***רשתות תקשורת פרויקט סיום – SQL***

***מגישים: אביב תורג'מן - 208007351***

***אלון סויסה – 211344015***

***תוכן עניינים:***

הסבר כללי על הפרויקט............................................. 3

הוראות קימפול והרצה............................................... 4

TCP\_server.py......................................................... 5

RUDP\_server.py...................................................... 7

DNS\_server.py........................................................ 9

DHCP\_server.py...................................................... 10

functions.py........................................................... 11

client.py................................................................. 13

דיאגרמת מצבים....................................................... 15

הקלטות wireshark עם/בלי איבוד פאקטות................... 16

שאלות נוספות..........................................................22

**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:**

RUDP  
server

TCP  
server

סגירת קשר

Client

שליחת שאילתות וקבלת תשובות

ה-client יציג על המסך את תוצאות השאילתה

Client

סגירת קשר

פתיחת קשר

Client

ה-client יציג על המסך את תוצאות השאילתה

Client

שליחת שאילתות וקבלת תשובות

פתיחת קשר

בחירת פרוטוקול

TCP

RUDP

בקשת/קבלת כתובת השרת

בקשת/קבלת חיבור לרשת

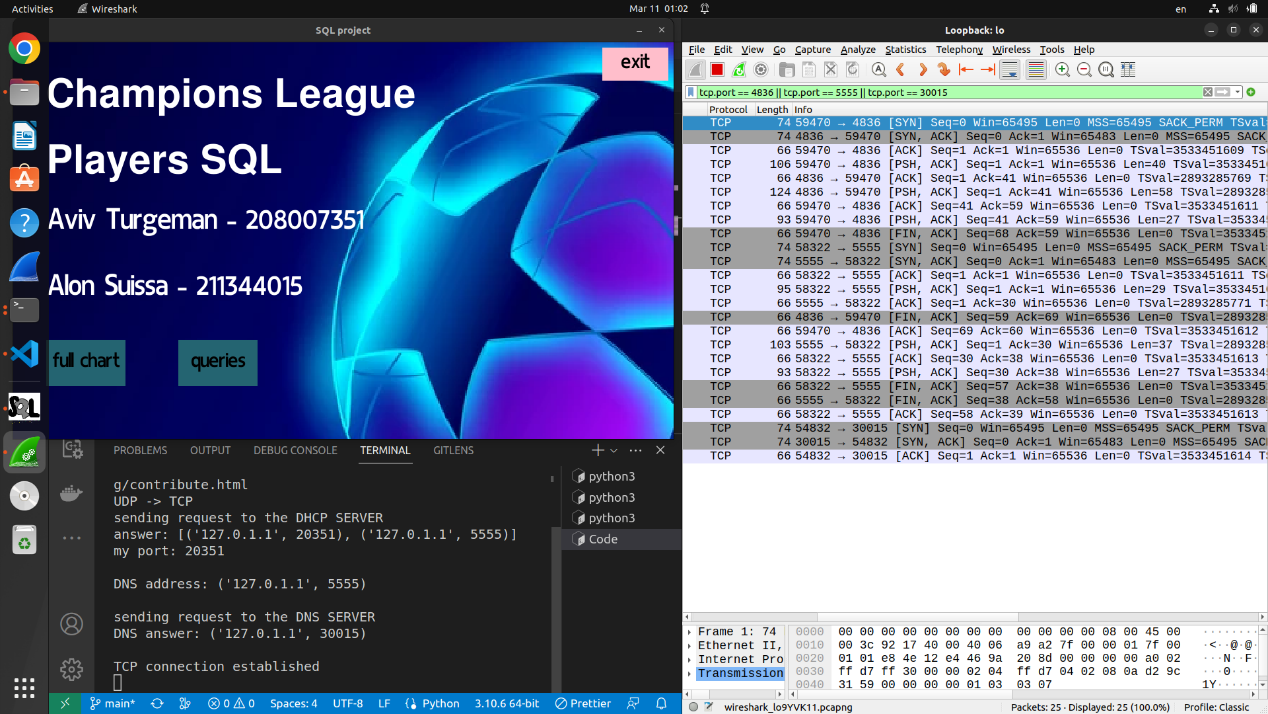
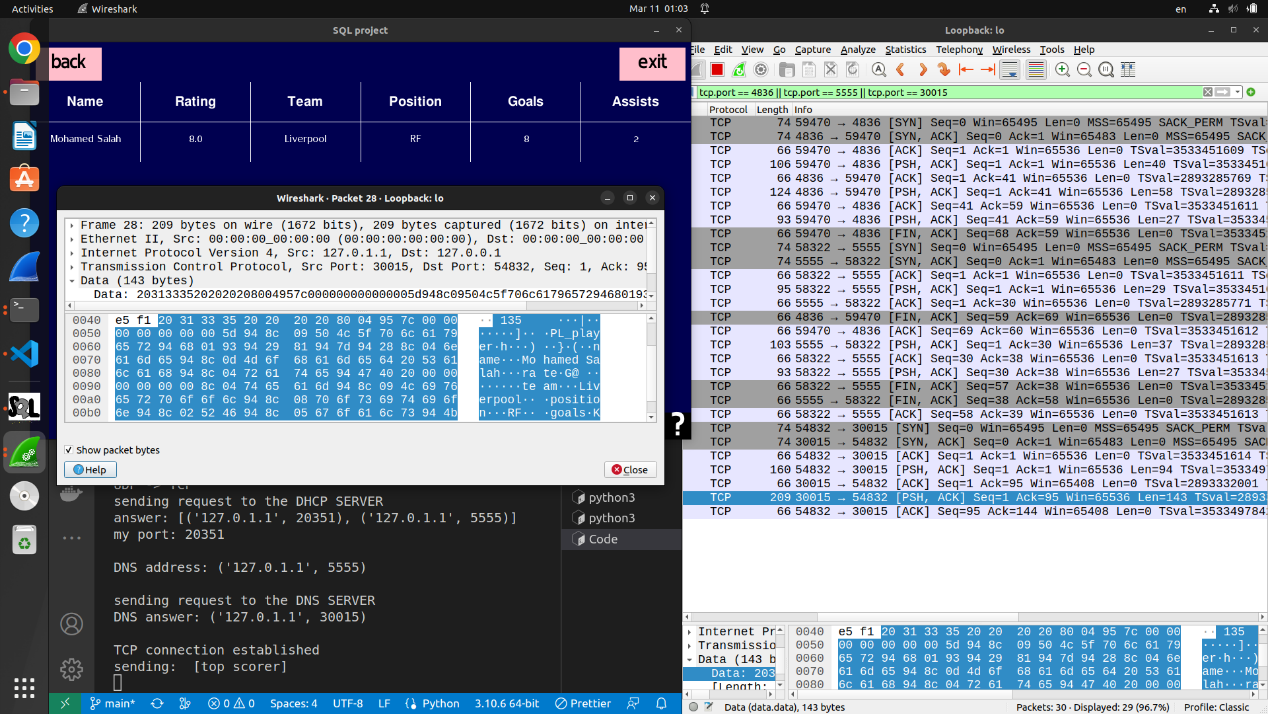
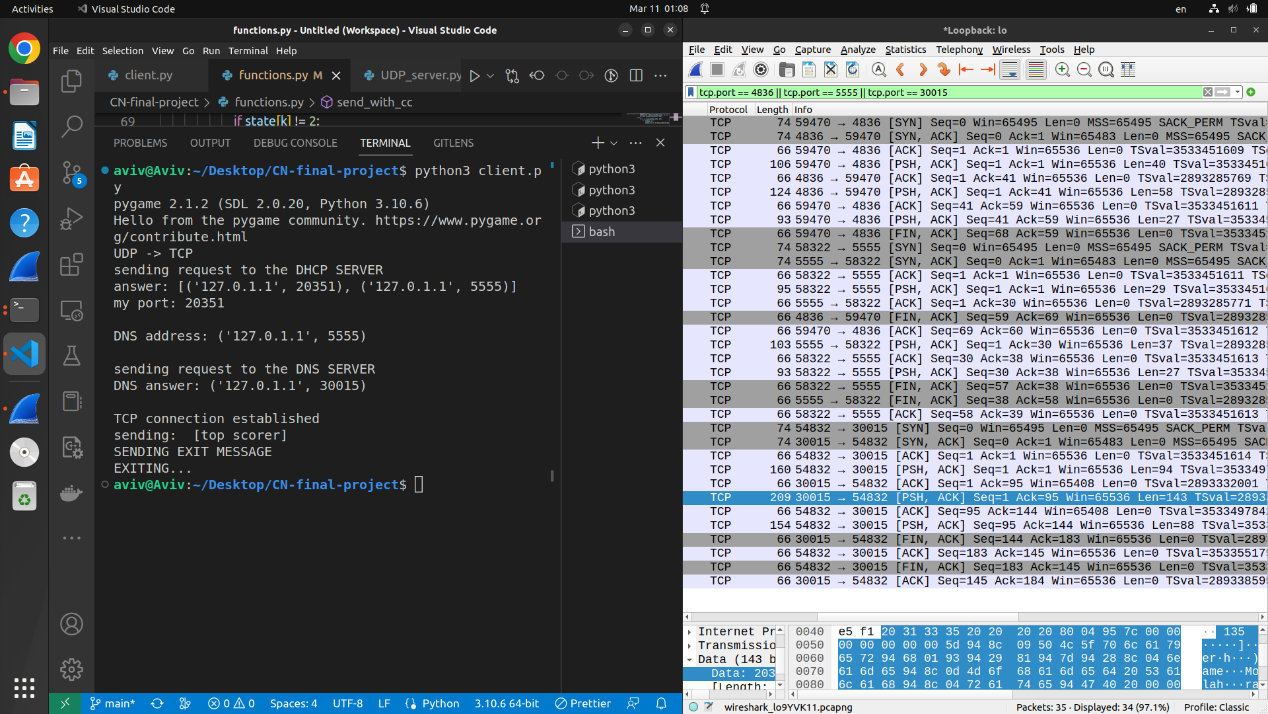
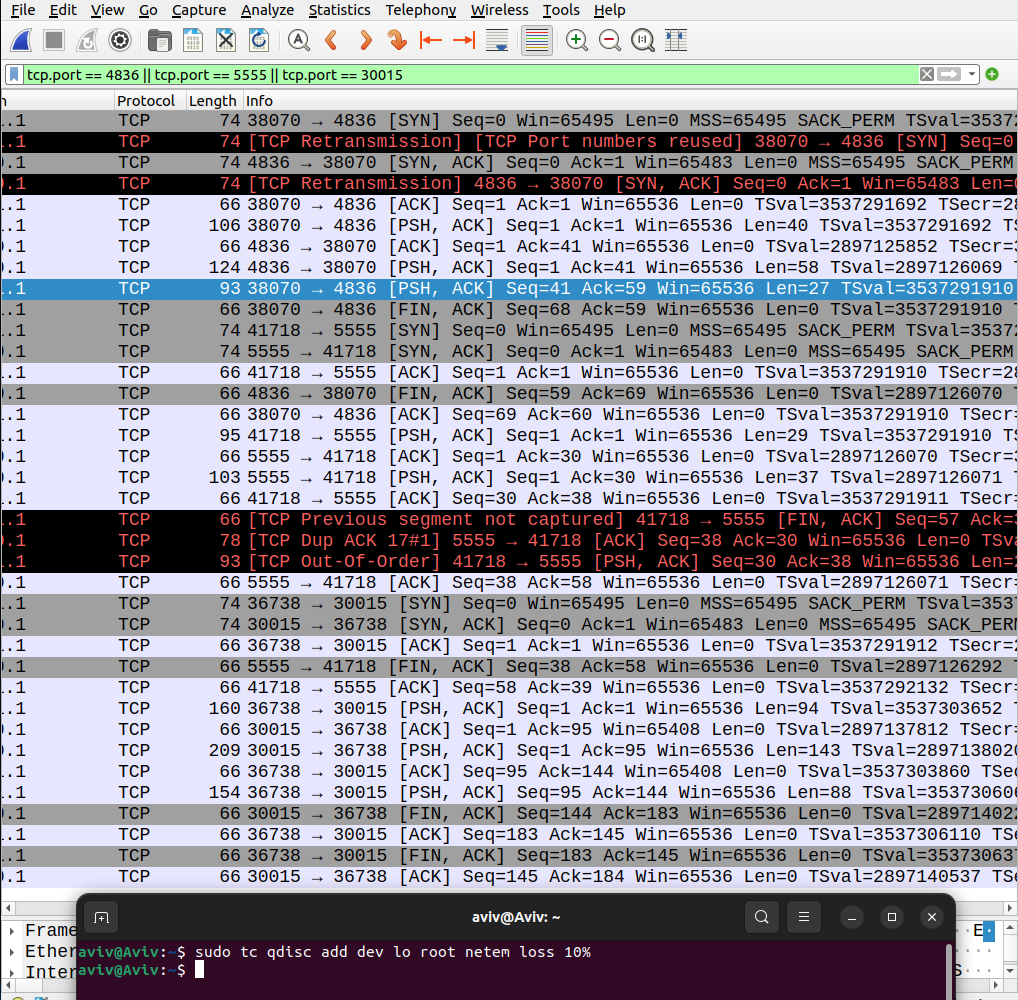
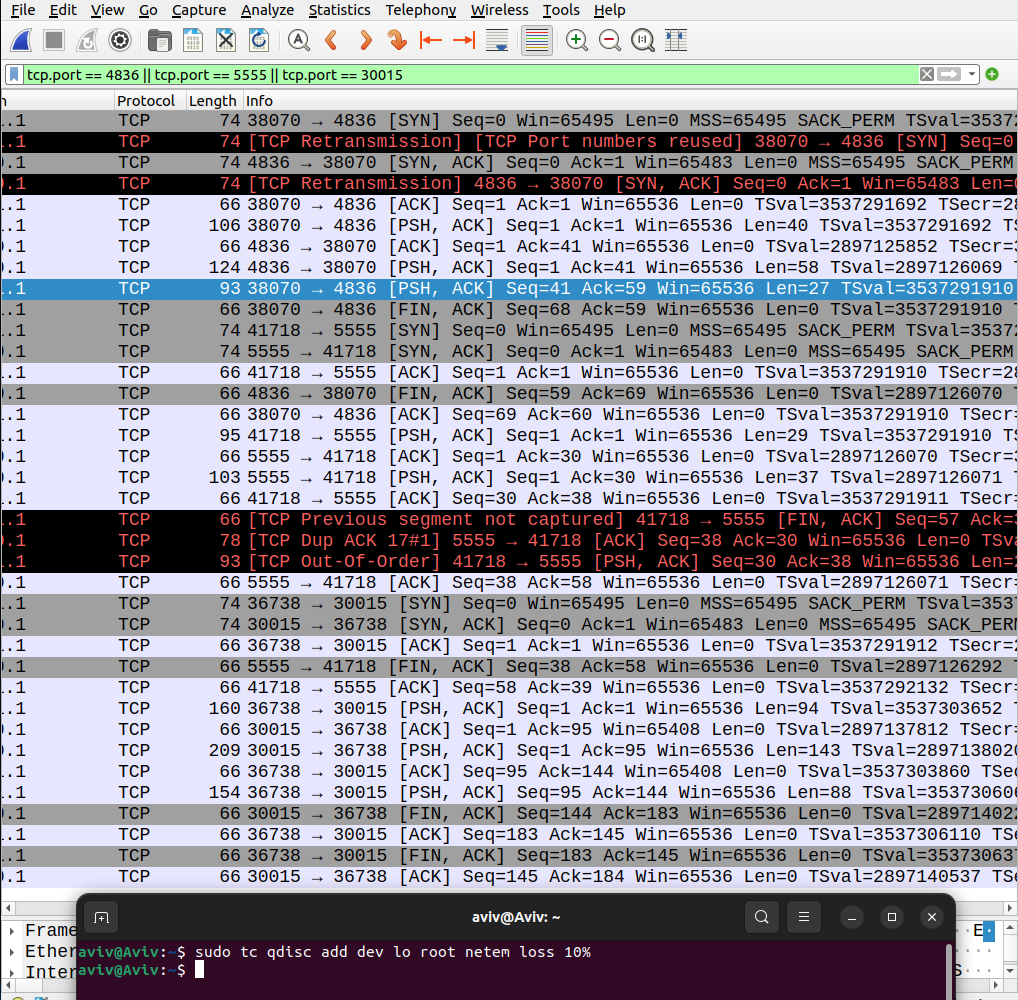
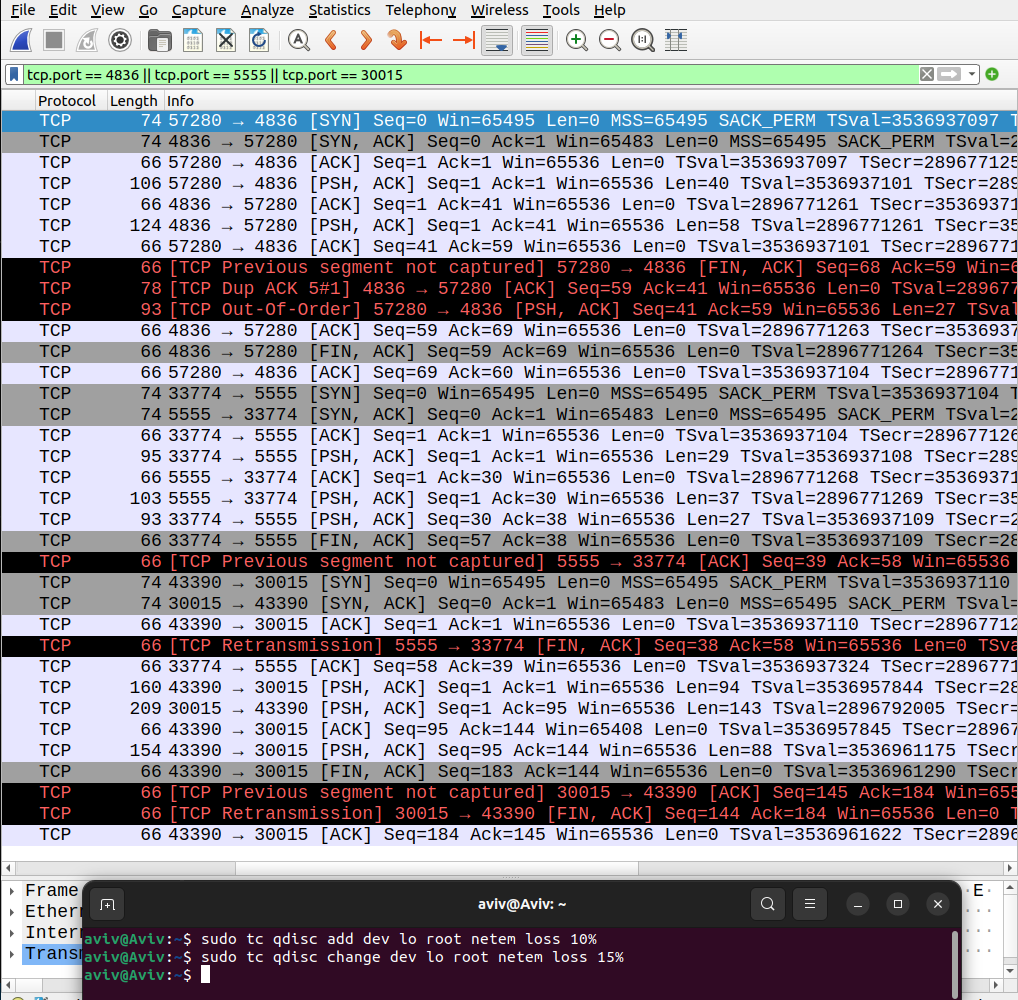
DHCP

DNS

Client

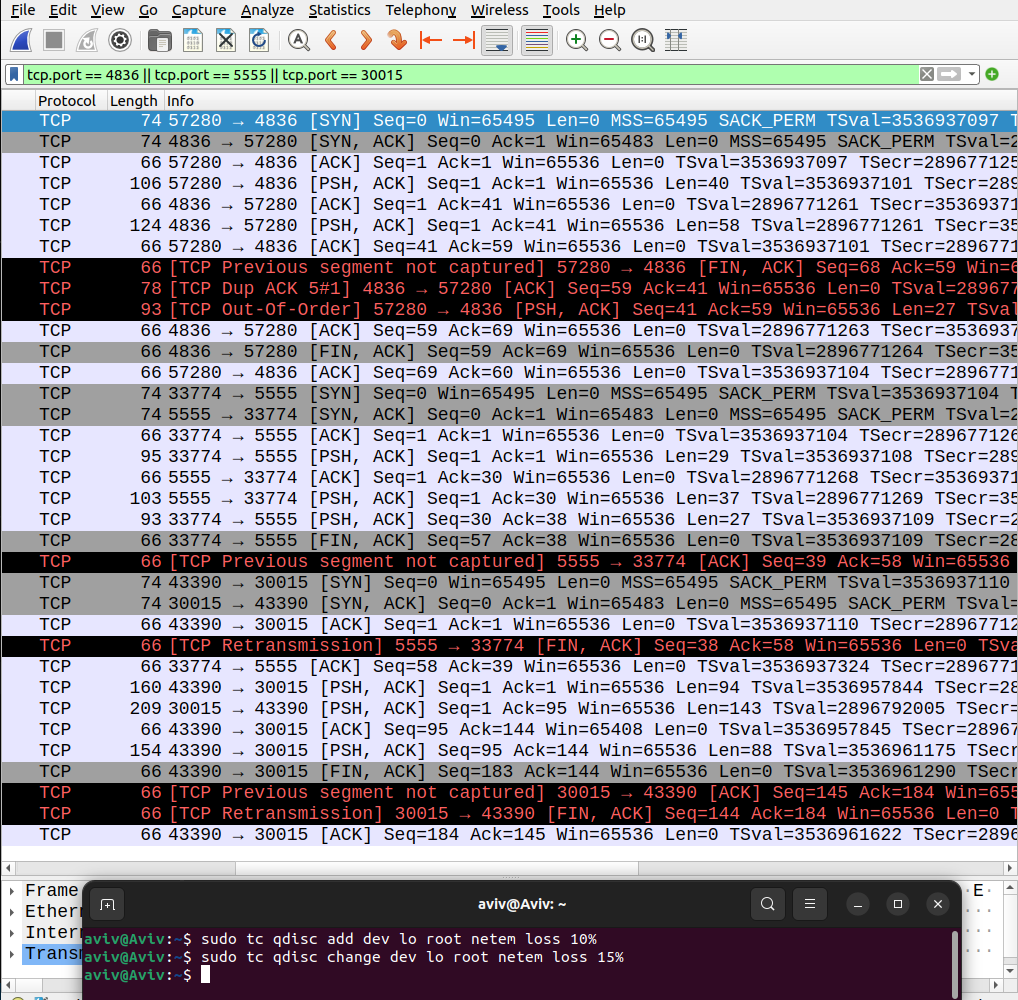
**הקלטות wireshark:**

**TCP:**

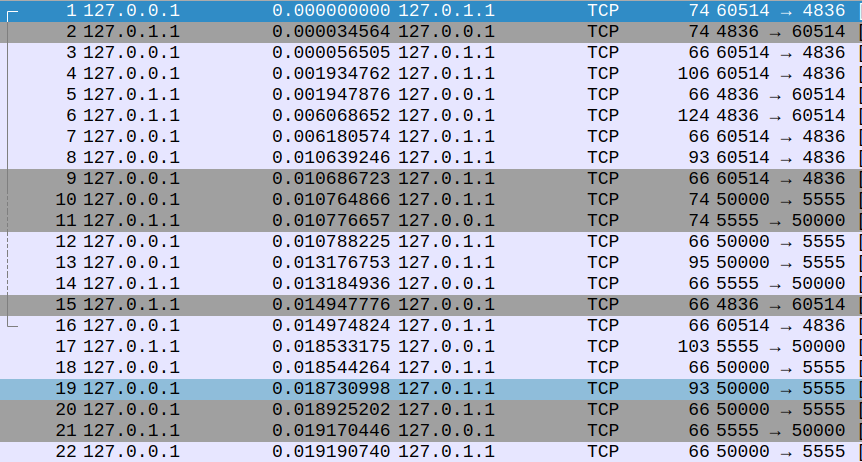
* **0% איבוד -**ניתן לראות בתמונה הראשונה כי תחילה נתחבר מהלקוח לשרת ה-DHCP ותהיה לחיצת יד משולשת ואחרי ששרת ה-DHCP יחזיר ללקוח תשובה, הם יסגרו בניהם את הקשר   
  עם ([FIN, ACK], [FIN, ACK], [ACK]). כנ"ל לגבי ה-DNS.  
    
    
  לאחר מכן, ניתן לראות בתמונה השנייה שאחרי שנשלח שאילתה לשרת, נקבל תשובה 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 בניהם את הקשר עם ([FIN, ACK], [FIN, ACK], [ACK]).
* **10% איבוד -**כמובן שכל שלבי הקשר פה יהיו זהים ל-0% אך כעת יהיה לנו איבודי פאקטות.  
    
  שידור חוזר:ניתן לראות שהסגמנט נשלח שוב אחרי ששליחת הסגמנט לא הגיע לשרת.  
  TCP Dup ACK, TCP Out-Of-Order: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.
* **15% איבוד -**כמובן שכל שלבי הקשר פה יהיו זהים ל-0% אך כעת יהיה לנו איבודי פאקטות.שידור חוזר:

אותו הסבר כמו ב-10%.

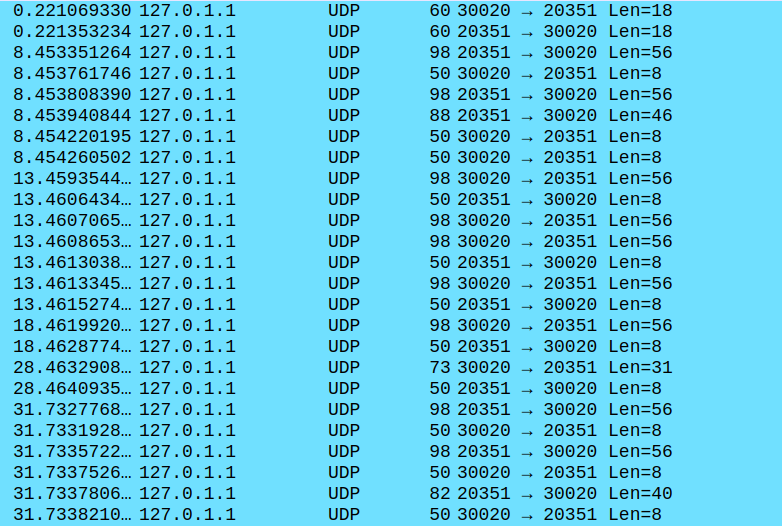
TCP Dup ACK, TCP Out-Of-Order:



אותו הסבר כמו ב-10%.

 **RUDP:**

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



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.

**שאלות נוספות:**

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.

**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.