SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta informatiky a informačných technológií

Ilkovičova 2, 842 16 Bratislava 4

ZADANIE 2 – UDP KOMUNIKÁTOR

Meno a priezvisko: Marko Stahovec

Dátum vypracovania: 17.11.2021

Cvičiaci: Ing. Kristián Košťál, PhD.

1. Zadanie

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.

Program musí byť organizovaný tak, aby oba komunikujúce uzly mohli **prepínať medzi funkciou vysielača a prijímača** bez reštartu programu - program nemusí (ale môže) byť vysielač a prijímač súčasne. Pri predvedení riešenia je podmienkou hodnotenia schopnosť doimplementovať jednoduchú funkcionalitu na cvičení.

2. Zmeny oproti návrhu

2.1 Dôležité zmeny a inovácie

- 1. Nový typ správy - **SWAP**

  Tento typ správy bol potrebný z dôvodu **signalizácie zmeny rolí** na oboch stranách. Primárne šlo o zjednodušenie zmeny role v prípade, ak ju inicializoval server.

- 2. **Restart flag** pri neobdržaní **DPING**-u

  Ak klient nepríjme odpoveď na vlastný DPING zo strany serveru, **pokúsi sa poslať DPING ešte raz** pred ukončením posielania tohto typu správ. Tak sa zabezpečí perzistentnosť pri neočakávanom stratení správy v ojedinelých prípadoch.

- 3. **Restart flag** pri posielaní dát

  V prípade, ak klient neobdrží v istom čase správu DATTC alebo DATTE, **pokúsi sa poslať správu ešte raz**, ak je restart_flag stále True. Premenná restart_flag sa resetuje pri každej korektne doručenej správe.

- 4. **Klient počúva a server posiela**

  Klientovi bola pridaná funkcionalita na báze dvoch nových vlákien, kde jedno z nich **počúva na správy** zo strany serveru a druhé **prijíma vstup od klienta** a vracia ho do hlavného vlákna. Vďake prvému menovanému vláknu je klient schopný prijímať správy od servera a konať podľa týchto správ, napr. zmeniť rolu ak si to server vyžiadal alebo ukončiť spojenie, ak bol server vypnutý. To implikuje fakt, že aj server vie odteraz inicializovať isté typy akcií na tomto spojení.

2.2 Malé zmeny a detaily

- 1. **Nová knižnica concurrent**, konkrétne modul futures

  Modul futures bol použitý pri vstupe z menu klienta, aby nebol situovaný v hlavnom vlákne, ale vedľajšom. Táto knižnica bola použitá z dôvodu jednoduchej práce s návratovou hodnotou z vlákna.

- 2. **Nové konštanty**

  Konštanty ohľadom časových **timeoutov**, **maximálnych veľkostí správ** apod. na jednoduchšiu refaktorizáciu a jednoduchšiu zmenu číselných údajov na rôznych miestach v kóde. Vďaka takémuto prístupu sa eliminuje väčšina náhodných čísel v kóde, ktoré majú odteraz deskriptívne pomenovanie.

- 3. **Čakanie servera pri výpadku klienta**

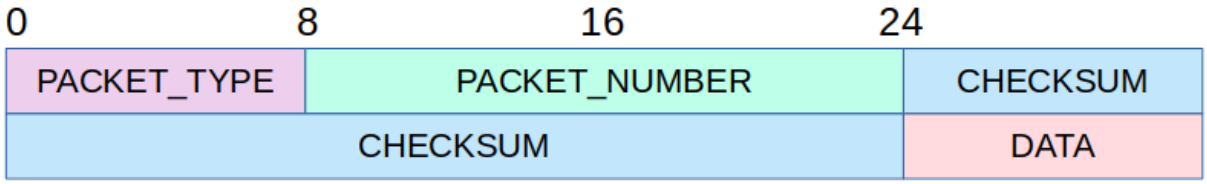
  Novým vylepšením pri prijímaní dát je čakania zo strany servera v prípade, ak klient prestane posielať správy, resp. sa jeho správy začnú strácať. Okrem timeoutu bola zavedená konštanta IDLE_MESSAGE_THRESHOLD, ktorá predstavuje, koľkokrát má server čakať v timeoute pri nedoručení správy.

3. Návrh riešenia a protokolu

Toto riešenie je postavené na nasledovných prostriedkoch:

- Operačný systém: **Linux**, distro Ubuntu
- Programovací jazyk: **Python**, verzia 3.8.10
- Programovacie prostredie: **Pycharm**
- Používateľské rozhranie: **konzola/terminál**

Môj protokol je postavený na nasledovnej **hlavičke**:



- **Packet_type** (**1B**): označuje **typ správy**, ktorá je vysielaná jedným z uzlov. Toto pole hlavičky bude vo svojej podstate reprezentované a prenášané ako typ **unsigned char**. Táto skutočnosť je dôležitá preto, lebo táto implementácia využíva knižnicu struct a v nej metódu pack, ktorou je docielená konverzia na konkrétny dátový typ.

Rôzne typy správ (packet_types) sú zobrazené v tabuľke nižšie:

PACKET_TYPE (dekad. číslo)	PACKET_TYPE (akronym)	VÝZNAM
0	INIT	Inicializuje pripojenie
1	FINIT	Inicializuje prenos súboru, prvý posiela klient serveru a server odpovedá rovnako
2	MINIT	Inicializuje prenos správy, prvý posiela klient serveru a server odpovedá rovnako
3	DATT	Signalizuje dátový datagram, posiela klient serveru
4	DATTC	Signalizuje úspešné prenesenie dát, posiela server klientovi, ide o odpoveď na DATT
5	DATTE	Signalizuje chybu pri rozbalení dát, posiela server klientovi, ide o odpoveď na DATT
6	DPING	Imituje funkcionalitu pingu, udržiava spojenie medzi klientom a serverom
7	CTERM	Signalizuje ukončenie spojenia jednej strany
8	SWAP	Signalizuje výmenu rolí medzi dvojicou klient a server

- **Packet_number** (**2B**): číslo správy, ktoré reprezentuje **poradie fragmentu**, ktoré prichádza prijímacej strane. V prípadoch správ, ktoré sú skôr informatívne ako napr. pri type správy DPING je táto hodnota nastavená na 0 a nijako neovplyvňuje význam samotnej správy. Táto hodnota sa sieťou posiela ako unsigned short, z čoho vyplýva, že najväčší možný počet správ odosielateľný po sieti je 65535.

- **Checksum** (**4B**): predstavuje **zvyšok po výpočte CRC**. Slúži ako kontrola dát, či boli na serveri prijaté v korektnej podobe. Popis metódy, ktorú som si zvolil, sa nachádza v časti 4 nižšie.

- **Data** (**nB**, **n < 1465**): samotné **dáta**, ktoré sú posielané medzi pripojenými zariadeniami. Ich **maximálna veľkosť** môže byť **1465B**. Číslo 1465 vzniklo z odčítania MTU (Maximum transmission unit) pre Ethernet II a odčítaní hlavičiek protokolov z nižších vrstiev nasledovne:
    - 1500 -> Ethernet II MTU (už bez hlavičky a FCS z konca rámca)
    - 1500 - 20 = 1480 -> odčítanie IP hlavičky na sieťovej vrstve
    - 1480 - 8 = 1472 -> odčítanie UDP hlavičky na transportnej vrstve
    - 1472 - 7 = 1465 -> odčítanie mojej hlavičky

4. Checksum

Na kontrolu správnosti dát prostredníctvom checksum-u som si zvolil algoritmus **crc32**. Najprv som si chcel vytvoriť vlastný na báze binárnych operácií XOR a AND a bit shiftov, no tento algoritmus nebol dostatočne rýchly a neprodukoval dostatočne "náhodné" hodnoty. Preto som si naprogramoval crc32 s pomocou zdroja [1], ktorý **replikuje knižničnú podobu zlib.crc32()**. Vďaka tomu som mohol tomu algoritmu lepšie pochopiť a uvidieť, ako taký kód vyzerá, keďže princíp mi je jasný.

4.1 Dôležité poznatky

CRC (Cyclic redundancy check) je **metóda na detekciu chýb** pri prenose dát, ktorá je založená na **zvyšku z polynomiálneho delenia**.

CRC32, resp. 4-bajtové CRC, používa na delenie tento konkrétny polynóm:

$$\mathbf{x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1}$$

Môj algoritmus taktiež využíva CRC tabuľku, ktorá obsahuje predom vypočítané hodnoty pre všetky možné bajty, vďaka čomu pri samotnom výpočte checksum-u nemusí za každým razom rátať CRC pre samostatný bajt znova a znova. Vďaka tomu môže výpočet cykliť priamo po bajtoch a nie po bitoch, čo je vo svojej podstate 8-krát rýchlejšie.

Princíp algoritmu je vysvetlený na príklade nižšie. Ešte predtým je potrebné povedať, že tento algoritmus ráta CRC metódou **LSB** (Lowest-significant bit), kvôli čomu sa binárna podoba vyššie spomenutého polynómu musí **reverznúť**. To je potrebné z dôvodu, aby sa bit-shifty mohli používať priamo a nebola potrebná operácia navyše.

4.2 Opis algoritmu

Opis algoritmu a jeho postupovanie v kóde je zobrazený nižšie aj so všetkými potrebnými komentármi postupovania. Pozostáva z dvoch funkcií:

- **create_crc_lookup_table**() -> vytvorí CRC tabuľku, podľa ktorej môže robiť výpočty po bajtoch a nie po bitoch.
- **mycrc32**(data) -> samotná výpočtová funkcia, ktorá vracia finálny checksum a utilizuje CRC tabuľku, ktorá sa vytvorí iba raz pri rátaní prvého checksumu.

```
# this table holds pre-generated bit-shifted/XOR-ed CRC values,
# so algorithm can cycle through bytes using this table
crc_table = []


def create_crc_lookup_table():
    a = []
    for i in range(256):  # cycle through all possible byte-values
        k = i  # assign current byte-value to a variable
        for j in range(8):  # work through all the bits
            if k & 1:  # if first bit is 1...
                k ^= 0x1db710640  # do XOR division with generator polynomial
            # bit-shift first bit in order to either get of the first
            # one generated by division on line 12, or simply discard
            # first bit since it was previously zero
            k = k >> 1
        a.append(k)  # append the result value to the
    return a


def mycrc32(data):
    global crc_table
    if not crc_table:
        crc_table = create_crc_lookup_table()
    crc = 0xffffffff  # initial state of crc
    for byte in data:  # cycle through all the bytes of given data
        # access the pre-generated value from the crc table
        # using current crc state and current byte value and XOR
        # afterwards
        crc = crc_table[(crc & 0xff) ^ byte] ^ (crc >> 8)
    return crc ^ 0xffffffff
```

Nižšie je aj ukážka výpočtu CRC pre jeden byte, ktorého binárna reprezentácia je rovná 01011000, čo je dekadicky 88, čo je v ascii tabuľke písmeno X. Pre jednoduchosť príkladu som zvolil **CRC8 LSB**, aby výpočet nebol príliš zdĺhavý, ale výpočet pri CRC32 z môjho projektu je identický iba s rozdielnym polynómom.

	0				4				8				12				
vstup	0	1	0	1	1	0	0	0									
polynóm	1	0	0	0	0	0	1	1	1								
rev. vstup	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	
>> 1	0	0	0	0	0	0	0	0	0								
=	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	
>> 1		0	0	0	0	0	0	0	0	0							
=	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	
>> 1			0	0	0	0	0	0	0	0	0						
=	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	
XOR				1	0	0	0	0	0	1	1	1					
=	0	0	0	0	1	0	1	0	0	1	1	1	0	0	0	0	
XOR					1	0	0	0	0	0	1	1	1				
=	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	
>> 1						0	0	0	0	0	0	0	0	0			
=	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	
XOR							1	0	0	0	0	0	1	1	1		
=	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	
>> 1								0	0	0	0	0	0	0	0	0	
=	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	
CRC									0	1	0	0	0	1	1	0	

5. ARQ

ARQ (Automatic repeat request) je metóda kontrolovania pre prenos dát, ktorá používa špeciálne typy správ, ktoré sa všeobecne nazývajú **acknowledgements** (ďalej len ACK). V kombinácii s ACK túto metódu dopĺňajú aj **časovače**, ktoré majú určený konkrétny čas, ktorý čakajú, kým sa neobdrží ACK.

Kombináciou týchto dvoch prvkov je možné zabezpečiť **spoľahlivý prenos dát na nespoľahlivom kanáli**.

Ja som si zvolil pre toto zadanie metódu **Stop-and-wait ARQ**. Ide o spôsob, v ktorom vysielač po odoslaní správy prechádza do stavu počúvania, kedy čaká na ACK správu od prijímača. Ako aj z názvu vyplýva, ide teda o metódu, kde vysielač nikdy neodošle viac ako jednu správu, pokiaľ prvý nebol druhou stranou akceptovaný.

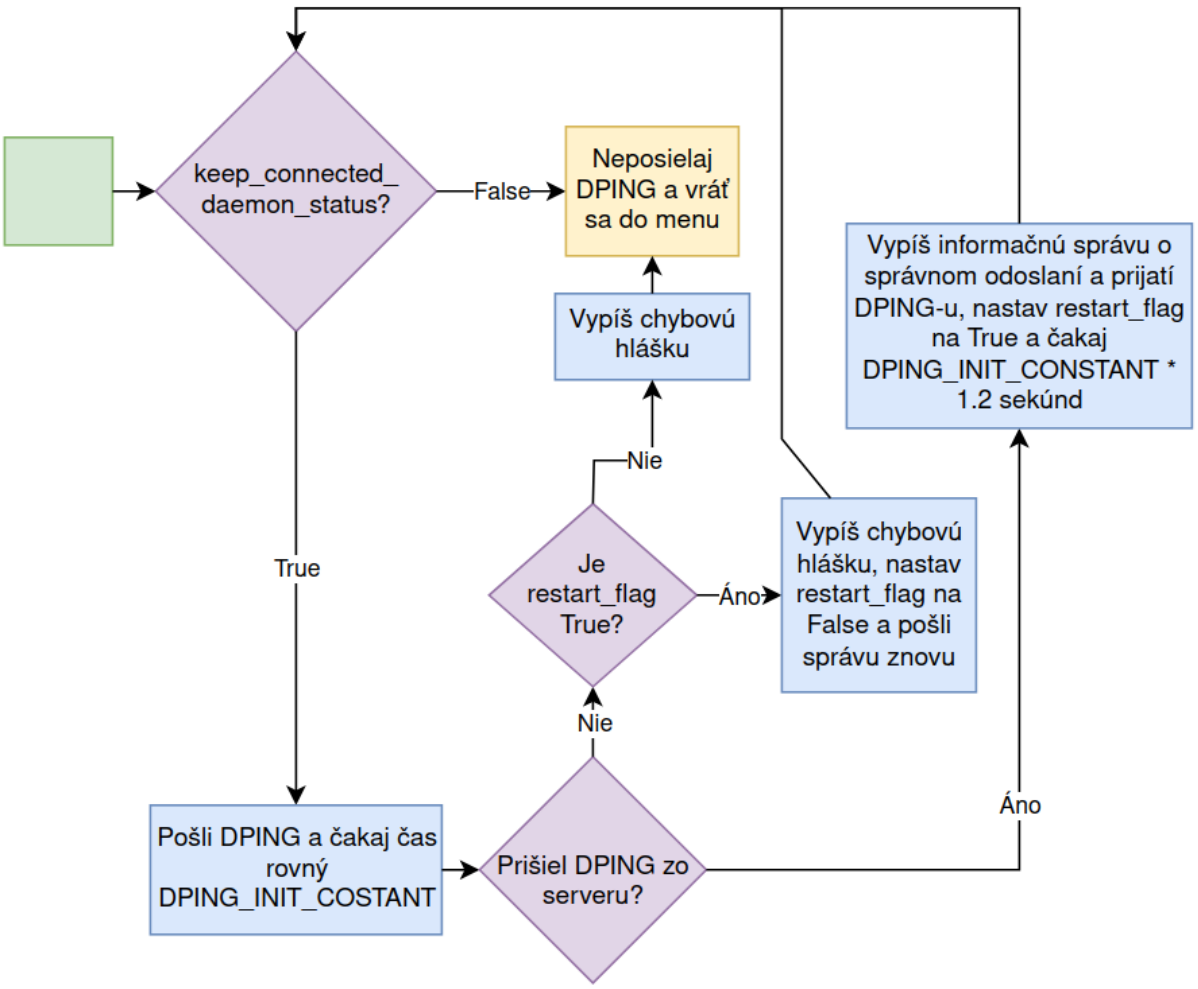
V mojom riešení sú na signalizáciou týchto ACK správ vyhradené typy **DATTC** a **DATTE**, ktoré znamenajú ***Data Transfer Correct*** a ***Data Transfer Error***. DATTC a DATTE sú správy, ktoré môže posielať výlučne tá strana, ktorá prijíma správu alebo súbor. Týmito správami signalizuje korektnosť doručenia správy skrz vyššie spomenuté CRC, ktoré sa musí na strane prijímateľa zhodovať s CRC, ktoré je zahrnuté v hlavičke správy.

Následne klient podľa DATTC alebo DATTE vyhodnotí, či pošle predchádzajúci paket znovu, alebo sa môže pokračovať ďalej.

6. Metódy pre udržanie spojenia

Metóda, ktorá udržuje spojenie medzi dvojicou klient-server, je založená na **posielaní špeciálnej signalizačnej správy DPING**, ktorú posiela ako prvý klient. Ak server túto správu dostane, pošle identickú správu klientovi naspäť, ktorý to vyhodnotí ako korektné doručenie tejto správy, a teda server je pripravený prijímať dáta. Táto správa sa posiela **každých DPING_INIT_CONSTANT * 1.2 sekúnd**, čo je konštanta, ktorou sa dá tento interval ovládať. “Down-time”, ktorý klient čaká na odpoveď na svoj DPING, je nastavený na konštantu **DPING_INIT_CONSTANT**, ktorá je v aktuálnom stave rovná 5 sekundám.

Toto udržiavanie spojenia správami DPING je možné kedykoľvek vypnúť a zapnúť v menu na strane klienta zmenou boolean hodnoty premennej **keep_connected_daemon_status**.



7. Diagramy pre klienta a server

Fungovanie klienta a servera som rozdelil do 5 flow-chartov nasledovne:

- Inicializácia a menu klienta
- Posielanie dát zo strany klienta
- Inicializácia a menu servera
- Prijímanie dát na strane servera
- Sekvenčný diagram komunikácie

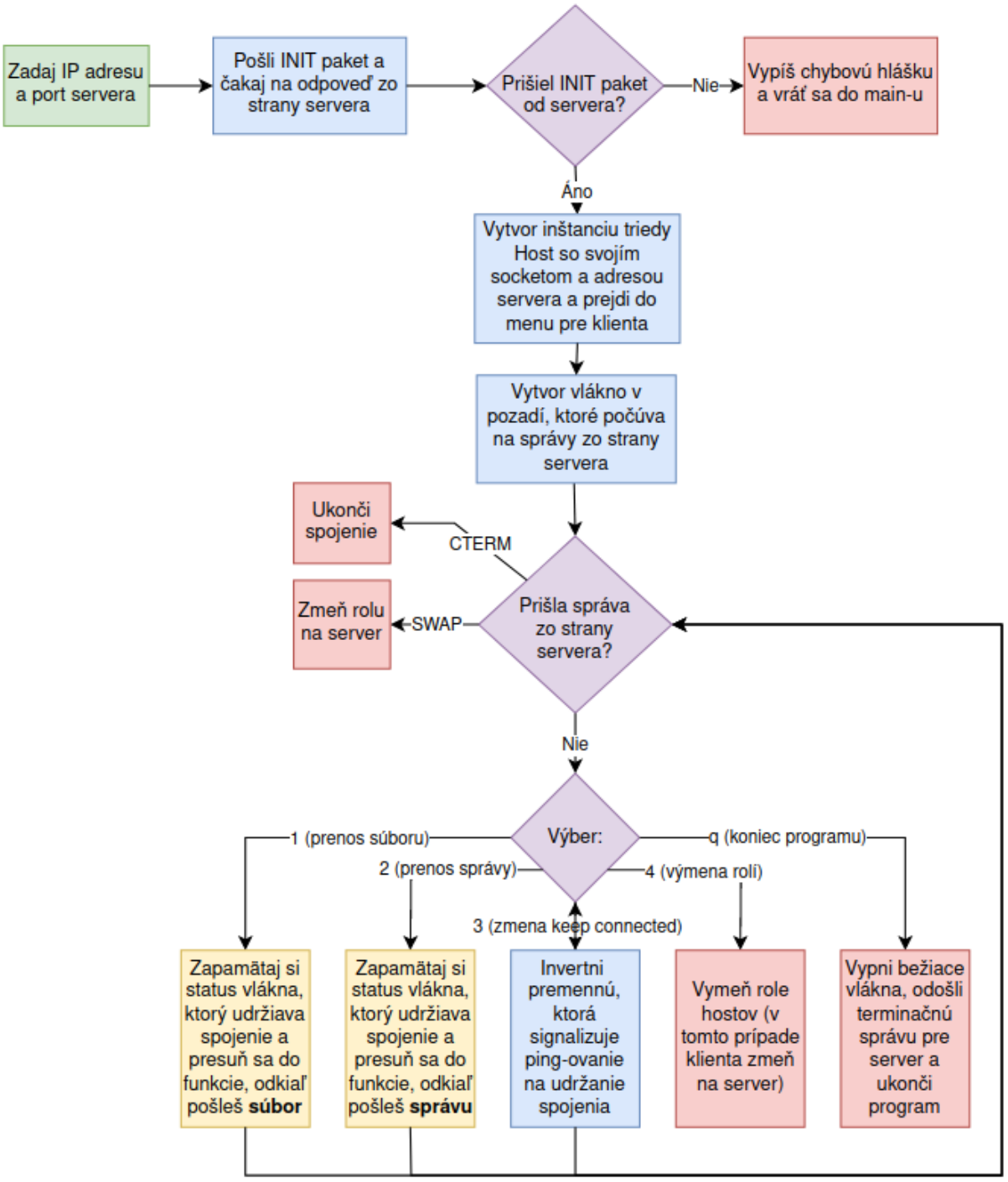
Legenda farieb pre bunky vo flow-charte:

- **modrá** - vykonáva nejakú činnosť
- **fialová** - rozhoduje pri rôznych podmienkach
- **zelená** - počiatočná bunka

- **červená** - terminačná alebo ukončujúca bunka
- **žltá** - táto bunka sa rozvetvuje vo flow-charte pod ňou

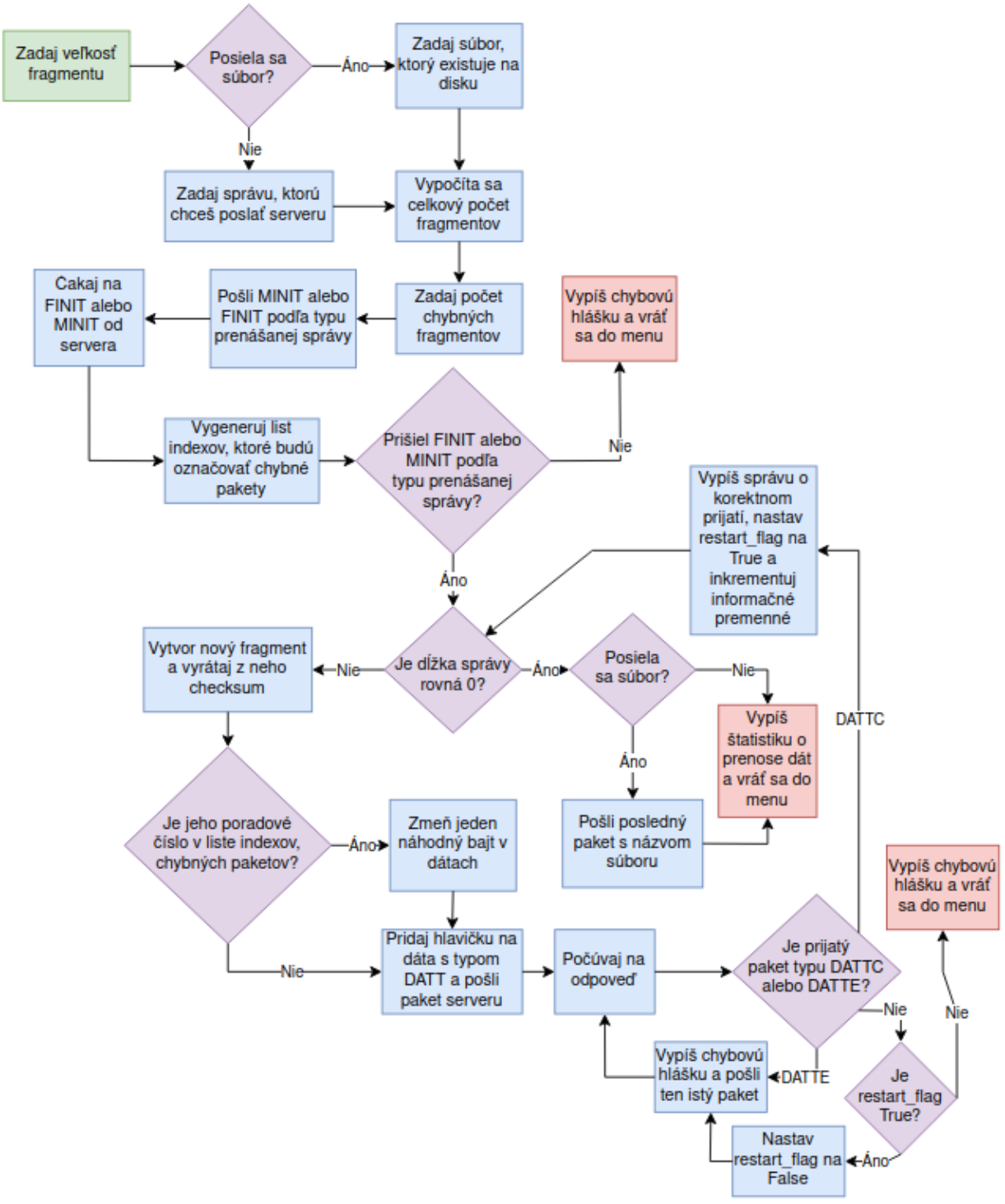
7.1 Inicializácia a menu klienta

Ak sa používateľ v menu rozhodne pre rolu klienta, program sa spustí funkciu **client_start()**, ktorá je reprezentovaná zelenou bunkou nižšie. Následne, obe žlté bunky budú zobrazené vo flow-charte 6.2, kde tieto bunky zdieľajú rovnakú funkciu s jedným rozdielnym argumentom.



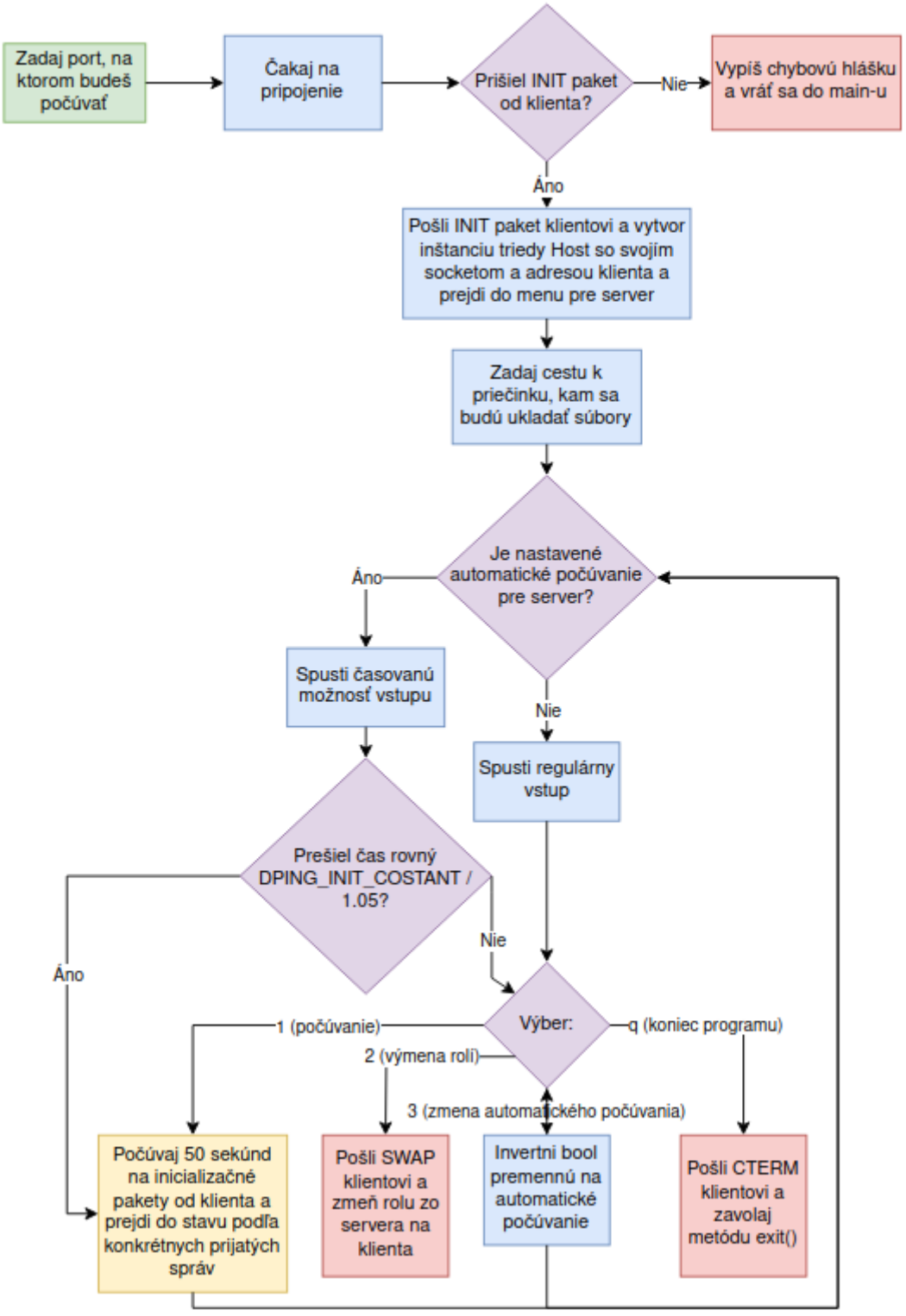
7.2 Posielanie dát zo strany klienta

Proces zasielania dát z pohľadu klienta je naznačený vo flow-charte nižšie. Do zelenej bunky sa klient dostane stlačením 1 alebo 2 v menu, skrz čo sa spustí funkcia **send_data**(client, messagetype) s korešpondujúcim typom správy.



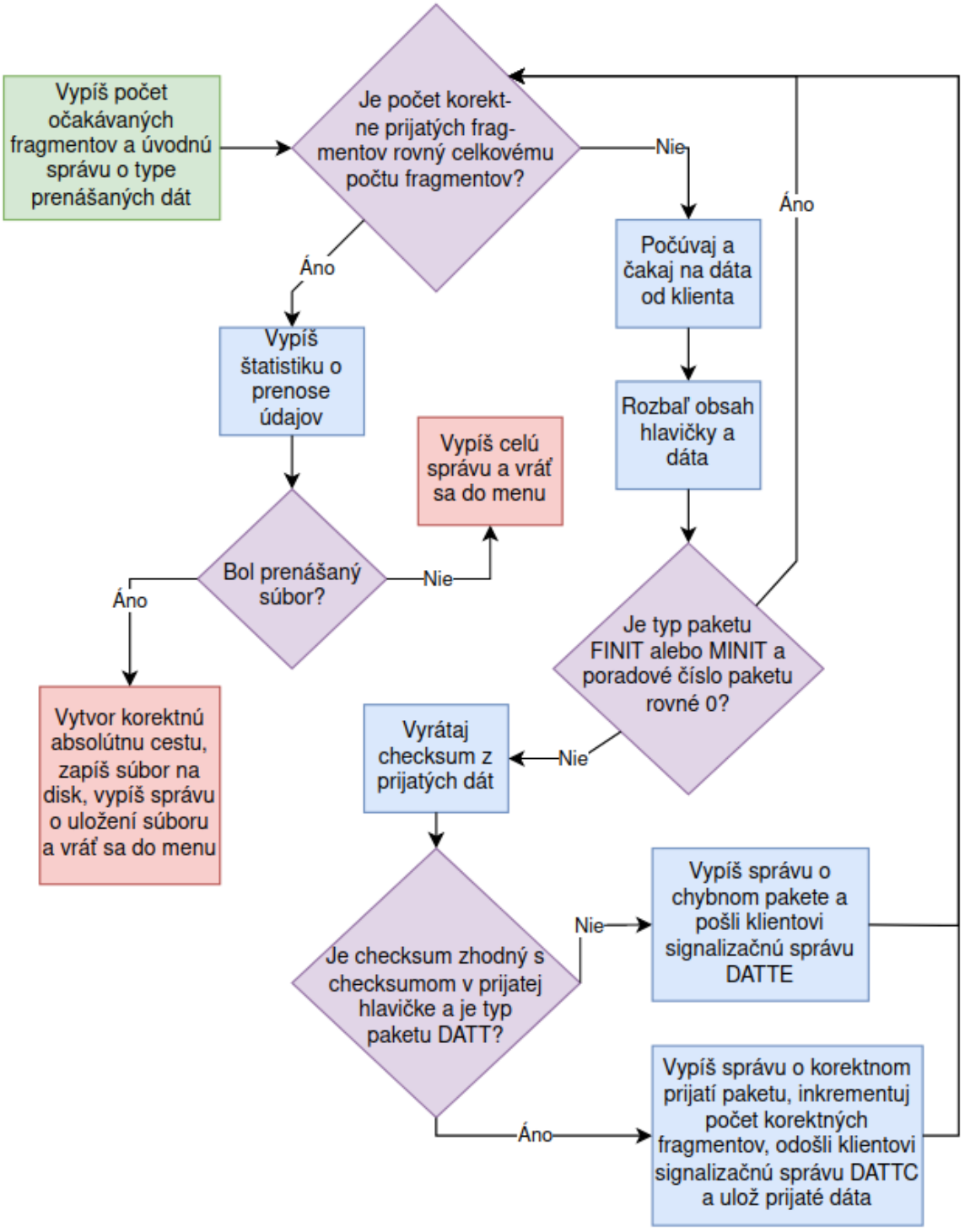
7.3 Inicializácia a menu servera

Ak sa používateľ v menu rozhodne pre rolu servera, program sa spustí funkciu **server_start()**, ktorá je reprezentovaná zelenou bunkou nižšie.



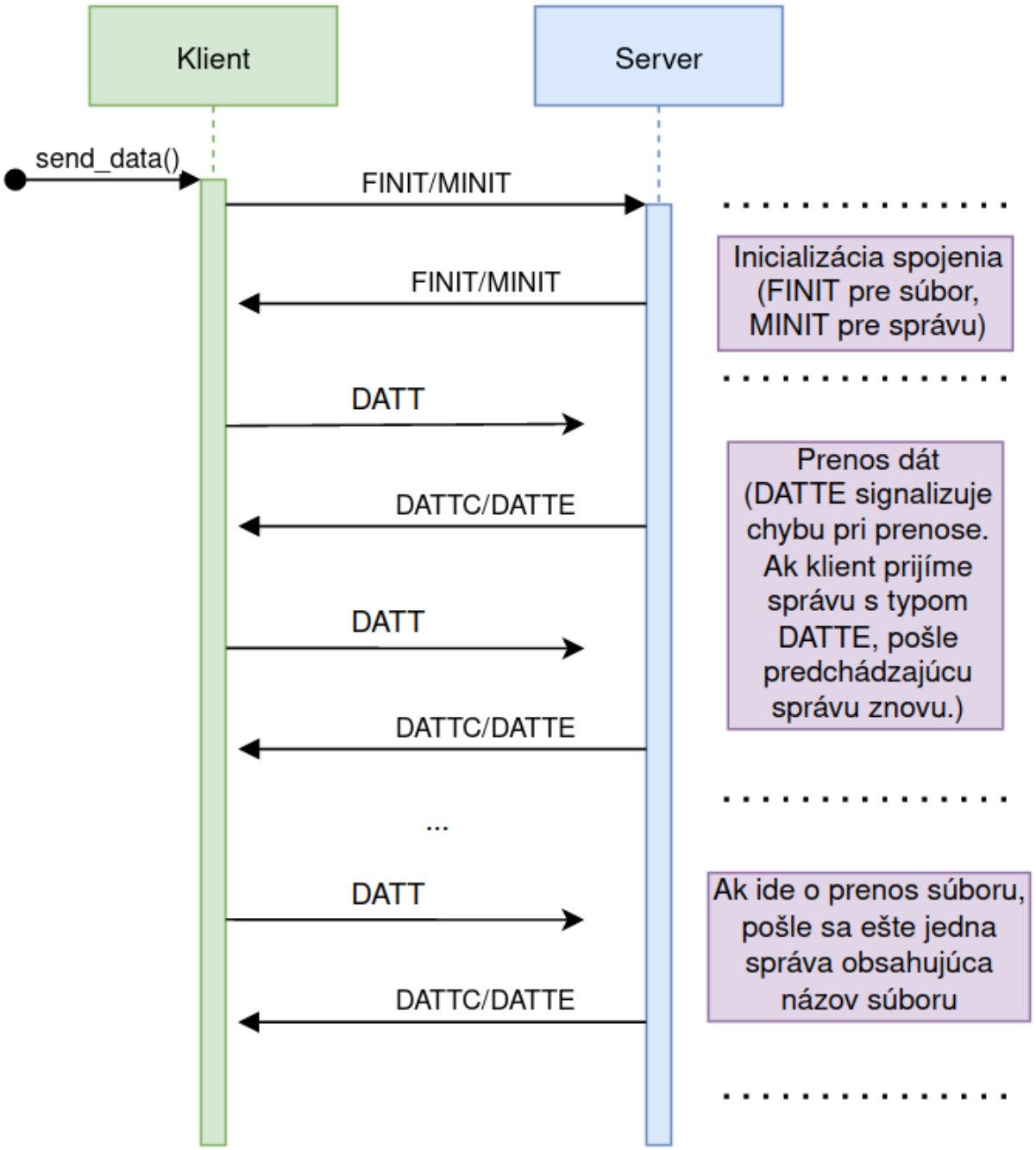
7.4 Prijímanie dát na strane servera

receive_data(server, messagetype, total_fragments, dest_directory) je funkcia, ktorá je zobrazená na diagrame v tejto časti. Jej úlohou je prijímanie dát, vypisovanie informačných správ a výpis celkovej správy či uloženia súboru do dest_directory.



7.5 Sekvenčný diagram komunikácie

V tomto diagrame je naznačená výmena typov správ pri prenose jedného súboru, resp. jednej správy. Posledná výmena pod troma bodkami je platná iba výhradne pre prenos súboru, pri posielaní správy sa názov súboru nemusí prenášať.


Klient
Server
send_data()
FINIT/MINIT
FINIT/MINIT
Inicializácia spojenia (FINIT pre súbor, MINIT pre správu)
DATT
DATTC/DATTE
Prenos dát (DATTE signalizuje chybu pri prenose. Ak klient prijíme správu s typom DATTE, pošle predchádzajúcu správu znovu.)
DATT
DATTC/DATTE
...
DATT
DATTC/DATTE
Ak ide o prenos súboru, pošle sa ešte jedna správa obsahujúca názov súboru

8. Knižnice, triedy a funkcie

8.1 Knižnice

Zoznam knižníc a ich využitie:

- **os** - na overenie absolútnej cesty k súboru, či súbor existuje a aká je jeho veľkosť
- **sys** - na časovo-obmedzený vstup prostredníctvom select.select na čítanie z sys.stdin
- **select** - na umožnenie čítania z I/O streamov, konkrétne stdin
- **random** - na náhodné generovanie indexov fragmentov, ktoré budú obsahovať chyby
- **struct** - na enkódovanie dátových typov na menšie zložky metódou pack() a následné dekódovanie metódou unpack()
- **threading** - na časovaný vstup pre systém Windows a na daemon Thread, ktorý posiela signalizačné DPING rámce zo strany klienta
- **socket** - na uskutočnenie spojenia a výmenu informácií medzi zariadeniami
- **time** - na uspatie DPING-u na určitý čas
- **concurrent.futures** - pre použitie ThreadPoolExecutora na vrátenie hodnoty z funkcie, ktorá je vykonávaná v samostatnom vlákne

8.2 Triedy

V mojom riešení je použitá iba jedna trieda, ktorá sa volá Host, a tou budem reprezentovať obidve komunikujúce strany.

```
class Host:
    def __init__(self, socket_, other_host):
        self.socket_ = socket_
        self.other_host_addr_ = other_host
```

Atribúty:

- **socket_** (type: tuple) -> obsahuje socket, resp. IP adresu a port samého seba.
- **other_host_addr_** (type: tuple) -> obsahuje socket popisujúci druhú stranu, či už ide o klienta alebo server

8.3 Dôležité funkcie

Program je delený do viacerých funkcií, ktoré sú vo svojej podstate delené na funkcie pre klienta a funkcie pre server a niekoľko spoločných dôležitých funkcií.

Dôležité funkcie pre klienta:

- **client_start**() -> vyžiada od klienta IP adresu a port servera, na ktorý sa chce pripojiť. Následne odošle inicializačný paket (INIT), čím zistí, či je server na danej IP a porte počúva.
- **client_menu**(client) -> menu pre klienta, odkiaľ bude ovládať program, parametrom je inštancia triedy Host.
- **keep_connected_daemon_function**(client) -> touto funkciou udržiava klient spojenie so serverom posielaním DPING správ.
- **send_data**(client, messagetype, fragment_size, filen, message, no_of_fragments, error_packets, error_packets_indices) -> wrapper funkcia pre posielanie dát a organizovanie všetkých parametrov pre posielanie a inicializáciu prenosu dát. Jej parametrami sú trieda Host, stringová forma reprezentujúca typ správy, maximálna veľkosť fragmentu, názov súboru, celá prenášaná správa, počet fragmentov na celú správu, počet chybných paketov a ich konkrétne indexy.
- **client_message_listener**(client, server_status) -> funkcia bežiaca na vlákne, ktorá periodicky prijíma správy od serveru a následne ich presúva do main vlákna. Jej parametrami sú trieda Host s údajmi klientovej strany a Event s názvom server_status, ktorý signalizuje, kedy sa má funkcia prestať vykonávať.

Dôležité funkcie pre server:

- **server_start**() -> vyžiada od servera port, na ktorom bude počúvať a následne počká na pripojenie od klienta.
- **server_menu**(server) -> menu pre server pre jednoduché funkcionality, parametrom je inštancia triedy Host.
- **receive_data**(server, messagetype, total_fragments, dest_directory) -> prijíme dáta na základe inicializácie a typu správy od klienta. Jej parametrami sú trieda Host, typ správy a celkový počet očakávaných fragmentov, ktoré sa server dozvedel z inicializačného paketu MINIT alebo FINIT. Posledným parametrom je cieľový priečinok, do ktorého sa súbor uloží.
- **server_input_prompt**() -> spustí trojicu objektov z knižnice threading, konkrétne Event, Timer a Thread, ktoré spoločne docieľujú funkcionalitu časovaného vstupu pre server.

Dôležité spoločné funkcie:

- **create_header_and_payload**(packet_type, packet_number=0, checksm=0, data="") -> vytvorí hlavičku podľa parametrov a spojí ju do byte-streamu spolu s dátami.
- **unpack_message**(packet) -> rozbalí správu a vráti všetok jeho obsah po častiach

9. Používateľské rozhranie

Používateľské rozhranie je, ako bolo vyššie spomenuté, **orientované v konzolovom okne**. Pri zapnutí programu sa používateľovi zobrazí nasledujúci výpis:

```
$ -- UDP Communicator
$ -- Author: Marko Stahovec

# -- Choose 1 -> client
# -- Choose 2 -> server
# -- Choose q -> quit

# -- Choice:
```

Tento výpis zobrazuje 2 typy správ:

- $ -- predstavuje všeobecné informácie o programe
- # -- predstavuje číselný, resp. jednoznakový vstup od používateľa

Používateľ má teda tri možnosti: **zapnúť program v móde klienta, zapnúť program v móde servera alebo vypnúť celý program**.

9.1 Typy konzolových správ

Tento program dodržiava isté konvencie výpisu, skrz ktoré poskytuje deskriptívne výpisy, ktoré sú ľahko pochopiteľné. Všetky typy konzolových správ sa nachádzajú v tabuľke nižšie, vrátane 2 typov, ktoré boli spomenuté vyššie. **Niektoré z týchto správ slúžia ako popis pre vstup, na ktorý sa čaká od používateľa.**

Legenda:

- **modré** bunky: výpis
- **fialové** bunky: výpis pre prijímanie/odosielanie správ
- **oranžové** bunky: vstup

Skratka/znak správy	Popis
$ --	Informačný výpis o vlastnostiach programu
[INFO] --	Informačný výpis o aktuálnom procese alebo vykonaní posledného úkonu
[TYPE] --	Výpis vzťahujúci sa na konkrétny typ správy (TYPE môže byť nahradený za akýkoľvek packet_type)
[ERROR] --	Chybový výpis
[OUT] --	Výpis finálnej správy
Packet X |	Začiatok výpisu pre konkrétny paket a informácie o ňom
↓	Výpis pre chybnú správu a nasmerovanie za jej korektným doručením
# --	Číselný, resp. jednoznakový vstup alebo možnosť
[IP] --	Vstup pre formát IP adresy
[PORT] --	Vstup pre celočíselný formát portu
[INT] --	Celočíselný vstup
[STR] --	Vstup pre reťazec znakov

9.2 Server

Po stlačení dvojky v hlavnom menu sa spustí funkcia **server_start()**, kde sa vykoná inicializačný proces pre server. Následne je od užívateľa vyžiadaná **hodnota portu**, na ktorom má server počúvať a potom je vypísaná informačná správa o stave programu.

```
$ -- UDP Communicator
$ -- Author: Marko Stahovec

# -- Choose 1 -> client
# -- Choose 2 -> server
# -- Choose q -> quit

# -- Choice: 2
[PORT] -- Input port to listen to: 22222
[INFO] -- Waiting for a client...
```

V tomto stave server čaká na pripojenie od klienta. Po pripojení klienta na IP servera a ručne zadaný port je server ponúknutý ďalším vstupom, a to konkrétne zadaním **cesty k adresáru**, do ktorého sa budú ukladať prijaté súbory. Túto cestu si môže server kedykoľvek zmeniť v menu, do ktorého sa dostane po zadaní tejto cesty.

```
$ -- UDP Communicator
$ -- Author: Marko Stahovec

# -- Choose 1 -> client
# -- Choose 2 -> server
# -- Choose q -> quit

# -- Choice: 2
[PORT] -- Input port to listen to: 22222
[INFO] -- Waiting for a client...
[INIT] -- received, established connection from: ('192.168.0.187', 37932)

[STR] -- Choose destination directiory ('.' for current dir.): ./storage

-------------- SERVER --------------

# -- Choose 1 -> Continue listening
# -- Choose 2 -> Swap host roles
# -- Choose 3 -> Toggle automatic listening [current status: False]
# -- Choose 4 -> Change destination directory
# -- Choose q -> Quit

# -- Choice:
```

Ako možno vidieť na vzorovom výpise, po pripojení klienta a zadaní cieľového adresáru, server má niekoľko možností na ovládanie spojenia s klientom ako aj vlastné nastavenia. V tomto stave sa odporúča nastaviť serveru automatické počúvanie číslom 3. Následne po (DPING_INIT_CONSTANT / 5) sekundách prejde server do stavu počúvania.

Zvyšné možnosti vstupu pre server predstavujú ďalšie funkcionality, ktoré vyplývajú zo zadania alebo z bežných potrieb používateľa.

Po stlačení 3 prejde server do stavu počúvania:

```
$ -- UDP Communicator
$ -- Author: Marko Stahovec

# -- Choose 1 -> client
# -- Choose 2 -> server
# -- Choose q -> quit

# -- Choice: 2
[PORT] -- Input port to listen to: 22222
[INFO] -- Waiting for a client...
[INIT] -- received, established connection from: ('192.168.0.187', 37932)

[STR] -- Choose destination directiory ('.' for current dir.): ./storage

-------------- SERVER --------------

# -- Choose 1 -> Continue listening
# -- Choose 2 -> Swap host roles
# -- Choose 3 -> Toggle automatic listening [current status: False]
# -- Choose 4 -> Change destination directory
# -- Choose q -> Quit

# -- Choice: 3

-------------- SERVER --------------

# -- Choose 1 -> Continue listening
# -- Choose 2 -> Swap host roles
# -- Choose 3 -> Toggle automatic listening [current status: True]
# -- Choose 4 -> Change destination directory
# -- Choose q -> Quit

# -- Choice:
[INFO] -- Server is automatically listening...
```

Vzorový výpis po úspešnom prenose dát:

```
# -- Choice: 1
[FINIT] -- received from: ('192.168.0.185', 52153)

************************************
[INFO] -- EXPECTED FRAGMENTS: 7
[INFO] -- RECEIVING FILE...
************************************


-------------------------------------
Packet 1 | 15B | [DATT] | was accepted correctly.
Packet 2 | 15B | [DATT] | was REJECTED.
   ↓
Packet 2 | 15B | [DATT] | was accepted correctly.
Packet 3 | 15B | [DATT] | was accepted correctly.
Packet 4 | 15B | [DATT] | was accepted correctly.
Packet 5 | 15B | [DATT] | was REJECTED.
   ↓
Packet 5 | 15B | [DATT] | was accepted correctly.
Packet 6 | 15B | [DATT] | was REJECTED.
   ↓
Packet 6 | 15B | [DATT] | was accepted correctly.
Packet 7 | 9B | [DATT] | was accepted correctly.
-------------------------------------


************************************
[INFO] -- CORRECTLY DELIVERED FRAGMENTS: 7
[INFO] -- PACKETS WITH ERROR: 3
************************************


-------------------------------------
[INFO] -- File /map4.txt created successfully.
Full path: /home/marko/PycharmProjects/pks2/storage/map4.txt
-------------------------------------
```

Následne sa server vracia späť do menu.

9.3 Klient

Ak sa používateľ v hlavnom menu rozhodne pre rolu klienta, spustí sa funkcia **client_start()**, kde prebehne inicializačný proces pre stranu klienta. V tejto funkcii sa od používateľa vyžiada **IP adresa servera a port** na ktorom počúva. Po úspešnej výmene INIT správ je klient presunutý do svojho menu.

```
$ -- UDP Communicator
$ -- Author: Marko Stahovec

# -- Choose 1 -> client
# -- Choose 2 -> server
# -- Choose q -> quit

# -- Choice: 1
[IP] -- Input IP address of a server (Your IP: 192.168.0.187): 192.168.0.187
[PORT] -- Input port number of a server: 22222
[INIT] -- received, connected to address: ('192.168.0.187', 22222)


-------------- CLIENT --------------

# -- Choose 1 -> Transfer a file
# -- Choose 2 -> Transfer a message
# -- Choose 3 -> Toggle keep connected status [current status: False]
# -- Choose 4 -> Swap host roles
# -- Choose q -> Quit

# -- Choice:
```

Klient je ponúknutý piatimi možnosťami, ktoré sú samovysvetľujúce. Prvou sa klient presunie do stavu **prenosu súboru**, druhou možnosťou prejde na **prenos textovej správy**, treťou zmení **aktuálny stav vlákna** na udržiavanie spojenia so serverom, štvrtou si vyžiada **zmenu rolí** so serverom a možnosťou q **ukončí program** na svojej strane. Takéto ukončenie bude signalizované aj na strane servera, že sa klient odpojil.

Po stlačení 3 na vstupe sa zmení stav vlákna, čo má za efekt posielanie **udržiavacích DPING správ** zo strany klienta.

```
-------------- CLIENT --------------

# -- Choose 1 -> Transfer a file
# -- Choose 2 -> Transfer a message
# -- Choose 3 -> Toggle keep connected status [current status: False]
# -- Choose 4 -> Swap host roles
# -- Choose q -> Quit

# -- Choice: 3

-------------- CLIENT --------------

# -- Choose 1 -> Transfer a file
# -- Choose 2 -> Transfer a message
# -- Choose 3 -> Toggle keep connected status [current status: True]
# -- Choose 4 -> Swap host roles
# -- Choose q -> Quit

# -- Choice: [DPING] -- successful, connection is maintained.
[DPING] -- successful, connection is maintained.
[DPING] -- successful, connection is maintained.
[DPING] -- successful, connection is maintained.
[DPING] -- successful, connection is maintained.
```

Vzorový výpis pri prenose textovej správy z pohľadu klienta je zobrazený na obrázku nižšie. Po takom prenose je klient presunutý späť do menu, ktoré bude načítané v pôvodnom stave, v akom bolo ponechané.

Ako možno vidieť na výpise nižšie, tak klient je ponúknutý niekoľkými parametrami, ktorými si môže nastaviť atribúty prenosu, ako napr. **maximálna veľkosť** jedného **fragmentu** či **počet chybných správ**, ktoré by v takom prípade detekované ako chybné na strane servera.

```
# -- Choice: 2
[INT] -- Input fragment size: 15
[STR] -- Input message: This is my first message. Is server listening?
[INT] -- Set amount of error packets: 3
[MINIT] -- Waiting for acknowledgment
[MINIT] -- Waiting for acknowledgment
[MINIT] -- Waiting for acknowledgment
[MINIT] -- received from: ('192.168.0.187', 22222)


-------------------------------------
Packet 1 | 15B | [DATTE] received, retransmitting error fragment...
   ↓
Packet 1 | 15B | [DATTC] received.
Packet 2 | 15B | [DATTC] received.
Packet 3 | 15B | [DATTE] received, retransmitting error fragment...
   ↓
Packet 3 | 15B | [DATTC] received.
Packet 4 | 15B | [DATTC] received.
Packet 5 | 15B | [DATTE] received, retransmitting error fragment...
   ↓
Packet 5 | 15B | [DATTC] received.
Packet 6 | 13B | [DATTC] received.
-------------------------------------


*************************************
[INFO] -- CORRECTLY DELIVERED FRAGMENTS: 6
[INFO] -- PACKETS WITH ERROR: 3
*************************************
```

Po tomto výpise sa klient ocitne opäť v rovnakom menu.

10. Záver

Myslím si, že takáto implementácia na stanovenú problematiku vyhovuje všetkým podmienkam a je dostatočne čitateľná a robustná, keďže používateľské rozhranie je samovysvetľujúce a samotný program primerane efektívny vzhľadom na všetky aspekty. Zadanie bolo koncipované vyhovujúco, jediným problémom bol fakt, že doň bolo natlačených priveľa komplikácií, ktoré boli skôr súbojom s programovacím jazykom ako samotnou sieťarinou.

11. Zdroje

[1]: https://rosettacode.org/wiki/CRC-32#Python