

# **קורס תקשורת ומחשוב**

**פרויקט גמר**

**שם הסטודנטיות:**

**גל כהן 316138411**

**דנה צרצנקוב 208625293**

**שם המרצה:**

**ד"ר עמית דביר**

**ניתן להשיג את כל הפרויקט והקבצים תחת הקישור הבא:**

**<https://drive.google.com/drive/folders/1esSehNZYQA9XcDLIWE8WTLfM3RpcGfZT?usp=sharing>**

**תשפ"ב, סמסטר א'**

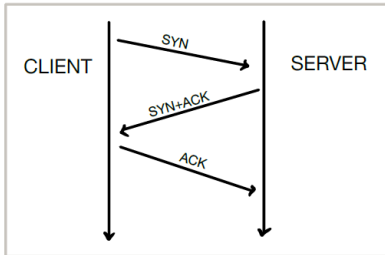
**הפקולטה למדעי הטבע, המחלקה למדעי המחשב**

## הקדמה

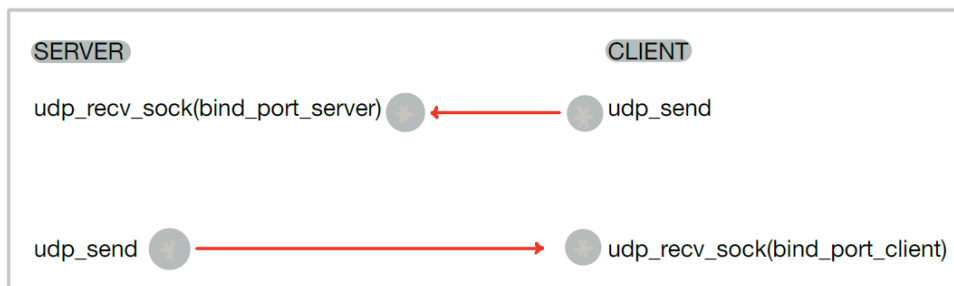
הפרויקט מחולק לשימוש בשתי פרוטוקולי תעבורה שונים - TCP ו-UDP.

**TCP** הוא פרוטוקול הנמצא בשכבת התעבורה, שמטרתו היא לאפשר העברת מידע בצורה אמינה ורציפה. לעומתו, **UDP** הוא פרוטוקול תקשורת המשמש בעיקר ביצירת חיבור עם אחזור נמוך והוא אמין ורציף בצורה פחותה מ-TCP. מנגד, UDP מאפשר האצה של השידורים ע"י מתן אפשרות להעברת נתונים.

אופן ביצוע פרוטוקול TCP: הלקוח שולח לשרת בקשה לפתיחת קשר (SYN). לאחר מכן, השרת מגיב לבקשה כך שמתבצע חיבור. השרת שולח אישור פתיחת הקשר (SYN-ACK) ולבסוף הלקוח שולח אישור לשרת על סיום בניית הקשר (ACK).



אופן ביצוע פרוטוקול UDP: אופן ביצוע הפרוטוקול הינו פשוט לעומת TCP. הוא אינו מצריך אישורים מיוחדים, אלא דורש רק העברה של תוכן ההודעה כך שהלקוח מקבל את התוכן במהירות ללא טיפול בשגיאות במידה וקיימות. לכן בביצוע שלנו (בעיקר במימוש חלק ב- קבצים) נראה כי מצד הלקוח קיימים שני מימושים של פרוטוקול UDP (שליחה וקבלה) ובדומה גם בצד השרת. צד השרת יקבל את האישור מצד הלקוח וישלח את המידע הדרוש אל הלקוח תחת מנגנון של FAST reliable UDP.



למעשה נדרשנו ליצור מערכת מסרים מידיים פרימיטיבית המבוססת על תקשורת. נוכל להבחין לאורך כל הפרויקט כי מימשנו את הפרוטוקול <sup>1</sup> S&W - stop and wait לייצוג וטיפול בשליחת קבצים, הודעות העוברות ועוד.

<sup>1</sup> S&W – stop and wait) השולח שולח את החבילה ומחכה ל-ACK (אישור) של החבילה. ברגע שה-ACK מגיע לשולח, הוא משדר את החבילה הבאה ברציפות. אם ה-ACK לא מתקבל, הוא משדר שוב את החבילה הקודמת.

## הפרויקט שלנו

הפרויקט שלנו מחולק ל 4 מחלקות:

*Client.py* – מחלקת הלקוח.

*Server.py* – מחלקת השרת.

*Consts.py* - מחלקת "הקבועים" שמרכזת את כל המידע הקבוע שאנו מעוניינים בו.

*Tests.py* - מחלקת הטסטים.

בנוסף, יש לנו 2 תיקיות של קבצים:

*server files* - רשימת הקבצים הנמצאים בשרת.

*client files* - רשימת הקבצים שמתקבלים כאשר הלקוח מבקש להוריד קובץ.

## מחלקת *client.py* :

תפקידה של המחלקה היא לדמות את צד הלקוח.

### שימושי המערכת:

הלקוח יכול לבצע מספר פעולות והינם:

1. להתחבר לשרת *connect* - לחיצה על הספרה 1
2. להתנתק מהשרת *disconnect* – לחיצה על הספרה 2
3. לשלוח הודעה פרטית ללקוח אחר *message\_user* - לחיצה על הספרה 3
4. לשלוח הודעה לכל הלקוחות המחוברים *message\_all* - לחיצה על הספרה 4
5. לקבל את רשימת הלקוחות המחוברים *get\_all\_clients* - לחיצה על הספרה 5
6. לקבל את רשימת הקבצים שקיימים בשרת *get\_all\_files* - לחיצה על הספרה 6
7. להוריד מתוך רשימת הקבצים קובץ *download\_file* - לחיצה על הספרה 7
8. לקבל את ההודעות שנשלחו לאותו לקוח מהשרת *get\_all\_messages* - לחיצה על הספרה 8

### מימוש המערכת:

**פונקציה *handle\_int\_input()*** - תפקידה של פונקציה זו היא לקבל מהמשתמש את הספרה המציגה את הפעולה שהוא רוצה לבצע (ראה מעלה), במידה וערך שהמשתמש נתן לא תואם לאחת הספרות המתאימה לפעולות הלקוח האפשריות נציג שגיאה ללקוח שעליו להזין ערך מתאים.

**פונקציה *print\_server\_files(msg\_type, message\_data)*** – תפקידה של פונקציה זו היא להדפיס את רשימת הקבצים הנמצאים בשרת. פעולה זו קורית כאשר הלקוח מבקש מהשרת לקבל את רשימת הקבצים (מספר 6) או להוריד קובץ מתוך רשימת הקבצים (מספר 7). בפונקציה זו אנו עוברים על רשימת הקבצים שמגיעים מהשרת ומדפיס כל שם קובץ.

**פונקציה *send\_message(sock, message)*** – תפקידה של פונקציה זו היא לשלוח ל server הודעות.

**פונקציה *recv\_message(sock)*** – תפקיד פונקציה זו היא לקבל את ההודעות מה server ולהעבירה ל client. צורת ההודעה מן השרת היא כך :

| < serverToClientMsgType > ~ < SenderNickName > ~ < MessageData > |

במידה והתקבלה הודעה ריקה מהשרת, זאת אומרת שהלקוח התנתק מהשרת ותופיע לו הערה וכן התוכנית תתנתק. אחרת, מפצלים את ההודעה שנשלחה לפי "||" ולפי "~" ומחזירים את סוג ההודעה (המספר שלה), שם השולח ותוכן ההודעה.

**פונקציה *recv\_thread(client)*** – אנו פותחים thread לכל פעולה של הלקוח (ניתוק, התחברות וכו'). תפקידה של פונקציה זו היא להתאים את הפעולה בהתאם לסוג הבקשה של הלקוח - אנו מקבלים מהפונקציה *recv\_message* את סוג ההודעה, ההודעה ושם השולח. לדוגמה, במידה והלקוח רצה להתנתק (מספר 2) אנו רוצים להדפיס על המסך הודעת התנתקות וכן לצאת מהתוכנית. דוגמה נוספת, במידה והלקוח רצה לשלוח הודעה לכל המשתמשים המחוברים במערכת (ספרה 4) , הגדרנו בשרת כי בתוך רשימת ההודעות כל הודעה תופרד

באמצעות "<>" ובכך יתאפשר לנו בקלות רבה יותר להפריד בין כל הודעה והודעה בחלק של הלקוח. אזי במידה וקיים אותו תו "<>" בהודעה, נפריד בין כל ההודעות באמצעותו ונדפיס את ההודעות בהתאם ללקוח. אחרת, נדפיס רק את ההודעה בשלמותה (מכיוון שאם לא קיים התו אז אין מספר הודעות אלא רק אחת).  
כאשר הלקוח ירצה להוריד קובץ (ספרה 7), נפתח socket UDP כפי שהתבקשנו, נפתח את הקובץ שהלקוח בחר לכתיבה בבינארי וכן נעבור על כמות הביטים ונגדיר

*checksum – 4 bits ,sequence number – 1 bit,data – the rest bits*

כל פעם נבדוק עבור ה seq checksum האם הם תואמים למידע שהתקבל. במידה והם לא מותאמים אז נמשיך ונדלג על אותו חבילה. לבסוף נגדיר אישור העברה *ack\_response* ונשלח אותו לשרת. כחלק מפקודת *bind* נקשר עם IP ריק שהינו 0,0,0,0 מכיוון שאנו רוצים שכל הsocket יקבל תקשורת מכל IP .

**פונקציה *client\_main()*** - תפקידה להגדיר ולשלוח לשרת את המידע שקיבלנו מהמשתמש בכדי שיפעל בהתאם. כפי שנאמר מעלה, לכל פעולה שהלקוח מבקש לעשות הקדמנו *thread*. כחלק מהספרות של המשתמש הגדרנו *max\_value* - החסם העליון של הערכים שהמשתמש יכול לשים (הערכים האפשריים הינם 1-8). במהלך הפונקציה עבור כל בקשה של הלקוח אנו מסגננים את סוג ההודעה ולפיכך אנו מגדירים את הטיפול ושולחים לבסוף את סוג ההודעה וכן את תוכן ההודעה במידה וקיים אצל השרת.

## מחלקת `server.py` :

תפקידה של המחלקה היא לדמות את צד השרת.

### שימושי המערכת:

אצל השרת יופיעו מספר הודעות:

1. כאשר הלקוח יתחבר למערכת אצל השרת יופיע כי הלקוח התחבר. לדוגמה:  
*Connected to address ('127.0.0.1', 64345)*  
*Client: gal was connected to the server successfully*
2. כאשר הלקוח ירצה להתנתק, יופיע אצל השרת איזה לקוח התנתק מהתחברות. לדוגמה:  
*[+] Client: gal sent disconnection request, disconnecting*  
*[!] Client: gal was disconnected successfully*
3. כאשר הלקוח רוצה להוריד קובץ מתוך רשימת הקבצים בשרת, יופיע בשרת את אחוזי ההורדה של הקובץ וכן את הביט האחרון של הקובץ כמבוקש. לדוגמה:  
*[+] User gal downloaded 21.37% of the file 12.jpg. Last byte is: 212*  
*[+] User gal downloaded 42.73% of the file 12.jpg. Last byte is: 212*  
*[+] User gal downloaded 64.1% of the file 12.jpg. Last byte is: 248*  
*[+] User gal downloaded 85.47% of the file 12.jpg. Last byte is: 154*  
*[+] User gal downloaded 100.0% of the file 12.jpg. Last byte is: 217*

### מימוש המערכת:

בתחילת המחלקה הגדרנו רשימה של `threads`, מילון של לקוחות `clients_dict` המורכב משמות המשתמשים ב `key` וכן ב `value` את `socketn`, מילון של ההודעות של המשתמשים `user_msgs_db` המורכב משמות המשתמשים ב `key` וכן ב `values` את רשימת ההודעות.

**פונקציה `add_msg_to_db(sender, recipient_nickname, msg_content)`** - תפקידה להכניס את רשימת ההודעות של הלקוח לתוך המילון שהגדרנו. במידה והמשתמש קיים בתוך המילון, נכניס לשם שלו את ההודעות שקיבל מהשרת בנוסף להודעות הקודמות שלו. במידה ולא קיים, כלומר לא היו לו הודעות קודמות, נכניס את ההודעה שהתקבלה.

**פונקציה `corrupt_packet()`** - מטרתה של הפונקציה היא להרוס במכוון חבילה באמצעות שינוי של ביט אחד בודד.

דרך הפעולה של הפונקציה היא להגדיר `index` רנדומלית וכן באותו `index` בחבילה להוסיף ביט אחד ובכך החבילה המקורית תשתנה.

**פונקציה** `send_file_rdt(message_data, nickname, client_sock)` - מטרת הפונקציה היא לשלוח את הקובץ ללקוח דרך socket UDP באמצעות FAST reliable UDP – המילים אחרות RDT. תחילה, מגדירים את מיקום הקובץ על פי פונקציית `get_file_path_by_name(file_name)`. נבדוק האם גודל הקובץ גדול מ `MAX_FILE_SIZE` אשר הגדרנו במחלקת הקבועים שלנו. במידה וישנה שגיאה, נדפיס הודעת שגיאה כיוון שהקובץ לא עומד בגבול הגודל שקבענו. בהמשך, נשלח הודעה ריקה ל client והודעה מסוג disconnect וכך נתנתק. אם אין שגיאה, נפתח את הקובץ לקריאה בקובץ בינארי. נגדיר:

`udp_send_sock` ו `udp_recv_sock` – אשר מותאמים לפרוטוקול UDP כמתבקש, יצרנו 2 socket מכיוון שאין לנו אפשרות לשלוח ולקבל באותו ה socket, לשם כך קיים socket לשליחה וכן לקבלה. לאחר מכן, אנו בוחרים port מתוך הטווח שהוגדר לנו במטלה. במידה ונבחר port שנלקח כבר, נגריל port חדש. נשלח ל client הודעה עם ה port הנבחר כך שה `message_data` יוגדר כ `((len(data) + "<>" + port) - זהו הפרוטוקול שהגדרנו עם ההפרדה של התו "<>" במטרה להקל עלינו בהפרדת המשתנים, כך בעצם תיאמנו את החבילה בתקשורת ביניהם. כחלק מפקודת bind נקשר עם bind_ip שהינו 0,0,0,0 מכיוון שאנו רוצים שיקבל תקשורת מכל IP. כחלק משליחת הקובץ, נגדיר זמן למצב בו יהיה timeout. בנוסף, הקצבנו שזמן זה יהיה שנייה - לאחר זמן זה יהיה timeout ואותה חבילה תשלח מחדש. בחלק השני של הפונקציה, אנו בודקים כחלק מאמינות פרוטוקול UDP את כמות המידע שהועבר ללקוח. מבנה החבילה של UDP מורכב מ:`

sequence number – bit 1, checksum – 4 bits, data – the rest

נרצה לבדוק האם החבילה היא בעלת אותו seq וchecksum לחבילה שהתקבלה. במידה והן זהות, החבילה מקבלת אישור וכן מתחילה הורדת הקובץ. בנוסף, מודפסת הודעה המראה על אחוזי ההורדה והביט האחרון של החבילה שנשלחה מהשרת. במידה ויש שינוי בנתונים, החבילה תגיע למצב של timeout כאשר במצב זה היא תמשיך להישלח שוב ושוב בלולאה.

**פונקציה** `get_file_path_by_name(file_name)` - מטרתה של פונקציה זו היא להחזיר את המיקום המדויק של הקובץ בהתאם לבחירה של הלקוח. במהלך הפונקציה אנו משתמשים במודל os המאפשר לנו לגשת אל מערכת הפעלה ובכך לקחת את ה path של הקבצים. `os.getcwd()` – מאפשר לנו לגשת אל מיקום התיקייה הנוכחית שבה המשתמש נמצא. באמצעות `join` אנו מצרפים את הגישה לתוך תיקיית `"server_files"`. לאחר מכן אנו עוברים על כל רשימת הקבצים שנמצאים באותה התיקייה ומחזירים את מיקום הקובץ בהתאם לבחירת הלקוח.

**פונקציה** `remove_and_disconnect_from_client(client, nickname)` - מטרת הפונקציה היא לגרום להתנתקות של הלקוח במידה ובחר בפעולה זו. בפונקציה אנו מנתקים את socket של הלקוח באמצעות `client.close()` וכן מוצאים מתוך `clients_dict` את שם הלקוח שהתנתק.

**פונקציה** `get_server_folder_content()` - מטרת הפונקציה היא להחזיר את כל רשימת הקבצים שנמצאים בתיקייה של server כאשר בין כל שם קובץ יהיה את התו "#" (כך שנוכל לפצל את רשימת הקבצים בדרך קלה יותר בצד הלקוח). בדומה לפונקציה `get_file_path_by_name(file_name)` נעבור על תיקיית הקבצים בהתאם למיקומם ונכניס את הקבצים לתוך רשימה ובין כל קובץ נשים את התו "#".

**פונקציה** `sent_message(target_nickname, message)` - מטרת הפונקציה היא שליחת ההודעה ל `target_nickname`. פרוטוקול ההודעה אצלנו נראה כך:

| < *serverToClientMsgType* > ~ < *SenderNickName* > ~ < *MessageData* > |

**פונקציה `recv_message(client)` - מטרת הפונקציה היא לקבל את המידע המתקבל מ `client`.**  
בפונקציה אנו מפצלים את התו "|" ואת התו "~" ובכך מחזירים את סוג ההודעה ואת תוכן ההודעה.  
פרוטוקול ההודעה אצלנו נראה כך:

| < *ClientToServerMsgType* > ~ < *MessageData* > |

**פונקציה `notify_new_client(new_client_nickname)` - מטרת הפונקציה היא לשלוח הודעה לכל**  
המשתמשים המחוברים בשרת כאשר הלקוח מתחבר לשרת. מגדירים הודעה הנשלחת מה `server` ל `client` שסוגה  
הוא `connect`, לאחר מכן עוברים על כל `client_dict` ושולחים את ההודעה לכל שמות הלקוחות מלבד הלקוח  
הנוכחי.

**פונקציה `handle_connection(client, nickname)` - מטרת פונקציה זו היא להתאים את שליחת ההודעה**  
ל `client` בהתאם לסוג הבקשה שהתקבלה מאותו לקוח. לדוגמה, במידה וסוג הבקשה שהתקבלה היא  
`message_all` אנו נשלח הודעה מה `server` ל `client` המורכבת מסוג ההודעה שהתקבלה מהלקוח. הודעה זו היא  
מסוג שליחת הודעה לכל המשתמשים המחוברים. בנוסף, היא מכילה את תוכן ההודעה. במקרה כזה לא נרצה  
לשלוח את אותה הודעה גם ללקוח הספציפי המבקש את הפעולה לכן נדלג על משתמש זה בזמן שליחת ההודעה  
לכל המשתמשים בלולאה. דוגמה נוספת - כאשר הלקוח מבקש להוריד קובץ מהשרת, בכדי לאפשר הורדת קבצים  
במקביל כמבוקש במטלה אנו רוצים לאפשר שימוש באותה פעולה במקביל לכמה לקוחות על כן נפתח `thread`  
ספציפי לפעולה ההורדה ונשלח לפונקציה `send_file_rdt`.

**פונקציה `server_main()` - מטרתה לטפל ולהגדיר את ההודעות המופיעות בחיבור למערכת. בפונקציה זו הגדרנו**  
`socket TCP` וכן הגדרנו כי השרת יוכל "להקשיב" לכמה חיבורים במקביל. כמו כן, הגדרנו את הודעת ההתחברות  
של הלקוחות וכן את הדפסות המופיעות על המסך. בנוסף, פתחנו `thread` לכל פעולת חיבור לשרת וכן לכל בקשה  
מצד הלקוח. כל ה `threads` הוכנסו ל `thread_list` שהגדרנו בתחילת התוכנית.



## קובץ *consts.py* :

מטרת קובץ זה היא הגדרת הקבועים בתוך הפרויקט על מנת להקל על השימוש בהם.

**פונקציה *calculate\_checksum(data)*** - מטרת פונקציה זו היא לחשב את ה *checksum* . דרך פעולה היא להמיר את ה *data* ל *string* באמצעות ספריית *hashlib* של *python* ופונקציית *hash - sha1* ואז נתייחס לאותה המחזורת כביטים וניקח את ארבעת הביטים האחרונים המייצגים את ה *checksum* של ה *data*.

### הקבועים:

*BIND\_IP* = "0.0.0.0" - במטרה שיוכלו לקבל תקשורת מכל IP.

*SERVER\_IP* = "127.0.0.1" - localhost or loopback address

כל המחשבים משתמשים בכתובת הזו ככתובת שלהם, אבל היא לא מאפשרת למחשבים לתקשר עם מכשירים אחרים כפי שכתובת IP אמיתית עושה.

*PORTS\_RANGE* = 15

*MAX\_CONNECTIONS* = 5

*SERVER\_NICKNAME* = "SERVER"

*UDP\_MAX\_DATAGRAM\_SIZE* = 1024 - גודל מקסימלי למידע

*MAX\_FILE\_SIZE* = 9 \* *UDP\_MAX\_DATAGRAM\_SIZE* - 9 זה מקסימום של ה sequence number

*MAX\_DATA\_SIZE* = *UDP\_MAX\_DATAGRAM\_SIZE* - 5 - מקסימום חבילת UDP פחות גודל

checksum + sequence number

בתוך קובץ זה ישנם מספר מחלקות :

*class ClientToServerMsgType(enum.Enum)* - מגדיר את סוג ההודעה הנשלחת מה client לserver.

*class ClientToServerMsg* - מגדירה את מבנה ההודעה מהclient ל server כך:

| < ClientToServerMsgType > ~ < MessageData > |

*class ServerToClientMsgType(enum.Enum)* - מגדיר את סוג ההודעה הנשלחת מהserver לclient.

*class ServerToClientMsg* - מגדירה את מבנה ההודעה מהserver לclient כך :

| < ServerToClientMsgType > ~ < SenderNickName > ~ < MessageData > |

שם השולח יכול להיות גם שם השרת בנוסף לשם הלקוח האמיתי.

## קובץ *tests.py* :

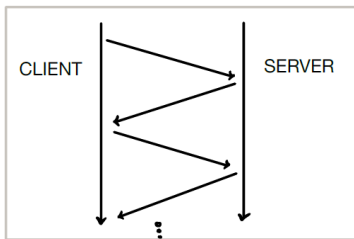
קובץ זה בעצם מגדיר את מחלקת הבדיקות שלנו. במחלקה זו, בדקנו את שליחת ההודעה בין הclient לserver וכן בין server ל- client.

בנוסף, ביצענו בדיקות על שליחת הקבצים בפרויקט, בדקנו שקליטת רשימת הקבצים מתוך תיקיית הקבצים בשרת מעודכנת וכן האם הפונקציה *corrupt\_packet* אכן משבשת לנו את החבילות.

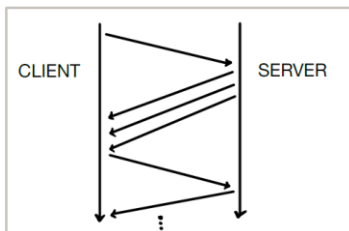
## שאלות עבור חלק ב'

### שאלה 1:

דיאגרמת מצבים - קיימים 4 מצבים אפשריים בהם נראה את המערכת מתמודדת.



**מצב רגיל:** מתקבלת הודעה בשרת מצד הלקוח להורדת קובץ מסוים למשל. השרת שולח חזרה את החבילה הראשונה ללקוח. הלקוח מעדכן את השרת שקיבל את החבילה וכך השרת ממשיך לחבילה הבאה וכן הלאה. אופן בדיקת התאמת החבילה מתבצעת באופן ידני שאנחנו יצאנו עם חישוב checksum - שמתקבל אותו רצף ביטים נכון.



**מצב עם איבוד חבילות + עיכוב:** מימשנו שני מקרים אלה באופן דומה. לדוגמא: הורדת קובץ – מתבצעת בקשה מצד הלקוח להורדת קובץ מסוים והבקשה מתקבלת בצד השרת. כעת במצב של איבוד חבילה הכוונה היא שהחבילה אינה הגיעה בשלמותה אל הלקוח ולכן לא מתקבלת הודעה מטעם הלקוח בשרת עד שמתקבלת החבילה. נראה מבט עמוק יותר מה הכוונה באיבוד חבילה – המערכת שלנו הגדרנו פונקציה שנקראת "corrupt\_packet" שלמעשה היא לוקחת את החבילה ומשנה בה ביט אחד ככה שהיא משנה את הרצף באופן מכוון, שזה מה שבעצם קורה באיבוד חבילה. במצב זה שאין אישור מהלקוח, אחרי שניה של המתנה השרת משדר את אותה החבילה בדיוק שוב עד לאישור של הלקוח להמשך שליחת החבילה. באופן דומה לגבי עיכוב – השרת אינו מקבל אישור מצד הלקוח ולאחר המתנה של שניה הוא שולח שוב את אותה החבילה, בדומה לאיבוד חבילה.

**מצב של timeout:** מצב קבוע של שליחת החבילה שוב ושוב מצד השרת.

### שאלה 2:

איך המערכת מתגברת על האיבוד חבילות-

במערכת שלנו, אנו בודקים בהעברת הקובץ בכל פעם האם ישנם שגיאות בהעברת הביטים (חישוב checksum ו seq). במידה ואין שגיאות זאת אומרת שהחבילה הגיעה בשלמותה, לעומת זאת במידה וישנן שגיאות כלומר מספר הביטים שהתקבלו שונה ממספר הביטים שנשלחו אז אין כניסה לתוך הלולאה ונגיע למצב של timeout במצב כזה אנחנו נשלח שוב פעם את החבילה עד לקבלת משלוח תקין ומושלם.

### שאלה 3:

איך המערכת מתגברת על בעיות latency-

בדומה לאיבוד חבילות, במידה והחבילה (המידע) לא הגיעה בזמן שהקוצב לה - הקצבנו שנייה (settimeout(1)). נשלח שוב את החבילה עד שתתקבל הודעה בצד השרת מטעם הלקוח.

## נבחן תצלומים של Wireshark – נפרט כל מצב.

ביצוע הסנפה של מידע שבוצע במערכת. הורדה של קובץ ספציפי מתוך client-files:

No.	Time	Source	Destination	Protocol	Length	Info
34	14.062071	10.100.102.53	239.255.255.250	SSDP	206	M-SEARCH * HTTP/1.1
35	14.068866	192.168.56.1	239.255.255.250	SSDP	206	M-SEARCH * HTTP/1.1
36	14.068972	192.168.115.1	239.255.255.250	SSDP	206	M-SEARCH * HTTP/1.1
37	14.069023	10.100.102.53	239.255.255.250	SSDP	206	M-SEARCH * HTTP/1.1
46	25.748930	127.0.0.1	127.0.0.1	UDP	37	60942 → 55002 Len=5
47	25.749389	127.0.0.1	127.0.0.1	UDP	1056	60943 → 60941 Len=1024
48	25.749779	127.0.0.1	127.0.0.1	UDP	37	60942 → 55002 Len=5
49	25.750294	127.0.0.1	127.0.0.1	UDP	1056	60943 → 60941 Len=1024
50	25.750512	127.0.0.1	127.0.0.1	UDP	37	60942 → 55002 Len=5
51	25.751052	127.0.0.1	127.0.0.1	UDP	1056	60943 → 60941 Len=1024
52	25.751239	127.0.0.1	127.0.0.1	UDP	37	60942 → 55002 Len=5
53	25.751691	127.0.0.1	127.0.0.1	UDP	1056	60943 → 60941 Len=1024
54	25.751879	127.0.0.1	127.0.0.1	UDP	37	60942 → 55002 Len=5
55	25.752361	127.0.0.1	127.0.0.1	UDP	1056	60943 → 60941 Len=1024
56	25.752541	127.0.0.1	127.0.0.1	UDP	37	60942 → 55002 Len=5
57	25.752982	127.0.0.1	127.0.0.1	UDP	1056	60943 → 60941 Len=1024
58	25.753166	127.0.0.1	127.0.0.1	UDP	37	60942 → 55002 Len=5
59	25.753618	127.0.0.1	127.0.0.1	UDP	322	60943 → 60941 Len=290
60	25.753824	127.0.0.1	127.0.0.1	UDP	37	60942 → 55002 Len=5

ראשית נראה את העברת הקובץ תחת פרוטוקול UDP ולא אחר מכיוון שיצרנו שהעברת הקבצים תתבצע בפרוטוקול זה. ניתן לראות את תחילת הקשר ביניהם בכך שיש אישור לבקשת מידע מהלקוח לשרת. בכל פעם שיש העברת חבילה מסוימת נבחין כי קיים אישור (ack) לחבילה הבאה - סך כל החבילות משלימות את העברת כל ה-data כולה של אותה תמונה. ככה על פי אישור זה הפרוטוקול ממשיך לשליחת החבילה הבאה להשלמת ההעברה כולה = הורדת הקובץ.

נממש איבוד חבילות באופן מכוון ע"י שורת פקודה המבצעת זאת. בתמונה למטה נראה ביצוע של 40% איבוד חבילות. נממש אחוז כה גבוה מכיוון שברגע שיש איבוד נמוך יחסית לא נוכל לראות בהבדל באופן ממשי.

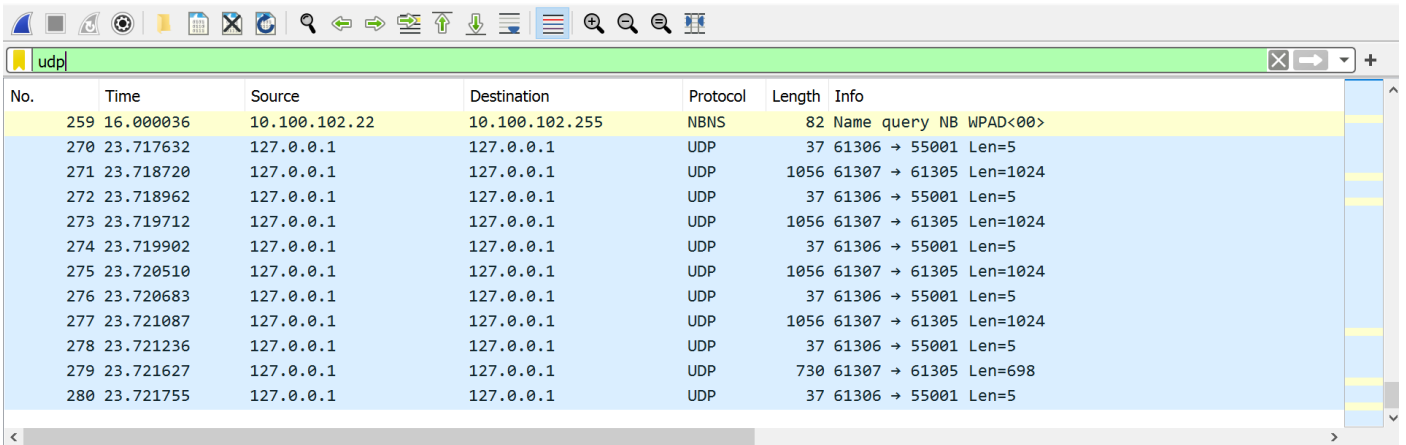
No.	Time	Source	Destination	Protocol	Length	Info
35	29.076828	127.0.0.1	127.0.0.1	UDP	37	59239 → 55011 Len=5
36	29.077532	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
37	29.077991	127.0.0.1	127.0.0.1	UDP	37	59239 → 55011 Len=5
38	29.078581	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
39	30.081957	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
40	32.094317	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
41	39.146517	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
42	42.175557	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
43	44.194078	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
44	45.200838	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
45	47.217525	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
46	48.226265	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
47	49.232332	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
48	52.255458	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
49	53.266048	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
50	55.286600	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
51	56.295444	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
52	57.310346	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024
53	58.311133	127.0.0.1	127.0.0.1	UDP	1056	59240 → 59238 Len=1024

במקרה זה בשונה מהמקרה הקודם, נראה כאן כי רק חלק קטן מתוך כל המידע עבר: שורה 35 מראה על אישור מהלקוח לשרת, בשורה 36 נשלחת חבילה בגודל המקסימלי ע"י השרת ללקוח ולאחר מכן ניתן להבין כי שוב התקבל אישור מטעם הלקוח על קבלת החבילה ובקשת החבילה הבאה. משורה 38 והלאה ניתן לראות כי אותה חבילה נשלחת שוב ושוב מהשרת אל הלקוח, והלקוח אינו מקבל אותו עקב איבוד החבילות. כלומר החבילה אינה מתקבלת

אצל הלקוח אז ברגע שהשרת אינו מקבל תשובה מהלקוח (ack) תוך שניה (הגדרנו כך) אז הוא ישלח שוב ושוב עד להודעה חזרה מהלקוח.

וידיו התאמה כי כל חבילה הנשלחת שוב ושוב היא אותה חבילה בדיוק מתבצעת בעזרת השוואת הרצף הבינארי בכל שליחה – במידה והן זהות מדובר על אותה החבילה במידה ושונות אחרת.

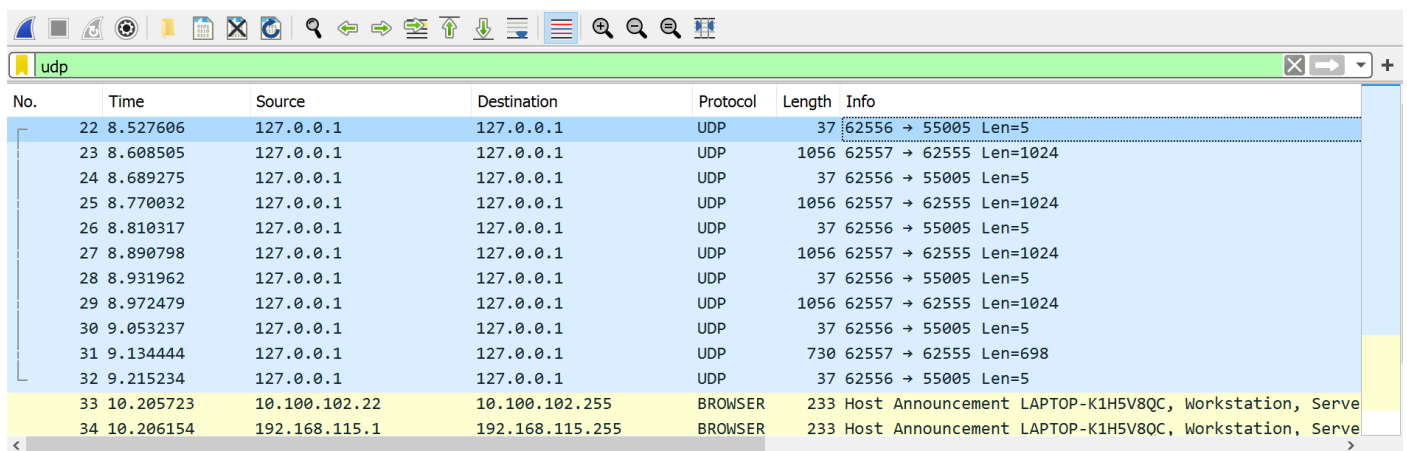
נממש מקרה של עיכוב המערכת בעזרת שורת פקודה המבצעת זאת. בתמונה למטה נראה את ההרצה הרגילה ללא



No.	Time	Source	Destination	Protocol	Length	Info
259	16.000036	10.100.102.22	10.100.102.255	NBNS	82	Name query NB WPAD<00>
270	23.717632	127.0.0.1	127.0.0.1	UDP	37	61306 → 55001 Len=5
271	23.718720	127.0.0.1	127.0.0.1	UDP	1056	61307 → 61305 Len=1024
272	23.718962	127.0.0.1	127.0.0.1	UDP	37	61306 → 55001 Len=5
273	23.719712	127.0.0.1	127.0.0.1	UDP	1056	61307 → 61305 Len=1024
274	23.719902	127.0.0.1	127.0.0.1	UDP	37	61306 → 55001 Len=5
275	23.720510	127.0.0.1	127.0.0.1	UDP	1056	61307 → 61305 Len=1024
276	23.720683	127.0.0.1	127.0.0.1	UDP	37	61306 → 55001 Len=5
277	23.721087	127.0.0.1	127.0.0.1	UDP	1056	61307 → 61305 Len=1024
278	23.721236	127.0.0.1	127.0.0.1	UDP	37	61306 → 55001 Len=5
279	23.721627	127.0.0.1	127.0.0.1	UDP	730	61307 → 61305 Len=698
280	23.721755	127.0.0.1	127.0.0.1	UDP	37	61306 → 55001 Len=5

דילאי:

כעת נראה הרצה של המערכת, עם אותם פעולות על המערכת והורדה של אותו הקובץ עם דילאי:



No.	Time	Source	Destination	Protocol	Length	Info
22	8.527606	127.0.0.1	127.0.0.1	UDP	37	62556 → 55005 Len=5
23	8.608505	127.0.0.1	127.0.0.1	UDP	1056	62557 → 62555 Len=1024
24	8.689275	127.0.0.1	127.0.0.1	UDP	37	62556 → 55005 Len=5
25	8.770032	127.0.0.1	127.0.0.1	UDP	1056	62557 → 62555 Len=1024
26	8.810317	127.0.0.1	127.0.0.1	UDP	37	62556 → 55005 Len=5
27	8.890798	127.0.0.1	127.0.0.1	UDP	1056	62557 → 62555 Len=1024
28	8.931962	127.0.0.1	127.0.0.1	UDP	37	62556 → 55005 Len=5
29	8.972479	127.0.0.1	127.0.0.1	UDP	1056	62557 → 62555 Len=1024
30	9.053237	127.0.0.1	127.0.0.1	UDP	37	62556 → 55005 Len=5
31	9.134444	127.0.0.1	127.0.0.1	UDP	730	62557 → 62555 Len=698
32	9.215234	127.0.0.1	127.0.0.1	UDP	37	62556 → 55005 Len=5
33	10.205723	10.100.102.22	10.100.102.255	BROWSER	233	Host Announcement LAPTOP-K1H5V8QC, Workstation, Serve
34	10.206154	192.168.115.1	192.168.115.255	BROWSER	233	Host Announcement LAPTOP-K1H5V8QC, Workstation, Serve

ניתן לראות את ההבדל באופן ניכר. נתמקד בעמודה של הזמן (time): בתמונה הראשונה ניתן לראות יש הבדל מאוד מינימלי בין שליחת הבקשה מצד הלקוח ושליחת החבילה מצד השרת וכן הלאה עד להעברת החבילה כולה. לדוגמה נתבונן בשורות 270 – 275: ניתן לראות כי יש הרצה מהירה מאוד הבדל של פחות ממאית שניה, בערך בזמן של 0.003 עברו כבר 3 חבילות מתוך כל המידע להשלמת ההורדה. בהשוואה לצילום השני על פי העמודה של הזמן נחשב כי בערך בזמן של 0.404 עברו 3 חבילות מתוך כל המידע להשלמת ההורדה. כן מתבטא העיכוב במערכת.

לכל הצילומים האלה ניתן לראות קובץ pcap מתאים בקובץ מצורף תחת השם: "pcap\_records".

# איך מריצים את המערכת?

במערכת רצה בסביבות עבודה התומכות בשפת פיתון, לדוגמא: *PyCharm, visual studio*. נתמכת גם במערכות הפעלה שונות (windows, Linux).

תחילה, תחת הקישור שמוצג בתחילת הפרויקט יש להוריד את הקובץ ZIP המכיל קבצי פיתון ולהעבירם לתוך פרויקט חדש שנפתח מראש מאחת סביבות העבודה השונות.

שנית, נפרט אופן ריצה בכל אחת ממערכות ההפעלה.

## הרצה ב – windows:

נדרש לפתוח לפחות שני טרמינלים (cmd), ננתב את המערכת לפי מיקום התוכנה במחשב ונריץ:  
בטרמינל הראשון נריץ את שורת הפקודה הבאה:

```
"python server.py"
```

ישר מתקבלת הודעה כי השרת נפתח והוא מאזין להתחברות.

בטרמינל השני נריץ את שורת הפקודה הבאה:

```
"python client.py"
```

ישר תתקבל הודעה איזה כינוי תבחר לעצמך במערכת, מיד אחרי מתקבלת ההודעה באופן אוטומטי של איזה פעולות נרצה לבצע להמשך השימוש (קיים הסבר מפורט במחלה client.py).

במידה ותרצו יותר מלקוח אחד במערכת, יש צורך לפתוח עוד טרמינלים בהתאם לכמות הלקוחות.

## הרצה ב – Linux:

בדומה ל – windows, השוני בשורת הפקודה:  
לפתיחת טרמינל לשרת נריץ את שורת הפקודה הבאה:

```
"python3 server.py"
```

לפתיחת טרמינל ללקוח נריץ את שורת הפקודה הבאה:

```
"python3 client.py"
```

מצורף סרטון המראה דוגמה לאופן הרצה של המערכת תחת אותו קישור.

# Enjoy!

## חלק ג'

### שאלה 1 :

בהינתן מחשב חדש המתחבר לרשת נראה איך כל ההודעות שעוברות מהחיבור הראשוני ל – switch ועד שההודעה מתקבלת בצד השני של הצ'אט:

ברגע שנחבר פיזית את מחשב לרשת, כלומר נכניס ל-LAN, לכבל רשת ועד שמתחיל המעבר של הביטים (בין אם זה חיבור ישיר לראוטר או חיבור ל-switch של כמה מחשבים ומשם קיים חיבור לראוטר) מתחיל החיבור לאינטרנט בכך שהספק של הראוטר נותן למחשב שלנו כתובת IP וחיבור לכתובת פורט מסוימת (\*). בשלב זה המחשב מחובר לאינטרנט וקיימת לנו גישה לכתובות ורשתות שונות.

כאשר יש לנו כתובת IP ואנו משתמשים באפליקציה כלשהי, במקרה שלנו צ'אט, או לכל אפליקציה למיניה נתבונן במימוש של המודל כאשר ברמה העליונה האפליקציה. נראה כי הצ'אט אכן חלק מרמת האפליקציה מכיוון: הינו כולל את כל ה-GUI, תוכן ההודעה, פרסום, שפה קידוד ועוד.

ברגע שנשלחת הודעה מכתובת המקור, למעשה לוקחים את כל המידע הנ"ל ומבצעים מעין כפסולציה כלשהי ומתחילים לרדת בשכבות המודל. נגיע לשלב של network, ההודעה יוצאת באיזושהי פאקטה שמיועדת לכתובת IP מסוימת ברשת ואז לפי טבלאות הניתוב, דבר זה לא קורה ב-switch אלא קורה ברכיבים של שכבה שלישית, לדוגמה ראוטר, לכן אותו ראוטר יודע לאיזה ראוטר חיצוני לנתב את המידע עד שזה יגיע לאיזושהי רשת פנימית שה-IP שאנו שולחים אליו את ההודעה, כלומר שרת היעד, נמצא בה. ברגע שההודעה הגיע לשם בשביל למצוא את המחשב הספציפי, כלומר את השרת הספציפי שמפעיל את הצ'אט לצורך העניין, נרד עוד שכבה למטה לכתובת MAC ואז נבדוק לאיזה MAC ההודעה נשלחה ואז לפי מידע זה נדע לנתב את ההודעה ב-LAN ברשת הפנימית לאותו מחשב (ע"י הראוטר של אותה רשת). לאחר מכן נראה את השכבה שהמידע מועבר בה באופן פיזית (מעבר של ביטים). רשת היעד אותה חיפשנו יודעת שאליו מכוונת ההודעה על פי ה-MAC וה-IP שמלכתחילה הגיע אליו על פי רשת המקור, ובשלב הבא מציג את התוכן של ההודעה (DATA) – כך בעצם נראה את זה בצ'אט בשכבות העליונות. ברגע שנשלחת הודעה בחזרה לרשת המקור, קורה אותו דבר הפוך רק שהוא כבר יודע מה ה-IP וה-MAC של המקור, מכיוון שהוא נחשף למידע הזה בהודעות שנשלחו אליו כבר ולכן הינו יודע לנתב את המידע לרשת ולמחשב הספציפיים של המקור.

לסיכום שליחת ההודעה עובדת לפי המודל – ברגע ששלחנו את ההודעה נרד בשכבות וכאשר ההודעה מגיעה ליעד שלה היא מתחילה לעלות בשכבות חזרה עד לרמת האפליקציה.

(\*) התפקיד של פתיחת פורטים נועד לעזור לנו לומר לראוטר שלנו שהתוכנה שאותה הוא חוסם, אינה מזיקה ושיאפשר לה לפעול.

### שאלה 2 :

CRC (בעברית: בדיקת יתירות מחזורית) משמש לאיתור שגיאות בהעברת נתונים .

לפני העברת המידע מחושב ה-CRC ומתווסף למידע המועבר. לאחר העברת המידע, המצד המקבל מאשר באמצעות ה-CRC שהמידע הועבר ללא שינויים.

### שאלה 3 :

ההבדלים הינם :

- http 1.0 , http 1.1 , http 2.0 עובדות על גבי פרוטוקול TCP לעומת QUIC העובד על גבי פרוטוקול UDP.
- http 1.1, http 2.0 , QUIC בעלי מהירות גבוהה יותר בשליחת המידע .
- http 1.1-מאפשרת שליחת מספר בקשות וכן קבלת מספר תגובות.
- http 1.0 - מאפשרת שליחת בקשה בודדת וכן קבלת תגובה בודדת.
- http 2.0 –משתמשת בריבוב הכוונה היא היכולת לשדר מספר אותות בו זמנית על אותו התווך.
- http 1.0 - המידע מועבר בדרך לא מוצפנת שכן הוא מועבר בקובץ טקסט .
- http 1.1 , http 2.0 –המידע מועבר בדרך יותר מוצפנת שכן הוא מועבר בקובץ בינארי.
- QUIC - המידע מועבר בדרך מוצפנת.

### שאלה 4:

מספר port הוא ערוץ תקשורת שממוספר בין 1 ל65000. כל התקני הרשת משתמשים בהם ולרובם יש את היכולת לשנות אותם בעת הצורך. הם נוצרו במקור כדי לאפשר לתוכניות מרובות להשתמש באותה כתובת IP. הסיבה הנפוצה ביותר לצורך להשתמש במספרי port היא גישה מרחוק. לדוגמא, מצב שבו יש לאדם 2 מצלמות רשת, המתחברות דרך אותו נתב והוא רוצה להתחבר מרחוק לשתי המצלמות דרך port 80. מכיוון שלא ניתן להעביר port בודד ליותר מכתובת IP מקומית אחת בו זמנית, לא ניתן לגשת לשתי המצלמות. הפתרון הוא להשתמש בשתי port נפרדים- במקרה זה, ניתן להשתמש בport 8000 עבור HTTP port במצלמה אחת וב8001 עבור המצלמה השנייה.

### שאלה 5:

subnet (רשת משנה) הוא חלק מרשת גדולה יותר-רשתות משנה הן מחיצה לוגית של רשת IP למקטעי רשת מרחבים וקטנים יותר.

IP הינה שיטה לשליחת מידע ממחשב אחד לשני דרך האינטרנט. כל מחשב או משתמש באינטרנט בעל לפחות כתובת IP המשמשת כמזהה ייחודי. בדרך כלל, ארגונים ישתמשו ברשת משנה כדי לחלק רשתות גדולות לתת רשתות קטנות יותר ויעילות יותר. אחת המטרות של רשת משנה היא לפצל רשת גדולה לקבוצה של רשתות קטנה יותר ומקשורות ביניהן כדי לאפשר גישה יותר יעילה בין הרשתות. בדרך זו, התעבורה לא צריכה לזרום דרך מסלולים מיותרים, מה שמגביר את מהירויות הרשת.

רשתות משנה משומשות לדברים רבים, ביניהם:

- הקצאה מחדש של כתובות IP
- הקלה על עומס ברשת-אם חלק גדול מתעבורת הארגון אמור להיות משותף באופן קבוע בין אותו אשכול מחשבים, הצבתם באותה רשת משנה יכולה להפחית את תעבורת הרשת. ללא רשת משנה, כל המחשבים והשרתים ברשת יראו פקטות המכילות מידע מכל מחשב אחר.

- שיפור אבטחת הרשת-רשת משנה מאפשרת למנהלי רשת לצמצם איומים ברחבי הרשת ע"י הסגר של חלקים שנפגעו ברשת והצבת קשיים למסיגי גבול לנוע ברשתות הארגון.

## שאלה 6:

MAC address הינה משמשת כמזהה ייחודי הנמצא על כל רכיב תקשורת. הוא מוטבע בדרך כלל בכרטיס הרשת של המחשב או המודם. כתובות MAC בפורמטים מסוימים מורכבים מ-48 סיביות, מה שמאפשר כ-2 בחזקת 48 כתובות שונות.

כתובת ה-IP נדרשת כדי ליצור טבלת ניתוב שמאפשרת העברה של חבילות תקשורת במהירות. הן כתובות לוגיות וניתנות לניתוב. לדוגמא, מחשב 1 יכול ללמוד את כתובת ה-IP של מחשב 2. עם זאת, כתובות MAC הן פיזיות ואינן ניתנות לניתוב. כתובת MAC תשודר תמיד אך היא תוגבל למקטע הרשת של תחום Subnet. לכן, מחשב 1 לא יכול ללמוד את כתובת ה-MAC של מחשב 2.

מכאן, ניתן להבין מדוע למחשבים יש גם כתובות MAC וגם כתובות IP- כתובות MAC מטפלות בחיבור הפיזי ממחשב למחשב בעוד שכתובות IP מטפלות בחיבור הלוגי הניתן לניתוב הן ממחשב למחשב והן מרשת לרשת.

## שאלה 7:

Routers או נתבים הם התקני רשת מחשבים שתפקידם נוגע בעיקר ב-2 פונקציות-

1. יצירה ותחזוקה של רשת מקומית

2. ניהול הנתונים הנכנסים והיוצאים מהרשת וכן נתונים הנעים בתוך הרשת. בנוסף, זה עוזר לטיפול במספר רשתות ומנתב את הקשר ביניהן.

Switch או מתג הוא התקן רשת מחשבים המחובר התקנים שונים יחד ברשת מחשבים אחת. בנוסף, Switch עשוי לשמש גם לניתוב מידע בצורה של נתונים אלקטרוניים הנשלחים דרך רשתות.

NAT (Network address translation) הינו שיטה לניתוב ברשתות מחשבים, בה כתובת IP שיוצאת משימוש תחזור אל מאגר הכתובות ותינתן לשימוש חוזר במחשב אחר במקרה הצורך. אחד השימושים הנפוצים ב-NAT הוא חיבור של מספר מחשבים המשתמשים באותה הרשמת המקומית לאינטרנט באמצעות כתובת IP אחת בלבד.

התפקיד המרכזי המשותף לנתב ומתג הוא שבשני המקרים מדובר על התקני חיבור ברשת. נתב משמש לבחירת הניתוב היעיל ביותר שבה פקטת מידע תגיע ליעדה. מתג מאחסן את הפקטה שהגיעה אליו, מעבד אותה כדי לקבוע את היעד שלה ומעביר את המידע ליעד ספציפי. עם זאת, ההבדל המרכזי ביניהם הוא שנתב מחבר יחד רשתות שונות ואילו מתג מחבר התקנים מרובים זה לזה ליצירת רשת אחת.

כמו כן, קיימים הבדלים משמעותיים נוספים- המתג פועל על שכבת קישור נתונים (Data link layer) ורשת. מנגד, נתב עובד על השכבה הפיזית, שכבת קישור נתונים ושכבת רשת.

הבדל מהותי נוסף היא העובדה שנתבים יכולים לעבוד הן במצבי רשת קווית ואלחוטית ומהצד השני, מתגים מוגבלים לחיבורי רשת קווית. הנתב מציע שירותי NAT וכן שירותים נוספים, בעוד שהמתג לא מציע שירותים כאלו. בסוגים שונים של סביבות רשת, קיים יתרון מבחינת המהירות לנתב ולמתג- ב-MAN או EAN, הנתב יעבוד מהר יותר מהמתג ואילו בסביבת LAN קיים יתרון למתג מבחינת המהירות.



כאשר משווים בין טכניקת NAT לנתב, אפשר לראות ש-NAT עובד עם נתב כדי לאפשר ניתוב IP פרטי לרשת ציבורית. נתב לא עושה זאת בעצמו, ולכן יש צורך ב-NAT. NAT מתרגם IP פרטי לIP ציבורי, מנגד, פעולת הניתוב היא בעצם העברת פקטות מידע מרשת אחת לאחרת וזהו ההבדל המרכזי בין הפעולות של הנתב ו-NAT.

## שאלה 8:

IPv4 הוא פרוטוקול האינטרנט המנתב כיום את רוב תעבורת האינטרנט. כיום, על אף שהפרוטוקול מספק 4,294,967,296 כתובות ייחודיות, מספר זה הפך לבלתי מספיק ביחס למספר המכשירים ברחבי העולם המחוברים לאינטרנט.

בכדי להתמודד עם בעיה זו, פותחו טכנולוגיות שמטרתן הייתה להאט את קצב דלדול הכתובות הייחודיות-

• CIDR – classless inter-domain routing – שיטת CIDR הוטמעה בשנת 1993 בכדי להאט את המצוי המהיר של כתובות IPv4. השיטה הייתה שיטה חדשה וקומפקטית לייצוג כתובות IP. בסימון CIDR, כתובות נכתבות עם סיומת המוכנסת באמצעות קו נטוי המציין את מספר הביטים של הקידומת. לדוגמא, הסיומת 16 משמעותה ש-16 הסיביות הראשונות מתוך 32 הסיביות של כתובת IPv4 מוגדרות ע"י הרשת ושאר 16 הסיביות שנותרו מוגדרות ע"י המארז.

• NAT – כאמור, NAT מאפשרת לספקיות אינטרנט וחברות להשתמש בכתובות IP פרטיות מיוחדות כדי לחבר רשתות מחשבים לאינטרנט באמצעות כתובת IP ציבורית אחת. בדרך זו, הם יכולים להשתמש ב-IPv4 עבור רשת פרטית שלמה במקום כתובת IP לכל התקן רשת. בצורה כזו, NAT אפשר להאט את המיצוי המהיר של כתובות IPv4

במקביל, ניסו לפתח שיטות ופרוטוקולים שונים שישמשו כפתרון לטווח הרחוק. אחד מהן היא יציאה של גרסה עוקבת ל-IPv4 הנקראת IPv6. IPv6 הוא פרוטוקול בשכבת הרשת, ששימושו הינו דומה לשימוש ב-IPv4 – העברת נתונים ברשתות מבוססות מיתוג מנות, לרבות ברשת האינטרנט. כתובת IPv6 מורכבת מ-128 סיביות-64 הראשונות משמשות לזיהוי תת הרשת וה-64 האחרונות משמשות כמזהה ממשק. ערכו של מזהה הממשק נקבע ע"י כתובת ה-MAC לרוב.

## שאלה 9:

נפרט באופן כללי על הפרוטוקולים OSPF, RIP, BGP. OSPF – עובד בתוך AS, משתמש בעלות העברת הנתונים לצורך חישוב המרחק, ותמיד יבחר את הנתוב הזול ביותר להעברת חבילה מהמקור אל היעד – בדרך כלל יותר דינאמי משאר הפרוטוקולים. RIP – סופר את מספר הצעדים הקצר ביותר בראוטר אחד לראוטר אחר. BGP – מנהל בעזרת טבלה של הרשתות המחוברות אליו, והקשרים ביניהן לבין רשתות אחרות, ומבצע החלטות ניתוב על בסיס הקשרים בין הרשתות ומדיניות המוכתבת בצורה ידנית על ידי מנהל הרשת.

באופן כללי כאשר נתבים שונים לומדים על תת רשת מסוימת שיושבת מאחורי ראוטר כלשהו, אז אותו ראוטר מפרסם לשכנים שלו באותו AS הודעות כתלות בפרוטוקול, כלומר בחירת המסלול שהכי טוב להגיע אליו כך שכל השכנים שלהם יכירו אותם, ברגע שכל השכנים שלו ב-AS מכירים אותו. נראה בשאלה זו את השילוב והקישוריות של הפרוטוקולים.

נתבונן ונסביר את הערך של X בכל ארבעת הסעיפים האחרונים. X מייצג מה הוא הפרוטוקול האחרון שהוא רואה.

הנתב 3C - מכיוון שהוא הנתב החיצוני ב-AS3 אז הוא מדבר בBGP עם ה AS4 , על פי נתון(d).

הנתב 3A – נתב זה לומד מנתב 3C. אחרי ש3C למד מ OSPF , אחרי הפרוטוקול האחרון שהוא ראה הינו OSPF.

הנתב 1C – לומד אחרי 3a ומזה שהוא מדבר איתו, על פי נתון (d), הפרוטוקול הינו BGP.

הנתב 2C – לומד בתוך AS2, על פי נתון (a), בפרוטוקול OSPF.