

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Počítačové a komunikačné siete

Zadanie č.2

UDP komunikátor

Akademický rok 2022/2023

Meno: Ján Ágh

Cvičiaci: Ing. Lukáš Mastilák

Dátum: 10.12.2022

Počet strán: 16

Obsah

1	Stručný opis problematiky a zadania	1
2	Navrhnuté riešenie	2
2.1	Použité programové prostriedky	2
2.2	Organizácia projektu	2
2.3	Primárne vlastnosti riešenia	3
2.3.1	Návrh hlavičky vlastného protokolu	3
2.3.2	Opis spôsobu inicializácie a ukončenia komunikácie	4
2.3.3	Opis metódy kontrolnej sumy (CRC)	5
2.3.4	Opis použitej ARQ metódy	6
2.3.5	Opis metódy na udržiavanie spojenia	7
2.3.6	Opis spôsobu zmeny úloh	7
2.4	Zmeny oproti návrhu	8
3	Opis dôležitých častí programu	9
3.1	Diagram priebehu programu	9
3.2	Zabalenie a rozbalenie paketov	10
3.3	Odosielanie a prijímanie dátových paketov	11
3.4	Evaluácia prijatých signálov	12
3.5	Systém doručovania keep-alive signálov	13
4	Ukážky testovacích scenárov	14
4.1	Prenos správy	14
4.2	Výmena úloh	15
4.3	Prenos súboru	16

1 Stručný opis problematiky a zadania

Cieľom zadania je navrhnúť program umožňujúci komunikáciu dvoch zariadení v lokálnej sieti typu Ethernet s využitím vlastného protokolu nad protokolom User Datagram Protocol (UDP), ktorý pracuje na transportnej vrstve sieťového modelu TCP/IP.

Program sa skladá z dvoch častí, a to vysielacej (klient) a prijímacej (server), a umožňuje používateľovi preposielať jednoduché textové správy a súbory ľubovoľného formátu. Používateľ má taktiež možnosť zadať si maximálnu veľkosť fragmentu, ktorá nesmie presiahnuť veľkosť 1500B, aby nedochádzalo k opätovnej fragmentácii na datalinkovej vrstve.

Program obsahuje kontrolu chýb pri komunikácii a znovuvyžiadanie chybných fragmentov, pričom na túto skutočnosť využíva signalizačné správy odosielané počas komunikácie. Počas doby trvania komunikácie jednotlivé strany v intervale každých 5 sekúnd odosielajú paket slúžiaci na udržiavanie spojenia druhej strane dovtedy, kým sa používateľ nerozhodne manuálne komunikáciu prerušiť alebo nevznikne nečakaná chyba.

2 Navrhnuté riešenie

2.1 Použité programové prostriedky

Na riešenie úlohy bol použitý programovací jazyk Python verzie 3.10.8. Riešenie bolo vyvíjané v prostredí Visual Studio Code verzie 1.72.1 a pri jeho vývoji boli využité základné objektovo orientované princípy vrátane dedenia.

2.2 Organizácia projektu

Program pozostáva z niekoľkých základných súborov, ktoré sú nevyhnutné pre jeho funkčnosť. Každý zo spomínaných súborov plní istú úlohu počas vykonávania programu.

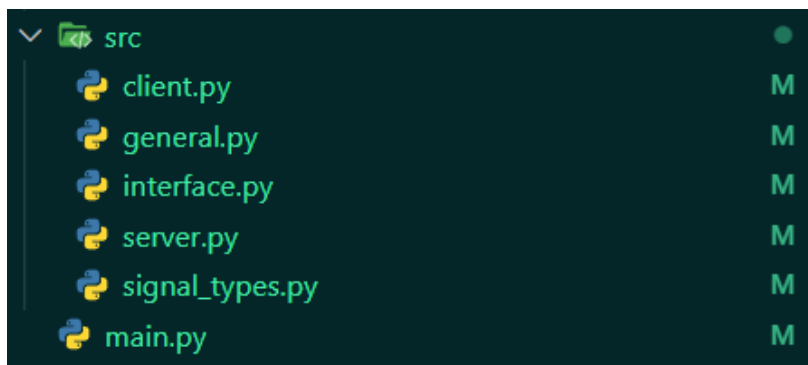
Prvým súborom je *interface.py* nachádzajúci sa v adresári *src*. Tento súbor obsahuje statickú triedu *Interface*, ktorá slúži na poskytovanie spätnej väzby používateľovi pomocou konzolových výpisov a taktiež akceptovanie vstupných informácií. Tvoria ju metódy *initial_setup()* na prvotné nastavenie programu (výber možnosti klient – server), *initialize_client()* a *initialize_server()* na zobrazenie menu a akceptovanie vstupu patriaceho ku klientskej, resp. serverovskej časti programu a ďalšie, menej významné a pomocné metódy.

Dvojica súborov *client.py* a *server.py* v adresári *src* obsahujú triedy s konkrétnymi implementáciami príslušných častí programu – klientskej časti v triede *Client* a serverovskej časti v triede *Server*. Pomocný súbor *adresses.py* obsahuje dôležité údaje využívané pri výmene úloh.

Súbor *general.py* a v ňom sídliaca trieda *General* predstavuje rodičovskú triedu pre *Client* aj *Server* a obsahuje dôležité metódy využívané oboma týmito triedami, ako napríklad inicializácia a rušenie spojenia, odosielanie a analýza paketov a mnoho ďalších.

V súbore *signal_types.py* je umiestnená enumeračná trieda *Signals.py* so zadanými konštantami predstavujúcimi typy signálov, ktoré dokážu jednotlivé strany programu prijímať, úspešne analyzovať a odosielať.

Posledným a zároveň najdôležitejším súborom je *main.py*, tvoriaci jadro celého programu a spájajúci zvyšné triedy a súbory do jedného celku. Ide o vstupný bod programu a v ňom sa vytvoria príslušné objekty tried *Server* a *Client*.



Obr. 2.1 Adresárová štruktúra implementácie

2.3 Primárne vlastnosti riešenia

Nadchádzajúca podkapitola vo veľkej miere obsahuje identické informácie, aké sa nachádzajú aj v dokumente návrhu programu, s menšími úpravami. Vykonané zmeny oproti návrhu nie sú príliš rozsiahleho charakteru.

2.3.1 Návrh hlavičky vlastného protokolu

Súčasťou zadania je aj návrh vlastného protokolu, ktorý bude pracovať nad protokolom UDP. Keďže protokol UDP neodosiela žiadne potvrdenia týkajúce sa prijatia dát (a tým pádom druhá strana komunikácie nemá žiadny spôsob dozvedieť sa, či dáta prišli v poriadku a v správnom poradí), úlohou navrhnutého protokolu bude vyriešiť aj túto problematiku. Štruktúra protokolu je opísaná nižšie:

1. Typ paketu – informácia, o aký druh paketu sa jedná:

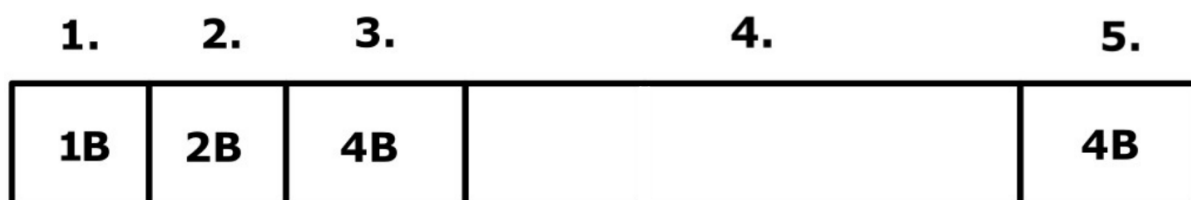
- **0** – paket *SYN*, označujúci začiatok nadväzovania spojenia
- **1** – paket *SYN+ACK*, odpoveď druhej strany na požiadavku spojenia
- **2** – paket *ACK*, odpoveď na viacero požiadaviek (definitívne potvrdenie spojenia, odpoveď na keep-alive pakety, odpoveď na prijaté fragmenty...)
- **3** – paket *keep-alive* na overovanie konektivity druhej strany
- **4** – paket *FIN* na ukončenie spojenia
- **5** – paket *FIN+ACK* na odpoveď k požiadavke ukončenia spojenia
- **6** – paket slúžiaci na opätovné vyžiadanie dát
- **7** – paket na inicializáciu odosielania dát (obsahuje typ alebo názov s príponou súboru)
- **8** – paket na odoslanie súboru alebo textovej správy
- **9** – paket na oznámenie zmeny rolí (z klienta sa stane server a naopak)

2. Identifikácia prenosu – každý proces prenosu má vlastné identifikačné číslo

3. Poradové číslo – slúži na správne zoskupenie fragmentov na strane prijímateľa

4. Dáta – prenášané údaje

5. CRC – kontrolný súčet na detekciu chýb v prenášanom pakete



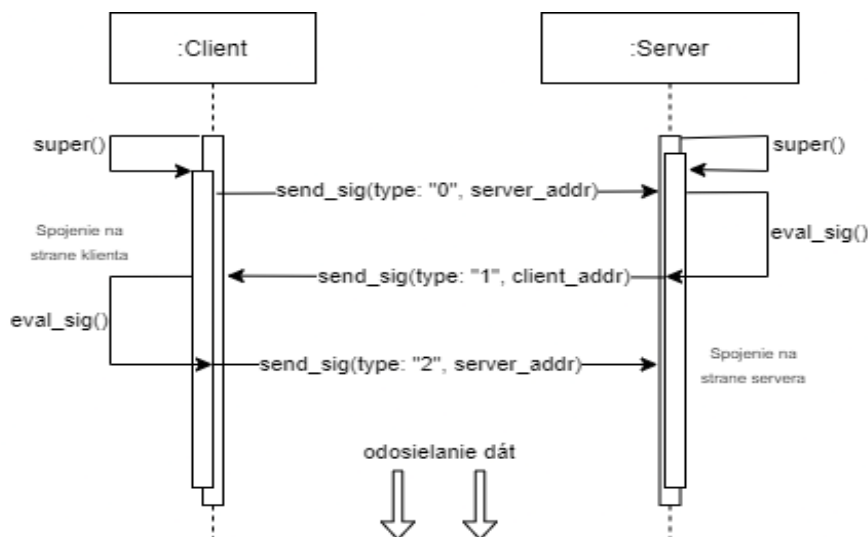
Obr. 2.1 Znázornenie hlavičky protokolu

Dôležitou poznámkou je fakt, že okrem signálov typu 7 a 8 sa všetky druhy signálov prenášajú v pakete samostatne (paket pri ich odosielaní neobsahuje žiadny iný údaj, iba číslo signálu uvedené v prvej časti hlavičky protokolu).

2.3.2 Opis spôsobu inicializácie a ukončenia komunikácie

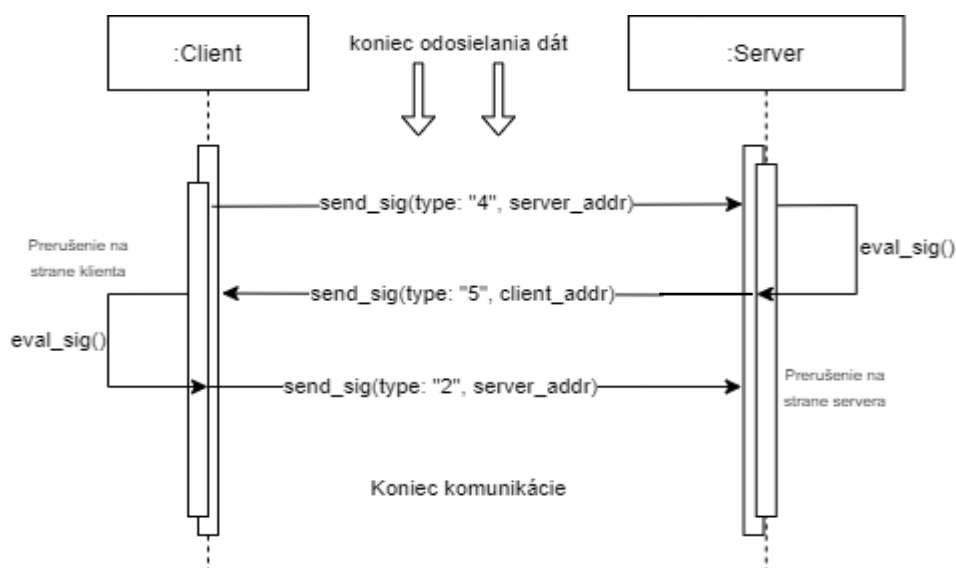
Pred začiatkom každej komunikácie / prenosu je z pohľadu zadania dôležité nadviazať spoľahlivé spojenie s prijímacou stranou. Keďže protokol User Datagram Protocol (UDP) pracuje nespoľahlivo (bez spojenia a overovania), o nadviazanie a rušenie spojenia sa stará vlastný protokol, navrhnutý v predchádzajúcej podkapitole, pomocou špeciálnych druhov paketov uvedených v časti “Typ paketu”.

O začiatok prvej inicializácie komunikácie sa stará odosielateľ (klient), ktorý sa snaží nadviazať spojenie so serverom formou *three-way handshake*. Urobí tak odoslaním inicializačného paketu typu 0 – *SYN*. Akonáhle server identifikuje klientovu požiadavku, odošle mu späť paket typu 1 – *SYN+ACK* na potvrdenie akceptovania požiadavky spojenia (*ACK*) spolu s vlastnou inicializačnou požiadavkou (*SYN*). Po doručení uvedeného paketu sa komunikácia na strane klienta považuje už za nadviazanú, je však potrebné odoslať na stranu servera ešte jeden paket typu 2 – *ACK* na potvrdenie prijatia predchádzajúceho paketu. Komunikácia sa považuje za nadviazanú aj na strane servera až po prijatí tohto paketu. Po úspešnom uskutočnení *three-way handshake* je možné začať prenos dát odoslaním paketu 7.



Obr. 2.2 Sekvenčný diagram nadviazania spojenia

Po úspešnom prenose dát sa klient môže rozhodnúť oficiálne prerušiť komunikáciu. Začiatok prerušenia komunikácie odštartuje odoslaním paketu typu 4 – *FIN*, z ktorého sa server dozvie o snahe ukončiť spojenie. Úlohou servera je odoslať odpoveď vo forme paketu typu 5 – *FIN+ACK*, v ktorom oboznámi klienta o prijatí úvodného paketu typu 4 – *FIN* (*ACK*) a zároveň doručí vlastnú žiadosť o prerušenie komunikácie (*FIN*). Akonáhle klient prevezme spomínaný paket, považuje komunikáciu za úspešne prerušenú a v poslednom kroku odošle na server paket typu 2 – *ACK* na potvrdenie prijatia predchádzajúceho paketu. Server považuje spojenie za úspešne ukončené až po prijatí tohto potvrdzovacieho paketu.



Obr. 2.3 Sekvenčný diagram prerušenia spojenia

2.3.3 Opis metódy kontrolnej sumy (CRC)

Výpočet aj evaluácia správnosti kontrolného súčtu pri každom odoslanom a prijatom pakete sa realizuje pomocou metódy `crc32()` pochádzajúcej z knižnice *zlib*. Uvedená metóda ako argument akceptuje dáta spolu s navrhnutou hlavičkou a vráti 32-bitovú celočíselnú hodnotu. Táto hodnota sa v uvedenom formáte vloží do hlavičky protokolu, paket dorazí na cieľovú adresu a tam sa opätovne uskutoční výpočet sumy s cieľom zistiť, či údaje dorazili bezchybne. Chybovosť sa overuje porovnaním nového výpočtu s hodnotou uloženou v hlavičke, pričom ak sa zhodujú, program pokračuje v ďalšej analýze paketu a odošle späť paket typu 2, ale ak nastane nezhoda indikujúca chybu, prijatý paket sa odstráni a odošle sa paket typu 5 na vyžiadanie opätovného odoslania údajov.

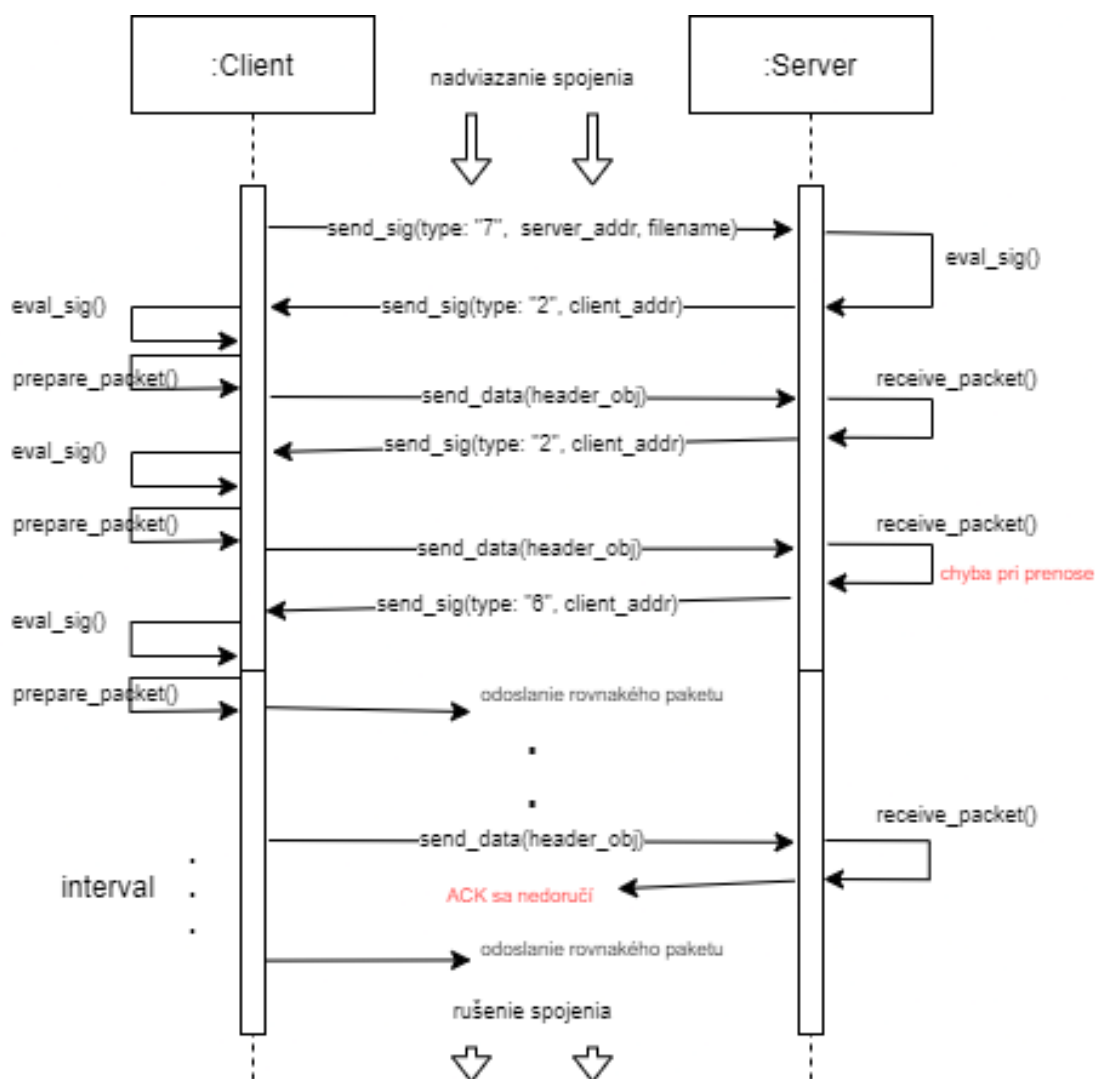
Uvedená metóda počas výpočtu kontrolného súčtu kombinuje dve operácie – binárny posun vľavo a následne operáciu XOR výsledku s binárnou hodnotou 1 0000 0100 1100 0001 0001 1101 1011 0111. Krátky opis samotného algoritmu by vyzeral nasledovne:

- Obráť vstupnú hodnotu a prirad' na jej koniec celkovo 32 núl
- Vykonaj operáciu XOR vstupu s hodnotou 0xFFFFFFFF
- Ak je prvý bit výsledku rovný 1, vykonaj XOR s vyššie písaným binárnym polynómom
- Vykonaj binárny posun celého výsledku vľavo a posuň číslo vpravo
- Opakuj kroky 3 a 4, dokým prvých 8 bitov výsledku nebude nulových
- Vykonaj operáciu XOR vstupu s hodnotou 0xFFFFFFFF a obráť výsledok

2.3.4 Opis použitej ARQ metódy

ARQ (Automatic Repeat Quest) je rozsiahla skupina metód detekcie a korekcie chýb využívaných pri prenose dát. Pre zaistenie spoľahlivého prenosu dát po nespoľahlivom kanáli používa signály na potvrdenie príjmu. Úlohou príjemcu je indikovať odosielateľovi úspešné a bezchybné prijatie paketu odoslaním kladného potvrdenia, pričom ak odosielateľ neobdrží uvedené potvrdenie do doby uplynutia zadefinovaného časového intervalu, odošle daný paket opäť. Počet znovuodoslaní paketov je obmedzený a ak sa prekročí maximálny počet pokusov, nastane chyba v komunikácii a spojenie sa považuje za prerušené.

Komunikátor pre overovanie správnosti odosielania dát využíva ARQ metódu *Stop and Wait*, ktorej princíp je veľmi jednoduchý. Odosielateľ posiela pakety po jednom, pričom po každom odoslanom pakete očakáva signál kladného potvrdenia zo strany príjemcu. Nasledujúci paket je odoslaný až po prijatí spomenutého potvrdenia. Na druhej strane, od prijímateľa sa očakáva, že po prijatí paketu toto potvrdenie odošle. Samozrejme, ak prijatý paket obsahuje chybné údaje (zistené napr. tým, že vypočítaný kontrolný súčet sa nerovná súčtu uvedeného v hlavičke), namiesto jednoduchého potvrdenia prijatia sa odosielateľovi doručí paket typu 6, slúžiaci na znovuvyžiadanie predchádzajúceho paketu.



Obr. 2.4 Sekvenčný diagram ARQ metódy Stop and Wait

2.3.5 Opis metódy na udržiavanie spojenia

Odosielanie paketov typu 3 (na overovanie konektivity druhého komunikujúceho zariadenia) sa uskutočňuje v odlišnom vlákne ako primárna komunikácia, s cieľom predchádzať kolíziám. Uvedené pakety odosiela vysielací účastník komunikácie prijímaťúcemu v pravidelných intervaloch dĺžky 5 sekúnd, pričom po odoslaní sa očakáva aj odpoveď prichádzajúca z opačnej strany vo forme paketu typu 2 – *ACK*. Ak odpoveď nedorazí do piatich sekúnd, odošlú sa ešte ďalšie 4 pakety typu 3 (tiež v intervaloch 5 sekúnd) a ak ani na tieto žiadosti nepríde odpoveď, spojenie sa považuje za nefunkčné a automaticky sa preruší. Týmto by mali byť ošetrené situácie, kedy sa prvý paket typu 2 – *ACK* podarilo odoslať, ale nepodarilo sa ho doručiť (nedojde k automatickému ukončeniu spojenia).

Ak klient na odoslané pakety typu 3 nedostane odpoveď, považuje druhú stranu na nedosiahnuteľnú a okamžite preruší spojenie (nevyužije sa oficiálne prerušenie kombináciou paketov *FIN*, *FIN+ACK*, *ACK*, keďže server by na ne neodpovedal).

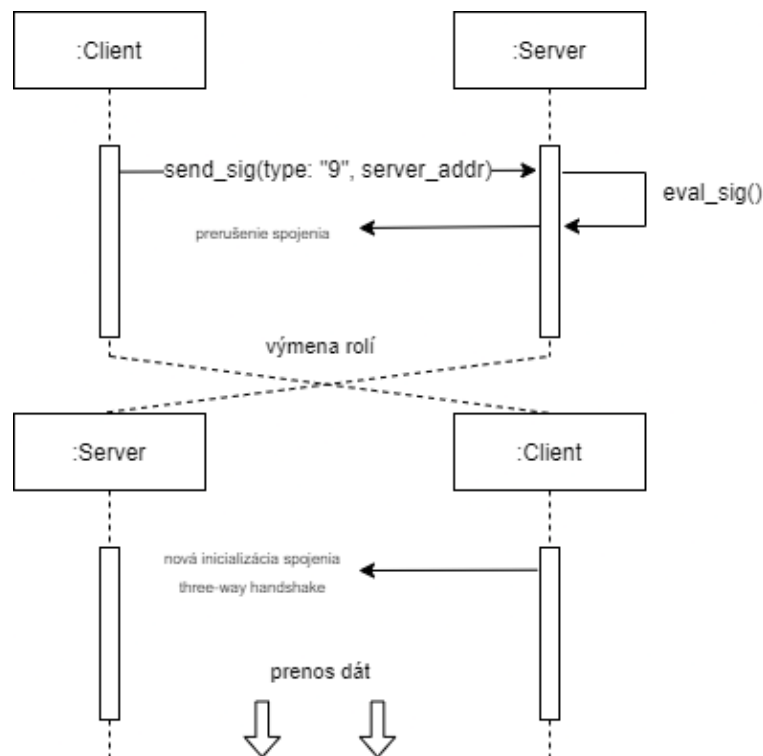
Rovnako, v prípade, ak server nedostane v očakávanej dobe 5 po sebe nasledujúcich paketov typu 3 zo strany klienta, považuje klienta za odpojeného a prestane počúvať, pričom dôležitý je fakt, že k prerušeniu počúvania dojde iba v prípadoch, keď bol klient reálne k serveru pripojený (ak server iba čaká na inicializáciu spojenia zo strany klienta, neprestáva počúvať).

2.3.6 Opis spôsobu zmeny úloh

Súčasťou programu je aj možnosť automatickej výmeny úloh medzi klientom a serverom. Proces výmeny úloh musí byť inicializovaný na strane klienta, pričom po jeho skončení sa z klienta stane server a zo servera klient. Proces prebieha nasledovne:

- Používateľ na strane klienta v menu vyberie možnosť *Change mode*.
- Na server sa odošle paket typu 3 – *SWICH* a očakáva sa odpoveď.
- Zo strany servera dorazí paket typu 4 – *FIN* a tromi krokmi sa oficiálne ukončí spojenie
- Následne sa vykonávanie programu na oboch stranách vráti do *while* cyklu v *main.py*, kde sa uloží nová hodnota do premennej *entity_type*, podľa ktorej sa v novej iterácii cyklu zvolí správna možnosť (v tomto prípade zmena úloh).
- Dôležité údaje, ako IP adresa a port servera sú neustále uložené v globálnych premenných, kvôli čomu ich po výmene úloh používateľ nebude musieť manuálne zadávať.
- Po inicializácii dvoch nových entít (klienta a servera) dojde k ich automatickému spojeniu využitím *three-way handshake*.
- Účastníci komunikácie sú pripravení na prenos dát.

Základnou úlohou inicializačnej strany je doručiť druhému účastníkovi paket typu 9 – oznámenie o zmene rolí, pričom rozdiel oproti iným druhom komunikácie je, že sa tu neočakáva odpoveď druhej strany vo forme paketu typu 2 – *ACK*. Po odoslaní paketu si klient automaticky zmení svoju rolu, pričom akonáhle druhej strane dorazí paket typu 9, aj ona vykoná rovnaký krok.



Obr. 2.5 Sekvenčný diagram výmeny úloh

2.4 Zmeny oproti návrhu

V porovnaní s originálnym návrhom programu a jeho funkčnosťou nastalo niekoľko zmien, žiadna z nich však nie je príliš rozsiahla. Zoznam zmien sa nachádza nižšie:

- Zmena veľkosti položky *Typ paketu* z 2 na 1 Byte v hlavičke
- Odstránenie položky *Veľkosť paketu* z hlavičky (dá sa vypočítať na strane servera)
- Pakety typu 3 – *keep-alive* sú odosielané len zo strany klienta na stranu servera

Pri prvej z uvedených možností bolo cieľom znížiť počet prenášaných dát, pretože údaje obsiahnuté v tejto položke hlavičky (číselné hodnoty od 0 po 9, predstavujúce typ signálu) je reálne možné prenášať (zakódovať) aj v 1 Byte.

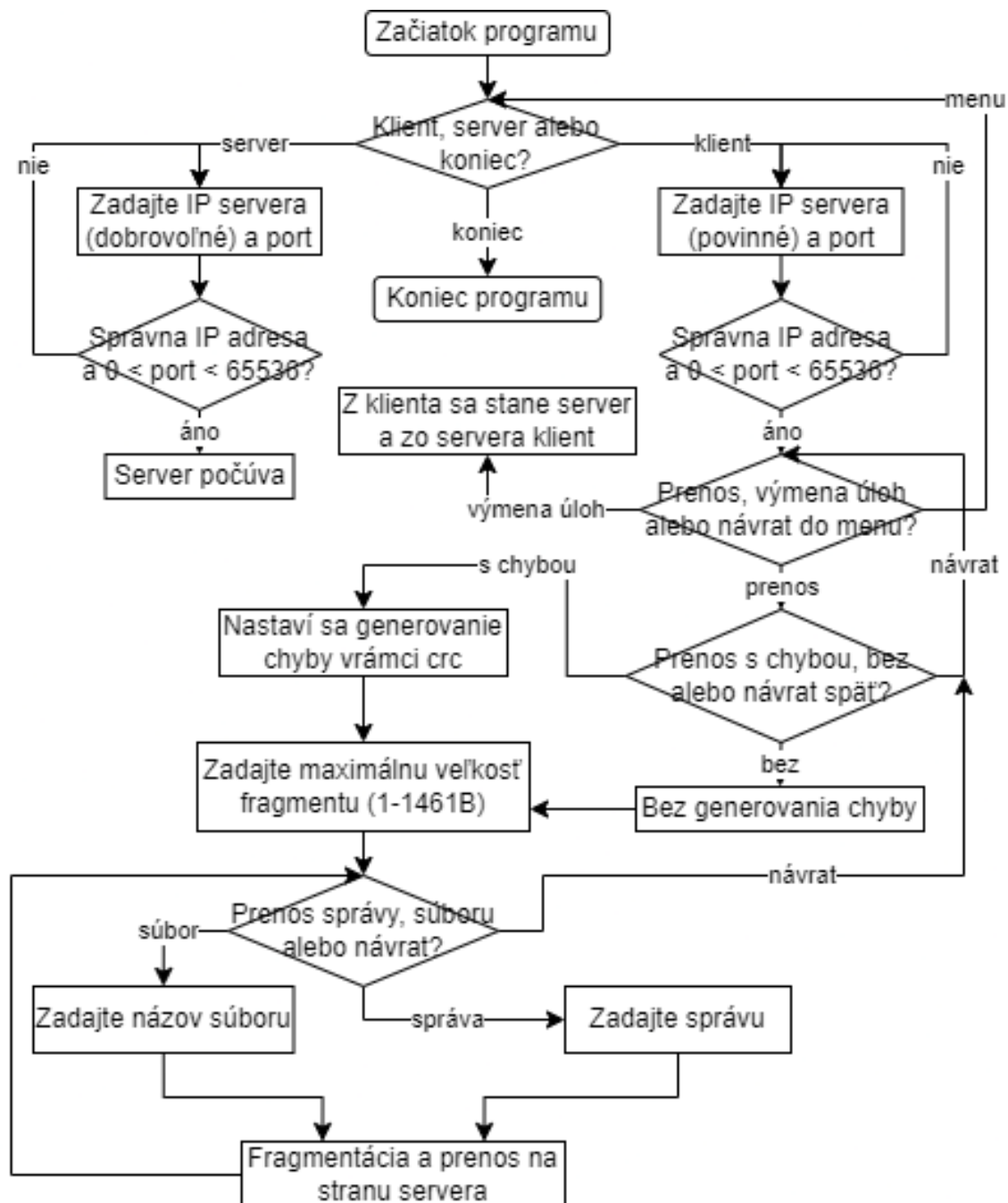
Výsledkom druhej zmeny bolo taktiež zníženie veľkosti prenášaných dát, pretože samotnú veľkosť paketu nie je potrebné odosielať na server (tento údaj je možné jednoducho určiť aj na strane servera výpočtom na základe prenesených dát).

Posledná, tretia zmena, týkajúca sa *keep-alive* paketov bola dôležitá z logických dôvodov. Po uskutočnení spojenia stačí, ak bude klient odosielať *keep-alive* pakety na stranu servera a server na ich prijatie reagovať odoslaním paketu typu 2 – *ACK*, pretože už v tomto prípade dokážu obe strany reagovať na prerušenie komunikácie (server dokáže prerušiť spojenie, ak sa mu v očakávanom časovom intervale nedoručí paket typu 3, a klient dokáže taktiež predčasne ukončiť komunikáciu, ak mu za vopred prednastavený časový interval po odoslaní *keep-alive* paketu nedorazí zo strany servera paket typu 2 – *ACK*).

3 Opis dôležitých častí programu

Nasledujúca kapitola sa venuje examinácii kľúčových súčastí programu, ako napríklad zabalenie a rozbalenie paketov, evaluácia signálov, doručovanie *keep-alive* signálov a odosielanie a prijímanie dátových paketov.

3.1 Diagram priebehu programu



Obr. 3.1 Diagram inicializácie oboch entít v programe

3.2 Zabalenie a rozbalenie paketov

Pod pojмами zabalenie a rozbalenie sa rozumie prevod údajov obsiahnutých v pakete do takého tvaru, v akom sa dokážu efektívne preniesť cez sieť od klienta k serveru (v jazyku Python prevod bežných dátových typov na typ *bytes* alebo *bytearray*) a následne konvertovanie týchto údajov späť do ich pôvodného tvaru.

Konvertovanie paketu na byty a jeho následné odoslanie rieši metóda *send_packet()* triedy *General*, ktorá na vstupe akceptuje typ odosielaného paketu, adresu prijímateľa, zoznam údajov odosielaných v pakete a taktiež informáciu, či sa v náhodných paketoch majú pred odoslaním vygenerovať chyby, alebo nie (posledné dve vymenované parametre sú dobrovoľné). Ak ide o jednoduchý signál (napr. *ACK*, *SYN*, *FIN*, *keep-alive...*), v pakete sa okrem zakódovanej hodnoty tohto signálu neprenášajú žiadne iné informácie. Ak sa ale jedná o pakety typu *DATA_INIT* alebo *DATA*, paket obsahuje všetky údaje uvedené v hlavičke, pričom hodnota *crc* sa vypočíta priamo v tejto metóde.

```
def send_packet(self, type: int, addr: tuple[str, int], data: list = list(), error: bool = False) → None:
    packet = bytearray()

    match type:
        case 0 | 1 | 2 | 3 | 4 | 5 | 6 | 9:
            packet = type.to_bytes(1, 'big')
        case 7 | 8:
            packet.extend(type.to_bytes(1, 'big'))
            packet.extend(data[0].to_bytes(2, 'big'))
            packet.extend(data[1].to_bytes(4, 'big'))
            packet.extend(data[2])

    crc_result = crc32(data[2])
    crc_result = crc_result + 1 if randint(0, 100) > 90 and error else crc_result

    packet.extend(crc_result.to_bytes(4, 'big'))

    self.entity_socket.sendto(packet, addr)
```

Obr. 3.2 Metóda *send_packet()* triedy *General*

Na strane servera rieši prevod prijatých byteov späť do ich originálneho tvaru metóda *decode_data_packet()*. Ako to už z jej názvu vyplýva, používa sa len na prevod paketov typu *DATA* alebo *DATA_INIT*, pretože, ako už bolo vyššie spomenuté, v prípade paketov iných typov sa prenáša iba typ signálu a jeho hodnota sa dekoduje priamo na mieste, kde sa využíva.

```
def decode_data_packet(self, packet: bytes) → list:
    decoded_packet = list()
    decoded_packet.append(int.from_bytes(packet[: 1], 'big'))
    decoded_packet.append(int.from_bytes(packet[1 : 3], 'big'))
    decoded_packet.append(int.from_bytes(packet[3 : 7], 'big'))
    decoded_packet.append(packet[7 : -4])
    decoded_packet.append(int.from_bytes(packet[-4 :], 'big'))
    return decoded_packet
```

Obr. 3.3 Metóda *decode_data_packet()* triedy *General*

3.3 Odosielanie a prijímanie dátových paketov

Odosielanie dátových paketov sa z logických dôvodov uskutočňuje iba na strane klienta, preto sa skupina metód *create_transfer_header()*, *create_fragments()* a *transfer_data()* nachádza iba v triede Client.

Úlohou prvej z menovaných metód je vytvorenie paketu typu 7 – *DATA_INIT* so všetkými potrebnými údajmi. Pri tomto pakete sa do políčka *poradové číslo* uloží počet fragmentov, ktoré má server očakávať, a do *dáta* sa vkladá v prípade textovej správy slovo „text“ a v prípade súboru názov aj s príponou.

```
def create_transfer_header(self, content: bytes, type: str, error: int = 2, path: str = "") → bool:
    transfer_id = random.randint(1, 65535)
    self.run_heartbeat = False
    size = len(content)
    fragments = ceil(size / self.max_frag_size)
    header = [transfer_id, fragments, str.encode(type)]

    self.send_packet(Signals.DATA_INIT.value, self.addresses, header)
    data, addr = self.entity_socket.recvfrom(self.buffer_size)

    if int.from_bytes(data[: 1], 'big') == Signals.ACK.value:
        print("client - beginning of file transfer")
        if not self.transfer_data(content, header, False if error == 2 else True):
            return False

    if type == "text":
        Interface.client_console_output(size, fragments, False)
    else:
        Interface.client_console_output(size, fragments, True, type, path)
    return True
```

Obr. 3.4 Metóda *create_transfer_header()* triedy Client

Druhá metóda *create_fragments()* rieši rozdelenie prenášaných dát v prípade, ak sú väčšie, ako používateľom nastavená maximálna veľkosť fragmentu. V rámci nej sa vytvoria jednotlivé pakety, pričom ku každému fragmentu dát je už v tomto prípade priložené aj identifikačné číslo paketu. Po dokončení fragmentácie metóda vracia zoznam paketov pripravených na prenos po sieti.

```
def create_fragments(self, content: bytes, header: list) → list:
    fragmented_packets = list()
    begin, end = 0, self.max_frag_size

    for index in range(0, header[1]):
        data = content[begin : end if index != header[1] - 1 else content.__len__()]
        fragmented_packets.append([header[0], index + 1, data])

        begin += self.max_frag_size
        end += self.max_frag_size

    return fragmented_packets
```

Obr. 3.5 Metóda *create_fragments()* triedy Client

Posledná metóda *transfer_data()* sa stará o samotný proces odosielania pripravených dát, pričom dokáže reagovať aj na problémy v prenose. Ak sa paket nesprávne doručí, zo servera sa vráti správa typu 6 – *ERROR* a klient došle identický paket znova. Rovnaký výsledok nastane, ak sa zo strany servera nevráti správa typu 2 – *ACK* potvrdzujúca správnosť prijatia paketu – po uplynutí časového intervalu sa paket automaticky odošle znova.

```

def transfer_data(self, content: bytes, header: list, error: bool) → bool:
    packets = self.create_fragments(content, header)

    for index, packet in enumerate(packets):
        counter = 0

        while True:
            self.send_packet(Signals.DATA.value, self.addresses, packet, error)

            try:
                self.entity_socket.settimeout(5.0)
                data, addr = self.entity_socket.recvfrom(self.buffer_size)
                self.entity_socket.settimeout(None)

                if int.from_bytes(data[: 1], 'big') == Signals.ACK.value:
                    print(f"client - delivered { index + 1 }/{ packets.__len__() } - { len(packet[2]) }B")
                    break
                elif int.from_bytes(data[: 1], 'big') == Signals.ERROR.value:
                    print("client - re-sending latest packet - error in data")
            except socket.error:
                counter += 1
                print("client - re-sending latest packet - no server response")

            if counter > 4:
                return False

    return True

```

Obr. 3.6 Metóda *transfer_data()* triedy *Client*

Prijímanie dátových paketov sa uskutočňuje na strane servera v rámci metódy *receive_data()*. Server z prvého paketu *DATA_INIT* pozná presný počet očakávaných dátových paketov a po prijatí každého z nich si vypočíta vlastné *crc*, ktoré porovnáva s *crc* uloženým v pakete. Ak sa nezhodujú, vyžiada si daný paket opäť, v opačnom prípade pripojí jeho obsah k premennej *received*. Je ošetrený aj prípad duplicitných paketov.

```

def receive_data(self, transfer_id: int, fragments: int) → tuple[bytearray, int]:
    received, length = bytearray(), 0

    for index in range(0, fragments):
        counter = 0

        while True:
            try:
                self.entity_socket.settimeout(5.0)
                data, addr = self.entity_socket.recvfrom(self.buffer_size)
                self.entity_socket.settimeout(None)
            except socket.error:
                counter += 1
                print(f"server - packet { index + 1 }/{ fragments } did not arrive")

            if counter > 4:
                return (received, 0)

            continue

            packet = self.decode_data_packet(data)

            if crc32(packet[3]) == packet[4] and packet[1] == transfer_id and index + 1 == packet[2]:
                print(f"server - packet { index + 1 }/{ fragments } correct - { len(packet[3]) }B")
                length += len(packet[3])
                self.send_packet(self.eval_sig(data[: 1]).value, addr)
                received.extend(packet[3])
                break
            elif crc32(packet[3]) == packet[4] and packet[1] == transfer_id and index + 1 > packet[2]:
                print(f"server - packet { index + 1 }/{ fragments } already received")
                self.send_packet(self.eval_sig(data[: 1]).value, addr)
            else:
                print(f"server - packet { index + 1 }/{ fragments } error in data")
                self.send_packet(Signals.ERROR.value, addr)

    return (received, length)

```

Obr. 3.7 Metóda *receive_data()* triedy *Server*

3.4 Evaluácia prijatých signálov

Na automatické určenie správnej odpovede pre každý prijatý paket bola zhotovená metóda *eval_sig()* triedy *General*. Štruktúra metódy je veľmi jednoduchá. V prvom kroku sa zistí hodnota signálu konvertovaním z byteov na typ integer. Následne sa už iba vyberie správny typ odpovede na daný signál (na každý signál sa odpovedá nejakým preddefinovaným spôsobom, napr. na signál typu 0 – SYN sa ako odpoveď očakáva signál typu 1 – SYN+ACK, na signál typu 6 – ERROR sa odpovie znovuodoslaním predchádzajúceho paketu a signálom typu 8 – DATA a podobne) a táto hodnota sa vráti späť.

```

def eval_sig(self, signal: bytes) → Signals:
    signal_type = int.from_bytes(signal, 'big')

    match signal_type:
        case 0:
            return Signals.SYN_ACK
        case 4:
            return Signals.FIN_ACK
        case 6:
            return Signals.DATA
        case 1 | 3 | 5 | 7 | 8:
            return Signals.ACK

```

Obr. 3.8 Metóda `eval_sig()` triedy `General`

3.5 Systém doručovania keep-alive signálov

Akonáhle sa vytvorí spojenie medzi klientom a serverom, klientska strana inicializuje odosielanie paketov typu 3 – *keep-alive* a doručuje ich na server v pravidelnom intervale 5 sekúnd, pričom server na ich príchod reaguje spätným odoslaním paketu typu 2 – *ACK*. Dôležitou poznámkou je, že odosielanie *keep-alive* paketov prebieha iba v prípadoch, keď sa práve neuskutočňuje žiadny prenos dát medzi koncovými uzlami (počas prenosu dát je pozastavené a odosielanie sa obnoví v okamihu dokončenia prenosu dát). Systém *keep-alive* paketov využíva multithreading na klientskej strane z dôvodu, aby používateľ dokázal interagovať s programom a nebol obmedzovaný formou zablokovania programu.

```

def init_keepalive(self) → None:
    def keep_connection(entity_socket: socket) → None:
        while True:
            if self.run_keepalive:
                try:
                    self.send_packet(Signals.KEEP_ALIVE.value, self.addresses)
                except OSError:
                    return
            try:
                entity_socket.settimeout(5.0)
                data, addr = entity_socket.recvfrom(self.buffer_size)
                entity_socket.settimeout(None)

                if int.from_bytes(data[: 1], 'big') == Signals.ACK.value:
                    self.keepalive_error_count = 0
                    sleep(5.0)
                except socket.error:
                    self.keepalive_error_count += 1

                if self.keepalive_error_count == 1:
                    print()
                    print("client - no keepalive response from server")

                if self.keepalive_error_count > 4:
                    print("client - aborted connection due to inactivity")
                    self.run_keepalive = False
            else:
                sleep(0.5)

    self.entity_thread = threading.Thread(target = keep_connection, args = [self.entity_socket])
    self.entity_thread.start()

```

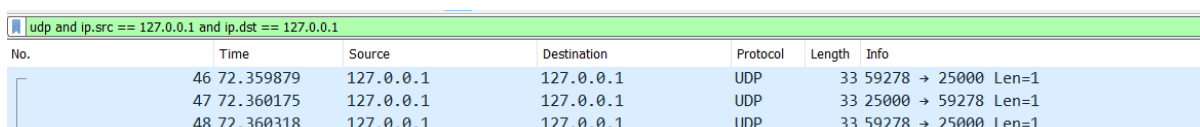
Obr. 3.9 Metódy `init_keepalive()` a `keep_connection()` triedy `Client`

4 Ukážky testovacích scenárov

V tejto kapitole sú podrobne analyzované testovacie scenáre pre odosielanie textových správ aj súborov, pričom sa kladie dôraz na celý priebeh komunikácie (vrátane inicializácie a prerušenia spojenia).

4.1 Prenos správy

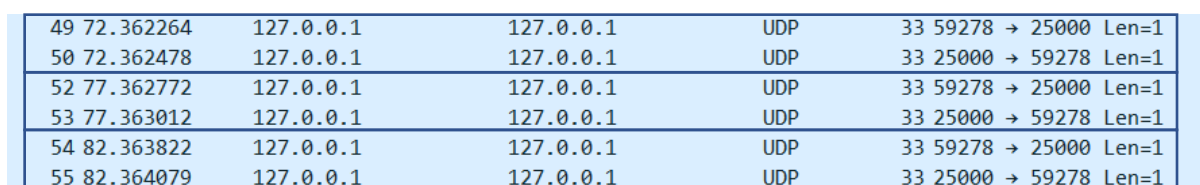
V programe Wireshark sa nastaví filter na protokol *UDP* spolu so zdrojovou a cieľovou IP adresou a v prvom kroku nastane inicializácia spojenia medzi klientom a serverom formou *three-way handshake*, ktorá je znázornená na obr. 4.1. Port servera je momentálne 25000.



No.	Time	Source	Destination	Protocol	Length	Info
46	72.359879	127.0.0.1	127.0.0.1	UDP	33	59278 → 25000 Len=1
47	72.360175	127.0.0.1	127.0.0.1	UDP	33	25000 → 59278 Len=1
48	72.360318	127.0.0.1	127.0.0.1	UDP	33	59278 → 25000 Len=1

Obr. 4.1 Three-way handshake s paketmi SYN, SYN+ACK a ACK

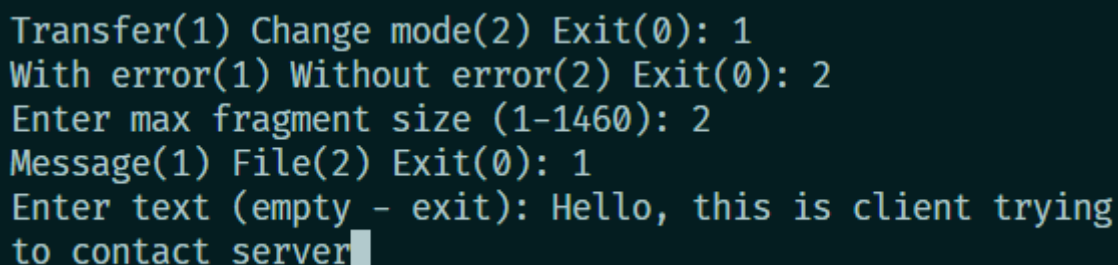
Po inicializácii nastane okamžité odosielanie paketov typu 3 – *keep-alive* klientom na stranu servera, pričom na každý z nich server odpovie paketom typu 2 – *ACK*. Na obr. 4.2 je možné tiež vidieť skutočnosť, že uvedené pakety sú odosielané každých 5 sekúnd.



49	72.362264	127.0.0.1	127.0.0.1	UDP	33	59278 → 25000 Len=1
50	72.362478	127.0.0.1	127.0.0.1	UDP	33	25000 → 59278 Len=1
52	77.362772	127.0.0.1	127.0.0.1	UDP	33	59278 → 25000 Len=1
53	77.363012	127.0.0.1	127.0.0.1	UDP	33	25000 → 59278 Len=1
54	82.363822	127.0.0.1	127.0.0.1	UDP	33	59278 → 25000 Len=1
55	82.364079	127.0.0.1	127.0.0.1	UDP	33	25000 → 59278 Len=1

Obr. 4.2 Odosielanie keep-alive paketov a prijímanie odpovede

Najprv zo strany klienta odošleme správu obsahujúcu text „Hello, this is client trying to contact server“ bez automatického generovania chýb a s veľkosťou fragmentu 2B. Správa obsahuje 46 znakov (každý znak predstavuje 1B, čiže celková veľkosť sa rovná 46B), takže so súčasnými nastaveniami sa správa doručí s použitím 23 fragmentov.



```
Transfer(1) Change mode(2) Exit(0): 1
With error(1) Without error(2) Exit(0): 2
Enter max fragment size (1-1460): 2
Message(1) File(2) Exit(0): 1
Enter text (empty - exit): Hello, this is client trying
to contact server
```

Obr. 4.3 Nastavenia odosielania textovej správy

Na obr. 4.4 je zobrazenie paketov prenosu textovej správy. Prvý zo zoznamu je paket typu 7 – *DATA_INIT* obsahujúci typ prenášaných dát (v tomto prípade obsahuje reťazec „text“). Na tento paket server odpovie signálom typu 2 – *ACK*, po doručení ktorého klient začína prenos jednotlivých fragmentov (páry označené modrou, kde prvý paket je vždy dátový a druhý potvrdzovací).

126	331.591575	127.0.0.1	127.0.0.1	UDP	47 51408 → 25000 Len=15
127	331.592167	127.0.0.1	127.0.0.1	UDP	33 25000 → 51408 Len=1
128	331.592577	127.0.0.1	127.0.0.1	UDP	45 51408 → 25000 Len=13
129	331.593865	127.0.0.1	127.0.0.1	UDP	33 25000 → 51408 Len=1
130	331.594496	127.0.0.1	127.0.0.1	UDP	45 51408 → 25000 Len=13
131	331.595199	127.0.0.1	127.0.0.1	UDP	33 25000 → 51408 Len=1
132	331.595640	127.0.0.1	127.0.0.1	UDP	45 51408 → 25000 Len=13
133	331.596284	127.0.0.1	127.0.0.1	UDP	33 25000 → 51408 Len=1
134	331.596713	127.0.0.1	127.0.0.1	UDP	45 51408 → 25000 Len=13
135	331.597359	127.0.0.1	127.0.0.1	UDP	33 25000 → 51408 Len=1

Obr. 4.4 Prenos textovej správy bez generovania chýb

Po správnom ukončení dátového prenosu sa reštartuje proces odosielenia *keep-alive* signálov a bude prebiehať do nového dátového prenosu alebo ukončenia komunikácie.

4.2 Výmena úloh

Výmena úloh sa inicializuje paketom typu 9 – *SWITCH*, ktorý je prvý spomedzi paketov na obr. 4.5. Nasleduje oficiálne prerušenie komunikácie (pakety č. 9, 10 a 11 – *FIN*, *FIN+ACK* a *ACK*) inicializované serverom. Po prerušení sa jednotlivé uzly prepnú do opačných režimov, ako prvý sa nastaví server a následne s ním nový klient automaticky spustí nadviazanie spojenia formou *three-way handshake* (pakety 12, 13 a 14). Po nich nasledujú dvojice *keep-alive* paketov spolu s odpoveďami *ACK*. Server sa teraz nachádza na porte 51409.

8	9.585758	127.0.0.1	127.0.0.1	UDP	33 51409 → 25000 Len=1
9	9.585980	127.0.0.1	127.0.0.1	UDP	33 25000 → 51409 Len=1
10	9.586140	127.0.0.1	127.0.0.1	UDP	33 51409 → 25000 Len=1
11	9.586249	127.0.0.1	127.0.0.1	UDP	33 25000 → 51409 Len=1
12	10.988404	127.0.0.1	127.0.0.1	UDP	33 25000 → 51409 Len=1
13	10.988721	127.0.0.1	127.0.0.1	UDP	33 51409 → 25000 Len=1
14	10.988911	127.0.0.1	127.0.0.1	UDP	33 25000 → 51409 Len=1
15	10.991245	127.0.0.1	127.0.0.1	UDP	33 25000 → 51409 Len=1
16	10.991520	127.0.0.1	127.0.0.1	UDP	33 51409 → 25000 Len=1
17	15.992833	127.0.0.1	127.0.0.1	UDP	33 25000 → 51409 Len=1
18	15.993233	127.0.0.1	127.0.0.1	UDP	33 51409 → 25000 Len=1
19	20.994171	127.0.0.1	127.0.0.1	UDP	33 25000 → 51409 Len=1
20	20.994415	127.0.0.1	127.0.0.1	UDP	33 51409 → 25000 Len=1
21	25.995365	127.0.0.1	127.0.0.1	UDP	33 25000 → 51409 Len=1
22	25.995607	127.0.0.1	127.0.0.1	UDP	33 51409 → 25000 Len=1
23	30.996328	127.0.0.1	127.0.0.1	UDP	33 25000 → 51409 Len=1
24	30.996574	127.0.0.1	127.0.0.1	UDP	33 51409 → 25000 Len=1

Frame 8: 33 bytes on wire (264 bits), 33 bytes captured (264 bits) on interface \Device\NPF_{Loopback}, id 0
 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
 User Datagram Protocol, Src Port: 51409, Dst Port: 25000
 Data (1 byte)
 Data: 09
 [Length: 1]

Obr. 4.5 Výmena úloh dvoch uzlov

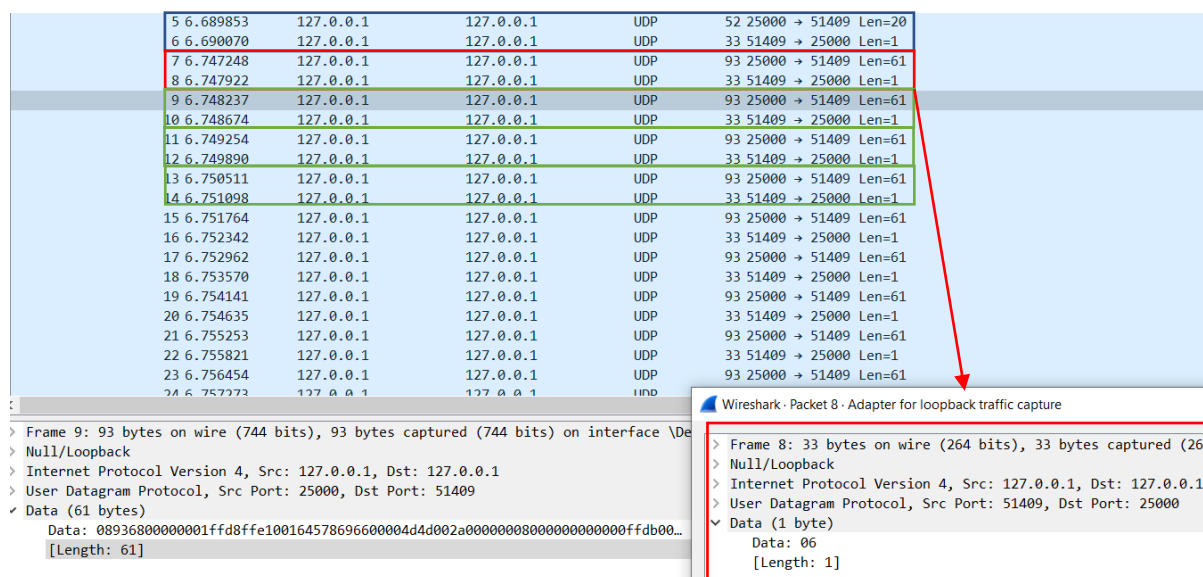
4.3 Prenos súboru

Po výmene úloh nasleduje prenos súboru s nastaveniami uvedenými na obr. 4.6. Do políčka pre absolútnu cestu k súboru sa nezadalo nič, aby došlo k prenosu predvoleného súboru z adresára *send*. Súbor sa posiela po fragmentoch veľkosti 50B. Počas prenosu sa taktiež budú generovať chyby v *crc* kódoch niektorých prenášaných paketov.

```
Transfer(1) Change mode(2) Exit(0): 1
With error(1) Without error(2) Exit(0): 1
Enter max fragment size (1-1460): 50
Message(1) File(2) Exit(0): 2
Enter file path (empty - sends predefined file):
```

Obr. 4.6 Nastavenia odosielania súboru

Inicializácia odosielania dát aj v tomto prípade začína paketom typu 7 – *DATA_INIT*, na ktorý server odpovedá paketom *ACK*. Následne nasleduje odosielanie dát vo dvojiciach paketov, buď *DATA – ACK* v prípade správneho doručenia paketu, alebo *DATA – ERROR* v prípade znovuvyžiadania dát. Dátový paket má veľkosť 61B (kombinácia 11B hlavičky navrhnutého protokolu s 50B dátami). Hneď v prvom doručenom pakete (č. 7 na obr. 4.7, červený rám) server detekoval chybu a preto naň odpovedal paketom typu 6 – *ERROR* (č.8). Po znovuodoslaní identického paketu v ňom už nebola nájdená žiadna chyba a ako odpoveď sa poslal paket *ACK*. Dvojice so správnym prenosom sú v zelenom ráme.



No.	Time	Source	Destination	Protocol	Length	Info
5	6.689853	127.0.0.1	127.0.0.1	UDP	52	25000 → 51409 Len=20
6	6.690070	127.0.0.1	127.0.0.1	UDP	33	51409 → 25000 Len=1
7	6.747248	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61
8	6.747922	127.0.0.1	127.0.0.1	UDP	33	51409 → 25000 Len=1
9	6.748237	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61
10	6.748674	127.0.0.1	127.0.0.1	UDP	33	51409 → 25000 Len=1
11	6.749254	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61
12	6.749890	127.0.0.1	127.0.0.1	UDP	33	51409 → 25000 Len=1
13	6.750511	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61
14	6.751098	127.0.0.1	127.0.0.1	UDP	33	51409 → 25000 Len=1
15	6.751764	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61
16	6.752342	127.0.0.1	127.0.0.1	UDP	33	51409 → 25000 Len=1
17	6.752962	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61
18	6.753570	127.0.0.1	127.0.0.1	UDP	33	51409 → 25000 Len=1
19	6.754141	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61
20	6.754635	127.0.0.1	127.0.0.1	UDP	33	51409 → 25000 Len=1
21	6.755253	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61
22	6.755821	127.0.0.1	127.0.0.1	UDP	33	51409 → 25000 Len=1
23	6.756454	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61
24	6.757273	127.0.0.1	127.0.0.1	UDP	93	25000 → 51409 Len=61

Wireshark - Packet 8 - Adapter for loopback traffic capture

> Frame 9: 93 bytes on wire (744 bits), 93 bytes captured (744 bits) on interface \Device\NPF{...}

> Null/Loopback

> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

> User Datagram Protocol, Src Port: 25000, Dst Port: 51409

> Data (61 bytes)

Data: 0893680000001ffdbffe100164578696600004d4d002a0000000800000000000000ffdb00...

[Length: 61]

> Frame 8: 33 bytes on wire (264 bits), 33 bytes captured (264 bits) on interface \Device\NPF{...}

> Null/Loopback

> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

> User Datagram Protocol, Src Port: 51409, Dst Port: 25000

> Data (1 byte)

Data: 06

[Length: 1]

Obr. 4.7 Prenos súboru s generovaním chýb