FIIT STU

Počítačové a komunikačné siete

Zadanie 2: Komunikácia s použitím UDP protokolu
Cvičenie: Štvrtok 10:00-11:40, Košťál
Adam Miškove

Obsah

1. Zadanie:

Zadanie úlohy:

Navrhnite a implementujte program s použitím vlastného protokolu nad protokolom UDP (User Datagram Protocol) transportnej vrstvy sieťového modelu TCP/IP. Program umožní komunikáciu dvoch účastníkov v lokálnej sieti Ethernet, teda prenos textových správ a ľubovoľného súboru medzi počítačmi (uzlami).

Program bude pozostávať z dvoch častí – vysielacej a prijímacej. Vysielací uzol pošle súbor inému uzlu v sieti. Predpokladá sa, že v sieti dochádza k stratám dát. Ak je posielaný súbor väčší, ako používateľom definovaná max. veľkosť fragmentu, vysielajúca strana rozloží súbor na menšie časti - fragmenty, ktoré pošle samostatne. Maximálnu veľkosť fragmentu musí mať používateľ možnosť nastaviť takú, aby neboli znova fragmentované na linkovej vrstve.

Ak je súbor poslaný ako postupnosť fragmentov, cieľový uzol vypíše správu o prijatí fragmentu s jeho poradím a či bol prenesený bez chýb. Po prijatí celého súboru na cieľovom uzle tento zobrazí správu o jeho prijatí a absolútnu cestu, kam bol prijatý súbor uložený.

Program musí obsahovať kontrolu chýb pri komunikácii a znovuvyžiadanie chybných fragmentov, vrátane pozitívneho aj negatívneho potvrdenia. Po prenesení prvého súboru pri nečinnosti komunikátor automaticky odošle paket pre udržanie spojenia každých 5-20s pokiaľ používateľ neukončí spojenie. Odporúčame riešiť cez vlastne definované signalizačné správy.

Program musí mať nasledovné vlastnosti (minimálne):

1. Program musí byť implementovaný v jazykoch C/C++ alebo Python s využitím knižníc na prácu s UDP socket, skompilovateľný a spustiteľný v učebniach. Odporúčame použiť python modul socket, C/C++ knižnice sys/socket.h pre linux/BSD a winsock2.h pre Windows. Iné knižnice a funkcie na prácu so socketmi musia byť schválené cvičiacim. V programe môžu byť použité aj knižnice na prácu s IP adresami a portami:
arpa/inet.h
netinet/in.h
2. Program musí pracovať s dátami optimálne (napr. neukladať IP adresy do 4x int).
3. Pri posielaní súboru musí používateľovi umožniť určiť cieľovú IP a port.
4. Používateľ musí mať možnosť zvoliť si max. veľkosť fragmentu.
5. Obe komunikujúce strany musia byť schopné zobrazovať:
    a. názov a absolútnu cestu k súboru na danom uzle,
    b. veľkosť a počet fragmentov.
6. Možnosť simulovať chybu prenosu odoslaním minimálne 1 chybného fragmentu pri prenose súboru (do dátovej časti fragmentu je cielene vnesená chyba, to znamená, že prijímajúca strana deteguje chybu pri prenose).
7. Prijímajúca strana musí byť schopná oznámiť odosielateľovi správne aj nesprávne doručenie fragmentov. Pri nesprávnom doručení fragmentu vyžiada znovu poslať poškodené dáta.
8. Možnosť odoslať 2MB súbor a v tom prípade ich uložiť na prijímacej strane ako rovnaký súbor, pričom používateľ zadáva iba cestu k adresáru kde má byť uložený.

2. Zmeny voči návrhu môjho riešenia:

A. ARQ metóda:

V ohľade implementácie ARQ metódy neboli žiadne zmeny. Plánoval som použiť Stop and Wait ARQ, a tak sa aj stalo.

B. Kontrola správnosti prenesených dát:

Túto problematiku som plánoval riešiť použitím jednoduchého bitového checksumového algoritmu, no vzhľadom na spätnú väzbu od nášho pána cvičiaceho a spolužiakov, som sa rozhodol použiť Cyclic Redundancy Check(crc32b).

Bol implementovaný pomocou funkcie **crc32()** z pythonovskej knižnice **zlib.** Táto metóda je založená na nasledovnom generačnom polynóme:

$g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

Pomocou tohto prevedie string dát do stringu, ktorého veľkosť je 8 bytov, pričom reprezentuje hexadecimálnu hodnotu 32 bitovej binárnej postupnosti. Tento typ checksumu je efektívnejší a spoľahlivejší ako moja predošlá voľba, takže nebolo čo vyberať.

C: Maximálna veľkosť jedného packetu:

V návrhu bola táto hodnota opísaná ako 1500-20-8-(veľkost mojej hlavičky)bytov, čo sa nezmenilo, avšak zmenila sa veľkosť mojej hlavičky, ktorá sa opíše v čoskoro.

D: Štruktúra hlavičky vlastného protokolu:

Štruktúra hlavičky sa jednoznačne od návrhu zmenila, pričom v návrhu sa skladala z Checksumu(2B), Flagu(1B), Sekvenčného čísla(1B), a DĺžkyDĺžky daného fragmentu(4B). Teda kompletná veľkosť hlavičky by bola 2+1+1+4 = 8 bytov.

Toto sa celkom zmenilo, teda vo finálnej fáze neobsahuje informáciu o dĺžke daného fragmentu, vzhľadom na fakt, že táto informácia sa dá veľmi jednoducho dodatočne vypočítať.
Veľkosť checksumu sa zmenila z 2 na 4 bajty.
Veľkosť sekvenčného čísla sa zmenila na jediný bit.
Veľkosť flagov sa zmenila na 7 bitov. (teda sekvenčné číslo a flag ukladám do jedného bajtu)
Veľkosť hlavičky sa tým pádom zmenila z 8B na:
Checksum(4B) + Seq number(1b) + Flag(7b) = 5B

Teda aj maximálna veľkosť jedného packetu sa zmenila z 1472-8 na 1472-5=1467 B

3. Prehľad fungovania riešenia:

Moje riešenie pozostáva z dvoch hlavných častí, klient a server. Program je obsiahnutý v jednom zdrojovom súbore, pričom sa v ňom nachádzajú obe tieto spomenuté časti.

Na začiatku programu dostane užívateľ na výber, či chce spustiť program ako klienta, alebo server(clientSocket(), server socket()). Prirodzene pre prenos je potrebné zabezpečiť oboje tieto strany, takže treba program zapnúť dvakrát, raz pre server, a raz pre klienta. Po výbere typu uzla je potrebné zadať IP adresu (localhost/127.0.0.1) a port, napríklad 12345.

A) Klient:

Po tomto vstupe sa obe strany zapnú, a čaká sa na input do strany klienta. Na strane klienta zadáme maximálnu veľkosť fragmentu(v bytoch, a reprezentuje prenesené dáta bez hlavičky), pričom ak nepatrí do intervalu <1, maximálna veľkosť packetu>, bude automaticky pridelená blízka korektná hodnota .

Vyberieme si, či chceme posielať správu, alebo súbor. Rozdiel je v tom, že súbor je načítaný z absolútnej cesty(zo vstupu), a uložený na ďalšiu absolútnu cestu(takisto zo vstupu). Správa sa jednoducho načíta zo vstupu, pošle a vypíše na druhom uzle.

Po výbere typu prenosu si vyberieme, či chceme simulovať chybu vo fragmente, ak áno, vyberieme si index(poradie, lenže začína od nuly a nie jednotky) chybného fragmentu. Táto simulácia chyby funguje tak, že v danom fragmente je odobratý posledný byte z dátovej časti. Túto chybu po prijatí fragmentu detekuje server a vypýta si daný packet znova.

Po tomto výbere je čas na špecifikovanie správy/súboru. Pre správu zadáme string so správou, pre súbor zadáme absolútnu cestu sourcu, a destinácie. Retrospektívne je jasné, že nie je príliš inteligentné zadávať destination path na strane klienta, no predsa pre zachovanie autenticity to tu nechám.

Nasledovne je potrebné naše prenášané dáta rozdeliť na fragmenty podľa maximálnej veľkosti fragmentov. Na toto slúži funkcia **splitIntoFragments()**, ktorá danú správu potrebne rozdelí a fragmenty uloží do listu.

```
def splitIntoFragments(maxFragment, data):                    #tato funkcia rozdeli cely subor/spravu do fragmentov podla
    allFragments = []
    if(len(data)>maxFragment):                                #ak bude viac fragmentov
        fragmentCount = int(len(data)/maxFragment)
        if(len(data)%maxFragment!=0):
            fragmentCount+=1
        for i in range(0,fragmentCount):
            if(i == fragmentCount-1):
                fragmentData = data[(maxFragment * (i)):len(data)]
            else:
                fragmentData = data[(maxFragment*(i)):(maxFragment*(i+1))]
            checksum = zlib.crc32(fragmentData).to_bytes(checksumSize, "big")          #pouzitie crc32 checksumu na detekci
            flagAndSeq = setFlagAndSeqNum("SEND", i % 2)
            fragment = bytes("", encoding = "utf-8") + checksum + flagAndSeq + fragmentData        #setnutie fragmentu-> sk
            allFragments.append(fragment)
    else:                                                     #ak bude iba 1 fragment
        checksum = zlib.crc32(data).to_bytes(checksumSize,"big")
        flagAndSeq = setFlagAndSeqNum("SEND", 0)
        fragment = bytes("", encoding = "utf-8") + checksum + flagAndSeq + data               #setnutie fragmentu ako vyss
        allFragments.append(fragment)
```

Fragmenty sa skladajú z hlavičky a dátovej časti.

Hlavička, teda checksum, flag a sequence number je vytvorená pomocou **zlib.crc32() <-** checksum(4B), a flag + sequence number(1B) je vytvorený pomocou funkcie **setFlagAndSeqNum().**

```
def setFlagAndSeqNum(flag, seq):            #jednoducho setnem f
    if (flag == "ACK"):
        flag = ACK
    elif (flag == "ERR"):
        flag = ERR
    elif (flag == "SEND"):
        flag = SEND

    combined = int(flag+seq).to_bytes(seqandflagsize, "big")
    return combined
```

(Na spätné prečítanie týchto informácií slúži jej protiklad, **getFlagAndSeqNum()**, ktorý z daného bajtu vyvodí flag a sequence number.)

Tento list fragmentov vráti funkcia späť klientovi a ten vytvorí inicializačný packet funkciou **createInitializationPacket(),** ktorý je potrebný pre informovanie serveru o našom dátovom prenose. Tento packet obsahuje svoj checksum(4B), veľkosť prenášaných dát(3B), počet fragmentov(3B) svoj originálny flag(1B) a ak je prenášaný súbor namiesto správy, aj destination path.

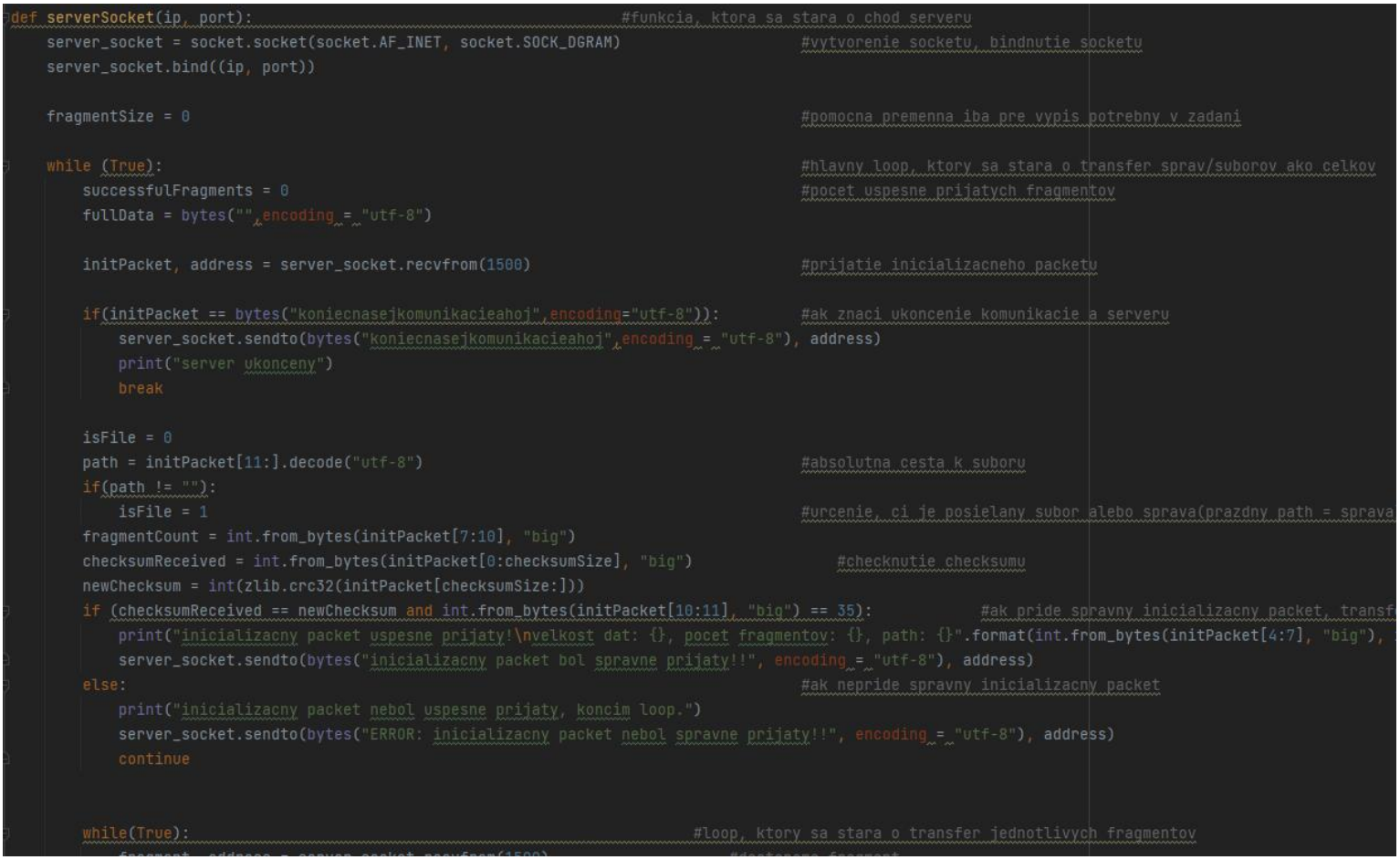
Po úspešnom prijatí tohto packetu naším serverom je klient pripravený na prenos spomínanej správy/súboru. Vzhľadom na flow control Stop and Wait ARQ, posiela vždy jeden fragment, a po prijatí ACK packetu od serveru so správnym sekvenčným číslom, čo signalizuje bezproblémový prenos daného fragmentu a jeho odpovede, môže posielať ďalší. Pri odoslaní chybného fragmentu(detekcia cez checksum) je daný fragment na pokyn serveru odoslaný znova.

Ak by nebol prijatý či už fragment, alebo odpoveď (ACK packet), posielal by sa daný fragment znova.

Po úspešnom odoslaní a prijatí všetkých fragmentov sa pošle posledný packet signalizujúci koniec prenosu, a ak naňho dostaneme korektnú odpoveď, prenos sa ukončí a znova sa dostaneme do časti na začiatku, kde vyberáme medzi správou alebo súborom.

B) Server:

Strana serveru sa riadi pomocou funkcie **serverSocket()**, pričom sa o jeho chod starajú dva vnorené while loopy. V prvom, vyššie postavenom, sa rieši problematika prijatia packetu o posielaní dát, teda inicializačného packetu. Taktiež sa v tejto časti rieši prijatie a odpoveď na ukončovací packet, ktorý ukončí komunikáciu medzi našimi dvomi uzlami, a vráti program do prvotného stavu výberu role(klient/server).



```
def serverSocket(ip, port):                                    #funkcia, ktora sa stara o chod serveru
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)                    #vytvorenie socketu, bindnutie socketu
    server_socket.bind((ip, port))

    fragmentSize = 0                                                                    #pomocna premenna iba pre vypis potrebny v zadani

    while (True):                                                                       #hlavny loop, ktory sa stara o transfer sprav/suborov ako celkov
        successfulFragments = 0                                                         #pocet uspesne prijatych fragmentov
        fullData = bytes("",encoding = "utf-8")

        initPacket, address = server_socket.recvfrom(1500)                              #prijatie inicializacneho packetu

        if(initPacket == bytes("koniecnasejkomunikacieahoj",encoding="utf-8")):         #ak znaci ukoncenie komunikacie a serveru
            server_socket.sendto(bytes("koniecnasejkomunikacieahoj",encoding = "utf-8"), address)
            print("server ukonceny")
            break

        isFile = 0
        path = initPacket[11:].decode("utf-8")                                          #absolutna cesta k suboru
        if(path != ""):
            isFile = 1                                                                  #urcenie, ci je posielany subor alebo sprava(prazdny path = sprava
        fragmentCount = int.from_bytes(initPacket[7:10], "big")
        checksumReceived = int.from_bytes(initPacket[0:checksumSize], "big")                #checknutie checksumu
        newChecksum = int(zlib.crc32(initPacket[checksumSize:]))
        if (checksumReceived == newChecksum and int.from_bytes(initPacket[10:11], "big") == 35):            #ak pride spravny inicializacny packet, transf
            print("inicializacny packet uspesne prijaty!\nvelkost dat: {}, pocet fragmentov: {}, path: {}".format(int.from_bytes(initPacket[4:7], "big"),
            server_socket.sendto(bytes("inicializacny packet bol spravne prijaty!!", encoding = "utf-8"), address)
        else:                                                                           #ak nepride spravny inicializacny packet
            print("inicializacny packet nebol uspesne prijaty, koncim loop.")
            server_socket.sendto(bytes("ERROR: inicializacny packet nebol spravne prijaty!!", encoding = "utf-8"), address)
            continue


        while(True):                                                        #loop, ktory sa stara o transfer jednotlivych fragmentov
            fragment, address = server_socket.recvfrom(1500)                #dostanem fragment
```

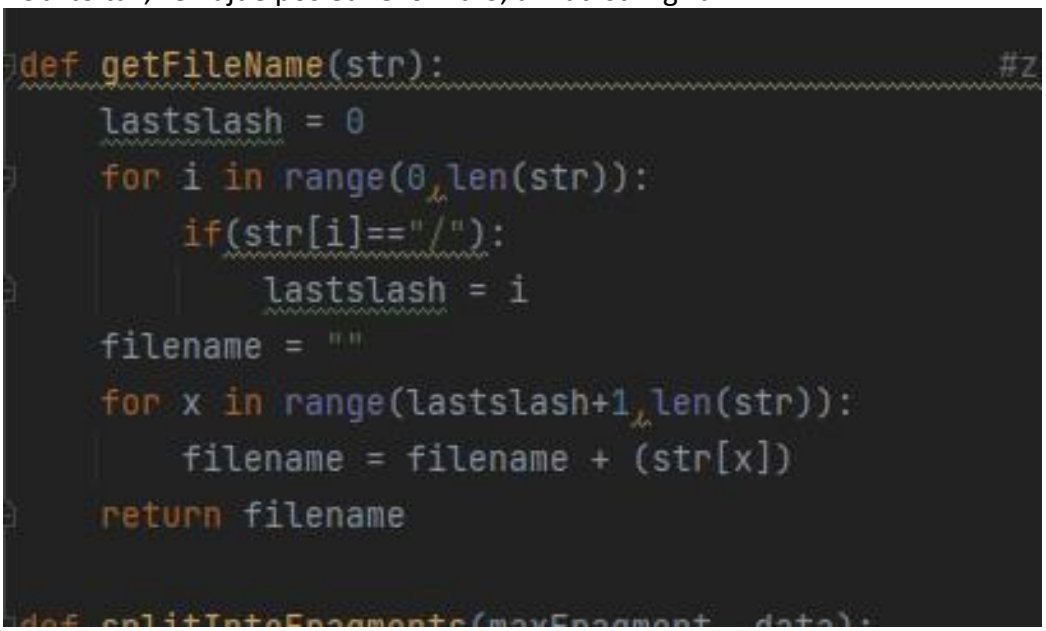
V rámci druhého(vnoreného) while loopu sa rieši problematika prijímania fragmentov počas prenosu, a odosielania odpovedí s flagmi a sekvenčnými číslami. Pri úspešnom prijatí nepoškodeného fragmentu sa odošle packet s flagom ACK, a sekvenčným číslom zodpovedajúcim ďalšiemu očakávanému packetu. Toto sekvenčné číslo je vzhľadom na implementovanú ARQ metódu Stop and Wait vždy iba 0 alebo 1, keďže sa neodosiela ďalší fragment v poradí bez prijatia odozvy s korektným flagom(ACK) a sekvenčným číslom. Ak server dostane poškodené dáta, teda nesedí checksum, odošle odpoveď s flagom “ERR”, pričom si vypýta znova rovnaký fragment cez rovnaké sekvenčné číslo. Ak dostane fragment, ktorý už má(toto sa môže stať pri výskyte oneskorenia/straty odoslaného ACKu), odosiela klasicky flag ACK so sekvenčným číslom nasledujúceho fragmentu, no daný prijatý fragment prirodzene zahodí, keďže ho už má.

Klient po úspešnom odoslaní všetkých fragmentov(a aj prečítaní správnej odpovede) odošle packet, ktorý signalizuje koniec prenosu tejto správy/súboru. Po jeho prečítaní serverom, a prečítaním odpovede serveru naším klientom, sa obe strany vrátia do svojho predošlého štádia, teda čakania na inicializáciu nového prenosu a odoslanie inicializačného packetu.

C) Ostatné pomocné funkcie:

Okrem vyššie spomenutých **splitIntoFragments()** a funkcií na prácu s flagmi a sekvenčnými číslami - **setFlagAndSeqNum() & getFlagAndSeqNum()** sa používajú aj funkcie**:**

getFileName(), ktorá zo stringu s absolútnou cestou k súboru získa jeho meno. Robí to tak, že nájde posledné lomítko, a vráti string za ním.



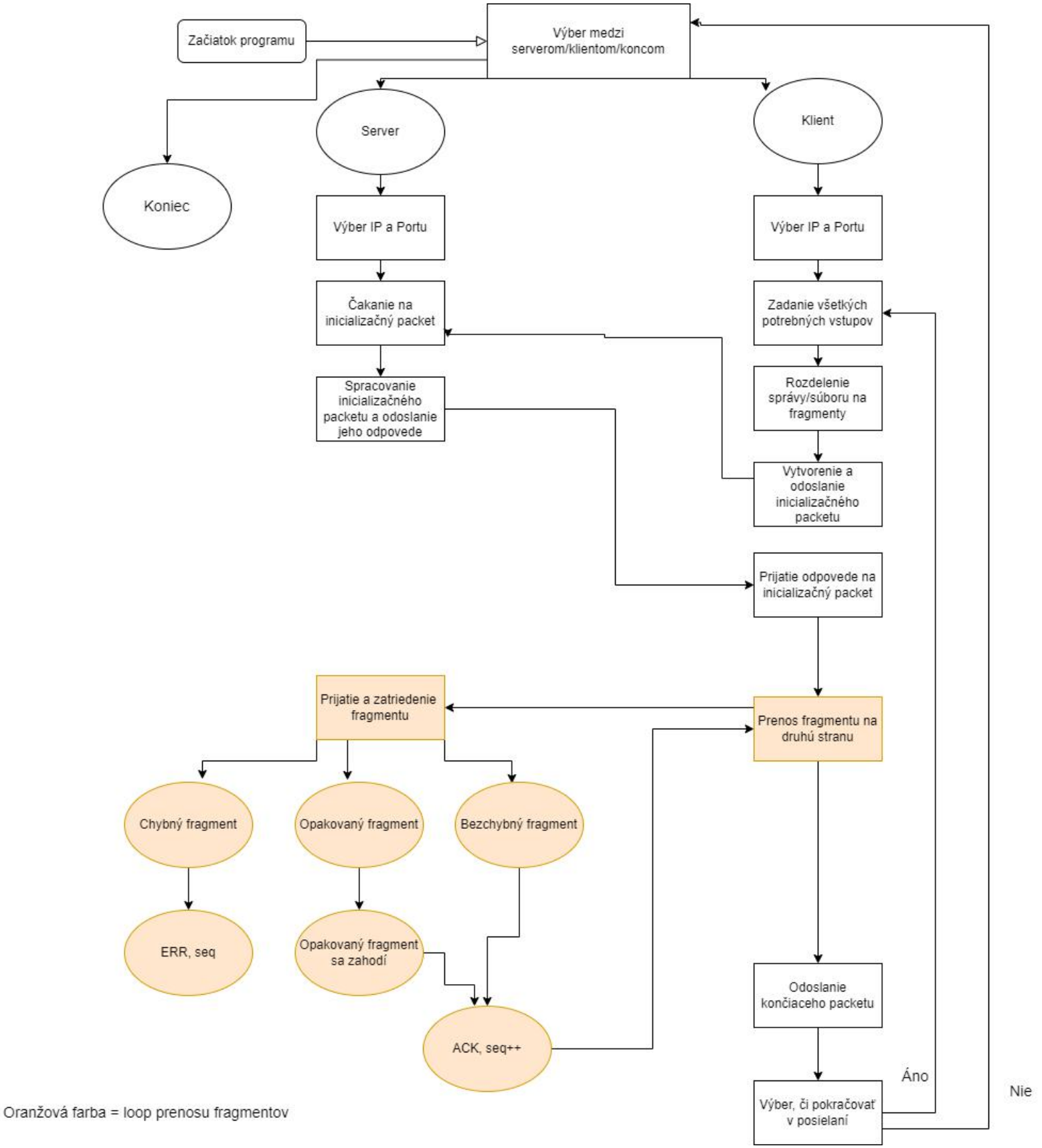
```
def getFileName(str):                                    #z
    lastslash = 0
    for i in range(0,len(str)):
        if(str[i]=="/"):
            lastslash = i
    filename = ""
    for x in range(lastslash+1,len(str)):
        filename = filename + (str[x])
    return filename
```

createInitializationPacket() je funkcia, ktorá jednoduchým spôsobom vytvára inicializačný packet, pričom sa skladá z nasledovných údajov:

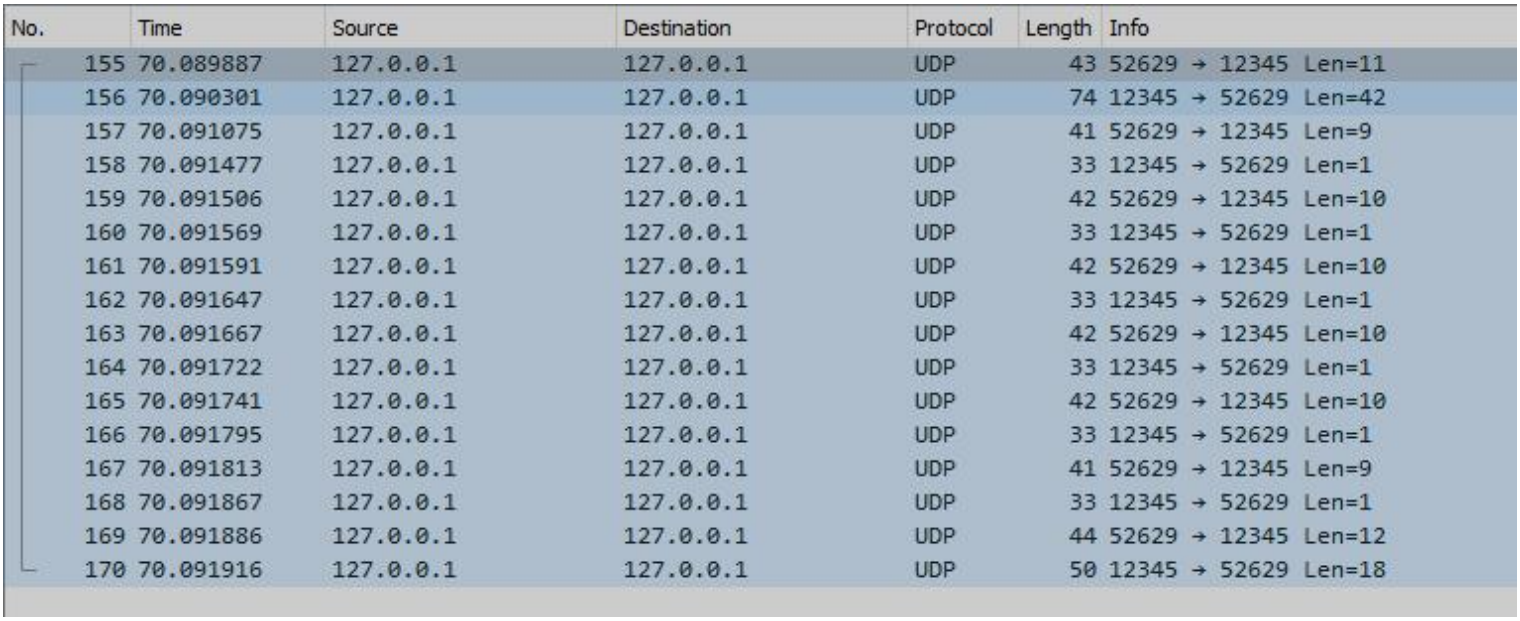
```
# initPacket == 4B checksum, 3B dataSize, 3B fragmentCount, 1B flag, ???B pathDest
```

pathDest(absolútna cesta k destinácii súboru) sa appenduje na koniec, iba v prípade, že je prenášaný subjekt súbor.

4. Diagram fungovania programu:



5. Príklad prenosu vo Wiresharku:



No.	Time	Source	Destination	Protocol	Length	Info
155	70.089887	127.0.0.1	127.0.0.1	UDP	43	52629 → 12345 Len=11
156	70.090301	127.0.0.1	127.0.0.1	UDP	74	12345 → 52629 Len=42
157	70.091075	127.0.0.1	127.0.0.1	UDP	41	52629 → 12345 Len=9
158	70.091477	127.0.0.1	127.0.0.1	UDP	33	12345 → 52629 Len=1
159	70.091506	127.0.0.1	127.0.0.1	UDP	42	52629 → 12345 Len=10
160	70.091569	127.0.0.1	127.0.0.1	UDP	33	12345 → 52629 Len=1
161	70.091591	127.0.0.1	127.0.0.1	UDP	42	52629 → 12345 Len=10
162	70.091647	127.0.0.1	127.0.0.1	UDP	33	12345 → 52629 Len=1
163	70.091667	127.0.0.1	127.0.0.1	UDP	42	52629 → 12345 Len=10
164	70.091722	127.0.0.1	127.0.0.1	UDP	33	12345 → 52629 Len=1
165	70.091741	127.0.0.1	127.0.0.1	UDP	42	52629 → 12345 Len=10
166	70.091795	127.0.0.1	127.0.0.1	UDP	33	12345 → 52629 Len=1
167	70.091813	127.0.0.1	127.0.0.1	UDP	41	52629 → 12345 Len=9
168	70.091867	127.0.0.1	127.0.0.1	UDP	33	12345 → 52629 Len=1
169	70.091886	127.0.0.1	127.0.0.1	UDP	44	52629 → 12345 Len=12
170	70.091916	127.0.0.1	127.0.0.1	UDP	50	12345 → 52629 Len=18

Správa: ahojahojahojahojahojahoj (6x”ahoj”)(DĺžkA = 24B)
Nastavená maximálna veľkosť fragmentu: 5B, teda správa sa rozdelí do 5 fragmentov, pričom posledný má iba 4 znaky zo správy
Headersize = 5B
Teda každý fragment prenesie dáta o dĺžke 10 bytov. Chyba je simulovaná v prvom fragmente(odobratie posledného bytu z dátovej časti(10B->9B))

Analýza:
155. Packet = inicializačný packet
156. Packet = jeho odpoveď
157. Packet = prvý fragment(vrátane spomenutej vnesenej chyby)
158. Packet = odpoveď na chybný fragment, vyžiadanie rovnakého
159. Packet = prvý fragment(tentokrát bez chyby)
160. Packet = odpoveď na fragment bez chyby(ACK, sekvenčné číslo ďalšieho fragmentu)
161. - 168. Packet = opakovanie predošlých dvoch, len pre nasledujúce fragmenty(poslednému ostali iba 4 znaky z odosielanej správy 10->9B)
169. Packet = packet o ukončení prenosu správy
170. Packet = jeho odpoveď

6. Záver

Toto zadanie som riešil v jazyku Python, v IDE PyCharm.
Medzi použité knižnice patrí knižnica socket, ktorá je spomenutá v samotnom zadaní, a knižnica zlib, ktorej použitie je spomenuté v časti 2.
Program umožnuje používateľovi vytvoriť server a klient, medzi ktorými je za korektného nastavenia parametrov možný prenos dát, teda správy alebo súboru.
Používateľ má možnosť vybrať mnoho rôznych vstupov a parametrov, ktoré ovplyvňujú chod programu, napríklad maximálnu veľkosť fragmentu alebo simuláciu chyby v prenose fragmentov.
Program je schopný prenášať súbory veľkosti aj 2MB.