

# **רשתות מחשבים ואינטרנט 1 – פרויקט גמר**

## ***Online Client – Server Group Chats***

**מגיש: יובל דיסטניק**

**תעודת זהות: 318904760**

**מערכת הפעלה: Windows 11 (64 – bit)**

**שפת תכנות וגרסה: Python 3.9**

## 1. הקדמה – Introduction

בפרויקט זה נממש חדרי צ'אט קבוצתיים מקוונים. מטרת הפרויקט הינה לשפר את יכולות התכנות ולהעמיק את ההיכרות עם שכבות התעבורה והאפליקציה.

בפרויקט זה תמומש ארכיטקטורת לקוח-שרת, כפי שנלמדה במהלך הסמסטר. ארכיטקטורה זו מורכבת משתי ישויות – לקוח אשר צורך שירותים, ושרת אשר מספק שירותים. במקרה שלנו, הלקוח הוא משתמש המעוניין בקיום שיח קבוצתי בצ'אט, והשרת הוא המחשב האמון על ניהול חדרי הצ'אט. שירותיו של השרת כוללים בין היתר מתן גישה אל חדרי הצ'אט, שליחת הודעות אל המשתמשים וכולי.

## 2. דרישות - Requirements

### 2.1. צד שרת - Server Side

בחלק זה נתאר את קוד צד השרת. מומש שרת התומך בקיום חדרי צ'אט מרובי-משתתפים, עם מערכת זיהוי משתמש. כלומר, לקוח המעוניין להתחבר אל השרת ולהיכנס לאחד מחדרי הצ'אט חייב לספק שם משתמש וסיסמה תקינים.

מומשו גם שני סוגי משתמש – משתמש רגיל ומשתמש מסוג מנהל. המשתמש הרגיל יכול להירשם אל השרת כאשר הוא מתחבר בפעם הראשונה ולבחור שם משתמש וסיסמה. כאשר המשתמש כבר רשום, הוא יכול להתחבר אל השרת לאחר שהוא מספק את שם המשתמש והסיסמה שלו. לאחר התחברות, המשתמש יכול לבחור להתחבר לכל אחד משלושת חדרי הצ'אט הקיימים. משתמש רגיל המחובר לחדר הצ'אט יכול לשלוח הודעות, להעלות קבצים אל השרת, לראות את רשימת המשתמשים המחוברים אל אותו חדר צ'אט, ולראות את כלל ההודעות שנשלחו בחדר הצ'אט עד כה. משתמש מסוג מנהל, בנוסף להרשאות של משתמש רגיל, יכול גם להוציא משתמשים מהצ'אט, לחסום משתמשים מהצ'אט (ואז המשתמש לא יכול להתחבר שוב לצ'אט) ולשחרר חסימה של משתמשים. הגדרתי את האפליקציה כך שבעת הפעלת השרת, משתמש מסוג מנהל נוצר באופן אוטומטי וכל משתמש אחר שנרשם הוא משתמש רגיל.

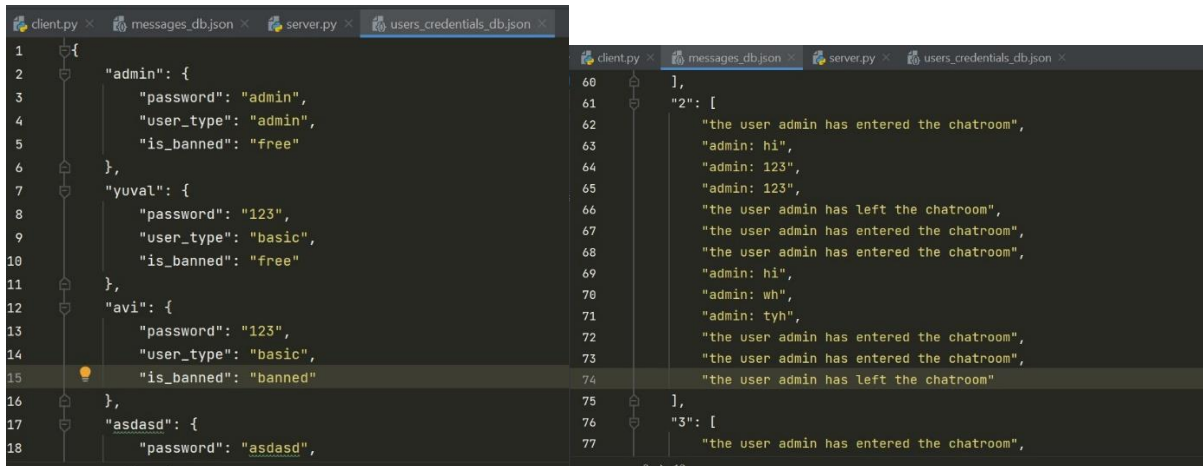
אחסון הנתונים המתמיד מומש ע"י מסדי נתונים מסוג קבצי JSON. קובץ אחד אחראי לאחסון את פרטי המשתמשים – שם משתמש, סיסמה והרשאות. קובץ נוסף, אחראי לאחסון את כל ההודעות שנשלחו בכל אחד מחדרי הצ'אט, מחולקות לפי החדרים בהן נשלחו ולפי סדר שליחתן. בכך, אנו מבטיחים את אמינות המידע, אפילו אם השרת קורס או מאותחל. כאשר תוכנית השרת מתחילה לרוץ, השרת בודק אם קיימים קבצי ה-JSON המהווים את מסד הנתונים. אם הם לא קיימים (זו פעם ראשונה שהשרת רץ), השרת יוצר אותם. אם הם קיימים (השרת חוזר לרוץ לאחר קריסה/אתחול) השרת שולף את תוכנם אל זיכרון התוכנית על מנת שיוכל להשתמש במידע.

כפי שניתן לראות, מצב מסדי הנתונים בריצתו הראשונה של השרת :

```
client.py x messages_db.json x server.py x
1 {"1": [], "2": [], "3": []}
```

```
client.py x messages_db.json x users_credentials_db.json x server.py x
1 {
2   "admin": {
3     "is_banned": "free",
4     "password": "admin",
5     "user_type": "admin"
6   }
7 }
```

ולאחר זמן מה של שימוש, משתמשים שנרשמו והודעות שנשלחו:



```

1 {
2   "admin": {
3     "password": "admin",
4     "user_type": "admin",
5     "is_banned": "free"
6   },
7   "yuval": {
8     "password": "123",
9     "user_type": "basic",
10    "is_banned": "free"
11  },
12  "avi": {
13    "password": "123",
14    "user_type": "basic",
15    "is_banned": "banned"
16  },
17  "asdasd": {
18    "password": "asdasd",
19  }
20 }

```

```

60 ],
61 "2": [
62   "the user admin has entered the chatroom",
63   "admin: hi",
64   "admin: 123",
65   "admin: 123",
66   "the user admin has left the chatroom",
67   "the user admin has entered the chatroom",
68   "the user admin has entered the chatroom",
69   "admin: hi",
70   "admin: wh",
71   "admin: tyh",
72   "the user admin has entered the chatroom",
73   "the user admin has entered the chatroom",
74   "the user admin has left the chatroom"
75 ],
76 "3": [
77   "the user admin has entered the chatroom",

```

על מנת למנוע הצפה (שליחת בקשות מוגברת לשרת כגון DDOS), מומש מנגנון *rate – limiting* המאפשר למשתמש לשלוח עד ל-20 הודעות בשנייה (ערך זה נקבע שרירותית וניתן לשינוי לפי הצורך). במידה ומשתמש חוצה את הרף הנ"ל, קופצת הודעת שגיאה רלוונטית המונעת ממנו לשלוח הודעות נוספות עד שהוא לא סוגר את חלון השגיאה.



כמבוקש, השרת תומך ב-3 חדרי צ'אט שונים (ערך שרירותי וניתן לשינוי) היכולים לרוץ ללא תלות אחד בשני. המערכת הוגדרה כך שמשתמש יכול להיכנס אך ורק לאחד מ-3 החדרים הקיימים, ורק מנהל המערכת יכול ליצור חדרי נוספים על ידי ביצוע שינויים קלים בקוד התוכנית. נציין כי הוצאת והרחקת המשתמשים מומשה ע"י יצירת חדר רביעי (חדר יס') המחזיק את המשתמשים הנ"ל ומונע מהם לשלוח ולקבל הודעות עד ליציאתם מהאפליקציה.

\* בנוגע למעבר משתמשים בין חדרים, נכון למועד הגשת הפרויקט סטטוס הפיצ'ר הזה הוא תמיכה של 100% מצד השרת, אך עדיין קיים באג המונע מהפיצ'ר לעבוד בצד הלקוח. לכן, הפיצ'ר הוסר על מנת לאפשר תפקוד תקין של האפליקציה והכפתור הרלוונטי הוסר מחלון הצ'אט.

**תיעוד והסברים – קוד צד השרת:****Imports:**

The *socket* library for communication handling, the *threading* library for concurrency, the *os* library for system-related operations, the *json* library for data storage, and the *cryptography* modules for encryption.

**Constants:**

Defined several constants for modularity and readability, such as paths to database files, the server IP address, port number, message format, message size, allowed login tries, encryption key and IV, and various message strings for communication purposes.

**Functions and Classes:****Encrypt()**

Inputs: plaintext: str - The plaintext message to be encrypted.

Outputs: ciphertext: bytes - The encrypted ciphertext.

Operation: Encrypts the provided plaintext message using AES-CBC encryption.

**Decrypt()**

Inputs: ciphertext: bytes - The encrypted ciphertext to be decrypted.

Outputs: decrypted\_data: str - The decrypted plaintext message.

Operation: Decrypts the provided ciphertext using AES-CBC decryption.

**Server.init()**

Inputs: None

Outputs: None

Operation: Initializes the server by checking for existing databases, loading user credentials and messages, setting up the server socket, and defining necessary constants and attributes.

**Server.check\_for\_existing\_db()**

Inputs: None

Outputs: None

Operation: Checks if the required database files exist and initializes them if not.

**Server.initialize\_system\_administrator()**

Inputs: None

Outputs: None

Operation: Initializes the system administrator if no users exist in the database.

### **Server.load\_users\_credentials()**

Inputs: None

Outputs: A nested dictionary containing user credentials (users\_credentials).

Operation: Loads user credentials from the database file to the program memory.

### **Server.load\_messages()**

Inputs: None

Outputs: messages: A dictionary containing messages, divides by rooms (messages).

Operation: Loads messages from the database file to the program memory.

### **Server.save\_user\_credentials()**

Inputs: None

Outputs: None

Operation: Saves updated user credentials from the program memory to the database file.

### **Server.save\_messages()**

Inputs: None

Outputs: None

Operation: Saves updated messages from the program memory to the database file.

### **Server.start()**

Inputs: None

Outputs: None

Operation: Initiates an infinite loop to handle new client connections.

### **Server.handle\_client()**

Inputs: client\_socket: socket - The socket object representing the client connection,

client\_address: tuple – The IP address and port number of the client.

Outputs: None

Operation: Handles communication with a client, including authentication, message reception, and broadcasting messages.

### **Server.receive\_message()**

Inputs: client\_socket: socket - The socket object representing the client connection,

client\_username: str - The username of the client,

client\_chatroom: str - The chatroom of the client.

Outputs: None

Operation: Receives messages from the client and processes them accordingly.

### **Server.receive\_file()**

Inputs: client\_socket: socket - The socket object representing the client connection,

file\_size: str - The size of the file to be received,

file\_name: str - The name of the file to be received.

Outputs: data: bytes - The contents of the received file.

Operation: Receives a file from the client and saves it locally.

### **Server.broadcast\_message()**

Inputs: sender\_username: str - The username of the message sender,

message: str - The message to be broadcasted,

administrative\_message: int - Flag indicating if the message is administrative.

Outputs: None

Operation: Broadcasts a message to all connected clients that are at the same room with the sender, optionally marking it as administrative.

### **Server.send\_previous\_messages()**

Inputs: client\_username: str - The username of the client,

client\_chatroom: str - The chatroom of the client.

Outputs: None

Operation: Sends previous messages sent in a room to a client upon connection.

### **Server.check\_credentials()**

Inputs: username: str - The username to be checked,

password: str - The password to be checked.

Outputs: status: int - Authentication status (0 for success, -1 for failure).

Operation: Checks user credentials against the stored credentials.

### **Server.sign\_in\_user()**

Inputs: client\_socket: socket - The socket object representing the client connection.

Outputs: status: int - Authentication status (0 for success, -1 for failure),

client\_username: str - The username of the client,

client\_password: str - The password of the client.

Operation: Authenticates a user during sign-in process.

### **Server.check\_user\_ban()**

Inputs: client\_socket: socket - The socket object representing the client connection,

client\_username: str - The username of the client.

Outputs: client\_ban\_status: str - The ban status of the client,

client\_user\_type: str - The user type of the client.

Operation: Checks if a user is banned and retrieves their user type.

### **Server.sign\_up\_user()**

Inputs: client\_socket: socket - The socket object representing the client connection.

Outputs: client\_username: str - The username of the signed-up client,

client\_password: str - The password of the signed-up client.

Operation: Registers a new user during sign-up process.

### **Server.verify\_chatroom()**

Inputs: client\_socket: socket - The socket object representing the client connection.

Outputs: valid\_chatroom: int - Flag indicating chatroom validity (0 for success, -1 for failure),

client\_chatroom: str - The chatroom chosen by the client.

Operation: Verifies the chatroom chosen by the client.



### **Server.kick\_out\_user()**

Inputs: client\_username: str - The username of the client performing the kick-out action,  
user\_to\_kick: str - The username of the user to be kicked out.

Outputs: None

Operation: Kicks out a user from the chatroom.

### **Server.ban\_user()**

Inputs: client\_username: str - The username of the client performing the ban action,  
user\_to\_ban: str - The username of the user to be banned.

Outputs: None

Operation: Bans a user from the chatroom.

### **Server.unban\_user()**

Inputs: client\_username: str - The username of the client performing the unban action,  
user\_to\_unban: str - The username of the user to be unbanned.

Outputs: None

Operation: Unbans a user previously banned from the chatroom.

### **Main()**

Inputs: None

Outputs: None

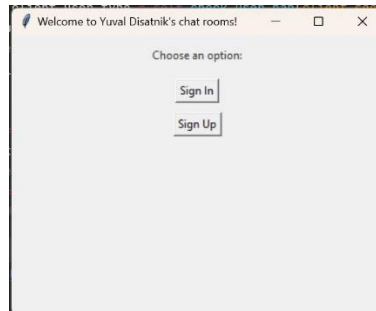
Operation: Initializes and starts the server.



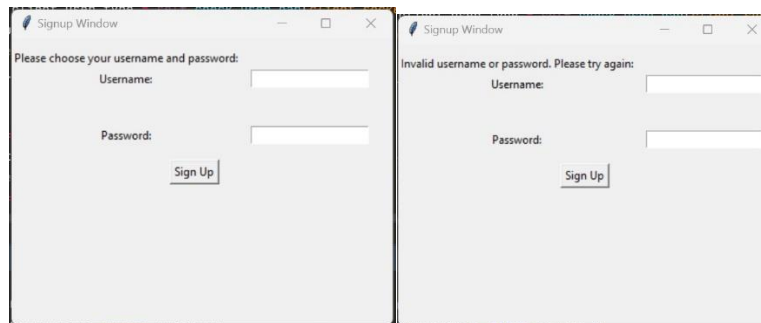
## 2.2. צד לקוח - Client Side

בחלק זה נתאר את קוד צד הלקוח. מומש לקוח התומך בזיהוי משתמש בטוח הכולל הזנת שם משתמש וסיסמה. כלומר, לקוח המעוניין להתחבר אל השרת ולהיכנס לאחד מחדרי הציאט חייב לספק שם משתמש וסיסמה תקינים.

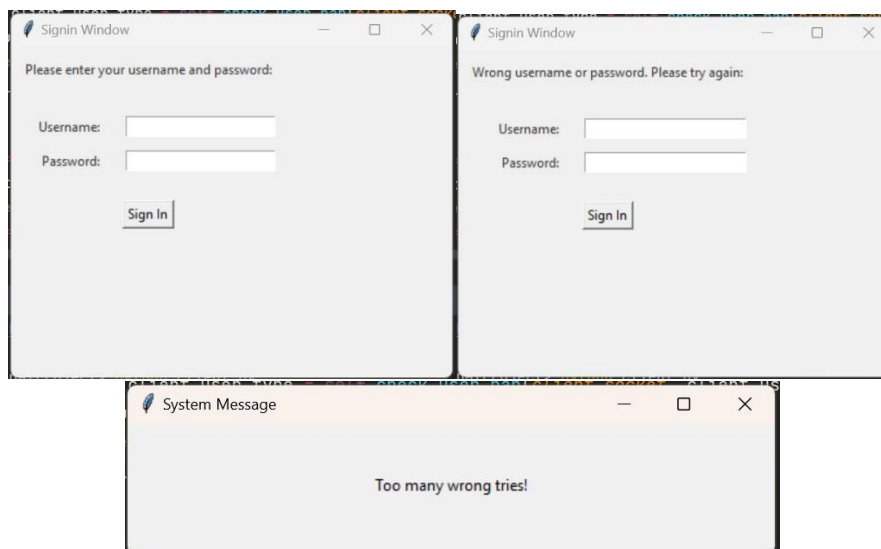
כאשר משתמש פותח את האפליקציה, נפתח חלון קבלת הפנים:



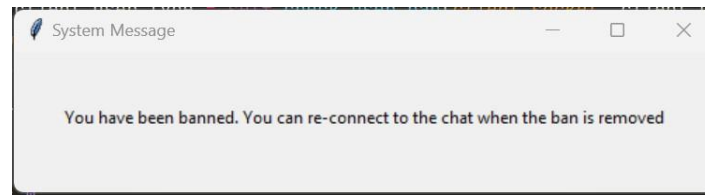
כאמור, משתמש יכול להירשם אל השרת כאשר הוא מתחבר בפעם הראשונה ולבחור שם משתמש וסיסמה. אם הוא אינו מזין שם משתמש וסיסמה תקינים, או ששם המשתמש שבחר כבר תפוס, הוא מקבל הודעה מתאימה והזמנות לבחור פרטים שוב.



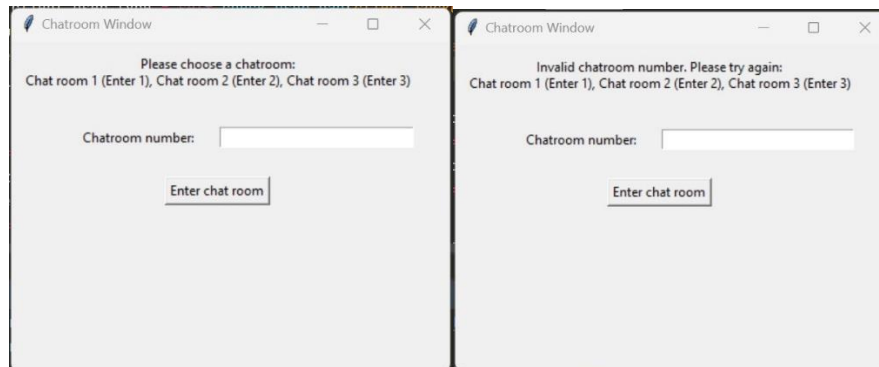
כאשר המשתמש כבר רשום, הוא יכול להתחבר אל השרת לאחר שהוא מספק את שם המשתמש והסיסמה שלו. אם הוא מזין פרטים שגויים הוא מקבל הודעה מתאימה. בחרתי להגדיר את המערכת כך שלאחר 3 ניסיונות כושלים המשתמש יצטרך לסגור את האפליקציה ולנסות להתחבר שוב.



בנוסף, אם משתמש חסום מנסה להתחבר אל הצ'אט, הוא מקבל את ההודעה הבאה :

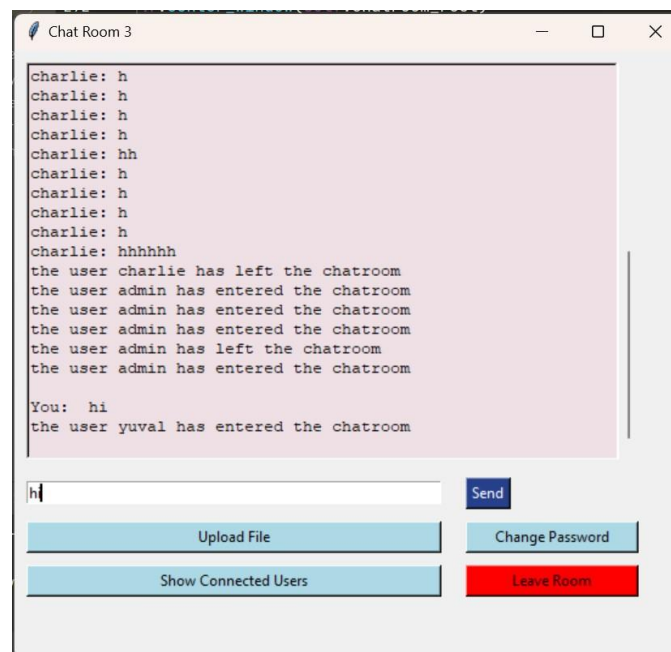


לאחר התחברות, המשתמש יכול לבחור להתחבר לכל אחד משלושת חדרי הצ'אט הקיימים. אם הוא בוחר חדר לא תקין, הוא מקבל על כך הודעה מתאימה.

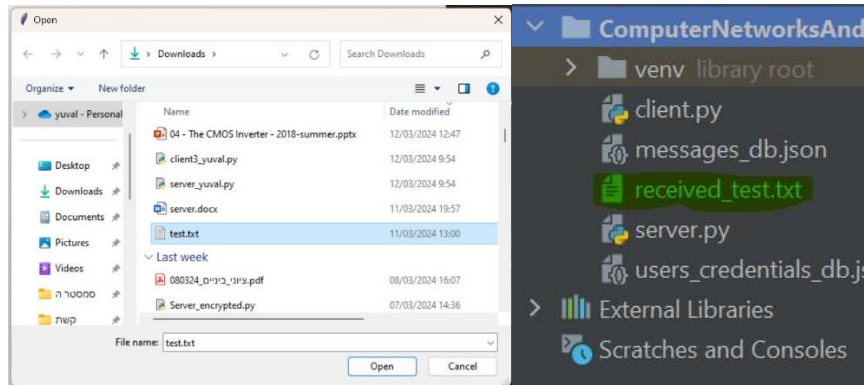


לאחר שלקוח מתחבר ובוחר חדר צ'אט, הוא נכנס אל חלון הצ'אט המרכזי שם הוא יכול לשלוח ולקבל הודעות, להעלות קבצים אל השרת, לשנות את הסיסמה שלו בבטחה, לצפות ברשימת כל המשתמשים המחוברים אל אותו חדר, ולצאת מהצ'אט.

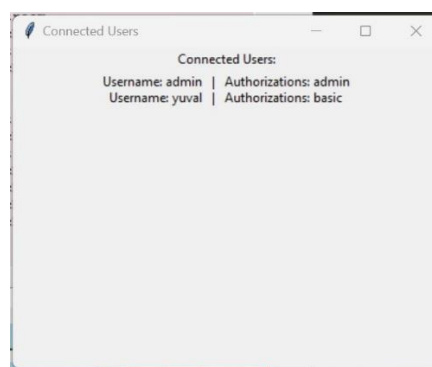
חלון הצ'אט המרכזי :



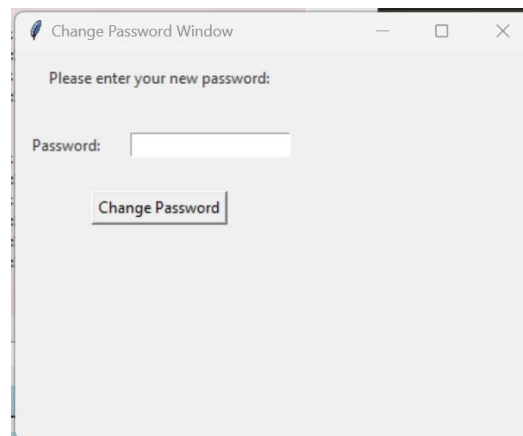
העלאת קבצים אל השרת, מצד הלקוח והאופן בו הם נשמרים (ע"י לחיצה על הכפתור *Upload File*):



צפייה ברשימת המשתמשים המחוברים לאותו החדר (ע"י לחיצה על הכפתור *Show Connected Users*):



שינוי סיסמה באופן בטוח (ע"י לחיצה על הכפתור *Change Password*):

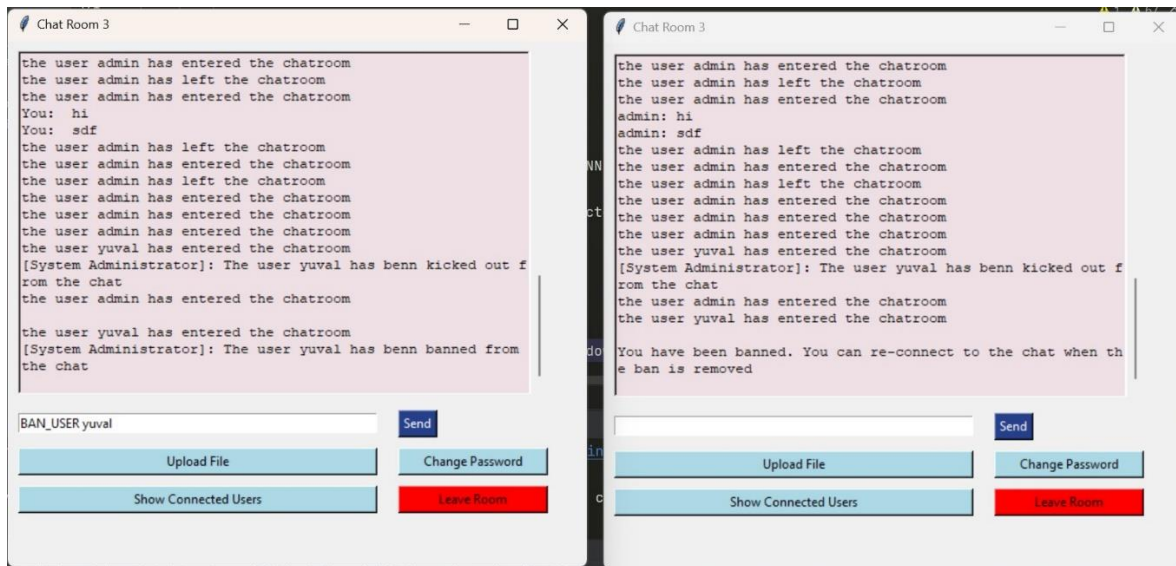


כאמור, ביכולתו של המנהל להעניף ולחסום/לשחרר משתמשים. אופן המימוש הוא מתן פורמט פקודה מובנה לביצוע כל אחת מהפעולות הללו וביצוען ע"י שליחת ההודעה כרגיל. כמובן, שישנן הגנות והתייחסויות למקרה בו משתמש ללא הרשאות מנסה לשלוח הודעות מסוג זה, או שההודעות נשלחו בפורמט לא תקין.

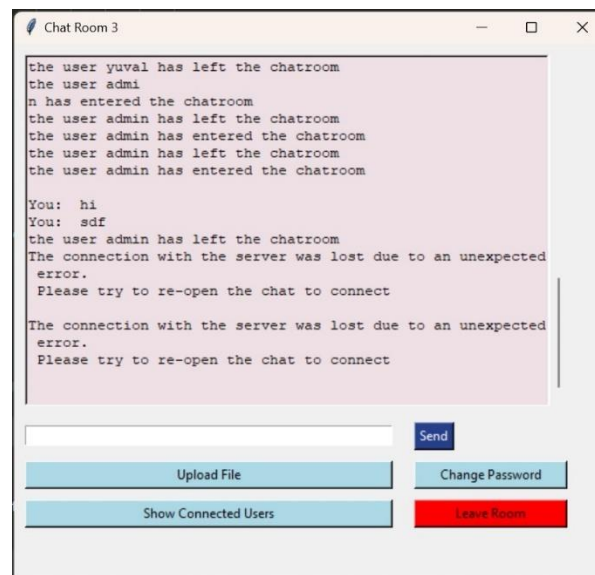
הפקודות שנקבעו הן:

`AN_USER "username", UNBAN_USER "username", KICK_USER "username"`

המשתמשים מקבלים הודעות רלוונטיות על הפעולה שהתבצעה:



בנוסף, מומש מנגנון התנהגות במקרה של נפילת השרת. אם הקשר אם השרת אבד, המשתמשים יקבלו הודעה מתאימה בעת ניסיון ביצוע כל אחת מהפעולות הנ"ל:



\* בנוגע לפיצור של העברת קבצים, נכון למועד הגשת הפרויקט הסטטוס הוא תמיכה בשליחת קבצים מהלקוח אל השרת, ואחסון הקובץ בצד השרת.

\* בנוגע לפיצור של הצפנת התקשורת בין השרת ללקוח, נכון למועד הגשת הפרויקט הסטטוס הוא באג בפענוח אחת מההודעות המתקבלות אצל הלקוח בתהליך ההתחברות (שגיאה: בייט לא חוקי לפענוח לפורמט 8 – utf). באופן כללי, המימוש מסתמך בסך הכול בהחלפת הפקודה `message.encode(FORMAT)` בפקודה `message.encrypt(message)` כאשר אנו מעוניינים לשלוח הודעה, והחלפת הפקודה `message.decode(FORMAT)` בפקודה `message.decrypt(message)` כאשר אנו מעוניינים לקבל הודעה. כאמור, בשל הבאג הפיצור של ההצפנה הוסר לחלוטין מן המימוש על מנת לאפשר תפקוד תקין של האפליקציה.

## תיעוד והסברים – קוד צד הלקוח:

### Imports:

The *socket* library for communication handling, the *threading* library for concurrency, the *tkinter* library for GUI, and several relevant modules for *tkinter*, the *time* library for time-related operations, the *os* library for system-related operations, the *webbrowser* library for opening URLs, and the *cryptography* modules for encryption.

### Constants:

Defined several constants for modularity and readability, such as the server IP address, port number, message format, message size, allowed login tries, encryption key and IV, and various message strings for communication purposes.

### Functions and Classes:

#### Encrypt()

Inputs: plaintext: str - The plaintext message to be encrypted.

Outputs: ciphertext: bytes - The encrypted ciphertext.

Operation: Encrypts the provided plaintext message using AES-CBC encryption.

#### Decrypt()

Inputs: ciphertext: bytes - The encrypted ciphertext to be decrypted.

Outputs: decrypted\_data: str - The decrypted plaintext message.

Operation: Decrypts the provided ciphertext using AES-CBC decryption.

#### Client.\_\_init\_\_()

Inputs: None

Outputs: None

Operation: Initializes the client, creates the GUI, establishes connection to the server, and handles sign-in or sign-up process.

#### Client.connect\_to\_server()

Inputs: None

Outputs: client\_socket: socket - The socket object representing the client connection.

Operation: Establishes a TCP connection to the server using the predefined IP address and port number.

#### Client.sign\_in\_user()

Inputs: gui: GUI - The GUI object for handling user interface.

Outputs: sign\_in\_status: int - Status of sign-in process (0 for success, -1 for failure).

Operation: Prompts the user to sign in, sends sign-in request to the server, and handles authentication.

### **Client.sign\_up\_user()**

Inputs: gui: GUI - The GUI object for handling user interface.

Outputs: sign\_up\_status: int - Status of sign-up process (0 for success, -1 for failure).

Operation: Prompts the user to sign up, sends sign-up request to the server, and handles registration.

### **Client.choose\_chatroom()**

Inputs: gui: GUI - The GUI object for handling user interface.

Outputs: None

Operation: Prompts the user to choose a chatroom and sends the chosen chatroom to the server for validation.

### **Client.start()**

Inputs: gui: GUI - The GUI object for handling user interface.

Outputs: None

Operation: Initiates the chat window GUI and begins message receiving and sending.

### **GUI.welcome\_window()**

Inputs: None

Outputs: None

Operation: Displays the welcome window GUI for choosing between sign-in and sign-up options.

### **GUI.sign\_in\_window()**

Inputs: is\_first\_attempt: int - Flag indicating if it's the first sign-in attempt.

Outputs: None

Operation: Displays the sign-in window GUI for entering username and password.

### **GUI.sign\_up\_window()**

Inputs: is\_first\_attempt: int - Flag indicating if it's the first sign-up attempt.

Outputs: None

Operation: Displays the sign-up window GUI for choosing a new username and password.

### **GUI.choose\_chatroom\_window()**

Inputs: is\_first\_attempt: int - Flag indicating if it's the first attempt to choose a chatroom.

Outputs: None

Operation: Displays the chatroom selection window GUI for choosing a chatroom to join.

### **GUI.chat\_window2()**

Inputs: client: Client - The client object for handling client-side Operations.

Outputs: None

Operation: Displays the chat window GUI for exchanging messages and performing various actions.

### **RateLimiter()**

Operation: A rate limiter class for controlling the rate of message sending.

### **Main()**

Inputs: None

Outputs: None

Operation: Initializes and starts the client application.

### 3. דגשים טכניים - Technical Highlights

במימוש הפרויקט נלקחו בחשבון מקרי קיצון, רובם מתייחסים לשגיאות מקריות או זדוניות שעלולות לצוץ במהלך ריצת האפליקציה. מימוש מנגנוני *try – except* מגן מפני מקרים של קריסה/אתחול של השרת, סגירה של חלונות ע"י המשתמש, ועוד, כפי שתוארו בחלק הקודם.

כמו כן, מומש ממשק משתמש גרפי (*GUI*) על מנת לשפר את חוויית המשתמש. ממשק המשתמש מומש באמצעות ספריית *tkinter* ומספר מודולים שהיא כוללת. ממשק המשתמש כולל חלונות לכל אחד משלבי ההתחברות, ההרשמה, בחירת חדר הצ'אט, חדר הצ'אט עצמו. כמו כן, ממשק המשתמש מנגיש את הפונקציונליות של האפליקציה בביצוע פעולות כמו שינוי סיסמה, העלאת קובץ לשרת, צפייה ברשימת המשתמשים המחוברים לחדר צ'אט כלשהו, ועזיבת הצ'אט. בנוסף, ממשק המשתמש מאפשר הקפצת חלונות התראה למשתמש במקרים הדורשים זאת.

### 4. הנחיות - Instructions

כאמור בדף השער, בחרתי לממש את הקוד על גבי מערכת הפעלה (*Windows 11 (64 – bit)* בשפת *Python* בגרסה 3.9. כמו כן, עבור החלקים של ביסוס, קיום וניהול התקשורת בין השרת ללקוח (*socket programming*) השתמשתי רק בספריה שראינו בכיתה (*socket*). באופן כללי, הספריות בהן השתמשתי במימוש הקוד הן ספריות הכלולות בשפה באופן דיפולטיבי ולא דורשות התקנות נוספות (מלבד הספרייה *cryptography* בה נעשה שימוש בשל ההנחיה המתירה על כך).

כמו כן, כנדרש במטלה, הקוד מתועד ומסמך זה מכיל הסבר על כל קובץ, פונקציה או מחלקה שנוצרו או שנעשה בהם שימוש, וצילומי מסך מתוך האפליקציה.

### 5. שאלות תיאורטיות - Theoretical Questions

#### 5.1. בטיחות - Security

5.1.1. נציין מספר חולשות אפשריות ונציע פתרונות שניתן לממש על מנת לשפר את בטיחות האפליקציה. חולשה אפשרית אחת, היא מנגנון אימות משתמש לא בטוח. על מנת להתחבר לצ'אט יש להזין שם משתמש וסיסמה שעלולים לדלוף. פתרון אפשרי הוא להטמיע אימות דו-שלבי כך שמשתמש שמעוניין להתחבר יצטרך לעבור עוד שלב כמו *MFA*. חולשה אפשרית נוספת, היא חשיפה למתקפת *DDOS*, בה ריבוי קיצוני של בקשות מהשרת פוגע בתפקוד האפליקציה. פתרון אפשרי הוא שדרוג מנגנון ה-*Rate Limiting* שמומש בצד הלקוח, ע"י התראה לשרת אם רף כלשהו של כמות בקשות נחצה וחסימת המשתמש שחצה אותו.

5.1.2. הסכנה בשליחת הודעות כ-*plaintext* היא חשיפת המידע המועבר בהודעות ליריב המאזין לערוץ התקשורת. במצב שכזה, תוכן ההודעות ואף מידע רגיש כגון פרטי משתמשים וסיסמאות יהיו חשופים. פתרון אפשרי הוא מימוש הצפנת *end – to – end* ע"י שימוש בסכמות קריפטוגרפיות אשר הוכחו כבטוחות, והצפנת המידע טרם שליחתו.

#### 5.2. מדרגיות - Scalability

5.2.1. ניתן לעצב את האפליקציה שלנו כך שתתמוך במספר רב של משתתפים. צעד שניתן לבצע הוא מימוש שרת התומך ב-*multi – threading*. כך השרת מקצה *thread* נפרד לכל לקוח מחובר ויכול לנהל את כל הבקשות המגיעות אליו בצורה א-סינכרונית ויעילה יותר. בנוסף, נוכל לממש את מסדי הנתונים שלנו בצורה יותר יעילה כאשר מדובר בהרבה דאטה (הרחבה בחלק הרלוונטי).



צווארי בקבוק אפשריים הם סדר גודל של פעולות המבוצעות על מסדי הנתונים (שליפה, עדכון, מחיקה), וביצוע פעולות המערבות מספר *thread*-ים ודורשות שימוש ב-*lock*.  
5.2.2. ישנן אסטרטגיות שניתן לממש על מנת לאזן את העומס על השרת ולנהל את משאביו ביעילות. אחת מהן היא הטמעת שכבות *cache*, כך שמידע שימושי יוחזק בשכבות ביניים ובקשות לגביו ייענו על ידיהן (במהירות וביעילות) ולא יגיעו אל השרת ואל מסדי הנתונים עצמם. אסטרטגיה נוספת היא ריבוי השרתים האמונים על מתן מענה לבקשות המגיעות מהלקוחות.

### 5.3. אמינות וסיבולת לתקלות - *Reliability and Fault Tolerance*

5.3.1. על מנת לשמור על אמינות האפליקציה ולהתמודד בכשלים ונפילות בלתי צפויות של השרת (או של הלקוח) מומשו מערכי התמודדות עם שגיאות באמצעות הצהרות *try – except*. מערכים אלו מומשו בנקודות בהן קריסה של כל אחד מהצדדים עלולה לפגוע בהתנהלות התקינה של האפליקציה. ספציפית במקרה של נפילת השרת, כאשר הלקוח מנסה לבצע פעולה כלשהי, התוכנית לא קורסת ומוצגת בפניו הודעה כי הקשר עם השרת אבד.  
5.3.2. ישנן מספר דרכים על מנת לוודא שהודעות שנשלחו לא נאבדו בדרך ואכן הגיעו אל השרת. אחת, היא מימוש מסד נתונים מתמיד (*Persistent DB*) על מנת לאחסן את המידע (פרטי משתמשים, הודעות) באופן בטוח כך שהשרת יוכל לגשת ולשחזר אותם גם לאחר קריסה. דרך נוספת, היא מימוש מערך *acknowledgement* המאשר כי הודעה התקבלה בהצלחה, והגדרת טיפול במקרה שלא.

### 5.4. אימות משתמש - *User Authentication*

5.4.1. אימות משתמשים הוא קריטי על מנת לוודא את זהות המשתמשים, ולבקר את נגישותם למידע ואפשרויות רגישות וחסויות. נוסף על כך, אימות משתמשים מאפשר למנוע גישות לא מאושרות לחשבונות פרטיים, ובכך שומר על פרטיות המשתמשים. מספר דרכים בהן ניתן לאמת את זהות המשתמשים הן אימות ע"י שם סיסמה (עם מגבלות חזקות וגיבוב), אימות מספר-שלבי הערב גם שימוש ב-*MFA*, ב-*SMS*, אמצעים ביומטריים וכולי.  
5.4.2. על מנת לשפר את הבטיחות הכללית של המערכת, ניתן להכריח משתמשים לבחור סיסמאות 'חזקות', לשמור את סיסמאות המשתמשים באופן מוצפן ומגובב, לאפשר למשתמשים לאפס את סיסמתם דרך אמצעי אחר (מייל, *SMS*) וכולי.

### 5.5. מקביליות וסנכרון - *Concurrency and Synchronization*

5.5.1. ישנם מספר אתגרים בניהול מערכת עם מספר חיבורים לשרת בו-זמנית. בינם ניהול גישה מקבילית למשאבים ומבני נתונים משותפים (מצב משתמשים, תורי הודעות...), מניעת מרוצים והשחתת מידע כאשר תהליכים מרובים שולפים ומשנים נתונים בו-זמנית, וסנכרון גישה לקטעים קריטיים בתוך הקוד על מנת להבטיח תפקוד תקין ועקביות.  
5.5.2. ניתן לממש שימוש בטוח ב-*thread*-ים וסנכרון במערכת ע"י שימוש ב-*lock*-ים. כלומר, כאשר השרת ניגש לבצע פעולה המערבת קריאה או כתיבה למסד הנתונים, או פעולה המערבת משתמשים רבים כגון *broadcast*, יופעל *lock* המונע מ-*thread*-ים נוספים לגשת למידע במהלך ביצוע הפעולה. כמו כן, ניתן לממש סנכרון ושימוש בטוח ע"י הטמעת פעולות אטומיות שאין להפריע למהלך ביצוען.

### 5.6. עיצוב פרוטוקול - *Protocol Design*

5.6.1. לטובת מימוש אפליקציית הצ'אט בחרתי בפרוטוקול *TCP*. בחרתי בו בגלל שהוא מתאפיין בהעברת מידע אמינה והוגנת, ללא שגיאות ולפי סדר השליחה. מאפיינים אלו הם קריטיים באפליקציה כמו

צ'אט. כמו כן, הפרוטוקול מאפשר פתיחה וסגירה של תקשורת דו-כיוונית וניתן למימוש בקלות ע"י ספריות *python* קיימות.

5.6.2. השרת והלקוח מבססים חיבור *TCP* ע"י תהליך לחיצת ידיים 3-כיוונית. תחילה, השרת מאזין ב-*Port* ספציפי ומחכה לחיבורים שיגיעו. הלקוח שולח בקשה בה הוא מביע את רצונו ליצור חיבור עם השרת (הודעת *SYN*). השרת מקבל את הודעת הלקוח, ושולח אישור על קבלתה ועל מוכנותו ליצור את החיבור (*SYNACK*). לבסוף, הלקוח מקבל את האישור של השרת ושולח אישור כי קיבל את האישור הנ"ל וכי החיבור נוצר (*ACK*). לאחר ביצוע פעולות אלה החיבור בין השרת לבין הלקוח מבוסס. מצידו של הלקוח נעשה שימוש בפקודות *socket(AF\_INET, SOCK\_STREAM)* ליצירת *TCP socket* ו-*connect((serverAddress, port))* ליצירת החיבור מצידו של השרת, נעשה שימוש בפקודות *socket(AF\_INET, SOCK\_STREAM)* ליצירת *TCP socket*, *bind((serverAddress, port))* על מנת להגדיר את הכתובת והפורט בה השרת מאזין, *listen(max clients)* על מנת להתחיל האזנה, ו-*accept()* על מנת לאשר חיבורים חדשים. כל הפקודות הנ"ל הן פקודות חוסמות (*blocking*). על מנת לנהל תקשורת תקינה, נגדיר כי האורך המקסימלי של פרגמנט שניתן לקבלה הוא 1024, וכי ההודעות מקודדות ומפוענחות חזרה בפורמט 'utf – 8'.

### 5.7. סידור ושליחת הודעות - *Message Ordering and Delivery*

5.7.1. ניתן להתייחס לבעיה פוטנציאלית בסידור ושליחת הודעות ע"י צירוף מספר סידורי או זמן שליחה אל ההודעה, כך השרת יוכל לדעת איזו הודעה נשלחה קודם. נוסף על כך, ניתן לשמור את ההודעות במסד הנתונים במבנה נתונים עם חשיבות לסדר. באופן זה, הסדר בו ההודעות נשמרו והוכנסו אל מסד הנתונים יישמר.

5.7.2. על מנת להבטיח שליחה אמינה ומתוזמנת של הודעות, ניתן לממש מנגנון אישור קבלה. כאשר הלקוח יקבל הודעה הוא ישלח הודעת אישור אל השרת. אם הודעת אישור שכזו לא מגיעה תוך זמן מוגדר מראש, השרת יישלח שוב את ההודעה.

### 5.8. אחסון מתמיד - *Persistent Storage*

5.8.1. אחסון מתמיד הוא חלק קריטי באפליקציה המאפשר לשמור על אמינות המידע ועל כך שמידע לא יאבד, במיוחד במקרים של קריסה או אתחול של השרת. באמצעות אחסון מתמיד השרת מאחסן מידע חיוני כגון פרטי משתמשים, היסטורית הודעות, הגדרות וכולי. בחרתי לממש את מסד הנתונים שלי בתור קבצי *JSON* - אחד לפרטי המשתמשים השומר לכל משתמש את שם המשתמש, הסיסמה וההרשאות שלו, ואחד להודעות השומר את כל ההודעות שנשלחו בצ'אט מחולקות לפי החדרים בהם נשלחו. בחרתי בפורמט זה בשל הפשטות, נוחות השימוש והקריאות שלו לעין אנושית. מאפיינים אלה מתאימים לסדר הגודל הקטן יחסית של האפליקציה ולדרישותיה מבחינת גודל הדאטה המאוחסן.

5.8.2. ישנן מספר גישות כיצד לממש את מסד הנתונים - (*relational DB (MySQL ...)*), (*NoSQL (MongoDB ...)*), קבצים (*txt, csv, JSON ...*) ועוד. בחירת גישה אחת על פני אחרת תלויה בדרישות האפליקציה, גודל הנתונים ועוד, ולכל אחת מהגישות יתרונות וחסרונות משלה. בחירתי בקבצי *JSON* מתאימה כל עוד גודל הנתונים סביר ומספר המשתמשים קטן, אך מעבר לכך הייתי עובר למימוש מסדי הנתונים כ-*relational DB* המאפשרים תמיכה בביצוע שאילתות מורכבות יותר לפי שדות וכולי, וגם בגישה מקבילית למסד הנתונים.