בס"ד

פרויקט סיום
רשתות
תקשורת

<u>מגישים:</u>

ג'מיל יעקב: 207894775

עמית שטיינמץ: 2072729373

תוכן עניינים:

DHCP server

DNS local server

FTP application server

- TCP
- RUDP
  - Reliability
  - Congestion Control
  - Flow Control

דיאגרמת מצבים

שאלות על הפרויקט

נציין ש:

בפרויקט זה עבדנו בסביבת עבודה – pycharm, במערכת הפעלה windows.

DHCP server

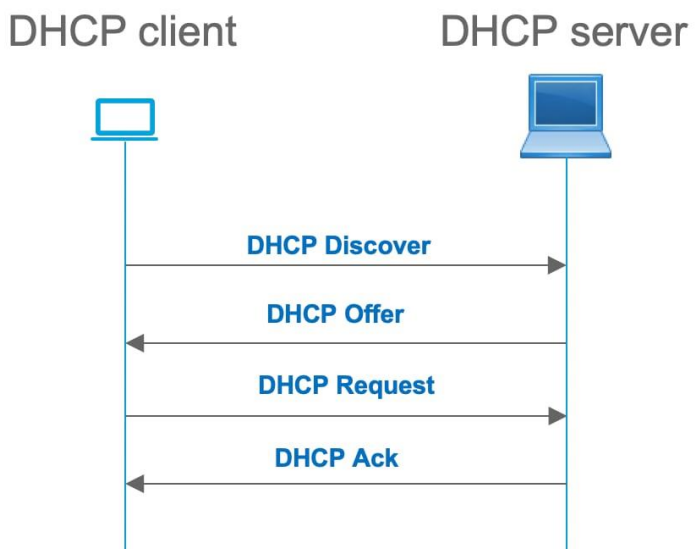
בחלק זה, נסביר מה זה DHCP בכלל ונציג קטעים מהקוד ונסביר אותם. בנוסף, נציג פלטים מהרצות שונות של התוכנית, ונוסיף גם הקלטות של ויירשארק.

נסביר בקטנה מה זה DHCP :

DHCP (Dynamic Host Configuration Protocol) הוא פרוטוקול רשת המשמש להקצאה אוטומטית של כתובות IP.

במילים אחרות, כאשר התקן מצטרף לרשת המשתמשת ב-DHCP, הוא מבקש כתובת IP משרת ה-DHCP. לאחר מכן השרת מקצה כתובת IP זמינה למכשיר, יחד עם הגדרות רשת אחרות כגון מסכת רשת המשנה ושער ברירת המחדל. זה מאפשר למכשירים להתחבר לרשת ולתקשר עם מכשירים אחרים ללא צורך בתצורה ידנית של הגדרות הרשת.

שיטת התקשורת של הפרוטוקול :



נתחיל בקוד:

קודם כל חשוב לנו להדגיש שיש המון שיטות איך אפשר לממש DHCP סרבר וקליינט כמו למשל להשתמש בספרייה struct ולבנות את הפקיטה מאפס.
אנחנו ספציפית בחרנו את האופציה הכי קלה , והכי מפחיתה סיכוי לפגים שעלולים להופיע בקוד

```
from scapy.all import *
from scapy.layers.dhcp import DHCP, BOOTP
from scapy.layers.inet import UDP, IP
from scapy.layers.l2 import Ether
```

<u>הסבר:</u>

עשינו import לספרייה המובנית בשפה שהיא ספרייה מאוד שימושית לבנות את הפקיטות שלנו עם כמה שפחות שגיאות ו פגים שהיינו עלולים לקבל בשיטות אחרות , כמובן עשינו גם כן import לכל המחלקות הרלוונטיות לנו בקוד.

```
# Define the IP address range for the pool
ip_pool = ['192.168.0.1', '192.168.0.2', '192.168.0.3', '192.168.0.4', '192.168.0.5', '192.168.0.6', '192.168.0.7', '192.168.0.8', '192.168.0.9']
```

הסבר:

כאן הגדרנו מאגר של IP ADRRESS כמובן כאן יכולים להגדיר מאגר הרבה יותר גדול (לפי הצורך).

```
# Define a function to assign IP addresses from the pool
def assign_ip():
    if len(ip_pool) > 0:
        ip_address = ip_pool.pop(0)  # Assign the first available IP address from the pool
        return str(ip_address)
    else:
        return None
```

הסבר:

כאן הגדרנו פונקציה מאוד פשוטה שמחזירה לנו את ה ip שהסרבר שולח לקליינט , ואז מוציא אותו מהמאגר .

```
def handle_dhcp_packet(packet):
    if DHCP in packet and packet[DHCP].options[0][1] == 1:
        print("DHCP Discover received")
        dhcp_offer = Ether(src=get_if_hwaddr(conf.iface), dst='ff:ff:ff:ff:ff:ff')/\
                     IP(src='192.168.1.1', dst='192.168.1.100')/\
                     UDP(sport=67, dport=68)/\
                     BOOTP(op=2, yiaddr=assign_ip(), siaddr='192.168.1.1', giaddr='192.168.1.1', xid=packet[BOOTP].xid)/\
                     DHCP(options=[('message-type', 'offer'), ('server_id', '192.168.1.1'), ('subnet_mask', '255.255.255.0'), 'end'])
        sendp(dhcp_offer, iface=conf.iface)
    elif DHCP in packet and packet[DHCP].options[0][1] == 3:
        print("DHCP Request received")
        dhcp_ack = Ether(src=get_if_hwaddr(conf.iface), dst='ff:ff:ff:ff:ff:ff')/ \
                   IP(src='192.168.1.1', dst='192.168.1.100') / \
                   UDP(sport=67, dport=68)/\
                   BOOTP(op=2, yiaddr=assign_ip(), siaddr='192.168.1.1', giaddr='192.168.1.1', xid=packet[BOOTP].xid)/\
                   DHCP(options=[('message-type', 'ack'), ('server_id', '192.168.1.1'), ('subnet_mask', '255.255.255.0'), 'end'])
        sendp(dhcp_ack, iface=conf.iface)
```

הסבר:

כאן בעצם החלק הכבד של הקוד , כאן מקבלים את ה- DICOVER אם הפקיטה היא אכן DISCOVER שולחים OFFER (משמעותו שהסרבר שולח ל CLIENT : יש לי IP רלוונטי , אתה רוצה ?) ואז מחכים לקבל REQUEST (משמעותו הקליינט מחזיר תשובה חיובית)אם קיבל , מכין ACK ושולח אותו לקליינט כמובן שכאן הסרבר שולח לקליינט את ה IP הפנוי . בתוך הפקיטה שזה ה YIADDR בשכבת ה BOOTP .

עכשיו מראים לכם איך הקליינט שלנו בקטנה , ואז נציג לכם איך זה עובד בדיוק דרך פלטים תכנית , ובנוסף הקלטות ויירשארק.

```
def handle_dhcp_packet(packet):
    if DHCP in packet and packet[DHCP].options[0][1] == 2:
        print("DHCP Offer received")
        dhcp_request = Ether(src=get_if_hwaddr(conf.iface), dst='ff:ff:ff:ff:ff:ff')/\
                       IP(src='0.0.0.0', dst='255.255.255.255')/\
                       UDP(sport=68, dport=67)/\
                       BOOTP(op=1, xid=packet[BOOTP].xid, chaddr=packet[BOOTP].chaddr)/\
                       DHCP(options=[('message-type', 'request'), ('requested_addr', packet[BOOTP].yiaddr), ('server_id', '192.168.1.1'), 'end'])
        sendp(dhcp_request, iface=conf.iface)
    elif DHCP in packet and packet[DHCP].options[0][1] == 5:
        print("DHCP Ack received")
        print("Assigned IP address: " + packet[BOOTP].yiaddr)
        return
```

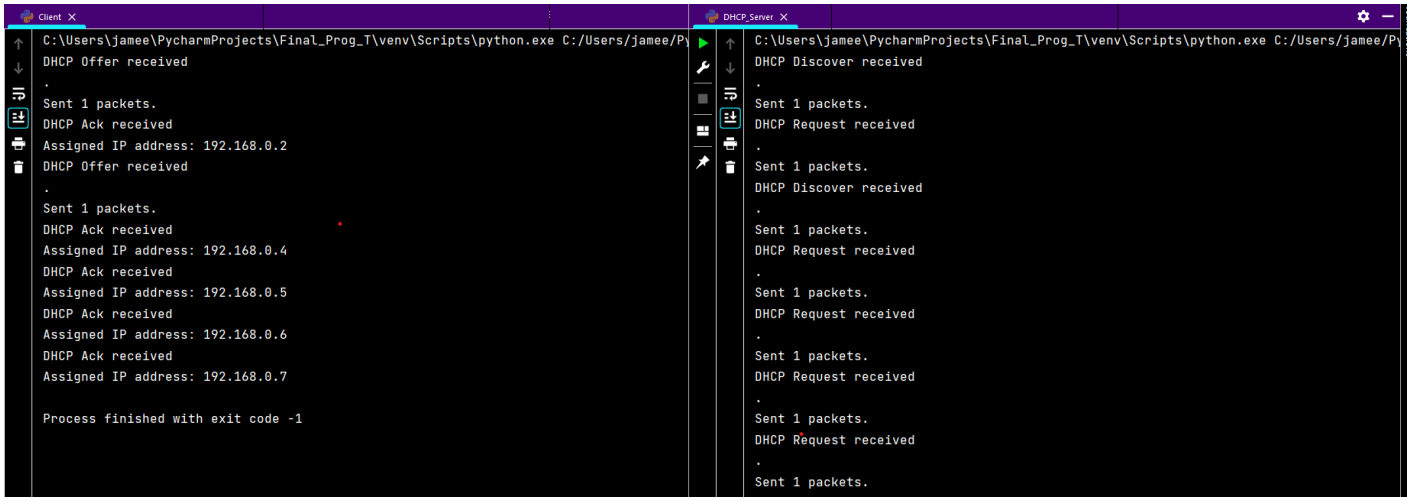
<u>הסבר:</u>

אז כאן זה עובד באותה צורה :

כמובן שה - DISCOVER נשלח ברגע שבו פותחים תקשורת כי כך מוגדרת הספרייה שאנחנו משתמשים בה כאן , מקבל OFFER ממקום מסוים עם IP רלוונטי , מחזיר REQUEST שהוא כן רוצה להתעסק עם ה IP שהוצע עליו , ואז הוא מחכה לקבל את ה ACK שהסרבר אמור לשלוח .

עכשיו מציגים לכם איך בדיוק זה עובד באמצעות פלטי קוד והקלטות ויירשארק :

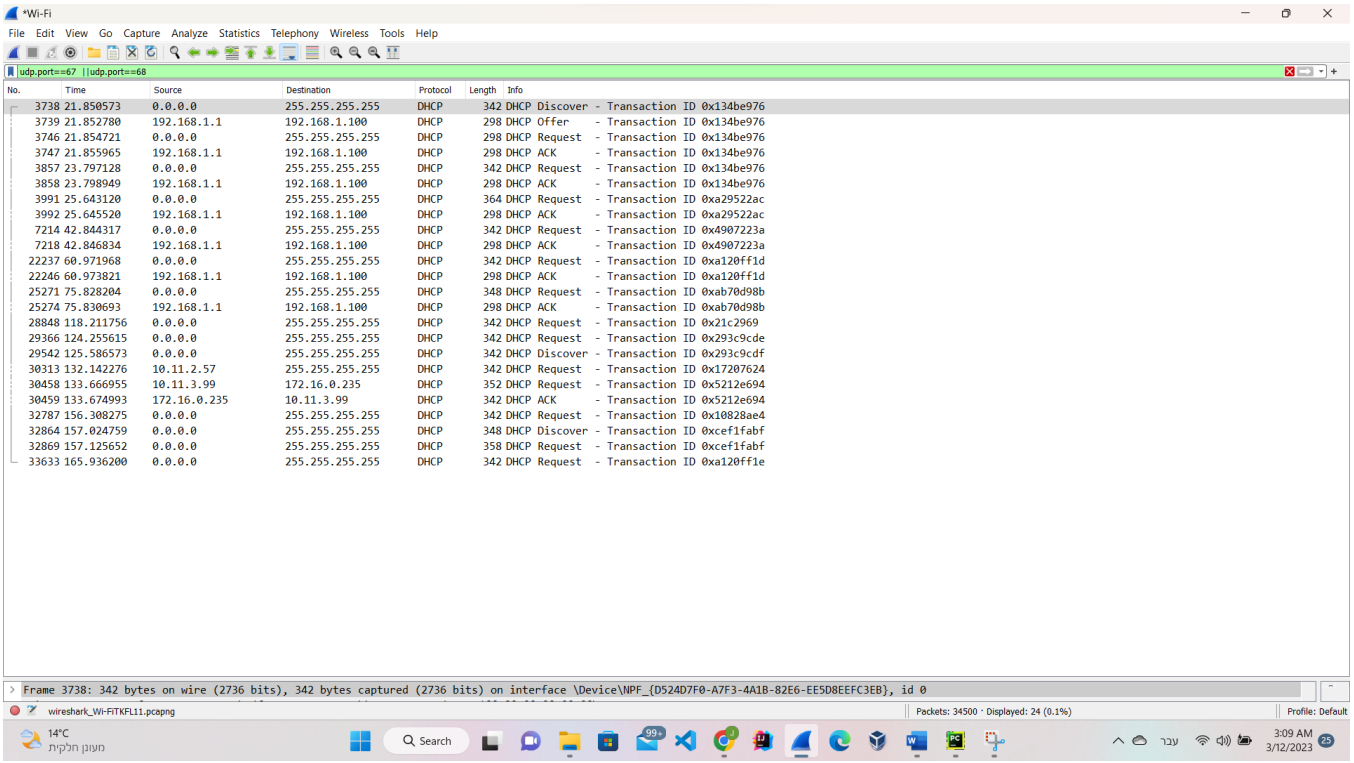
פלטי קוד:



אז כאן שמים לב שבצד ימין יש לנו את השרת ובצד שמאל יש לנו את הקליינט .

נסתכל בהתחלה על השרת , קבל DICOVER ואז שלח פקיטה אחת שזה יופיע בתחילת הקליינט (OFEER) ואז הוא שלח פקיטה שזה ה REQUEST מופיע לנו שזה התקבל בשרת בדיוק שורה אחת אחרי שליחה , סוף סוף השרת שולח ACK שמכיל בתוכו את הכתובת הפנויה במצב שלנו זה "192.168.0.2" , ואז חוזר חלילה על הלולאה .

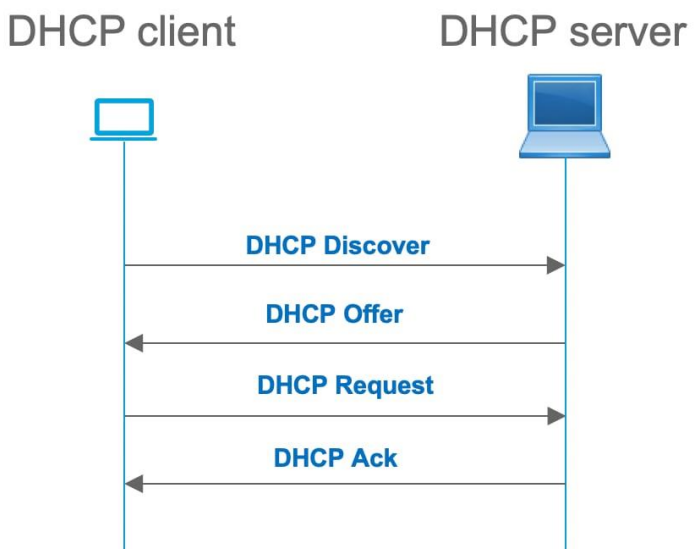
הקלטת ויירשארק :



כאן בקלות יכולים לראות את השלבים של הפרוטוקול

1. לקוח שולח DISCOVER
2. שרת שולח OFFER
3. לקוח שולח REQUEST
4. שרת שולח ACK

לפני שנסגור את החלון של DHCP נזכיר לכם את הדיאגרמה שלו בשביל להוכיח לכם שהתוכנית שלנו עובדת בצורה הנכונה


DHCP client
DHCP server
DHCP Discover
DHCP Offer
DHCP Request
DHCP Ack

DNS local server

בחלק זה, נציג קטעים מהקוד ונסביר אותם. בנוסף, נציג פלטים מהרצות שונות של התוכנית, ונוסיף גם הקלטות של ויירשארק.

נתחיל בקוד:

```
def main():

    # Set local IP address and port number
    local_ip = input("[CLIENT] Enter ip address: ")
    local_port = 1234

    addr = (local_ip, local_port)

    # Choose server to communicate with
    communication_server = input("[CLIENT] Choose a server: ").lower()

    if communication_server == 'ftp':
        ftp_server(addr)
    elif communication_server == 'dns':
        dns_server(addr)
```

הסבר:

זהו קוד מתוך הclient. ראשית, אנחנו מגדירים כתובת ip של מחשב הלקוח, ומגדירים port דרכו תעבור התקשורת שרת-לקוח. לאחר מכן, בהתאם לשרת שהלקוח מבקש ליצור איתו קשר, אנו קוראים לפונקציה הרלוונטית. במקרה שלנו – dns_server().

```python
def dns_server(addr):

    # Create socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    print("[CLIENT] socket created successfully")

    # Sending domain name and receiving match ip address from DNS server
    want_ip = "Y"

    while want_ip.upper() == "Y":

        # Send domain name to server
        domain_name = input("[CLIENT] Enter domain name for which the IP is needed: ")
        sock.sendto(domain_name.encode(), addr)
```

הסבר:

כעת, לאחר שהתבצעה קריאה לפונקציה dns_server(), בתוך הפונקציה, אנחנו יוצרים סוקט, ומתחילים לולאה, שבה כל עוד הלקוח רוצה להמשיך לשלוח domain names ולקבל ip תואם, הלולאה תמשיך.

בתחילת הלולאה הלקוח מכניס את הdomain name, ושולח אותו לשרת DNS ע"י פונקציית sendto. זה המקום לציין שהתקשורת ב DNS מבוסס על פרוטוקול UDP, לכן ביצירת הסוקט הכנסנו SOCK DGRAM והשתמשנו ב **to**send. עוד נציין, שצריך להמיר את הdomain לבייטים לפני השליחה.

```python
def main():

    # Example for pairs of domain names and IP addresses
    dns_table = {"www.google.com": "8.8.8.8",
                 "www.youtube.com": " 208.65.153.238",
                 "www.asus.com": "192.168.2.1", "www.cisco.com": "192.168.0.30"}

    # Creating UDP socket for communication with clients
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    print("[SERVER] Server is working right now..")

    ip_address = input("[SERVER] Enter client ip address: ")
    addr = (ip_address, 1234)

    # Bind server to a port which communication will go through
    sock.bind(addr)

    keep_going = "Y"

    while keep_going.upper() == "Y":

        # DNS is going to receive a domain name from client. The domain will be stor
        # address of sender will be stored in 'address' variable NOTE: 1024 - number
        domain_name, address = sock.recvfrom(1024)
```

הסבר:

זהו קוד מתוך השרת DNS. יש פה רשימה של זוגות – ערכים ומפתחות, כאשר המפתח הוא הdomain, והערך הוא הip המתאים. הרשימה יכולה להיות ארוכה מאוד, אבל הבאנו כמה דוגמאות בשביל המחשת התוכנית.

לאחר מכן, השרת יותר סוקט לשם תקשורת עם לקוחות, ועושה bind לפורט שגם הלקוח יתקשר דרכו.

גם כאן יש לולאה – התלויה בהחלטת הלקוח. כל עוד הלקוח ירצה להמשיך, הלולאה תמשיך. בתחילת הלולאה השרת מקבל ע"י recv**from** (UDP כבר אמרנו?) את הdomain שהלקוח רוצה לתרגם.

```
    print(f"[SERVER] Client wants the IP address of {domain_name}")

    # Fetch the IP address from table, and transfer it to bytes before sending
    match_ip = dns_table.get(domain_name, "Not found!")

    # Sending the match IP address to the sender address
    sock.sendto(match_ip.encode(), address)
    if match_ip == 'Not found!':
        print("[SERVER] IP for this domain doesn't exist")
    else:
        print("[SERVER] Sending...")

    # Receive from client decision if continue or quit
    keep_going, client_address = sock.recvfrom(1024)
    keep_going = keep_going.decode()

# Client decided to quit, closing connection
print("[SERVER] Exit program... ")
sock.close()
```

הסבר:

לאחר שהשרת קיבל את הdomain מהלקוח, הוא מחפש ברשימה שלו האם קיים מפתח עם שם כמו של הdomain. כעת, נקודה חשובה – אם אין ip תואם לdomain שהתקבל, השרת – שהוא שרת DNS מקומי, ישלח בקשה לשרת DNS מעליו בהיררכיה, האם יש אצלו ip תואם לdomain זה. אם גם אצלו אין, השרת הבא ישלח בקשה לשרת מעליו, עד סוף ההיררכיה במקרה הצורך. במידה והכתובת לא קיימת בכלל, תישלח ללקוח הודעה: Not found.

לבסוף, השרת ימתין להחלטה מצד הלקוח, האם להמשיך (keep_going) או לצאת.

```
    # Receive answer from server
    ip, address = sock.recvfrom(1024)

    reply_ip = ip.decode()

    print(f"[CLIENT] The IP for the domain name {domain_name} : {reply_ip}")

    want_ip = input("[CLIENT] Do you want to continue? Y/N: ").upper()

    # Update the server if continue or quit
    if want_ip == 'N':
        sock.sendto("N".encode(), addr)
        print("[CLIENT] Exit program...")
    else:
        sock.sendto("Y".encode(), addr)

# Done, closing connection
sock.close()
```

הסבר:

חזרנו לקוד אצל הלקוח. הלקוח מקבל ע"י recv**from** את הip המבוקש, מפענח אותו (decode) ומדפיס אותו.

כעת, הלקוח יצטרך להחליט האם הוא ממשיך או לא. ההחלטה נשלחת לשרת, שכאמור, ממתין לתגובה.

פלטים:

כעת, נציג פלטים מהרצה של התוכנית.

```
Run: DNS_server  client (1)
C:\Users\Owner\PycharmProjects\FinalProject\venv\Scripts\python.exe C:\Users\Owner\PycharmProjects\FinalProject\client.py
[CLIENT] Enter ip address: 127.0.0.1
[CLIENT] Choose a server: dns
[CLIENT] socket created successfully
[CLIENT] Enter domain name for which the IP is needed: www.google.com
[CLIENT] The IP for the domain name www.google.com : 8.8.8.8
[CLIENT] Do you want to continue? Y/N: Y
[CLIENT] Enter domain name for which the IP is needed: pizza
[CLIENT] The IP for the domain name pizza : Not found!
[CLIENT] Do you want to continue? Y/N: y
[CLIENT] Enter domain name for which the IP is needed: www.asus.com
[CLIENT] The IP for the domain name www.asus.com : 192.168.2.1
[CLIENT] Do you want to continue? Y/N: N
[CLIENT] Exit program...

Process finished with exit code 0
```

```
DNS_server  client (1)
C:\Users\Owner\PycharmProjects\FinalProject\venv\Scripts\python.exe C:\Users\Owner\PycharmProjects\FinalProject\DNS_server.py
[SERVER] Server is working right now..
[SERVER] Enter client ip address: 127.0.0.1
[SERVER] Client wants the IP address of www.google.com
[SERVER] Sending...
[SERVER] Client wants the IP address of pizza
[SERVER] IP for this domain doesn't exist
[SERVER] Client wants the IP address of www.asus.com
[SERVER] Sending...
[SERVER] Exit program...

Process finished with exit code 0
```

הסבר:

הכנסנו ללקוח ולשרת את כתובת הip של הlocal host – כלומר המחשב יתקשר עם עצמו.

בחלון של הלקוח בחרנו בשרת DNS לתקשורת.

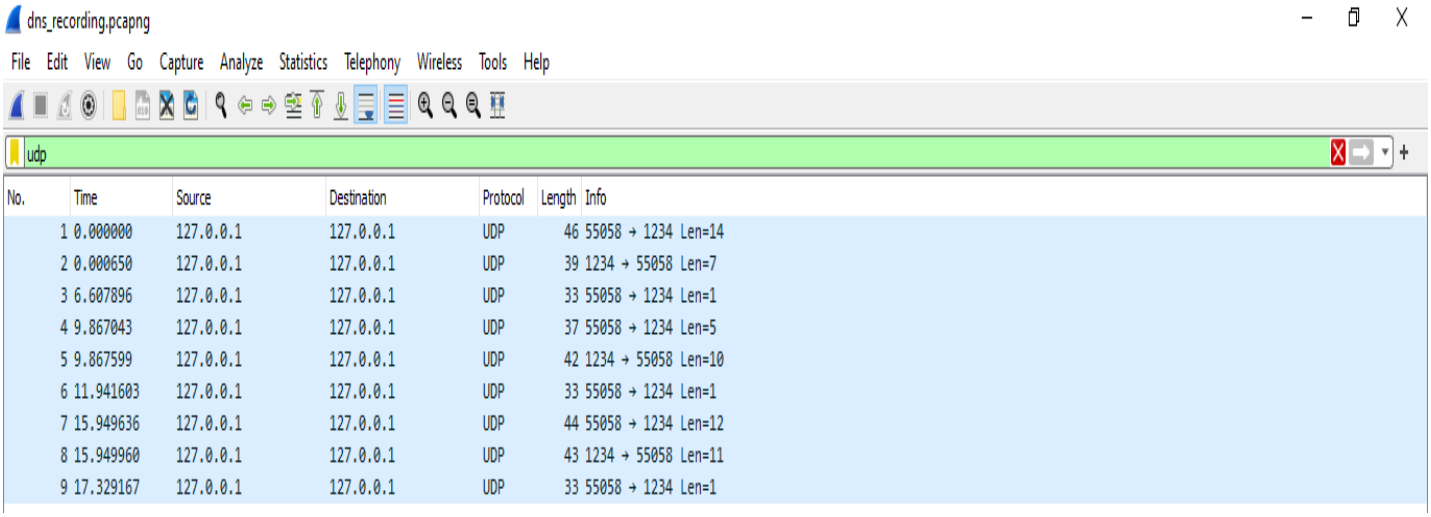
לאחר מכן ביקשנו לקבל את הip של הdomain: www.google.com השרת שלח לנו אותה – 8.8.8.8

ביקשנו להמשיך את התקשורת ואז הכנסנו domain לא קיים – pizza. השרת שלח הודעה שלא נמצא ip לdomain זה.

ביקשנו ip עבור domain קיים, קיבלנו אותו, וביקשנו להפסיק. התוכנית נסגרה בשתי הקצוות.

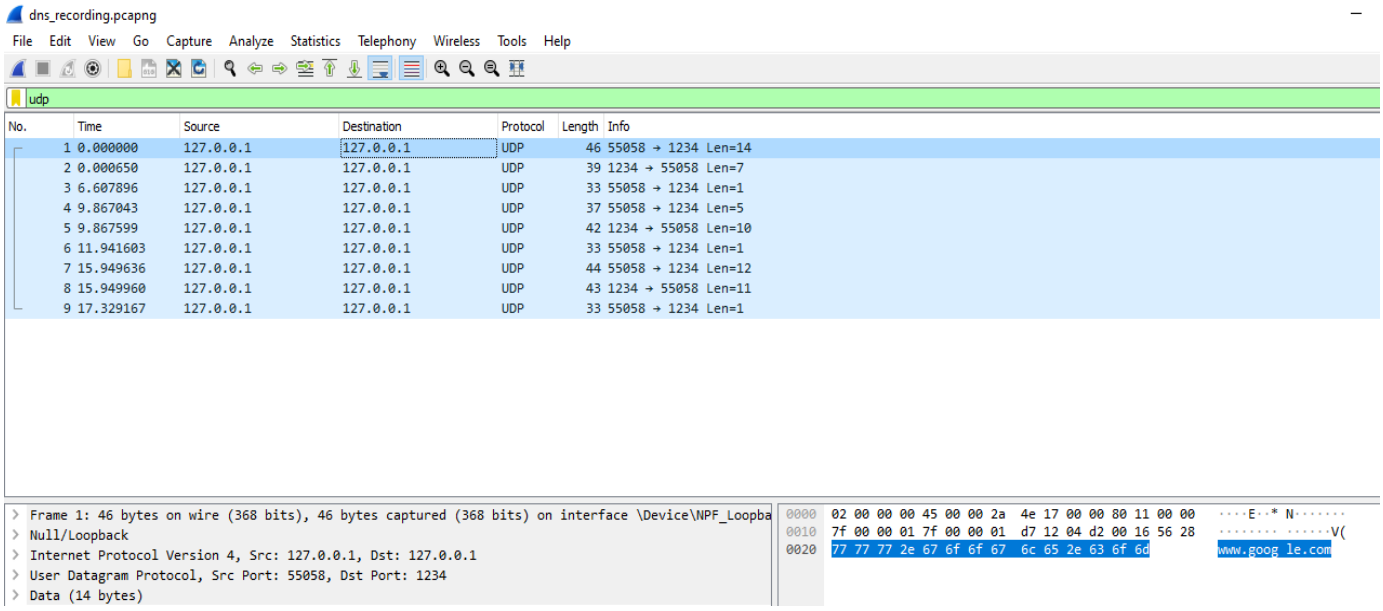
הקלטות wireshark:

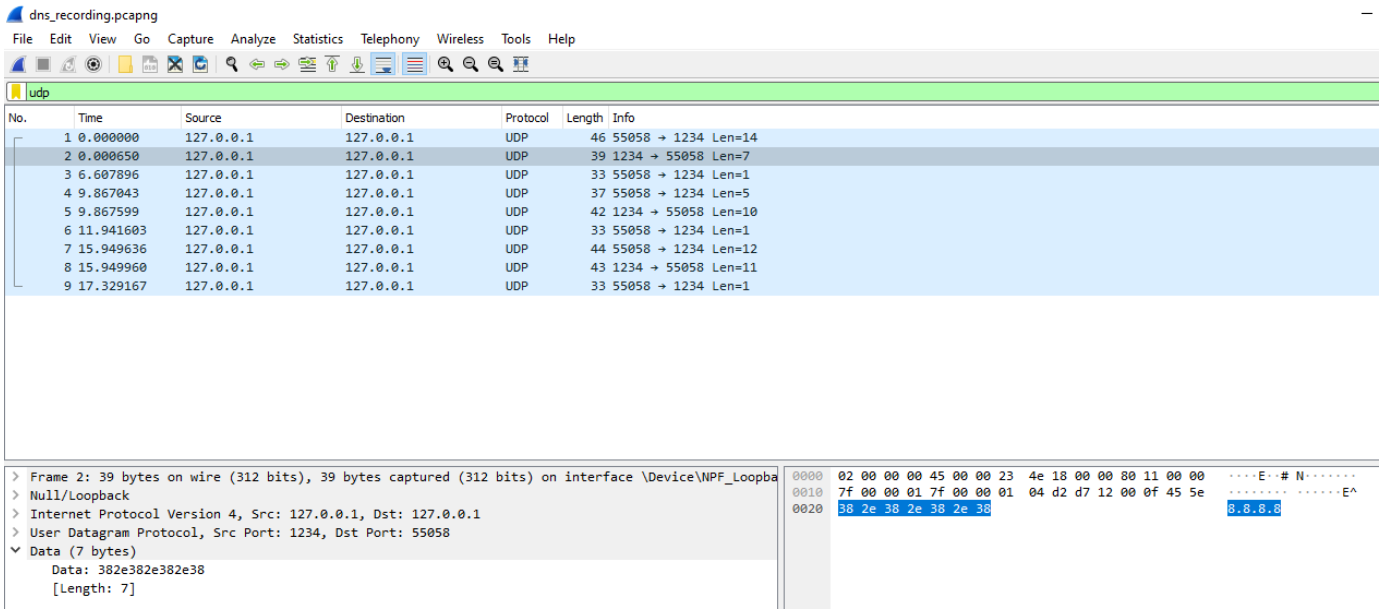
כעת, נציג הקלטות ויירשארק של התעבורה.



הסבר:

כך נראית ההקלטה של ההרצה. ניתן לראות תחת הקטגוריה protocol שהפרוטוקול הוא UDP, כמו שאמרנו. כתובת הip של המקור והיעד היא 127.0.0.1 כי המחשב מדבר עם עצמו. בנוסף ניתן לראות בצד את הפורט שהגדרנו 1234.





הסבר:

בתמונה הראשונה, לחצנו על הפאקטת בקשה – ניתן לראות למטה מימין שתוכן הפאקטה בין היתר הוא הdomain www.google.com.

ואכן בתמונה השנייה, בפאקטת התשובה, ניתן לראות מימין למטה את התשובה של השרת – 8.8.8.8

יש עוד פרטים על כל פאקטה, ההקלטה מצורפת לקבצי הפרויקט.

No.	Time	Source	Destination	Protocol	Length	Info
13	18.614251	127.0.0.1	127.0.0.1	UDP	33	53893 → 1234 Len=1
14	21.994656	127.0.0.1	127.0.0.1	UDP	37	53893 → 1234 Len=5
15	21.995230	192.168.244.119	192.168.244.255	NBNS	82	Name query NB PIZZA<00>
16	21.995724	172.25.240.1	224.0.0.251	MDNS	61	Standard query 0x0000 A pizza.local, "QM" question
17	21.996214	192.168.56.1	224.0.0.251	MDNS	61	Standard query 0x0000 A pizza.local, "QM" question
18	21.996512	192.168.244.119	224.0.0.251	MDNS	61	Standard query 0x0000 A pizza.local, "QM" question
19	21.996893	fe80::163e:4b6:e11c…	ff02::fb	MDNS	81	Standard query 0x0000 A pizza.local, "QM" question
20	21.997241	fe80::de19:56c6:844…	ff02::fb	MDNS	81	Standard query 0x0000 A pizza.local, "QM" question
21	21.997524	fe80::4012:64ba:460…	ff02::fb	MDNS	81	Standard query 0x0000 A pizza.local, "QM" question
22	21.998433	fe80::4012:64ba:460…	ff02::1:3	LLMNR	75	Standard query 0x5fdc A pizza
23	21.998610	192.168.244.119	224.0.0.252	LLMNR	55	Standard query 0x5fdc A pizza
24	21.998902	172.25.240.1	172.25.255.255	NBNS	82	Name query NB PIZZA<00>
25	21.999125	192.168.56.1	192.168.56.255	NBNS	82	Name query NB PIZZA<00>
26	21.999283	172.25.240.1	224.0.0.251	MDNS	61	Standard query 0x0000 A pizza.local, "QM" question
27	21.999478	192.168.56.1	224.0.0.251	MDNS	61	Standard query 0x0000 A pizza.local, "QM" question
28	21.999648	192.168.244.119	224.0.0.251	MDNS	61	Standard query 0x0000 A pizza.local, "QM" question
29	21.999874	fe80::163e:4b6:e11c…	ff02::fb	MDNS	81	Standard query 0x0000 A pizza.local, "QM" question
30	22.000678	fe80::de19:56c6:844…	ff02::fb	MDNS	81	Standard query 0x0000 A pizza.local, "QM" question
31	22.000868	fe80::4012:64ba:460…	ff02::fb	MDNS	81	Standard query 0x0000 A pizza.local, "QM" question
32	22.001458	fe80::de19:56c6:844…	ff02::1:3	LLMNR	75	Standard query 0xd440 A pizza
33	22.001578	192.168.56.1	224.0.0.252	LLMNR	55	Standard query 0xd440 A pizza
34	22.410530	fe80::4012:64ba:460…	ff02::1:3	LLMNR	75	Standard query 0x5fdc A pizza
35	22.410661	192.168.244.119	224.0.0.252	LLMNR	55	Standard query 0x5fdc A pizza
36	22.426599	fe80::de19:56c6:844…	ff02::1:3	LLMNR	75	Standard query 0xd440 A pizza
37	22.426665	192.168.56.1	224.0.0.252	LLMNR	55	Standard query 0xd440 A pizza

```
> Frame 14: 37 bytes on wire (296 bits), 37 bytes captured (296 bits) on interface \Device\NPF_Loopb
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 53893, Dst Port: 1234
> Data (5 bytes)
```

```
0000  02 00 00 00 45 00 00 21  15 b7 00 00 80 11 00 00   ····E··!  ····· ····
0010  7f 00 00 01 7f 00 00 01  d2 85 04 d2 00 0d de 95   ····· ····· ····· ····
0020  70 69 7a 7a 61                                     pizza
```

הסבר:

בהקלטה זו, מומחש השימוש בהיררכיית שרתי DNS.

כתובת שאינה נמצאת בDNS המקומי, נשלחת ע"י השרת המקומי לשרת ברמה אחת מעליו בהיררכיה, שגם לא מוצא אצלו כתובת ip תואמת לpizza. וכך הלאה בהיררכיה, עד שאף אחד לא מוצא תשובה, ולכן נשלחת ללקוח הודעה Not found.

FTP – Application Server

את השלב הזה אנחנו מבצעים בשתי גרסאות , גרסה שעובדת עם פרוטוקול TCP שזה פרוטוקול אמין ולכן העבודה כאן יחסית קלה , כי הפרוטוקול הזה מספק לנו שליחה אמינה , עם קצב בהתאם , הפקיטות מגיעות לפי הסדר הנכון , בלי איבוד ...
כל זה הפרוטוקול נותן לנו , לעומת זאת הגרסה השנייה היא הרבה יותר קשוחה שזה ה RUDP ושם טיפלנו בכל מה שאמרנו לעיל **בעצמנו** כי הפרוטוקול הזה לא נותן לנו את זה על כפית זהב כמו ה TCP.

נתחיל אתכם עם הגרסה הראשונה הקלה יחסית , נציג לכם קטעי קוד , קלטים , פלטים , ועוד הסברים שנוודא שהמסר שלנו הגיע לכם בצורה הברורה ביותר , וכתוצאה מכך תוכלו להשתמש בקלות באפליקציה שלנו 😊

אז בוא נסביר לכם קודם כל מה זה ה FTP בקטנה:

FTP קיצור של פרוטוקול העברת קבצים. זהו פרוטוקול רשת סטנדרטי המשמש להעברת קבצים בין שרתים ולקוחות דרך האינטרנט. FTP מאפשר למשתמשים להעלות ולהוריד קבצים משרת באמצעות שיטות אימות שונות, כולל שמות משתמש וסיסמאות, מפתחות ציבוריים ואישורים.

FTP משתמש בשני ערוצים: ערוץ פקודה וערוץ נתונים. ערוץ הפקודה משמש לשליחת פקודות בין הלקוח לשרת, בעוד ערוץ הנתונים משמש להעברת הקבצים בפועל.

TCP – version

תזכורת קטנה על TCP :

TCP ראשי תיבות של Transmission Control Protocol. זהו אחד מפרוטוקולי הליבה של חבילת פרוטוקול האינטרנט (IP), האחראית על העברת נתונים בין מחשבים דרך האינטרנט.

TCP מספק מסירה אמינה, מסודרת ונבדקת שגיאות של נתונים בין יישומים הפועלים על מארחים נפרדים. הוא משיג זאת על ידי יצירת חיבור וירטואלי בין שני מארחים, חלוקת הנתונים לחבילות, שליחת החבילות דרך האינטרנט והרכבה מחדש בסדר הנכון בקצה המקבל.

TCP משתמש בלחיצת יד תלת כיוונית כדי ליצור חיבור בין מארחים. לאחר יצירת החיבור, ניתן להעביר נתונים באופן דו-כיווני בין המארחים. TCP מספקת גם מנגנוני בקרת זרימה, בקרת גודש ומנגנוני זיהוי ושחזור שגיאות כדי להבטיח שהנתונים מועברים ביעילות ובאמינות.

TCP נמצא בשימוש נרחב על ידי יישומים הדורשים העברת נתונים אמינה, כגון דואר אלקטרוני, העברת קבצים וגלישה באינטרנט.

כעת נעבור לקוד:

```
def main():

    # Create a directory that will contain all the server files
    directory_name = input("[SERVER] Enter directory path for all of server's files")
    if not os.path.exists(directory_name):
        os.makedirs(directory_name)

    # Client address
    client_ip_address = input("[SERVER] Enter client ip address: ")
    addr = (client_ip_address, 1234)

    # Temporary socket to get updated by client which protocol to implement
    temp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    temp_sock.bind(addr)

    protocol, address = temp_sock.recvfrom(1024)

    protocol = protocol.decode()
```

הסבר:

ראשית, אנו יוצרים תיקייה שבה יהיו כל קבצי השרת.

לאחר מכן, מבקשים מהלקוח שיזין את כתובת הip שלו (ככה אנחנו גמישים בקוד, לא מקובעים על כתובת יחידה). יוצרים סוקט שדרכו תועבר ההחלטה באיזה פרוטוקול השרת ישתמש – TCP או RUDP.

```python
    if protocol == 'tcp':
        # Create tcp connection
        connection_socket = tcp_connection(addr)
        # Receive username and password and check if valid
        packet = connection_socket.recv(512).decode()
        validation(connection_socket, packet)
        # Execute client command
        choose_command(connection_socket, directory_name)
    elif protocol == 'rudp':
        connection_socket = rudp_connection()
    else:
        message = "You've typed a wrong protocol.."
        temp_sock.sendto(message.encode(), addr)
        print("[SERVER] Wrong protocol, shutting down...")
    # ---------------------------------------------------------


if __name__ == "__main__":
    main()
```

הסבר:

אם הפרוטוקול שנבחר הוא TCP – מה שאנו מראים כרגע, נבצע סדרת פעולות, נעבור עליהם ונציג אחת אחת כעת.

```python
def tcp_connection(addr):
    # create a socket object
    ftp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind the socket to a specific address and port
    ftp_socket.bind(addr)

    # Listen for incoming connections
    ftp_socket.listen(1)
    print('[Server] Server is listening...')

    # Accept incoming connections
    connection_socket, client_address = ftp_socket.accept()
    print(f'[Server] Accepted connection from {client_address[0]}')

    return connection_socket
```

הסבר:

הפעולה הראשונה – פתיחת קשר TCP. בהתחלה יוצרים סוקט, קושרים אותו לפורט דרכו תהיה התקשורת, מגדירים אותו שיקשיב ללקוח אחד בפעם, ומבצעים accept- אם תהיה בקשת connect מצד לקוח, הaccpet תקבל אותה.

```
def validation(connection_socket, packet):

    # Receive formal data from client (limited to 1024 bytes)
    client_username = packet.split()[0]
    client_password = packet.split()[1]

    # Validation check
    if not (client_username == 'aj123' and client_password == '2023a'):
        connection_socket.send("ERROR".encode())
        print("[SERVER] Incorrect username or password! exit program...")
        sys.exit()
    else:
        print("[SERVER] Username and password are correct")
        connection_socket.send("OK".encode())
```

הסבר:

הפעולה הבאה – אימות שם משתמש וסיסמא. הלקוח שולח שם משתמש וסיסמא והשרת בודק אם הם תואמים לפרטים שניתנו לו מראש. השרת ישלח ללקוח הודעה לגבי הפרטים – אם נכונים או לא.

```
def choose_command(connection_socket, directory_name):

    command = connection_socket.recv(512).decode()

    if command == "upload":
        upload(connection_socket, directory_name)
    elif command == "download":
        download(connection_socket)
    elif command == "list":
        list_of_files(connection_socket, directory_name)
    elif command == "close":
        close(connection_socket)
    # If the received command isn't valid, the client will fix it
```

הסבר:

הפעולה הבאה – בחירת פקודה. הלקוח אומר לשרת מה הוא רוצה לעשות, ובהתאם לפקודה נעשית קריאה לפונקציה של אותה פקודה.

לפני שנראה את הפקודות, נראה את מה שקרה עד עכשיו מנקודת המבט של הלקוח:

```
def main():

    # Set local IP address and port number
    local_ip = input("[CLIENT] Enter ip address: ")
    local_port = 1234

    addr = (local_ip, local_port)

    # Choose server to communicate with
    communication_server = input("[CLIENT] Choose a server: ").lower()

    if communication_server == 'ftp':
        ftp_server(addr)
```

הסבר:

הלקוח יכניס את הip שלו, ויבחר באיזה שירות הוא מעוניין (FTP במקרה שלנו).
תתבצע קריאה לפונקציה של FTP:

```
def ftp_server(addr):

    # Socket for deliver decisions about protocol type
    temp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Choose protocol and update the server
    protocol = input("[CLIENT] Choose communication protocol, TCP or RUDP: ").lower()
    temp_sock.sendto(protocol.encode(), addr)

    if protocol == 'tcp':
        tcp_connection_ftp(addr)
    elif protocol == 'rudp':
        rudp_connection_ftp(addr)
    else:
        print("[CLIENT] Wrong protocol, exit program...")
```

הסבר:

בפונקציה זו, הלקוח בוחר באיזה פרוטוקול FTP ישתמש – TCP או RUDP.

הלקוח בחר TCP, ניגש לפונקציה המתאימה.

```python
def tcp_connection_ftp(addr):

    # create a TCP socket and bind it to the local IP address and port
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print("[CLIENT] Socket created successfully")

    # Connect to server
    sock.connect(addr)
    print("[CLIENT] Socket connected successfully")

    # Send some formal details to server
    user_name = input("[CLIENT] Enter UserName: ")
    password = input("[CLIENT] Enter password: ")

    data = user_name + " " + password

    sock.send(data.encode())
    print("[CLIENT] Username and password were sent successfully, waiting for OK message from server...")

    # sleep, for visual effect only
    time.sleep(1)
```

```python
    # Wait to get OK message from server
    validation = sock.recv(512).decode()
    if validation == 'ERROR':
        sock.close()
        print("[CLIENT] Got an error message, exit program...")
        sys.exit()
    else:
        print("[CLIENT] Got an OK message from server")

    # Choose command
    command_type = input("[CLIENT] Enter command type: ")

    # Send command type to server
    sock.send(command_type.encode())

    if command_type == "upload":
        upload(sock)
    elif command_type == "download":
        download(sock)
    elif command_type == "list":
        list_of_files(sock)
    else:
        print("[CLIENT] Invalid command, exit program...")

    # Close the connection and socket
    sock.close()
```

הסבר:

סדר הפעולות:

יצירת קשר TCP מצד הלקוח. כלומר יצירת סוקט ובקשת connect. לאחר שהשרת יעשה accept, יודפס שהלקוח התחבר בהצלחה.

לאחר מכן, לצורך אימות, הלקוח יכניס שם משתמש וסיסמא וישלח אותם לשרת.

הלקוח מקבל הודעה מהשרת – או שגיאה במידה והפרטים לא נכונים, או OK.

לבסוף, הלקוח יבחר איזה פקודה הוא רוצה לעשות: upload, download, list

Upload – הכוונה שהלקוח רוצה להעלות קובץ לתיקיית קבצי השרת.

Download – הלקוח רוצה להוריד קובץ מתיקיית השרת.

List – הלקוח רוצה לראות רשימה של קבצי תיקיית השרת.

```
# Upload command: #
def upload(connection_socket, directory_name):

    # Receive file name from client
    file_name_received = connection_socket.recv(512).decode()
    last_backslash_index = file_name_received.rfind('\\')
    file_name = file_name_received[last_backslash_index + 1:]

    # Reset variable that will continue the packet with file
    file_to_upload = bytes()

    # Receive the file to upload
    while True:
        upload_chunk = connection_socket.recv(1024)
        file_to_upload += upload_chunk
        if not upload_chunk:
            break

    file_path = directory_name + "\\" + file_name

# Check if file is already exists, and create it (if not exist or need to be replaced)
if os.path.isfile(file_path):
    print(f"[SERVER] '{file_name}' is already exists, do you want to replace it?")
    replace = input("Y/N: ")
    if replace == 'Y':
        print("[SERVER] Replacing file...")
        with open(file_path, 'wb') as new_file:
            new_file.write(file_to_upload)
    else:
        print("[SERVER] Bye Bye...")
else:
    print("[SERVER] Uploading a new file...")
    with open(file_path, 'wb') as new_file:
        new_file.write(file_to_upload)
```

הסבר:

חזרנו לשרת. הפונקצייה upload נראית כך:

ראשית מקבלים מהלקוח בפאקטה נפרדת את שם הקובץ שהוא רוצה להעלות.

לאחר מכן, מקבלים בצ'אנקים את הקובץ (הוא יכול להיות כבד).

השרת בודק – אם הקובץ כבר קיים, הוא ישאל את הלקוח אם הוא רוצה להחליף אותו. אחרת – ייצור קובץ חדש.

```
def download(connection_socket):

    file_name_for_download = connection_socket.recv(512).decode()

    # Check if requested file exists in server directory, and download it (if exist)
    if not os.path.isfile(file_name_for_download):
        print("[SERVER] Sorry, requested file doesn't exist :(")
    else:
        with open(file_name_for_download, 'rb') as f:
            download_file = f.read()

        print("[SERVER] Downloading requested file...")
        packet_size = len(download_file)
        chunk = 1024
        i = 0
        while packet_size > 0:
            connection_socket.send(download_file[i:i + chunk])
            i = i + chunk
            packet_size -= chunk
```

הסבר:

הפונקצייה download נראית כך:

מקבלים מהלקוח את שם הקובץ שהוא רוצה להוריד.

השרת בודק אם בכלל קיים קובץ כזה בתיקייה שלו – אם לא, הוא ישלח הודעת התנצלות ללקוח.

אם קיים כזה קובץ, השרת ישלח בצ'אנקים ללקוח את הקובץ. (עוד מעט נראה איך זה מצד הלקוח).

```python
# List command: #
def list_of_files(connection_socket, directory_name):

    # Get the list from operating system
    files_list = os.listdir(directory_name)

    # Convert the list to bytes by 'dumps' function
    encoded_list = json.dumps(files_list).encode()

    # Sending the list
    print("[SERVER] Sending the list of files...")
    list_packet_size = len(encoded_list)
    list_chunk = 1024
    i = 0
    while list_packet_size > 0:
        connection_socket.send(encoded_list[i:i + list_chunk])
        i = i + list_chunk
        list_packet_size -= list_chunk
```

הסבר:

הפונקצייה list נראית כך:

מבקשים ממערכת ההפעלה את רשימת הקבצים בתיקייה.

מעבירים את הרשימה לתצוגה של בייטים כדי שיהיה אפשר לשלוח אותה.

לבסוף, שולחים את הרשימה בצ'אנקים (אולי הרשימה ממש ממש גדולה...)

נראה כעת את הפונקציות מנקודת מבטו של הלקוח, ונסיים בזה את חלק הקוד:

```python
# upload command #
def upload(sock):

    # Take care about file:
    with open(input("[CLIENT] Choose a file to upload: "), 'rb') as f:
        file = f.read()
        file_name = f.name.encode()

    # sending file name before sending file content
    sock.send(file_name)

    packet_size = len(file)
    upload_chunk = 1024
    i = 0
    while packet_size > 0:
        sock.send(file[i:i+upload_chunk])
        i = i + upload_chunk
        packet_size -= upload_chunk

    print("[CLIENT] File to upload sent successfully")
```

הסבר:

אצל הלקוח, פונקציית upload נראית כך:

פותחים את הקובץ שרוצים להעלות, בתצוגה בינארית, ומחלקים אותו לתוכן ולשם הקובץ.

שולחים קודם כל בנפרד את שם הקובץ לשרת. לאחר מכן שולחים בצ'אנקים את הקובץ.

```
# download command #
def download(sock):

    file_path_for_download = input("[CLIENT] Enter file path for download: ")
    sock.send(file_path_for_download.encode())

    # Receive the file from server:
    downloaded_file = bytes()

    while True:
        chunk = sock.recv(1024)
        if not chunk:
            break
        downloaded_file += chunk

    # Choose a folder to place the file in
    folder = input("[CLIENT] Choose a folder to place the file: ")
    last_backslash_index = file_path_for_download.rfind('\\')
    downloaded_file_name = file_path_for_download[last_backslash_index + 1:]
    downloaded_file_path = folder + "\\" + downloaded_file_name
```

```
    # Check if file is already exists, and create it (if not exist or need to be replaced)
    if os.path.isfile(downloaded_file_path):
        print(f"[CLIENT] '{downloaded_file_name}' is already exists, do you want to replace it?")
        replace = input("Y/N: ")
        if replace == 'Y':
            print("[CLIENT] Replacing file...")
            with open(downloaded_file_path, 'wb') as new_file:
                new_file.write(downloaded_file)
        else:
            print("[CLIENT] Bye Bye...")
    else:
        print(f"[CLIENT] Downloading {downloaded_file_name} ...")
        with open(downloaded_file_path, 'wb') as new_file:
            new_file.write(downloaded_file)
```

הסבר:

אצל הלקוח, פונקציית download נראית כך:

הלקוח מתבקש להכניס את שם הקובץ. שולח את השם לשרת. לאחר מכן השרת שולח לו את הקובץ (במידה וקיים) בצ'אנקים.

הלקוח בוחר איפה לשים על המחשב את הקובץ שהתקבל. במידה והקובץ כבר קיים, הוא יחליט אם להחליף אותו.

```
# List command #
def list_of_files(sock):

    # Receive the list from server:
    server_list = bytes()

    while True:
        list_chunk = sock.recv(1024)
        if not list_chunk:
            break
        server_list += list_chunk

    print("[CLIENT] List of files on FTP server directory:")
    print(server_list.decode())
```

הסבר:

אצל הלקוח, פונקציית list נראית כך:

הלקוח מקבל את הרשימה בצ'אנקים, ואז מדפיס אותה למסך.

כעת נציג פלטי הרצה:

```
C:\Users\Owner\PycharmProjects\FinalProject\venv\Scripts\python.exe C:\Users\Owner\PycharmProjects\FinalProject\client.py
[CLIENT] Enter ip address: 127.0.0.1
[CLIENT] Choose a server: ftp
[CLIENT] Choose communication protocol, TCP or RUDP: tcp
[CLIENT] Socket created successfully
[CLIENT] Socket connected successfully
[CLIENT] Enter UserName: aj123
[CLIENT] Enter password: 2023a
[CLIENT] Username and password were sent successfully, waiting for OK message from server...
[CLIENT] Got an OK message from server
[CLIENT] Enter command type: upload
[CLIENT] Choose a file to upload: C:\Users\Owner\PycharmProjects\FinalProject\check1.txt
[CLIENT] File to upload sent successfully
```

```
C:\Users\Owner\PycharmProjects\FinalProject\venv\Scripts\python.exe C:\Users\Owner\PycharmProjects\FinalProject\FTP_Server.py
[SERVER] Enter directory path for all of server's filesC:\Users\Owner\PycharmProjects\FinalProject\server_files
[SERVER] Enter client ip address: 127.0.0.1
[Server] Server is listening...
[Server] Accepted connection from 127.0.0.1
[SERVER] Username and password are correct
[SERVER] Uploading a new file...
```

הסבר:

זוהי הרצה של פקודת upload. ניתן לראות שהלקוח והשרת יוצרים את הסוקט ואת התקשורת. הלקוח שולח את פרטי האימות ומקבל אישור מהשרת. לאחר מכן הלקוח בוחר בפקודת upload ובקובץ אותו הוא רוצה להעלות, והשרת מעלה את הקובץ לתיקייה.

```
[CLIENT] Enter command type: download
[CLIENT] Enter file path for download: C:\Users\Owner\PycharmProjects\FinalProject\server_files\check1.txt
[CLIENT] Choose a folder to place the file: C:\Users\Owner\Desktop
[CLIENT] Downloading check1.txt ...
```

```
[SERVER] Downloading requested file...
```

הסבר:

זוהי הרצה של פקודת download. השרת בוחר קובץ להורדה ותיקייה אליה הקובץ יירד.

השרת מעדכן שהוא מוריד את הקובץ.

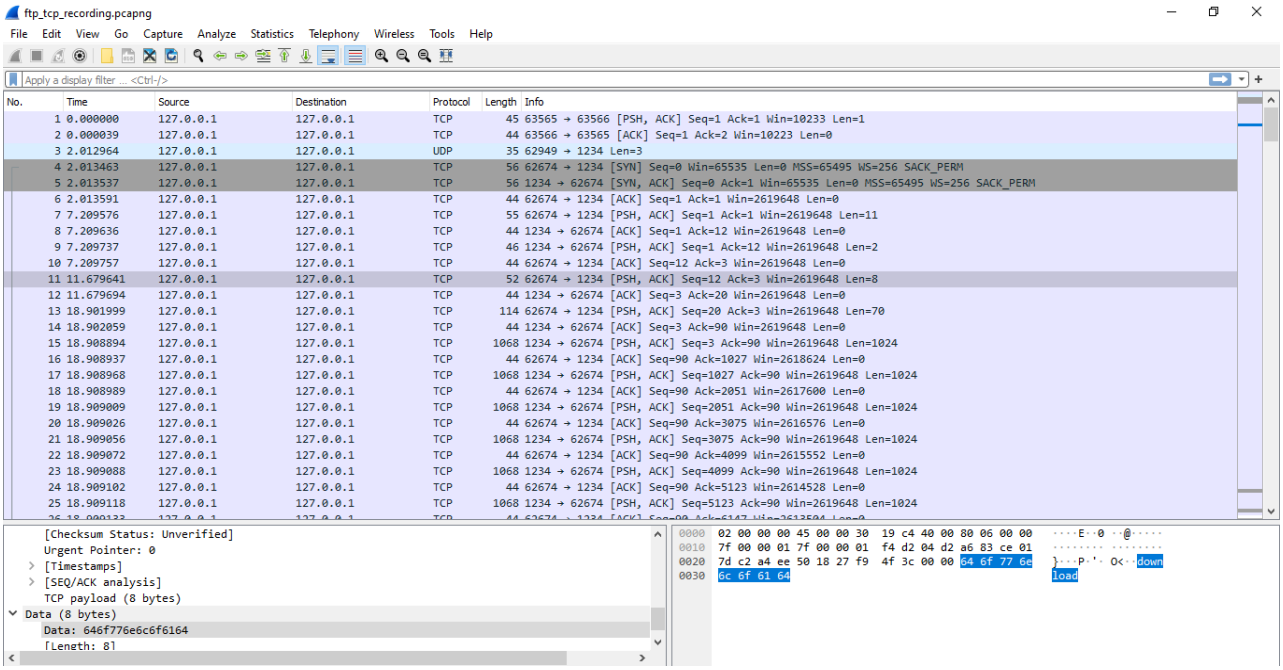
```
[CLIENT] Enter command type: list
[CLIENT] List of files on FTP server directory:
["check1.txt", "main.py", "RealTest.jpeg"]
```

```
[SERVER] Sending the list of files...
```

הסבר:

זוהי הרצה של פקודת list. השרת מעדכן ששולח את רשימת הקבצים בתיקייה, והלקוח מדפיס את הרשימה למסך.

כעת נציג הקלטות ויירשארק:



הסבר:

אפשר לראות שהתקשורת היא בפרוטוקול TCP.

הדגשנו את הפאקטה שבה הלקוח מבקש לבצע את פקודת download.

ההקלטה די ארוכה כי הלקוח ביקש להוריד תמונה ולא קובץ טקסט קצר.

ההקלטה מצורפת כמובן לתיקייה של הפרויקט.

RUDP-Version

אז כאן בעצם הגענו לחלק האחרון , והכי מעניין באפליקציה שלנו , שזה הגרסה של ה RUDP , אז בוא נסביר לכם בקטנה מה זה RUDP בכלל:

ראשי תיבות של Reliable User Datagram Protocol. זהו פרוטוקול שכבת תחבורה המשמש להעברת נתונים אמינה בין יישומים דרך רשת IP. RUDP מבוסס על פרוטוקול User Datagram (UDP), שהוא פרוטוקול חסר חיבור המספק העברת נתונים לא אמינה.

RUDP מוסיף תכונות אמינות על גבי UDP על ידי הטמעת מנגנונים לזיהוי שגיאות, רצף מנות, שידור חוזר ובקרת זרימה. תכונות אלו מבטיחות שהעברת נתונים מהימנה ויעילה על גבי הרשת, אפילו בנוכחות אובדן מנות, עיכובים וגודש.

ועכשיו אנחנו מניחים שאתם יודעים איפה מתרחש הקושי שלנו בשלב הזה , אבל אל דאגה טיפלנו בכל מה שצריך וסיפקנו לכם פרוטוקול RUDP שהוא אמין , יכול להתמודד עם עומס ויכול להתמודד עם איבוד חבילות , לשם כך הוספנו כמה מנגנונים מאוד מעניינים , ובנינו את הפקיטה שלנו עם הוספת מאפיינים לפי הצורך

אז בוא נתחיל בקוד ואז נראה לכם כל מני הקלטות , קלטים , פלטים בשביל להראות לכם איך זה עובד , זה בכלל לא מסובך , אל דאגה אנחנו מפרטים את הכל לפי הצורך😊.

הקוד:

```
elif protocol == 'rudp':

    print("[CLIENT] Socket created successfully")

    # Send SYN - handshake
    sock.sendto("0|SYN".encode(), addr)
    print("[CLIENT] Send SYN for hand-shake")
    seq_num += 1

    # Wait to receive SYN-ACK
    ans_first_ack, sender_address = sock.recvfrom(1024)
    ans_first_ack = int(ans_first_ack.decode().split("|")[0])

    if ans_first_ack == 0:
        print("[CLIENT] SYN-ACK received")
    else:
        print("[CLIENT] Something went wrong, exit program...")
        sys.exit()

    # Send formal details
    send_packet_udp(sock, details, addr)

    # Wait to get OK message from server
    validation = receive_packet_udp(sock, addr).decode()
```

הסבר:

את החלק שקדם לבחירת הפרוטוקול כבר ראינו. כעת, לאחר שנבחר RUDP, הלקוח עושה לחיצת ידיים עם השרת – שולח לו SYN, ומחכה לקבל אות מהשרה – SYN-ACK.

זה אחד ההבדלים בין RUDP לudp, לחיצת הידיים הזאת.

הרעיון שלנו בכתיבת הקוד הוא להכמיס את כל השימוש בrudp, כלומר כמה שיותר אחידות בכתיבת הקוד של TCP וrudp, חוץ מהמקומות הבאים:
* לחיצת הידיים, כמו שאמרנו
* בשליחה וקבלה של פאקטה, כלומר בתקשורת עצמה בין הלקוח והשרת, יש פונקציות מיוחדת שידאגו לאמינות הפרוטוקול – כלומר יהפכו אותו מudp לrudp.

נציג את הפונקציות המיוחדות ונסביר עליהם:

```
def send_packet_udp(sock, data, addr):

    global seq_num

    # Packet looks like: seq_num | ID | data
    packet = (str(seq_num) + "|" + str(generate_packet_id()) + "|").encode()
    packet += data  # assume data is encode already

    # Send packet
    sock.sendto(packet, addr)
    seq_num += 1

    # Saving packet sending time
    packet_time = time.time()

    # Client stop until receiving ACK from server
    receive_ack(sock, packet, addr, packet_time)
```

<u>הסבר:</u>

זוהי פונקציית שליחת פאקטות באמצעות פרוטוקול RUDP.

מה שאנחנו עושים, זה במקום לשלוח רק את המידע ה"מעניין", הdata, אנחנו מוסיפים לפניו גם seq_num וגם ID.

המטרה של הseq_num היא מרכזית. הseq_num מספק לנו מידע אם איזה פאקטה אנחנו מדברים.

לאחר הרכבת הפאקטה בצורה הזו, נשלח אותה. נקדם את הseq_num, ונשמור את הזמן בו היא נשלחה – כל זה בשביל אמינו, תיכף נראה איך זה מתבטא.

לאחר השליחה, אנחנו מחכים לקבל ACK מהמקבל, כדי שנדע שהוא אכן קיבל את הפאקטה. עד שלא נקבל את הack הנכון, לא יישלחו שאר הפאקטות – **כלומר נשמרת פה התכונה של זרימה בסדר נכון של פאקטות – FC.**

```python
def receive_ack(sock, data, addr, packet_time):

    global congestion_window, threshold

    # Extract packet content, in case that the packet will be needed to send again
    packet_content = data.split("|".encode())[2]

    # Prepare tp get an ACK from server
    packet, server_address = sock.recvfrom(1024)
    decoded_packet = packet.decode()

    # Extract seq_num to check reliability
    ack = int(decoded_packet.split("|")[0])

    # If client received the correct ack, everything is great
    if ack == seq_num - 1:
        # Works good, can increase the job window, carefully
        congestion_window += 1
        if congestion_window > threshold:
            congestion_window = threshold
        if congestion_window > MAX_WINDOW_SIZE:
            congestion_window = MAX_WINDOW_SIZE
```

<u>הסבר:</u>

הלאה, נכנסנו לפונקציה של קבלת ack.

מקבלים פאקטת ack, מחלצים ממנה את הseq_num, ובודקים האם הוא זהה לseq_num הנוכחי. אם כן – הרי שהמקבל עדכן אותנו שהוא קיבל את הפאקטה שלנו בהצלחה. נוכל במקרה זה גם להגביר מעט את העומס, כי הקצב הוא טוב – אבל אם החלון גדול מדי, לא נגדיל את העומס – **בזה מתקיימת התכונה של בקרת עומס – CC.**

בכללי, כל עניין ההמתנה עד קבלת ack, מקיים את תכונת האמינות ARQ.

[ARQ ראשי תיבות של Automatic Repeat Request, שהוא פרוטוקול המשמש במערכות תקשורת כדי להבטיח העברת נתונים אמינה בין שני מכשירים על פני ערוץ רועש.

ב-ARQ, השולח שולח חבילת נתונים למקלט ומחכה לאישור (ACK) מהמקלט. אם השולח לא מקבל ACK לאחר פרק זמן מסוים, הוא מניח שהחבילה אבדה ומשדר מחדש את החבילה. המקבל, עם קבלת חבילה, שולח ACK בחזרה לשולח כדי לאשר את קבלת החבילה].

```
# Received ack is smaller than current ack - point on packet lost, so we send the last packet again
else:
    send_packet_udp(sock, packet_content, addr)

# If too long time pass till packet sent, client send it again
if time.time() - packet_time > TIMEOUT:
    send_packet_udp(sock, packet_content, addr)
    # Maybe this happened because overload of job, so we decrease the job window
    congestion_window = 1
    threshold /= 2
    if threshold < 1:
        threshold = 1
```

<u>הסבר:</u>

אם הseq_num הנוכחי לא שווה לack שקיבלנו, הוא בהכרח קטן ממנו – כי אי אפשר לקבל ack על פאקטה שעדיין לא שלחנו.

אז אם הseq_num הנוכחי קטן מהack, סימן שהיה איבוד חבילות, ומה שנעשה זה פשוט לשלוח את הפאקטה האחרונה שוב.

נשים לב שמכיוון שהתהליך חוזר על עצמו עם כל שליחה של פאקטה, לא יכול להיווצר בדרך זו רצף של איבוד חבילות, כי אנחנו מתעקשים לשלוח חבילה שנאבדה שוב פעם.

טיפלנו כאן בבעיית איבוד החבילות.

בנוסף, אם עבר יותר מדי זמן (הקצבנו כמה שניות בודדות, זה מספיק), ועדיין לא קיבלנו סימן שהפאקטה התקבלה בהצלחה, אנחנו שולחים אותה שוב.

כלומר טיפלנו פה בבעיית ההשהיות של זמן.

לבסוף, ליתר ביטחון, אנחנו מקטינים את חלון העבודה כדי לוודאי שאין עומס, כי אולי בגלל העומס ההשהיות נוצרו.

```python
def receive_packet_udp(sock, addr):

    global last_packet_received, window_size, seq_num

    # When receiving packet - first check seq_number for reliability
    data, client_address = sock.recvfrom(BUFFER_SIZE + 8)
    packet_content = data.split("|".encode())[2]
    ack = int((data.split("|".encode())[0]).decode())

    # If seq_num identical to received ack - everything is ok, and we can increase job window
    if seq_num == ack:
        window_size += 1
        if window_size > MAX_WINDOW_SIZE:
            window_size = MAX_WINDOW_SIZE
        # Update client with ack, that packet has safely received
        ack_update = str(ack) + "|" + str(generate_packet_id()) + "|" + "ACK"
        sock.sendto(ack_update.encode(), addr)

    # In case of duplicate - ignore the packet
    elif seq_num > ack:
        return None
```

<u>הסבר:</u>

פונקציה אחרונה – קבלת פקאטה בפרוטוקול RUDP.

דבר ראשון, כשמתקבלת פאקטה, אנחנו מוודאים שהיא נשלחה לפי הסדר – ע"י בדיקת seq_num שלה, אם הוא זהה לseq_num הנוכחי. **שוב רואים כאן טיפול בזרימה נכונה של פאקטות – עיקרון הFC**

אם הם זהים, הכל טוב, אפילו נגדיל את חלון העבודה – כי הכל זורם טוב בינתיים.

בנוסף, **כדי לשמור על ARQ,** המקבל שולח ACK חזרה כדי לעדכן את השולח שהפאקטה התקבלה בשלום.

כעת, איך נטפל בשכפולים של פאקטות?

נשים לב, שאם הseq_num שחילצנו מהפאקטה שהתקבלה, קטן מהseq_num הנוכחי,

זה מעיד על שכפול! וזאת מכיוון שאיך הseq_num הנוכחי הגיע לאן שהוא הגיע?

זה קורה כאשר כל הפאקטות עם seq_num נמוך יותר נשלחו בהצלחה! לכן אם קיבלנו פאקטה עם seq_num שכבר עברנו, אנחנו יודעים שהפאקטה הזאת כבר טיילה פה בעבר.

לכן, פשוט נתעלם ממנה, לא נחזיר את ערכה למשתנה שבו נעשתה הקריאה לפונקציה. זה יהיה הטיפול שלנו בעותקים.

```
# Shouldn't happen, because if seq_num < ack it means that packet lost occurred, And we already take care of
# other functions. But for safety, we exit from program, it is better to try again
else:
    print("[SERVER] Packet lost... waiting to receive lost packets")
    sys.exit()

seq_num += 1

return packet_content
```

<u>הסבר:</u>

במקרה האחרון, יש איבוד חבילות. עכשיו זה לא אמור לקרות, כי הטיפול באיבוד פאקטות כבר קורה במקום אחר, כמו שהראינו מקודם.

אבל ליתר ביטחון, דאגנו שאם יש איבוד פאקטות – שאנחנו יודעים עליו פה כי איך ייתכן שהseq_num הנוכחי קטן מהseq-num שחילצנו מהפאקטה אם לא על ידי כך שנאבדו פאקטות בדרך? – דאגנו שפשוט נצא מהתוכנית. כי אם הגענו לכאן ולא טופל האיבוד כמו שצריך, סימן שיש בעיה חמורה בתקשורת ועדיף לנסות להתחיל אותה מחדש.

<u>כעת נראה פלטים של הרצת התוכנית בפרוטוקול RUDP:</u>

```
client (1)
C:\Users\Owner\PycharmProjects\FinalProject\venv\Scripts\python.exe C:\Use
[CLIENT] Enter ip address: 127.0.0.1
[CLIENT] Choose a server: ftp
[CLIENT] Choose communication protocol, TCP or RUDP: rudp
[CLIENT] Enter UserName: aj123
[CLIENT] Enter password: 2023a
[CLIENT] Socket created successfully
[CLIENT] Send SYN for hand-shake
[CLIENT] SYN-ACK received
[CLIENT] Got an OK message from server
[CLIENT] Enter command type: list
[CLIENT] List of files on FTP server directory:
["check1.txt", "DNS_server.py", "main.py", "RealTest.jpeg"]

Process finished with exit code 0
```

```
FTP_Server
C:\Users\Owner\PycharmProjects\FinalProject\venv\Scripts\python.exe C:\Use
[SERVER] Enter directory path for all of server's files: C:\Users\Owner\Py
[SERVER] Enter client ip address: 127.0.0.1
[SERVER] Sending SYN-ACK...
[SERVER] Username and password are correct
[SERVER] Sending the list of files...
[SERVER] Finish sending

Process finished with exit code 0
```

הסבר:

ניתן לראות שהתוכנית מצליחה לרוץ, כלומר כל הסיכונים שיכל לקרות – כמו איבוד חבילות, או השהיות, וכו', הכל טופל כמו שצריך והמידע הועבר בשלמותו מצד לצד.

בנוסף ניתן לראות בהדפסות את לחיצת הידיים ע"י שליחת SYN ע"י הלקוח והחזרת SYN-ACK ע"י השרת.

נציג כעת הקלטות ויירשארק:

ftp_rudp_recording.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

udp.port == 20373 || udp.port == 30775

No.	Time	Source	Destination	Protocol	Length	Info
33	58.226291	127.0.0.1	127.0.0.1	UDP	36	30775 → 20373 Len=4
36	63.180651	127.0.0.1	127.0.0.1	UDP	37	30775 → 20373 Len=5
37	63.180911	127.0.0.1	127.0.0.1	UDP	41	20373 → 30775 Len=9
38	63.181058	127.0.0.1	127.0.0.1	UDP	50	30775 → 20373 Len=18
39	63.181185	127.0.0.1	127.0.0.1	UDP	42	20373 → 30775 Len=10
40	63.181281	127.0.0.1	127.0.0.1	UDP	41	20373 → 30775 Len=9
41	63.181352	127.0.0.1	127.0.0.1	UDP	42	30775 → 20373 Len=10
42	69.612873	127.0.0.1	127.0.0.1	UDP	47	30775 → 20373 Len=15
43	69.613065	127.0.0.1	127.0.0.1	UDP	42	20373 → 30775 Len=10
44	78.945426	127.0.0.1	127.0.0.1	UDP	106	30775 → 20373 Len=74
45	78.945675	127.0.0.1	127.0.0.1	UDP	41	20373 → 30775 Len=9
46	78.964970	127.0.0.1	127.0.0.1	UDP	54	20373 → 30775 Len=22
47	78.965239	127.0.0.1	127.0.0.1	UDP	42	30775 → 20373 Len=10
48	78.965448	127.0.0.1	127.0.0.1	UDP	40	20373 → 30775 Len=8
49	78.965646	127.0.0.1	127.0.0.1	UDP	42	30775 → 20373 Len=10

```
> Frame 36: 37 bytes on wire (296 bits), 37 bytes captured (296 bits) on interface \Device\NPF_Loopb
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 30775, Dst Port: 20373
> Data (5 bytes)

0000  02 00 00 00 45 00 00 21  7e a4 00 00 80 11 00 00   ····E··! ~······ 
0010  7f 00 00 01 7f 00 00 01  78 37 4f 95 00 0d 68 2f   ····· x7O···h/
0020  30 7c 53 59 4e                                     0|SYN
```

הסבר:

בהקלטה כאן ניתן לראות שהתקשורת התבצעה בפרוטוקול UDP, כל הפאקטות עברו בשלום. ניתן לראות את הפורטים שהגדרנו כבקשתכם לפי המספרים 20, 30 והתעודות זהות שלנו.

בנוסף נשים לב שאכן כל פאקטה שנשלחה עם seq_num מלווה אחריה בפאקטת ACK עם אותו seq_num – מה שמעיד על סדר נכון.

שאלות על החומר

מהן ההבדלים בין QUIC לבין TCP ?

TCP (פרוטוקול בקרת שידור) ו-QUIC (חיבור אינטרנט מהיר UDP) הם שניהם פרוטוקולים המשמשים להעברת נתונים דרך האינטרנט. עם זאת, ישנם כמה הבדלים משמעותיים בין השניים לשם כך הכנו לכם טבלה המסכמת את ההבדלים בניהם :

	TCP	QUIC
יצירת חיבור	משתמש בלחיצת יד תלת כיוונית ליצירת חיבור, הכוללת שליחה וקבלה של שלוש מנות בין הלקוח לשרת לפני שניתן להחליף נתונים.	משתמש בחבילה אחת כדי ליצור חיבור, שיכול להיות מהיר ויעיל יותר.
	TCP משתמש במנגנון שנקרא "בקשה חוזרת אוטומטית" (ARQ) לשידור חוזר של מנות אבודות. כאשר חבילה נשלחת דרך חיבור TCP, היא כוללת מספר רצף המזהה את מיקום החבילה בתוך זרם הנתונים. המקלט שולח אישור (ACK) עבור כל חבילה שהוא מקבל, המציין את מספר הרצף של החבילה הבאה שהוא מצפה לקבל. אם השולח לא מקבל ACK עבור חבילה בתוך פרק זמן מסוים, הוא מניח שהחבילה אבדה ומפעיל שידור חוזר.	משתמש במנגנון שנקרא "פסק זמן לשידור חוזר" (RTO) לשידור חוזר של מנות אבודות. כאשר חבילה נשלחת דרך חיבור QUIC, היא כוללת חותמת זמן המציינת מתי היא נשלחה. אם השולח לא מקבל אישור על החבילה תוך פרק זמן מסוים, השולח מניח שהחבילה אבדה ומפעיל שידור חוזר. מנגנון ה-RTO מחשב את מרווח הזמן בין שליחת חבילה לציפייה לאישור, בהתבסס על זמן הלוך ושוב (RTT) בין השולח למקבל. אם לא מתקבלת

אמינות	השולח שומר עותק של כל חבילה שנשלחה ב"תור שידור חוזר" עד שהוא מקבל ACK עבור החבילה. כאשר השולח מזהה שחבילה אבדה, הוא משדר מחדש את החבילה ומחכה ל-ACK מהמקלט. אם השולח לא מקבל ACK עבור החבילה המשודרת תוך פרק זמן מסוים, הוא ישדר את החבילה שוב, עד למספר מקסימלי של ניסיונות שידור חוזר. TCP כולל גם תכונה הנקראת "שידור חוזר מהיר" שיכול להפעיל שידור חוזר של מנות לפני שפג פסק הזמן המלא של השידור החוזר. תכונה זו מזהה את התרחשותם של שלושה ACKs כפולים (ACKs עבור אותו מספר רצף) ומניחה שהחבילה המתאימה אבדה. לאחר מכן השולח יכול לשדר מחדש את החבילה מבלי לחכות לפסק הזמן המלא של השידור החוזר.	אישור בתוך מרווח זמן זה, השולח ישדר מחדש את החבילה. QUIC כולל גם תכונה הנקראת "שידור חוזר מוקדם" שיכול להפעיל שידור חוזר של מנות לפני שה-RTO יפוג. תכונה זו מזהה דפוסים של אובדן מנות ומשדרת מחדש מנות שסביר להניח שיאבדו לפני ההמתנה לפקיעת ה-RTO המלא. שידור חוזר מוקדם יכול להפחית את זמן ההשהיה ולשפר את הביצועים על ידי הפחתת משך הזמן הדרוש להתאוששות מאובדן מנות.
הצפנה	TCP מספק הצפנה להעברת נתונים באמצעות פרוטוקול TLS משמש ליצירת חיבור מאובטח בין הלקוח לשרת, ומצפין את כל הנתונים המועברים ביניהם. במהלך לחיצת היד הראשונית בין הלקוח לשרת, מתרחשת <u>סדרה של שלבים לאימות והקמת מפתח סודי משותף להצפנה.</u> זה כולל החלפת אישורים המאמתים את זהות הצדדים ומשא ומתן על אלגוריתמי הצפנה ופרמטרים נוספים. לאחר יצירת חיבור ה-TLS, TCP משתמש בהצפנת מפתח סימטרי כדי להצפין את הנתונים המועברים. המשמעות היא שאותו מפתח משמש הן להצפנה והן לפענוח הנתונים. TCP כולל גם מנגנונים לבדיקת תקינות, כגון סיכומי בדיקה, המבטיחים שלא יתעסקו בנתונים במהלך השידור. בסה"כ , השילוב של TLS והצפנת מפתח סימטרי בשימוש ב-TCP מספק אבטחה חזקה להעברת נתונים דרך האינטרנט, ומבטיח שהנתונים מועברים בצורה מאובטחת ולא ניתן ליירט או לחבל בהם על ידי גורמים לא מורשים.	משתמש בשילוב של מנגנוני הצפנה ואימות כדי לספק אבטחה להעברת נתונים. QUIC משתמש בפרוטוקול TLS כדי ליצור חיבור מאובטח בין הלקוח לשרת. לחיצת היד הראשונית בין הלקוח לשרת כוללת החלפת אישורים המאמתת את זהות הצדדים <u>וקובעת מפתח סודי משותף</u> להצפנה. לאחר מכן, QUIC משתמש בתכונה ייחודית הנקראת RTT המאפשרת ללקוח לשלוח נתונים לשרת בצורה מאובטחת מבלי לחכות ללחיצת יד מלאה. זה מושג על ידי מתן אפשרות ללקוח לשלוח נתונים מוצפנים יחד עם הודעת לחיצת היד הראשונית שלו, והשרת יכול לפענח ולעבד את הנתונים באופן מידי, מבלי לחכות לסיום לחיצת היד המלאה. QUIC משתמש בצפנים AEAD להצפנה, המספקים גם הגנה על סודיות ושלמות להעברת נתונים. צפני AEAD משלבים פונקציות הצפנה ואימות לפעולה אחת, מה שמפחית את התקורה של פעולות הצפנה ומשפר את הביצועים.

ריבוי

ל-TCP אין תכונת ריבוי מובנית כמו QUIC. עם זאת, ניתן להשיג ריבוי על גבי TCP באמצעות טכניקות כגון HTTP/2 או WebSockets.
HTTP/2 הוא פרוטוקול שמורכב על גבי TCP ומספק ריבוי של בקשות ותגובות HTTP. ב-HTTP/2, ניתן לשלוח ולקבל מספר בקשות ותגובות HTTP דרך חיבור TCP יחיד, באמצעות מנגנון הנקרא "סטרימינג". כל זרם מזוהה על ידי מזהה זרם ייחודי, המאפשר ללקוח ולשרת להבדיל בין מספר זרמי נתונים המועברים באותו חיבור.
WebSockets היא טכניקה נוספת שניתן להשתמש בה כדי להשיג ריבוי על גבי TCP. WebSockets מאפשרים תקשורת דו-כיוונית בין לקוח לשרת באמצעות חיבור TCP יחיד, באמצעות פרוטוקול מבוסס הודעות. זה מאפשר להעביר מספר סוגי נתונים בו-זמנית על אותו חיבור, מבלי לחסום או להפריע זה לזה.

עם זאת, ייתכן שהטכניקות הללו לא יספקו את אותה רמת ביצועים ואבטחה כמו תכונת הריבוי המובנית ב-QUIC.

ריבוי ב-QUIC מאפשר העברת זרמי נתונים מרובים בו-זמנית על פני חיבור QUIC יחיד. משמעות הדבר היא שניתן לשלוח סוגים שונים של נתונים, כגון משאבי דפי אינטרנט או בקשות API, דרך אותו חיבור מבלי לחסום או להפריע זה לזה.
ב-QUIC, כל זרם מזוהה על ידי מזהה זרם ייחודי, המוקצה על ידי השולח. כל זרם יכול לשאת נתונים עצמאיים, וניתן לשדר ולהתקבל מספר זרמים במקביל דרך חיבור יחיד.
ריבוי ב-QUIC מספק מספר יתרונות ראשית, הוא מפחית את מספר החיבורים הנדרשים עבור קבוצת נתונים נתונה, מה שיכול להפחית את זמן ההשהיה ולשפר את הביצועים. שנית, הוא מאפשר למקלט לתעדף ולעבד סוגים שונים של נתונים בנפרד, מה שיכול לשפר עוד יותר את הביצועים וההיענות. לבסוף, זה יכול לספק אבטחה טובה יותר, שכן כל הנתונים מוצפנים ומשודרים בחיבור יחיד, מה שמפחית את הסיכון של יירוט או שיבוש על ידי גורמים לא מורשים.

מהן ההבדלים בין VEGAS לבין CUBIC ?

VEGAS (בקרת גודש משתנה) ו-CUBIC (בקרת גודש מעוקב) הם שניהם אלגוריתמים לבקרת גודש המשמשים ב-TCP (פרוטוקול בקרת שידור) כדי לווסת את זרימת הנתונים ברשת. עם זאת, ישנם כמה הבדלים עיקריים בין שני האלגוריתמים:

	VEGAS	CUBIC
מנגנון בקרה	VEGAS משתמשת במנגנון מבוסס חלון כדי לשלוט על הגודש, אשר מחושב על סמך זמן הנסיעה הלוך ושוב משוער של הרשת (RTT). VEGAS מודד את הזמן שלוקח למנות להישלח מהשולח למקלט ובחזרה, ומתאים את גודל החלון בהתבסס על ה-RTT המשוער.	CUBIC, לעומת זאת, משתמשת בחלון גודש שגדל ומתכווץ בהתאם לרמת הגודש הנוכחית. CUBIC משתמש בפונקציה מעוקבת כדי לחשב את גודל החלון, מה שמאפשר לו להסתגל לתנאי רשת משתנים מהר יותר.
זיהוי גודש	VEGAS מזהה גודש על ידי מדידת הזמן שלוקח לשידור ואישור מנות, והשוואה לסף מוגדר מראש. אם חריגה מהסף, VEGAS מפחית את קצב השליחה כדי למנוע עומס.	CUBIC, לעומת זאת, משתמשת בגישה אגרסיבית יותר לזיהוי גודש, על ידי ניטור מספר המנות הלא מאושרות ברשת. כאשר מספר החבילות הלא מאושרות מגיע לסף, CUBIC מפחית את קצב השליחה בכמות גדולה יותר.

הֲגִינוּת	VEGAS נועד לספק שיתוף הוגן של משאבי רשת בין זרימות מרובות, על ידי התאמת קצב השליחה בהתבסס על ה-RTT המשוער של כל זרימה.	CUBIC, לעומת זאת, מותאמת לרשתות ארוכות ברוחב פס ויכול להיות פחות הוגן בשיתוף משאבי רשת בין זרימות מרובות. הסיבה לכך היא ש-CUBIC נותנת עדיפות למקסימום התפוקה על פני הגינות.

לסיכום, VEGAS ו-CUBIC הם שני אלגוריתמים שונים לבקרת גודש המשמשים ב-TCP. בעוד VEGAS משתמשת במנגנון מבוסס חלון כדי לשלוט על הגודש ומתעדפת הגינות, CUBIC משתמשת בפונקציה מעוקבת כדי לחשב את גודל החלון והיא מותאמת לרשתות ברוחב פס גבוה ולמרחקים ארוכים.

פרוטוקול BGP

BGP (Border Gateway Protocol) הוא פרוטוקול המשמש להחלפת מידע ניתוב בין מערכות אוטונומיות שונות (ASes) ברשת מחשבים, במיוחד באינטרנט. מערכת אוטונומית היא אוסף של רשתות מחוברות הנמצאות בשליטה של תחום ניהולי אחד. BGP מאפשרת למערכות האוטונומיות הללו להחליף מידע על הרשתות שהן אחראיות עליהן, והנתיבים שהתנועה יכולה לעבור כדי להגיע לרשתות אלו.

BGP הוא פרוטוקול נתיב-וקטור, מה שאומר שהוא בוחר את הנתיב הטוב ביותר לניתוב תעבורה בהתבסס על מספר גורמים, כגון אורך הנתיב, יציבות הנתיב ומדיניות המוגדרת על ידי ה-ASes עצמם. BGP משתמש במערכת מורכבת של תכונות כדי לתאר את המאפיינים של נתיבים שונים, ולהשוות ולבחור את הנתיב הטוב ביותר לניתוב תעבורה.

BGP משמש בעיקר את ספקי שירותי האינטרנט (ISP) ורשתות ארגוניות גדולות כדי להתחבר לרשתות אחרות ולנתיבי החלפה. כאשר ספק שירותי אינטרנט מתחבר לרשת אחרת, הוא משתמש ב-BGP כדי להחליף מידע על הרשתות שכל AS אחראי עליהן, והנתיבים שהתנועה יכולה לעבור כדי להגיע לרשתות אלו. זה מאפשר ניתוב יעיל של תעבורה בין רשתות שונות, מה שמאפשר למשתמשים לגשת לתוכן ולשירותים מכל מקום בעולם.

BGP הוא מרכיב קריטי בתשתית האינטרנט, מכיוון שהוא מאפשר לאינטרנט לתפקד כרשת גלובלית של רשתות מחוברות. עם זאת, BGP הוא גם פרוטוקול מורכב הדורש תצורה וניהול קפדניים כדי להבטיח שתעבורה מנותבת בצורה יעילה ומאובטחת. הגדרות שגויות או שגיאות ב-BGP עלולות לגרום לניתוב שגוי של התעבורה או לאיבוד, או עלולות

להוביל לפרצות אבטחה או התקפות על הרשת. כתוצאה מכך, BGP כפוף למחקר ופיתוח מתמשכים כדי לשפר את המהימנות, האבטחה והיעילות שלה.

האם פרוטוקול BGP עובד על פי מסלולים קצרים?

BGP (Border Gateway Protocol) לא בהכרח עובד לפי מסלולים קצרים. BGP הוא פרוטוקול ניתוב וקטור נתיב, כלומר הוא בוחר את הנתיב הטוב ביותר לניתוב תעבורה בהתבסס על מספר גורמים, כגון אורך הנתיב, יציבות הנתיב ומדיניות המוגדרת על ידי המערכות האוטונומיות. (ASes) עצמם.

תהליך בחירת הנתיבים של BGP מורכב ומתחשב במספר גורמים. לדוגמה, BGP עשוי להעדיף נתיב ארוך יותר על פני נתיב קצר יותר אם הנתיב הארוך יותר יציב יותר או בעל התאמה טובה יותר למדיניות. המשמעות היא ש-BGP לא תמיד בוחר את הנתיב הקצר ביותר לניתוב תעבורה.

תהליך בחירת הנתיבים של BGP מבוסס על סט של תכונות המתארות את המאפיינים של נתיבים שונים. תכונות אלו כוללות דברים כמו אורך הנתיב, מספר ה-ASes שהנתיב עובר ומאפיינים שונים הקשורים למדיניות. ASes יכולים להגדיר את נתבי ה-BGP שלהם כדי לתעדף תכונות מסוימות על פני אחרים, מה שיאפשר להם לשלוט על אופן ניתוב התעבורה בין הרשתות שלהם לרשתות אחרות.

לסיכום, BGP לא תמיד בוחר את הנתיב הקצר ביותר לניתוב תנועה. במקום זאת, תהליך בחירת הנתיב של BGP לוקח בחשבון מגוון גורמים, כולל אורך נתיב, יציבות ומדיניות שהוגדרו על ידי ה-ASes עצמם.

מהן ההבדלים בין BGP לבין OSPF ?

OSPF הוא גם פרוטוקול ניתוב המשמש ברשתות מחשבים, אך הוא שונה בהיקפו ובמטרתו מ BGP.

BGP כמו שאמרנו לעיל הוא פרוטוקול ניתוב בין-דומיין המשמש להחלפת מידע ניתוב בין מערכות אוטונומיות שונות (ASes) (הסברנו לעיל) באינטרנט. הוא משמש בעיקר את ספקי שירותי האינטרנט (ISP) ורשתות ארגוניות גדולות כדי להתחבר לרשתות אחרות ולנתיבי החלפה. BGP משתמש בפרוטוקול נתיב-וקטור כדי לקבוע את הנתיב הטוב ביותר לניתוב תעבורה בין ASes.

OSPF, לעומת זאת, הוא פרוטוקול ניתוב תוך-דומיין המשמש להחלפת מידע ניתוב בתוך מערכת אוטונומית אחת. הוא מיועד לשימוש בתוך רשת של ארגון יחיד, כגון רשת ארגונית או קמפוס. OSPF משתמש בפרוטוקול מצב קישור כדי לקבוע את הנתיב הטוב ביותר לניתוב תעבורה בין נתבים בתוך אותו AS.

ההבדל העיקרי בין BGP ל-OSPF הוא ש-BGP משמש להחלפת מסלולים בין מערכות אוטונומיות שונות, בעוד ש-OSPF משמש להחלפת מסלולים בתוך מערכת אוטונומית אחת. המשמעות היא ש-BGP משמש לחיבור רשתות שונות יחד, בעוד ש-OSPF משמש לחיבור נתבים בתוך אותה רשת. BGP משתמש גם בפרוטוקול נתיב-וקטור כדי לקבוע את הנתיב הטוב ביותר לניתוב תעבורה בין ASes, בעוד OSPF משתמש בפרוטוקול מצב קישור כדי לקבוע את הנתיב הטוב ביותר לניתוב תעבורה בתוך AS יחיד.

הבדל מרכזי נוסף בין BGP ל-OSPF הוא המורכבות שלהם. BGP הוא פרוטוקול מורכב הדורש תצורה וניהול קפדניים כדי להבטיח שהתעבורה מנותבת בצורה יעילה ומאובטחת בין ASes שונים. OSPF, לעומת זאת, הוא פרוטוקול פשוט יותר המשמש בעיקר ברשת של ארגון יחיד.

מהן ההבדלים בין DNS לבין ARP ?

DNS (מערכת שמות תחום) ו-ARP (פרוטוקול רזולוציית כתובת) שניהם פרוטוקולי רשת חשובים, אך הם משרתים מטרות שונות.

DNS משמש לתרגום שמות דומיינים הניתנים לקריאה אנושית (כגון www.example.com) לכתובות IP (כגון 192.168.0.1) שמחשבים יכולים להבין. שרתי DNS מתחזקים מסדי נתונים של שמות דומיין וכתובות IP, וכאשר משתמש מזין שם דומיין לדפדפן האינטרנט שלו, הדפדפן שולח בקשה לשרת DNS לתקן את שם הדומיין לכתובת IP.

ARP, לעומת זאת, משמש כדי לפתור את כתובת החומרה או ה-MAC של התקן לכתובת ה-IP שלו ברשת מקומית. כאשר מחשב רוצה לשלוח נתונים למכשיר אחר באותה רשת מקומית, הוא משתמש ב-ARP כדי לחפש את כתובת ה-MAC של המכשיר המשויך לכתובת IP נתונה. זה מאפשר לשלוח את הנתונים ישירות למכשיר הנכון ברשת.

לסיכום, DNS משמש לפתרון שמות דומיין לכתובות IP, בעוד ARP משמש לפתרון כתובות MAC לכתובות IP ברשת מקומית.

* כתובת Mac (בקרת גישה למדיה) היא מזהה ייחודי המוקצה לבקר ממשק רשת (NIC) לשימוש ככתובת רשת בתקשורת בתוך קטע רשת. זהו מספר 48 סיביות (מיוצג לפעמים בפורמט הקסדצימאלי) המוקצה פיזית למכשיר על ידי היצרן.