

דוח מסכם

# פרויקט גמר בקורס רשתות תקשורת מחשבים

## ניתוח תעבורה בפרוטוקול TCP/IP

קבוצה 5-61305

מגישות:

נעה סיאחצי

עדי חדד

מעין לוי

**תוכן עניינים:****חלק 1 - אריזת נתונים ולכידת מנות בעזרת Wireshark**

- ✓ יצירת קובץ ה-CSV ..... 3
- ✓ תיאור והסבר תהליך אריזת המנות (Encapsulation Process) ..... 3
- ✓ ניתוח תעבורה (Wireshark Analysis) ..... 6

**חלק 2 - כתיבת יישום רשת וניתוח תעבורה של אותו יישום**

- ✓ הסבר כללי על המערכת ומבנה הקוד ..... 10
- ✓ הוראות התקנה והרצה ..... 13
- ✓ דוגמאות קלט פלט וניתוחן ..... 14

**חלק 3 - תיאור שימוש בבינה מלאכותית****חלק 4 - יצירת GUI למשתמש - בונס**

## חלק 1: אריזה ולכידת מנות (Capture & Encapsulation)

בחלק זה של העבודה, נתמקד ביישום המעשי של תהליך התקשורת. נדגים כיצד הקוד קורא את ההודעות המאוחסנות בקובץ ה-CSV וממיר אותן לתעבורת רשת חיה, ובמקביל נציג את הממצאים כפי שנקלטו ב-Wireshark לצורך אימות וניתוח הפרוטוקול.

### ✓ יצירת קובץ ה-CSV

יצרנו את קובץ הקלט בעזרת AI, CHATGPT כדי לדמות תעבורת HTTP/Application תקינה. הקובץ מכיל את העמודות הבאות:

1. **msg\_id (מזהה הודעה):** מספר סידורי המאפשר מעקב אחר סדר הגעת החבילות ושלמותן ב-Wireshark.
2. **app\_protocol (פרוטוקול):** תווית המציינת את שם הפרוטוקול (כגון HTTP) שבו משתמשים בשכבת האפליקציה.
3. **src\_app (פורט מקור):** פורט הלקוח, המוזן לתוך שדה ה-Source בכותרת ה-TCP.
4. **dst\_appt (פורט יעד):** פורט השרת (הוגדר כ-12345), רלוונטי לסינון וזיהוי התעבורה הרלוונטית ב-Wireshark.
5. **message (תוכן):** המידע עצמו. נארז בסגמנט ה-TCP וניתן לקריאה כטקסט בזמן הניתוח.
6. **timestamp (תזמון):** קובע את מועד השליחה ומאפשר שליטה בקצב התעבורה (Delay) בזמן הריצה.

### ✓ תיאור והסבר תהליך אריזת המנות (Encapsulation Process)

תהליך האריזה בפרויקט מדמה את המתרחש במחסנית הפרוטוקולים (Protocol Stack) של מערכת ההפעלה, תוך מעבר משכבת האפליקציה ועד ליצירת רצף בייטים המוכנים למשלוח:

1. **שכבת היישום (Application):** ההודעה הגולמית מה-CSV מקודדת למחרוזת בייטים, (UTF-8) המהווה את המטען (Payload) של החבילה.
2. **שכבת התעבורה (TCP - Transport):** בניית **Header**: השתמשנו בפונקציה `build_tcp_header` כדי ליצור כותרת TCP תקנית בגודל 20 בתים. הכותרת כוללת את פורט המקור והיעד שהגדרנו ב-CSV. שדות `Source Port` ו-`Sequence Number` הוגרלו רנדומלית. **מנגנון ה-Checksum**: לחישוב אמינות הנתונים, בנינו "פסאודו-כותרת" (Pseudo-header) הכוללת את כתובות ה-IP ופרוטוקול ה-TCP, וביצענו חישוב Checksum על כלל החבילה. זהו

שלב קריטי המבטיח שהצד המקבל לא יפסול את החבילה כפגומה.  
**שימוש בדגלים (Flags):** הגדרנו דגל 180x (PSH+ACK). דגל ה-Push מורה למערכת ההפעלה ביעד להעביר את הנתונים מיד ליישום מבלי להמתין למילוי ה-Buffer.

### 3. שכבת הרשת (Network - IP):

הפונקציה `build_ip_header` אורזת את כותרת ה-TCP בתוך כותרת IP. כאן מוגדרים כתובות ה-IP.

### 4. התוצאה הסופית:

הקוד מחבר את שלוש השכבות (IP + TCP + Application Data) לרצף אחד המייצג Frame שלם שניתן להזרקה לכרטיס הרשת.

```
# Preview packet structure
src_ip = '127.0.0.1'
dst_ip = '127.0.0.1'
src_port = random.randint(1024, 65535)
dst_port = 12345
payload = b'Hello Packet (preview)'
pkt_preview = build_ip_header(src_ip, dst_ip, 20 + len(payload)) + build_tcp_header(src_ip, dst_ip, src_port, dst_port, payload) + payload
hexdump(pkt_preview)
```

```
0000 45 00 00 3e 21 e5 00 00 40 06 5a d3 7f 00 00 01  E...!...@.Z....
0010 7f 00 00 01 87 73 30 39 e5 33 ed f7 00 00 00 00  ....s09.3.....
0020 50 02 ff ff 13 1f 00 00 48 65 6c 6c 6f 20 50 61  P.....Hello Pa
0030 63 6b 65 74 20 28 70 72 65 76 69 65 77 29      cket (preview)
```

בצילום ניתן לראות את הבייטים הראשונים (...00 45) המייצגים את כותרת ה-IP ובסוף הרצף ניתן לזהות בבירור את הטקסט המקודד ("Hello Packet") מה שמוכיח אריזה מדויקת. (התמונה מהלכידה ב wireshark צולמה בתאריך שונה ולכן הביטים אינם תואמים בשלמות. אך ניתן לראות כי החבילה נארזה בצורה נכונה).

## ✓ פירוט והסבר על תהליך הרצת מחברת Jupiter:

### 1. טעינת הנתונים ותהליך השליחה:

בשלב הראשוני, נטען קובץ הנתונים (CSV) לתוך טבלת נתונים בשם `messages_df`. תהליך השליחה מתבצע באמצעות לולאת `for` על כלל השורות בטבלה. בכל איטרציה נשלף תוכן ההודעה (שדה `message`) ונשלח באמצעות אובייקט תעבורה (`transport`) שהוגדר מראש.

### 2. תהליך הכימוס (Encapsulation):

האובייקט `transport` מוגדר כמופע של המחלקה `RawTcpTranspor`. מחלקה זו אחראית על לוגיקת השליחה ועל בניית המנות. בעת הפעלת הפונקציה `transport.send` מתבצע קריאה לפונקציה `encapsulate`. מטרת פונקציה זו היא

"לארוז" את המידע משכבת היישום (הודעת הטקסט) ולהעבירו דרך שכבות התקשורת התחתונות עד ליצירת חבילה מלאה.

3. מתודולוגיית הלכידה (Wireshark Capture)  
פתחנו את Wireshark והאזנו לממשק Loopback, מכיוון שהתקשורת מתבצעת בתוך המחשב המקומי, החבילות אינן יוצאות לכרטיס הרשת הפיזי.  
כדי לבודד את המנות שלנו מכלל תעבורת המחשב, השתמשנו בפילטר תצוגה:  
`.tcp.port == 12345`

4. הרצנו את הסקריפט (Jupyter Notebook) ששלח את המנות.

5. עם סיום הלכידה, ביצענו השוואה בין ה-Hex Dump של החבילה ב-Wireshark לבין שורת ההודעה המקורית ב-CSV. האימות הראה התאמה של 100%, מה שמוכיח שתהליך האריזה והשליחה בוצע בהצלחה.

## ניתוח תעבורה (Wireshark Analysis)

לאחר הרצת הסקריפט הדרוש, באמצעות מחברת הפיתוח עם קלט ה-DATA (כפי שתואר מעלה), אשר מדגים יצירה ושליחה של מנות - כעת, נציג ניתוח מנה שנעשתה באמצעות Wireshark. הניתוח יוצג לפי סדר השכבות של המנה, החל מכותרות הפרוטוקולים, Headers, ועד לתוכן המידע, payload.

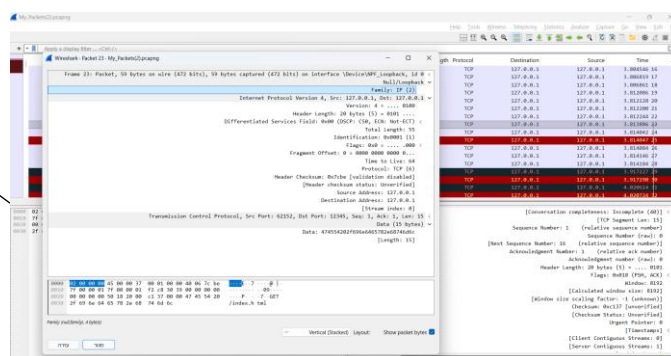
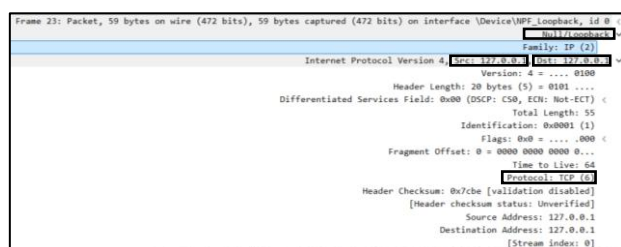
### 1. שכבת הרשת - Internet Protocol Version 4

בעקבות תהליך הכימוס הידני (Encapsulation) שבוצע בסקריפט, נרצה לאמת את הממצאים ב-Wireshark.

צילומי המסך מתוך האפליקציה נועדו להראות כי הגדרות שכבת הרשת בקוד תורגמו בהצלחה לכותרת IP תקינה, בדגש על כתובות המקור והיעד-127.0.0.1 ושדה הפרוטוקול.

על מנת לבודד את המנה הרלוונטית מתוך כלל תעבורת הרשת, השתמשנו בסינון הבא:

- `tcp.flags.push == 1 && tcp.flags.ack == 1 && tcp.port == 12345`
- `tcp.port == 12345`: סינון לפי פורט היעד שהוגדר בקוד הפיתוח.
- `tcp.flags.push == 1`: התמקדות במנות הנושאות מידע הדורשות העברה מיידית (Push), וסינון מנות ניהוליות.
- `tcp.flags.ack == 1`: וידוא כי המנה היא חלק מקישור פעיל ומאושר.



בצילום המסך ניתן לראות את כותרת ה-IP כפי שנלכדה ב-Wireshark. ניתן לראות שתהליך הכימוס בסקריפט בוצע בהצלחה.

### 1. כתובות - Addressing

כתובת המקור (Src) וכתובת היעד (Dst) הן 127.0.0.1 (מודגש בתמונה המוגדלת). כתובת זו הינה כתובת המחשב המקומי (Localhost) והיא מראה כי המנה הצליחה לעבור מתהליך אחד לשני בתוך אותו המחשב דרך ממשק ה-Loopback, כלומר מהמחשב המקומי לעצמו, ללא יציאה לרשת חיצונית, כפי שהוגדר בקוד הפיתוח בפונקציה `build_ip_header`.

## 2. שדה הפרוטוקול - Protocol

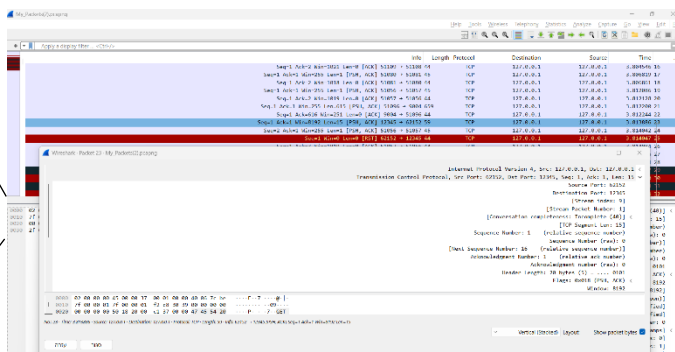
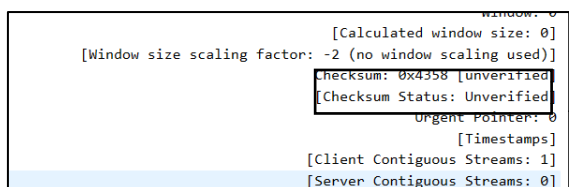
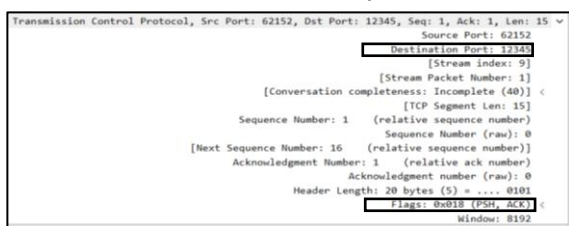
ניתן לראות כי המנה העברה בפרוטוקול TCP-6 (מודגש בתמונה המוגדלת). נתון זה קריטי לתהליך הכימוס, הוא מסמן למקבל ההודעה כי השכבה הבאה שיש לפענח היא כותרת TCP.

## 3. פרוטוקול Loopback

בשורה העליונה של פירוט המנה ניתן לראות את פרוטוקול loopback. כאשר מערכת ההפעלה מזהה כי כתובת היעד היא 127.0.0.1 (localhost) היא אינה משדרת את המנה לכרטיס הרשת הפיזי הפונקציה build\_ip\_header בקוד יוצרת מנת IP אך אינה יוצרת כותרת Ethernet.

## 2. שכבת התעבורה - Transmission Control Protocol

בהמשך לממצאי שכבת הרשת, שבה ראינו ששדה הפרוטוקול הצביע על קיומה של שכבת TCP, כעת נבחן את תוכן הכותרת עצמה. נרצה להראות כי הפרמטרים שהוזנו לפונקציה build\_tcp\_header תואמים את הפלט שנלכד וכי הדגלים משקפים את סטוס העברת המידע (Push) כפי שהוגדר בלולאת השליחה בקוד.



בצילום המסך ניתן לראות את כותרת ה-TCP כפי שנלכדה ב-Wireshark. הצילום מראה שהלוגיקה בסקריפט יושמה בהצלחה:

## • פורטים - Ports

ניתן לראות שפורט היעד (Dst Port) הוא 12345 (מודגש בתמונה). ערך זה תואם בדיוק למשתנה dst\_port = 12345 שהוגדר בקובץ הפייתון. פורט המקור (Src Port) הוא 62152 התואם את הפקודה random.randint, שנועדה לדמות הקצאת פורט דינמית של לקוח אמיתי.

## • דגלים - Flags

בשדה הדגלים ניתן לראות את הדגלים PSH + ACK. (מודגש בתמונה).  
בחלקו החמישי של הסקריפט, הרצנו פקודה עם הפרמטר flags=0x18. בניגוד לשלב יצירת הקשר (Handshake) שבו השתמשנו בדגל SYN (0x02), כאן מדובר במנה הנושאת מידע.

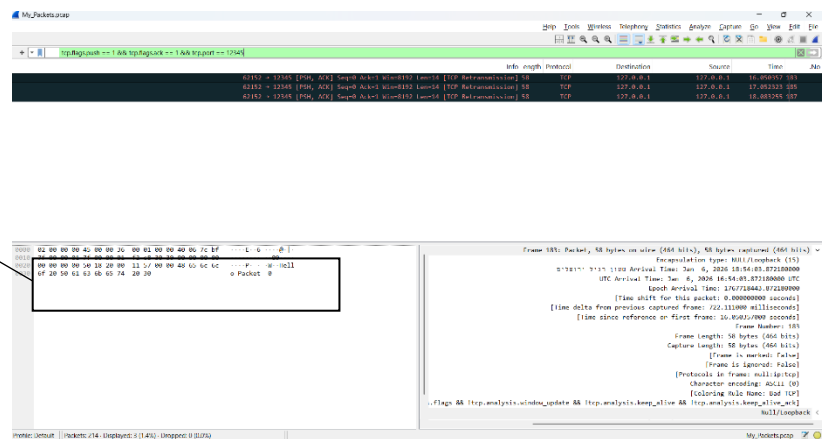
שדה הACK (Acknowledge) מצוין שהמנה מכילה אישור קבלה ושדה הPSH (Push) מסמן למקבל כי יש להעביר את המידע (Payload) מיד לשכבת האפליקציה, וכי הקישור תקין.

## • סיכום ביקורת - Checksum

ניתן לראות את הערך 0x4358 עם הסטטוס Unverified. הסטטוס "לא מאומת" נובע מכך שהתעבורה נלכדה ב Loopback ומערכת ההפעלה מבצעת אופטימיזציה ולכן לא תמיד מחשבת אותו בזמן הלכידה.  
הפונקציה checksum בקוד הפייתון ביצעה את החישוב המתמטי (סכימה ואינוורסיה של 16 ביט) כדי למלא שדה זה, כנדרש בפרוטוקול TCP לשמירה על שלמות המידע.

## 3. תוכן המידע - Payload

לאחר שווידאנו כי כותרות ה-IP וה-TCP נבנו ותפקדו כראוי, נרצה לוודא שגם השכבה הפנימית מתפקדת כראוי. מטרת צילום המסך הבא היא להוכיח כי המחרוזת שנשלחה מתוך קוד הפייתון עברה את תהליך הכימוס, שודרה ברשת, ופוענחה בהצלחה בצד המקבל ללא שיבושים.



```
00 36 00 01 00 00 40 06 7c bf 00 01 f2 c8 30 39 00 00 00 00 20 00 11 57 00 00 48 65 6c 6c 65 74 20 30
...E-6...@...
...09...
...P...W...Hell
o Packet 0
```

- בחלונית המידע הגולמי (Packet Bytes) בתחתית המסך, ניתן לזהות בבירור את הטקסט "Hello Packet 0". בצד שמאל ניתן לראות את הייצוג ההקסדצימלי (Hex) של תווים אלו.



- תוכן זה תואם את הפקודה `message.encode()` במחברת ההרצה.  
המחרוזת (String) הפכה לרצף של בתים (Bytes) לפני השליחה לSocket.  
היכולת של Wireshark להציג את הטקסט בחזרה מוכיחה כי הקידוד והשידור בוצעו בצורה תקינה.  
העובדה שהטקסט מופיע בדיוק כפי שנשלח, ללא תווים חסרים או משובשים, מהווה הוכחה סופית לכך שכל השכבות הקודמות (IP, TCP) תפקדו כהלכה ושמרו על שלמות המנה לאורך המעבר בLoopback.  
  
יש לציין, בניסוי זה, לא התבצע שידור פיזי (כבלים או אלחוט), שכן כאמור, כל התעבורה נוהלה באופן וירטואלי בתוך המחשב (Loopback Interface) ולכן, אין שום ייצוג לשכבה הפיזית.

## חלק 2 - יצירת יישום וניתוח תעבורה:

### הסבר כללי על המערכת ומבנה הקוד:

המערכת היא אפליקציית מסרים מיידיים (Instant Messaging) הפועלת במודל Client-Server על גבי פרוטוקול התעבורה TCP. המערכת מאפשרת למספר משתמשים להתחבר בו-זמנית, לזהות זה את זה ולנהל שיחות פרטיות בזמן אמת דרך ממשק גרפי (GUI). בנוסף, המערכת יודעת לנטר התנתקויות פתאומיות של משתמשים (התנתקויות רשת, ניתוק באמצע שיחה, סגירת צ'אט וכו') להתריע על כך למשתמש ולשרת, ולתפעל את הממשק בהתאמה.

### חלק א' - ארכיטקטורת המערכת

- **שכבת התעבורה: (Transport)** שימוש ב Sockets-מסוג SOCK\_STREAM להבטחת העברת נתונים אמינה, מסודרת וללא שגיאות.
- **שכבת היישום: (Application)** המידע נארז בפורמט JSON, המאפשר העברה מובנית של נתונים הכוללים את שם השולח, היעד ותוכן ההודעה.
- **מקביליות: (Concurrency)** השרת ממשש מנגנון Multi-threading המאפשר לו לנהל "שיחה" נפרדת עם כל לקוח מבלי לחסום חיבורים חדשים.

### חלק ב' - מבנה הקוד

#### 1. צד השרת - Server

השרת מהווה את רכיב הליבה של המערכת ואחראי על ניהול הקישוריות, אימות המשתמשים וניתוב ההודעות. כלל תעבורת הרשת, ובפרט הודעות על חיבור משתמשים חדשים וניתוק משתמשים קיימים, עוברות דרך השרת. השרת מנהל באופן ריכוזי את האירועים הללו כדי לשמור על סנכרון מלא של רשימת המשתמשים הזמינים לשיחה.

#### פונקציות מרכזיות ותהליכים:

- **start\_server()** - פונקציה זו יוצרת את ה-Socket הראשי ומאזינה לחיבורים נכנסים עבור כל בקשת התחברות, השרת יוצר Thread חדש המריץ את הפונקציה.
- **handle\_client(conn, addr)** - מנהלת את ה-Session מול לקוח ספציפי. הפונקציה רושמת את המשתמש במילון הגלובלי (clients), הממפה בין שם משתמש ל-Socket-שלו, ומנתבת את הלוגיקה לשלבים הבאים: בחירת חבר ושיחת הצ'אט עצמה. בסיום, היא דואגת לניקוי המשאבים ומחיקת המשתמש מהרשימה.

- **connect\_friend(conn, name)** - אחראית על לוגיקת הקישור בין משתמשים. הפונקציה בודקת האם החבר המבוקש קיים וזמין ברשימת הלקוחות המחוברים. אם החבר לא נמצא, היא מנהלת דו-שיח מול הלקוח המאפשר לו לנסות שוב או להתנתק.
- **chat\_friend(conn, friend\_name, name)** - לולאת העברת ההודעות. הפונקציה מאזינה להודעות מהלקוח ומנתבת אותן ליעדן. היא כוללת טיפול בפקודות מערכת כגון "list" להצגת מחוברים (וטיפול במקרי קצה כמו התנתקות פתאומית של הצד השני לשיחה).
- **Create\_Msg()** - פונקציית עזר המיישמת את פרוטוקול האפליקציה. כל הודעה נארזת לאובייקט JSON המכיל שדות מוגדרים (שולח, נמען, תוכן), מה שמאפשר סטנדרטיזציה ומונע שגיאות פענוח.

### מימוש ניהול המשתמשים בשרת נעשה באמצעות מבנה נתונים מסוג Dictionary מילון .

מבנה זה נבחר לצורך מיפוי חד-ערכי ויעיל בין המזהה הייחודי של המשתמש, ה Username- שמשמש כ-Key, לבין אובייקט החיבור הפעיל שלו- ה Socket שמשמש כ-Value. מילון הלקוחות (clients) בקוד השרת ממומש כמבנה נתונים מסוג Hash Table (טבלת גיבוב), שמאפשר גישה מהירה מאוד לנתונים.

כאשר מבצעים: `clients[name] = conn`

ערך הגיבוב מחושב על בסיס שם המשתמש והוא מתורגם למיקום פנימי בזיכרון שבו נשמר הקישור בין שם המשתמש לבין אובייקט החיבור (conn, שהוא socket).

בזכות מבנה הנתונים כאשר רוצים לשלוף את החיבור באמצעות: `clients["client1"]`

המערכת יכולה להגיע אליו בזמן קבוע ( $O(1)$ ) ברוב המקרים – כלומר בלי לעבור על כל המשתמשים אחד-אחד, אלא באמצעות חישוב ישיר שמוביל למקום הנכון.

## 2. צד הלקוח - Client

הלקוח משמש כממשק המשתמש. ממשק זה יודע לתמוך בביצוע שתי פעולות במקביל: המתנה לקלט מהמשתמש (הקלדה) והאזנה להודעות נכנסות מהשרת.

### פונקציות מרכזיות ותהליכים:

- **start\_client()** - הפונקציה הראשית שמבצעת את יצירת ה-Socket וההתחברות לשרת. היא מנהלת את תהליך ההזדהות (Handshake) הראשוני באופן סנכרוני, ולאחר מכן

מפעילה את תהליך ההאזנה ברקע. הלולאה הראשית בפונקציה זו אחראית אך ורק על קבלת קלט מהמקלדת ושליחתו לשרת.

- **receive\_messages(sock)** - פונקציה זו רצה ב-Thread נפרד (Daemon) ואחראית על האזנה לתעבורה מהשרת. היא מבצעת קריאה חוסמת (recv), מפענחת את הודעות ה-JSON ומציגה אותן למסך. הפרדה זו קריטית כדי שהודעות מחברים יופיעו מיד כשהן מגיעות, ללא תלות באם המשתמש מקליד כרגע או לא.

- **send\_message()** - שולחת את הקלט מהמשתמש לשרת לאחר קידוד ל-UTF-8.

- **friend\_connection(client\_socket)** - מטפלת בשלב הספציפי של בחירת החבר לשיחה מול השרת לפני תחילת הצ'אט.

**מנגנון סנכרון** - הקוד משתמש ב-Event.threading כדי לסנכרן בין ה-Thread המאזין ל-Thread הכותב. כאשר השרת שולח שאלה המצריכה תשובה ספציפית (כמו "enter yes" לניסיון חוזר), תהליך ההאזנה מסמן את הדגל, והלולאה הראשית יודעת להתייחס לקלט הבא כתשובה למערכת ולא כהודעת צ'אט רגילה.

### 3. סיכום האינטגרציה

המערכת מדגימה יישום מלא של תקשורת נתונים, כאשר השרת מחזיק את תמונת המצב העדכנית ומנתב תעבורה. הלקוחות פועלים בקצוות בצורה אסינכרונית. השימוש ב-JSON עוטף את המידע הגולמי ומאפשר לשני הצדדים ("Client" ו-"Server") לדבר באותה שפה, לזהות ניתוקים ולנהל שגיאות בצורה אלגנטית.

## הוראות התקנה והרצה:

כדי להריץ את המערכת בהצלחה, יש לוודא שסביבת הפיתוח כוללת Python 3 ומעלה.

### 1. דרישות מוקדמות:

אין צורך בהתקנת ספריות חיצוניות, המערכת מתבססת על ספריות מובנות של Python- socket, threading, json, tkinter.

### 2. שלבי ההרצה:

כדי לדמות סביבת רשת אמיתית עם מספר משתמשים, יש לבצע את השלבים הבאים בסדר זה:

#### ✓ הפעלת השרת:

- פתח טרמינל והרץ את פקודת ההפעלה: `python server.py`
- השרת ידפיס הודעה המאשרת שהוא מאזין בכתובת 0.0.0.0 ובפורט 10000.

#### ✓ הפעלת לקוח א':

- פתח טרמינל חדש והרץ: `python client.py`.
- במסך הכניסה, הזן כתובת IP (למשל 127.0.0.1), פורט 10000 ושם משתמש.

#### ✓ הפעלת לקוח ב':

- פתח טרמינל נוסף והרץ שוב את `python client.py`.
- הזן שם משתמש שונה כדי לאפשר לשרת למפות את המשתמשים בצורה נכונה.

#### ✓ תחילת הצ'אט:

- לאחר החיבור, השרת יבקש ממך להזין את שם החבר איתו תרצה לדבר.
- הזן את השם של הלקוח השני והתחל בהתכתבות.

#### ✓ פקודות צ'אט

- לאחר ההתחברות, ניתן להשתמש בפקודות המיוחדות הבאות:
- "list": הקלד פקודה זו כדי להציג רשימה בזמן אמת של כל המשתמשים המחוברים כעת.
- "Exit": הקלד פקודה זו כדי להתנתק מהשרת.

## דוגמאות קלט פלט וניתוח ב-Wireshark:

בחלק זה הרצנו את השרת, הרצנו שני לקוחות, ביצענו שיחה ביניהם ולכדנו ב-Wireshark את השלבים הבאים:

### לחיצת היד המשולשת [ACK] -> [SYN, ACK] -> [SYN]

The image shows a Wireshark capture of a TCP three-way handshake. The left pane displays a list of three packets. The middle pane shows the details of the first packet, which is a SYN packet from 10.169.139.175 to 10.169.139.170. The right pane shows the packet bytes in hexadecimal and ASCII.

בצילום המסך ניתן לראות את תהליך יצירת החיבור בין הלקוח (בעל IP 10.169.139.170) לשרת (בעל IP 10.169.139.175) בפרוטוקול TCP.

התהליך מורכב משלוש מנות (Packets) עוקבות: חבילה

חבילה ראשונה - **Client → Server: [SYN]**

הלקוח יוזם את ההתקשרות ושולח בקשת סנכרון לשרת בפורט 10000, חבילה זו מסמנת את הרצון לפתוח ערוץ תקשורת.

חבילה שנייה - **Server → Client: [SYN, ACK]**

השרת מקבל את הבקשה, מאשר אותה באמצעות דגל ה- ACK (Acknowledgment) ושולח בקשת סנכרון משלו (SYN) חזרה ללקוח.

חבילה שלישית - **Client → Server: [ACK]**

הלקוח מקבל את אישור השרת ושולח אישור סופי (ACK), עכשיו ה-Socket פתוח בשני הצדדים מעבר למצב (ESTABLISHED).

מערכת ההפעלה מוכנה להעברת הנתונים של אפליקציית הצ'אט.

תהליך זה מבטיח שהצדדים מסונכרנים ומוכנים לקלוט מידע ללא איבוד נתונים.

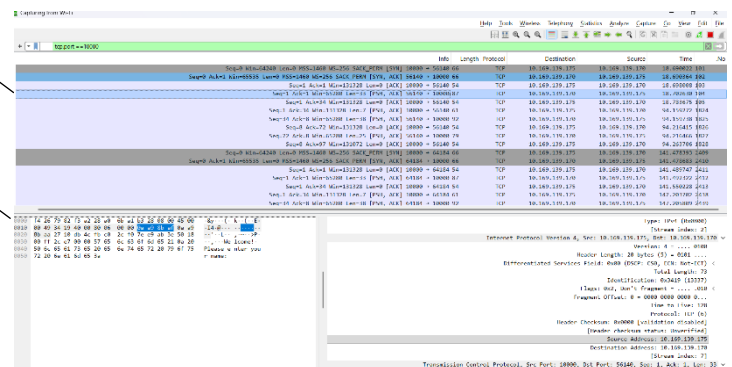
## העברת מידע (Data Transfer - PSH, ACK)

Seq=1 Ack=1 Win=65280 Len=33 PSH, ACK 56140 → 10000:87 TCP 10.169.139.170 10.169.139.175 10.702630 104

```

0000 f4 26 79 82 f3 a2 28 a0 6b a1 b3 28 08 00 45 00  .&y... ( k...-E-
0010 00 49 34 19 40 00 80 06 00 00 0a a9 8b af 0a a9  .I4.@... ..-...
0020 8b aa 27 10 db 4c fb c0 2c f0 7e e9 ab 3e 50 18  .-...-P
0030 00 ff 2c e7 00 00 57 65 6c 63 6f 6d 65 21 0a 20  .,...We lcome!-
0040 50 6c 65 61 73 65 20 65 6e 74 65 72 20 79 6f 75  .Please e nter you
0050 72 20 6e 61 6d 65 3a                               r name:

```



בתמונה אפשר לראות חבילה מפורט 10000 (השרת) לפורט 56140 (הלקוח).

56140 הוא פורט זמני שהמערכת של הלקוח בחרה כשהוא התחבר.

אורך המידע (Len) הוא 33 בתים.

החבילה מכילה מידע המסומנת בדגל [PSH,ACK]

1. **ACK (Acknowledgment)**: כל חבילה שנשלחת אחרי שלב לחיצת היד הראשוני חייבת לכלול דגל ACK. כלומר ההודעה הזו תקינה והיא חלק מהשיחה הקיימת.
2. **PSH (Push)**: דגל זה מנחה את פרוטוקול ה-TCP בצד המקבל לא להמתין למילוי ה-Buffer אלא להעביר את המידע מיד לשכבת האפליקציה, חיוני באפליקציות זמן אמת כמו צ'אט כדי למנוע השהיות.
3. **תוכן החבילה (Payload)**: בחלק התחתון של המסך ניתן לראות את המידע עצמו כפי שנשלח ברשת, בצילום רואים את המחרוזת "welcome! Please enter your name:".

## התחברות של 6 לקוחות:

צד שרת: ✓

```
Server is listening on 0.0.0.0:10000
Client connected from: ('10.169.139.170', 58338)
Added client1 to client list
Client connected from: ('10.169.139.170', 58341)
Added client2 to client list
Client connected from: ('10.169.139.170', 62182)
Added client3 to client list
Client connected from: ('10.169.139.170', 55955)
Added client4 to client list
Client connected from: ('10.169.139.170', 55957)
Added client5 to client list
Client connected from: ('10.169.139.170', 55959)
Added client6 to client list
```

```
def start_server():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        server.bind((HOST, PORT))
        server.listen()
        print(f"Server is listening on (HOST):{PORT}")
    except Exception as e:
        print(e)

    while True:
        conn, addr = server.accept()
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()
```

```
PS C:\Users\adiha> C:\Users\adiha\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/adiha/Desktop/HIT/Server.py"
Server is listening on 0.0.0.0:10000
Client connected from: ('10.169.139.170', 58338)
Added client1 to client list
Client connected from: ('10.169.139.170', 58341)
Added client2 to client list
Client connected from: ('10.169.139.170', 62182)
Added client3 to client list
Client connected from: ('10.169.139.170', 55955)
Added client4 to client list
Client connected from: ('10.169.139.170', 55957)
Added client5 to client list
Client connected from: ('10.169.139.170', 55959)
Added client6 to client list
```

בטרמינל של השרת נראה התחברות של 6 לקוחות client1 עד client6 אחד אחרי השני. השרת פועל במודל Multi-threading כדי לאפשר למספר לקוחות להתחבר ולנהל שיחות במקביל. ניהול המשתמשים המחוברים לשרת מתבצע באמצעות מבנה נתונים מסוג מילון, מבנה זה משמש למיפוי בין זהות המשתמש לבין ערוץ התקשורת שלו.

**הקצאת Threads:** עבור כל לקוח שמתחבר, השרת יוצר תהליכון ייעודי ונפרד. גישה זו מבטיחה שהשרת הראשי נשאר פנוי לקבלת לקוחות נוספים, בעוד שהתהליכונים מטפלים בתעבורת המידע מול הלקוחות הקיימים.

**זיהוי הלקוחות:** למרות שכל הלקוחות מגיעים מאותה כתובת IP (10.169.139.170), המערכת מבדילה ביניהם באמצעות פורט המקור הייחודי המוקצה לכל חיבור (לדוגמה: 58338, 58341).

**ניהול המשתמשים:** המפתחות הם שמות המשתמשים, כאשר כל שם הוא ייחודי. הערכים הם אובייקטי socket מסוג conn, המייצגים את חיבור התקשורת הפיזי בין השרת ללקוח.



## תחילת CONNECTION בין שני לקוחות וניתוק אחד הלקוחות:

צד שרת: ✓

נוכל לראות ש-client2 ו-client3 רוצים לדבר אחד השני, נפתח צ'אט ביניהם.

client2 wants to talk to client3  
client3 wants to talk to client2

```

1  """
2  Server module for the chat application.
3  Handles client connections and messaging.
4  """
5  import socket
6  import threading
7
8  HOST = "0.0.0.0"
9  PORT = 10000
10
11 clients = {}
12
13 def connect_friend(conn, name):
14     """Handles the logic for connecting two friends."""
15     ...
16
17 PS C:\Users\adiba> & C:\Users\adiba\AppData\Local\Programs\Python\Python311\python.exe "c:/Users/adiba/Desktop/NET/Computer Network
Server is listening on 0.0.0.0:10000
Client connected from: ('10.169.139.170', 58338)
Add client1 to client list
Client connected from: ('10.169.139.170', 58341)
Add client2 to client list
Client connected from: ('10.169.139.170', 62182)
Add client3 to client list
Client connected from: ('10.169.139.170', 55955)
Add client4 to client list
Client connected from: ('10.169.139.170', 55957)
Add client5 to client list
Client connected from: ('10.169.139.170', 55959)
Add client6 to client list
client6 wants to talk to client1
client1 wants to talk to client2
client2 wants to talk to client3
client3 wants to talk to client2
Client disconnected abruptly: ('10.169.139.170', 58341)
Removed client2 from client list
Socket closed for ('10.169.139.170', 58341)
  
```

צד לקוח: ✓

```

7  PORT = 10000
8
9  waiting_system_answer = threading.Event()
10
11 def getJsonMsg(msg):
12     try:
13         return json.loads(msg)
14     except:
15         return {}
16
17 def receive_messages(sock):
18     while True:
19         ...
20
21 PS C:\Users\P0035881\OneDrive - Ness Israel\Desktop\PythonProj
ectGroup100> & C:\Users\P0035881\AppData\Local\Programs\Python
Welcome!
Please enter your name: client2
Hello client2, you are now connected!
enter your friend's name: client3
Connection successful! You can now chat with client3.
hello from client2

client3: hello from client3
exit
PS C:\Users\P0035881\OneDrive - Ness Israel\Desktop\PythonProj
ectGroup100>
  
```

נראה את הצ'אט בין הלקוחות:

client2 שולח שלום ואחרי client3 שולח שלום.

client2 כותב exit ויוצא מהשיחה.

client3 שולח הודעה are you here? לא נשלחת לclient2 כי הוא כבר מנותק.

client3 מקבל הודעת מערכת שההודעה שלו לא נשלחה כי החבר שלו לא מחובר, ובמידה והוא

רוצה לדבר עם חבר אחר שיכתוב yes.

צד שרת ✓

```
Client disconnected abruptly: ('10.169.139.170', 58341)
Removed client2 from client list
Socket closed for ('10.169.139.170', 58341)
```

בתמונה רואים תיעוד של התנתקות לא צפויה של client2 מהשרת, הסרתו מרשימת המחוברים וסגירת החיבור.

ברגע שהלקוח סוגר את החלון בפתאומיות השרת מוחק את המשתמש מהמילון כדי למנוע שליחת הודעות למשתמש שאינו זמין יותר.

בסיום socket נסגר, הזיכרון במחשב והפורט משוחררים לטובת משתמשים אחרים.

Wireshark

Info	Length	Protocol	Destination	Source	Time	No.
10000 → 62182 [PSH, ACK] Seq=151 Ack=15 Win=65280 Len=29	83	TCP	10.169.139.170	10.169.139.175	496.040667	13999
10000 → 58341 [ACK] Seq=151 Ack=33 Win=65280 Len=0	54	TCP	10.169.139.170	10.169.139.175	496.087664	14000
62182 → 10000 [ACK] Seq=15 Ack=180 Win=65280 Len=0	54	TCP	10.169.139.175	10.169.139.170	496.092282	14001
62182 → 10000 [PSH, ACK] Seq=15 Ack=180 Win=65280 Len=18	72	TCP	10.169.139.175	10.169.139.170	513.762089	14098
10000 → 58341 [PSH, ACK] Seq=151 Ack=33 Win=65280 Len=29	83	TCP	10.169.139.170	10.169.139.175	513.762530	14099
10000 → 62182 [ACK] Seq=180 Ack=33 Win=65280 Len=0	54	TCP	10.169.139.170	10.169.139.175	513.805307	14100
58341 → 10000 [ACK] Seq=33 Ack=180 Win=65280 Len=0	54	TCP	10.169.139.175	10.169.139.170	513.850622	14102
58341 → 10000 [RST, ACK] Seq=33 Ack=180 Win=0 Len=0	54	TCP	10.169.139.175	10.169.139.170	653.945107	16248
62182 → 10000 [PSH, ACK] Seq=33 Ack=180 Win=65280 Len=19	73	TCP	10.169.139.175	10.169.139.170	856.493138	20741
10000 → 62182 [PSH, ACK] Seq=180 Ack=52 Win=65280 Len=104	158	TCP	10.169.139.170	10.169.139.175	856.493338	20742
62182 → 10000 [ACK] Seq=52 Ack=284 Win=65280 Len=0	54	TCP	10.169.139.175	10.169.139.170	856.544713	20744
62182 → 10000 [PSH, ACK] Seq=52 Ack=284 Win=65280 Len=3 [TCP PDU reassembled in 2113..	57	TCP	10.169.139.175	10.169.139.170	886.380865	20888
10000 → 62182 [PSH, ACK] Seq=284 Ack=55 Win=65280 Len=25	79	TCP	10.169.139.170	10.169.139.175	886.381106	20889
62182 → 10000 [ACK] Seq=55 Ack=309 Win=65280 Len=0	54	TCP	10.169.139.175	10.169.139.170	886.432611	20890

0000 f4 26 79 82 f3 a2 28 a0 6b a1 b3 28 08 00 45 00	0010 00 90 34 7a 40 00 80 06 00 00 0a a9 8b af 0a a9	0020 8b aa 27 10 f2 e6 cb 0b 3b 0c cc 32 fd 4e 50 18	0030 00 ff 2d 2e 00 00 53 79 73 74 65 6d 3a 20 4d 65	0040 73 73 61 67 65 20 6e 6f 2a 20 73 65 6e 74 2e 20	0050 46 72 69 65 6e 64 20 69 73 20 6e 6f 74 20 63 6f	0060 6e 6e 63 74 65 64 2e 0a 20 69 66 20 79 6f 75	0070 20 77 61 6e 74 20 74 6f 20 63 68 61 74 20 77 69	0080 74 68 20 64 69 66 66 72 65 6e 74 20 66 72 69 65	0090 6e 64 20 65 6e 74 65 72 20 79 65 73 3a 20
--	--	--	--	--	--	---	--	--	--

re (1264 bits), 158 bytes captured (1264 bits) on interface \Device\NPF\_{305FA4AC-F4A0-6B-A1-B3-28}, Dst: Intel\_E2:f3:a2 (f4:26:79:82:f3:a2)  
 .169.139.175, Dst: 10.169.139.170  
 ort: 10000, Dst Port: 62182, Seq: 180, Ack: 52, Len: 104

client3 ניסה לשלוח הודעה ל-client2 שהתנתק בפתאומיות.

השרת זיהה ש-client2 לא נמצא במילון המחוברים (clients).

השרת שלח הודעת מערכת ל-client3 שמסבירה את השגיאה ומציעה למשתמש להקליד yes כדי לבחור חבר חדש.

58341 → 10000 [RST, ACK] Seq=33 Ack=180 Win=0 Len=0 54 TCP 10.169.139.175 10.169.139.170 653.945107 16248

**דגל (Reset) RST** – מעיד על סיום פתאומי/מידי של חיבור ה-TCP מצד הלקוח, ולא על סגירה "מסודרת" ותקינה. הטרמינל בצד הלקוח נסגר בלי לבצע את תהליך הסגירה הסטנדרטי באמצעות לחיצת היד (handshake) של FIN. כתוצאה מכך, מערכת ההפעלה שלחה חבילת Reset כדי לשחרר מיד את משאבי הסוקט בצד השרת.

### שיחה עם משתמש אחר:

```
if you want to chat with diffrent friend enter yes:yes
enter your friend's name:client3
Connection successful! You can now chat with client3.
```

```
clients.py M X
client > clients.py > ...
7 PORT = 10000
8
9 waiting_system_answer = threading.Event()
10
11 def getJsonMsg(msg):
12     try:
13         return json.loads(msg)
14     except:
15         return {}
16
17 def receive_messages(sock):
18     while True:
19         try:
20             msg = sock.recv(1024).decode('utf-8')
21
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\P0035881\OneDrive - Ness Israel\Desktop\PythonProjectGroup100>
on314/python.exe "c:/Users/P0035881/OneDrive - Ness Israel/Desktop/PythonP
Welcome!
Please enter your name:client1
Hello client1, you are now connected!
enter your friend's name:client2
Error: User 'client2' is not connected. You cannot send messages.
if you want to chat with diffrent friend enter yes:yes
enter your friend's name:client3
Connection successful! You can now chat with client3.
list
System: Connected users: client1, client3
```

client1 רוצה לדבר עם client2, השרת בודק אם client2 קיים במילון clients, השרת לא מוצא אותו ברשימה אז הוא שולח הודעת שגיאה לclient1 ומחכה שהמשתמש יקליד "yes" כדי לנסות שוב. כאשר מתקבל "yes", הלולאה while True מתחילה מחדש, ומאפשרת למשתמש להזין "client3", שנמצא בהצלחה במילון — וכך נוצר חיבור הצ'אט.

## צפייה בכל המשתמשים המחוברים:

```
list
System: Connected users: client1, client3
```

The screenshot shows a Python IDE with a file named `clients.py`. The code defines a server that listens on port 10000. It has two main functions: `getJsonMsg(msg)` which parses incoming JSON messages, and `receive_messages(sock)` which handles the socket connection. The terminal output shows the server's interaction with clients. It prompts for a name, connects `client1`, and then `client2`. When `client2` tries to send a message, an error is shown: "Error: User 'client2' is not connected. You cannot send messages." After `client2` enters 'yes' to chat with a different friend, the server prompts for another name, `client3`. When `client3` connects, the terminal shows "Connection successful! You can now chat with client3." followed by a red box around the word `list` and the system message "System: Connected users: client1, client3".

```
clients.py M X
client > clients.py > ...

7  PORT = 10000
8
9  waiting_system_answer = threading.Event()
10
11 def getJsonMsg(msg):
12     try:
13         return json.loads(msg)
14     except:
15         return {}
16
17 def receive_messages(sock):
18     while True:
19         try:
20             msg = sock.recv(1024).decode('utf-8')
21
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\P0035881\OneDrive - Ness Israel\Desktop\PythonProjectGroup100>
on314/python.exe "c:/Users/P0035881/OneDrive - Ness Israel/Desktop/PythonP
Welcome!
Please enter your name:client1
Hello client1, you are now connected!
enter your friend's name:client2
Error: User 'client2' is not connected. You cannot send messages.
if you want to chat with different friend enter yes:yes
enter your friend's name:client3
Connection successful! You can now chat with client3.
list
System: Connected users: client1, client3
```

נוכל לראות שהמשתמש `client1` שולח את המילה `list`, השרת מקבל אותה בתוך הלולאה של `chat_friend` לפני העברת ההודעה לחבר.

השרת מזהה שהקלט הוא פקודה באמצעות הבדיקה: `if data_d.lower() == "list":`

אם התנאי מתקיים, הקוד ניגש למילון הגלובלי `clients`, שמשמש כ"זיכרון" של השרת לכל החיבורים הפעילים ( `clients.keys()` מכיל את כל שמות המשתמשים).

הפקודה `join(...)`, "לוקחת את המפתחות של המילון ומחברת אותם למחרוזת אחת (למשל `"client1, client3"`) כדי שיהיה קריא.

לאחר מכן השרת עוטף את הרשימה בהודעת `System` בפורמט `JSON` ושולח אותה חזרה רק ללקוח שביקש (`conn.sendall()`).

ההודעה `list` לא תישלח לחבר בצ'אט בזכות הפקודה `continue` שגורמת ללולאה להתחיל מחדש.

### חלק 3 - תיאור שימוש בבינה מלאכותית

במהלך העבודה על הפרויקט, נעזרנו ב AI (Gemini, Chat GPT) על מנת לבדוק את נכונות הקוד, לשפר את הקוד ולחשוב על מקרי קצה נוספים שיכולים לקרות במערכת מסוג זה, על מנת להגיע לפרויקט ברמה גבוהה ומקצועית.

נעזרנו בבינה המלאכותית בכמה שלבים:

- **יצירת נתוני הקלט (CSV) :**

בחלק א' של הפרויקט, נעזרנו ב AI כדי ליצור טבלה המדמה תעבורת HTTP/Application תקנית. ה AI יצר עבורנו קובץ CSV עם נתונים פיקטיביים, בו השתמשנו במהלך החלק הראשון של העבודה.

- **זיהוי ופתרון שגיאות (Debugging) :**

במהלך כתיבת הקוד נתקלנו בבעיות טכניות, כמו חסימת השרת נחסם (Blocking) ולא הצליח לקבל לקוחות חדשים בזמן שדיבר עם לקוח קיים. בעזרת ה-AI הבנו איך להטמיע מנגנון של Multi-threading שבו כל לקוח מקבל תהליכון נפרד בשרת, מה שפתר את הבעיה ואפשר עבודה במקביל.

- **בניית הלוגיקה של השרת והלקוח :**

בעזרת תשאול ה-AI, הבנו אילו מקרי קצה עלולים לקרות ולהשפיע על איכות המערכת. בהתאמה, הוספנו טיפולים ייעודיים בקוד שיודעים לתמוך במקרים אלו כפי שפורט קודם לכן.

- **עיצוב הממשק הגרפי (GUI) :**

השתמשנו בכלי כדי לעצב ממשק למשתמש.

#### דוגמאות לפרומפטים :

1. **ליצירת הנתונים :**

"צור לי קובץ CSV עם 10 שורות של הודעות המדמות תעבורת אפליקציה. תכלול את העמודות הבאות: עמודה של מזהה הודעה, שם הפרוטוקול, פורט מקור ופורט יעד, תוכן ההודעה וזמן שליחה."

2. **לפתרון שגיאות :**

"בניתי שרת TCP בפיייתון, כשאני מנסה לחבר לקוח שני הוא לא מצליח להתחבר עד שהלקוח הראשון מתנתק. איך אני משתמש ב Threading כדי שהשרת יטפל בכמה אנשים במקביל מבלי להיתקע?"

3. **ללוגיקה של השרת ומקרי הקצה:**

"מימשנו קוד התומך בהעברת מידע בתצורת שרת לקוח. באילו מקרים פתאומיים עשויים המשתמשים להיתקל שכדאי להוסיף לתמיכה בקוד שלנו?"

## חלק 4 - יצירת GUI למשתמש - בונוס

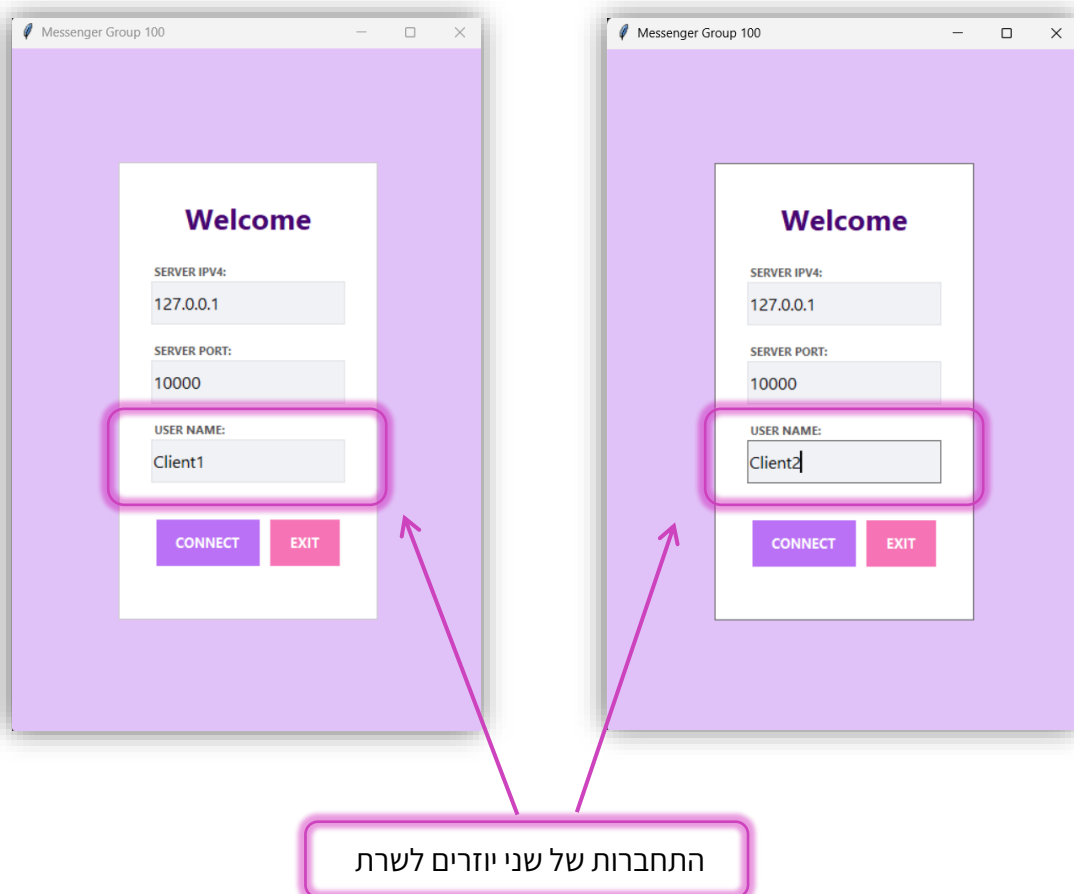
בחלק זה יצרנו ממשק למשתמש שיהיה נוח וויזואלי.

נעזרנו בבינה מלאכותית על מנת ליצור את הממשק הנכון ביותר אשר לא מסבך את המשתמש ובתוכו יש הוראות ברורות לתפעול והסברים במידה וישנם ניתוקים.

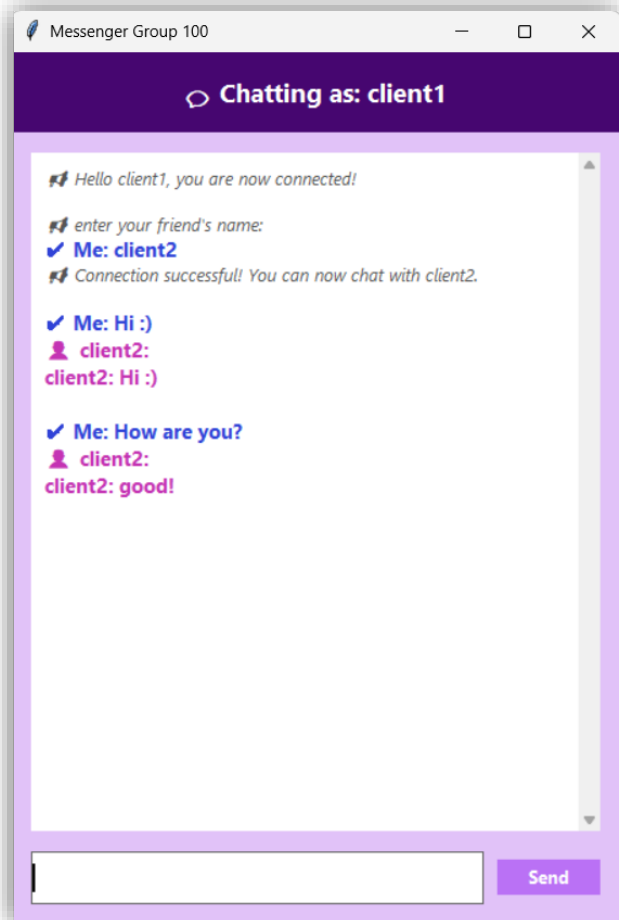
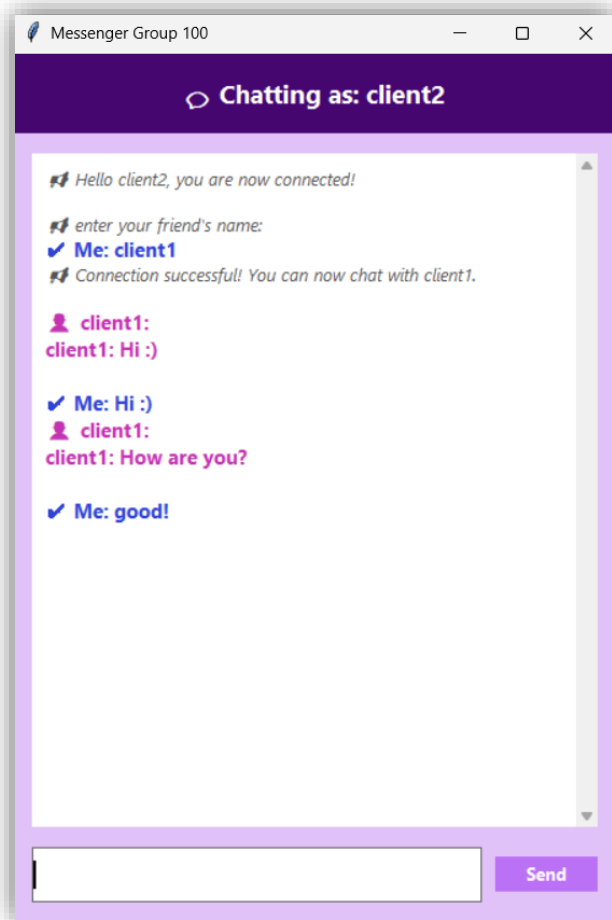
### ✓ מסך כניסה:

כולל 3 פרמטרים למילוי ע"י המשתמש ותלויים במס' המחשבים בשימוש:

1. SERVER IPV4 – כתובת IP פנימית (במידה וישנם 2 מחשבים, מכניסים את כתובת IP שמריץ את השרת).
2. SERVER PORT – 10000 (מתאים לשרת)
3. USER NAME – שם המשתמש לזיהוי בשרת.



## שיחה בין 2 משתמשים: ✓



## הודעת שגיאה במצב של ניתוק: ✓

### זריקת שגיאה מהשרת-

המשתמש השני התנתק ולכן  
ההודעה לא נשלחה.

במידה והמשתמש רוצה לפנות  
למשתמש אחר הוא יצטרך לכתוב  
את שמו והחיבור יתבצע.

