10 bold')
        self.master.option_add('*Label.Font', 'Arial 10')
        self.master.option_add('*Entry.Font', 'Arial 10')
        self.master.option_add('*ScrolledText.Font', 'Consolas 9') #
Monospace for log

        # --- Top Frame for Controls ---
        self.control_frame = tk.Frame(master, bd=2, relief=tk.GROOVE,
padx=5, pady=5)
        self.control_frame.pack(fill=tk.X, pady=5)

        self.status_label = tk.Label(self.control_frame, text="Server
Status: Not Running", fg="red", font='Arial 12 bold')
        self.status_label.pack(side=tk.LEFT, padx=10, pady=5)

        self.start_button = tk.Button(self.control_frame, text="Start
Server", command=self.start_server_gui, bg="#4CAF50", fg="white")
        self.start_button.pack(side=tk.LEFT, padx=5)

        self.stop_button = tk.Button(self.control_frame, text="Stop
Server", command=self.stop_server_gui, state=tk.DISABLED, bg="#f44336",
fg="white")
        self.stop_button.pack(side=tk.LEFT, padx=5)

        self.clear_button = tk.Button(self.control_frame, text="Clear Log",
command=self.clear_log, bg="#2196F3", fg="white")
        self.clear_button.pack(side=tk.RIGHT, padx=5)

        # --- Chat Log Display ---
        self.log_label = tk.Label(master, text="Server Log & Broadcast
History:", font='Arial 10 bold')
        self.log_label.pack(pady=(10, 0))
        self.log_text = scrolledtext.ScrolledText(master, state='disabled',
wrap=tk.WORD, bg="#f0f0f0", fg="#333", relief=tk.SUNKEN, bd=2)
        self.log_text.pack(pady=5, padx=10, fill=tk.BOTH, expand=True)

        # --- Server Broadcast Input ---
        self.broadcast_frame = tk.Frame(master, bd=2, relief=tk.GROOVE,
padx=5, pady=5)
        self.broadcast_frame.pack(fill=tk.X, pady=5)

        self.message_entry = tk.Entry(self.broadcast_frame, width=50,
relief=tk.FLAT, bd=1, bg="white", fg="#333")
        self.message_entry.pack(side=tk.LEFT, padx=5, fill=tk.X,
expand=True)
        self.message_entry.bind("<Return>",
self.send_broadcast_message_event) # Bind Enter key

        self.send_button = tk.Button(self.broadcast_frame,
text="Broadcast", command=self.send_broadcast_message, bg="#FFC107",
fg="black")
        self.send_button.pack(side=tk.RIGHT, padx=5)

        # Start checking message queue periodically
        self.master.after(100, self.process_queue)

    def log_message(self, message, sender="SERVER"):
        """Appends a message to the scrolled text widget with a
timestamp."""
        timestamp = datetime.now().strftime("%H:%M:%S")
        formatted_message = f"[{timestamp}] <{sender}> {message}"
        self.log_text.config(state='normal') # Enable editing
        self.log_text.insert(tk.END, formatted_message + "\n")
        self.log_text.yview(tk.END) # Auto-scroll to the bottom
        self.log_text.config(state='disabled') # Disable editing

    def clear_log(self):
        """Clears the content of the log text widget."""
        self.log_text.config(state='normal')
        self.log_text.delete(1.0, tk.END)
        self.log_text.config(state='disabled')

    def start_server_gui(self):
        """Starts the server in a separate thread."""
        if not self.is_server_running:
            self.is_server_running = True
            self.status_label.config(text="Server Status: Starting...",
fg="orange")
            self.start_button.config(state=tk.DISABLED)
            self.stop_button.config(state=tk.NORMAL)
            self.message_entry.config(state=tk.NORMAL)
            self.send_button.config(state=tk.NORMAL)

            self.server_thread = threading.Thread(target=self._run_server)
            self.server_thread.daemon = True # Allows main program to exit
even if thread is running
            self.server_thread.start()
            self.log_message(f"Attempting to start server on
{SERVER_IP}:{SERVER_PORT}")

    def stop_server_gui(self):
        """Stops the server and cleans up."""
        if self.is_server_running:
            self.is_server_running = False
            self.status_label.config(text="Server Status: Stopping...",
fg="orange")
            self.stop_button.config(state=tk.DISABLED)
            self.start_button.config(state=tk.NORMAL)
            self.message_entry.config(state=tk.DISABLED)
            self.send_button.config(state=tk.DISABLED)

            # Close server socket to break accept() loop
            if self.server_socket:
                try:
                    self.server_socket.shutdown(socket.SHUT_RDWR)
                    self.server_socket.close()
                except OSError as e:
                    self.log_message(f"Error shutting down server socket:
{e}")
                self.server_socket = None

            # Close all client sockets
            with clients_lock:
                for client_sock, _, _ in clients: # Iterate (socket,
address, name)
                    try:
                        client_sock.shutdown(socket.SHUT_RDWR)
                        client_sock.close()
                    except OSError:
                        pass # Ignore errors if already closed
                clients.clear()

            self.log_message("Server stopped.")
            self.status_label.config(text="Server Status: Not Running",
fg="red")

    def _run_server(self):
        """Actual server logic running in a separate thread."""
        try:
            self.server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
            self.server_socket.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEADDR, 1)
            self.server_socket.bind((SERVER_IP, SERVER_PORT))
            self.server_socket.listen(MAX_CLIENTS)
            message_queue.put(f"[SERVER] Server listening on
{SERVER_IP}:{SERVER_PORT}")
            self.master.after(0, lambda:
self.status_label.config(text="Server Status: Running", fg="green"))

            while self.is_server_running:
                try:
                    self.server_socket.settimeout(0.5) # Set a small
timeout to check self.is_server_running
                    client_socket, client_address =
self.server_socket.accept()

                    # Check if max clients reached
                    with clients_lock:
                        if len(clients) >= MAX_CLIENTS:
                            message_queue.put(f"[SERVER] Max clients
reached. Refusing connection from {client_address}.")
                            client_socket.sendall("Server is full. Please
try again later.".encode('utf-8'))
                            client_socket.close()
                            continue

                    # Start a new thread for each client
                    client_handler =
threading.Thread(target=self._handle_client, args=(client_socket,
client_address))
                    client_handler.daemon = True
                    client_handler.start()
                    message_queue.put(f"[SERVER] Accepted connection from
{client_address}. Active threads: {threading.active_count() - 2}") # -2 for
main and server thread

                except socket.timeout:
                    continue # Timeout occurred, check
self.is_server_running again
                except OSError as e:
                    if self.is_server_running: # Only log if not
intentionally stopped
                        message_queue.put(f"[ERROR] Server accept error:
{e}")
                    break # Break loop on other OS errors (e.g., socket
closed)
                except Exception as e:
                    message_queue.put(f"[ERROR] Unexpected error in server
loop: {e}")
                    break

        except socket.error as e:
            message_queue.put(f"[ERROR] Server socket binding/setup error:
{e}")
            self.master.after(0, lambda:
self.status_label.config(text="Server Status: Error", fg="red"))
        finally:
            if self.server_socket:
                self.server_socket.close()
            message_queue.put("[SERVER] Server socket closed.")
            self.master.after(0, lambda:
self.status_label.config(text="Server Status: Not Running", fg="red"))
            self.master.after(0, lambda:
self.start_button.config(state=tk.NORMAL))
            self.master.after(0, lambda:
self.stop_button.config(state=tk.DISABLED))


    def _handle_client(self, client_socket, client_address):
        """Handles communication with a single client in its own thread."""
        client_name = f"Client-{client_address[1]}" # Default name

        try:
            # First message from client should be their name
            initial_data = client_socket.recv(BUFFER_SIZE).decode('utf-8')
            if initial_data.startswith("NAME:"):
                client_name = initial_data[len("NAME:"):]
                message_queue.put(f"[SERVER] Client {client_address}
identified as '{client_name}'.")
                self._broadcast_message(f"'{client_name}' has joined the
chat.", "SERVER")
            else:
                message_queue.put(f"[SERVER] Client {client_address} sent
unexpected initial data. Assuming name '{client_name}'.")
                # Handle the first message as a regular message if not a
name
                message_queue.put(f"[{client_name}] {initial_data}")
                self._broadcast_message(initial_data, client_name)

            with clients_lock:
                clients.append((client_socket, client_address,
client_name))

            while self.is_server_running:
                message = client_socket.recv(BUFFER_SIZE).decode('utf-8')
                if not message: # Client disconnected
                    break

                message_queue.put(f"[{client_name}] {message}")
                self._broadcast_message(message, client_name)

        except ConnectionResetError:
            message_queue.put(f"[DISCONNECTED] Client '{client_name}'
({client_address}) reset connection.")
        except Exception as e:
            message_queue.put(f"[ERROR] Error handling client
'{client_name}' ({client_address}): {e}")
        finally:
            with clients_lock:
                if (client_socket, client_address, client_name) in clients:
                    clients.remove((client_socket, client_address,
client_name))
            try:
                client_socket.close()
            except OSError:
                pass # Ignore if already closed
            message_queue.put(f"[DISCONNECTED] Client '{client_name}'
({client_address}) left.")
            self._broadcast_message(f"'{client_name}' has left the chat.",
"SERVER")
```

```
    def _broadcast_message(self, message, sender_name):
        """Sends a message to all connected clients except the sender."""
        # Format message as it should appear on clients
        full_message = f"<{sender_name}> {message}".encode('utf-8')

        with clients_lock:
            clients_to_remove = []
            for client_sock, addr, name in clients:
                if name != sender_name: # Don't send back to sender
                    try:
                        client_sock.sendall(full_message)
                    except Exception as e:
                        message_queue.put(f"[BROADCAST ERROR] Could not
send to '{name}' ({addr}): {e}")
                        clients_to_remove.append((client_sock, addr, name))
# Mark for removal

            # Remove clients that failed to receive
            for client_sock, addr, name in clients_to_remove:
                if (client_sock, addr, name) in clients: # Check again in
case it was already removed by its handler
                    clients.remove((client_sock, addr, name))
                    try:
                        client_sock.close()
                    except OSError:
                        pass
                    message_queue.put(f"[CLEANUP] Removed disconnected
client '{name}' ({addr}).")

    def send_broadcast_message_event(self, event=None):
        """Handles sending broadcast message when Enter key is pressed."""
        self.send_broadcast_message()

    def send_broadcast_message(self):
        """Sends the message from the server's input field to all connected
clients."""
        if not self.is_server_running:
            messagebox.showwarning("Server Not Running", "Server must be
running to broadcast messages.")
            return

        message = self.message_entry.get().strip()
        if not message:
            return # Don't send empty messages

        self.log_message(message, sender="SERVER (Broadcast)") # Log
server's own message
        self._broadcast_message(message, "SERVER") # Send to all clients
        self.message_entry.delete(0, tk.END) # Clear input field

    def process_queue(self):
        """Periodically checks the message queue and updates the GUI."""
        while not message_queue.empty():
            message = message_queue.get()
            self.log_message(message, sender="SERVER Log") # Use a distinct
sender for internal logs
        self.master.after(100, self.process_queue) # Check again after
100ms
```

```
    def on_closing(self):
        """Handles closing the Tkinter window."""
        if self.is_server_running:
            if messagebox.askokcancel("Quit Server", "Server is running. Do
you want to stop it and quit?"):
                self.stop_server_gui() # Ensure server is stopped
gracefully
                self.master.destroy()
        else:
            if messagebox.askokcancel("Quit Server", "Do you want to
quit?"):
                self.master.destroy()

if __name__ == "__main__":
    from datetime import datetime # Import here for logging timestamps
    root = tk.Tk()
    app = ChatServerGUI(root)
    root.mainloop()
```

`tkinter_chat_client.py` (Enhanced GUI Client Application)

Python
```
import socket
import threading
import tkinter as tk
from tkinter import scrolledtext, messagebox, simpledialog
import queue # For thread-safe communication with GUI

# --- Client Configuration ---
DEFAULT_SERVER_IP = '127.0.0.1' # Default server IP for convenience
DEFAULT_SERVER_PORT = 12345     # Default server Port for convenience
BUFFER_SIZE = 1024

message_queue = queue.Queue() # Queue for messages to be displayed on GUI

class ChatClientGUI:
    def __init__(self, master):
        self.master = master
        master.title("Python Chat Client")
        master.geometry("550x650") # Increased size
        master.resizable(True, True)
        master.protocol("WM_DELETE_WINDOW", self.on_closing)

        self.client_socket = None
        self.receive_thread = None
        self.is_connected = False
        self.client_name = ""

        # --- Styling ---
        self.master.option_add('*Font', 'Arial 10')
        self.master.option_add('*Button.Font', 'Arial 10 bold')
        self.master.option_add('*Label.Font', 'Arial 10')
        self.master.option_add('*Entry.Font', 'Arial 10')
        self.master.option_add('*ScrolledText.Font', 'Consolas 9') #
Monospace for chat

        # --- Connection/Name Frame ---
        self.conn_name_frame = tk.Frame(master, bd=2, relief=tk.GROOVE,
padx=5, pady=5)
```

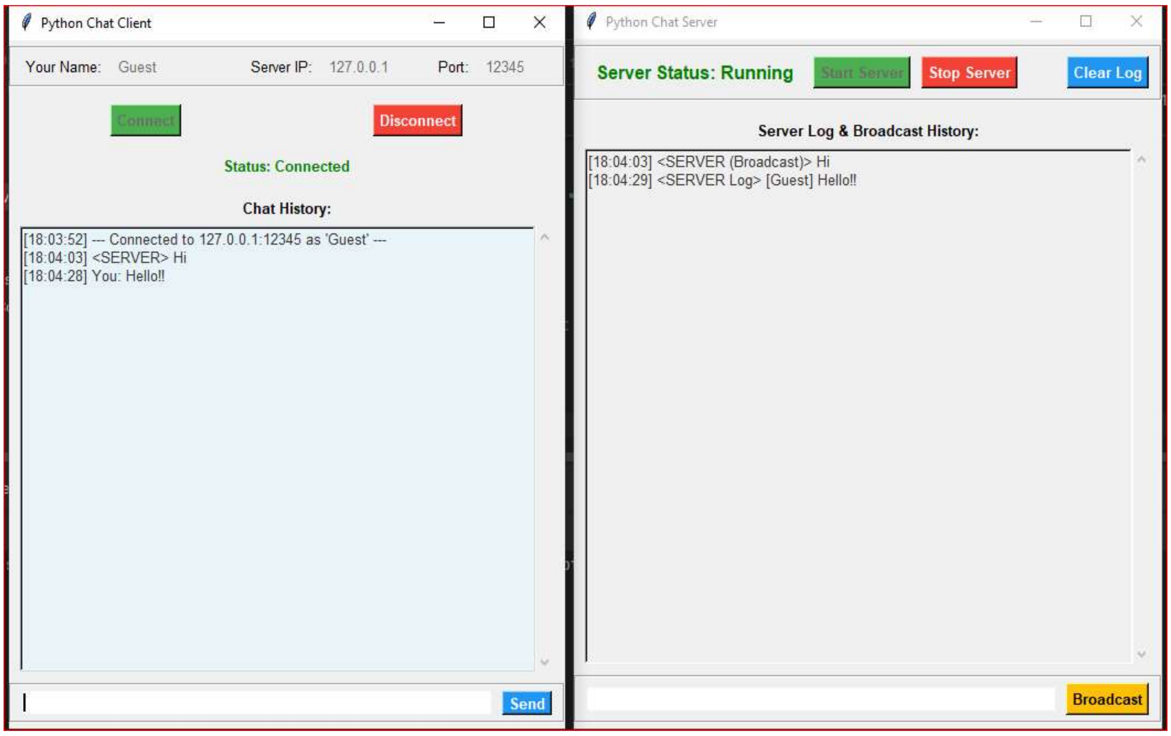
```
        self.conn_name_frame.pack(fill=tk.X, pady=5)

        self.name_label = tk.Label(self.conn_name_frame, text="Your Name:")
        self.name_label.pack(side=tk.LEFT, padx=5)
        self.name_entry = tk.Entry(self.conn_name_frame, width=15,
relief=tk.FLAT, bd=1, bg="white", fg="#333")
        self.name_entry.pack(side=tk.LEFT, padx=5)
        self.name_entry.insert(0, "Guest") # Default name

        self.ip_label = tk.Label(self.conn_name_frame, text="Server IP:")
        self.ip_label.pack(side=tk.LEFT, padx=5)
        self.ip_entry = tk.Entry(self.conn_name_frame, width=12,
relief=tk.FLAT, bd=1, bg="white", fg="#333")
        self.ip_entry.pack(side=tk.LEFT, padx=5)
        self.ip_entry.insert(0, DEFAULT_SERVER_IP)

        self.port_label = tk.Label(self.conn_name_frame, text="Port:")
        self.port_label.pack(side=tk.LEFT, padx=5)
        self.port_entry = tk.Entry(self.conn_name_frame, width=7,
relief=tk.FLAT, bd=1, bg="white", fg="#333")
        self.port_entry.pack(side=tk.LEFT, padx=5)
        self.port_entry.insert(0, str(DEFAULT_SERVER_PORT))

        # --- Connect/Disconnect Buttons ---
        self.button_frame = tk.Frame(master, padx=5, pady=5)
        self.button_frame.pack(fill=tk.X)

        self.connect_button = tk.Button(self.button_frame, text="Connect",
command=self.connect_to_server, bg="#4CAF50", fg="white")
        self.connect_button.pack(side=tk.LEFT, padx=10, pady=5,
expand=True)

        self.disconnect_button = tk.Button(self.button_frame,
text="Disconnect", command=self.disconnect_from_server, state=tk.DISABLED,
bg="#f44336", fg="white")
        self.disconnect_button.pack(side=tk.LEFT, padx=10, pady=5,
expand=True)

        self.status_label = tk.Label(master, text="Status: Disconnected",
fg="red", font='Arial 10 bold')
        self.status_label.pack(pady=5)

        # --- Chat Display Area ---
        self.chat_label = tk.Label(master, text="Chat History:",
font='Arial 10 bold')
        self.chat_label.pack(pady=(10, 0))
        self.chat_display = scrolledtext.ScrolledText(master,
state='disabled', wrap=tk.WORD, bg="#e8f4f8", fg="#333", relief=tk.SUNKEN,
bd=2)
        self.chat_display.pack(pady=5, padx=10, fill=tk.BOTH, expand=True)

        # --- Message Input Area ---
        self.message_input_frame = tk.Frame(master, bd=2, relief=tk.GROOVE,
padx=5, pady=5)
        self.message_input_frame.pack(fill=tk.X, pady=5)

        self.message_entry = tk.Entry(self.message_input_frame, width=40,
relief=tk.FLAT, bd=1, bg="white", fg="#333")
        self.message_entry.pack(side=tk.LEFT, padx=5, fill=tk.X,
expand=True)
```

```
        self.message_entry.bind("<Return>", self.send_message_event) # Bind
Enter key to send

        self.send_button = tk.Button(self.message_input_frame, text="Send",
command=self.send_message, bg="#2196F3", fg="white")
        self.send_button.pack(side=tk.RIGHT, padx=5)

        self.set_ui_state(connected=False) # Initial UI state

        # Start checking message queue periodically
        self.master.after(100, self.process_queue)

    def set_ui_state(self, connected):
        """Adjusts UI elements based on connection status."""
        self.is_connected = connected
        if connected:
            self.name_entry.config(state=tk.DISABLED)
            self.ip_entry.config(state=tk.DISABLED)
            self.port_entry.config(state=tk.DISABLED)
            self.connect_button.config(state=tk.DISABLED)
            self.disconnect_button.config(state=tk.NORMAL)
            self.message_entry.config(state=tk.NORMAL)
            self.send_button.config(state=tk.NORMAL)
            self.status_label.config(text="Status: Connected", fg="green")
            self.message_entry.focus_set() # Focus on message input for
immediate typing
        else:
            self.name_entry.config(state=tk.NORMAL)
            self.ip_entry.config(state=tk.NORMAL)
            self.port_entry.config(state=tk.NORMAL)
            self.connect_button.config(state=tk.NORMAL)
            self.disconnect_button.config(state=tk.DISABLED)
            self.message_entry.config(state=tk.DISABLED)
            self.send_button.config(state=tk.DISABLED)
            self.status_label.config(text="Status: Disconnected", fg="red")

    def display_message(self, message):
        """Appends a message to the chat display with a timestamp."""
        timestamp = datetime.now().strftime("%H:%M:%S")
        formatted_message = f"[{timestamp}] {message}"
        self.chat_display.config(state='normal')
        self.chat_display.insert(tk.END, formatted_message + "\n")
        self.chat_display.yview(tk.END)
        self.chat_display.config(state='disabled')

    def connect_to_server(self):
        """Initiates connection to the server."""
        self.client_name = self.name_entry.get().strip()
        if not self.client_name:
            messagebox.showerror("Error", "Please enter your name to join
the chat.")
            return

        server_ip = self.ip_entry.get().strip()
        server_port_str = self.port_entry.get().strip()

        if not server_ip or not server_port_str:
            messagebox.showerror("Error", "Server IP and Port are
required.")
            return
        try:
            server_port = int(server_port_str)
            if not (1024 <= server_port <= 65535):
                raise ValueError("Port number out of valid range (1024-
65535).")
        except ValueError as e:
            messagebox.showerror("Error", f"Invalid Port number: {e}")
            return

        self.client_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        try:
            self.client_socket.connect((server_ip, server_port))
            self.set_ui_state(connected=True)
            self.display_message(f"--- Connected to
{server_ip}:{server_port} as '{self.client_name}' ---")

            # Send client name to server immediately after connection

self.client_socket.sendall(f"NAME:{self.client_name}".encode('utf-8'))

            self.receive_thread =
threading.Thread(target=self._receive_messages)
            self.receive_thread.daemon = True # Allow main program to exit
            self.receive_thread.start()

        except ConnectionRefusedError:
            messagebox.showerror("Connection Error", "Connection refused.
Make sure the server is running and reachable.")
            self.set_ui_state(connected=False)
        except Exception as e:
            messagebox.showerror("Connection Error", f"An unexpected error
occurred during connection: {e}")
            self.set_ui_state(connected=False)

    def disconnect_from_server(self):
        """Disconnects from the server."""
        if self.client_socket:
            try:
                # Send a disconnect message to the server (optional, but
polite)
                self.client_socket.sendall(f"<{self.client_name}> has left
the chat.".encode('utf-8'))
                self.client_socket.shutdown(socket.SHUT_RDWR) # Signal
shutdown
                self.client_socket.close()
            except OSError:
                pass # Ignore errors if socket is already closed or
connection was reset
            except Exception as e:
                print(f"Error during socket shutdown/close: {e}")
            self.client_socket = None
        self.set_ui_state(connected=False)
        self.display_message("--- Disconnected from server ---")
        self.client_name = "" # Clear name on disconnect


    def _receive_messages(self):
        """Thread function to continuously receive messages from the
server."""
        while self.is_connected:
            try:
                message = self.client_socket.recv(BUFFER_SIZE).decode('utf-
8')
                if not message: # Server disconnected or sent empty message
                    message_queue.put("[DISCONNECTED] Server disconnected.
Please reconnect.")
                    self.master.after(0, self.disconnect_from_server) #
Schedule UI update on main thread
                    break
                message_queue.put(message) # Put message into queue for GUI
update
            except ConnectionResetError:
                message_queue.put("[DISCONNECTED] Server reset connection.
Please reconnect.")
                self.master.after(0, self.disconnect_from_server)
                break
            except Exception as e:
                message_queue.put(f"[ERROR] Error receiving message: {e}")
                self.master.after(0, self.disconnect_from_server)
                break

    def send_message_event(self, event=None):
        """Handles sending message when Enter key is pressed."""
        self.send_message()

    def send_message(self):
        """Sends the message from the input field to the server."""
        if not self.is_connected or not self.client_socket:
            messagebox.showwarning("Not Connected", "You are not connected
to the server.")
            return

        message = self.message_entry.get().strip()
        if not message:
            return # Don't send empty messages

        try:
            # Send the message (server will prepend sender name for
broadcast)
            self.client_socket.sendall(message.encode('utf-8'))
            self.display_message(f"You: {message}") # Display own message
immediately
            self.message_entry.delete(0, tk.END) # Clear input field
        except Exception as e:
            messagebox.showerror("Send Error", f"Failed to send message:
{e}")
            self.disconnect_from_server() # Disconnect on send error

    def process_queue(self):
        """Periodically checks the message queue and updates the GUI."""
        while not message_queue.empty():
            message = message_queue.get()
            self.display_message(message)
        self.master.after(100, self.process_queue) # Check again after
100ms

    def on_closing(self):
        """Handles closing the Tkinter window."""
        if self.is_connected:
            if messagebox.askokcancel("Quit Chat", "You are connected. Do
you want to disconnect and quit?"):
```

```
                self.disconnect_from_server() # Ensure socket is closed
                self.master.destroy()
        else:
            if messagebox.askokcancel("Quit Chat", "Do you want to quit?"):
                self.master.destroy()

if __name__ == "__main__":
    from datetime import datetime # Import here for timestamps
    root = tk.Tk()
    app = ChatClientGUI(root)
    root.mainloop()
```

How to Run the Enhanced GUI Chat Application:

1. **Save the files:**
   - Save the server GUI code as `tkinter_chat_server.py`.
   - Save the client GUI code as `tkinter_chat_client.py`.
   - Place them in the same directory.
2. **Start the server:**
   - Open your terminal or command prompt.
   - Navigate to the directory where you saved the files.
   - Run: `python tkinter_chat_server.py`
   - A Tkinter window titled "Python Chat Server" will appear. Click the **"Start Server"** button. The status will change to "Running".
3. **Start one or more clients:**
   - Open **new** terminal/command prompt windows for each client you want to run.
   - In each new terminal, navigate to the same directory.
   - Run: `python tkinter_chat_client.py`
   - A Tkinter window titled "Python Chat Client" will appear for each.
4. **Connect and Chat!**
   - In each client window:
     - Enter a unique **"Your Name"** (e.g., Alice, Bob, Charlie).
     - Ensure the "Server IP" and "Port" are correct (default `127.0.0.1:12345` for local testing).
     - Click the **"Connect"** button.
   - Once connected, the status will turn green. Type messages in the input field at the bottom and press **Enter** or click **"Send"**.
   - Messages will appear in:
     - The **server's log** window.
     - The chat display of **all connected clients** (including the sender's own window).



```
[18:03:52] --- Connected to 127.0.0.1:12345 as 'Guest' ---
[18:04:03] <SERVER> Hi
[18:04:28] You: Hello!!
```

```
[18:04:03] <SERVER (Broadcast)> Hi
[18:04:29] <SERVER Log> [Guest] Hello!!
```

5. **Disconnect/Quit:**
   - **Client:** Use the **"Disconnect"** button or close the client window (which will prompt for confirmation) to gracefully close the client connection.
   - **Server:** Click the **"Stop Server"** button on the server GUI to shut it down. Closing the server window will also prompt for confirmation to stop the server.