Universitatea Națională de Știință și Tehnologie POLITEHNICA București

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Proiect RPC Chat în Python

Profesor coordonator

Elena-Cristina STOICA

Studentă,

Lavinia-Marilena ANGAN

Anul 2025

Cuprins

Lista figurilor

Lista acronimelor

RPyC (Remote Python Call) – Apel Python la Distanță

RPC (Remote Procedure Call) – Apel de Procedură la Distanță

TCP (Transmission Control Protocol) – Protocol de Control al Transmisiei

UDP (User Datagram Protocol) – Protocol de Datagrame pentru Utilizatori

VS Code (Visual Studio Code) – Visual Studio Code

AMD (Advanced Micro Devices) – Dispozitive Micro Avansate

RAM (Random Access Memory) – Memorie cu Acces Aleator

Tkinter (Tool Command Language Interface) – Interfață pentru Limbajul de Comandă a Instrumentelor

Introducere

Acest proiect implementează un sistem de chat client-server utilizând biblioteca RPyC (Remote Python Call) în limbajul Python, cu o interfață grafică bazată pe Tkinter (Tool Command Language Interface). Sistemul permite mai multor utilizatori să se conecteze la un server central, să trimită mesaje și să le primească în timp real, simulând o aplicație de chat distribuită. Serverul gestionează mesajele și clienții conectați, în timp ce clientul oferă o interfață grafică intuitivă pentru interacțiunea utilizatorului. Proiectul a fost dezvoltat pentru a explora comunicarea între procese utilizând RPyC, gestionarea firelor de execuție și crearea unei interfețe grafice funcționale, toate integrate într-o aplicație practică.

Motivație

Motivația principală pentru realizarea acestui proiect a fost dorința de a înțelege și aplica concepte de comunicare client-server într-un context real, utilizând Python, un limbaj accesibil și versatil. Alegerea RPyC ca metodă de comunicare a fost determinată de simplitatea sa în implementarea apelurilor de procedură la distanță, comparativ cu alte tehnologii mai complexe. De asemenea, am fost interesat să combin aceste concepte cu o interfață grafică, pentru a crea o aplicație care să fie nu doar funcțională, ci și utilizabilă de către un utilizator obișnuit. Proiectul a oferit o oportunitate de a exersa gestionarea firelor de execuție pentru a asigura o experiență fluidă în interfața grafică, precum și de a depana probleme practice, cum ar fi erorile de conexiune și timeout-urile.

Obiective

Obiectivele proiectului au fost următoarele:

1. **Implementarea unui sistem de chat funcțional**: Crearea unui server care să gestioneze mai mulți clienți și să permită schimbul de mesaje în timp real.
2. **Dezvoltarea unei interfețe grafice**: Utilizarea Tkinter pentru a oferi utilizatorilor o interfață intuitivă, cu o zonă de afișare a mesajelor, un câmp de introducere și un buton de trimitere.
3. **Asigurarea stabilității conexiunii**: Gestionarea erorilor de conexiune, cum ar fi timeout-urile și deconectările, prin implementarea unui mecanism de reconectare și ajustarea timeout-urilor RPyC.
4. **Optimizarea performanței**: Reducerea frecvenței polling-ului pentru a minimiza încărcarea serverului, asigurând în același timp o experiență de chat în timp real.
5. **Documentarea procesului**: Crearea unei documentații detaliate care să includă bibliotecile utilizate, suportul hardware și software, capturi de ecran ale codului și rezultatelor, precum și concluzii bazate pe experiența individuală.

Aceste obiective au fost stabilite pentru a asigura că proiectul nu doar îndeplinește cerințele funcționale, ci și oferă o bază solidă pentru învățarea și aplicarea conceptelor de programare distribuită și interfețe grafice.

Biblioteci și Pachete Utilizate

1.1. RPyC (Remote Python Call)

RPyC este o bibliotecă Python care permite apeluri de procedură la distanță (Remote Procedure Call) între procese sau mașini.[1] În acest proiect, RPyC este utilizat pentru a facilita comunicarea bidirecțională între server (*server.py*) și clienți (*client.py*), permițând trimiterea și primirea mesajelor în timp real.

Module utilizate:

- **rpyc**: Modulul principal al bibliotecii, utilizat pentru stabilirea conexiunilor și definirea serviciilor.

În *client.py*, funcția ***rpyc.connect*** este utilizată pentru a conecta clientul la server:

```
# Configurare timeout-uri mai lungi
self.conn = rpyc.connect(
    "localhost",
    18813,
    service=ClientService,
    config={
        "sync_request_timeout": 30,  # 30 secunde pentru cereri sincrone
        "allow_public_attrs": True,
        "allow_pickle": True
    }
```

Figura 1.1 *Funcția rpyc.connect*

Aici, ***rpyc.connect*** creează o conexiune la serverul care rulează pe localhost pe portul 18813, folosind serviciul ***ClientService***. Parametrul *config* ajustează timeout-ul pentru cereri sincrone la 30 de secunde.

- **rpyc.utils.server.ThreadedServer**: Clasă utilizată în *server.py* pentru a crea un server care gestionează mai mulți clienți în fire de execuție separate.

Exemplu din *server.py*:

```
server = ThreadedServer(ChatService, port=18813, reuse_addr=True)
print("Starting RPC Chat Server on port 18813...")
server.start()
```

Figura 1.2 *Clasa ThreadedServer*

ThreadedServer inițializează un server care rulează serviciul *ChatService* pe portul 18813. Opțiunea ***reuse_addr=True*** permite refolosirea portului în cazul în care serverul este repornit rapid.

- **rpyc.Service**: Clasă de bază pentru definirea serviciilor expuse, utilizată atât în *server.py* (pentru ChatService), cât și în *client.py* (pentru ClientService).
-

În server.py, *ChatService* definește metodele expuse pentru clienți:

```
class ChatService(rpyc.Service):
    def exposed_send_message(self, sender, message):
        """Metodă expusă pentru trimiterea mesajelor"""
        with self.lock:
            formatted_message = f"[{sender}]: {message}"
            self.messages.append(formatted_message)
            print(f"Received message: {formatted_message}")
            self.broadcast_message(formatted_message)
            return f"Message from {sender} received"
```

Figura 1.3 *Clasa ChatService in server.py*

Metoda ***exposed_send_message*** este accesibilă clienților și permite trimiterea unui mesaj de la un client către server, care apoi îl distribuie tuturor clienților conectați.

În client.py, *ClientService* definește o metodă expusă pentru a primi mesaje de la server:

```python
class ClientService(rpyc.Service):
    def exposed_receive_message(self, message):
        """Metodă expusă pentru primirea mesajelor de la server"""
        if app:
            app.display_message(f"Received: {message}")
```

Figura 1.4 *Clasa ChatService in client.py*

Metoda ***exposed_receive_message*** este apelată de server pentru a notifica clientul despre un mesaj nou.

- **rpyc.connect**: Funcție utilizată în *client.py* pentru a stabili conexiunea client-server, așa cum s-a arătat mai sus.

1.2. Tkinter

Tkinter este biblioteca standard Python pentru crearea interfețelor grafice. În *client.py*, Tkinter este utilizat pentru a construi fereastra de chat, oferind utilizatorului o interfață grafică pentru a trimite și primi mesaje.

Module utilizate:

- **tkinter**: Modulul principal pentru crearea ferestrelor și widget-urilor. Widget-uri precum *Entry* (câmp de introducere) și *Button* (buton) sunt utilizate pentru a permite utilizatorului să introducă și să trimită mesaje.[7]
- **tkinter.scrolledtext.ScrolledText**: Widget pentru afișarea textului cu derulare, folosit pentru zona de chat. ScrolledText creează o zonă de text cu derulare automată, unde sunt afișate mesajele primite și trimise.

1.3. Threading

Biblioteca threading este utilizată pentru a gestiona fire de execuție, permițând executarea simultană a mai multor activități, cum ar fi polling-ul mesajelor în fundal fără a bloca interfața grafică.

Utilizarea firelor de execuție (threading) în Python reprezintă un concept esențial pentru rularea simultană a mai multor activități în cadrul aceluiași program.[4] În contextul proiectului RPC Chat, threading-ul joacă un rol crucial în gestionarea polling-ului mesajelor și a comunicării client-server, iar ideile din articol pot fi aplicate pentru a înțelege și îmbunătăți implementarea.

Module utilizate:

- **threading.Thread**: Clasă pentru crearea și gestionarea firelor de execuție.

- **threading.Lock**: Obiect pentru sincronizarea accesului la resurse partajate, utilizat în server.py pentru a preveni accesul concurent la lista de mesaje și clienți.

1.4. Time

Modulul time este utilizat pentru a introduce întârzieri în execuție, esențiale pentru controlul frecvenței polling-ului și gestionarea reconectărilor. Funcția utilizată este ***time.sleep(seconds)*** care suspendă execuția pentru un număr de secunde.

1.5. Socket

Socket-urile sunt puncte de comunicare la nivel de rețea care permit schimbul de date între procese, fie pe aceeași mașină, fie pe mașini diferite, utilizând protocoale precum TCP (Transmission Control Protocol) sau UDP (User Datagram Protocol). Socket-urile TCP, utilizate în mod implicit de RPyC în proiectul RPC Chat, oferă o conexiune fiabilă, orientată pe flux, care garantează livrarea datelor în ordine și fără pierderi. [5] Acestea sunt fundamentale pentru aplicațiile de rețea, cum ar fi serverele web, aplicațiile de chat sau transferul de fișiere, deoarece permit comunicarea bidirecțională între un client și un server prin trimiterea și primirea de mesaje.

Modulul socket este utilizat pentru a gestiona erorile de conexiune la nivel de rețea, în special în contextul stabilirii conexiunii cu serverul. Clasa utilizată este ***socket.timeout*** sugerează o excepție ridicată în caz de timeout al conexiunii.

În *client.py*, această excepție este gestionată în metoda ***connect_to_server_with_retry***:

```
try:
    # Configurare timeout-uri mai lungi
    self.conn = rpyc.connect(
        "localhost",
        18813,
        service=ClientService,
        config={
            "sync_request_timeout": 30,  # 30 secunde pentru cereri sincrone
            "allow_public_attrs": True,
            "allow_pickle": True
        }
    )
    self.is_connected = True
    self.display_message(f"Successfully connected to server on attempt {attempt + 1}.")
    return
except (socket.timeout, TimeoutError, ConnectionRefusedError) as e:
    self.display_message(f"Connection attempt {attempt + 1} failed: {e}")
```

Figura 1.5 *Metoda connect_to_server_with_retry*

Aici, *socket.timeout* este una dintre excepțiile capturate pentru a detecta problemele de conexiune, cum ar fi un server indisponibil.

Suport Hardware și Software pentru Rulare

2.1. Hardware

Hardware-ul utilizat pentru dezvoltarea și rularea proiectului este un sistem entry-level, suficient pentru a rula aplicația de chat client-server, care nu are cerințe ridicate de procesare sau rețea, deoarece funcționează local.

Sistem utilizat:

Nume dispozitiv: DESKTOP-988G00H

Procesor: AMD Athlon Gold 3150U with Radeon Graphics, 2.40 GHz. Acest procesor dual-core este adecvat pentru rularea aplicațiilor Python simple, cum ar fi serverul și clienții din acest proiect. Frecvența de 2.40 GHz permite gestionarea simultană a serverului și a mai multor clienți fără întârzieri semnificative.

RAM instalat: 4,00 GB (3,38 GB utilizabile). Deși memoria RAM este limitată, este suficientă pentru acest proiect, deoarece aplicația nu consumă resurse intensive. Serverul (server.py) stochează mesajele într-o listă în memorie, iar clienții (client.py) rulează o interfață grafică ușoară bazată pe Tkinter. Totuși, rularea simultană a mai multor clienți (ex. 5 sau mai mulți) ar putea încetini sistemul din cauza memoriei limitate.

Sistem de operare: Windows 10. Windows 10 este compatibil cu toate bibliotecile utilizate (RPyC, Tkinter, etc.) și oferă suport nativ pentru Python și Visual Studio Code.

Rețea:

- **Conexiune locală (localhost)**: Proiectul rulează pe aceeași mașină, deci nu sunt necesare cerințe speciale de rețea. Serverul și clienții comunică prin localhost (adresa IP 127.0.0.1) pe portul 18813.
- **Detalii suplimentare**: Deși aplicația funcționează local, codul permite modificarea adresei localhost din client.py cu o adresă IP reală pentru a rula serverul și clienții pe mașini diferite. În acest caz, ar fi necesare o conexiune stabilă de rețea (ex. Wi-Fi sau Ethernet) și configurarea firewall-ului pentru a permite traficul pe portul 18813.

Proiectul ocupă un spațiu minim pe disc, constând doar din fișierele server.py și client.py (aproximativ 10 KB în total). Instalarea Python [8] și a bibliotecii RPyC [9] adaugă aproximativ 100 MB de spațiu ocupat pe disc.

2.2. Software

Configurația software include sistemul de operare, mediul de dezvoltare, versiunea Python și bibliotecile necesare pentru rularea aplicației.

Sistem de operare: Versiunea utilizată este Windows 10 Home, 64-bit, build 19045. Aceasta oferă suport complet pentru Python 3.12.4 și Visual Studio Code, precum și pentru bibliotecile utilizate în

proiect. Windows 10 include, de asemenea, suport nativ pentru firewall, care a fost configurat pentru a permite traficul pe portul 18813 (deși, în cazul rulării locale, acest pas nu a fost necesar).

Mediu de dezvoltare:

Visual Studio Code (VS Code)

Utilizat ca editor de cod principal, VS Code este un editor ușor, cu suport excelent pentru Python prin extensii, oferind funcționalități precum depanare, terminal integrat și evidențierea sintaxei.

Python

Python este un limbaj de programare versatil, utilizat în dezvoltarea web, automatizare, analiză de date și învățare automată. Fiind ușor de învățat și folosit, este preferat atât de începători, cât și de profesioniști.[3] Este folosit pentru:

- **Analiza datelor & Machine Learning** – biblioteci precum TensorFlow și Pandas permit procesarea și vizualizarea datelor.
- **Dezvoltare Web** – framework-uri precum Django și Flask facilitează crearea de aplicații web.
- **Automatizare & Scripting** – Python ajută la reducerea sarcinilor repetitive.
- **Testare Software** – folosit pentru testare automată și prototipare.

Datorită suportului extins și comunității active, Python rămâne un limbaj ideal pentru o gamă largă de aplicații.

Biblioteci:

- **RPyC (Remote Python Call)**: RPyC este esențial pentru comunicarea client-server, permițând apeluri de procedură la distanță între server.py și client.py.
- **Tkinter**: Tkinter este utilizat în client.py pentru a crea interfața grafică, inclusiv zona de chat, câmpul de introducere și butonul de trimitere.
- **Threading, Time, Socket**:
- *threading*: Folosit în server.py pentru gestionarea clienților în fire separate și în client.py pentru polling-ul mesajelor.
- *time*: Utilizat în client.py pentru a introduce întârzieri în polling și reconectare.
- *socke*t: Utilizat în client.py pentru a gestiona erorile de conexiune, cum ar fi socket.timeout.

2.3. Configurare Suport Personal

Proiectul a fost creat într-un director numit *PD*, cu calea completă: *E:\FACULTATE\ANUL 4\SEM 2\PD*

Structura proiectului prezintă două fișiere create folosind VS Code:

1. server.py: Conține logica serverului, responsabil pentru gestionarea clienților și mesajelor.
2. client.py: Conține logica clientului, inclusiv interfața grafică bazată pe Tkinter.

După crearea fișierelor și descărcarea bibliotecilor necesare am trecut la testarea inițială a funcționalității aplicației folosind terminalul Command Prompt.

Prezentarea Codului și Rezultatelor Rulării

3.1. Structura Proiectului și Mediul de Dezvoltare

Această secțiune prezintă mediul de lucru în care a fost dezvoltat și rulat proiectul, inclusiv structura directorului și fișierele implicate.

Proiectul este organizat într-un director numit PD, situat pe discul E cu calea *E:\FACULTATE\ANUL 4\SEM 2\PD*. Mediul de dezvoltare utilizat este Visual Studio Code (VS Code), care afișează structura proiectului în panoul Explorer. În VS Code, sunt deschise fișierele server.py și client.py, iar calea directorului este vizibilă în bara de titlu a aplicației.

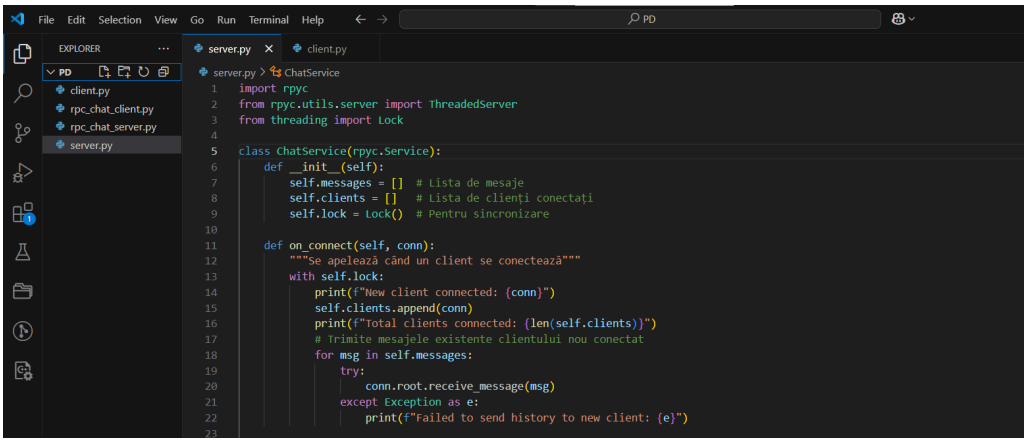

Figura 3.1 *Structura proiectului*

3.2. Lansarea Serverului

Serverul este lansat folosind comanda: python server.py. Comanda este rulată în terminalul Command Prompt. Serverul afișează mesajul *„Starting RPC Chat Server on port 18813...”* și așteaptă conexiuni.

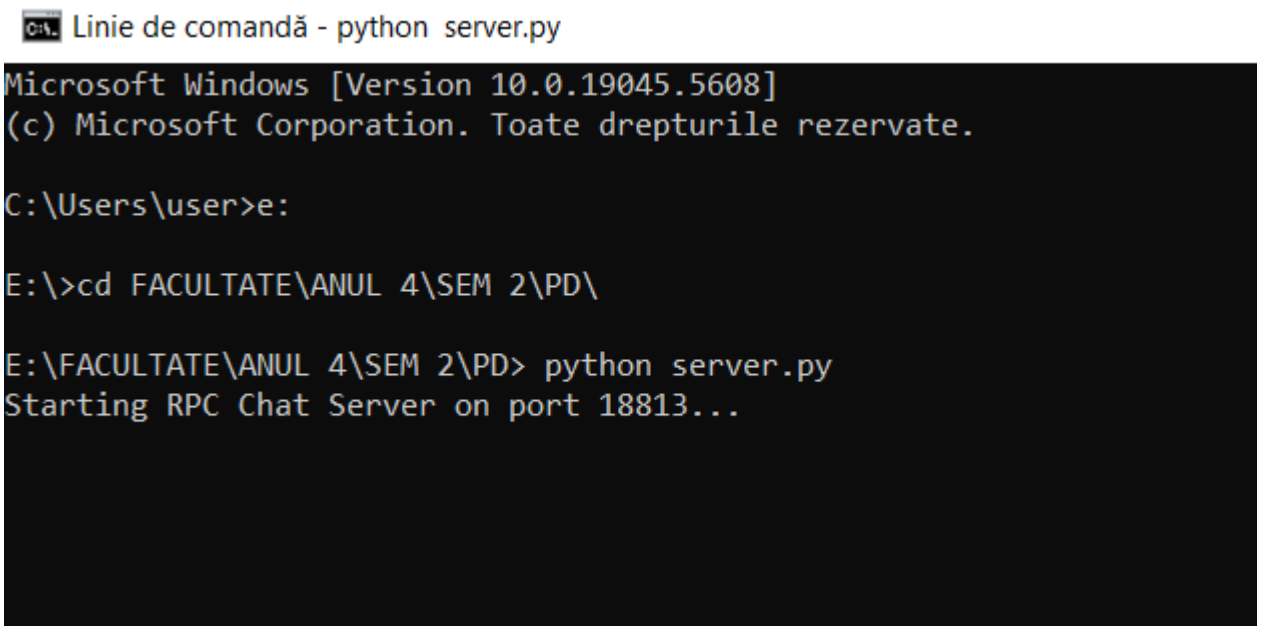

Figura 3.2 *Lansare Server*

3.3. Lansarea Clientului

Clientul este lansat într-un alt terminal, folosind comanda: `python client.py`.Utilizatorul introduce un nume (ex. „Lavi”), apoi se deschide fereastra grafică a clientului. Fereastra afișează mesajul inițial: „Successfully connected to server on attempt 1.” și „Connected as Lavi. Type your message below.”

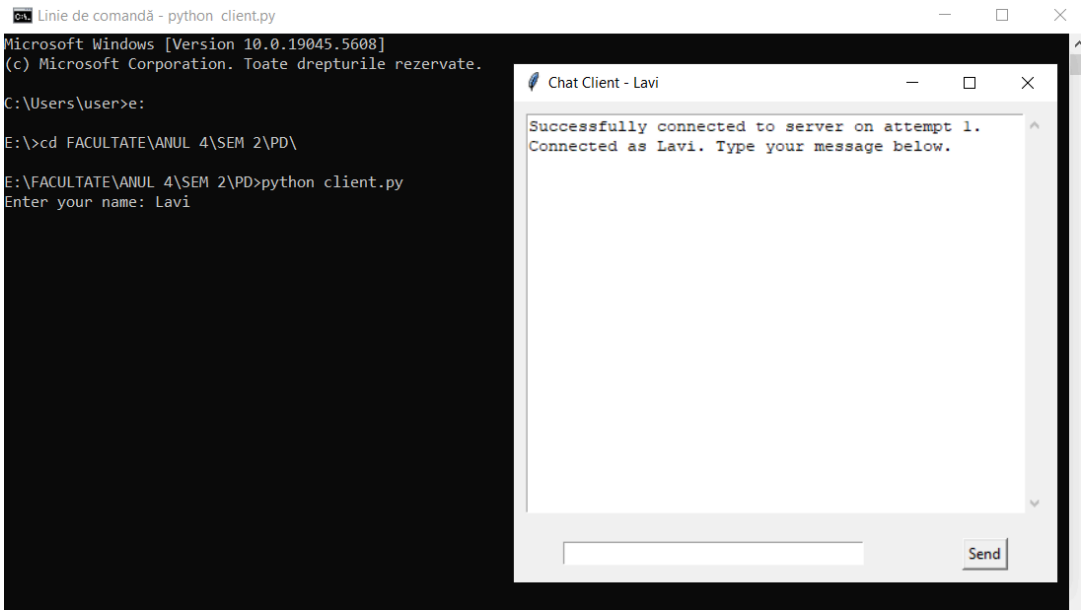

Figura 3.3 *Lansare client 1*

3.4. Testare Distribuită

Au fost rulați doi clienți, unul cu numele „Lavi” și altul cu numele „Alex”, ambii conectându-se la serverul local. Clienții trimit mesaje, iar acestea sunt afișate în ambele ferestre, demonstrând funcționalitatea de broadcast a serverului.

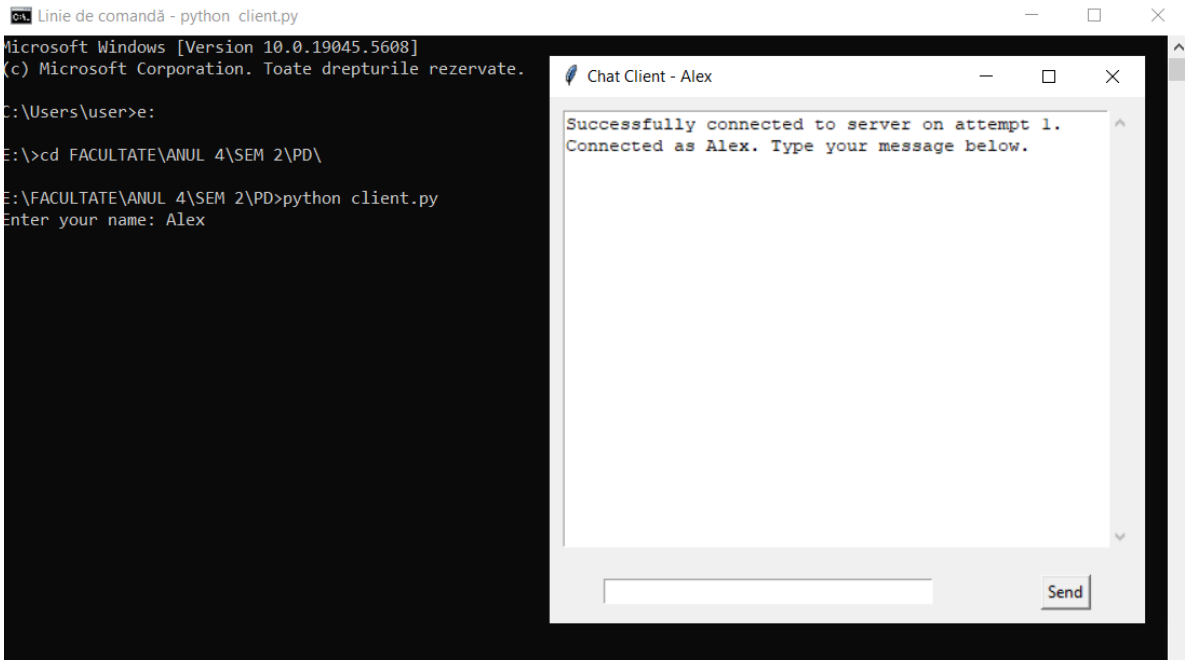

Figura 3.4 *Lansare al doilea client*

Figura 3.5 *Apariția clienților în terminalul serverului*

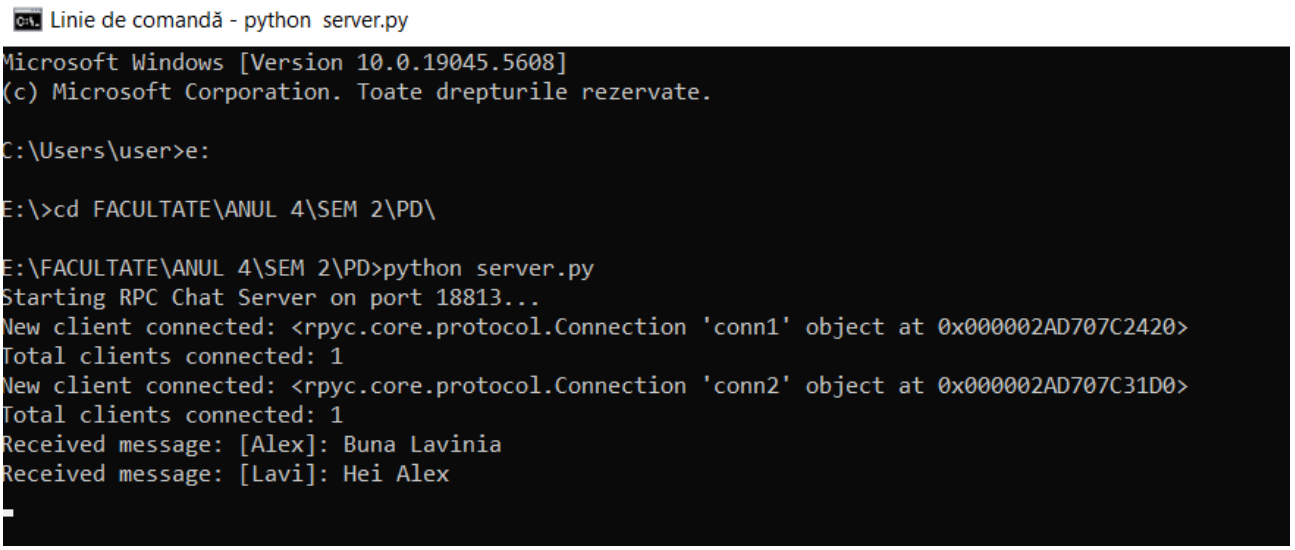

Figura 3.6 *Mesajele clienților în terminalul serverului*

Concluzii

Dezvoltarea proiectului RPC Chat a reprezentat o experiență practică valoroasă, care mi-a permis să aprofundez utilizarea bibliotecii RPyC pentru comunicarea client-server și să integrez o interfață grafică funcțională cu Tkinter. Un aspect semnificativ al muncii mele a fost gestionarea erorilor de conexiune, cum ar fi „result expired”, care m-a determinat să ajustez timeout-urile și să implementez un mecanism de reconectare. Acest proces a necesitat o înțelegere detaliată a modului în care RPyC gestionează apelurile la distanță, precum și o optimizare atentă a frecvenței polling-ului pentru a evita supraîncărcarea serverului, reducând intervalul de la 1 secundă la 2 secunde. Testarea cu mai mulți clienți pe aceeași mașină a confirmat funcționalitatea aplicației, dar a evidențiat și limitările hardware-ului meu, în special memoria RAM de 4 GB, care a dus la o ușoară încetinire atunci când am rulat simultan trei clienți.

Un alt punct important al experienței mele a fost configurarea mediului de lucru în Visual Studio Code și integrarea tuturor componentelor software necesare. Instalarea Python 3.12.4 și a bibliotecii RPyC a fost simplă, dar configurarea corectă a interpretorului și a extensiei Python în VS Code a necesitat atenție pentru a asigura rularea fără probleme. De asemenea, am învățat să folosesc firele de execuție (`threading`) pentru a menține interfața grafică responsivă în timp ce polling-ul mesajelor rulează în fundal, ceea ce a fost o provocare inițială, dar a contribuit la o experiență de utilizare fluidă. În ansamblu, proiectul mi-a consolidat abilitățile de programare în Python, m-a învățat să depanez probleme practice și mi-a oferit o înțelegere mai clară a comunicării distribuite.

Reflectând asupra muncii mele, consider că am atins obiectivele propuse, dar aș putea îmbunătăți aplicația în viitor prin adăugarea unor funcționalități precum notificări vizuale pentru mesaje noi sau gestionarea mai eficientă a resurselor pe sisteme cu memorie limitată. De asemenea, testarea pe mașini diferite, în loc de localhost, ar fi oferit o perspectivă mai realistă asupra performanței în rețea. Proiectul a fost o oportunitate de a aplica cunoștințele teoretice într-un context real, iar provocările întâmpinate m-au ajutat să devin mai atent la detalii și mai bine pregătit pentru dezvoltarea aplicațiilor distribuite.

Bibliografie

[1] RPyC - Transparent, Symmetric Distributed Computing
https://rpyc.readthedocs.io/en/latest/

[2] Microsoft, “Visual Studio: IDE and Code Editor for Software Developers”
https://visualstudio.microsoft.com/

[3] What Is Python Used For?
https://www.coursera.org/articles/what-is-python-used-for-a-beginners-guide-to-using-python

[4] GeeksforGeeks, „Multithreading in Python | Set 1,” Ianuarie 2025
https://www.geeksforgeeks.org/multithreading-python-set-1/

[5] Real Python, „Socket Programming in Python (Guide),”
https://realpython.com/python-sockets/

[6] Elena-Cristina Stoica, “Apelul procedurilor la distanta RPC - Remote Procedure Call” , curs de “Programare Distribuită”

[7] “tkinter — Python interface to Tcl/Tk”
https://docs.python.org/3/library/tkinter.html

[8] Python Software Foundation, „Python Downloads,”
https://www.python.org/downloads/

[9] PyPI, „rpyc 5.3.1,”
https://pypi.org/project/rpyc/

ANEXE

Anexa 1 - Codul pentru server.py

```
import rpyc
from rpyc.utils.server import ThreadedServer
from threading import Lock

class ChatService(rpyc.Service):
    def __init__(self):
        self.messages = []  # Lista de mesaje
        self.clients = []   # Lista de clienți conectați
        self.lock = Lock()  # Pentru sincronizare

    def on_connect(self, conn):
        """Se apelează când un client se conectează"""
        with self.lock:
            print(f"New client connected: {conn}")
            self.clients.append(conn)
            print(f"Total clients connected: {len(self.clients)}")
            # Trimite mesajele existente clientului nou conectat
            for msg in self.messages:
                try:
                    conn.root.receive_message(msg)
                except Exception as e:
                    print(f"Failed to send history to new client: {e}")

    def on_disconnect(self, conn):
        """Se apelează când un client se deconectează"""
        with self.lock:
            print(f"Client disconnected: {conn}")
            if conn in self.clients:
                self.clients.remove(conn)
            print(f"Total clients connected: {len(self.clients)}")

    def exposed_send_message(self, sender, message):
        """Metodă expusă pentru trimiterea mesajelor"""
        with self.lock:
            formatted_message = f"[{sender}]: {message}"
            self.messages.append(formatted_message)
            print(f"Received message: {formatted_message}")
            self.broadcast_message(formatted_message)
            return f"Message from {sender} received"

    def exposed_get_messages(self, last_index):
        """Metodă expusă pentru obținerea mesajelor noi"""
        with self.lock:
            if last_index < 0 or last_index >= len(self.messages):
                last_index = -1
```

```
            new_messages = self.messages[last_index + 1:]
            return new_messages, len(self.messages) - 1

    def broadcast_message(self, message):
        """Trimite mesajul către toți clienții conectați"""
        with self.lock:
                  print(f"Broadcasting message to {len(self.clients)}
clients: {message}")
            disconnected_clients = []
            for client in self.clients:
                try:
                    client.root.receive_message(message)
                except Exception as e:
                    print(f"Failed to send to a client: {e}")
                    disconnected_clients.append(client)

            # Elimină clienții deconectați
            for client in disconnected_clients:
                if client in self.clients:
                    self.clients.remove(client)
                            print(f"Total clients after broadcast:
{len(self.clients)}")

if __name__ == "__main__":
    server = ThreadedServer(ChatService, port=18813, reuse_addr=True)
    print("Starting RPC Chat Server on port 18813...")
    server.start()
```

Anexa 2 - Codul pentru client.py

```
import rpyc
import threading
import time
import tkinter as tk
from tkinter import scrolledtext
import socket

class ClientService(rpyc.Service):
    def exposed_receive_message(self, message):
        """Metodă expusă pentru primirea mesajelor de la server"""
        if app:
            app.display_message(f"Received: {message}")

class ChatApp:
    def __init__(self, root, client_name):
        self.root = root
        self.client_name = client_name
        self.root.title(f"Chat Client - {client_name}")
        self.conn = None  # Initialize conn as None
        self.is_connected = False  # Track connection status
```

```
        # Configurare interfață
            self.chat_area = scrolledtext.ScrolledText(root, width=50,
height=20, wrap=tk.WORD)
            self.chat_area.grid(row=0, column=0, columnspan=2, padx=10,
pady=10)

        self.input_field = tk.Entry(root, width=40)
        self.input_field.grid(row=1, column=0, padx=10, pady=10)
                    self.input_field.bind("<Return>", lambda event:
self.send_message())  # Trimitere cu Enter

                    self.send_button = tk.Button(root, text="Send",
command=self.send_message)
        self.send_button.grid(row=1, column=1, padx=10, pady=10)

        # Încercare de conectare la server cu retry
        self.connect_to_server_with_retry()
        if not self.conn:
            self.display_message("Failed to connect to server. Please
ensure the server is running and try again.")
            return

          self.display_message(f"Connected as {client_name}. Type your
message below.")

        # Variabile pentru polling
        self.last_index = -1
        self.stop_polling = False

        # Pornește polling-ul mesajelor
                                            self.polling_thread =
threading.Thread(target=self.poll_messages)
        self.polling_thread.daemon = True
        self.polling_thread.start()

        # Gestionare închidere fereastră
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)

            def connect_to_server_with_retry(self, max_retries=3,
retry_delay=5):
        """Încercă să se conecteze la server cu retry"""
        for attempt in range(max_retries):
            try:
                # Configurare timeout-uri mai lungi
                self.conn = rpyc.connect(
                    "localhost",
                    18813,
                    service=ClientService,
                    config={
```

```
                                "sync_request_timeout": 30,  # 30 secunde
pentru cereri sincrone
                            "allow_public_attrs": True,
                            "allow_pickle": True
                        }
                    )
                    self.is_connected = True
                     self.display_message(f"Successfully connected to server
on attempt {attempt + 1}.")
                    return
                                    except  (socket.timeout,  TimeoutError,
ConnectionRefusedError) as e:
                    self.display_message(f"Connection attempt {attempt + 1}
failed: {e}")
                    if attempt < max_retries - 1:
                           self.display_message(f"Retrying in {retry_delay}
seconds...")
                        time.sleep(retry_delay)
            self.display_message("Could not connect to server after maximum
retries.")
            self.is_connected = False

    def display_message(self, message):
        """Afișează un mesaj în zona de chat"""
        self.chat_area.insert(tk.END, message + "\n")
           self.chat_area.see(tk.END)  # Derulează automat la ultimul
mesaj

    def send_message(self):
        """Trimite mesajul introdus către server"""
        if not self.is_connected or not self.conn:
             self.display_message("Cannot send message: Not connected to
server.")
            return
        message = self.input_field.get().strip()
        if message:
            try:
                                                        response    =
self.conn.root.send_message(self.client_name, message)
                self.display_message(f"Server: {response}")
                self.input_field.delete(0, tk.END)
            except Exception as e:
                self.display_message(f"Error sending message: {e}")
                     if "result expired" in str(e) or "connection" in
str(e).lower():
                    self.is_connected = False
                      self.display_message("Connection lost. Attempting
to reconnect...")
                    self.reconnect()
```

```
    def poll_messages(self):
        """Verifică periodic mesajele noi de la server"""
        while not self.stop_polling:
            if not self.is_connected or not self.conn:
                        time.sleep(5)   # Așteaptă înainte de a încerca
reconectarea
                self.reconnect()
                continue
            try:
                                                new_messages,   new_index   =
self.conn.root.get_messages(self.last_index)
                for msg in new_messages:
                    self.display_message(f"Received: {msg}")
                self.last_index = new_index
            except Exception as e:
                self.display_message(f"Error polling messages: {e}")
                     if "result expired" in str(e) or "connection" in
str(e).lower():
                    self.is_connected = False
                      self.display_message("Connection lost. Attempting
to reconnect...")
                    self.reconnect()
                 time.sleep(2)   # Reducem frecvența polling-ului la 2
secunde

    def reconnect(self):
        """Încearcă să se reconecteze la server"""
        if self.conn:
            try:
                self.conn.close()
            except:
                pass
        self.conn = None
        self.connect_to_server_with_retry()
        if self.is_connected:
              self.last_index = -1  # Resetează indexul pentru a primi
toate mesajele noi

    def on_closing(self):
        """Închide conexiunea și fereastra"""
        self.stop_polling = True
        if self.polling_thread.is_alive():
            self.polling_thread.join()
        if self.conn:
            try:
                self.conn.close()
            except:
                pass
        self.root.destroy()
```

```
def run_client(client_name):
    global app
    root = tk.Tk()
    app = ChatApp(root, client_name)
    root.mainloop()

if __name__ == "__main__":
    client_name = input("Enter your name: ")
    run_client(client_name)
```