Zadanie 2
Komunikácia s využitím UDP protokolu

Meno: Tomáš Rafaj
Cvičenie: Štvrtok 8:00

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 10-60s pokiaľ používateľ neukončí spojenie. Odporúčame riešiť cez vlastne definované signalizačné správy

Návrh - zmeny

Hlavička - zmena poradia CRC pred DATA, z dôvodu jednoduchšie packovania

TYP	POCET	PORADOVE CISLO	VELKOST	CRC, CHECKSUM	DATA

Typ hlavičky (1B)
Využil som nakoniec char.
Hypoteticky som nič extra oproti pôvodnému nezmenil a mohol som takto využiť až 255 rôznych signálov.

Označenie

TYPE a - slúži na 3 way-handshake
TYPE b - slúži na 3 way-handshake
TYPE c - slúži na odosielanie/prijímanie súboru
TYPE r - slúži na odosielanie/prijímanie opravených fragmentov súboru
TYPE F - slúži na ukončenie súboru a prijatie súborého názvu
TYPE E - slúži na odchod z program (EXIT)
TYPE s - slúži na odosielanie/prijímanie správy
TYPE g - slúži na odosielanie/prijímanie opravených fragmentov správy
TYPE G - slúži na ukončenie a vypísanie správy
TYPE X - slúži na KEEP-ALIVE

Tieto body ostali nezmenené.
Počet (2B)
Počet fragmentov, je max teda 2 na 16tu (2B).

Poradové číslo (2B)
Hlavička taktiež obsahuje poradové číslo fragmentu aby sa to neskôr spracovalo v správnom poradí. Jej najväčší prípad je max počet, t.j. 2 na 16.

Veľkosť (4B)
Najväčší fragment môže mať najviac 1500B - veľkosť hlavičky, to je z dôvodu limitu linkovej vrstvy.
Najmenší fragment by mal byť 1B + veľkosť hlavičky.

CRC | CHECKSUM (4B)
CRC kontrola slúži na zahashovanie, uloží sa na koniec správy, kde sa uloží. V prípade, že nesedí správa sa odošle znova zo zdroja, odkiaľ sa posielala.

Chybovanie
Vynútená chyba vlastne vyplýva z toho, urobíme to, že zmeníme fragment-ový byte už po uložení CRC hodnoty a potom pri checkovaní CRC to zistí.

Implementačná časť návrhu

Čo sa implementácie týka tak tam boli viaceré zmeny, a preto nebudem písať jednotlivé a presné rozdiely. Napíšem sem finálnu implementáciu a opis riešenia. Pokúsim sa zvýrazniť veci, ktoré sa zmenili.

Implementácia

Implementačne to funguje celkom priamočiaro, sú 2 možnosti, buď sa jedná o klienta alebo o server. Vieme si nastaviť server PORT, a o oboch si vieme nastaviť komunikačnú IP. Klientov port je fixne 1235. Posielam po balíčkom 5 / 10 alebo 20. Iné som netestoval, veľkosť tohto balíku nie je možné z implementačných dôvodov meniť a to najmä z dôvodu obmedzenia využitia premenných alebo spôsobených delay-ov serveru. V prípade, že klient nepošle alebo pošle chybné fragmenty, server si všetky tieto fragmenty znovu vyžiada, pričom počíta s tým, že už nebudú vynechané ani pokazené. V prípade, že príde finálna správa / resp. finálny fragment tak server vypíše v prípade súboru jeho veľkosť a absolútnu cestu kam bol vložený. V prípade správy vypíše správu do konzole. Pri každej správe, preposielaní, finálnej správe je kontrola CRC pomocou CRC-16. Čo sa týka implementácií tak súborová aj správová sú veľmi podobné a je tam malá časť zmien, prakticky týkajúcich sa najmä výpisu a pri správe je .decode a encode charov. Veľkosť fragmentov je možná od 1 do 1452. Hlavička má vždy veľkosť 20. Po prijatí prvého súboru alebo správy púšťame keep-alive na inom threade, pričom každý keep-alive beží 120 sekúnd a timeout je nastavený na 45 sekúnd, čo nám v ideálnom prípade dáva až 165 sekúnd na zadanie ďalšieho kroku, v prípade 45 sekundovej neaktivity sa server odpojí. Klient sa nezatvára ani v prípade neúspešného keep-alive voči serveru a je potrebné ho vypnúť pomocou zadania klávesy 2. Užívateľské rozhranie cez konzolu ešte opíšem v inom bode.

Server prijíma správy vo while 1 cykle až do ukončovacej správy / chybného keep-alive.

```
while 1:
    try:
        socket_server.settimeout(45)
        data, address = socket_server.recvfrom(1472)
        data_local = struct.unpack('ciiii', data[:20])
        CRC_local = crc16_func(data[:16] + data[20:])
        if data_local[0] == b'X':
```

Spracováva jednotlivé typy hlavičiek, ktoré boli uvedené v opravenom návrhu.

Klient zapína jednotlivé časti programu na základe vstupu

```
while 1:
    if what_to_do == 0:
        message_lcl = struct.pack('ciiii', b'X', 0, 0, 0, CRC)
        t1 = threading.Thread(target=keep_alive,
                              args=(socket_client_IP, socket_client_PORT, message_lcl, socket_client))
        t1.start()
    what_to_do = input('Please, choose an option -> 1 - Send file | 2 - Quit (return 0) | 3 - Send text\n')
```

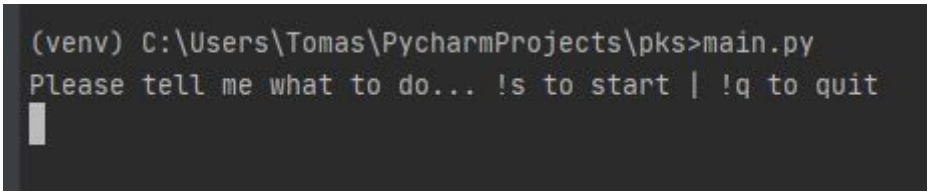
Kde 1 nás prepína na výber súboru, 2 návrat do menu a 3 na zadanie textu.

Zhrnutie hlavných zmien

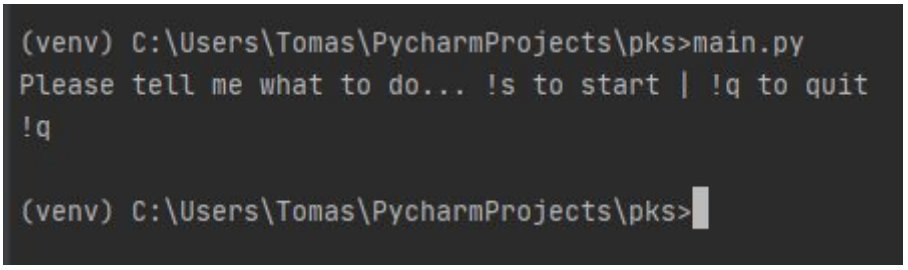
- Hlavička
- Čiastočné zmenenie typov správ
- Neukončuje program pri ACK, keep-alive tiež nevypíname
- Iné fungovanie keep-alive
    - keep alive nefunguje na TRUE / FALSE systéme ale na timeoute, každý keep-alive signál má platnosť 120 sekúnd

Užívateľské rozhranie

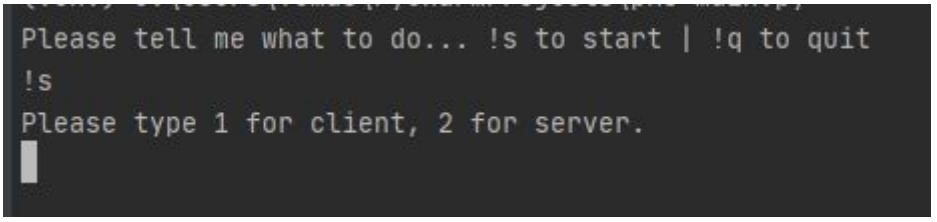
Po spustení programu príde request na štart alebo vypnutie programu.



V prípade zadania !q sa program vypína

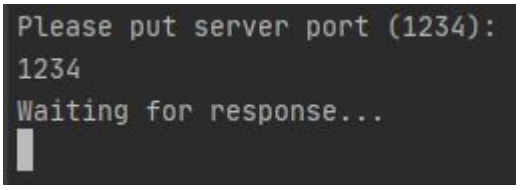


V prípade zadania !s sa program dostáva na výber server / klient



Server zapíname vždy prvý, z dôvodu že klient sa po zadaní údajov ihned pokúsi pripojiť, čiže server už musí bežať na rovnakej IP a aj porte.

V prípade zadania možnosti server si vypýta jeho port, kde bude čakať / počúvať na 3 way handshake



V prípade, že zapneme klienta, zadáme jeho vstupné dáta tak klient vyzerá takto.

```
(venv) C:\Users\Tomas\PycharmProjects\pks>main.py
Please tell me what to do... !s to start | !q to quit
!s
Please type 1 for client, 2 for server.
1
Client
Put server / communication IP address (127.0.0.1)
127.0.0.1
Put server port to send data to (1234)
1234
Server successfully responded.
Please, choose an option -> 1 - Send file | 2 - Quit (return 0) | 3 - Send text
```

Teraz už klient čaká na zadanie vstupu, avšak nie donekonečna a je tam timeout na 45 sekúnd. Ak klient nič nepovie, odpojí sa server a otvorí menu.

```
(venv) C:\Users\Tomas\PycharmProjects\pks>main.py
Please tell me what to do... !s to start | !q to quit
!s
Please type 1 for client, 2 for server.
2
Server
Please put server port (1234):
1234
Waiting for response...
Client connected.
```

```
Please put server port (1234):
1234
Waiting for response...
Client connected.
No response.
Exiting...
Please tell me what to do... !s to start | !q to quit
```

Teraz treba klienta samozrejme ukončiť zadaním quit, pretože to nestihol. A môžeme skúsiť znova.

```
Server successfully responded.
Please, choose an option -> 1 - Send file | 2 - Quit (return 0) | 3 - Send text
2
Please tell me what to do... !s to start | !q to quit
```

V prípade napríklad odosielania dát na server vyzerá užívateľské rozhranie takto

```
(venv) C:\Users\Tomas\PycharmProjects\pks>main.py
Please tell me what to do... !s to start | !q to quit
!s
Please type 1 for client, 2 for server.
1
Client
Put server / communication IP address (127.0.0.1)
127.0.0.1
Put server port to send data to (1234)
1234
Server successfully responded.
Please, choose an option -> 1 - Send file | 2 - Quit (return 0) | 3 - Send text
1
Please put file location
VPN.exe
C:\Users\Tomas\PycharmProjects\pks\VPN.exe
Please put size of 1 fragmental (possible -> 1-1452) In case of other input, program crashes.
1452
Total size of fragments in bytes :
20095616
Total amount of fragments :
13840
On purpose error simulation, please put number from 1 to 13840
0
Sending file... please wait.
Successfully sent.

Please, choose an option -> 1 - Send file | 2 - Quit (return 0) | 3 - Send text
```

Serverová strana vyzerá takto, kde zobrazuje ako idú fragmenty, indexuje ich od nuly. Na konci vypíše úplnú cestu, veľkosť v byte-och a aj počet fragmentov.

```
Received packet number 13822 with size of 1452 bytes.
Received packet number 13823 with size of 1452 bytes.
Received packet number 13824 with size of 1452 bytes.
Received packet number 13825 with size of 1452 bytes.
Received packet number 13826 with size of 1452 bytes.
Received packet number 13827 with size of 1452 bytes.
Received packet number 13828 with size of 1452 bytes.
Received packet number 13829 with size of 1452 bytes.
Received packet number 13830 with size of 1452 bytes.
Received packet number 13831 with size of 1452 bytes.
Received packet number 13832 with size of 1452 bytes.
Received packet number 13833 with size of 1452 bytes.
Received packet number 13834 with size of 1452 bytes.
Received packet number 13835 with size of 1452 bytes.
Received packet number 13836 with size of 1452 bytes.
Received packet number 13837 with size of 1452 bytes.
Received packet number 13838 with size of 1452 bytes.
Received packet number 13839 with size of 1388 bytes.
End of packet.
Corrupted or not received files: (n-1)th in package
[]
I am sending request to get them again.
-------------------
Received last packet containing name of the file....
VPN.exe
Saving as VPN.exe with size of 20095616 bytes in 13840 fragments.
C:\Users\Tomas\PycharmProjects\pks\files\VPN.exe
Waiting for response...
```

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

Je implementovaný v jazyku python.

2. Program musí pracovať s dátami optimálne (napr. neukladať IP adresy do 4x int).

Neukladám IP do int a iné veci, ukladám to do stringu / array-ov, python toto rieši prakticky automaticky.

3. Pri posielaní súboru musí používateľovi umožniť určiť cieľovú IP a port.

Toto umožňuje pri spustení používateľského rozhrania a výberu 1 (klient)

4. Používateľ musí mať možnosť zvoliť si max. veľkosť fragmentu.

Toto umožňuje.

5. Obe komunikujúce strany musia byť schopné zobrazovať: a. názov a absolútnu cestu k súboru na danom uzle, b. veľkosť a počet fragmentov.

Obe zobrazujú absolútnu cestu, veľkosť v byte-och (to je v podstate veľkosť fragmentov) a aj počet fragmentov.

6. Možnosť simulovať chybu prenosu odoslaním minimálne 1 chybného fragmentu pri prenose súboru (do fragmentu je cielene vnesená chyba, to znamená, že prijímajúca strana deteguje chybu pri prenose).

Je možné simulovať zadaním chybného indexu, kam sa vloží chyba. V prípade zadania 0 sa chyba nesimuluje.

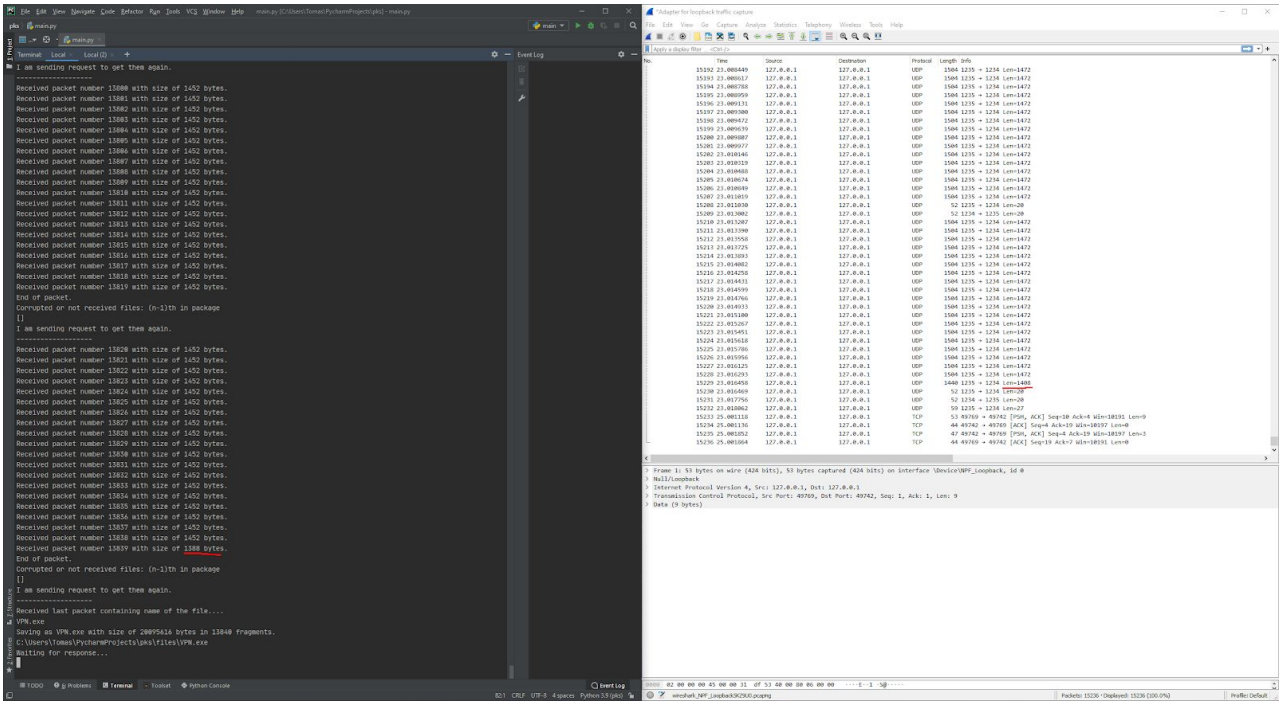
7. Prijímajúca strana musí byť schopná oznámiť odosielateľovi správne aj nesprávne doručenie fragmentov.

Funguje, pomocou vyžiadania znovuposlania chybných / nedoručených fragmentov.

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.

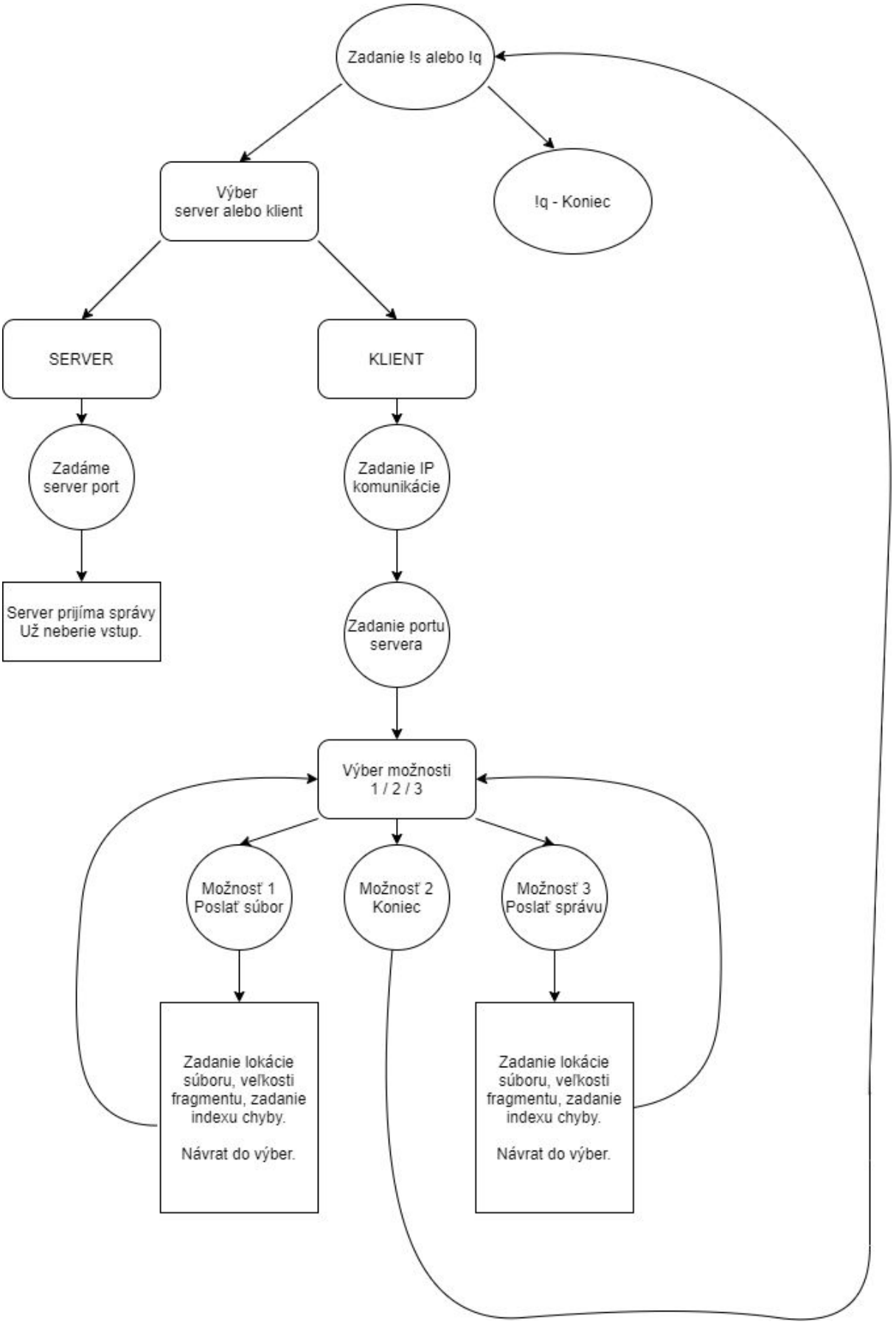
Funguje mi aj súbor 19MB, teda minimálne (ten som testoval). Cesta je zadaná v zdrojovom kóde, je možné ju zmeniť.

Program sa zobrazuje vo wiresharku



1504 je preto lebo loopback môže pridať až do 1526, kde loopback tam aktuálne pridáva 4.

Diagram fungovania



Vysvetlenie fungovania spracovania súboru

Vysvetlím hlavný algoritmus spracovania, tento algoritmus je prakticky rovnaký u správy.

```
elif data_local[0] == b'c':
    socket_server.settimeout(None)
    if first_packet == 1:
        first_packet = 0
        temp_array = []
        for ff in range(data_local[1]):
            temp_array.append(ff)
    TYPE = b'c'
    if data_local[2] != 0:
        if data_local[4] != CRC_local:
            print('Incorrect packet number ' + str(curr_index + data_local[2] - 1))
            chybne_fragmenty.append(data_local[2] - 1)
        else:
            cntr = cntr + 1
            tmp = data_local[2] - 1
            if tmp in temp_array:
                #print('Array ' + str(temp_array))
                whole_array.insert(curr_index + data_local[2] - 1, data[20:])
                print('Received packet number ' + str(
                    curr_index + data_local[2] - 1) + ' with size of ' + str(data_local[3]) + ' bytes.')
                file_size = file_size + data_local[3]
                temp_array[data_local[2] - 1] = -1
    if data_local[2] == 0 and data_local[4] == CRC_local:
        print('End of packet.')
        first_packet = 1
        for kk in range(len(temp_array)):
            if temp_array[kk] != -1:
                if len(chybne_fragmenty) != 0:
                    for zz in range(len(chybne_fragmenty)):
                        if chybne_fragmenty[zz] == temp_array[kk]:
                            break
                        elif zz == len(chybne_fragmenty) - 1:
                            chybne_fragmenty.append(temp_array[kk])
                else:
                    chybne_fragmenty.append(temp_array[kk])

        curr_index = curr_index + data_local[1]
        redukovane_chybne = []

        if len(chybne_fragmenty) != 0:
            for tt in range(len(chybne_fragmenty)):
                if chybne_fragmenty[tt] != -1:
                    redukovane_chybne.append(chybne_fragmenty[tt])

        print('Corrupted or not received files: (n-1)th in package')
        print(redukovane_chybne)
        print('I am sending request to get them again.')
        print('-------------------')
        socket_server.sendto(
            struct.pack('ciiii', TYPE, POCET_FRAGMENTOV, PORADOVE_CISL0, VELKOST, CRC) +
            bytearray(redukovane_chybne), (socket_server_IP, socket_client_PORT2))
        temp_array = []
```

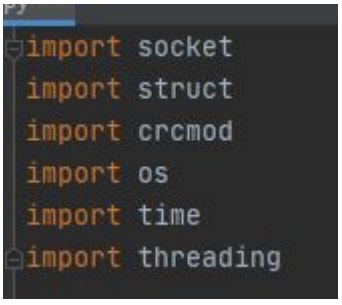
Ružová časť kontroluje CRC, v prípade, že je zlé / nesedí tak ide o chybný packet, nezapíše sa a dodá sa do chybných.

Zelenožltá časť ukladá fragmenty na ich pridelené miesto, pričom ich vymažame z temp_array-u aby sa vyhlo duplikátom.

Červená časť spojí chybné fragmenty (zlé CRC) a fragmenty, čo neprišli. Následne redukciou odstránime indexy označené ‘-1’ a pošleme si všetky v array-i všetky fragmenty, ktoré server potrebuje aby klient preposlal.

Zhrnutie
Program je plne funkčný, spĺňa všetky minimálne podmienky programu. Obsahuje všetky bodovo hodnotené odrážky ako je posielanie menšieho ako je fragment, poslanie viac ako 2MB a iné...

Program využíva viaceré knižnice



socket pre samotný beh programu
struct pre packovanie hlavičky
crcmod na výpočet CRC-16
os na absolútnu cestu súboru
time na sleeping / delays
threading na multithreading, ktorý bol potrebný pri keep-alive