Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4

Predmet / Subject

–Počítačové a komunikačné siete / Computer and communication networks –

- Dokumentácia / Documentation -

Zadanie č.2 – dokumentácia

Ak. Rok / Academic term: 2022/2023, zimný semester

Cvičiaci / Instructors:
Ing. Lukáš Mastiľák

Študent / Student:
Adam Grík

Bratislava, 2022.

Obsah / Content:

1. Úvod

Daný dokument je dokumentáciou k zadanie č. 2 z predmetu Počítačové a komunikačné siete. Úlohou v tomto zadaní bolo navrhnúť a implementovať vlastnú komunikáciu medzi dvoma uzlami s využitím protokolu UDP. Program pozostáva z dvoch častí – vysielacej a prijmácej, to znamená že jeden uzol posiela dáta (textová správa, súbor), naopak uzol na druhej strane tieto údaje príjma. Vysielacia strana môže tak isto požiadať o zmenu rolí, v tom prípade sa role na týchto uzloch navzájom vymenia.

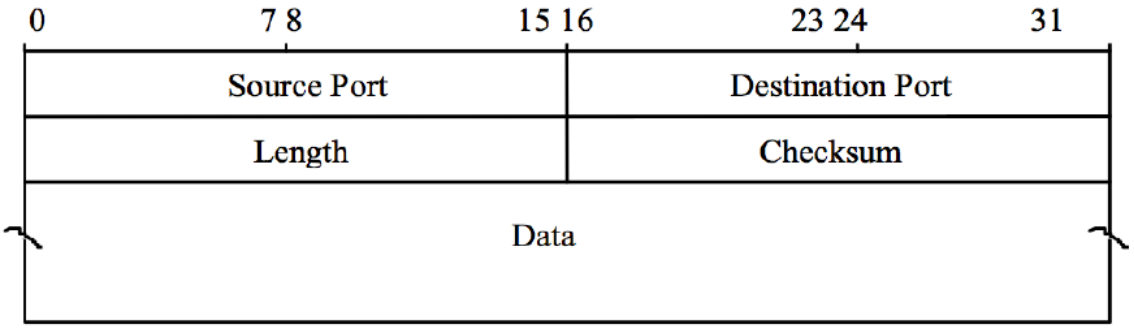
1.1. UDP komunikácia

Protokol UDP je bez spojenia, bez potvrdenia a preto sa považuje za nespoľahlivý. Väčšinou sa využíva na klient-server aplikácie. Aplikačné dáta (textové správy, súbory) balí do tzv. datagramov.

Nevýhodou tohto protokolu je hlavne to, že nezriaďuje spojenie pred prenosom dát, nepotvrdzuje prijaté dáta, nedeteguje straty a tak isto nie je možnosť požadovať opakovanie prenosu dát.

1.2. UDP datagram

Maximálna veľkosť jedného datagramu je 65 535 bajtov, avšak od tohto musíme odpočítať IP hlavičku, UDP hlavičku a tak isto aj moju vlastnú hlavičku. IP hlavička má 20 bajtov, UDP hlavička má 8 bajtov, a moja vlastná hlavička pri posielaní textovej správy má 5 bajtov a pri posielaní súboru má 7 bajtov.
To znamená že, 65 535 – 20 – 8 – 7 = 65 500 alebo 65 535 – 20 – 8 – 5 = 65 502. V tomto programe nechceme fragmentovať na linkove vrstve to znamená že, datagram bude mať maximálnu veľkosť 1500 bajtov, tak isto aj v tomto prípade musíme odpočítať IP hlavičku, UDP hlavičku a moju vlastnú hlavičku. To znamená že maximálna veľkosť správy ktorú budeme môcť poslať je 1465 alebo 1467 bajtov.



2. Všeobecné informácie

Riešenie zadania č. 2 som implementoval v programovacom jazyku Python. Ovládanie programu a používateľské rozhranie bude bežať v konzolovom výpise.

Ak si používateľ zvolí rolu klienta, tak si program od neho vypýta IP adresu a port serveru, po zadaní týchto dvoch údajov dá program používateľovi na výber z niekoľkých možností.

Ak si klient zvolí možnosť poslania textovej správy, tak si program od neho vypýta n iekoľko údajov. Ako prvé musí používateľ zadať samotnú správu, následne veľkosť fragmentu a ako posledné klient zadá koľko chýb si praje simulovať.

Pri zvolení možnosti poslania súboru to prebieha v podstate podobne a to tak, že klient musí ako prvé zadať absolútnu cestu k súboru, a následne opäť zadať veľkosť fragmentu a počet koľko chýb si praje simulovať.

Ak klient zvolí možnosť že si chce vymeniť role, tak uzly si medzi sebou jednoducho vymenia packety a následne sa im prehodia role, to znamená že klient bude server a server bude klient.

Ak chce klient skončiť program, tak si jednoducho zvolí tú danú možnosť, opäť si uzly medzi sebou vymenia packety a program bude ukončený.

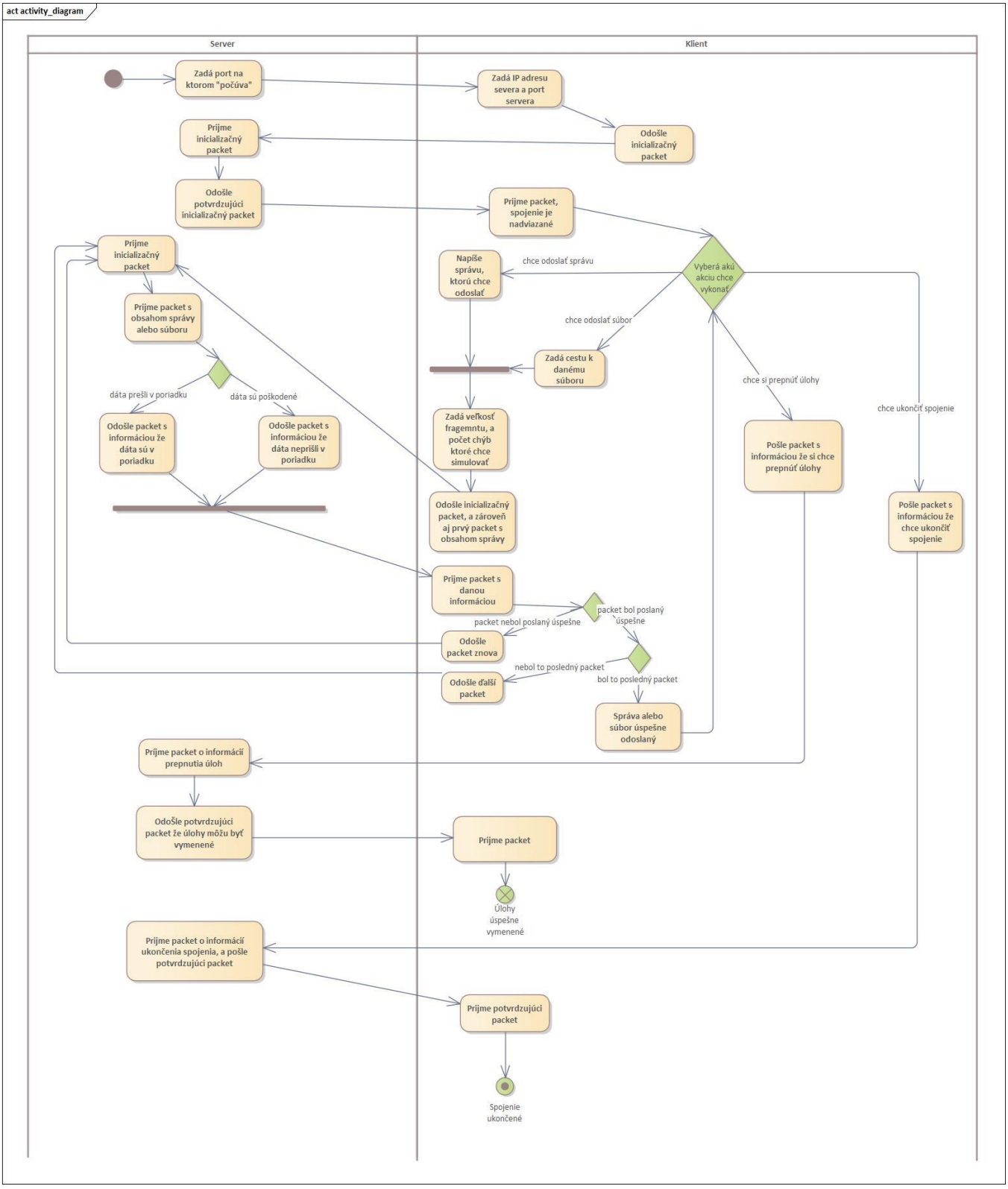
Pri každom posielaní packetu, či už je to packet s údajmi, packet na výmenu alebo ukončenie programu uzol na strane klienta čaká 10 sekúnd na potvrdenie, ak toto potvrdenie neprijde tak to zopakuje ešte 2 krát a ak ani potom nepríde potvrdenie, tak program je ukončený.

Na druhej strane ak si používateľ zvolí rolu serveru, tak na začiatku programu mu iba stačí zadať číslo portu na ktorom bude server “počúvať”, následne bude používateľ zakaždým vyzvaný na interakciu serveru, po danej interakcii sa používateľovi na strane servera zobrazia údaje o konkrétnych packetoch či boli prijaté úspešne alebo neúspešne a nakoniec program zobrazí dané údaje o správe resp. súbore (veľkosť, obsah, počet packetov, veľkosť fragmentu atď.).

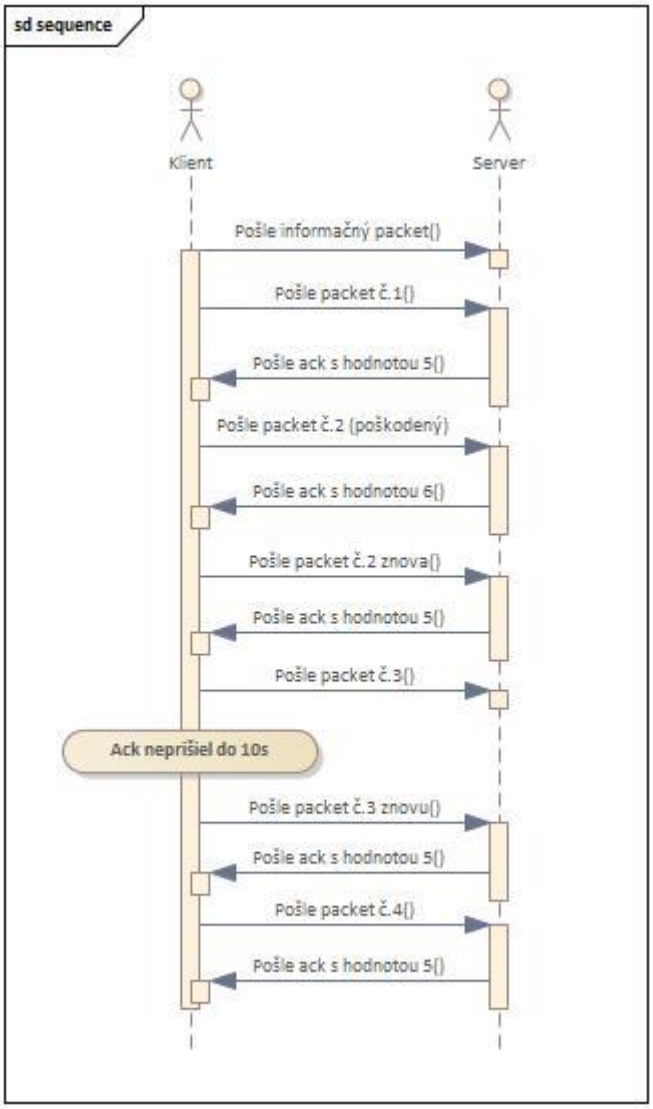
Pri nečinnosti oboch strán bude bežať tzv. keep alive metóda, to znamená že dané uzly si budú každých 5 sekúnd vymieňať packety. Ak server nebude tri krát po sebe po 5 sekundách reagovať na keep alive packety od klienta, tak program bude automaticky ukončený.

3. Diagramy spracovávania komunikácie

3.1. Diagram aktivít celého programu



3.2. Sekvenčný diagram – ARQ metóda – STOP & WAIT



4. Návrh štruktúry vlastnej hlavičky

4.1. Zmeny v štruktúre hlavičiek oproti návrhu

Pri implementácií som vykonal zmeny v niektorých štruktúrach vlastných hlavičiek.

Do hlavičky informatívnych správ som pri prípade že sa odosiela súbor , pridal dátovú časť, kde sa bude vždy nachádzať názov samotného súboru ktorý sa prenáša a takisto v prípade posielania súboru som pole s informáciou o celkovom počte packetov zvýšil z 2B na 4B, z dôvodu, že ak posielame súbor napr. 2MB a nastavíme veľkosť fragmentu na 1B, tak číslo ktoré reprezentuje počet packetov je príliš veľké nato aby sa zmestilo do 2B.

V hlavičkách ktoré posielajú dané údaje správy resp. súboru som zmazal bajty ktoré reprezentovali veľkosť fragmentu daného packetu.

A takisto pri posielaní súboru som pole kde sa nachádza informácia o poradí packetu zvýšil z 2B na 4B, z dôvodu, že ak posielame súbor napr. 2MB a nastavíme veľkosť fragmentu na 1B, tak číslo ktoré reprezentuje poradie packetu je príliš veľké nato aby sa zmestilo do 2B.

4.2. Hlavička inicializačných packetov

Inicializačná hlavička sa bude nachádzať v inicializačných správach, ktoré zabezpečujú nadviazanie spojenia. Tieto packety budú obsahovať iba jeden bajt.

1.bajt
Typ

Typ – môže mať hodnotu buď 1 alebo 2.
Hodnotu 1 odosiela klient keď chce začať spojenie, a hodnotu 2 odosiela server keď chce potvrdiť inicializovanie spojenia.

4.3. Hlavička informatívneho packetu pred posielaním správy

Pred samotným posielaním packetov s obsahom správy, bude klient posielať tzv. informatívnu správu ktoré bude obsahovať iba hlavičku a bude vyzerať následovne.

Hlavička bude mať veľkosť 6 bajtov.

1.bajt	2. – 3. bajt	4. – 5. bajt	6.
Typ	Veľkosť správy (súboru)	Počet fragmentov	Typ správy

Typ – na tomto bajte sa bude nachádzať vždy hodnota 3, čo znamená že sa jedná o informatívny packet

Veľkosť správy – na týchto dvoch bajtoch sa bude nachádzať veľkosť celej správy, alebo súboru ktorý bude odosielaný.

Počet fragmentov – na týchto dvoch bajtoch sa bude nachádzať celkový počet packetov ktoré

budú odoslané.

Typ správy – na tomto bajte sa bude nachádzať hodnota 1, pretože budeme posielať správu.

4.4. Hlavička informatívneho packetu pred posielaním súboru

Pred samotným posielaním packetov s obsahom súboru, bude klient posielať tzv. informatívnu správu ktoré bude obsahovať iba hlavičku a bude vyzerať následovne.

Hlavička bude mať veľkosť 8 bajtov, následne packet bude obsahovať ešte aj dátovú časť ktorá bude obsahovať názov súboru.

1.bajt	2. – 3. bajt	4. – 7. bajt	8.	9. – n.
Typ	Veľkosť správy (súboru)	Počet fragmentov	Typ správy	Názov súboru

Typ – na tomto bajte sa bude nachádzať vždy hodnota 3, čo znamená že sa jedná o informatívny packet

Veľkosť správy – na týchto dvoch bajtoch sa bude nachádzať veľkosť celej správy, alebo súboru ktorý bude odosielaný.

Počet fragmentov – na týchto dvoch bajtoch sa bude nachádzať celkový počet packetov ktoré budú odoslané.

Typ správy – na tomto bajte sa bude nachádzať hodnota 2, pretože budeme posielať súbor

Názov súboru – v tejto dátovej časti sa bude nachádzať názov súboru.

4.5. Hlavička packetu s údajmi

Pri packetoch, ktoré obsahujú už samotné údaje ktoré ideme posielať bude mať hlavička veľkosť 5 bajtov prípade ak posielame správu, v prípade ak posielame súbor bude mať veľkosť 7 bajtov a budú vyzerať následovne.

Správa:

1.bajt	2. – 3. bajt	4. – 5. bajt
Typ	Poradie packetu	Checksum

Súbor:

1.bajt	2. – 5. bajt	6. – 7. bajt
Typ	Poradie packetu	Checksum

Typ – Na tomto bajte sa bude vždy nachádzať hodnota 4, ktorá znamená to, že sa posiela packet s údajmi.

Poradie packetu – na týchto bajtoch sa bude nachádzať číslo, ktoré predstavuje poradie packetu.

Checksum – na týchto dvoch bajtoch sa bude nachádzať hodnota checksum.

4.6. Hlavička potvrdzujúcich packetov

Potvrdzujúce packety bude posielať iba server, a to vždy po odoslaní jedného packetu s údajmi. Táto správa bude mať veľkosť 1 bajt čo znamená, že samotná hlavička bude mať 1 bajt.

1.bajt
Typ

Typ – na tomto bajte sa bude nachádzať bud hodnota 5 alebo 6. Hodnota 5 bude znamenať že daný packet od klienta prešiel v poriadku bez poškodenia a straty dát, a hodnota 6 bude znamenať že daný packet od klienta neprešiel v poriadku a bude ho musieť klient odoslať ešte raz.

4.7. Hlavička keep alive packetov

Keep alive packety bude odosielať aj klient aj server a budú mať veľkosť iba 1 bajt.

1.bajt
Typ

Typ – nezáleží na tom či bude packet odosielať klient alebo server, na tomto bajte sa bude vždy nachádzať hodnota 8, čo znamená že sa jedná o packet na udržanie spojenia, ktorý sa bude odosielať každých 5 sekúnd.

4.8. Hlavička packetu ukončujúceho spojenie

Tento packet sa bude posielať ak klient bude chcieť ukončiť spojenie.

1.bajt
Typ

Typ – na tomto bajte sa bude nachádzať vždy nachádzať hodnota 9, ktorú zašle klient ako prvý, server ju príjme a ak pošle aj on packet s hodnotou 9, tak bude spojenie úspešne ukončené

4.9. Hlavička packetu na výmenu rolí

Tento packet sa bude posielať ak klient bude chcieť vymeniť role. Tento packet bude ako prvý posielať klient, a server odošle potvrdzujúci packet s tou istou hodnotou.

1.bajt
Typ

Typ – na tomto bajte sa bude vždy nachádzať hodnota 7

5. Opis metód a funkčností

5.1. Opis metódy kontrolnej sumy

Na metódu kontrolnej sumy použijem CRC funkciu. Metóda CRC funguje tak, že ako prvé si zistíme dĺžku (L) tzv. divisora, ktorý je známy pre obe strany. Ako dalšie pridáme L-1 bitov na koniec správy, následne sa použije binárna operácia XOR a zvyšok čo dostanem po tejto operácií je výsledná hodnota CRC.
V mojom programe budem konrétne pracovať s metódou, ktorá sa volá *crc_hqx* a je z knižnice *binascii*. Táto metóda počíta 16-bitovú hodnotu CRC z poskytnutých údajov. Ako jeden z parametrov tejto funkcie berie počiatočné CRC, vo veľa prípadoch je táto hodnota nastavená na 0, tak to je aj v mojom programe.

Metóda pracuje konkrétne s týmto polynómom (divisorom):

$$x^{16} + x^{12} + x^5 + 1$$

5.2. Opis fungovania ARQ

Pre ARQ metódu som si vybral typ Stop & Wait. Tento typ funguje vo všeobecnosti tak, že klient posiela packety po jednom, a vždy čaká na potvrdzujúcu správu od serveru a až potom odošle ďalší packet alebo odošle ten istý packet znova pokiaľ nedošiel v poriadku.

V mojom programe to bude fungovať tak, že klient bude posielať packety po jednom, vždy keď odošle jeden packet, tak bude čakať na potvrdzujúci packet od serveru. V tomto packete od serveru sa bude nachádzať buď hodnota 5 alebo 6. Ak sa tam bude nachádzať hodnota 5, znamená to, že packet prišiel v poriadku a klient môže odoslať ďalší packet. Ak sa tam bude nachádzať hodnota 6, znamená to, že packet neprišiel v poriadku takže klient odošle daný packet ešte raz. A tento cyklus prebieha stále pri každom packete, až kým nebudú poslané všetky packety.

V prípade že klientovi nepríde do 10 sekúnd potrdzujúci packet od serveru, tak odošle packet s danými údajmi ešte raz, ak sa táto situácia zopakuje 3 krát po sebe, tak program bude ukončený.

5.3. Metóda na udržanie spojenia – keep alive

Metódu na udržanie spojenia budem riešiť pomocou threadu a cez vlastné signalizačné správy. Tieto signalizačné správy sa budú posielať každých 5 sekúnd. Klient bude odosielať serveru každých 5 sekúnd packet o veľkosti 1 bajt, kde sa bude nachádzať hodnota 8. Ak serveru dorazí tento packet, tak mu odošle naspäť packet o veľkosti 1 bajt, s rovnakou hodnotou čiže 8.

Ak klient odošle serveru packet s hodnotou 8, a server mu do 5 sekúnd neodošle potvrdzujúci packet s hodnotou 8 naspäť, tak klient pošle serveru znova packet s hodnotou 8, ak sa táto situácia zopakuje 3 krát po sebe, tak program bude ukončený.

5.4. Simulovanie chybného packetu

Program je navrhnutý tak, že dokáže simulovať poslanie chybného packetu, a to tak, že používateľ na strane klienta si môže zvoliť aký počet chýb chce simulovať, následne program taký počet chybných packetov aj nasimuluje. Samotnú chybu v packete simulujem tak, že v hodnote checksum odrátam hodnotu -1.

5.5. "Time exceptions"

V programe mám zakomponované aj tzv. "timeout exceptions",. To znamená že pri každom posielaní packetu zo strany klienta, je nastavený nejaký časový timeout. Funguje to tak, že ak klient pošle packet (packet s údajmi, packet na výmenu rolí, packet na ukončenie spojenia), tak vždy čaká 10 sekúnd na potvrdzovací packet od serveru, ak tento packet nepríde do 10 sekúnd, tak klient pošle ten istý packet znovu. Ak sa tento proces zopakuje 3 krát po sebe, spojenie sa ukončí. V prípade metódy keep alive je to vysvetlené vyššie.

6. Opis niektorých vybraných častí kódu

V tejto časti sa budem stručne snažiť opísať niektoré hlavné časti môjho implementovaného kódu.

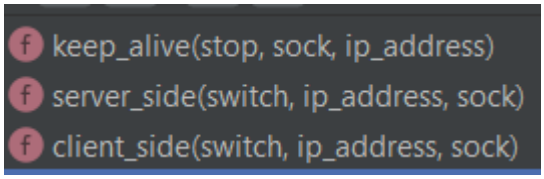
6.1. Použité knižnice

V mojom programe používam niekoľko knižníc, ako napr. threading (na prácu s threadom), struct (na prácu s bajtami), socket (na prácu s UDP socktemi), binascii (pre výpočet crc).

```
import binascii
import math
import socket
import threading
from struct import pack, unpack
import random
from time import sleep
import os
from _thread import interrupt_main
```

6.2. Vytvorené metódy

V mojom programe sa nachádzajú 3 metódy plus hlavná metóda main. Metóda keep alive je implementovaná ako thread, a slúži na udržanie spojenia. Metódy server_side a client_side je už implementácia samotných rolí.



6.3. Algoritmus na posielanie packetov so správou

Pri posielaní správy sa najprv od používateľa, ktorý je ako klient vypýtam zo vstupu všetky potrebné údaje, následne mu odošlem prvý informatívny packet, ešte predtým ako idem posielať samotné packety. Potom si vytvorím random pole chýb, na základe hodnoty akú zadal používateľ

```
if (menu == "m"):
    stop_threads = True
    t1.join()

    message = input("Napis spravu, ktoru chces poslat: ")

    print("Velkost spravy: ", len(message))

    while True:
        size_of_fragment = input("Zadaj velkost fragmentu: ")
        if (int(size_of_fragment) > 1467):
            print("---------")
            print("Velkost fragmentu je prilis velka.")
            print("---------")
            continue
        else:
            break

    number_of_fragments = math.ceil(len(message) / int(size_of_fragment))

    print("Pocet packetov na odoslanie: ", number_of_fragments)

    number_of_errors = input("Zadaj kolko chyb si prajes nasimulovat: ")

    sock.sendto(pack("c", str.encode("3")) +
                pack("c", str.encode("1")) + pack("h", len(message)) + pack("h", number_of_fragments),
                ip_address)

    # create a values for simulating errors
    list = range(0, int(number_of_fragments))
    randoms = random.sample(list, int(number_of_errors))
    randoms.sort()
    print(randoms)
```

Potom v cykle while, ktorý beží pokiaľ nedosiahnem počet všetkých fragmentov, posielam po jednom packety. A to tak, že najprv si vytvorím hlavičku, a následne za túto hlavičku pridám ešte samotné dáta. Pritom ešte ak sú splnené podmienky, tak robím simuláciu chyby, a to tak že odpočítam od checksum hodnoty hodnotu -1. Potom celý packet pošlem serveru. Následne čakám na potvrdzujúci packet od servera, ak je v ňom hodnota 5 posielam ďalší packet, ak je tam hodnota 6, nastavím index na aktuálnu hodnotu a idem posielať znova ten istý packet.

```
while i < number_of_fragments:
    string = ""
    tmp = index

    try:
        for element in range(int(size_of_fragment)):
            if (index < len(message)):
                string += message[index]
                index = index + 1
        header = pack("c", str.encode("4")) + pack("h", i)
        checksum = binascii.crc_hqx(header + str.encode(string), 0)
        if randoms_index < len(randoms):
            if i == randoms[randoms_index]:
                checksum = checksum + 1  # test
                randoms_index = randoms_index + 1
        header = header + pack("H", checksum)
        sock.sendto(header + str.encode(string), ip_address)

        if (pom == 0):
            data, ip_address = sock.recvfrom(1500)

        sock.settimeout(10)

        data, ip_address = sock.recvfrom(1500)
        number_of_except = 0
        data = data.decode()

        if (data == '5'):
            i = i + 1
        else:
            index = tmp
        pom = pom + 1
```

A nakoniec mám timeout exception, ktorý je presnejšie popísaný v kapitole vyššie.

```
except(socket.timeout):
    number_of_except = number_of_except + 1
    if (number_of_except < 3):
        index = tmp
        pom = pom + 1
        continue
    else:
        print("----------")
        print("Server nereaguje, program je ukonceny")
        exit()
```

6.4. Algortimus na príjmanie packetov so správou

Celé príjmanie packetov prebieha v cykle while, ktorý beží až kým hodnota nebude rovná počtu všetkých packetov, pričom tento údaj sme zistili z informatívneho packetu. Následne čitam packet po packete, kde pomcou funckií *decode* alebo *unpack* rozkladám dané údaje, dáta a ukladám si ich do premenej. Potom overujem či mi sedí checksum, ak nesedí odošlem packet s hodnotou 6 a vypíšem do konzoly že packet bol prijatý neúspešne, ak sedí odošlem packet s hodnotou 6 a vypíšem do konzoly že packet bol prijatý úspešne.

```
if (type_of_message[0] == b'1'):

    number_of_fragments = unpack("h", data[4:6])

    i = 0
    message = ""
    size_of_fragment = 0
    size_of_last_fragment = 0
    while i < number_of_fragments[0]:
        data, ip_address = sock.recvfrom(1500)

        client_checksum = unpack("H", data[3:5])
        client_message = data[5:].decode()
        client_packet_number = unpack("h", data[1:3])

        checksum = binascii.crc_hqx(data[:3] + data[5:], 0)

        if (i == 0):
            size_of_fragment = len(client_message)
        if (i == (number_of_fragments[0] - 1)):
            size_of_last_fragment = len(client_message)

        if (checksum == client_checksum[0]):
            if (client_packet_number[0] == (i - 1)):
                continue
            print("Packet cislo", client_packet_number[0], "bol prijaty uspesne.")
            message = message + client_message
            sock.sendto(pack("c", str.encode("5")), ip_address)
            i = i + 1
        else:
            if (client_packet_number[0] == (i - 1)):
                continue
            print("Packet cislo", client_packet_number[0], "bol prijaty NEUSPESNE.")
            sock.sendto(pack("c", str.encode("6")), ip_address)
```

Následne už len vypíšem do konzoly všetky potrebné údaje o danej správe.

```
print("--------------")
print("BOLA PRIJATA NOVA SPRAVA")
print("Obsah poslanej spravy: ", message)
print("Velkost spravy: ", len(message))
print("Celkovy pocet fragmentov: ", number_of_fragments[0])
print("Velkost fragmentu: ", size_of_fragment)
print("Velkost posledneho fragmentu: ", size_of_last_fragment)
print("--------------")
```

6.5. Metóda keep alive

Metóda keep alive je implementovaná ako thread, v podstate beží stále a vďaka nej klient odosiela packety každých 5 sekúnd. Na začiatku sa vždy opýtam podmienku *if stop()*, čo znamená že či thread nebol zrušený z dôvodu že sa idú posielať packety s údajmi. Metóda obsahuje aj *time exception* ktorý je nastavený na 5 sekúnd.

```
def keep_alive(stop, sock, ip_address):
    pom = 0
    no_response = 0
    while True:
        sock.settimeout(5)
        try:
            if stop():
                return
            if (pom == 0 or no_response > 0):
                sock.sendto(pack("c", str.encode("8")), ip_address)
                pom = pom + 1

            data, ip_address = sock.recvfrom(1500)
            data = data.decode()

            sleep(5)
            if (data == '8'):
                no_response = 0
                sock.sendto(pack("c", str.encode("8")), ip_address)
        except(socket.timeout):
            no_response = no_response + 1
            if (no_response < 3):
                continue
            else:
                print("\n---------")
                print("Spojenie bolo zrusene, server nereaguje.")
                print("---------")
                interrupt_main()
                return
```

7. Používateľské rozhranie

7.1. Klient

Takto vyzerá používateľské rozhranie na strane klienta. Zeleným písmom sú vyznačené písmená ktoré zadával používateľ do konzoly.

```
C:\Users\adam4\FIIT\zadanie2\venv\Scripts\python.exe C:/Users/adam4/FIIT/zadanie2/main2.py
Server --> s
Klient --> k
Zadaj rolu pod ktorou chces vykonavat proces: k
Zadaj IP adresu servera: 127.0.0.1
Zadaj port servera: 4040
----------
Uspesne spojenie so serverom:  ('127.0.0.1', 4040)
----------
Vyber si moznost: (m --> poslat spravu, f --> poslat subor, s --> vymenit si ulohy, e --> skoncit program) :m
Napis spravu, ktoru chces poslat: ahoj, ja som klient
Zadaj velkost fragmentu: 5
Pocet packetov na odoslanie:  4
Zadaj kolko chyb si prajes nasimulovat: 2
----------
Bude odoslana sprava !
Obsah spravy:  ahoj, ja som klient
Velkost spravy:  19
Pocet packetov na odoslanie:  4
Velkost fragmentu:  5
Velkost posledneho fragmentu:  4
----------
```

7.2. Server

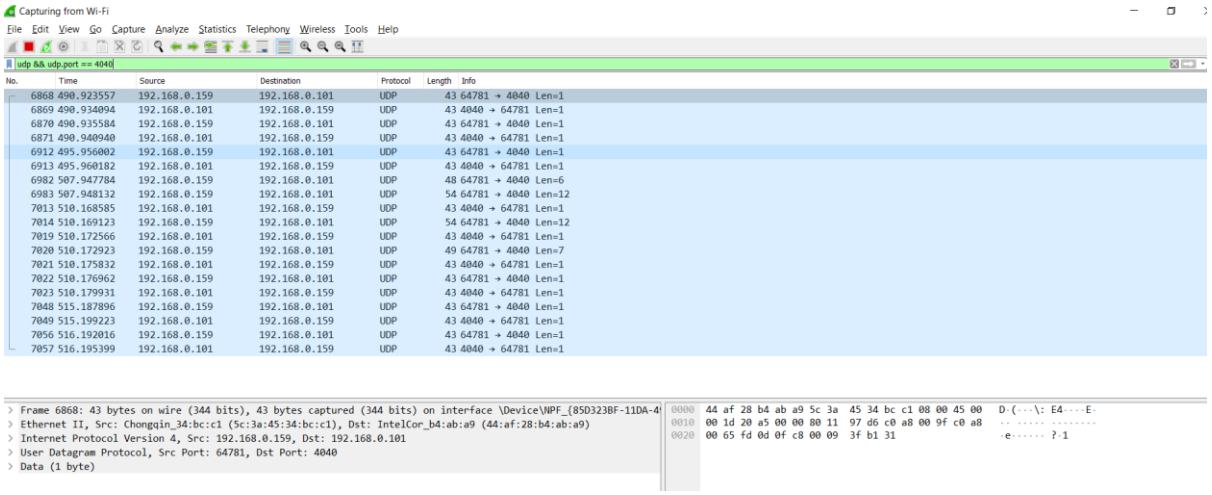
Takto vyzerá používateľské rozhranie na strane serveru.

```
C:\Users\adam4\FIIT\zadanie2\venv\Scripts\python.exe C:/Users/adam4/FIIT/zadanie2/main.py
Server --> s
Klient --> k
Zadaj rolu pod ktorou chces vykonavat proces: s
Zadaj cislo portu pre server: 4040
----------
Spojenie inicializovane z adresy:  ('127.0.0.1', 52432)
Pre interakciu servera stlac enter
Cakam na data
----------
Packet cislo 0 bol prijaty uspesne.
Packet cislo 1 bol prijaty NEUSPESNE.
Packet cislo 1 bol prijaty uspesne.
Packet cislo 2 bol prijaty NEUSPESNE.
Packet cislo 2 bol prijaty uspesne.
Packet cislo 3 bol prijaty uspesne.
--------------
BOLA PRIJATA NOVA SPRAVA
Obsah prijatej spravy:  ahoj, ja som klient
Velkost spravy:  19
Celkovy pocet packetov:  4
Velkost fragmentu:  5
Velkost posledneho fragmentu:  4
--------------
```

8. Wireshark

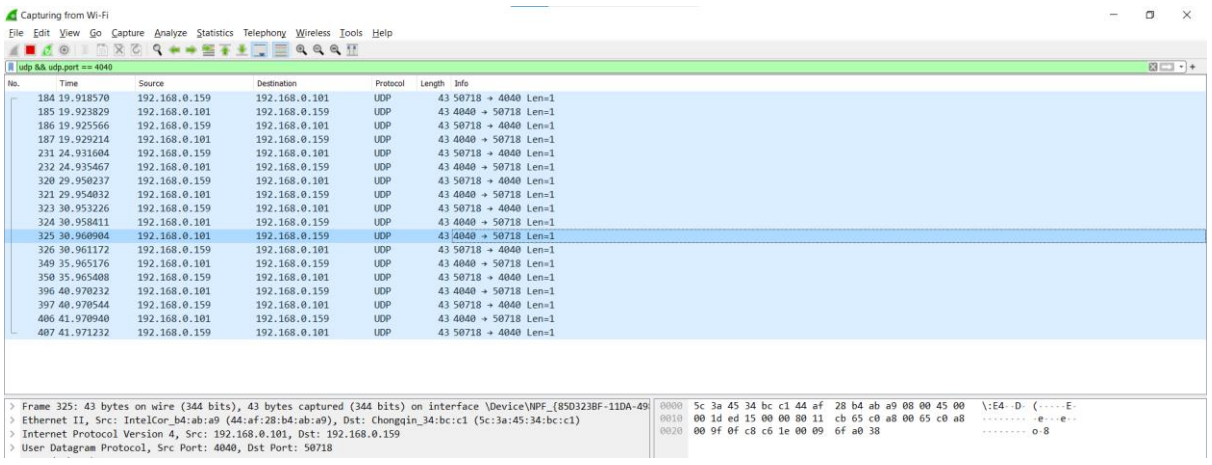
8.1. Posielanie správy

Na screenshote nižšie môžeme vidieť zachytenú komunikáciu medzi dvoma uzlami, ktoré komunikovali cez môj program. Komunikácia zachytáva packety od nadvaziana spojenia, cez posielanie správy až po ukončenie spojenia.



8.2. Výmena rolí

Na tomto screenshote je zachytená komunikácia, ale tento krát sa jedná o výmenu rolí. Môžeme vidieť že na od označeného packetu si vymenili role, čo dokazuje to že keep alive packety začala posielať ako prvá opačná strana, a tak isto ukončenie spojenia inicializovala druhá strana.



9. Záver

Vďaka testovaniu rôznych scenárov som zistil, že mnou navrhnutá implementácia pre UDP komunikáciu medzi dvoma uzlami funguje správne. Otestovaných bolo viacero scenárov, a všetko funguje tak ako má. Program spĺňa všetky požiadavky ktoré sú uvedené v zadaní, a takisto pracuje s datámi optimálne.