

פרויקט גמר - רשתות תקשורת

**שחר זידל - 211990700
מוהנד ספי - 208113381**

פברואר 2023

:FTP

הוא פרוטוקול תקשורת להעברת קבצים המבוסס שרת לקוח והוא נמצא בשכבת האפליקציה.

הפרוטוקול מאפשר כל מיני פעולות עם או על קבצים:-
השרת הנותן שירות כגון :-

- הורדה של קובץ מהשרת.
- העלאה של קובץ לשרת.
- מחיקת קובץ.
- שליחת רשימת הקבצים שיש אצל השרת.



TCP RENO

הוא חלק מפרוטוקולי TCP האחראיים על בקרת הלחץ/הצפיפות ברשת. אנחנו בתור מתכנתים רוצים לייצר את החוויה הטובה ביותר בשימוש באפליקציה שלנו. אין דבר יותר מעצב מאפליקציה שנתקעת או לא עומדת בעומס של הרשת. מידע נאבד, חלונות נתקעים, לא כיף. כל האפליקציות וכל השירותים בסופו של דבר יושבים על אותו רוחב פס. אם אנחנו לא נתחשב ברוחב הפס שיש לנו, האפליקציה שלנו לא תתפקד כמו שצריך. הראוטרים מסוגלים להעביר X כמות מידע ולא יותר. אז הושגה ההבנה שנוכל להתאים את האפליקציה שלנו לעבוד לפי רוחב הפס הנוכחי שיש לנו. יכול להיות שבתנאי רשת קשים נעביר מידע יותר לאט, אולי בחבילות יותר קטנות, והיוזר בסך הכל יחווה ירידה ביכולת השימוש שלו באפליקציה. אבל זה יאפשר לנו לבחור איזה פונקציונליות אנחנו רוצים להוריד מהאפליקציה, ולהתנהל באופן דינמי אל מול עומסי הרשת. מה שיתן לנו מגוון רחב הרבה יותר של פתרונות למצבים של עומס ברשת.

אנחנו בחרנו לעקוב אחר העקרונות של ה-CC algorithm של RENO במימוש שלנו ל-RUDP. נרחיב איך הפרוטוקול עובד ונדגים זאת על שליחת מידע.

הפרוטוקול מתבסס על מעגל המורכב מ-4 שלבים. המעגל הזה רץ כל הזמן. בחלק הראשון של שליחת המידע, אנחנו רוצים להבין מול מה אנחנו עומדים. אין לנו דרך באמת לדעת מה רוחב הפס שמוקנה לנו כרגע, מעבר לעובדה שהוא יכול להשתנות בכל שנייה. נתחיל לשלוח פקטות, ועל כל ack מוצלח שנקבל, נגדיל את גודל החלון או chunk המידע ב-2. הגדילה היא אקספוננציאלית, וזה השלב במעגל שמכונה slow start. אנחנו יכולים לעבור לשלב הבא באחת משלושת הדרכים:

- בעקבות זיהוי פקטות שנאבדו
 - חלון הקבלה של השרת התמלא או הגיע לגבול המוגדר מראש
 - חלון השליחה אצל השולח מגיע לקו אותו מותח משתנה בשם ssthresh. זה גבול הנמצא אצל הלקוח ומוגדר מראש.
- # נציין שגבול המוגדר מראש לא בהכרח מוגדר מראש לפני ריצת התוכנית. מאחר שהרשת היא דינמית נעדיף לקבוע את הגבולות האלה בזמן הריצה.

אם אחד התנאים התמלאו, ניכנס למצב שקוראים לו congestion avoidance. זהו מצב שבו אנחנו מבינים שאנחנו מתקרבים לרוחב המקסימלי של הפס שלנו, ובשביל להימנע ממצב של איבוד פקטות וירידה שאינה נשלטת על ידינו בפונקציונליות האפליקציה, נבחר בעצמנו איך אנחנו מפחיתים את העומס שאנחנו מייצרים על הרשת. במקרה של שליחת הקבצים ו-reno, נפחית את הגדילה של החלון. כלומר, עכשיו החלון יגדל ב-1.2 כל פעם ולא פי 2.

נניח שהלקוח שולח לשרת קובץ. הלקוח יחלק את הקובץ לפקטות וישלח לשרת פקטה כל פעם. אם

הלקוח יזהה איבוד פקטות - הוא יכנס לשלב הנקרא fast retransmit. הלקוח יכול לזהות שלא התקבלה פקטה ב-2 דרכים:

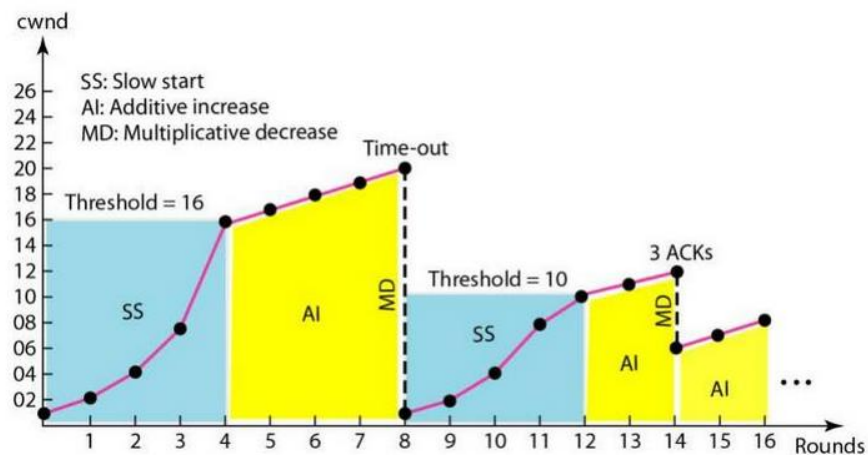
- הלקוח מודד זמנים מרגע השליחה. אם לא חזר ack במשך זמן מסוים, הלקוח יסיק מזה שהפקטה שניסה לשלוח נאבדה ברשת וישלח אותה מחדש. אפשר לקבוע את זמן ההמתנה מראש, או לחשב את הזמן הממוצע שלוקח לחבילה להישלח לשרת ול-ack לחזור ללקוח, מה שיתן לנו מדידת זמנים יותר דינמית ורלוונטית לרגע.
- לאחר כל חבילה שנשלחה, השרת יחזיר ack. בפקטת ה-ack, רשום מה המספר הסידורי של החבילה שהתקבלה. לשרת יש גם טיימר. אם השרת לא יקבל את החבילה לאחר זמן מסוים, שגם מחושב על פי אחת מ-2 הדרכים באופציה הראשונה, הוא ישלח את ההודעת ה-ack עם המספר הסידורי של החבילה האחרונה שהוא קיבל. ככה בעצם מתקבל duplicate ack. אם יתקבלו 3 duplicate acks, הלקוח יעצור את הטיימר - הוא לא יחכה ל-timeout, וישלח את החבילה עם המספר הסידורי שב-ack שוב.

חשוב לציין ש-3 duplicate acks לדעתנו זה הרבה פחות נורא מהאופציה הראשונה. אם קיבלנו 3 duplicate acks אנחנו יודעים שלשרת הייתה בעיה בקבלת אחת הפקטות, אבל שהוא כרגע ממשיך לקבל פקטות. אם הגענו לסוף הזמן שמוגדר להמתנה ל-ack, נשלח את ההודעה שוב, אבל כרגע אנחנו לא יודעים מה מצבו של השרת. בהתאם, אם הגענו לסוף הזמן, נקטין את החלון ל-1 ולא לחצי.

לאחר שקרו אחד משני התנאים, והלקוח זיהה שלא התקבלה פקטה, מעבר לשליחת הפקטה מחדש, השרת יפחית את גודל החלון ב-2 (או יותר), וגם יפחית את ה-ssthresh ב-2. אנחנו מבינים שפה חצינו את הגבול מבחינת היכולת של הרשת, ולא נרצה להגיע למקום הזה שוב. בשלב הזה סגרנו מעגל ואנחנו מתחילים את התהליך מחדש. אבל, הפעם נקפוץ ישר ל-congestion avoidance, ונדלג על שלב ה-slow start.

נציין שלכל התהליך שקורה מהרגע שהגענו לשלב של fast retransmit קוראים fast recovery.

TCP congestion policy: *Example*



שרת ה-DHCP:

כאשר מחשב מתחבר ב-WIFI או בכבל רשת לראוטר/switch מסוים, הוא צריך לקבל מהשרת כתובת שאיתה יוכל לתקשר עם השירותים שהשרת מציע לו.
בשביל זה, השרת מחזיק שרת אחר הנקרא שרת DHCP- dynamic host configuration protocol - שרת שמטרתו לנהל את כתובות ה-IP הפנויות ולהקצות אותן למחשבים המתחברים לשרת.

תהליך קבלת ה-IP אורך 4 שלבים:

1. **dhcp discover** - הלקוח מתחבר לתווך, ושולח על התווך הודעת broadcast לכל מי שיכול לשמוע אותו. ההודעה נקראת dhcp discover, ומטרתה להגיד לשרתי ה-dhcp ש(בתקווה) מחוברים לתווך שהוא רוצה לקבל כתובת IP על מנת להשתמש בשרת.
2. **dhcp offer** - שרת ה-dhcp מקבל את ההודעה של הלקוח, ובזמן שכל המחשבים האחרים שקיבלו את ההודעה מתעלמים ממנה, הוא מתוכנת להתייחס אליה. הוא מציע ללקוח כתובת IP, כתובת subnet mask וכתובת default gateway.
3. **dhcp request** - הלקוח שולח הודעה שהוא מקבל את הכתובת שהשרת הציע לו. לפעמים יש מספר שרתי dhcp ולא רק אחד. אז הודעת הלקוח, גם מודיעה לשרת ה-dhcp שהציע את הכתובת שהוא(הלקוח) מקבל אותה, וגם מודיעה לשרתי ה-dhcp האחרים שהוא לא צריך את הכתובת שלהם, ושהם יכולים להחזיר את הכתובת לרשימת הכתובות הפנויות שיש להם.
4. **dhcp ack** - הודעה משרת ה-dhcp הנבחר אל הלקוח, המודיעה ללקוח שהוא מאשר שקיבל את הודעת ה-dhcp request והקצה לו את הכתובת. בכך מסתיים תהליך קבלת כתובת ה-IP.

המימוש שלנו לשרת-

אנחנו החלטנו לממש את השרת מ-0 ולתכנן את פעילותו. נסביר בפירוט איך הוא עובד.
לשם כך השתמשנו בספריית struct בפיתון וב-socket מסוג udp.

פורמט ההודעות -

אנחנו הגדרנו מספר סוגי הודעות - לכל הודעה יש מספר המציין אותה. רצינו לשלב את סוג ההודעה ואת הדגל המציין האם ההודעה היא בקשה או תגובה באותו אינדיקטור.

במספרים הבאים, הביט הכי ימני מציין האם החבילה היא בקשה(=0) או תגובה(=1):
שני הביטים שאחריו (ביט 1 וביט 2 מימין) מציינים האם מה סוג הבקשה כאשר:

- dhcp discover = 00
- dhcp offer = 01
- dhcp request = 10
- dhcp ack = 11

בסך הכל קיבלנו:

- dhcp discover = 00 + 0 = 000 = 0
- dhcp offer = 01 + 1 = 011 = 3
- dhcp request = 10 + 0 = 100 = 4
- dhcp ack = 11 + 1 = 111 = 7

המספרים האלה מייצגים את סוג הפקטות.

מבנה החבילה dhcp discover:

```
def pack_discover(self):
    PTYP = "0"
    TID = get_random_string(8)
    packet = struct.pack("s 8s", PTYP.encode("ascii"), TID.encode("ascii"))
    return packet, TID
```

המשתנה TID מייצג transaction ID. כלומר, סוג של תעודת זהות לתקשורת ה-DHCP הנוכחית, שמלווה את התקשורת של הלקוח עם שרת ה-dhcp מתחילתה ועד סופה. היא משמשת לנו עוד גורם מאבטח שהודעות שמגיעות אלינו (הלקוח) אינן מזויפות, או מגיעות בטעות משרת אחר. לצורך יצירת ה-TID השתמשנו בפונקציה שיוצרת מחרוזת רנדומלית בגודל 8 תווים המורכבת מאותיות קטנות בלבד. בנוסף, נוכל לראות שאנחנו אורזים כאן את הסוג של הפקטור dhcp discover שהוא 0.

ההודעה נשלחת ב-broadcast (לכתובת 255.255.255.255) לפורט 67, שהוא הפורט המסורתי שמציין את השרת בתקשורת dhcp. בהגעת החבילה לשרת, השרת פורק את החבילה, ואם סוג החבילה אינו 0 הוא מודיע על תקלה.

מבנה החבילה dhcp offer:

```
def pack_offer(self, TID):
    PTYP = "3"
    OIPADDR = occupied_ip_addresses()
    ODFTGTWY = "192.168.1.1".encode("ascii")
    OSBNTMSK = "255.255.255.0".encode("ascii")
    LEASETIME = 43200

    package = struct.pack("s 8s i 11s 13s i", PTYP.encode("ascii"), TID.encode("ascii"), OIPADDR,
                          ODFTGTWY, OSBNTMSK, LEASETIME)
    return package
```

אנחנו בחרנו להקצות ללקוחות שלנו כתובות מהסוג 10.0.0.0-254/24. בחרנו רשת קטנה השייכת ל-class C, כאשר ברשת שלנו יש רק 255 כתובות פנויות. מכאן, כתובת ה-subnet mask שהגדרנו הינה 255.255.255.0. כתובת ה-default gateway הינה 192.168.1.1 כנהוג.

על מנת להקצות ללקוח כתובת IP, נשתמש במתודה occupied_ip_addresses() שבנינו. המתודה occupied_ip_addresses() - הסבר:

```
def occupied_ip_addresses():
    for i in range(1, 255):
        if available_addresses[i][0] == 0:
            available_addresses[i] = (1, 0)
            return i
    return "no more available addresses"
```

השרת מחזיק משתנה גלובלי מסוג list. ב-list יש 255 מבני נתונים מסוג tuple, המכילים שני שדות בתוכם. האינדקסים של ה-list, מייצגים את הסיומות של כתובות ה-IP שאנחנו מקצים ברשת שלנו - מ-0 ועד 254, כאשר אינדקס 50 למשל מייצג את הכתובת 10.0.0.50. השדה השמאלי של כל tuple (באינדקס 0), מציין האם הכתובת תפוסה כרגע או פנויה. (1 = תפוסה, 0 = פנויה). השדה הימני (באינדקס 1), מציין את הזמן שבו הוקצתה הכתובת ללקוח.

כאשר לקוחות מתחברים לשרת ומקבלים כתובת IP, שרת ה-dhcp יודע על כך, כי הוא זה שמקצה להם את הכתובת. אך, כאשר לקוחות מתנתקים מהרשת מסיבה זו או אחרת, הם (בשיא חוצפתם!!) לא מעדכנים אף אחד. התוצאה היא שהשרת יכול לחשוב לאורך הרבה זמן שכתובות מסוימות מוקצות ללקוחות, למרות שהן מזמן לא בשימוש. יש מספר דרכים לטפל בנושא הזה, ביניהן:

- אפשר מדי כמה שעות, לבדוק האם הלקוחות המחוברים לשרת באמצעות למשל הודעת ping דרך פרוטוקול icmp. השרת ינתק באופן יזום לקוחות שלא החזירו pong.
- אפשר להקצות זמן מסוים, שאחריו שרת ה-dhcp ינתק באופן יזום את כל הכתובות, ויתן לכולם להתחבר מחדש. ברגע שהלקוחות ינותקו מהרשת הם יתחברו באופן אוטומטי מחדש אל שרת ה-

.dhcp

לדרך הזו יש יתרון על פני הדרך הקודמת, אך גם חסרון.
היתרון - נניח שהאפליקציה שלנו נורא מפורסמת והתחברו אלינו בו זמנית 255 לקוחות. כולם קיבלו כתובת IP. לאחר מכן, נניח שלאחר 3 שעות, הלקוחות שהיו מחוברים לכתובות 10.0.0.2-253 התנתקו, ורק הלקוחות שמחוברים לכתובות 10.0.0.1 ו-10.0.0.254 נשארו מחוברים. כל פעם שהשרת יריץ בדיקה וישלח פינגים לכולם, הוא ישלח 255 פינגים. למרות, שלאחר 3 שעות יש לו רק 2 לקוחות מחוברים. למה שהוא ישלח שוב 255 פינגים בהרצה הבאה ולא רק 2 פינגים (בהנחה שלא התחברו לקוחות חדשים)?
באפליקציה שלנו זאת לא כזאת בעיה, כי יש לנו 255 כתובות בסך הכל. אבל אם היינו משתמשים ב-class A, היה לנו 16,777,214 כתובות. קצת מסובך.
אם ננתק את כולם וניתן להם להתחבר מחדש, הם יתחברו מחדש לפי הסדר, ונקטין את סיבוכיות. החיסרון - ניתוק הלקוחות ללא ידיעתם היא פעולה שלבדה יכולה לשבש את התקשורת. אם במקרה המחשבים לא מתחברים אוטומטית חזרה לשרת או מתעכבים בהתחברות, הדבר ייצור בעיות latency אצל הלקוח, ואין נורא כמו לקוח עצבני שלא מבין למה לא עובדת לו הרשת.



אנחנו מימשנו את הדרך השנייה בשרת. הקצנו זמן של 12 שעות לכל כתובת. הזמן מתחיל להיספר מהרגע שהשרת מקבל את הודעת ה-dhcp request, כי עד אז הוא לא יכול להיות בטוח שהלקוח בחר בו.

מימוש הדרך השנייה - תהליך פינוי הכתובות:

בשרת שבנינו יש thread שרץ בנפרד מהשרת. ה-thread מתחיל לרוץ כאשר השרת מתחיל לעבוד. בתוכו, רצה לולאה אין סופית הפועלת כך- ה-thread יפעיל מתודה שתרוץ על כל הכתובות, וינתק כתובות שזמן האתחול שלהן היה לפני יותר מ-86400 שניות (יום שלם). לאחר שהוא מסיים, הוא הולך לישון לשעתיים (שנ"צ זה חשוב). בסך הכל ה-thread בודק כל שעתיים האם יש עוד כתובות שהוא יכול לנתק.

לאחר כל התהליך, נשלח את החבילה בהודעת broadcast נוספת, מאחר שאין ללקוח כתובת עדיין. נציין, שבתהליך הפריקה של חבילת dhcp offer על ידי הלקוח, מתבצעת השוואה בין ה-TID שהלקוח שלח לבין זה שקיבל בחבילה, מה שמגביר את אבטחת השרת. בנוסף, כאשר הלקוח מקבל את 3 הכתובות, (ip, addr, subnetmask and default gateway) הוא שומר אותם בתור שדות באובייקט.

מבנה החבילה dhcp request :

```
def pack_request(self, ipaddr, TID):
    PTYPE = "4"
    packet = struct.pack("s Bs i", PTYPE.encode("ascii"), TID.encode("ascii"), ipaddr)
    return packet
```

החבילה הפעם מכילה את סוג החבילה-4, את ה-TID על מנת שהוא ימשיך ללוות את התקשורת, ואת כתובת ה-IP שהוא קיבל, על מנת שהשרת יבין במי מדובר ויאתחל את הטיימר שלו בעת קבלת ההודעה. במימוש שלנו, ההודעה נשלחת ישירות אל השרת, שאת הכתובת שלו אנחנו יודעים בשלב הזה. (הכתובת מוחזרת מהמתודה recvfrom() המובנית בפיתון, שהופעלה בקבלת הודעת ה-dhcp offer).

לבסוף, מבנה החבילה dhcp ack :

```
def pack_ack(self, string, TID):
    PTYPE = ''
    if string == "9":
        PTYPE = "9"
    elif string == "7":
        PTYPE = "7"
    packet = struct.pack("s 8s", PTYPE.encode("ascii"), TID.encode("ascii"))
    return packet
```

אם הקצאת הכתובת עברה בהצלחה, הסוג שיחזור מהמתודה `unpack_ack()` הוא 7 - כסוג הפקטה `dhcp` `ack`. אחרת, (אם קרתה שגיאה ולא כתוב אצל השרת שהוא הקצה את הכתובת בכלל, או אם הלקוח שלח את המספר 1- במשתנה `IPADDR`, מה שאומר שהשרת כרגע מלא) הוא יחזיר את המספר 9 - מספר שמבחינת הלקוח מהווה אינדיקציה שההקצאה לא עברה בהצלחה והוא יפנה לשרת ה-`dhcp` ויתחיל את התהליך מחדש.

#בקבלת ה-`ack`, הלקוח שוב משווה את ה-`TID` מהסיבות שציינו קודם לכן.

נציין נקודה חשובה:

```
# DHCP request
while self.IPADDR == "" or self.SUBNETMASK == "" or self.DEFAULTGATEWAY == "":
    self.DHCP_communication(s)

# DNS request
while self.server_ip_address == "":
    self.DNS_communication(s)
```

שני השרתים שלנו עובדים בחיבורי `udp` בלבד, ללא התחשבות ב-`reliability` או שימוש ב-`tcp` לקבלת אמינות יותר גבוהה. בחרנו לעשות ככה מאחר שהתקשורת עם שני השרתים מאוד קצרה - ב-`dhcp` נשלחות סה"כ 4 הודעות בפקטות מאוד קטנות, ב-`dns` התקשורת קצת יותר כבדה בגלל הפנייה לשרתים אחרים אך לא עד כדי כך. לדעתנו, היתרון שבחיבור יותר אמין לא מספיק כדי להצדיק את העובדה שהוא לוקח הרבה יותר זמן, ספציפית לתקשורות קצרות שכאלה. לכן, הוספנו אצל הלקוח לולאה אינסופית עבור כל פנייה לאחד מהשרתים. לפני הפנייה לכל אחד מהשרתים, כל המשתנים מאותחלים להיות "" . השרתים מתוכננים להשאיר את המשתנים "" במידה והתקשורת לא צלחה. כך, במקום לוודא את אמינות החיבור, אם משהו לא צלח נשלח את הלקוח לשרתים שוב, עד שהכל יעבוד.

שרת ה-DNS: הסבר קצר על Scapy:

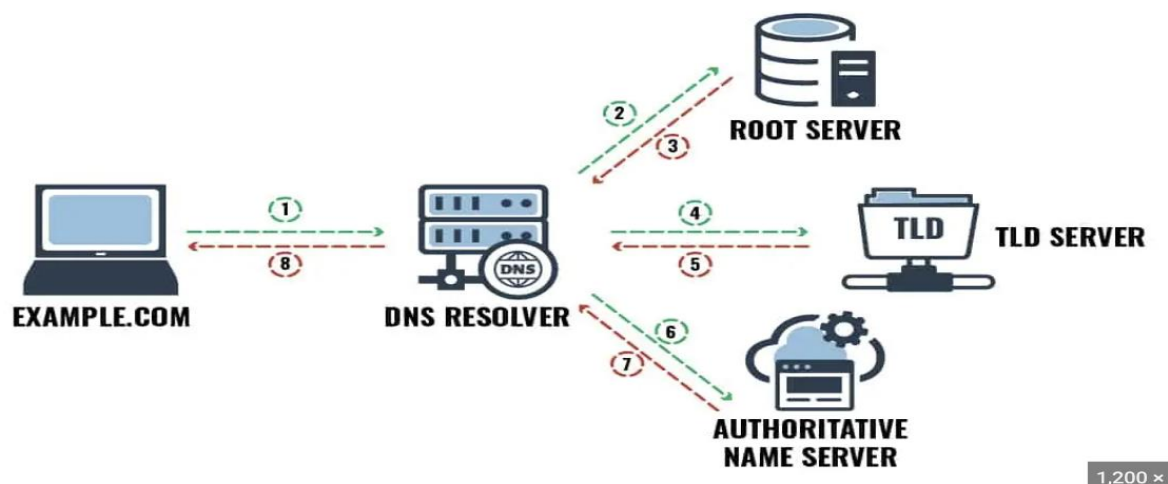
היא ספרייה שמאפשרת להסניף ולשלוח פקטות בתצורתן המסורתית (כלומר, כמו שהן נראות ב-wireshark). השימוש העיקרי של הספרייה הזו הוא לזייף ולשלוח פקטות מסוג dhcp offer או dhcp ack בפרוטוקול dhcp, או פקטות dns. נשים לב שאלה בדיוק הפקטות הקריטיות שהשרת שולח ללקוח. אם אנחנו בתור פורצים, יכולים להסניף ולזייף פקטות של שנשלחות מהשרת, ובסופו של דבר לגרום ללקוח לחשוב שאנחנו השרת, זאת נקודת פתיחה נהדרת לשלל מתקפות סייבר.

הוא פרוטוקול בשכבת האפליקציה. תפקידו העיקרי הינו לתרגם שמות דומיינים לבין כתובת ה ip הרלוונטית. הוא בנוי מהיררכיה, כך ש קיים איזורים מסוימים שמבקר בה בכדי לקבל את ה ip המתאים של הדומיין, האזורים הם בעצם הסימות של כל דומיין, כמו org, com, il, ...

איך עובד תהליך ה-dns:

- שרת ה-DNS שלנו שולח שאילתה ראשונה ל-root dns server
- שרת ה-root dns מכווין אותנו לשרת ה-TLD המתאים לנו.
- שרת ה-DNS שלנו שולח הודעה לשרת ה-TLD.
- שרת ה-TLD מכווין אותנו לשרת המהימן של הדומיין שלנו Authoritative DNS server
- בשלב הזה, אם כל השרתים שה-TLD הכווין אותנו אליהם לא מתאימים, נחזיר תשובה שלילית- כלומר, שהשרת הזה לא קיים.

במידה והלקוח כלשהו שלח בקשה לדומיין כלשהו (וזאת פעם ראשונה שהשרת dns מקבל אותה), אז הוא אכן יעבור על התהליך שהוזכר מעל, אבל בפעם השנייה השאת dns יבדוק קודם כל במטמון אם הרשומה של השאילתה הזו כבר קיימת, ולפי זה הוא יחזיר תשובה להמשיך את התהליך או לא.



- בשרת שלנו, אנחנו משתמשים ב-scapy כדי לדמות בקשת DNS לשרת. שם האפליקציה שלנו לא רשום באף דומיין. לכן, גם אם נפנה לכל שרת dns כל שהוא, תחזור לנו תשובה שלילית. אבל, אנחנו בכל זאת רוצים שתחזור איזשהי תשובה חיובית מהשרת, ולכן השרת עובד ככה:
1. הלקוח פותח socket בחיבור udp עם השרת שלי.
 2. השרת משתמש בספרייה scapy ובעזרתה שולח בקשת dns. בבקשת ה-dns הוא פונה לשרת ה-dns המקומי שעל המחשב שלי על ידי פנייה לכתובת 127.0.0.1.
 3. התשובה חוזרת שלילית כי אין שם כזה באף שרת dns ציבורי.
 4. השרת שלי מחזיר את ה-IP שהאפליקציה יושבת עליו בפועל.

רק לאחר קבלת הכתובת משרת ה-dns, הלקוח מתחיל לפעול אל מול האפליקציה.

עצירת העלאה/הורדה באמצע:

לא רצינו לממש עצירה בזמנים קבועים, כמו למשל לתת ללקוח להחליט האם הוא רוצה שהעלאת/הורדת הקובץ תעצור ב-50% או בזמן אחר, מאחר שאנחנו מבינים שזוהי לא המטרה של עצירת ההעלאה/ההורדה. שליחה של קובץ גדול יכולה לנצל את רוב משאבי המחשב, ואם היוזר רוצה לעשות פעולה אחרת במחשב, הוא צריך להיות מסוגל לעצור את הקובץ ולהמשיך את שליחתו לאחר זמן בלתי מוגבל.

ניסינו לממש את עצירת השליחה בצורה אחרת. יש קובץ נוסף שצריך להריץ ביחד עם התוכנה- שמו הוא stopper_thread.py. ה- stopper_thread.py הוא thread שירץ במקביל לאפליקציה שלנו, בתחילת כל פונקציה.

הוא פועל בצורה הבאה-

- בזמן העלאת האפליקציה, השרת stopper_thread יעלה ויפתח קשר ב-tcp עם הלקוח. חשוב לנו מאוד שהקשר יהיה tcp ולא udp, כי חיוני שההודעה תתקבל על ידי השרת ולא תיאבד. אנחנו צריכים לוודא שההודעה הגיעה, אחרת התהליך לא יעצור על אף בקשת הלקוח.
- השרת יתחיל להאזין לבקשות מהלקוח.
- כאשר הלקוח יצטרך להעלות/להוריד קובץ, בתחילת הפונקציות, הוא ישלח הודעה ל- stopper_thread.
- כאשר ה- stopper_thread יקבל את ההודעה מהלקוח, הוא ייצור ויפעיל thread חדש המכיל את הפונקציה הבאה:

```
def to_stop_or_not_to_stop(thread):
    choice2 = ""
    while 1:
        choice = input("do you want to pause the process? Y/N")
        if choice == "Y":
            thread.wait()
            while choice2 != "C":
                choice2 = input("insert the letter C when you want to continue the process")
            thread.notify()
            choice2 = ""
```

האפליקציה הראשית שלנו (ה-Rudpclient) עובדת על thread (בשם guard_thread). משמע, ניתן לעשות לה wait() ו-notify(). הפונקציה wait() מקצה מה-thread שהיא מופעלת עליו את המשאבים של מערכת ההפעלה בכוח, וגורמת לו לישון. הפונקציה notify() מעירה אותו בחזרה, והוא ממשיך בדיוק מאותה נקודה שבה הוא עצר. הפונקציה מקבלת את ה-id של ה-thread עליו רצה האפליקציה הראשית.

- ה- stopper_thread ישאל את היוזר בצורה אינסופית, כל עוד ממשיכה השליחה, האם הוא רוצה לעצור את התהליך. אם היוזר יכניס Y, הפונקציה תכניס את ה-thread הראשי למצב של המתנה. לאחר מכן, ה- stopper_thread אומר ליוזר להכניס את האות C כאשר הוא רוצה להמשיך את התהליך. כאשר הוא יכניס את האות C, ה- stopper_thread יעשה ל-thread הראשי notify(), וההעלאה תמשיך בדיוק מהמקום שהיא עצרה בו.
- כאשר ה-Rudpclient יסיים להעלות/להוריד את הקובץ, הוא ישלח ל- stopper_thread שהוא סיים וה- stopper_thread יפסיק את הפונקציה ויחזור למצב של האזנה עד הפעם הבאה שיקבל הודעה מה-Rudpclient.

לצערנו, נתקלנו בתקלות בעת הנסיון למימוש הרעיון, ובשל חוסר הבנה מספקת של הנושא וחוסר בזמן, לא הספקנו למצוא פתרון לתקלות ונכון לכרגע המימוש לא עובד. בחרנו בכל זאת לצרף את המימוש שלנו לפרוייקט ולהסביר מה ניסינו לעשות, כי לדעתנו זאת דרך מאוד טובה לעצור את התהליך בצורה המיטבית בשביל היוזר. אנו מקווים שלאחר צבירת ידע נוסף, נצליח לתקן ולממש את הפתרון הזה. אז, נכון לכרגע, השרת stopper_thread הוא שרת מאזין בלבד, ולא עוצר בפועל את ה-thread.

סוף סוף הגענו למה שלשמו התכנסנו - האפליקציה:

בתקשורת מחשבים, יעילות ואמינות הם מושגים המתנגשים אחד עם השני. יעילות באה על חשבון אמינות ואמינות באה על חשבון יעילות. כל מפתח של אפליקציה צריך להבין מה הצרכים שלו ולבחור את החיבור בהתאם. למשל, יכול להיות שאנחנו רוצים UDP (הפועל בשיטת "נשלח ונשכח") כי הוא מאוד מהיר, אבל הוא לא מספיק אמין בשבילנו. מצד שני, אנחנו לא זקוקים לרמת האמינות שיש ב-TCP (העובד בשיטת "נוודא ונהנה"), שהוא איטי משמעותית. כך נולד לו ה-RUDP - Reliable UDP.

הוא עובד בצורה הזו- אנחנו משתמשים בחיבור UDP שהוא כמעט החיבור הבסיסי ביותר מבחינת אמינות, ודרך שכבת האפליקציה, נוסף מנגנוני CC, flow control ו-reliability שיהפכו את האפליקציה ליותר אמינה, ועדיין יותר יעילה מחיבור TCP רגיל.

נפרט כעת על כל אמצעי שהוספנו התורם להגברת האמינות.

1. **מבחינת flow control** - אנחנו מימשנו את פרוטוקול stop and wait בגרסה קצת יותר משופרת. כל קובץ שאנחנו שולחים, אנחנו מחלקים ל-chunks, כאשר גודל ה-chunk המינימלי הוא 2048 בייטים. אם צד השרת קיבל את החבילה, הוא ישלח הודעת ACK המכילה מספר סידורי גדול ב-1 מהחבילה שקיבל, מה שיבשר לצד הלקוח, שהשרת קיבל את החבילה. במידה וצד השרת לא קיבל את החבילה לאחר זמן מסוים, השרת ישלח הודעת ACK המכילה את מספר הסידורי של ה-ACK האחרון שהוא שלח (בעצם, הוא לא מקדם את המספר הסידורי, ושולח "כאילו" את החבילה האחרונה שהוא שלח שוב). צד הלקוח ישווה את המספר הסידורי שהוא שלח עם המספר הסידורי שהוא קיבל. אם הם שווים, צד הלקוח ידע שמסיבה מסוימת צד השרת איבד את החבילה. וישלח אותה שוב.
2. **אמרנו קודם שבמידה וצד השרת לא קיבל את החבילה לאחר זמן מסוים, הוא ישלח ACK חוזר. קבענו מה צריך להיות זמן ההמתנה הממוצע בדרך הבאה:**
בצד השרת, כל פעם שמתקבלת פקטה חדשה, אנחנו מאתחלים טיימר שמחשב את הזמן החל מקבלת החבילה ועד הפעם הבאה שאנחנו מקבלים חבילה. כל פעם, נוסף את הזמן שמדדנו למבנה נתונים מסוג list, ונחשב מחדש את ה-RTT_delay הממוצע. הוא זה שיקבע כמה זמן צד השרת ימתין עד שיקבע שהחבילה לא הגיעה, וישלח ACK חוזר.
הוספנו 2 שניות לכל זמן המתנה ממוצע, לצורך טיפול בסטיות תקן.
3. **הודעות ACK:**
הודעות ה-ACK שלנו בנויות כך:

```
def send_ack(self, last_seq_num: int, s: socket, dest, ack_type=16):  
    packet = struct.pack("2h", ack_type, last_seq_num)  
    s.sendto(packet, dest)
```

כל הודעת ACK תכיל שני דברים - המספר הסידורי של החבילה וסוג ה-ACK. אם סוג ה-ACK הוא 16- מדובר ב-ACK רגיל
אם סוג ה-ACK הוא 17- זה אומר שכרגע אנחנו שולחים את ה-ACK האחרון על הפקטה האחרונה. צד השרת מקבל אינדיקציה מצד הלקוח שמדובר ב-chunk האחרון, ובשביל להעביר ללקוח שהוא הבין שמדובר ב-ACK האחרון שהוא שולח, הוא שולח בסוג ה-ACK17

כעת, נפרט על האמצעים שהוספנו התורמים להגברת היעילות: תהליך השליחה:

כאשר מתחילה ההעלאה/ההורדה, גודל ה-chunk המינימלי הוא 2048 בייטים. אנחנו מתחילים במימוש של slow start, על מנת לבחון את יכולות הרשת. כל פעם ששלחנו צ'אנק וקיבלנו ACK, גודל ה-chunk גדל פי 2. נשים לב שגודל ה-chunk לא יעלה מעל ל-64K אף פעם, מאחר ש-64K הוא הגודל המקסימלי שאפשר להעביר בבת אחת בחיבור UDP.
על כל חבילה שלא תגיע, כפי שדיברנו קודם, ישלח ACK חוזר. הצד השולח, שהוא זה שמקבל את הודעות ה-ACK, עוקב אחר ההודעות ופועל בצורה הבאה:

- אם התקבלו כבר 2 ACK כפולים, ה-chunk יפסיק לגדול פי 2 לאחר כל שליחה מוצלחת, ויעבור לגדול פי 1.2, זאת על מנת לאפשר הכלה של מצב הרשת העמוס תוך שימוש בגודל הצ'אנק שיש לנו כרגע. להתהליך הזה קוראים congestion avoidance
 - אם התקבלו 3 ACK כפולים, נאתחל את גודל ה-chunk חזרה להיות 2048 בייטים, ונקפוץ ישר לשלב של congestion avoidance. תהליך זה נקרא fast recovery.
- בשני המצבים, ברגע שקיבלנו ACK חוזר, נשלח את אותה פקטה בשנית.

שליחת הפקטות החוזרות עובדת בצורה הבאה:

במהלך כל שליחה והורדה, אתחלנו מבנה נתונים מסוג list שמכיל tuple בעל 3 שדות. השדה באינדקס 0 מכיל את המספר הסידורי של החבילה, השדה באינדקס 1 מכיל מצביע למיקום שממנו התחלנו לקרוא את הקובץ בשליחת ה-chunk, והשדה באינדקס 2 מכיל מצביע למיקום שבו סיימנו לקרוא.
לתשומת לב הבודק, הסבר נרחב יותר ומפורט יותר על הפרוטוקול reno שאנחנו מימשנו את האפליקציה שלנו על פי העקרונות שלו, נמצא בחלק של ההסבר על פרוטוקול tcp reno ב-PDF.

לפני הצגת הקוד עצמו, כמה הערות מאוד חשובות:

1. חוויתנו תופעה מוזרה כאשר ניסינו להריץ את האפליקציה שלנו תוך כדי שהמחשבים שלנו מחוברים לרשת ה-wifi של האוניברסיטה. כנראה שבהגדרות הרשת, הרשת חוסמת חלק מההודעות בשרת שלנו ולכן האפליקציה כלל לא עבדה. במעבר לרשת wifi פרטית התקלה הסתדרה.
 2. על מנת לקצר תהליכים בהתחברות לשרת שלנו, הגדרנו משתמש דיפולטיבי, שאפשר להיכנס דרכו ובכך לחסוך את ההרשמה לאפליקציה.
משתמש ראשון: username: shahar, password: aaaaaa
איך מעלים את האפליקציה:
 3. בנוסף לסרטון שהוספנו להגשה, המראה איך להפעיל את האפליקציה, נוסיף כאן הוראות בקצרה: ראשית, יש להעלות את ארבעת הקבצים הבאים. נשים לב- חייבים להשתמש ב-sudo בעת הרצת הקבצים! מאחר שלינוקס לא מאפשר להאזין ללא הרשאות מנהל: dhcpserver.py, dnsserver.py, Rudpserver.py, stopper_thread.py.
רק לאחר שארבעת הקבצים באוויר, יש להפעיל את הקובץ החמישי: guard_thread.py.
לאחר מכן, יש להתחבר עם שם המשתמש והסיסמא הדיפולטיביים או להירשם ואז להתחבר.
 4. נציין שבחרנו להריץ את האפליקציה עצמה- הקובץ Rudpclient.py על thread נפרד מה-main. עשינו זאת מאחר שהעבודה על thread נפרד מייצרת לנו סביבה סטרילית להרצת האפליקציה, בצורה שגם נוכל לכבות את השרת אם משהו ממש ישתבש.
- לראייה, לבחירת היוזר, אם בעקבות תנאי רשת קשים או בעיה אחרת האפליקציה נתקעה לגמרי במהלך אחת מהפונקציות, ברוב המוחלט של המקרים, היוזר יוכל ללחוץ על ctrl+c ולהתנתק מהמשתמש שלו באפליקציה. לאחר מכן, הוא יוכל להתחבר מחדש ולהמשיך כרגיל, ללא צורך להקריס את האפליקציה.

איך עובד ה-guard_thread?

כאשר אנחנו מעלים את הקובץ guard_thread.py, אנחנו מאתחלים אובייקט של המחלקה Guardthread, ומפעילים דרכו את הפונקציה file_transfer_app(). בתוך האפליקציה, מוקם thread אחד שיריצ את

```
def file_transfer_app(self):
    thread = threading.Thread(target=self.guard_thread_act())
    dest = ("127.0.0.1", 20334)
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(dest)
    thread.start()
    request = str(thread.ident).encode("ascii")
    if request is not None:
        client.send(request)
    else:
        print("shit")
    print("message sent")

def guard_thread_act(self):
    state = threading.current_thread()
    rudp_client = RUDPClient()
    while 1:
        if self.sign_in():
            rudp_client.client()
        else:
            exit(-1)
```

הפונקציה guard_thread_app(). זה ה-thread הראשי שהולך להריץ לנו את הלוקו של האפליקציה עצמה- ה-Rudpclient. הפונקציה guard_thread_app() מריצה בצורה אינסופית את התהליך הבא- היא שולחת את היוזר שהפעיל את האפליקציה לפונקציה sign_in(). אם היוזר הצליח להתחבר, נריץ את הפונקציה client(), שהיא הפונקציה הראשית ב-Rudpclient שאחראית על כל התהליך מול השרת. אם הוא פורץ (!!!!INTRUDER!), הוא לא יוכל להשתמש באפליקציה שלנו ונסגור את התוכנה. אם הלוקו מעולם לא נרשם לאפליקציה בעבר, הפונקציה תשאל אותו האם הוא רוצה להירשם. בנוסף, לאחר כמה נסיונות שגויים להכנסת הסיסמא, הוא יקבל את האופציה להחליף סיסמא.

איך עובד הקוד:

שרת ה-FTP שלנו תומך במספר פעולות: העלאת קובץ והורדת קובץ עובדות בצורה כמעט זהה. בשתיהן יש צד לקוח וצד שרת, שעובדים אחד עם השני ומיישמים את כל העקרונות של reliability, cc flow control שהגדרתי לעיל.

בתחילת בקשה מכל סוג, הלקוח שולח לשרת פקטה מוגדרת מראש עם השדות הבאים:

```
request_type, padded_file_name, expected_sequence_num, \
data_length = struct.unpack("h 50s h i", data)
```

סוג הבקשה:

0- בשביל upload

1- בשביל download

2- בשביל show_files

3- בשביל remove_file

שם הקובץ - מוגבל לגודל של 50 בייטים

מספר ה-ACK הנוכחי- מתחיל ב-1, מוגדר להיות 0 כאשר אין בו צורך במתודה מסויימת.

גודל הקובץ- רלוונטי רק ל-upload.

ברגע שפקטת הבקשה מגיעה לשרת, הוא קורא לפונקציה המתאימה לפי הנתונים שקיבל בפקטה.

```
while 1:
    try:
        data, address = s.recvfrom(MAX_RECV_BYTES)
        dest = (address[0], client_port)
        request_type, padded_file_name, expected_sequence_num, \
        data_length = struct.unpack("h 50s h i", data)
        if request_type == 0:
            print("hello client I will upload to you the file , in milliseconds")
            self.accept_upload_request(data, s, dest)
        if request_type == 1:
            print("hello client I will download to you the file , in milliseconds")
            self.accept_download_request(data, s, dest)
        if request_type == 2:
            print("hello client I will sent to you the files , in milliseconds")
            self.accept_show_files_request(s, dest)
        if request_type == 3:
            self.accept_remove_file_request(data, s, dest)
    except:
        raise
```

בכל מתודה, נשלחת הודעת ACK שמאשרת את הגעתה של ההודעה הראשונה לשרת.

המתודה accept_upload_request():

בצד הלקוח:

1. שולח את ההודעה הראשונית לשרת ומחכה ל-ACK ממנו
2. כל עוד המצביע שקורא מהקובץ ועוד גודל ה-chunk הנוכחי שאנחנו עתידים לשלוח עדיין קטן מגודל הקובץ:
3. נוסף לרשימה את מיקום המצביע, את המספר הסידורי של הפקטה ואת המיקום הסופי של המצביע לאחר קריאת הבייטים מהקובץ ושליחתם.
4. נשלח את הפקטה ונחכה ל-ACK. כאן נכנס לפעולה העקרון
5. לאחר שקיבלנו את ה-ACK קורה כל מה שדובר לעיל- בהתאם ל-slow start ול-fast recovery גודל ה-chunk גדל, אם הגענו ל-3 ACK כפולים נפעיל תהליך של fast retransmit.
6. נצא מהלולאה כאשר נסיים לשלוח את הקובץ במלואו או כאשר תישאר לנו שארית בייטים שקטנה מגודל ה-chunk שיש לנו כרגע.
7. נשלח פקטה אחרונה, שבה ה-request_type יהיה שווה ל-3-, מה שיבשר גם לשרת שמדובר בפקטה האחרונה.
8. נחכה ל-ACK אחרון מהשרת- סוג ה-ack יהיה הפעם 17, מה שיסמן ללקוח שהשרת קיבל את החבילה האחרונה ולא מצפה לקבל עוד חבילות.

בצד השרת:

1. לאחר שהשרת קיבל את ההודעה הראשונית הוא שולח ACK ראשון.
2. בתוך לולאה אינסופית, נמדוד את הזמן הנוכחי.
3. (עוד לולאה בתוך הלולאה האינסופית) כל עוד לא עברו עדיין `RTT_average_delay` שניות:
(`RTT_average_delay` זה הזמן הממוצע שלוקח למעגל - שליחת ack ללקוח -> שליחת חבילה נוספת מהלקוח לשרת -> קבלת החבילה על ידי השרת -> חוזר חלילה)
4. אם החבילה לא התקבלה, תקפוץ שגיאה שתיתפס ב-`except`, ונישלח חזרה לתנאי הלולאה לבדוק האם כבר עברו `RTT_average_delay` שניות.
- 5.

```
while time.time() <= current_time + average_RTT_delay:
    try:
        packet = None
        packet, address = s.recvfrom(MAX_RECV_BYTES)
    except:
        continue
```

6. אם החבילה לא התקבלה בשלמותה, הפקודה `struct.unpack` תקפיץ שגיאה שתיתפס ב-`except`, ונישלח חזרה לתנאי הלולאה לבדוק האם כבר עברו `RTT_average_delay` שניות. כלומר, אם חבילה לא התקבלה בשלמותה, אנחנו מתייחסים לזה כאילו לא קיבלנו אותה בכלל.

```
try:
    request_type, expected_sequence_num, data_length, file_data \
        = struct.unpack("2h i " + str(chunk_size) + "s", packet)
except:
    continue
```

7. אם סוג החבילה שהתקבלה הוא לא 0 או -3, יש כאן טעות והגיעה אלינו חבילה שלא צריכה להגיע אלינו, אז נצא מהפונקציה.

```
if request_type != 0 and request_type != -3:
    print("FRAUD!!!")
    return
```

8. אם עברנו את כל הבדיקות בהצלחה, נכתוב את ה-`data` מתוך הקובץ בקובץ החדש שפתחנו, נגדיל את גודל ה-`chunk` ונקדם את המספר הסידורי ב-1. לאחר מכן נצא מהלולאה בצורה טבעית, ונגיע לחלק של שליחת ה-`ack`.
9. אם לא עברנו את כל הבדיקות, וכבר עברו `RTT_average_delay` שניות, נצא מהלולאה הפנימית, ונגיע לחלק של שליחת כאשר לא קידמנו את המספר הסידורי ב-1. כך מתבצעת שליחת ה-`ack` הכפול.
10. לבסוף, כאשר נקבל חבילה אשר ה-`request_type` שלה שווה ל-3, נשלח `ack` אחרון אשר ה-`acktype` שלו הוא 17, ונצא מהפונקציה.

הפונקציה `accept_download_request()`:

הפונקציה הזאת עובדת בדיוק באותה צורה כמו הפונקציה `upload`, רק שהפעם, הלקוח הוא צד השרת והוא הצד המקבל, והשרת הוא צד הלקוח, והוא הצד השולח.

הפונקציה `show_files()`:

- הפונקציה `show files` מחזירה ללקוח את רשימת הקבצים השמורים כרגע אצל השרת. איך היא עובדת:
- השרת מקבל את הבקשה הכללית, כאשר סוג הבקשה הפעם הוא 2.
 - השרת פותח קובץ חדש בשם `serverfilelist.txt` באחסון שלו, וכותב לשם את כל שמות הקבצים השמורים אצלו ברשימה.
 - השרת שולח ללקוח `ack`, כאשר ה-`ack_type=18`. זאת על מנת ליצור הפרדה בין הודעות ה-`ack` ולוודא בצורה יותר טובה שכל ה-`ack` הגיעו לאותו המקום.

- הלקוח מקבל את ה-ack, ומשתמש בפונקציה download_request על מנת להוריד את הקובץ הספציפי ששמו serverfilelist.txt. הוא מאחסן את תוכן הקובץ אצלו בקובץ אחר שנקרא clientfilelist.txt.
- כעת, כל מה שנשאר הוא לקרוא מהקובץ את רשימת הקבצים שמאוחסנים אצל השרת, ולהדפיס/להחזיר אותה ללקוח.

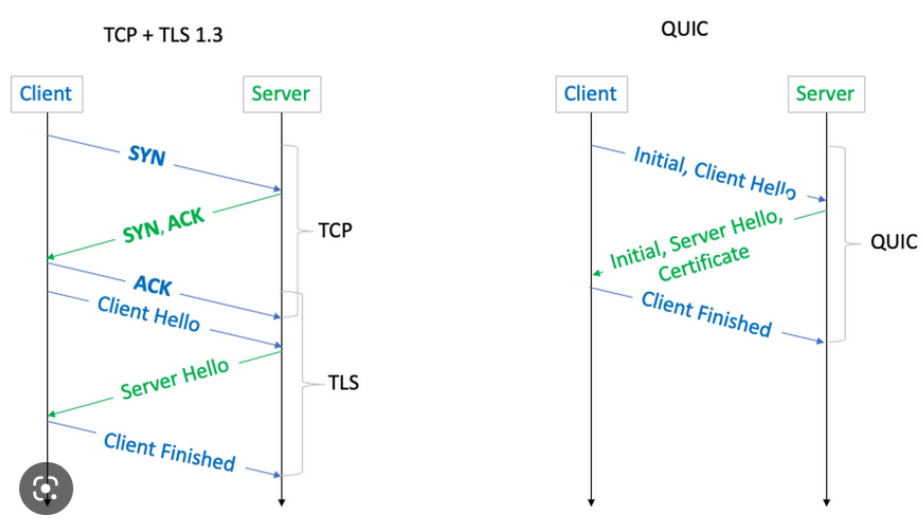
הפונקציה remove file():

- הפונקציה remove file מוחקת קובץ מהשרת, בהוראת הלקוח.
- הלקוח שולח לשרת את פקטת הבקשה הרגילה, כאשר ה-request_type=3. פקטת הבקשה מכילה בנוסף את השם של הקובץ אותו הלקוח רוצה למחוק.
 - השרת מקבל את הבקשה ומוחק את הקובץ מהאחסון שלו ומרשימת הקבצים.
 - השרת שולח ללקוח ack, כאשר ה-ack_type=19. זאת על מנת ליצור הפרדה בין הודעות ה-ack ולוודא בצורה יותר טובה שכל ה-ack הגיעו לאותו המקום.
 - הלקוח משתמש באותו מנגנון שהשתמשנו בו בקבלת הודעות ה-ack בפונקציות upload/download. אם לא התקבל ה-ack עד סוף ה-RTT_average_delay, מודפסת הודעה שהייתה בעיית תקשורת, מה שיסמן לשרת שאי אפשר להיות בטוחים שהקובץ אכן נמחק, ושכדאי לנסות שוב.

שאלות

1. מנה לפחות ארבעה הבדלים עיקריים בין פרוטוקול TCP ל QUIC

א. בפרוטוקול tcp בגרסה הכי חדשה, יש לנו תהליך ארוך מאוד לפתיחת קשר. tcp הינו פרוטוקול שלא מאובטח, אז על גבי לחיצת הידיים של ה-tcp, שהיא כבר משולשת, יש לנו עוד 2 הודעות שקשורות להצפנה, בפרוטוקול tls. פרוטוקול quic ממש מקצר את התהליך הזה ל-3 הודעות בלבד, כאשר ה-syn וה-client hello נשלחים באותה הודעה הראשונה, בהודעה החוזרת מהשרת כבר יש את ה-certificate לצורך ההצפנה ביחד עם ה-server hello וה-syn\ack, וההודעה השלישית היא כבר ack מהלקוח המכריז על פתיחת הקשר.



ב. בפרוטוקול quic, אריזת הפקטות היא קצת שונה מאשר ב-TCP. אנחנו עובדים עם מספר streams שונים שיוצאים מאותה שכבת האפליקציה. כל stream מכיל זרם שונה של בייטים. בסופו של דבר, החבילה כן מכונסת ונשלחת ביחד. אבל גם כשהיעד מקבל את הפקטה, הוא פותח את החבילה מחדש וכל זרם של בייטים הולך ל-stream אחר. אפשר לדמות את התהליך הזה לשליחה של חבילות מעלי אקספרס - נניח שהזמנתי מהרבה מוכרים ביחד, במקום לשלוח לי כל חבילה בנפרד ואני אצטרך ללכת כל פעם לדואר, אורזים לי את כל החבילות בחבילה אחת גדולה ושולחים לי לארץ. ככה, גם אם אחת החבילות הגיעה פגומה, זה לא מפריע לכל האחרות להגיע ביחד איתה. היתרון הגדול בזה, הוא שאם שלחתי מספר זרמים של בייטים ביחד ואחד נפגע, זה לא מפריע לזרמים האחרים להגיע ליעדם, בניגוד ל-TCP, ששם אם חבילה נפגמה/לא הגיעה, קודם כל נעצור, נשלח מחדש ורק אז נמשיך לשלוח עוד מידע.

ג. QUIC בנוי על גבי UDP שהוא פרוטוקול לא אמין במובן של- לא אכפת אם החבילה נמסרה בהצלחה לקצה השני או לא, התפקיד הוא רק לשלוח את החבילה. אבל TCP עצמו הוא פרוטוקול אמין, הוא מוודא שהחבילה שלך תגיע לקצה השני. נציין שגם TCP וגם UDP הם פרוטוקולי שכבת התעבורה, ו-QUIC בנוי על גבי UDP.

ד. כשאנחנו שולחים בקשות HTTP, גם אם נשתמש בהצפנת SSL או TLS, ההצפנה תקפה רק לשכבות שמעל שכבת TCP, מאחר ש-TLS/SSL אחראיים להצפנה של שכבת האפליקציה ולא לשכבות שמתחת. אזי יוצא מצב שה-header של שכבת האפליקציה עדיין חשוף. בפרוטוקול QUIC הפקטה בנויה אחרת, ככה שכמעט הכל מוצפן, חוץ מה-checksum, פורט היעד, פורט המקור וגודל הפקטה.

2. מנה לפחות שני הבדלים עיקריים בין Vegas ל Cubic

א. האלגוריתם Vegas מודד את ה-RTT של פקטות כדי לקבוע את העומס ברשת. כלומר, אם ה-RTT הוא ארוך יותר מהממוצע, נדע להגיד שיש עומס, ואם ה-RTT קטן מהממוצע, נקבע שיש פחות עומס. אלגוריתם cubic לעומת זאת, בדומה ל-reno, משנה את גודל החלון בצורה דינמית, לפי תקינות של הפקטות שנשלחות ומגיעות ברשת.

ב. האלגוריתם TCP VEGAS מעדיף עיכוב כללי בשליחת הפקטות על פני איבוד פקטות - הוא יעדיף לשלוח לאט מאשר לשלוח מהר ולאבד פקטות בעקבות כך. לעומת זאת, פרוטוקול cubic מגדיל את החלון בצורה מאוד קיצונית ביחס גם ל-tcp reno ול-tcp vegas, מה שגורם לאיבוד יותר משמעותי של פקטות בכל שליחה, אבל גם מאפשר רוחב פס יותר גבוה.

3. הסבר מהו פרוטוקול BGP, במה הוא שונה מ OSPF והאם הוא עובד על פי מסלולים קצרים

פרוטוקול BGP הוא פרוטוקול ניתוב, והוא הפרוטוקול העומד בבסיס מערכת הניתוב העולמית של האינטרנט. BGP יוצר יציבות רשת על ידי הבטחה שנתבים יכולים להסתגל לכשלים במסלול: כאשר נתיב אחד יורד, נתיב חדש נמצא במהירות. במסגרת הפרוטוקול, כל ראوتر מקבל החלטות ניתוב על סמך נתיבים, המוגדרים על ידי כללים או מדיניות רשת שנקבעו על ידי מנהלי רשת ידנית. כלומר, הוא ימצא את המסלולים הקצרים ביותר על בהתחשב בהגבלות שמנהל הרשת נתן לו.

איך זה עובד?

כל נתב שומר על טבלת ניתוב השולטת כיצד לנתב את החבילות. מידע על טבלת ניתוב נוצר על ידי תהליך BGP בנתב, בהתבסס על מידע הנכנס מנתבים אחרים, והמידע שכבר עכשיו נמצא בטבלת הניתוב.

פרוטוקול OSPF:



פרוטוקול OSPF הוא פרוטוקול ניתוב דינמי מסוג link state זאת אומרת שהוא תמיד מחזיק מפה מלאה של טיפולוגיית שרת.

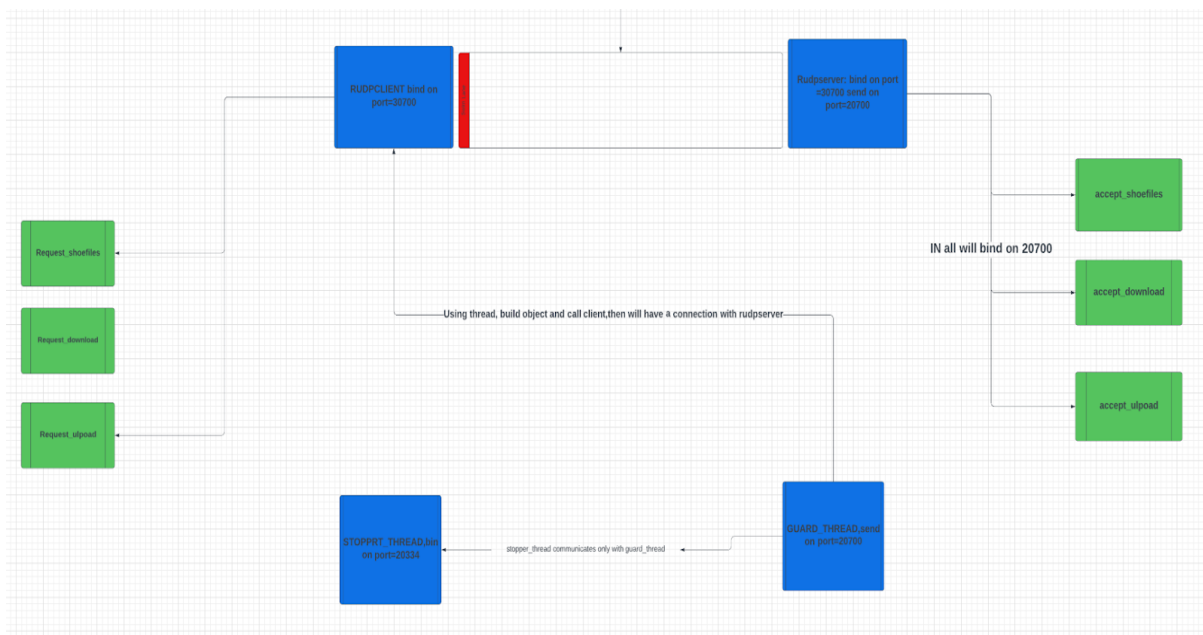
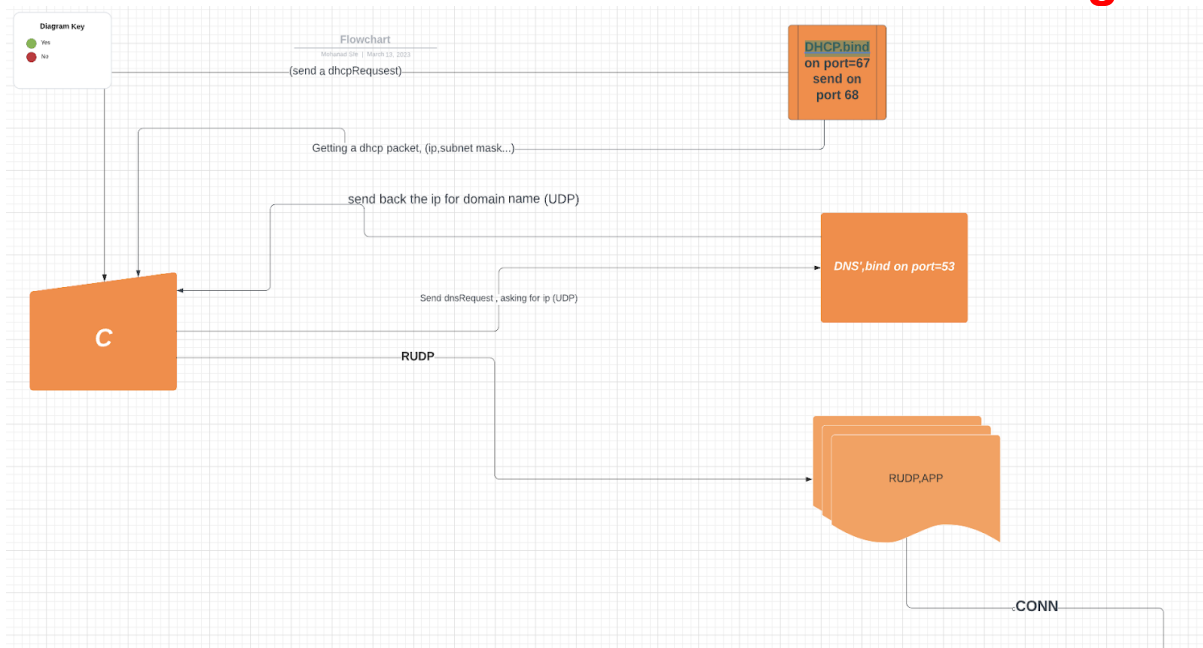
הפרוטוקול הזה עובד ברשתות פנימיות בעיקר, כאשר פרוטוקול BGP הוא פרוטוקול שמחבר בין הרשתות הפנימיות. פרוטוקול OSPF מוצא את המסלולים הכי קצרים בהתחשב בספקיות האינטרנט של הרשת המקומית.

4. מה ההבדלים בין פרוטוקול ARP ל DNS:

1. פרוטוקול DNS פועל בשכבת האפליקציה. לעומת זאת, פרוטוקול ARP פועל בשכבה הפיזית.

2. פרוטוקול DNS מקבל שם של דומיין מסויים וממיר אותו לכתובת IP לעומת זאת, פרוטוקול ARP ממפה בין כתובות IP לכתובות MAC

UML diagram



ביבליוגרפיה -

<https://www.geeksforgeeks.org/tcp-tahoe-and-tcp-reno/?ref=rp>

-: For DHCP

https://github.com/playma/simple_dhcp/blob/master/dhcp_server.py

-:For DNS

<https://www.digitalwhisper.co.il/files/Zines/0x73/DW115-4-DHCP-Spoofing.pdf>

For UML :-

https://lucid.app/lucidchart/bff587a4-dc1e-4675-8039-f2e90421835c/edit?viewport_loc=760%2C1993%2C4225%2C2206%2C0_0&invitationId=inv_5ad322ab-0843-4df1-8fcd-9e8158d9e137

הקלטות wireshark עבור ניתוח של איבוד פקטות:

5 אחוז איבוד פקטות

```
(finalfinalprojectT) shahar@shahar-X442UQR:~/Downloads/finalfinalprojectT$ sudo python3 guard_thread.py
please insert user name: shahar
please insert your password: aaaaaa
DHCP client is starting...

sending DHCP discovery...
received DHCP offer
sending DHCP request...
got IP address and ready to start communicate :)
received DHCP ack

Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit 1
Hello user ,You chose to work with upload
Please enter name of the file you want to upload abcde.txt
Please enter name of the new fileabcde1.txt
startover
finished with: 127.0.0.1

Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit 1
Hello user ,You chose to work with upload
Please enter name of the file you want to upload abcde.txt
Please enter name of the new fileabcde1.txt
finished with: 127.0.0.1

Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit
```

בתמונה מעל, נוכל לראות שאנחנו מריצים את הקובץ gurad_thread.py רק לאחר הרצת 4 הקבצים שכתובים כאן. כך אנחנו ניגשים לאפליקציה.

dnsserver.py, dhcpserver.py, stopper_thread.py, RudpClient.py, Rudpservice.py

לאחר מכן, היוזר יכניס את שם המשתמש והסיסמא ויתחבר. לאחר שהוא יתחבר, הוא מיד יקבל רשימה של כל הפונקציות שהאפליקציה תומכת בהן כגון:- מחיקה, הורדה, העלאה של קובץ וגם פעולה שמראה את כל הקבצים שיש בצד השרת.

בדוגמא המצולמת במסך הטרמינל, יש שימוש באופציה של UPLOAD.

ההקלטה של המקרה הנזכר מעל (עם איבוד של 5%)

5%loss_dow_recording.pcapng

Apply a display filter ... <=>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.1.1	UDP	104	20700 → 30700 Len=60
2	0.000253187	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
3	0.000370574	127.0.0.1	127.0.1.1	UDP	104	20700 → 30700 Len=60
4	0.000450345	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
5	0.000535614	127.0.0.1	127.0.0.1	UDP	2100	30700 → 20700 Len=2056
6	0.000605224	127.0.0.1	127.0.1.1	UDP	48	20700 → 30700 Len=4
7	9.347256360	127.0.0.1	127.0.1.1	UDP	104	20700 → 30700 Len=60
8	9.347943891	127.0.0.1	127.0.0.1	UDP	2100	30700 → 20700 Len=2056
9	11.348535175	127.0.0.1	127.0.1.1	UDP	48	20700 → 30700 Len=4
10	11.348634410	127.0.0.1	127.0.0.1	UDP	2100	30700 → 20700 Len=2056
11	11.348662211	127.0.0.1	127.0.1.1	UDP	48	20700 → 30700 Len=4
12	11.348739435	127.0.0.1	127.0.0.1	UDP	4148	30700 → 20700 Len=4104
13	11.348786550	127.0.0.1	127.0.1.1	UDP	48	20700 → 30700 Len=4
14	11.348854158	127.0.0.1	127.0.0.1	UDP	8244	30700 → 20700 Len=8200
15	11.348903815	127.0.0.1	127.0.1.1	UDP	48	20700 → 30700 Len=4
16	11.348962020	127.0.0.1	127.0.0.1	UDP	164...	30700 → 20700 Len=16392
17	11.349002064	127.0.0.1	127.0.1.1	UDP	48	20700 → 30700 Len=4
18	11.349041819	127.0.0.1	127.0.0.1	UDP	328...	30700 → 20700 Len=32776
19	11.349225331	127.0.0.1	127.0.1.1	UDP	48	20700 → 30700 Len=4

> Frame 1: 104 bytes on wire (832 bits), 104 bytes captured (832 bits) on interface any, id 0
 > Linux cooked capture v1
 > Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.1.1
 > User Datagram Protocol, Src Port: 20700, Dst Port: 30700
 > Data (60 bytes)

קודם כל , רואים שיש העברה של הפקטות מ $src = 127.0.0.1$ ו $dst = 127.0.0.1$ עם גודל פאקטה משתנה.

portServer = 30700 , portClient = 20700

הפאקטה בשורה הראשונה שרואים בתמונה מעל, היא פקטת הבקשה של הלקוח לשרת, והיא בגודל 104 בייטים.

ניתן לראות שלאחר פקטת הבקשה כמו לאחר כל פקטה שליחה אחרת מהלקוח לשרת, נשלחת חזרה הודעת ACK בגודל 48 בייטים.

לאחר מכן, נוכל לראות שהשליחה עובדת כמו שצריך, ושה-chunk גדל בהתאם למימוש של slow start.

download loss 5%

5%loss_recording.pcapng

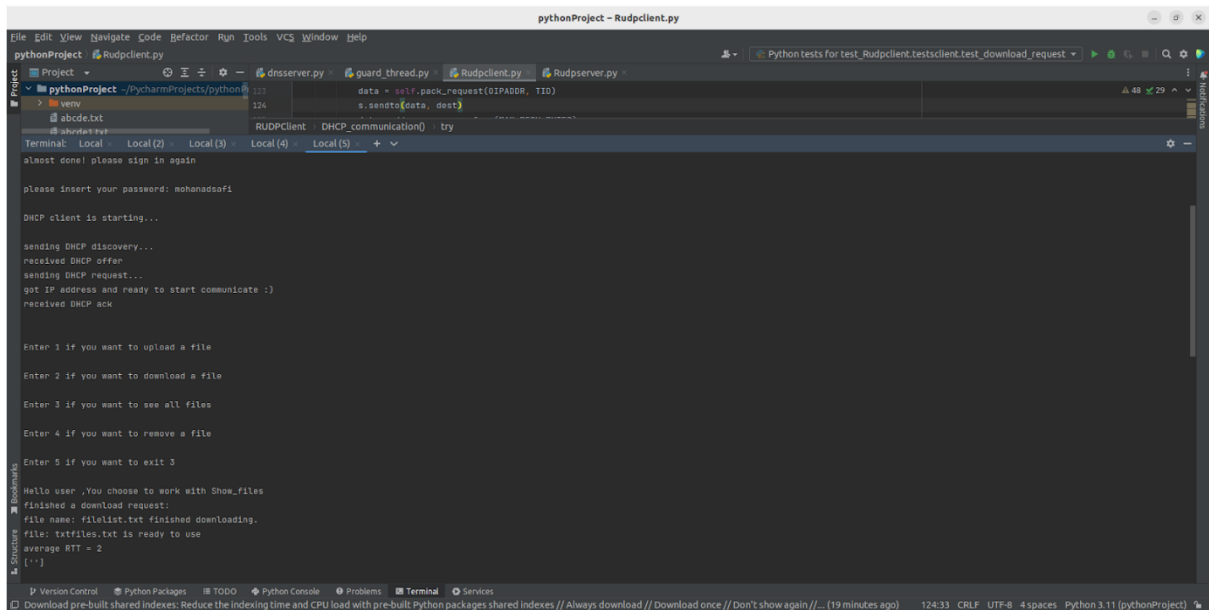
Apply a display filter ... <=>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.1.1	UDP	104	20700 → 30700 Len=60
2	0.000215551	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
3	0.000435846	127.0.0.1	127.0.1.1	UDP	2100	20700 → 30700 Len=2056
4	0.000484071	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
5	0.000592098	127.0.0.1	127.0.1.1	UDP	4148	20700 → 30700 Len=4104
6	0.000663665	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
7	0.000776874	127.0.0.1	127.0.1.1	UDP	8244	20700 → 30700 Len=8200
8	0.000868704	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
9	0.000991095	127.0.0.1	127.0.1.1	UDP	164...	20700 → 30700 Len=16392
10	0.001069077	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
11	0.001161058	127.0.0.1	127.0.1.1	UDP	328...	20700 → 30700 Len=32776
12	0.001230071	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4

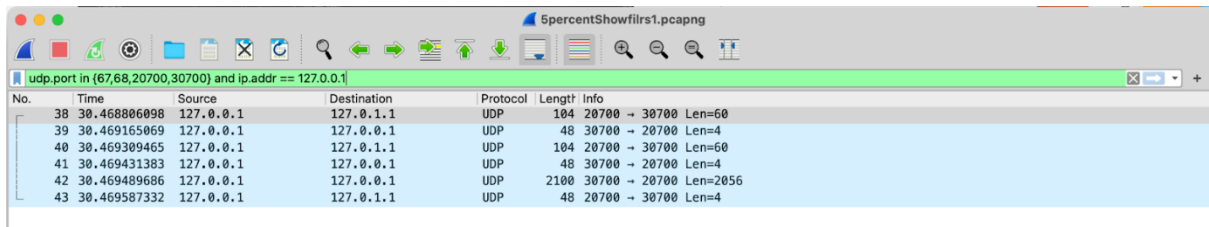
בהקלטה הזו מוקלט תהליך הורדה.

תהליך ההורדה זהה לתהליך ההעלאה, רק שבתהליך ההורדה, הלקוח הוא צד השרת שמקבל את הקובץ והשרת הוא צד הלקוח ששולח את הקובץ. נוכל לראות גם כאן שאין בעיות משמעותיות בשליחה ושהכל עובד כמתוכנן גם ב-5% איבוד פקטות.

Show_files 5%



```
pythonProject - Rudclient.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject Rudclient.py
Project
pythonProject - PycharmProjects/pythonProject
venv
abcde.txt
RUDClient DHCP_communication() try
Terminal: Local Local (2) Local (3) Local (4) Local (5)
almost done! please sign in again
please insert your password: mohanadsafi
DHCP client is starting...
sending DHCP discovery...
received DHCP offer
sending DHCP request...
got IP address and ready to start communicate :)
received DHCP ack
Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit 3
Hello user ,You choose to work with Show_files
finished a download request:
file name: fileList.txt finished downloading.
file: txtfiles.txt is ready to use
average RTT - 2
['']
Python tests for test_Rudclient.testclient.test_download_request
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always download // Download once // Don't show again // ... (19 minutes ago) 124:33 CRLF UTF-8 4 spaces Python 3.11 (pythonProject)
```



No.	Time	Source	Destination	Protocol	Length	Info
38	30.468806098	127.0.0.1	127.0.1.1	UDP	104	20700 → 30700 Len=60
39	30.469165069	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
40	30.469309465	127.0.0.1	127.0.1.1	UDP	104	20700 → 30700 Len=60
41	30.469431383	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
42	30.469489686	127.0.0.1	127.0.0.1	UDP	2100	30700 → 20700 Len=2056
43	30.469587332	127.0.0.1	127.0.1.1	UDP	48	20700 → 30700 Len=4

במקרה הזה, יש שליחה של בקשה מהלקוח שמבקש את הרשימה של הקבצים הנמצאים אצל השרת, ואז השרת יחזיר את הרשימה בחזרה דרך ה-socket.

ואז רואים פקטה הראשונה שהיא בעצם שליחת הבקשה (בשתי הפקטות בגודל 104) וכל אחת קיבלה ack .

ולאחר מכן, רואים שיש פקטה שהיא בעצם בגודל 2100 , שזו הרשימה הריקה שרואים אותה בהדפסה שבטרמינל.

יש כאן הקלטה נוספת של showfiles ב-5% איבוד, כדי להראות גם את הסיטואציה שבה רשימת הקבצים אינה ריקה. לשם כך, קודם העלנו קובץ לשרת.

```
pythonProject - RudpcClient.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject - RudpcClient.py
Project: pythonProject - PycharmProjects/pythonProject
venv 124 data = self.pack_request(01PADDR, IID)
abode.txt s.sendto(data, dest)
abode.py: RudpcClient DHCP_communication() try
Terminal: Local Local (2) Local (3) Local (4) Local (5) + v
Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit 1
Hello user ,You chose to work with upload
Please enter name of the file you want to upload abode.txt
Please enter name of the new fileabode1.txt
Finished with: 127.0.0.1
Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit 3
Hello user ,You chose to work with Show_files
Finished a download request:
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built python packages shared indexes // Always download // Download once // Don't show again // (19 minutes ago) 12433 CRLF UTF-8 44spaces Python 3.11 (pythonProject)
```

```
pythonProject - RudpcClient.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonProject - RudpcClient.py
Project: pythonProject - PycharmProjects/pythonProject
venv 124 data = self.pack_request(01PADDR, IID)
abode.txt s.sendto(data, dest)
abode.py: RudpcClient DHCP_communication() try
Terminal: Local Local (2) Local (3) Local (4) Local (5) + v
Please enter name of the new fileabode1.txt
Finished with: 127.0.0.1
Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit 3
Hello user ,You chose to work with Show_files
Finished a download request:
File name: filelist.txt #finished downloading.
File: txtfiles.txt is ready to use
average RTT = 2
['abode1.txt']
Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built python packages shared indexes // Always download // Download once // Don't show again // (19 minutes ago) 12433 CRLF UTF-8 44spaces Python 3.11 (pythonProject)
```

No.	Time	Source	Destination	Protocol	Length	Info
40	33.689809605	127.0.0.1	127.0.1.1	UDP	104	20700 → 30700 Len=60
41	33.690132038	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
42	33.690346087	127.0.0.1	127.0.1.1	UDP	2100	20700 → 30700 Len=2056
43	33.690397425	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
52	39.431008056	127.0.0.1	127.0.1.1	UDP	104	20700 → 30700 Len=60
53	39.431524362	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
54	39.431644246	127.0.0.1	127.0.1.1	UDP	104	20700 → 30700 Len=60
55	39.432018557	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
56	39.432134583	127.0.0.1	127.0.0.1	UDP	2100	30700 → 20700 Len=2056
57	39.432275546	127.0.0.1	127.0.1.1	UDP	48	20700 → 30700 Len=4

בשורות 40 עד 44 ניתן לראות שיש פקטה של בקשה וה-ack שלה, ואז שליחה של המידע (שזה העצם הבקשה של ההעלאה).

ומיד רואים עוד פעם שיש עוד פקטה של בקשה ל showfiles שגם כן קיבלה את ה ack שלה , ואז המידע נשלח מהשרת ללקוח ומתקבל ה-ack מהלקוח לשרת. הפונקציה showfiles פועלת בעזרת download, וההקלטה הזו ממחישה היטב את ההתהפכות של צד השרת וצד הלקוח. כעת, השרת שולח ללקוח הודעות, והלקוח שולח לשרת את הודעות ה-ack.

10% איבוד פקטות

```
Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit 1
Hello user ,You chose to work with upload
Please enter name of the file you want to upload abcde.txt
Please enter name of the new fileabcde1.txt
startover
startover
startover
finished with: 127.0.0.1

Enter 1 if you want to upload a file
Enter 2 if you want to download a file
Enter 3 if you want to see all files
Enter 4 if you want to remove a file
Enter 5 if you want to exit
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	104	20700 → 30700 Len=60
2	0.000571540	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
3	0.000837678	127.0.0.1	127.0.0.1	UDP	2100	20700 → 30700 Len=2056
4	0.000938224	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
5	0.001092752	127.0.0.1	127.0.0.1	UDP	4148	20700 → 30700 Len=4104
6	0.001689590	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
7	0.001910731	127.0.0.1	127.0.0.1	UDP	8244	20700 → 30700 Len=8200
8	4.001158908	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
9	4.001268581	127.0.0.1	127.0.0.1	UDP	164...	20700 → 30700 Len=16392
10	6.001443710	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
11	6.001557063	127.0.0.1	127.0.0.1	UDP	164...	20700 → 30700 Len=16392
12	8.001726526	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
13	8.001837156	127.0.0.1	127.0.0.1	UDP	197...	20700 → 30700 Len=19668
22	11.002036604	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
23	14.002347298	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
24	17.002656712	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
25	17.002783254	127.0.0.1	127.0.0.1	UDP	2100	20700 → 30700 Len=2056
26	17.002848943	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
27	17.002910702	127.0.0.1	127.0.0.1	UDP	4148	20700 → 30700 Len=4104
28	17.002989097	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
29	17.003064389	127.0.0.1	127.0.0.1	UDP	8244	20700 → 30700 Len=8200
30	17.003150167	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4
31	17.003264089	127.0.0.1	127.0.0.1	UDP	164...	20700 → 30700 Len=16392
32	17.003317338	127.0.0.1	127.0.0.1	UDP	48	30700 → 20700 Len=4

> Frame 32: 48 bytes on wire (384 bits), 48 bytes captured (384 bits) on interface any, id 0
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 30700, Dst Port: 20700
> Data (4 bytes)

בהקלטה של איבוד פקטות 10% ניתן לראות בצורה הטובה ביותר את ההתמודדות של האפליקציה שלנו עם איבוד פקטות. נוכל לראות שבשורות 1-13 השליחה מתנהלת כרגיל, וה-chunk גדל כל פעם כמתוכנן. אבל אז החל משורה 14 עד 16 יש לנו אירוע של איבוד כמה פקטות. צד השרת ישלח ack כפולים המסמנים על כך שהפקטה לא הגיעה. כאשר נגיע ל-3 ack כפולים, מבחינת האפליקציה שלנו זאת אינדיקציה מספיק טובה לכך שהפקטות כבר לא יגיעו. אז גודל החלון מתאפס לגודל הדיפולטיבי שלו, ואנחנו נכנסים ל-slow start מחדש. נשים לב שגם בטרמינל נוכל לראות את השגיאה- כאשר מודפס startover למסך.