רשתות תקשורת פרויקט סיום – SQL

מגישים: אביב תורג'מן - 208007351

אלון סויסה – 211344015

תוכן עניינים:

הסבר כללי על הפרויקט:

אנחנו בחרנו לעשות את הפרויקט SQL על קבוצות מליגת האלופות. המידע של טבלת ה-SQL הוא השחקנים של הקבוצות והנתונים שלהם (שם, דירוג, קבוצה, תפקיד, גולים, בישולים).

תחילה יצרנו מחלקה בשם PL_player.py שמאתחלת שחקן לפי הנתונים הנ"ל ונותנת גישה לנתונים אלו עם פונקציות get.

לאחר מכן יצרנו מחלקה חדשה בשם DATA.py שתחזיק את כל המידע של הקבוצות והשחקנים בתוך מערך של PL_player שנקרא data. בנוסף, יצרנו מחלקה שנקראת query_object.py שמטרתה היא ליצור שאילתות ויש לה מתודה אחת בשם do_query אשר מחזירה מערך של PL_player לפי השאילתה.

לאחר מכן יצרנו 2 שרתים (TCP_server.py, RUDP_server.py) שיש להם גישה ל-DATA ותפקידם הוא להחזיר ללקוח תשובה לשאילתות על השחקנים.

בנוסף, יצרנו מחלקה שנקראת client.py אשר מממשת את צד הלקוח ותפקידה הוא לשאול את השרת (איזה שתבחר) שאילתות לגבי השחקנים ולהציג את התשובה על המסך.

על מנת שלא נכתוב קוד כפול של פונקציות למימוש הקשר בין השרת ללקוח, יצרנו מחלקה שנקראת functions.py אשר מחזיקה 4 מתודות (send_with_cc, increase_window, receive, checksum).

הוראות קימפול והרצה:

תחילה יש להתקין את הספרייה pygame:

windows/ MAC OS:

https://www.pygame.org/wiki/GettingStarted

Linux:

https://www.geeksforgeeks.org/install-pygame-in-linux

לאחר שהתקנו pygame, יש להריץ את השרתים: DNS, DHCP:

```
python DHCP.py
python DNS.py
```

אחר כך יש להריץ תחילה את אחד השרתים
(TCP_server.py, UDP_server.py) בטרמינל נפרד כך:

python TCP_server.py **או** python RUDP_server.py

לאחר ששלושת השרתים האלו רצים, יש להריץ את client.py
ולבחור ב- RUDP/TCP בהתאם לשרת אותו בחרתם להריץ.

הערות:

- ב- linux יש לכתוב python3 במקום python.
- ניתן להריץ את כל השרתים ב-PyCharm אך יש לשים לב שה- configuration מוגדר ל-python3 ולא מתחת.
- את הפרויקט עשינו על ווינדוס לכן על מנת לחוות חוויה מלאה של הפרויקט (סאונד וגופנים של הכתב), יש להריץ על ווינדוס.

:TCP_server.py

למחלקה זו יש גישה לספריות:
socket, threading, pickle, PL_player, query_object, DATA
על מנת לפתוח קשר עם כמה לקוחות ולשלוח את המידע הנחוץ.
תחילה הגדרנו כמה משתנים קבועים (Caps Lock) הגדרנו משתנה בשם LEN_HEADER_SIZE = 8 אשר יהווה את גודל ה-header שהוספנו להודעה, header זה יגיד לנו מהו גודל ההודעה. לאחר מכן הגדנו כי הפורמט שבו "נצפין"/"נפענח" את המידע הוא 'utf-8' ושגודל הצ'אנקים של המידע שנשלח יהיו 32 בתים. הגדרנו את ה-PORT ל-30015, וכדי לקבל את ה-IP של מחשב זה השתמשנו ב-

```
IP = socket.gethostbyname(socket.gethostname())
```

לאחר מכן יצרנו משתנה מסוג טאפל בשם ADDR מה-IP וה-PORT שיצרנו שהוא הכתובת יעד לסוקט ופתחנו TCP סוקט ליצירת קשר.

```
ADDR = (IP, PORT)
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # TCP socket
server.bind(ADDR)  # binding the address
```

למחלקה זו יש 3 מתודות:

- **handle_client(conn, addr)** – מתודה זו מקבלת 2 פרמטרים:
  1. conn – אובייקט מסוג סוקט של הלקוח בו מטפלים
  2. addr – הכתובת של אותו לקוח

  מתודה זו אחראית על מימוש קבלת השאילתה מהלקוח ושליחה של המידע המתאים ללקוח.
  המתודה מקבלת מידע מהלקוח ע"י פונקציית recv, ומפלטרת לפיו. אם ההודעה היא שאילתה, אז נקבל את התשובה לשאילתה עם המתודה filter_by_queries(queries), ונשלח את התשובה בחזרה ללקוח עם פונקציית send.
  נעיר על כך שנשתמש בספרייה pickle ובייחוד במתודות pickle.loads ו-pickle.dumps על מנת להמיר את המידע שנקבל מבתים למחרוזות וההפך כיוון שהמידע שנשלח הוא בבתים.
  המתודה ממשיכה לעבוד בלולאה אינסופית ב-thread שמותאם ללקוח עד אשר נקבל מהלקוח הודעת יציאה.

- **start()** – מתודה זו יוצרת thread חדש לכל לקוח שמנסה להתחבר עם השרת ומקצה ל-thread את handle_client עם הסוקט והכתובת של הלקוח כמשימה.

- **filter_by_queries(queries)** – מתודה זו מקבלת פרמטר בשם queries שהוא רשימה של אובייקטים מסוג query_obj ממחלקת query_object.
לאחר קבלת השאילתות כפרמטר, המתודה פונה למחלקת query_object, מבצעת את השאילתות שיש ברשימה ומחזירה רשימה של PL_player אשר מהווה תשובה לשאילתות שהתקבלו כפרמטר.

:RUDP_server.py

למחלקה זו יש גישה לאותם ספריות כמו TCP_server.py ואותם משתנים קבועים חוץ מ- LEN_HEADER_SIZE.
בנוסף למשתנים אלו, הוספנו למחלקה הזו את המשתנים הבאים:
PORT_CHANGE – משתנה אשר בעזרתו ניצור פורט חדש לכל לקוח.
clients[] – רשימה של לקוחות.
change_port_lock, client_lock – שני מנעולים שנועדו כדי לעדכן את שני המשתנים לעיל באופן אסינכרוני.
באותו אופן של TCP, נפתח סוקט UDP ונקשר אותו.

```
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  # UDP socket
server.bind(ADDR)  # binding the address
```

למחלקה זו קיימים 6 מתודות:

- **filter_by_queries(queries)** – אותה פונקציה כמו של השרת TCP.
- **get_change_port()** – מתודה זו מחזירה את מספר הסוקטים שפתחנו עד כה על מנת ליצור פורט חדש בהתאם ללקוח החדש.
- **add_client(client)** – מתודה זו מקבלת כתובת ומוסיפה אותה לרשימת הלקוחות (clients[]).
- **remove_client(client)** – מתודה זו מקבלת כתובת ומוחקת אותה מרשימת הלקוחות (clients[]).
- **start()** – תחילה מתודה זו מקבלת מהלקוח את גודל הצ'אנק לשליחה ו
  יוצרת thread חדש לכל לקוח שמנסה להתחבר עם השרת ומקצה ל-thread את handle_client עם הסוקט והכתובת של הלקוח כמשימה, ומוסיפה את הכתובת של לקוח זה לרשימת הלקוחות (clients[]).

handle_client(ip, port, chunk) – מתודה זו מקבלת ip, port ו-chunk. ה- ip וה- port של הלקוח ליצירת טאפל בשם addr אשר מהווה את כתובת הלקוח, ו-chunk אשר יהווה את גודל הסגמנטים שיתקבלו וישלחו למימוש Flow Control.
תחילה ניצור 5 משתנים: סוקט UDP, פורט ייחודי ללקוח, כתובת השרת ,דגל השווה ל-1 ומשתנה שיאחסן את המידע של ההודעה.
לאחר מכן נקשר את סוקט ה-UDP לכתובת השרת, נדאג לשחרר את הפורט הנוכחי בעת סגירת הקשר ע"י setsockopt ונדאג שפונקציית ()socket.recvfrom לא תהיה פונקציה חוסמת ע"י שימוש ב- socket.setblocking(False) .
לאחר מכן נסיים את פתיחת הקשר.
אחר כך נקבל מהלקוח שאילתות / הודעת יציאה ונשלח לו תשובה בהתאם.

DNS_server.py:

מחלקה זו עובדת בדיוק כמו מחלקת TCP_server.py רק שבמקום לקבל שאילתות ולהחזיר תשובה לפי השאילתה, היא מקבלת מילת מפתח לחיפוש כתובת של השרת איתו הלקוח מתקשר,
מחפשת את הכתובת במשתנה מסוג מילון שיצרנו שממפה כתובות ושולחת ללקוח את הכתובת של שרת זה.

הערות:

- הקשר מתבצע באמצעות סוקט TCP
- בשימוש שלנו בפרויקט זה, יש ל-DNS רק כתובת אחת אך הוא יכול להחזיק **יותר** מכתובת אחת ולהחזיר את הכתובת הרצויה בהתאם לבקשת הלקוח

:DHCP_server.py

תפקידה של המחלקה DHCP_server.py הוא לקשר את הלקוח לרשת ולשלוח לו את הכתובת ברשת של הלקוח ושל שרת ה-DNS.

בניגוד ל-DHCP אמיתי, המחלקה שלנו עובדת על פרוטוקול TCP והלקוח יודע את ה-IP של ה-DHCP שלנו ולא מחפש אותו ב- broadcast.

למחלקה זו יש גישה לספריות:
socket, threading, pickle
על מנת לפתוח קשר עם כמה לקוחות ולשלוח את המידע הנחוץ.

למחלקה זו יש 2 מתודות: handle_client(conn,addr) ו- ()start.

- **handle_client(conn,addr) -** מתודה זו מקבלת 2 פרמטרים:
  1. conn – אובייקט מסוג סוקט של הלקוח בו מטפלים
  2. addr – הכתובת של אותו לקוח

  מתודה זו מקבלת מהלקוח בקשה לקשר אותו לשרת DNS מסוים ושולחת ללקוח רשימה שמכילה את כתובת הלקוח וכתובת ה-DNS לחיבור.
  המתודה מקבלת מידע מהלקוח ע"י פונקציית recv, ומפלטרת לפיו. אם קיבלנו בקשה להתחבר, נשלח ללקוח רשימה שמכילה את כתובת הלקוח וכתובת ה-DNS לחיבור עם פונקציית send. נעיר על כך שנשתמש בספרייה pickle ובייחוד במתודות pickle.loads ו-pickle.dumps על מנת להמיר את המידע שנקבל מבתים למחרוזות וההפך כיוון שהמידע שנשלח הוא בבתים.
  המתודה ממשיכה לעבוד בלולאה אינסופית ב-thread שמותאם ללקוח עד אשר נקבל מהלקוח הודעת יציאה.
  כאשר נקבל הודעת יציאה אז נסגור את conn.

- **()start –** מתודה זו יוצרת thread חדש לכל לקוח שמנסה להתחבר עם השרת ומקצה ל-thread את handle_client עם הסוקט והכתובת של הלקוח כמשימה.

functions.py:

מחלקה זו מספקת לנו את המתודות ההכרחיות למימוש RUDP כגון:
send_with_cc, increase_window, receive, checksum

הסבר על המתודות הנ"ל:

- **checksum(x)** – מחזירה ערך עגול של מספר הביטים חלקי 8
- **increase_window(window_size)** - בודקת מהו גודל החלון והאם להגדיל אותו בקצב מהיר (פי 2) או בקצב איטי (פלוס 1).
- **send_with_cc(curr_sock, addr, msg, chunk = CHUNK)** – מתודה זו מקבלת 4 פרמטרים:
  1. **curr_sock** – UDP סוקט
  2. **addr** – כתובת לשליחת ההודעה
  3. **msg** – הודעה לשליחה
  4. **chunk** – אם לא מוכנס כפרמטר הוא מוגדר למשתנה הקבוע CHUNK=32 אשר מהווה את גודל הסגמנטים שישלחו

תחילה נחלק את ההודעה לשליחה לסגמנטים בגודל chunk ונוסיף לכל סגמנט 3 האדרים: גודל ההודעה, אינדקס של הסגמנט ו-checksum.
לכל סגמנט (לפי אינדקס) יש מיקום ברשימה state וברשימה timestamps וכך נדע אם הסגמנט הגיע או לו ואם קפץ לו timeout וצריך לשלוח אותו שוב. בנוסף יהיה לנו מערך dup_ack אשר יקבל את האינדקס האחרון שקיבל ack וכמה פעמים ברצף יתקבל ה-ack. רשימה זו תדאג לבעיות Latency.

נתחיל מחלון שליחה בגודל 1. נשלח את ההודעה וכאשר נקבל ack על ההודעה, נשנה ערכים בהתאם ב-dup_ack ואם כמות ה-ack הרצופים גדולה מ-10 נקבל כי יש בעיית latency, נקטין את חלון השליחה פי 2 ונשלח שוב את ההודעה לפי האינדקס שלא קיבל הודעה. אם לא הייתה בעיית latency אז נגדיל את חלון השליחה בעזרת המתודה increase_window.

לכל סגמנט שנשלח ולא קיבל ack, נבדוק האם עבר 5 שניות מהרגע שהוא נשלח, אם זה תכף לאחד+ מהם אז נקטין את חלון השליחה בחזרה ל-1.

אם כל האיברים ברשימה state הם 2 אז התקבלו כל הסגמנטים ונצא.

- **receive(curr_sock, addr, chunk = CHUNK) -> list –** מתודה זו מקבלת 3 פרמטרים:
  1. **curr_sock –** UDP סוקט
  2. **addr –** כתובת לשליחת ack'ים
  3. **chunk -** אם לא מוכנס כפרמטר הוא מוגדר למשתנה הקבוע CHUNK=32 אשר מהווה את גודל הסגמנטים שישלחו

  תחילה נגדיר את גודל ההודעה לפי הגודל של chunk וגודל ההאדרים ונגדיר מערך chunks שאליו נכניס את ההודעות שנקבל ושנרכיב אחר כך להודעה שלמה ומערך indexes שיאשר לנו איזה סגמנטים התקבלו כבר על מנת שלא נכניס כפילויות למערך chunks נגדיר בנוסף משתנה בשם get_size אשר יהווה את גודל הסגמנט לקבלה (לפי גודל ה-chunk וההאדרים), ונגדיר עוד 3 משתני גודל לעזרה בספירת כמות הבתים שקיבלנו (bytes_received), מה רצף הסגמנטים שהתקבלו (max_seq_index) וסך גודל ההודעה (msg_len).

  תחילה בלולאה אינסופית, נקבל הודעה מהסוקט בגודל get_size, אם קיבלנו יותר מידע מגודל ההאדרים אז נבדוק האם ה-data שקיבלנו עובר בדיקת checksum והמידע נכון, במידה וזאת הפעם הראשונה שהתקבלה ההודעה (בעזרת המערך indexes), נכניס את ההודעה שקיבלנו למערך chunks ונעדכן את כמות הבתים שקיבלנו עד כה לפי גודל ההודעה. לאחר מכן, נעדכן בהתאם את רצף הסגמנטים שיתקבלו ונשלח בחזרה ack ל-addr.
  במידה וקיבלנו את כל הבתים לפי (msg_len ו-bytes_received), נמיין את המערך chunks ונכניס את ה-data של הסגמנטים לרשימה בשם full_msg ונחזיר את full_msg.

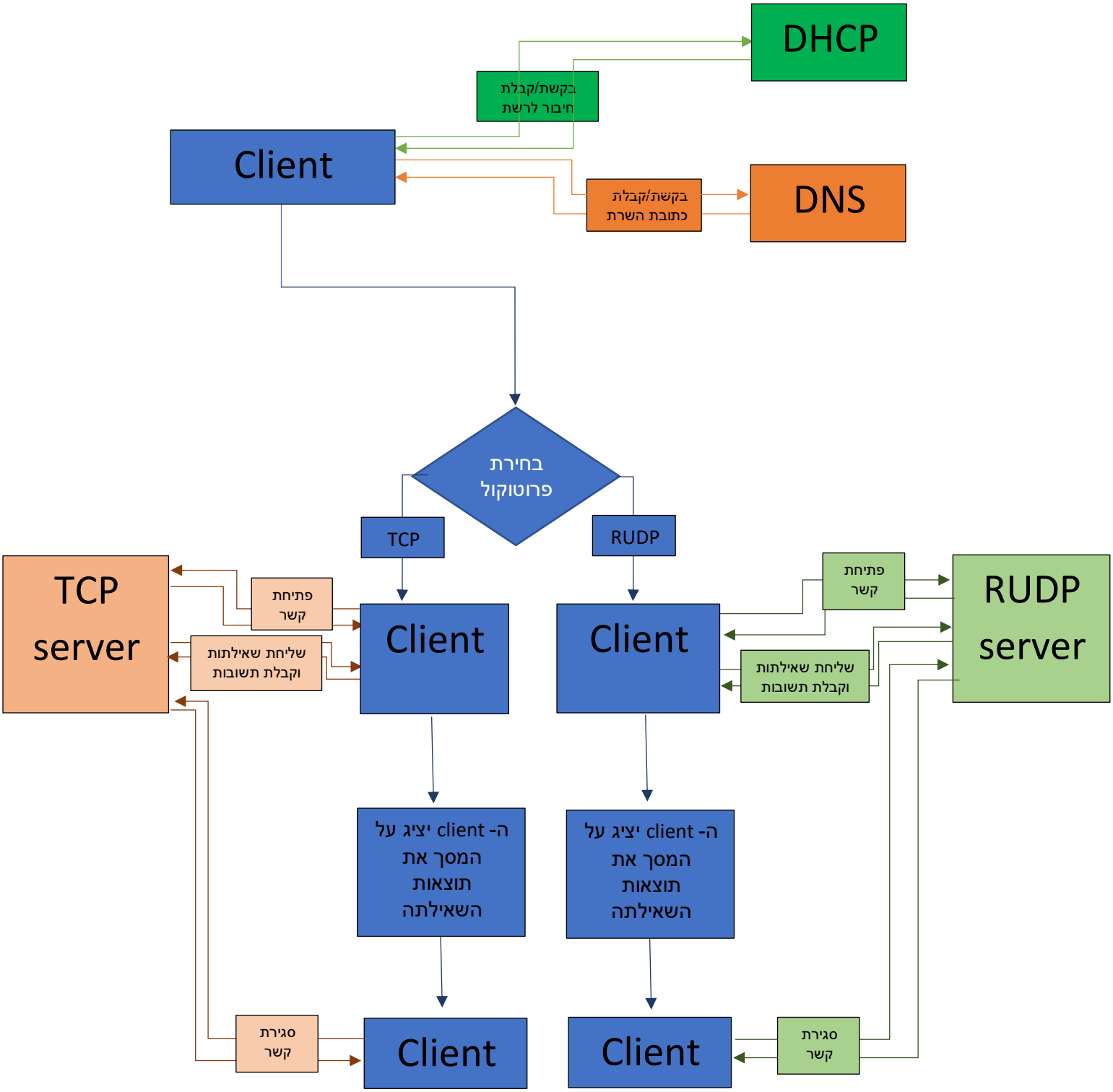
client.py:

מחלקה זו מממשת את צד הלקוח.
הלקוח יתחבר לשרת ה-DHCP אשר ייתן לו את הכתובת ברשת של הלקוח ושל שרת ה-DNS, לאחר מכן הלקוח יתחבר לשרת ה-DNS אשר יביא לו את הכתובת של שרת ה-TCP/RUDP בהתאם לבחירה של הלקוח באיזה פרוטוקול הוא רוצה לעבוד.
כעת הלקוח יתקשר עם שרת ה-TCP (נניח כרגע לשם הנוחות כי בחר TCP, באותה מידה הלקוח היה יכול לבחור ב-RUDP).
הלקוח ישלח שאילתות לשרת ויקבל תשובה בצורה של רשימה של PL_player שאותה יציג על המסך.
על מנת לאפשר את הקשר הזה ושבאמת הלקוח יוכל לתקשר עם כל השרתים הנ"ל בצורה נכונה, יצרנו את המתודות הבאות:
dhcp_connection, dns, connect_to_socket, send_queries, tcp_send, udp_send.

- **dhcp_connection()** – במתודה זו נתחבר לשרת ה-DHCP עם פרוטוקול TCP רגיל כאשר ידוע לנו כתובת שרת ה-DHCP ונשלח לו בקשת התחברות לרשת (הודעת התחברות) נקבל תשובה מהשרת בצורת רשימה בעלת 2 כתובות (טאפלים) ברשת (במקרה שלנו זה לא באמת ברשת אלא בתוך המחשב שלנו כלומר פורט חדש וה-IP של המחשב) כתובת באינדקס 0 של הלקוח וכתובת באינדקס 1 של שרת ה-DNS. נשמור את הכתובות האלה במשתנים המתאימים להם, נשלח Fin ack ונסגור את הסוקט.
- **dns()** – עובד בדיוק באותה צורה של dhcp_connection() רק שהלקוח לא מקבל מהשרת רשימה של 2 כתובות אלא כתובת אחת של השרת TCP/RUDP אליו הוא מעוניין להתחבר.
- **connect_to_socket()** – מתודה זו יוצרת סוקט ללקוח לקבלה ושליחה של הודעות לשרת לפי סוג הפרוטוקול.
  1. פרוטוקול TCP – יוצרת סוקט TCP רגיל.
  2. פרוטוקול UDP – יוצרת סוקט UDP עם הכתובת שה-DHCP הביא ללקוח ומבצעת לחיצת יד משולשת עם שרת ה-RUDP.

- **send_queries(queries_to_send: list) -** אם הפרוטוקול הוא TCP, מתודה זו שולחת את queries_to_send עם tcp_send.
  אם הפרוטוקול הוא UDP, מתודה זו שולחת את queries_to_send עם udp_send.
- **tcp_send(queries_l: list) –** שולחת את queries_l עם הסוקט TCP ש- connect_to_socket() יצר, מקבלת בחזרה תשובה ומציגה את התשובה על מסך ה- pygame.
  (תקשורת TCP רגילה).
- **udp_send(queries_l: list) –** שולחת את queries_l לסוקט UDP ש- connect_to_socket() יצר עם המתודה functions.send_with_cc(...) של המחלקה functions.
  לאחר מכן נקבל תשובה לשאילתות מהשרת דרך המתודה functions.receive(...) של המחלקה functions, ומציגה את התשובה על מסך ה- pygame.

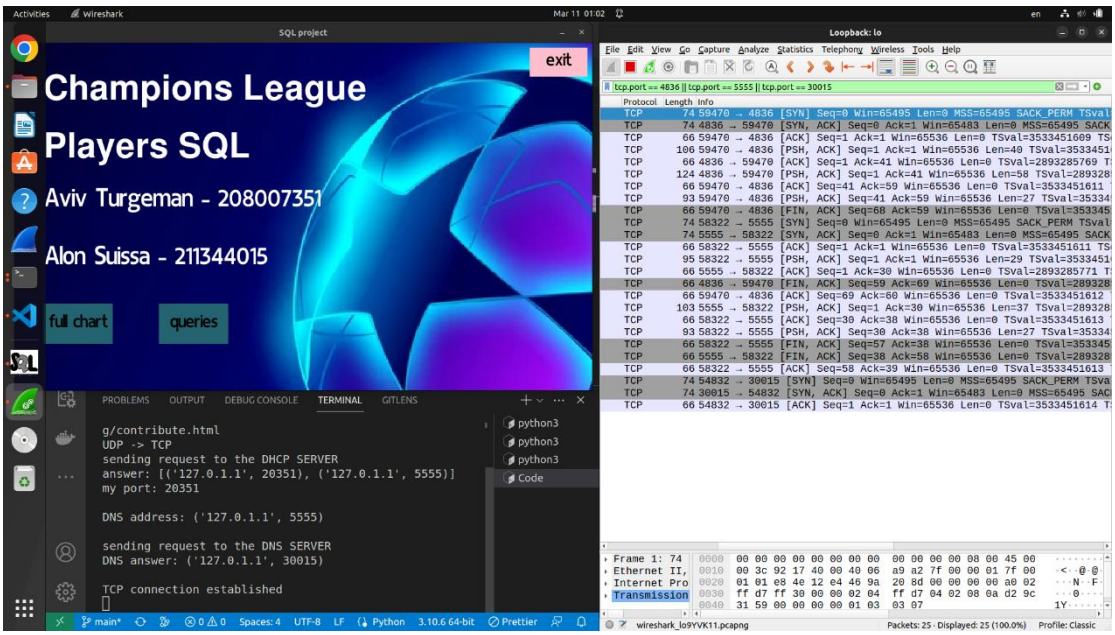
דיאגרמת מצבים:

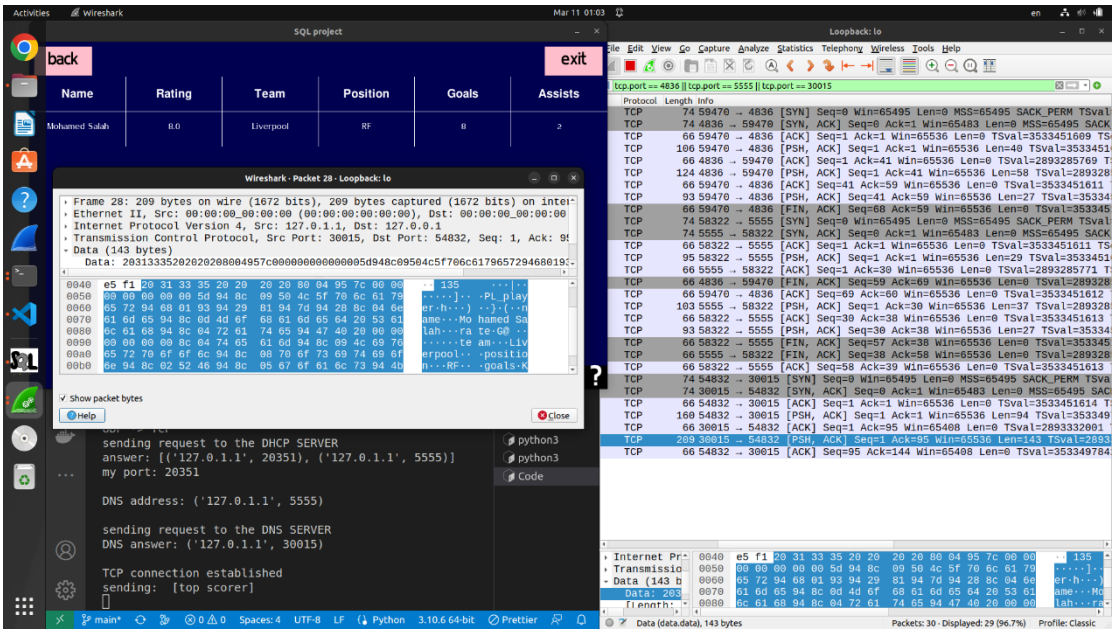


הקלטות wireshark:

TCP:

- **0% איבוד -**

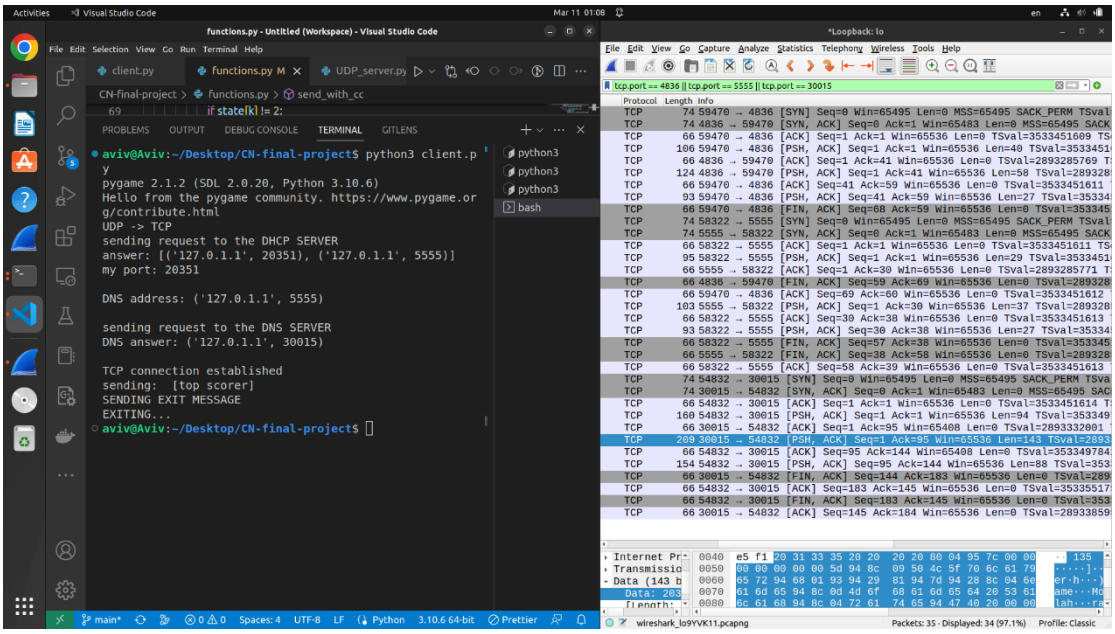




ניתן לראות בתמונה הראשונה כי תחילה נתחבר מהלקוח לשרת ה-DHCP ותהיה לחיצת יד משולשת ואחרי ששרת ה-DHCP יחזיר ללקוח תשובה, הם יסגרו בניהם את הקשר עם ([FIN, ACK], [FIN, ACK], [ACK]). כנ"ל לגבי ה-DNS.

לאחר מכן, ניתן לראות בתמונה השנייה שאחרי שנשלח שאילתה לשרת, נקבל תשובה משרת ה-TCP ושהתוצאה שהתקבלה משרת ה-TCP אכן התשובה שמוצגת על מסך ה-GUI.

לבסוף,



נשלח מהלקוח לשרת הודעת יציאה ושרת ה-TCP והלקוח יסגרו ביניהם את הקשר עם ([FIN, ACK], [FIN, ACK], [ACK]).

- **<u>10% איבוד -</u>**

כמובן שכל שלבי הקשר פה יהיו זהים ל-0% אך כעת יהיה לנו איבודי פאקטות.

<u>שידור חוזר:</u>

.1	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 38070 → 4836 [SYN] Seq=0
.1	TCP	74	4836 → 38070 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM
.1	TCP	74	[TCP Retransmission] 4836 → 38070 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=

ניתן לראות שהסגמנט נשלח שוב אחרי ששליחת הסגמנט לא הגיע לשרת.

<u>:TCP Dup ACK, TCP Out-Of-Order</u>

.1	TCP	66	[TCP Previous segment not captured] 41718 → 5555 [FIN, ACK] Seq=57 Ack=
.1	TCP	78	[TCP Dup ACK 17#1] 5555 → 41718 [ACK] Seq=38 Ack=30 Win=65536 Len=0 TSv
.1	TCP	93	[TCP Out-Of-Order] 41718 → 5555 [PSH, ACK] Seq=30 Ack=38 Win=65536 Len=
.1	TCP	66	5555 → 41718 [ACK] Seq=38 Ack=58 Win=65536 Len=0 TSval=2897126071 TSecr
.1	TCP	74	36738 → 30015 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=353
.1	TCP	74	30015 → 36738 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PER
.1	TCP	66	36738 → 30015 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3537291912 TSecr=
.1	TCP	66	5555 → 41718 [FIN, ACK] Seq=38 Ack=58 Win=65536 Len=0 TSval=2897126292
.1	TCP	66	41718 → 5555 [ACK] Seq=58 Ack=39 Win=65536 Len=0 TSval=3537292132 TSecr

השרת לא קיבל סגמנט ומבקש אותו שוב ולאחר שנשלח שוב סגמנט, השרת מתריע על כך שהסגמנטים שהתקבלו הם לא בסדר הנכון, כלומר התקבלו למשל 3 סגמנטים 1,3,2 אבל סגמנט מס' 2 התקבל לאחר 3.

- **<u>15% איבוד -</u>**
כמובן שכל שלבי הקשר פה יהיו זהים ל-0% אך כעת יהיה לנו איבודי פאקטות.

<u>שידור חוזר:</u>

```
TCP        66 [TCP Retransmission] 5555 → 33774 [FIN, ACK]
```

אותו הסבר כמו ב-10%.

<u>TCP Dup ACK, TCP Out-Of-Order:</u>

```
TCP        66 [TCP Previous segment not captured] 57280 → 4836 [FIN, ACK] Seq=68 Ac
TCP        78 [TCP Dup ACK 5#1] 4836 → 57280 [ACK] Seq=59 Ack=41 Win=65536 Len=0 TS
TCP        93 [TCP Out-Of-Order] 57280 → 4836 [PSH, ACK] Seq=41 Ack=59 Win=65536 Le
```

אותו הסבר כמו ב-10%.

:RUDP

1	127.0.0.1	0.000000000	127.0.1.1	TCP	74	60514 → 4836
2	127.0.1.1	0.000034564	127.0.0.1	TCP	74	4836 → 60514
3	127.0.0.1	0.000056505	127.0.1.1	TCP	66	60514 → 4836
4	127.0.0.1	0.001934762	127.0.1.1	TCP	106	60514 → 4836
5	127.0.1.1	0.001947876	127.0.0.1	TCP	66	4836 → 60514
6	127.0.1.1	0.006068652	127.0.0.1	TCP	124	4836 → 60514
7	127.0.0.1	0.006180574	127.0.1.1	TCP	66	60514 → 4836
8	127.0.0.1	0.010639246	127.0.1.1	TCP	93	60514 → 4836
9	127.0.0.1	0.010686723	127.0.1.1	TCP	66	60514 → 4836
10	127.0.0.1	0.010764866	127.0.1.1	TCP	74	50000 → 5555
11	127.0.1.1	0.010776657	127.0.0.1	TCP	74	5555 → 50000
12	127.0.0.1	0.010788225	127.0.1.1	TCP	66	50000 → 5555
13	127.0.0.1	0.013176753	127.0.1.1	TCP	95	50000 → 5555
14	127.0.1.1	0.013184936	127.0.0.1	TCP	66	5555 → 50000
15	127.0.1.1	0.014947776	127.0.0.1	TCP	66	4836 → 60514
16	127.0.0.1	0.014974824	127.0.1.1	TCP	66	60514 → 4836
17	127.0.1.1	0.018533175	127.0.0.1	TCP	103	5555 → 50000
18	127.0.0.1	0.018544264	127.0.1.1	TCP	66	50000 → 5555
19	127.0.0.1	0.018730998	127.0.1.1	TCP	93	50000 → 5555
20	127.0.0.1	0.018925202	127.0.1.1	TCP	66	50000 → 5555
21	127.0.1.1	0.019170446	127.0.0.1	TCP	66	5555 → 50000
22	127.0.0.1	0.019190740	127.0.1.1	TCP	66	50000 → 5555

תחילה נעיר שהחיבור של הלקוח לסרברים DHCP ו-DNS הוא אותו חיבור כמו ב-TCP והאיבודים יהיו אותם איבודים.

0.221069330	127.0.1.1	UDP	60	30020 → 20351 Len=18
0.221353234	127.0.1.1	UDP	60	20351 → 30020 Len=18
8.453351264	127.0.1.1	UDP	98	20351 → 30020 Len=56
8.453761746	127.0.1.1	UDP	50	30020 → 20351 Len=8
8.453808390	127.0.1.1	UDP	98	20351 → 30020 Len=56
8.453940844	127.0.1.1	UDP	88	20351 → 30020 Len=46
8.454220195	127.0.1.1	UDP	50	30020 → 20351 Len=8
8.454260502	127.0.1.1	UDP	50	30020 → 20351 Len=8
13.4593544…	127.0.1.1	UDP	98	30020 → 20351 Len=56
13.4606434…	127.0.1.1	UDP	50	20351 → 30020 Len=8
13.4607065…	127.0.1.1	UDP	98	30020 → 20351 Len=56
13.4608653…	127.0.1.1	UDP	98	30020 → 20351 Len=56
13.4613038…	127.0.1.1	UDP	50	20351 → 30020 Len=8
13.4613345…	127.0.1.1	UDP	98	30020 → 20351 Len=56
13.4615274…	127.0.1.1	UDP	50	20351 → 30020 Len=8
18.4619920…	127.0.1.1	UDP	98	30020 → 20351 Len=56
18.4628774…	127.0.1.1	UDP	50	20351 → 30020 Len=8
28.4632908…	127.0.1.1	UDP	73	30020 → 20351 Len=31
28.4640935…	127.0.1.1	UDP	50	20351 → 30020 Len=8
31.7327768…	127.0.1.1	UDP	98	20351 → 30020 Len=56
31.7331928…	127.0.1.1	UDP	50	30020 → 20351 Len=8
31.7335722…	127.0.1.1	UDP	98	20351 → 30020 Len=56
31.7337526…	127.0.1.1	UDP	50	30020 → 20351 Len=8
31.7337806…	127.0.1.1	UDP	82	20351 → 30020 Len=40
31.7338210…	127.0.1.1	UDP	50	30020 → 20351 Len=8

כמו שניתן לראות, שתי הפאקטות הראשונות הן פאקטות ack ואחר כך, הלקוח ישלח query לשרת בסגמנטים באורך 56 לכל היותר והשרת יאשר שהוא קיבל את כל הסגמנטים וישלח ללקוח בחזרה את התשובה גם כן בסגמנטים באורך 56 לכל היותר והלקוח יאשר ברגע שקיבל את כל הסגמנטים וישלח לשרת query שמטרתו היא להגיד לשרת שהוא מעוניין לצאת והשרת ישלח לו אישור ויסגור גם את הסוקט שלו.

<u>שאלות נוספות:</u>

1) הבדלים עיקריים בין QUIC ל-TCP:

א. מנגנוני האבטחה של QUIC נמצאים בשכבת האפליקציה ושל TCP נמצאים בשכבת התעבורה.

ב. QUIC נשלח על גבי UDP

ג. TCP עושה לחיצת יד משולשת בכל מקרה ו-QUIC פותח קשר בצורה מהירה יותר.
אם היה חיבור קודם אז הוא עושה 0 RTT ואם לא היה חיבור קודם אז הוא עושה 1 RTT.

2) הבדלים עיקריים בין Vegas ל- Cubic:

א. Vegas קובע את רמת העומס ברשת בעיקר לפי העיכוב של הפאקטות ופחות לפי האיבודים של החבילות ברשת.

ב. וגאס מגיב מהר מאוד לעומס בשל העובדה שהוא מקטין את חלון הגודש מיד ברגע שהוא מזהה שיש גודש בניגוד ל-Cubic שמגיב לאט כאשר הגודש מצטבר על ידי הקטנת החלון לאט ובהדרגה.

3) BGP הוא פרוטוקול ניתוב שדרך הניתוב שלו היא ניהול טבלה של הרשתות המחוברות אליו והקשרים בניהן ועל פי זה נתב שעובד עם BGP יחליט לאן לנתב את החבילות המגיעות אליו בדרכן ליעדן (בנוסף יש פרמטרים של מנהל הרשת שמשפיעים).

בפרוטוקול זה כל נתב מקבל החלטה באופן עצמאי ולכן יכול להיות מושפע מגורמים רבים (כמו מהירות התקשורת עם שכנים שונים או מדיניות שנקבעת על ידי מנהל הנתב) לאחר מכן הנתב מעביר את המסלול המומלץ על פי החלטתו לשכנים שלו והם כאמור יקבלו החלטה בעצמם על המסלול המומלץ ויעבירו לשכנים שלהם...

פועל יוצא של האמור לעיל הוא שהפרוטוקול אינו בוחר על פי מסלול קצר כי, כאמור, ישנן שיקולים רבים המעורבים בהחלטה לאן להעביר את החבילה.

לעומת זאת אלגוריתם OSPF תמיד בוחר מסלול עי חישוב המסלול הזול ביותר בעזרת אלגוריתם דיקסטרה.

5) ARP ממפה כתובת MAC לכתובת IP ברשת מקומית

לעומת DNS אשר ממפה כתובת דומיין לכתוב IP ברשת מרוחקת

איבוד חבילות ו- Latency:

איבוד חבילות:

המערכת שלנו מתגברת על איבוד חבילות בכך שלכל חבילה הוספנו header'ים שיתנו לנו מידע על החבילה. הוספנו לכל חבילה header שיסמן את האינדקס של החבילה על מנת לדעת איזה חבילות הלקוח קיבל ואיזה לא. בנוסף, יצרנו רשימה בשם timestamps ככמות החבילות ומשתנה timeout כך שברגע שנשלח חבילה, נעדכן ברשימה timestamps באינדקס של החבילה ששלחנו את חתימת הזמן לאותו רגע.
בכל איטרציה בלולאת האינסופית, נבדוק לכל חותמת זמן ששונה מ-0, האם ההפרש בינה לזמן הנתון הוא גדול מה-timeout שהגדרנו. במידה וכן אז נקבל TIMEOUT, נחזיר את חלון השליחה שלנו ל-1, נסמן את החבילה הזאת כחבילה שלא נשלחה ברשימה אחרת שמעדכנת את סטטוס החבילה (2 = קיבלנו ack, 1 = שלחנו ולא קיבלנו עדיין ack, 0 = לא שלחנו).

latency:

יצרנו רשימה בשם dup_ack בעלת 2 איברים כך שאינדקס 0 אומר מהו האינדקס של החבילה שקיבלה dup ack ואינדקס 1 אומר כמה פעמים קיבלנו עליו dup ack ומשתנה dup_limit שיהווה את המקסימום dup ack שניתן לקבל. אם הגענו למצב ש- dup_ack[1] >= dup_limit (latency) אז נקטין את חלון השליחה בחצי.