

ספר פרויקט - רשתות תקשורת

מגישים:

אליאן אילוק 214787483

אריאל לבוביץ 326535242

רואי סיבוני 214662439

תכלת שבח 212671358

הבהרות והערות לקוד ולקובץ:

1. בנוגע לתוצאות של ההרצות עם אחוז איבוד שונה בין כל הרצה, נשמים לב כי הזמן והמהירות של קצב שליחת הפקטות נשאר קבוע. אנחנו מאמינים שזה נובע מהאלגוריתם שליחה מחדש של פקטות שהוגדרו כאבודות, שבעצם לא מחכה כמות זמן מסוימת וישר שולח וכך אין הבדל מהותי בין כל הרצה
2. בתיעוד הקוד תוכלו לשים לב לפונקציות שתוכננו להיבנות לתקשורת מרובת משתמשים, בפועל בפרוייקט זה לא השתמשנו בהם כי התבקשנו לבצע תקשורת עם קליינט יחיד.
3. רצינו להוסיף ולהדגיש שעקב הרצון שלנו שהכל יהיה ברור, הדפסנו הרבה מקומות בתהליך השליחה של הפקטה כדי להמחיש באיזה מקום אנחנו נמצאים בקוד בזמן ריצה וכתוצאה מכך זמן ריצת התוכנית גדל באופן משמעותית כלומר עם הדפסות הריצה יכולה לקחת כ- 90 שניות בזמן שהריצה בלי ההדפסות יכולה לקחת מספר שניות בודדות (כ- 10 שניות). (הסטטיסטיקות שמוצגות בפרוייקט הן על התוכנית עם ההדפסות ואנחנו בודקים על כמות הפקטות שנשלחו כי זה מקביל למדידת זמנים)
4. קובץ ההסברים של quicFunc מצורף בpdf אחר מכיוון שהדוקס לא הצליח להכניס אותו

Table of contents:

חלק יבש.....	3
TestQuicChat.....	6
Server.py.....	8
Client.py.....	14
QuicPackage.py.....	21
QuicFunc.py.....	24
User Manual for GUI.....	27
Results for resend by seq:.....	32
results running by time.....	35
Running by time + sequence number:.....	41
Wireshark recording- explanation.....	47

חלק יבש

1- חמש חסרונות של TCP:

- א- כחלק מפרוטוקול TCP כאשר פקטה כלשהיא הולכת לאיבוד, יש צורך בלשחזר אותה ולשלוח אותה פעם נוספת, בזמן הביניים בין שמגלים שפקטה אכן אבדה ועד שמשחזרים אותה ושולחים אותה פעם נוספת, כל הפקטות שנמצאות כבר בתוך הרשת, יוצאות מהרשת "והולכות לאיבוד" גם כן.
- ב- כאשר פקטה מסוימת הולכת לאיבוד, פרוטוקול TCP יעצור וימנע מפקטות אחרות להמשיך לעבור ברשת עד שנשחזר את הפקטה האבודה ונשלח אותה פעם נוספת ונקבל עליה אישור שהיא הגיעה בהצלחה.
- ג- ישנו דילימי כאשר יוצרים את קשר ה-TCP מכיוון שצריך לבצע לחיצת ידיים משולשת. במידה ואחד הצדדים מעוניין לאבטח את הקשר צריך לבצע לחיצת ידיים פעם נוספת. ניתן להבין כי כל לחיצת ידיים לוקחת זמן ומעכבת את יצירת הקשר והתחלת העברת המידע.
- ד- header של פרוטוקול TCP ישנם מספר שדות כאשר כל אחד מוגבל לכמות ביטים מסוימת. בנוסף לשדות הרגילים, יש גם שדה (בגודל 40 בטים) שהוא עבור שדה אופציונלי. כל אחד מהשדות הוא קטן מאוד (2-4 בטים לכל אחד) ולכן הגודל הקטן של השדות האלה מגביל את הביצועים של פרוטוקול TCP כאשר עובדים במהירות גבוהה מאוד.
- ה- פרוטוקול TCP משתמש ובנוי על מספרי פורטים IP ואלו דברים שבמהלך החיים יכולים להשתנות, ולכן במידה שאכן אחד מהם משתנה זה יגרום לקשר הקיים להתפרק ולכל המידע שעבר עד כה להיאבד. על מנת להתגבר על זה יש צורך בלחיצת ידיים משולשת נוספת.

2- חמש תפקידים שכל פרוטוקול תעבורה צריך למלא הם

- א- הגדרת מזהה חיבור ומזהה נתונים
- ב- העברת מידע אמינה עם בקרת זרימה מבוססת חלון-זמן, עושים זאת כך שלכל פקטה שעוברת שמים מזהה כלשהוא (בדרך כלל מספר סידורי) וכך ניתן לעקוב אחר כל הפקטות ולראות אם כולם הגיעו בשלמותם כמו שצריך או שפקטה מסוימת אבדה וצריך להתמודד עם זה.
- ג- Congestion control על מנת להגביל את מספר הפקטות שנמצאות בתוך הרשת
- ד- ניהול חיבורי תעבורה- חיבור ייחודי בין 2 צדדים, יצירה ופירוק של קשר, שליטה על העברת המידע בין 2 הצדדים, שינוי כתובת ה-IP בהתאם לשינויים מסוימים.
- ה- אבטחת המידע שעובר בין 2 הצדדים

3- לחיצת הידיים של QUIC

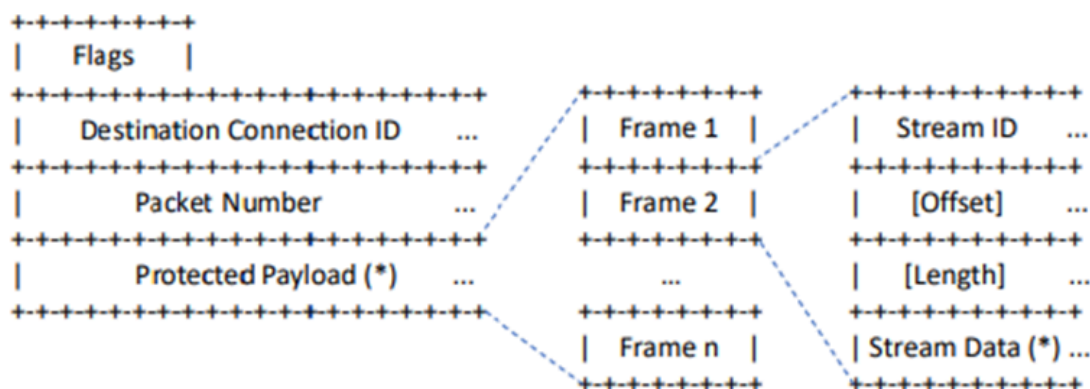
בלחיצת הידיים של פרוטוקול QUIC אנחנו משלבים בין לחיצת ידיים תעבורתית ולחיצת ידיים קריפטוגרפית ביחד, כך אנו יכולים לקבל את כל המידע הדרוש לנו בזמן של RTT אחד בלבד וזה מביא למינימום זמן את זמן פתיחת הקשר המאובטח בין הצדדים. על מנת לקבל את ה-ID של כל אחד מהצדדים, הפרוטוקול שולח פקטה ראשונית כאשר כל צד יאכלס את השדה שלו בפקטה עם ה-ID שהוא מעוניין בו לשליחה וקבלה של פקטות נוספות בעתיד. לאחר קבלת הפקטה הראשונית, השרת יכול(אם הוא רוצה בכך) לבחור לשלוח פקטה נוספת שתכיל טוקן מסוים שהלקוח יצטרך להשתמש בו שוב על מנת להמשיך בתהליך לחיצת הידיים. בנוסף לכך, הודעות לחיצת ידיים של ה-TLS גם כן נמצאות בתוך הפקטות הראשוניות האלה על מנת להבטיח שהקשר ושליחת הפקטות העתידיות יהיה מאובטח ב-RTT אחד בלבד.

תהליך לחיצת ידיים זאת משפר את החסרונות שיש ל-TCP בכך שלעומת TCP שלא יכול להתאושש משינוי פורט או כתובת IP מסוים, QUIC יכול לשרוד את השינויים האלה בעזרת השימוש ב-ID של כל צד, לאחר השינוי הצד שהשתנה יכול לשלוח פקטה עם הכתובת החדשה להמשיך הקשר, ועל מנת לאשר את השינוי הצד השני יבצע בדיקה שניתן להמשיך את שליחת הפקטות והמשיך הקשר(שולחים מידע רנדומלי לכתובת החדשה ומחכים לתגובה שתכלול את אותו המידע הרנדומלי שנשלח בדיוק). בנוסף לכך, המידע שעבר לא ילך לאיבוד כמו מה שקורה ב-TCP אלה ממשיכים מאותו מקום בדיוק והמידע נשמר.

4- מבנה החבילה של פרוטוקול QUIC

לפרוטוקול זה יש 2 סוגים של headers, בהתחלה משתמשים בפקטות ארוכות לחיבור הקשר בין הצדדים והן צריכות להכיל כמה שדות שיכילו מידע שקשור ללחיצת הידיים. לאחר שהחיבור עבד, ישנם מספר שדות שאין בה יותר שימוש ולכן ניצור header אחר שיכיל רק את השדות הרלוונטיים מפה והלאה(וזאת על מנת לייעל את העברת הנתונים ולחסוך בזיכרון). כל פקטה תכיל מספר ייחודי שתסמל את המספר הסידורי של הפקטה שיסביר את סדר הפקטות שמועברות ויעזור לנו לשים לב במידה ופקטות הולכות לאיבוד ולא מגיעות ליעד, וזה גם יכול להגיד לנו בכל רגע נתון כמה פקטות יש בתוך הרשת. בנוסף, לעומת השדה האופציונלי של ה-TCP שיכול להכיל עד 40 בתים ולכן להכיל רק עד 3 ACKS, QUIC יכול לתמוך בעד 256 סוגים שונים של ACKS וכך מבנה החבילה של QUIC משפר את ה-TCP.

מבנה החבילה המקוצר של QUIC:



5- כאשר חבילות מגיעות באיחור או לא מגיעות בכלל, פקטה נחשבת בתור פקטה שנאבדה במקרה ack עבור פקטה "מאוחרת" יותר התקבלה בזמן שאכן ack עבור הפקטה הזו עדיין לא התקבלה וכאשר threshold מסוים התקבל. על מנת לאתר איבודי פקטות, פרוטוקול QUIC משתמש ב-2 שיטות:

א- על ידי מספור של הפקטות (sequence number)- כאשר המספר הסידורי של הפקטה שהתקבלה קטן מהack האחרון שהתקבל, כל הפקטות שמספרם הסידורי הם בין המספר הסידורי של הפקטה האחרונה שהתקבלה לבין האכן האחרון שהתקבל נחשבים "אבודים" וצריך לשלוח אותם מחדש.

ב- על ידי זמן השליחה של הפקטות- כאשר פקטה מסוימת מקבלת ack אזי מודדים את הזמן הגעה שלה, threshold שלנו הוא הזמן שאנחנו נותנים לפקטה להגיע לפני שאנחנו מכריזים עליה כעל פקטה "אבודה". במידה ומאז שקיבלנו את הפקטה האחרונה, הזמן שעבר הוא גדול יותר מהthreshold שהגדנו אזי אנחנו מכריזים על הפקטה הבאה כעל פקטה "אבודה" ואנו צריכים לשלוח אותה מחדש.

כאשר גילינו שפקטות מסוימות הלכו לאיבוד, המידע מוכנס לפקטות חדשות עם מספר סידורי חדש(ללא קשר לאיבוד פקטות) והן נשלחות מחדש.

6- בקרת העומס (congestion control) של פרוטוקול QUIC עובדת כך-

כמו TCP גם בפרוטוקול QUIC יש חלון מסוים שמהווה גבול מסוים של מספר הביטים שהלקוח יכול לשלוח לשרת כל פעם ובכך העברת המידע לא "מציפה" את הרשת ומעבירה את המידע במידתיות ובכך ניתן לאפשר העברת מידע בצורה אופטימלית בין הלקוח לשרת ולווסת את מספר הביטים שנמצאים בתוך הרשת בכל פעם כך שהקשר יפעל בצורה אופטימלית.

TestQuicChat

מחלקה זו מיועדת לבדיקת התקשורת באמצעות QUIC וכוללת מספר בדיקות יחידה לאימות פונקציונליות שונות של שליחה וקבלת חבילות תקשורת.

פונקציות במחלקה:

setUp

הגדרות ראשוניות לפני כל בדיקה. מאתחלת כתובת לקוח לשימוש בבדיקות.

test_send_package

בדיקה זו מאמתת שחבילה אחת נשלחת בהצלחה מהלקוח לשרת. הבדיקה משתמשת במתודה send_package מהמודול QuicFunc. היא מוודאת שהפונקציה sendto של מופע הסוקט נקראה ושהכתובת שאליה נשלחה החבילה מתאימה לכתובת שהוגדרה בהגדרות המבחן.

test_send_packages_from_file

בדיקה זו בודקת את יכולת המערכת לשלוח רצף של חבילות נתונים מקובץ. הבדיקה מדמה קריאה מקובץ עם נתונים מסוימים ובודקת שהחבילות נשלחות כראוי דרך הפונקציה send_packages_from_file. היא משתמשת במודול MagicMock לדמיית פעולת הקובץ.

test_receive_package

בדיקה זו בודקת את תהליך קבלת חבילה ועיבודה. היא משתמשת במודול patch לדמיית קבלת חבילה מהרשת ולאחר מכן בודקת שהתוכן של החבילה שנקלטה תואם למה שציפינו לקבל.

test_resend_lost_packages

בדיקה זו מאמתת שחבילות שאבדו במהלך השידור נשלחות מחדש. הבדיקה מגדירה חבילה כאילו היא אבדה ולאחר מכן בודקת שפונקציית השליחה מחדש resend_lost_packages פועלת כשורה.

test_incorrect_package_format

בדיקה זו נועדה לאתר ולהתמודד עם חבילות בפורמט שגוי. היא בודקת שהמערכת מעלה שגיאה במקרה של נתונים לא תקינים, מה שמדגים את יכולת המערכת לזהות ולנהל כשלים בפורמט של הנתונים.

test_connection_timeout

בדיקה זו בודקת כיצד המערכת מגיבה לטיימאאוט של החיבור. היא משתמשת ב-patch לדמיית טיים-אוט על פעולת קבלת נתונים מהסוקט, ומוודאת שהמערכת מזהה ומטפלת בזמן המתנה החורג.

test_full_message_cycle

בדיקה זו מבצעת סימולציה של שליחה, קבלה ואישור של חבילה, בדוקה שהמעגל השלם של תקשורת עובד כשורה. היא משלבת שליחה, קבלה ואישור כדי לבדוק את יכולת המערכת לטפל בתהליכים רבים במקביל.

test_edge_case_package_handling

בדיקה זו מתמקדת במקרי קצה בהם נשלחות חבילות עם נתונים ריקים או חבילה גדולה במיוחד. היא בודקת את התנהגות המערכת במקרים אלו לוודא שהמערכת יכולה לטפל גם בנתונים לא סטנדרטיים.

test_high_volume_package_handling

בדיקה זו נועדה לבדוק את עמידות המערכת בפני נפח גבוה של תקשורת במהלך זמן קצר. היא דומה לבדיקת עומס, שבה המערכת צריכה להתמודד עם כמות גדולה של חבילות מהר מאוד.

שימוש במודולים

- `unittest`: מודול ליצירת מבחני יחידה.
- `MagicMock` ו-`patch`: מהמודול `unittest.mock`: למיתוג ובדיקת קוד שמבוסס על פונקציות או מחלקות שמבצעות גישה למשאבים חיצוניים.
- `socket`: מודול לניהול תקשורת רשת.
- `QuicFunc` ו-`QuicPackage`: מודולים מותאמים אישית שמכילים את הפונקציונליות הספציפית להעברת וקליטת חבילות באמצעות QUIC, את המודלים האלה אנחנו בנינו.

Server.py

clarification:

program flow- to know and understand the flow of the functions through the output stream, at run time.

STAT

stats for the file transmission, string parsing and printing to the output stream

```
def stat(start_time,end_time,filename):
    global total_time,total_send,total_speed
    elapsed_time = (end_time - start_time) * 1000 # Convert to milliseconds
    file_size = os.path.getsize(filename)
    speed = file_size / (elapsed_time / 1000) # Convert elapsed time back to seconds for speed calculation
    total_time += elapsed_time
    total_send += 1
    total_speed += speed

    print("-----STATISTICS-----")
    print("-----FOR THE CURRENT FILE-----")
    print(f"Time taken to receive file: {elapsed_time:.2f} ms")
    print(f"Speed of file transfer: {speed:.2f} bytes/second")
    print("-----AVERAGE-----")
    print(f"average time :{total_time / total_send} ms")
    print(f"average speed :{total_speed / total_send} bytes/second")
```

The globals that are declared at the start are set as default as zero, used in stats to remember the past runs.

```
total_time=0 #to see the avg after sending couple of files
total_send=0
total_speed=0
```


Handle_client

```
"""
~~~~
handles two options for the client,
first option is to receiver a text message, usually a small one(about couple of kbs)
Second option is to send a text file, bigger than the text, up to 10 mb's as required in the assigment;
we parse the input to get the name of the file, and then we get the rest of the file in recv_file

"""

1 usage
def handle_client(client_address, name): # Takes client data and address as argument.
    """Handles a single client connection."""
    try:
        while True:
            # print("while") #for debugging
            data, _ = server_socket.recvfrom(BUFSIZ) #get the starting packet from the client
            new_package = QuicFunc.recreate_package(data) #parse the string to packet so it will be easier to work with.
            msg = new_package.payload
```

“ Text “ option

```
if msg[:4] == "TEXT": #if the pefix is TEXT then this is simple text
    print("enterd text")
    broadcast_text(new_package, name + ": ")
    print("sending done")
```

“ File “ option

```
elif msg[:4] == "FILE": #if the prefix is File then we are receving multiple packets.
    print("enterd file: " + msg[4:])
    receivedname = msg[4:]
    filename = str(Path.cwd()) + r'\\(2)' + receivedname
    with open(filename, "w") as file:
        recv_file(file, filename) #recv all the packets and return acks.
    print("done file") #just to know what part we are in the code, if we done a file also helps with debug
```

Recv_file

```
"""
    Called by handle_client function
    this func uses QuicFunc.recv_package_list_from_file
    and then redirect the list to write to file, which recreated the file that had been sent from the user (client)
    """

1 usage
def recv_file(file_towrite, filename): # gets file_towrite and the name of the file
    start_time = time.time() # the start time
    package_list = QuicFunc.recv_package_list_from_file(server_socket) # gets all the packets from the user and \
    # return array of all the packets

    write_to_file(package_list, file_towrite) # creates a replica of what we got

    end_time = time.time() # end time
    stat(start_time, end_time, filename) # for vars used for stats
    print("exit file") # used for debug and to understand program flow
```

broadcast text

shares msgs, with the gui to all clients.

```
"""
    broadcast the text msgs to the client (using our gui)
    """

2 usages
def broadcast_text(text_package: QuicPackage, prefix=""): # prefix is for name identificatio
    """Broadcasts a message to all the clients."""
    print(text_package.payload)
    text_package.payload = text_package.payload[:4] + prefix + text_package.payload[4:]
    for addr in client_addresses:
        print(client_address)
        QuicFunc.send_package(text_package, server_socket, addr)
    print("done broadcast")
```

broadcast_file

to all the users, throw a middle person (the server)

```
def broadcast_file(msg, prefix=""):
    path, s = os.path.split(msg)
    s = s[3:]
    for addr in client_addresses:
        server_socket.sendto(s, addr)
        file1 = open(msg, 'rb')
        slice = file1.read(128)
        while slice:
            server_socket.sendto(slice, addr)
            slice = file1.read(128)
```

new connection

```
"""
~~~~~
handles a new connection a broadcasts his name,
it's like a new user to the system the client assigns his name
and after this, he can send texts and everyone will know it's him
"""
usage
def new_connection(data, client_address):
    print("thread")
    client_addresses.add(client_address)
    new_package = QuicFunc.recreate_package(data)
    name = new_package.payload[4:]
    welcome = f'&TEXTWelcome %s! If you ever want to quit, you can just stop typing. The Connection ID is {CONNECTION_ID},' % name
    welcome_package = QuicPackage( pos: 0, welcome, CONNECTION_ID)
    QuicFunc.send_package(welcome_package, server_socket, client_address)
    msg = "TEXT%s has joined the chat!" % name
    new_package = QuicPackage( pos: 0, msg, CONNECTION_ID) # start the connection
    print(welcome)
    broadcast_text(new_package)
    client_thread = Thread(target=handle_client, args=(client_address, name))
    client_thread.start()
```

handshake

#being used by the wait_for_connection function

```
"""
~~~~~
checks for conenction, and sends back the connection id in which they will use.
"""

1 usage
def handshake(new_pack, address):
    if new_pack.payload == "hello":
        CONNECTION_ID = new_pack.connection_id
        print("The connection ID is: " + CONNECTION_ID)
        QuicFunc.send_package(new_pack, server_socket, address)
        return True
    return False
```

wait for connection

```
"""
~~~~~
waiting for the first connection and assure that we got the first packet
using handshake function to verify we got a connection.
"""

1 usage
def wait_for_connection():
    global CONNECTION_ID
    start = time.time()
    timeout = 1 # Timeout for retransmission
    while True:
        try:
            server_socket.settimeout(timeout)
            pack, address = server_socket.recvfrom(BUFSIZ)
            new_pack = QuicFunc.recreate_package(pack)

            if handshake(new_pack, address): #if we found the user we can stop waiting for him
                break

        except socket.timeout:
            if time.time() - start > timeout:
                print("Didn't receive pack in time, waiting again...")
                start = time.time() # Reset the start time for the next timeout

        except Exception as e:
            print(f"An error occurred: {e}")
```

definitions

for the connection to the server:

```
HOST = '127.0.0.1' #ip
PORT = 33002 #port
BUFSIZ = 1024 #buffsize for the socket
ADDR = (HOST, PORT)
CONNECTION_ID = 0 #the connection id for the server is always the same
```

the socket definition

```
#create a socket using the UDP protocol.
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, __value: 1)
server_socket.bind(ADDR)
```

main

```
if __name__ == "__main__":
    print("Waiting for connection...")
    flag_file = False
    # while True: #don't know if it's good or not
    time.sleep(5)
    wait_for_connection()
    server_socket.settimeout(None)
    data, client_address = server_socket.recvfrom(BUFSIZ)
    if client_address not in client_addresses:
        print("new connection, list: " + str(client_addresses))
        new_connection(data, client_address)
    else:
        if not flag_file:
            print("why?")
            print(threading.current_thread())
        else:
            print("main")
            print(file1)
```

Client.py

definitions

```
#global vairables being used for the resend algorithm
~~~~~
packages_list = {}
no_acks = []
ack_list = []
seq_threshold = 3
time_threshold = 0.002
```

valid mode#hellper for resending algo

```
"""
determines if we in need to resend, considering the mode.
"""

! usage
def valid_mode(mode):
    global no_acks
    global time_threshold

    match mode:
        case 1: # time
            return time.time() - no_acks[0].sent_time > time_threshold

        case 2: #seq
            return list(packages_list.values())[-1].seq - no_acks[0].seq > seq_threshold

        case 3: #both
            return list(packages_list.values())[-1].seq - no_acks[0].seq > seq_threshold and time.time() - no_acks[0].sent_time > time_threshold
```

send by seq algo #helper

```
"""
~~~~~
Algorithm to send by seq number.
"""

! usage
def send_by_seq(ack_seq):
```

send by time

send by time, uses the threshold and send_sleep to give a rough estimate when a packet should have come already if it didn't we need to resend it.

```
no_acks.append(packages_list[i])

'''
~~~~~
Algorithm to resend by time.
'''

!usage
def send_by_time(ack_seq):
    global ack_list, no_acks, packages_list, time_threshold, send_sleep

    if packages_list[ack_seq].sent_time > packages_list[ack_list[-2]].sent_time:
        index_start = int((send_sleep * 3) / time_threshold) #how many packets from the last should we check
        if (index_start > len(ack_list)): # if we are just starting we will reach index out of bounds
            pass
        else:
            for i in range(ack_list[-index_start] + 1, ack_seq):
                if i in ack_list:
                    continue
                no_acks.append(packages_list[i])
```

send by both, applying both algorithms to be more accurate.

```
'''
~~~~~
uses both algorithm to know better when we should resend a package without acknowledgement
'''

!usage
def send_by_both(ack_seq):
    global ack_list, no_acks, packages_list, time_threshold, send_sleep
    if ack_seq > ack_list[-2] + 1 and packages_list[ack_seq].sent_time > packages_list[ack_list[-2]].sent_time:
        if ack_seq > ack_list[-2] + 1: #the send_by
            for i in range(ack_list[-2] + 1, ack_seq):
                no_acks.append(packages_list[i])
```

Append noack

inserting a no acked packets that are potentially lost, if we later on got acknowledge tha they have reached the destination we will remove them. but for now we see them as lost. (uses send_by_time, send_by_seq, send_by_both functions)

```

"""
3 modes to choose the mode in which we are going to use
The mode is given by the user
the mode we have will have a different algorithm to determine a packet as lost or not.
"""

1 usage
def append_noack(mode, ack_seq):
    global no_acks
    global time_threshold
    global send_sleep

    match mode:
        case 1: # time
            send_by_time(ack_seq)
        case 2: #seq
            send_by_seq(ack_seq)
        case 3: #both
            send_by_both(ack_seq)

```

resending_algo

if the no_ack list is not empty then we want to empty it using the right mode
we are using func valid mode to determine it.

removes the packet it just rescinded from the current no_ack list

```

"""
resending using the mode we got from the user
also uses resend_lost_packages (which is in QuicFunc.py)
"""

1 usage
def resending_algo(package_list, no_acks, client_sockt, ADDR, mode):
    if len(package_list) > 0 and len(no_acks) > 0:
        if valid_mode(mode):
            print("resend")
            resend_lost_packages(package_list, no_acks, client_socket, SERVER_ADDR)
            no_acks.remove(no_acks[0])

```


acked

```
"""
    check if the packet is an ACK
"""

1 usage
def acked(data):
    return data[:3] == "ACK".encode('utf-8')
```

almost_exec

```
1 usage
def almost_exec(ack_seq): #almost someone who got acked sent again
    for index, packet in enumerate(no_acks): # we check if it's in the acks and remove it
        if not compare(packages_list[ack_seq], packet):
            no_acks.remove(packet)
```

receive function

using helper functions-

acked, almost_exec, resending_algo, append_noAck, valid_mode

```
"""
    Functions that manages the sent packets and resends if needed
    using functions:
        acked - checks if the packet that we got is a ACK packet
        almost_exec - checks if the packet is in the no_ack list if so removes it
        resending_algo - depending on the mode it chooses the algo to use
        append_noAck - depending on the mode, main reason to colerate with the resending_algo
        valid_mode - depending on the mode does it's check
"""

1 usage
def receive():
    """Handles receiving of messages."""

    global Running #needed for hand of connection currently doesn't work #tchelet
    global packages_list #dictionary contains all the packets we sent so far
    global no_acks # is list for the packets we need to resend sometimes we removre elements from it if we got a ack for them :)
    global flag_send #don't know #tchelet
    global ack_list #list of all seq's we got so far
```

mid part

we are checking if the ack that we got are in the no_ack list if so it removes it from the list, so we won't resend the acked packet.

then we check if we need to append into no ack

```

while Running: #when we need to keep running the thread we switch this off

    try:

        resending_algo(packages_list, no_acks, client_socket, SERVER_ADDR, mode) #checks there's packets to resend and resends
        data, n = client_socket.recvfrom(BUFSIZ)

        if acked(data): #if we got ack we check if we lost in the way someone else
            print(data.decode("utf-8"))
            ack_seq = int(data[5:])

            try:
                packages_list[ack_seq].recvack()
                ack_list.append(ack_seq)
                almost_exec(ack_seq)
                append_noack(mode, ack_seq)

            except(KeyError):
                print("KeyError: " + str(data[5:]))
                ack_list.append(int(data[5:]))

```

to show the msg on the gui.

```

    else:
        recv_package = recreate_package(data)
        msg = recv_package.payload[4:]
        msg_list.insert(END, *elements: msg) # Insert the received message into the message list
    except OSError as e: # Possibly server has left the chat.
        pass

```

send

```

"""
    send basic msg between the client to the server.
"""
3 usages
def send(event=None):
    """Handles sending of messages."""
    msg = "TEXT" + my_msg.get()
    my_msg.set("") # Clears input field.
    new_package = QuicPackage( pos: 0, str(msg), CONNECTION_ID)
    QuicFunc.send_package(new_package, client_socket, SERVER_ADDR)

    #client_socket.sendto(bytes("TEXT" + msg, "utf-8"), SERVER_ADDR)
    print(SERVER_ADDR)
    if msg == "{quit}":
        client_socket.close()
        root.quit()

```

send_attach #for files

we get the path to the file from the filedialog, by the user

```

"""
~~~~~
send attachment, it lets you select in the gui the file in the father directory as a default
after it, it usess the func send_packages_from_file
which create small packages and sends to the server
"""

1 usage

def sendattach():
    """Handles sending of files (attachments)."""
    global ack_list
    global no_acks

    filename = filedialog.askopenfilename(initialdir="../", title="Select A File", # ../ the father of this directory
                                         filetype=(("jpeg files", "*.txt"), #the file type we want to transmit
                                                    ("all files", "*.*")))

    if (filename == ""): #if we got nothing then we can't send natting
        client_socket.close()
        root.quit()

    return

```

we parse the input and then starting, to send the file using helper functions.

```

#parsing the input
s = filename.split("/")

with open(filename, "rb") as file1: #opening the file and transmitting to the server
    entry = "FILE" + str(s[-1])
    entry_package = QuicPackage( pos: 0, entry, CONNECTION_ID)
    ack_list.append(entry_package.seq)
    QuicFunc.send_package(entry_package, client_socket, SERVER_ADDR)
    QuicFunc.send_packages_from_file(file1, client_socket, SERVER_ADDR, packages_list, no_acks, CONNECTION_ID)

```

gui related functions:

```

def on_entry_click(event):
    """function that gets called whenever entry1 is clicked"""
    global firstclick
    if firstclick: # if this is the first time they clicked it
        firstclick = False
        entry_field.delete( first: 0, last: "end") # delete all the text in the entry

```

when we close the connetion and the gui

```

2 usages
def on_closing(event=None):
    """This function is to be called when the window is closed."""
    my_msg.set("{quit}")
    send()

```

```

def sendquit():
    global Running
    on_closing()
    root.destroy()
    Running = False

```

```

2 usages
def send_first_ack():
    print("Send first")
    pack = QuicPackage( pos: 0,  payload: "hello", CONNECTION_ID)
    QuicFunc.send_package(pack, client_socket, SERVER_ADDR)
    wait_for_ack()

```

```

1 usage
def wait_for_ack():
    print("Enter ack")
    start = time.time()
    timeout = 1 # Timeout for retransmission
    while True:
        try:
            client_socket.settimeout(timeout)
            data, n = client_socket.recvfrom(BUFSIZ)
            new_pack = QuicFunc.recreate_package(data)

            if new_pack.payload == "hello":
                print("Connection is good")
                return

        except socket.timeout:
            if time.time() - start > timeout:
                print("Didn't receive ack in time, resending...")
                send_first_ack()
                return

        except Exception as e:
            print(f"An error occurred: {e}")

```

The gui is created in this code, we created here the chat window and the buttons. there is also the creation of the socket and the input for connection ID and mode of packet loss recovery algorithm(time, sequence number or both).

```

269 root = Tk()
270 root.title("ChatI0")
271 root.geometry('500x500')
272
273 messages_frame = Frame(root)
274 my_msg = StringVar() # For the messages to be sent.
275 my_msg.set("Type your messages here.")
276 yscrollbar = Scrollbar(messages_frame) # To navigate through past messages.
277 xscrollbar = Scrollbar(messages_frame, orient=HORIZONTAL) # To navigate through long messages.
278 # Following will contain the messages.
279 msg_list = Listbox(messages_frame, height=20, width=50, xscrollcommand=xscrollbar.set, yscrollcommand=yscrollbar.set)
280 yscrollbar.pack(side=RIGHT, fill=Y)
281 xscrollbar.pack(side=BOTTOM, fill=X)
282 msg_list.pack(side=LEFT, fill=BOTH)
283 messages_frame.pack()
284 yscrollbar.config(command=msg_list.yview)
285 xscrollbar.config(command=msg_list.xview)
286
287 entry_field = Entry(root, width=42, textvariable=my_msg)
288 entry_field.bind('<FocusIn>', on_entry_click)
289 entry_field.bind("<Return>", send)
290 entry_field.place(x=15, y=344)
291 send_button = Button(root, text="Send", command=send)
292 send_button.place(x=280, y=340)
293 send_attach = Button(root, text="Send Attachment", command=sendattach)
294 send_attach.place(x=50, y=375)
295 send_button = Button(root, text="Quit Chat", command=sendquit)
296 send_button.place(x=175, y=375)
297 root.protocol(name="WM_DELETE_WINDOW", on_closing)
298 |
299 #----Socket code----
300 HOST = "127.0.0.1"
301 PORT = 33002
302 BUFSIZ = 1024
303 SERVER_ADDR = (HOST, PORT)
304 CONNECTION_ID=int(input("Enter the connection id"))
305 mode = int(input("Enter the mode of packet loss between 1-3\n"))
306 while mode not in [1,2,3]:
307     mode=int(input("bro be trolling 1-3"))
308
309
310 client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
311 send_first_ack()
312
313 receive_thread = Thread(target=receive)
314 receive_thread.start()
315 root.mainloop()
316

```

During the project, we thought it would be cool to add a GUI that opens once the connection is established. Through this GUI, we can send necessary information, such as an opening message to set the user's name. The interface would also include buttons for sending messages and attachments. Every time an action is performed in the GUI, it triggers a corresponding function in the communication protocol.

QuicPackage.py*

recreate && send packets

```
11 # function for recreating packets, we get encoded packet_and we decode it, each variable acording to its kind
12 11 usages
13 def recreate_package(str_package: bytes):
14     list_package = str_package.decode("utf-8").split(sep: "&", maxsplit: 4) # the packet encoded using & between each field
15     new_package = QuicPackage(int(list_package[1]), list_package[3], list_package[4]) # create the new packet
16     new_package.recreate_from_str(float(list_package[2]),
17                                   list_package[0]) # setting the time anf sequence number fot the packet
18     return new_package # return the new packet
19
20 # function for sending the packet using the "sendto" function from the socket library
21 14 usages
22 def send_package(package: QuicPackage, sock: socket, ADDR): # event is passed by binders.
23     """Handles sending of messages."""
24     sock.sendto(package.encode_package(), ADDR)
25
```

send packaged from file

function to get information from file, store it in packets and send the packets

```
26 # function to read information from the file, store it in packet objects and sending the packets
27 1 usage
28 def send_packages_from_file(file1, sock: socket, ADDR, package_list, no_ack, connect):
29     """Handles sending of files (attachments)."""
30     global send_sleep
31     pos = 0
32     print("Sending file...")
33     while True: # while we can still read from the file
34         slice = file1.read(268) # read 268 bytes from the file
35         # print("hello\n",slice)
36         if not slice: # if we are done reading from the file
37             time.sleep(0.5) # wait for 0.5 seconds
38             # new_package = QuicPackage(pos, "DONE",connect)
39             # send_package(new_package, sock, ADDR)
40             send_last_ack(connect, sock, ADDR) # send the last packet and wait for ack
41             return package_list # return all the packets we sent from the file
42
43     time.sleep(send_sleep) # wait
44     new_package = QuicPackage(pos, slice, connect) # create the new packet
45     package_list[new_package.seq] = new_package # add the new packet to the list
46     send_package(new_package, sock, ADDR) # send the packet
47     pos += 1 # increase the position of the packets by one
48
```

send last ack

intended to send the last packet and wait for acknowledgment
to verify that we ended the file sending stream

```

50 # function to send the last packet and wait for response from the server that we it was received successfully
    2 usages
51 def send_last_ack(con, client_socket, SERVER_ADDR):
52     print("Send last")
53     pack = QuicPackage( pos: 0, payload: "DONE", con) # create the last packet
54     send_package(pack, client_socket, SERVER_ADDR) # send the last packet
55     wait_for_ack(con, client_socket, SERVER_ADDR) # wait for the server to receive it, send again if it didn't
56
57
58 # function to wait for the server to receive the last packet, and send again if it didn't
    2 usages
59 def wait_for_ack(con, client_socket, SERVER_ADDR):
60     print("Enter ack")
61     start = time.time()
62     timeout = 1 # Timeout for retransmission
63     while True: # while the server don't get the packet
64         try:
65             client_socket.settimeout(timeout) # wait
66             data, n = client_socket.recvfrom(1024) # receive the acknowledged packet from the server
67             new_pack = recreate_package(data) # decoded the packet
68
69             if new_pack.payload == "DONE": # if the packet we received is the acknowledgment packet from the server
70                 print("sent last packet")
71                 return # quit the function
72
73         except socket.timeout:
74             if time.time() - start > timeout: # if we didn't receive the acknowledgment packet from the server in time
75                 print("Didn't receive last ack in time, resending...")
76                 send_last_ack(con, client_socket, SERVER_ADDR) # send again the last packet
77                 return

```

wait for last ack

server function wo wait to receive the last packet and to send acknowledgement about receiving it

```

83 # function for the server to receive the last ack and send an acknowledgment packet back to the client
    1 usage
84 def wait_for_last_ack(server_socket, new_package):
85     start = time.time()
86     timeout = 1 # Timeout for retransmission
87     while True: # while we didn't get the last packet
88         try:
89             server_socket.settimeout(timeout) # wait
90             pack, address = server_socket.recvfrom(1024) # receive the last packet
91             new_pack = recreate_package(pack) # decode the packet
92
93             if new_pack.payload == "DONE": # if the packet received is the last one, send acknowledgment packet
94                 # print("The connection ID is: " + CONNECTION_ID)
95                 send_package(new_pack, server_socket, address) # send the acknowledgment packet
96                 return
97
98         except socket.timeout:
99             if time.time() - start > timeout: # if we didn't receive the last packet from the server in time, wait
100                 print("Didn't receive last pack in time, waiting again...")
101                 start = time.time() # reset the start time for the next timeout
102
103         except Exception as e:
104             print(f"An error occurred: {e}")
105
106

```

recv packaged list from file && compare (simple compare)

functions to receive packets from the file sending by the client and to compare packets by their position

```

107 # function to receive the packets from the client
108 1usage
109 def recv_package_list_from_file(sock: socket):
110     package_list = [] # initialize a list of packets
111     while True: # while we get packets
112         package, addr = sock.recvfrom(1024) # receive the packets
113         if not package: # if the packet we received is None, there are no longer packets to receive
114             print("done recv")
115             return package_list # return the list of packets
116         new_package = recreate_package(package) # decode the received packet
117         if new_package.payload == "DONE": # if its the last one
118             # print("done recv")
119             wait_for_last_ack(sock, new_package) # wait to receive it and send ack
120             return package_list
121         # time.sleep(0.00002)
122         print("ack for ", new_package.getpos()) # send ack for receiving packets
123         new_package.send_ack(sock, addr) # send ack
124         package_list.append(new_package) # add the new packet to the list
125
126 # function to compare packets by position
127 2usages
128 def compare(x, y):
129     return x.getpos() - y.getpos()...# return True if the x packet is bigger that the y packet by its position

```

remove duplicates && write to file
functions to remove duplicated packets
so we won't write duplicated to the file

```

131 # function to remove duplicates packets, we know its duplicate by position
132 1usage
133 def remove_duplicates(list):
134     seen_x = set()
135     unique_packets = []
136
137     for pac in list:
138         if pac.getpos() not in seen_x:
139             unique_packets.append(pac)
140             seen_x.add(pac.getpos())
141
142     return unique_packets
143
144 # function to write the packets received from the file sent by the client into a new file
145 1usage
146 def write_to_file(package_list: [], file1):
147     unique_sorted_package_list = remove_duplicates(package_list)...# remove duplicated packets
148     package_list = sorted(unique_sorted_package_list, key=cmp_to_key(compare))...# sort the packets by position
149
150     f = open("packetPos.txt", "w")...# open the packetPos file to write their the packets position
151     for payload in package_list:
152         f.write(str(payload.getpos()) + "\n") #print packet pos
153         # f.write("\n")
154         file1.write(payload.payload) #print the packet itself
155     f.close()

```

resend lost packages
function to resend lost packets
called by the client

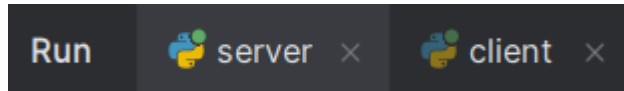

```
157 # function to resend lost packets
158 1 usage
159 def resend_lost_packages(package_list, no_ack, sock, ADDR):
160     print("resend lost package")
161     pack = no_ack[0] # send the first packet in the no_acks list, its the first packet that need to resend
162     print("I lost this :", pack.seq, "and this is the pos somehow", pack.getpos())
163     pack.update_for_resend() # update the packet for resend
164     send_package(pack, sock, ADDR) # send the packet again
165     package_list[pack.seq] = pack # add the packet to the list of sending packets
```

User Manual for GUI

As you can see, in our project we also created a GUI for texting and sending files from client to the server. The client side can send any text file in any size (we chose the second option for the implementation of the QUIC protocol, so the client can send also big files that can get to 10MB and even more).

In this page we will show you how to use the GUI properly and how to send texts and files from the client to the server:

First step: run the server and the client



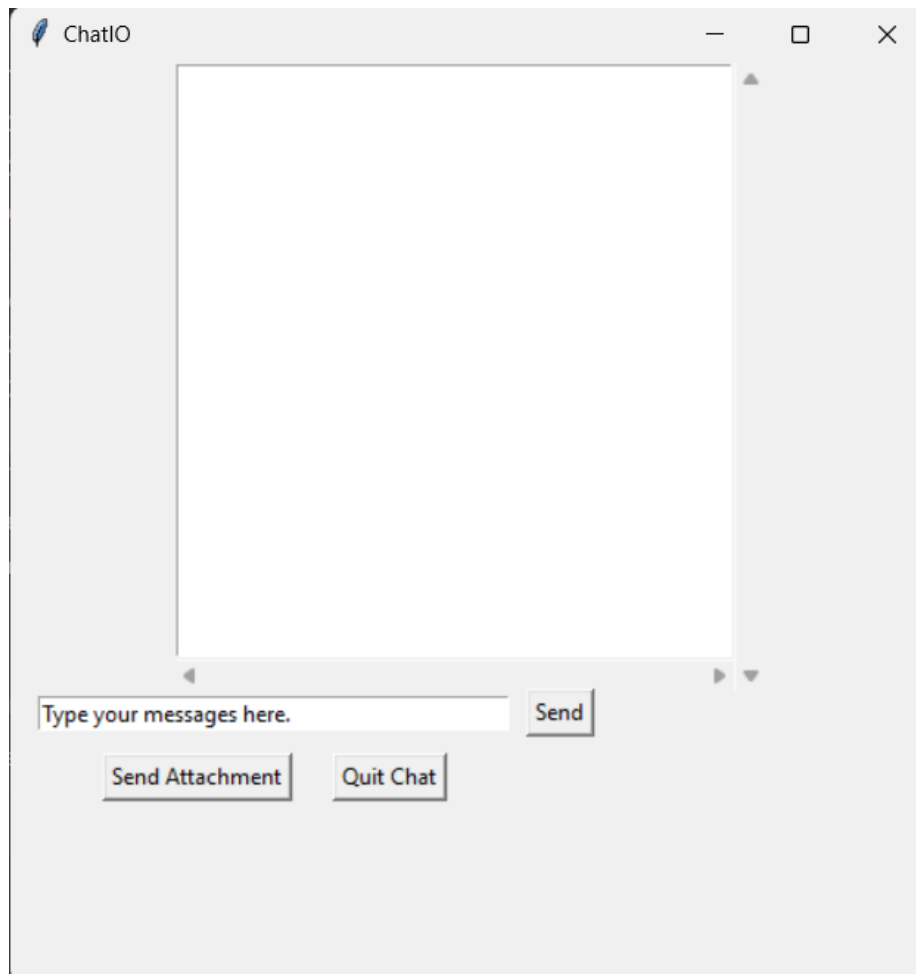
Second step: in the client side, type the connection ID for the connection between the server and the client. The connection ID is a unique identifier used to distinguish different connections between a client and a server. If the connection fails at some point, with the connection ID the client can connect again and continue the sending to the server from the exact same point where the connection failed without the worry that the files or messages sent earlier will be deleted or thrown away.

```
Enter the connection id
```

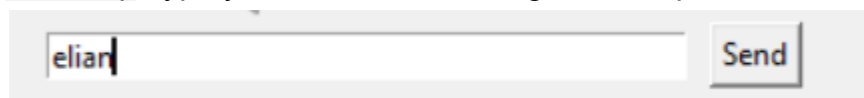
Third step: in the client side, type the scenario of packet loss handling you want the code to use: 1 for time, 2 for sequence number, 3 for both (time and sequence number)

```
Enter the mode of packet loss between 1-3
```

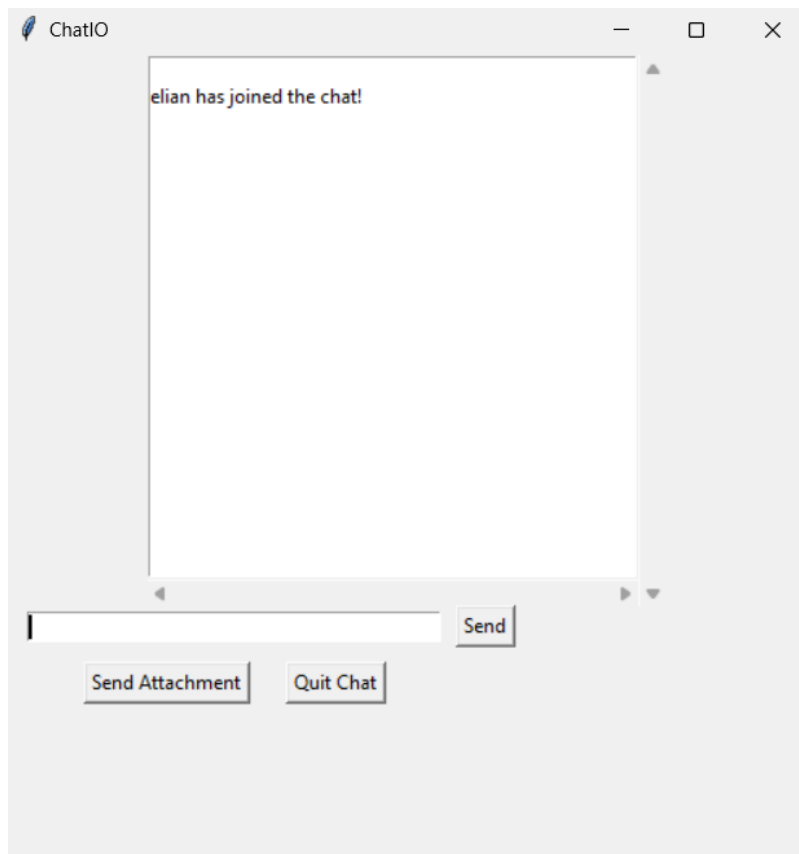
Fourth step: the GUI will open straight to the screen, this is how the GUI looks:



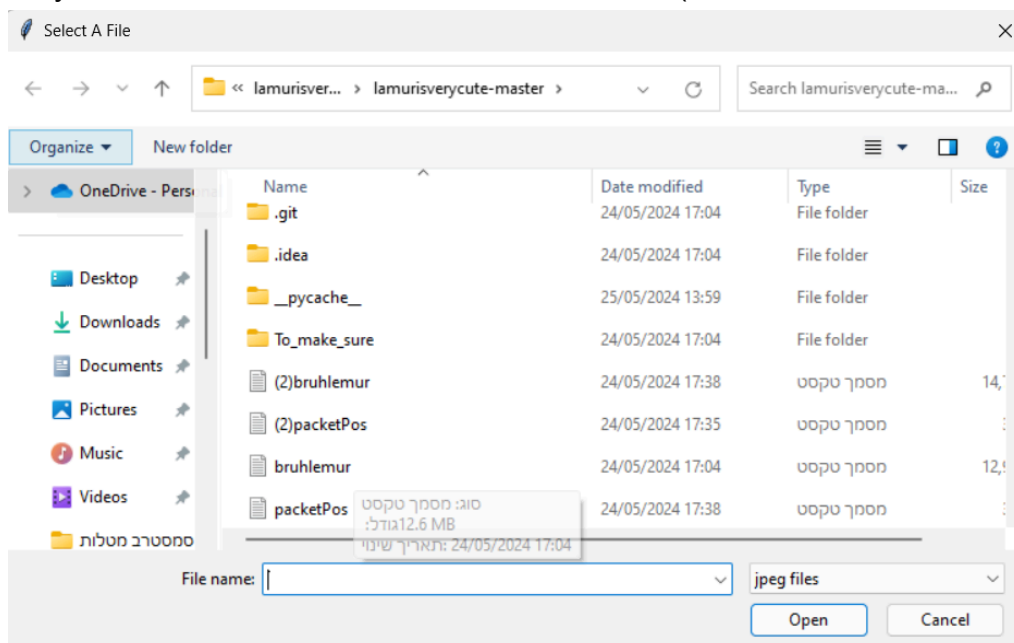
Fifth step: type your name in the filling box and press send:



the result will be- your name will be printed to the GUI screen as follows:



Sixth step: send the file you would like to send to the server by pressing the “Send Attachment” button, it will open the “File Explorer” and all you need to monitor to the file you would like to send and double click on it(or click once and select “open”) :



Seventh step: the file will be sending and you can monitor the progress of the sending in the terminal of the server and the client to see the packets getting sent to

The image displays two terminal windows side-by-side, showing network communication logs. The top window, titled 'Client', shows a series of 'ACK' messages from a client to a server, indicating successful receipt of data. The bottom window, titled 'Server', shows a series of 'ACK' messages from a server to a client, indicating successful receipt of data.

Client Window (Top):

```

Run  server  Client
ACK: 8895 sent!
ack for 8895
ACK: 8896 sent!
ack for 8896
ACK: 8897 sent!
ack for 8897
ACK: 8898 sent!
ack for 8898
ACK: 8899 sent!
ack for 8899
ACK: 8900 sent!
ack for 8894
ACK: 8901 sent!
ack for 8895
ACK: 8902 sent!
ack for 8896
ACK: 8903 sent!
ack for 8897
ACK: 8904 sent!
ack for 8898
ACK: 8905 sent!
ack for 8899
ACK: 8906 sent!
ack for 8900
ACK: 8907 sent!
ack for 8901
ACK: 8908 sent!

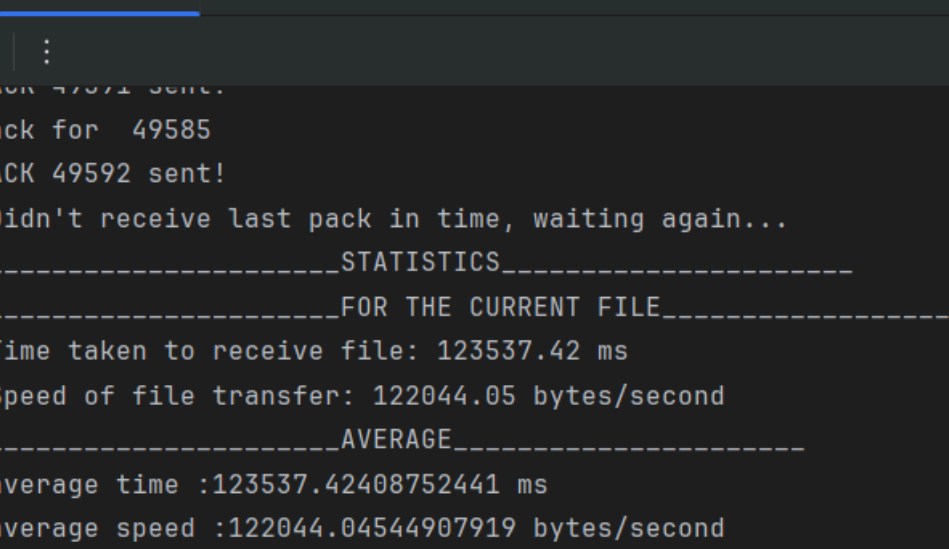
```

Server Window (Bottom):

```

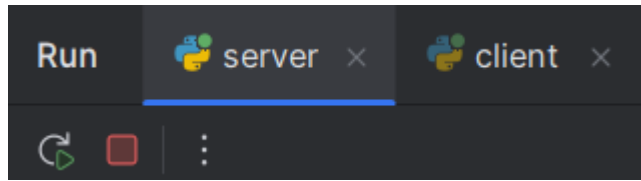
Run  server  Client
ACK: 4688
ACK: 4689
ACK: 4690
ACK: 4691
ACK: 4692
ACK: 4693
ACK: 4694
ACK: 4695
ACK: 4696
ACK: 4697
ACK: 4698
ACK: 4699
ACK: 4700
ACK: 4701
ACK: 4702
ACK: 4703
ACK: 4704
ACK: 4705
ACK: 4706
ACK: 4707
ACK: 4708
ACK: 4709
ACK: 4710
ACK: 4711
ACK: 4712
ACK: 4713
ACK: 4714
ACK: 4715

```



```
Run server client x
ack for 49585
ACK 49592 sent!
Didn't receive last pack in time, waiting...
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 123537.42 ms
Speed of file transfer: 122044.05 bytes/second
-----AVERAGE-----
average time :123537.42408752441 ms
average speed :122044.04544907919 bytes/second
exit file
done file
```

Ninth step: in this final step you can choose to send again the file(or different file) and return to the sixth step of you can close the connection by click on the “Quit Chat” or to terminate the python files by clicking on the red box:



Results for resend by seq:

with 0%

```
ACK 37654 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 84636.26 ms
Speed of file transfer: 119123.42 bytes/second
```

with 1%

```
ACK 38411 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 84977.13 ms
Speed of file transfer: 118645.57 bytes/second
```

with 2%

```
ACK 39204 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 86634.62 ms
Speed of file transfer: 116375.65 bytes/second
```

with 3%

```
ACK 40042 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 84976.83 ms
Speed of file transfer: 118645.99 bytes/second
AVERAGE
```

with 4%

```
ACK 40875 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 84604.98 ms
Speed of file transfer: 119167.46 bytes/second
```

with 5%

```
ACK 41769 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 84698.00 ms
Speed of file transfer: 119036.57 bytes/second
AVERAGE
```

with 6%

```
ACK 42609 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 83732.31 ms
Speed of file transfer: 120409.43 bytes/second
AVERAGE
```

with 7%

```
ACK 43555 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 83631.97 ms
Speed of file transfer: 120553.90 bytes/second
AVERAGE
```

with 8%

```
ACK 44458 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 84507.03 ms
Speed of file transfer: 119305.57 bytes/second
AVERAGE
```

with 9%

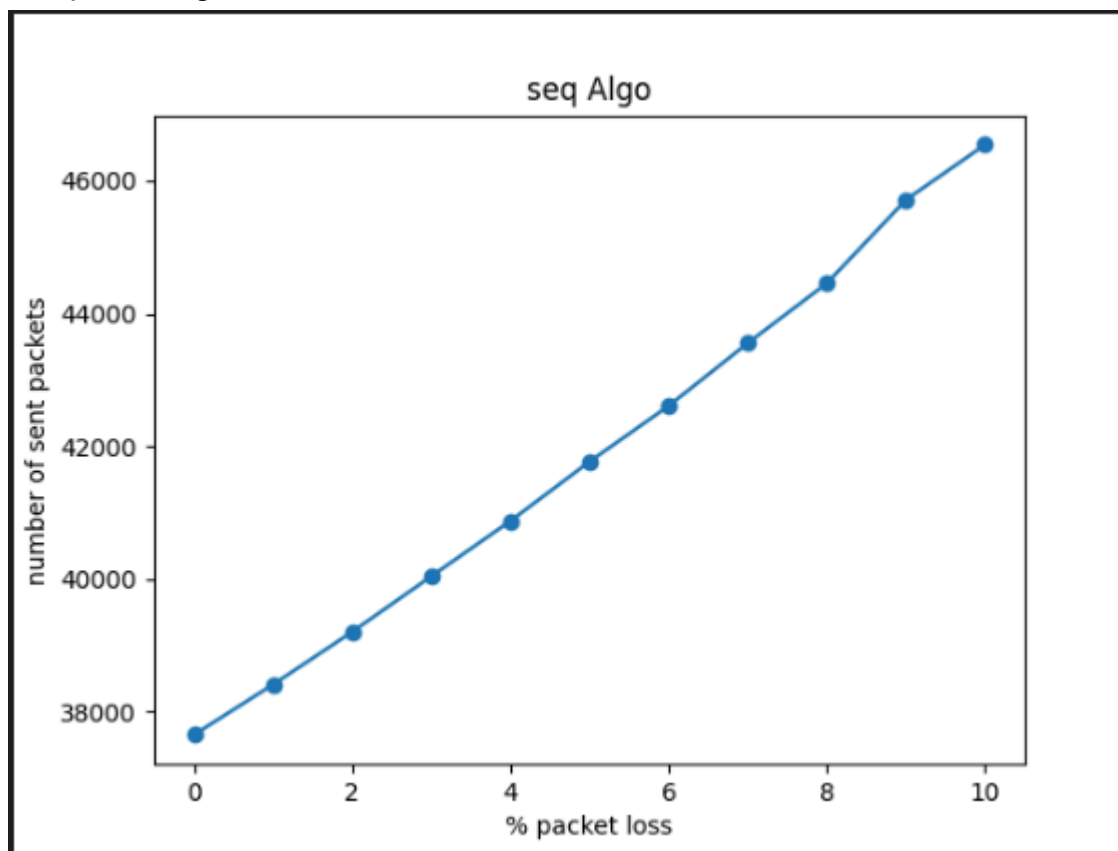
```
ACK 45715 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 85164.55 ms
Speed of file transfer: 118384.47 bytes/second
AVERAGE
```


with 10%

```
ACK 46545 sent!  
done recv  
  
-----STATISTICS-----  
-----FOR THE CURRENT FILE-----  
Time taken to receive file: 83893.39 ms  
Speed of file transfer: 120178.25 bytes/second
```

Graph that represent the chane of the among of packets send as a result of packet loss percentage



data size is 10.4 mb

result/ packet loss	0	1	2	3	4	5	6	7	8	9	10
Time(s)	84.5	85	86.6	85	84.6	84.5	83.7	83.6	84.5	85	83.9
last packet sequence	37654	38411	39204	40042	40875	41769	42609	43555	44458	45715	46545

results running by time

Statistics- for each packet loss scenario we ran the program and recorded it with Wireshark and calculate its statistics(time it was taken to send the file from the client to the server, speed for the sending in bytes per second and the number of packets we sent in total from the client to the server including the packet loss and the recovery):

with 0% packet loss:

```
ack for 37645
ACK 37652 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 107622.42 ms
Speed of file transfer: 93680.85 bytes/second
-----AVERAGE-----
average time :107622.41888046265 ms
average speed :93680.85297542291 bytes/second
exit file
done file
```

with 1% packet loss:

```
ACK 39059 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 106929.92 ms
Speed of file transfer: 94287.55 bytes/second
-----AVERAGE-----
average time :106929.92091178894 ms
average speed :94287.54752673206 bytes/second
exit file
done file
```

with 2% packet loss:

```
ACK 40594 sent!
done recv
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 108066.93 ms
Speed of file transfer: 93295.51 bytes/second
-----AVERAGE-----
average time :108066.93172454834 ms
average speed :93295.51453999271 bytes/second
exit file
done file
```

with 3% packet loss:

```
ACK 41977 sent!
done recv
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 105052.54 ms
Speed of file transfer: 95972.55 bytes/second
-----AVERAGE-----
average time :105052.53911018372 ms
average speed :95972.5494062108 bytes/second
exit file
done file
```

with 4% packet loss:

```
ACK 43506 sent!
done recv
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 106463.57 ms
Speed of file transfer: 94700.57 bytes/second
-----AVERAGE-----
average time :106463.56749534607 ms
average speed :94700.56505894123 bytes/second
exit file
done file
```

S:

with 5% packet loss:

```
ACK 45098 sent!
done recv
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 108242.21 ms
Speed of file transfer: 93144.44 bytes/second
-----AVERAGE-----
average time :108242.20585823059 ms
average speed :93144.44324245417 bytes/second
exit file
done file
```

with 6% packet loss:

```
ACK 46582 sent!
done recv
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 109752.11 ms
Speed of file transfer: 91863.02 bytes/second
-----AVERAGE-----
average time :109752.11215019226 ms
average speed :91863.01568577456 bytes/second
exit file
done file
```

with 7% packet loss:

```
ACK 48109 sent!
done recv
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 108149.82 ms
Speed of file transfer: 93224.01 bytes/second
-----AVERAGE-----
average time :108149.81722831726 ms
average speed :93224.01330291062 bytes/second
exit file
done file
```

with 8% packet loss:

```
ACK 49651 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 103317.55 ms
Speed of file transfer: 97584.19 bytes/second
-----AVERAGE-----
average time :103317.55352020264 ms
average speed :97584.19219661974 bytes/second
exit file
done file
```

with 9% packet loss:

```
ACK 50887 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 108429.17 ms
Speed of file transfer: 92983.84 bytes/second
-----AVERAGE-----
average time :108429.16655540466 ms
average speed :92983.83747004328 bytes/second
exit file
done file
```

with 10% packet loss:

```
ACK 52389 sent!
done recv

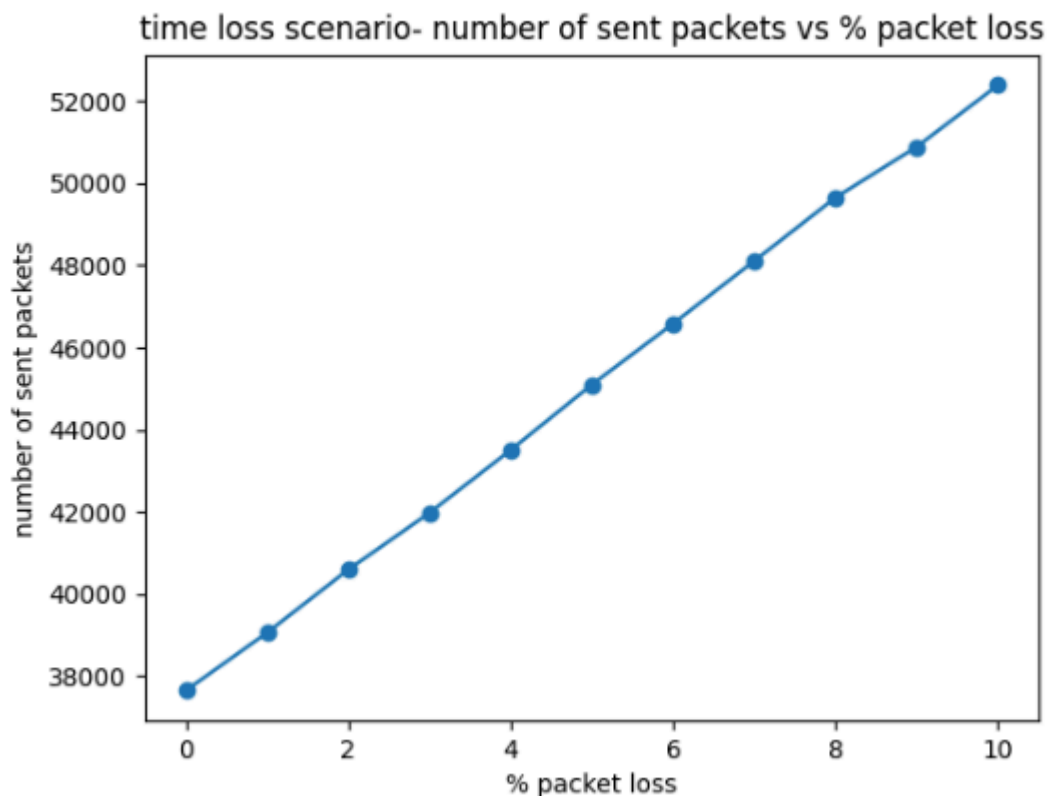
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 107747.25 ms
Speed of file transfer: 93572.32 bytes/second
-----AVERAGE-----
average time :107747.25389480591 ms
average speed :93572.31516863767 bytes/second
exit file
done file
```

Table of statistics:

result/%	6%	7%	8%	9%	10%
Time(ms)	109752.11	108149.82	103317.55	108429.17	107747.25
speed(b/s)	91863.02	93224.01	97584.19	92983.84	93572.32
last packet sequence	46582	48109	49651	50887	52386

result/%	0%	1%	2%	3%	4%	5%
Time(ms)	107622.42	106929.92	108066.93	105052.54	106463.57	108242.21
speed(b/s)	93680.85	94287.55	93295.51	95972.55	94700.57	93144.44
last packet sequence	37652	39059	40594	41977	43506	45098

The graph that show the relationship between the percentage of lost packets to the number of sent packets, as we can see we sent more packets when the percentage of lost packets is bigger and this is because when we lost a packet we need to send it again and when there is more % of packet loss there are more packets that get lost and we need to send again.



the code to create the Graph:

```
1 import matplotlib.pyplot as plt
2
3 # The Graph for the Time packet loss scenerio
4 x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] #packey loss percentage in %
5 # y = [108044,107919,109085,107416,105896,109106,110829,105578,109945,111132,111933]#time
6 # z=[93314,93423,92424,93860,95207,92406,90969,95494,91701,90722,90072]#speed
7 a=[37652,39059,40594,41977,43506,45098,46582,48109,49651,50887,52386]
8 # Create a plot
9 plt.plot(*args=x, a, marker='o')
10
11 # Add titles and labels
12 plt.title('time loss scenario- number of sent packets vs % packet loss')
13 plt.xlabel('% packet loss')
14 plt.ylabel('number of sent packets')
15
16 # Show the plot
17 plt.show()
18
19
```

Running by time + sequence number:

Statistic- for each packet loss scenario we ran the program and recorded it with Wirshark and calculate its statistics(time it was taken to send the file from the client to the server, and speed for the sending in bytes per econd):

with 0% packet loss:

```
ACK 37652 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 83425.06 ms
Speed of file transfer: 120852.89 bytes/second
-----AVERAGE-----
average time :83425.06051063538 ms
average speed :120852.89406190702 bytes/second
exit file
done file
```

with 1% packet loss:

```
ACK 38410 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 83302.99 ms
Speed of file transfer: 121029.99 bytes/second
-----AVERAGE-----
average time :83302.99258232117 ms
average speed :121029.98568792916 bytes/second
exit file
done file
```


with 2% packet loss:

```
ACK 39222 sent!
done recv
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 83094.77 ms
Speed of file transfer: 121333.27 bytes/second
-----AVERAGE-----
average time :83094.7654247284 ms
average speed :121333.27470709271 bytes/second
exit file
done file
```

with 3% packet loss:

```
ACK 40055 sent!
done recv
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 82911.76 ms
Speed of file transfer: 121601.08 bytes/second
-----AVERAGE-----
average time :82911.76223754883 ms
average speed :121601.0820167325 bytes/second
exit file
done file
```

with 4% packet loss:

```
ACK 40902 sent!
done recv
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 83111.26 ms
Speed of file transfer: 121309.20 bytes/second
-----AVERAGE-----
average time :83111.25564575195 ms
average speed :121309.20080155626 bytes/second
exit file
done file
```

with 5% packet loss:

```
ACK 41735 sent!  
done recv  
  
-----_STATISTICS_-----  
-----_FOR THE CURRENT FILE_-----  
Time taken to receive file: 83067.68 ms  
Speed of file transfer: 121372.83 bytes/second  
  
-----_AVERAGE_-----  
average time :83067.68250465393 ms  
average speed :121372.83352566311 bytes/second  
exit file  
done file  
|
```

with 6% packet loss:

```
ACK 42518 sent!  
done recv  
  
-----_STATISTICS_-----  
-----_FOR THE CURRENT FILE_-----  
Time taken to receive file: 82382.25 ms  
Speed of file transfer: 122382.67 bytes/second  
  
-----_AVERAGE_-----  
average time :82382.25269317627 ms  
average speed :122382.66945127014 bytes/second  
exit file  
done file
```

with 7% packet loss:

```
ACK 43498 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 83130.06 ms
Speed of file transfer: 121281.75 bytes/second
-----AVERAGE-----
average time :83130.06448745728 ms
average speed :121281.75362502219 bytes/second
exit file
done file
|
```

with 8% packet loss:

```
ACK 44480 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 83109.43 ms
Speed of file transfer: 121311.87 bytes/second
-----AVERAGE-----
average time :83109.42912101746 ms
average speed :121311.8668559153 bytes/second
exit file
done file
```

with 9% packet loss

```
ACK 45412 sent!
done recv

-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 83142.70 ms
Speed of file transfer: 121263.32 bytes/second
-----AVERAGE-----
average time :83142.70234107971 ms
average speed :121263.31856089477 bytes/second
exit file
done file
```

with 10% packet loss:

```
ACK 46348 sent!
done recv

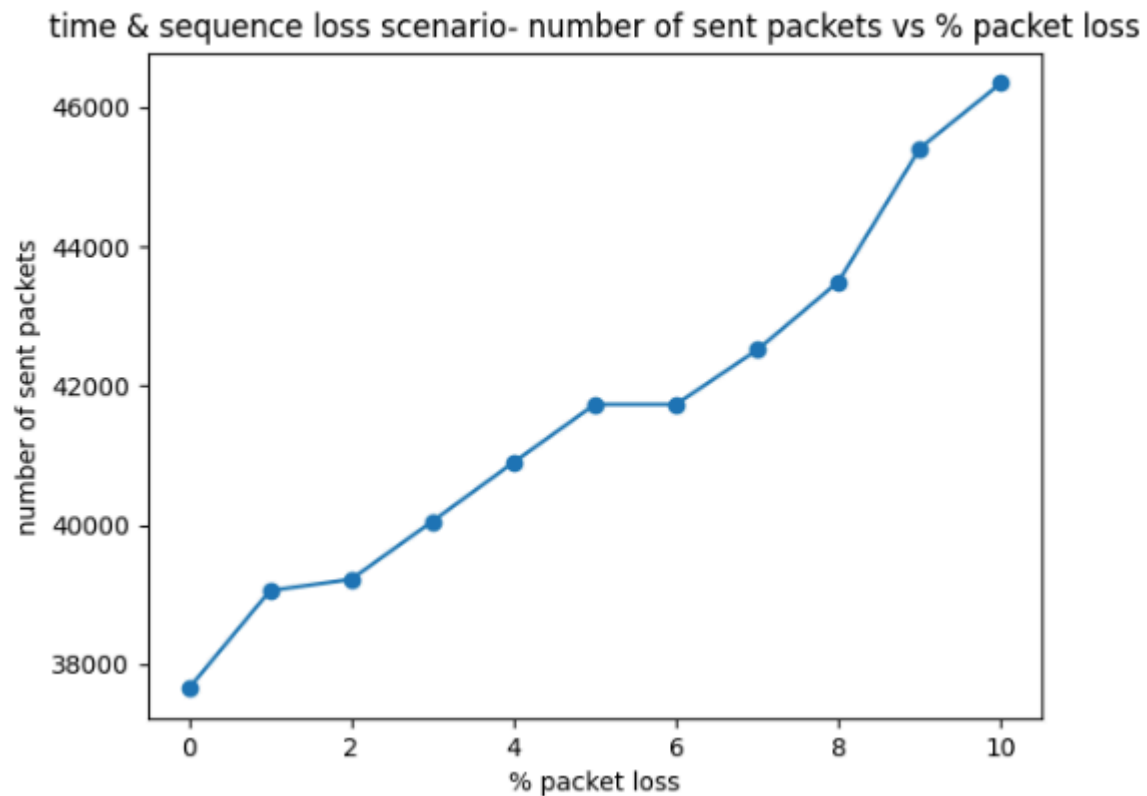
-----STATISTICS-----
-----FOR THE CURRENT FILE-----
Time taken to receive file: 82925.47 ms
Speed of file transfer: 121580.98 bytes/second
-----AVERAGE-----
average time :82925.47273635864 ms
average speed :121580.97707870504 bytes/second
exit file
done file
```

Table of statistics:

result/%	6%	7%	8%	9%	10%
Time(ms)	83067.68	82382.25	83130.06	83142.70	82925.47
speed(b/s)	121372.83	122382.67	121281.75	121263.32	121580.98
last seq	41735	42518	43498	45412	46348

result/%	0%	1%	2%	3%	4%	5%
Time(ms)	108044.68	107919.01	109085.24	107416.47	105896.39	109106.22
speed(b/s)	93314.72	93423.39	92424.61	93860.47	95207.78	92406.83
last seq	37652	39059	39222	40055	40902	41735

Graph:



as we can see, when there is bigger packet loss percentage we send more packets in total also if we handle the packet loss with both time and sequence number scenario together. As before, its because when a packet get lost we need to send it again so the more packet loss percentage means the more packets that are getting lost and the more packets we need to send again.

Wireshark recording- explanation

No.	Time	Source	Destination	Protocol	Length	Info
3	2.484513699	127.0.0.1	127.0.0.1	UDP	74	74 47482 -> 33002 Len=30
4	2.484674244	127.0.0.1	127.0.0.1	UDP	74	74 33002 -> 47482 Len=30
5	4.614938877	10.0.2.15	239.255.255.250	SSDP	217	M-SEARCH * HTTP/1.1
6	4.560565772	127.0.0.1	127.0.0.1	UDP	75	75 47482 -> 33002 Len=31
7	4.560808014	127.0.0.1	127.0.0.1	UDP	160	33002 -> 47482 Len=116
8	4.560831710	127.0.0.1	127.0.0.1	UDP	96	33002 -> 47482 Len=52
9	5.016132316	10.0.2.15	239.255.255.250	SSDP	217	M-SEARCH * HTTP/1.1
10	6.017883887	10.0.2.15	239.255.255.250	SSDP	217	M-SEARCH * HTTP/1.1
11	6.722831226	127.0.0.1	127.0.0.1	UDP	85	47482 -> 33002 Len=41
12	6.725742245	127.0.0.1	127.0.0.1	UDP	337	47482 -> 33002 Len=293
13	6.725826817	127.0.0.1	127.0.0.1	UDP	50	33002 -> 47482 Len=6
14	6.727929668	127.0.0.1	127.0.0.1	UDP	337	47482 -> 33002 Len=293
15	6.728010375	127.0.0.1	127.0.0.1	UDP	50	33002 -> 47482 Len=6
16	6.738062945	127.0.0.1	127.0.0.1	UDP	330	47482 -> 33002 Len=292
17	6.738118540	127.0.0.1	127.0.0.1	UDP	50	33002 -> 47482 Len=6
18	6.732520652	127.0.0.1	127.0.0.1	UDP	338	47482 -> 33002 Len=294
19	6.732607039	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
20	6.734700313	127.0.0.1	127.0.0.1	UDP	338	47482 -> 33002 Len=294
21	6.734757480	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
22	6.736097424	127.0.0.1	127.0.0.1	UDP	338	47482 -> 33002 Len=294
23	6.736969066	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
24	6.739535259	127.0.0.1	127.0.0.1	UDP	337	47482 -> 33002 Len=293
25	6.739628906	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
26	6.741771300	127.0.0.1	127.0.0.1	UDP	338	47482 -> 33002 Len=294
27	6.741836704	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
28	6.743872272	127.0.0.1	127.0.0.1	UDP	338	47482 -> 33002 Len=294
29	6.743946320	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
30	6.746096486	127.0.0.1	127.0.0.1	UDP	338	47482 -> 33002 Len=294
31	6.746138683	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
32	6.748275725	127.0.0.1	127.0.0.1	UDP	339	47482 -> 33002 Len=295
33	6.748329292	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
34	6.750369459	127.0.0.1	127.0.0.1	UDP	339	47482 -> 33002 Len=295
35	6.750406662	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
36	6.753021454	127.0.0.1	127.0.0.1	UDP	339	47482 -> 33002 Len=295
37	6.753107637	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
38	6.756118963	127.0.0.1	127.0.0.1	UDP	339	47482 -> 33002 Len=295
39	6.757383287	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
40	6.758321132	127.0.0.1	127.0.0.1	UDP	339	47482 -> 33002 Len=295
41	6.758510207	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
42	6.760587291	127.0.0.1	127.0.0.1	UDP	339	47482 -> 33002 Len=295
43	6.760657539	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7
44	6.760807997	127.0.0.1	127.0.0.1	UDP	339	47482 -> 33002 Len=295
45	6.768951600	127.0.0.1	127.0.0.1	UDP	51	33002 -> 47482 Len=7

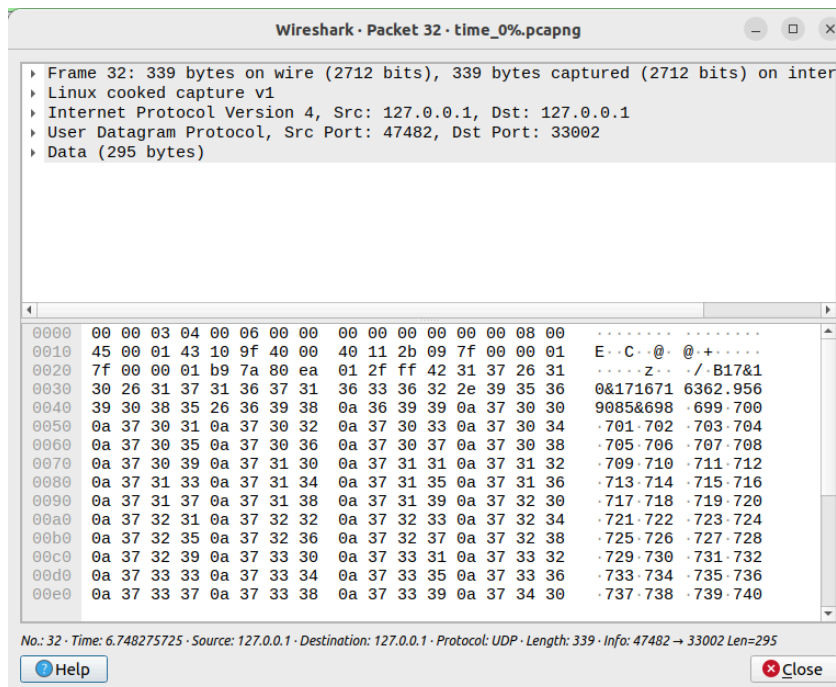
This is how a Wireshark recording looks like for our code, we can see the packets that were sent from the client to the server and the opposite. The client sent to the server packets, each in size of 295 bytes that include the file which the client sends to the server and more information that each packet contains such as sequence number, the time of the packet creation(which is the time the packet getting sent) and more. In return, the server sent ack that the packet received which is in length of 7 bytes.

This picture shows the packet for the handshake, the client send a packet with the payload “hello” and the connection ID(which in this example is 1) that the client would like to use.

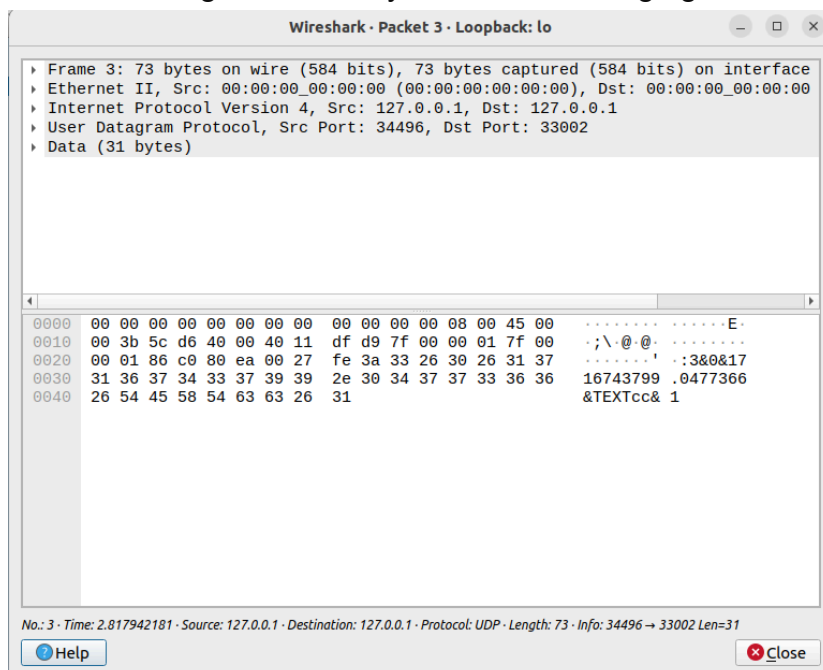
We can see the protocol we are using(UDP), the source port(47482) and the destination port(33002).

Wireshark - Packet 3 - time_0%.pcapng	
Frame 3: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface	
Linux cooked capture v1	
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	
User Datagram Protocol, Src Port: 47482, Dst Port: 33002	
Data (30 bytes)	
0000	00 00 03 04 00 06 00 00 00 00 00 00 00 00 00 00
0010	45 00 00 3a 0c c1 40 00 40 11 2f f0 7f 00 00 01
0020	7f 00 00 01 b9 7a 80 ea 00 26 fe 39 31 26 39 26
0030	31 37 31 36 37 31 36 33 35 38 2e 36 39 33 31 32
0040	34 33 26 08 05 6c 6c 6f 20 31

This picture shows the structure of an information packet. This built the same as the handshake packet, the only difference is that the payload is different and its the text from the file the client sent.



This is the packet the client send to the server with his name so he can start to send the files he would like. We can see the payload is “cc” is the name of the client and the frame is 3, so its the third packet that was sent between the client and the server, the first one is the “hello” packet from before(the first packet) its the same one which the client send the connection ID for the connection. The second packet is the packet the server send to let the client know the he received the opening packet and the connection is good and they can start messaging and sending files.



This is the last packet of the file that was sent from the client to the server, we can know that by seeing the payload which is “DONE”, this is the signal of the last packet that was sent about the file.

