Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Počítačové a komunikačné siete

Zadanie 2 - UDP komunikátor

Jakub Hrnčár
AIS ID: 110 800, ak. rok 2021/2022

1. Úvod do zadania

Zadanie má jednoznačný charakter: vytvoriť funkčný program na posielanie textových správ a súborov medzi dvoma PC. Nakoľko fungujeme v online režime, zadanie sa prezentuje iba skrz localhost.

Existuje viac algoritmov na posielanie dát, ja som zvolil ARQ metódu stop-and-wait, takže ďalšie dáta sa odošlú iba v prípade, že predošlé dáta sa riadne doručili.

Keďže vytvorený program bude slúžiť na komunikáciu medzi ľuďmi, je potrebné do istej miery optimalizovať aj užívateľský zážitok a interakciu človeka s programom.

Zadanie som vypracoval na Windows 10 v prostredí PyCharm, avšak narazil som na isté limitácie operačného systému a vývojového prostredia, o ktorých sa spomeniem neskôr.

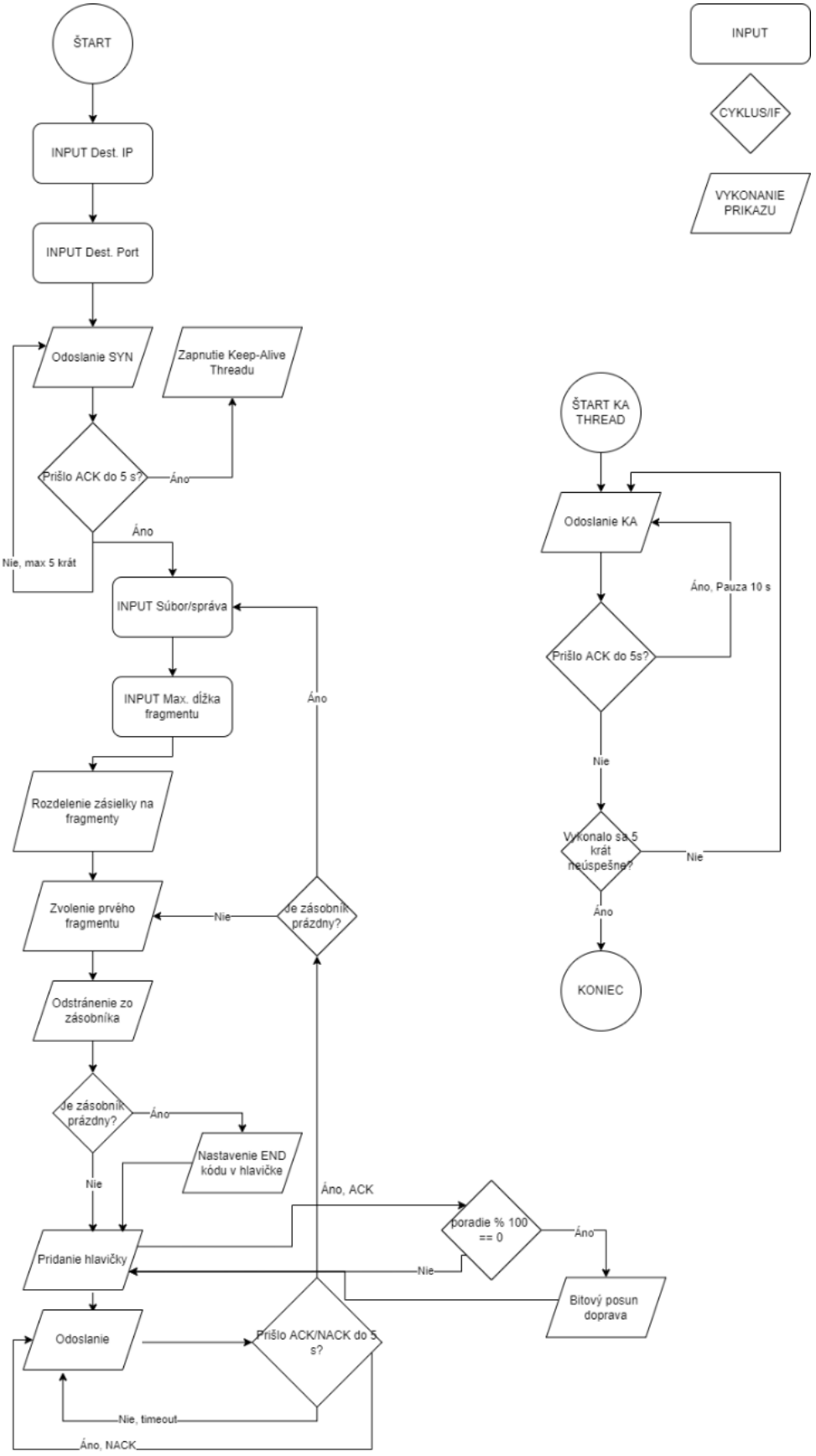
2. Zmeny oproti návrhu

Musel som vykonať malé množstvo zmien oproti navrhovanému riešeniu. Prvou zmenou je implementácia simulácie chyby, ktorú som vôbec nespomenul v návrhu. Chyba sa udeje bitovým posunom v hlavičke+dátovej časti fragmentu. Pri správach je každý 5. a pri súboroch každý 100. fragment chybný a teda bude na tieto fragmenty odpovedané NACK, ktoré vyžiada nanovo poslanie dát.

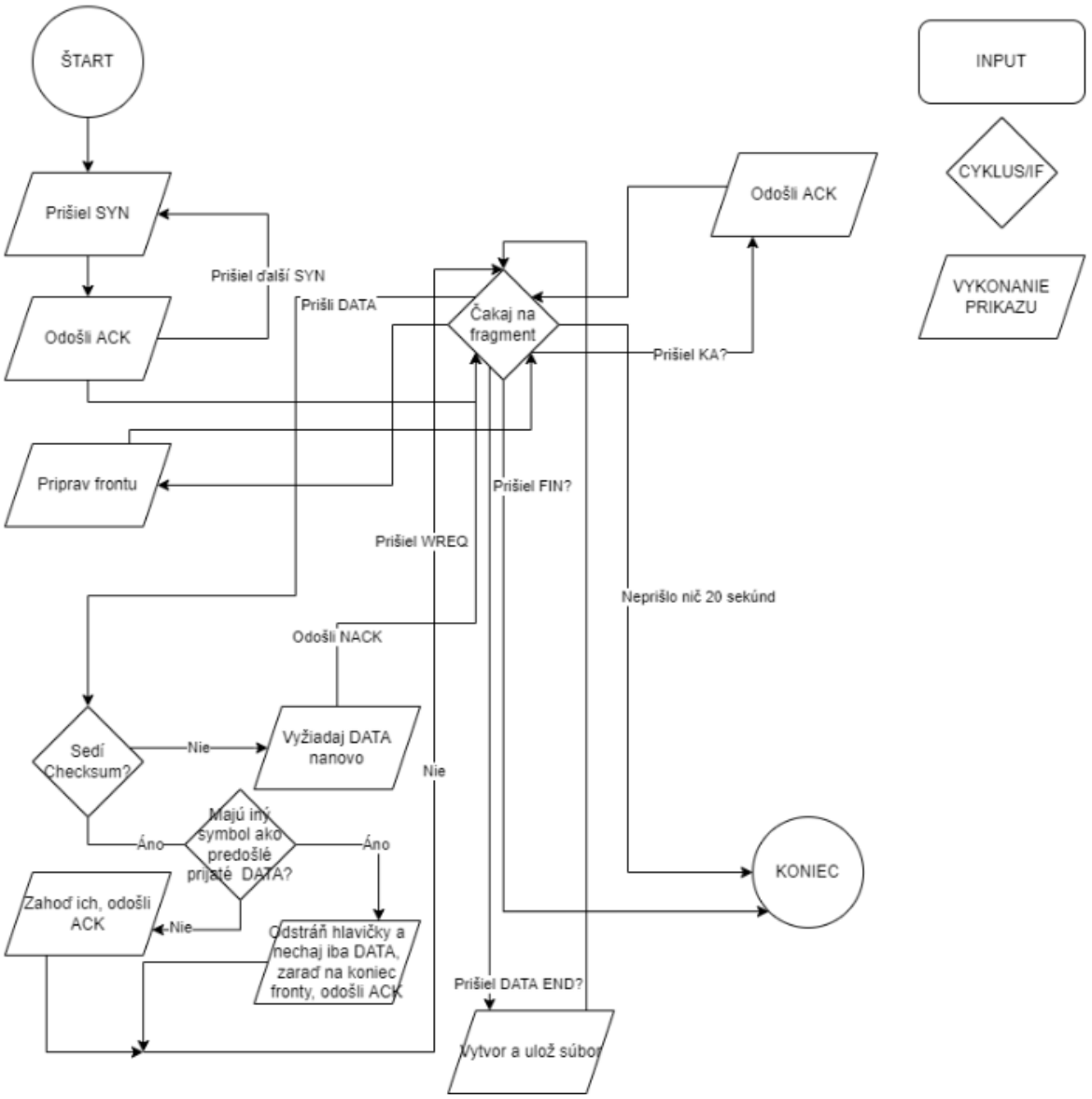
Keep-Alive thread sa zapína bezprostredne po nadviazaní komunikácie a prerušuje sa počas posielania súboru/správy, aby nenastali žiadne kolízne situácie, napríklad chybné prevzatie ACK správy na dáta vláknom keep-alive, čo spôsobí nutnosť opakovania poslania konkrétneho fragmentu. Taktiež, Keep-alive sa opakuje maximálne 5 krát a až potom vypne klienta, pričom nie je potrebné dávať Keep-Alive na strane servera do vlastného vlákna. Je to kvôli tomu, že keep-alive sa pauzuje počas odosielania správy/súboru.

Pridal som aj 20 sekundový časovač na strane servera, ktorý po jeho uplynutí vypne server a obnovuje sa prijatím akejkoľvek správy. Dôvod tejto zmeny je snaha o realistickú simuláciu servera. Ak sa dlhodobo na linke nič neprenáša, server nebude plytvať prostriedky na očakávanie komunikácie s neaktívnym klientom. Ďalej táto zmena pridáva na opodstatnenosti keep-alive funkcionality.

3. Blokový diagram celého programu



Obr.1 - Blokový diagram fungovania klient časti programu



Obr.2 - Blokový diagram fungovania server časti programu

4. Stručný opis algoritmov

Program je členený na veľa malých celkov, ktoré riešia jednotlivé problémy a aspekty komunikácie. Srdce programu tvorí samotná knižnica socket, ktorá vykonáva samotné posielanie dát po sieti. Knižnica sa inicializuje vytvorením objektu socket.

Nižšie uvedené metódy a funkcie sú spoločné pre server aj klient.

Metóda sendto()

```
socket.sendto(msg, (UDP_IP, UDP_PORT))
```

Metóda je zahrnutá v knižnici socket. V parametri msg sa nachádzajú bajty na odoslanie, takže v kontexte programu je to vždy

hlavička + dáta (ak sú). UDP_IP a UDP_PORT sú premenné, ktoré si nastaví užívateľ sám pri spustení programu.

Metóda recvfrom()

```
while data is None and tries < 5:
    socket.settimeout(5)
    try:
        tries += 1
        data, addr = sock.recvfrom(4096)
    except socket.timeout:
        sock.sendto(msg, (UDP_IP,
                          UDP_PORT))
        print("retrying")
```

Samotná metóda recvfrom() je zahrnutá v knižnici socket, vyžaduje parameter maximálnej veľkosti prijatej správy. Aby som si bol istý, že mi táto časť programu nebude spôsobovať problémy, nastavil som maximálnu hodnotu na 4096 bajtov, čo je viac ako dvojnásobok maximálnej hodnoty veľkosti fragmentu. Tento blok kódu sa nachádza vždy pri metóde recvfrom(). Spôsobuje timeout na metóde, čo zabezpečuje konečnosť programu. Ak by sa tento blok kódu v programe nenachádzal, metóda by čakala do nekonečna na novú správu. V tomto prípade je nastavený aj maximálny počet opakovaní pre čakanie na prijatie.

Niektoré často opakované a nemenné časti programu sa mi podarilo izolovať do funkcií:

Funkcia checksum()

Táto funkcia sa opakuje aj pre klienta aj pre server. Slúži na validáciu prijatého fragmentu. Vo funkcií používam (rovnako ako som spomenul v návrhu) CRC-16-CCITT, ktoré je v programe inicializované a uložené do globálneho objektu crc_caclulator.

```
check = crc_calculator.calculate_checksum(rest).to_bytes(2, "big")
```

Použitie CRC metódy je veľmi jednoduché, použijem metódu calculate_checksum, ktorá automaticky berie nastavený typ CRC; pošlem do jej parametru zbytok fragmentu bez začiatočného CRC v hlavičke. Metóda mi vráti číslo, ktoré konvertujem na 2 bajty. Následne

iba porovnám prijaté CRC s teraz vypočítaným CRC a vrátim True (rovná sa) alebo False (nerovná sa).

Funkcia decon_header():

Funkcia slúži na dekonštrukciu prijatej hlavičky na typ, kód a status. Je veľmi dôležitá, pretože túto operáciu je nutné vykonať pre každý fragment.

```
head = str(bin(int.from_bytes(data[2:3], "big")))[2::]
```

Fragment prijde v bajtoch, je teda nutné zobrať iba časť hlavičky bez CRC (preto 2:3 index), konvertovať bajty na integer, následne na binárne číslo a to následne na string, z ktorého odstránim formulku 0b, ktorá je hlavičkou pre binárne čísla. Ďalej iba na základe formátu hlavičky dekódujem typ, kód a status.

Funkcie pre klienta

establish()

Funkcia slúži na nadviazanie komunikácie so serverom, spustí sa automaticky po zadaní IP adresy a portu servera. Jej algoritmus zároveň slúži ako predloha pre zvyšné časti programu:

```
header = int("10011100", 2)  # Network, SYN, Status: 1, padding 00
header = header.to_bytes(1, "big")
check = crc_calculator.calculate_checksum(header).to_bytes(2, "big")
msg = check + header
sock.sendto(msg, (UDP_IP, UDP_PORT))
```

Na základe formátu hlavičky vytvorím príslušný SYN fragment, ktorému vypočítam CRC a následne hlavičku spolu s CRC odošlem na server.

Ďalší krok je čakanie na prijatie odpovede vo forme Network ACK, na čo použijem algoritmus pre recvfrom() spomenutý vyššie.

```
valid = checksum(data)
if valid:
   frag_type, frag_code, frag_status = decon_header(data)
```

```
    if frag_type == 2 and frag_code == 2:
        print("Connection established")
        return True
```

Na koniec iba validujem prijatý fragment a ak prišlo Network ACK, môžem úspešne prehlásiť nadviazanie komunikácie.

send_file()

Funkcia je pomerne dlhá, avšak opakuje sa v nej viackrát ten istý algoritmus z funkcie establish(), pretože je nutné poslať samostatne až 3 druhy fragmentov: Write Request, DATA, DATA END, pričom všetko sa môže deliť na viac fragmentov.

Write Request posiela buď v jednom alebo vo viac fragmentoch meno súboru.

```
name_copy = name_copy.encode()
prepared = name_copy[0:frag_length]
name_copy = name_copy[frag_length::]
```

Metóda .encode() konvertuje string na bajty, ktoré následne budem môcť poslať po sieti. Keďže používateľ má možnosť si zadať maximálnu veľkosť fragmentu (1B - 1469 B), takže je potrebné celú zásielku (v tomto prípade string názvu súboru) fragmentovať na menšie časti. Tento algoritmus sa používa pri každom odosielaní.

Keďže používam stop-n-wait ARQ metódu, tak po odoslaní každého fragmentu čakám na prijatie ACK, v prípade NACK alebo timeout-u pošlem dané dáta ešte raz. Pri posielaní dát sa v hlavičkách strieda status správa medzi hodnotami 0 a 1. Toto opatrenie som navrhol kvôli strate ACK na dáta.

Po úspešnom odoslaní mena súboru sa rovnakým algoritmom posielajú dáta s jedným rozdielom: nachádza sa tu simulovanie chyby. Pri súboroch, každý 100. fragment bude mať chybné dáta. Chybu simulujem bitovým posunom časti dát a hlavičky (bez CRC). Nakoniec iba odošlem DATA END fragment, ktorý taktiež obsahuje dáta, ale server podľa neho vie, že neprijdu už ďalšie dáta a môže uzavrieť súbor a uložiť ho.

send_msg()

Algoritmus použitý na vykonanie tejto funkcie zostáva totožný; používam tu tie isté princípy, ktoré používam pri posielaní mena súboru v predošlej funkcií. Zmenou je simulácia chyby, ktorá sa vykoná na každom 5. fragmente, pretože predpokladám, že správy budú mať všeobecne menej fragmentov ako súbory.

Thread a objekt keep_alive

Celá funkcionalita služby keep-alive je zahrnutá v objekte keep_alive. Jeho atribúty sú stavy služby: Working (či pracuje) a Alive (či thread žije). Objekt sa vytvorí pri úspešnom nadviazaní komunikácie a vlákno sa vytvorí a zapne bezprostredne po nadviazaní komunikácie, aby som predišiel vypršaniu spojenia.

Objekt má jedinú metódu, ktorou je keep_alive(). V tejto metóde je vytvorená celá funkcionalita služby. Algoritmus je pomerne jednoduchý; využíva prvky už známe z predošlých funkcií. Mám definovanú špeciálnu hlavičku pre keep_alive, ktorú odošlem na server a čakám maximálne 5 sekúnd na odpoveď. Ak odpoveď nepríde, opakujem poslanie 4 krát. V prípade opakovaného neúspešného poslania sa komunikácia vypne a užívateľ bude vrátený do hlavného menu. Ak ale komunikácia príde, celý thread sa uspí po dobu 10 sekúnd, kedy sa cyklus bude opakovať.

```
except ConnectionResetError:
    print("\nKEEP ALIVE FAILED, ANY INPUT WON'T MATTER")
    self.alive = False
    sock.close()
    self.received = False
```

Tu vidíme zachytenie ConnectionResetError, ktorý nastane ak sa server odpojí z komunikácie a klient sa bude snažiť poslať niečo na už neplatný socket. je to WinError, takže na Linuxe by nemalo k spusteniu tejto podmienky nastať. Vidíme, že socket sa teda uzavrie aj zo strany klienta a thread sa vypne.

Podstatnou časťou logiky keep_alive je fakt, že je nutné vypnúť službu počas posielania súboru/správy, kvôli zamedzeniu kolíznych situácií (keďže na KA aj na DATA sa odpovedá taktiež správou ACK, ktorú môže KA prijať).

```
message = input("Message: ")
```

```
keep_alive.work = False
keep_alive_thread.join()
send_msg(fragment, message)
print("Sent successfully")
keep_alive.work = True
keep_alive_thread =
threading.Thread(target=keep_alive.keep_alive)
keep_alive_thread.start()
```

Tu vidíme ukážku hlavnej funkcie klienta. Po zadaní správy sa vypne práca v threade, ktorý sa ukončí (môže tu nastať čakanie, pretože thread môže akurát byť uspaný). Následne sa odošle správa a thread sa musí nanovo vytvoriť a zapnúť. Bohužiaľ, v mojej implementácií nie je možné thread nechať uspaný počas odosielania správy/súboru.

finish()

Veľmi jednoduchá funkcia, ktorá slúži len na upovedomenie servera, že môže prestať počúvať a vrátiť sa do hlavného menu, pričom klient vykoná podobne. Slúži na to kód FIN, ktorý sa maximálne 5 krát pokúsim odoslať na server, ak odosielanie bude neúspešné tak sa tiež klient skončí, ale server o tom nemusí vedieť.

Po úspešnom/neúspešnom ukončení sa klient vráti do hlavného menu, kde sa užívateľ môže rozhodnúť akú rolu chce zvoliť alebo či chce celý program vypnúť.

Funkcie serveru

V servery sa taktiež opakujú funkcie **checksum()** a **decon_header()**.

Send_ack()

Do funkcie vstupuje parameter, ktorý určuje, ktorý typ ACK sa odošle (network, file, msg). Ďalej funkcia len pripraví ACK a odošle ho na socket.

Send_nack()

To isté ako funkcia send_ack(), avšak odosiela sa NACK, ktoré nemá typ, pretože z chybnej správy neviem správne získať typ, na ktorý by som odpovedal.

Server()

Main funkcia programu. Jej srdcom je recvfrom() zo socketu, ktoré má nastavený 20 sekundový timeout, po ktorom sa program kompletne resetuje do hlavného menu; spojenie zanikne. Tento odpočet ale začne vstupovať do platnosti iba v prípade nadviazaného spojenia.

Pred vykonaním ktorejkoľvek inej časti programu je nutné komunikáciu nadviazať. Stav či je alebo nie je nadviazaná uchovávam v premennej established; mení sa prijatím SYN a odoslaním ACK.

Súbor

Akonáhle je nadviazané spojenie, server sa riadi tým, čo mu príde a iba odpovedá na prijaté fragmenty. Na začiatku prijímania súboru je nutné prijať Write Request-y, ktorý nesie názov prijatého súboru. Vytváram tu buffer, do ktorého priebežne ukladám nové dáta názvu (tento algoritmus používam aj pri prijímaní správy/ dát súboru). To, či sú dáta nové alebo nie kontrolujem statusom fragmentu, ktorý sa musí striedať medzi 0 a 1. Akonáhle prídu 2 tie isté čísla po sebe, viem, že dáta už prišli a stačí na ne iba odoslať ACK, nemusím ich ukladať.

Prvý fragment dát spôsobí vytvorenie súboru s prijatým menom a začnú sa rovnakým spôsobom prijímať bajty súboru.

Na koniec príde posledný dátový fragment, ktorý má špecifický kód DATA END. Prijatím takéhoto fragmentu môžem bajty súboru uzavrieť a súbor pod prijatým menom a koncovkou uložiť a vypísať štatistické údaje o prenose.

Správa

Prijatie správy sa líši stratou nutnosti prijať Write Request, pretože správa nemá názov. Akonáhle zachytím prvý fragment typu msg, pripravím buffer a pripočítavam do neho nové prijaté dáta. Prijatím DATA END sa buffer uzavrie, konvertuje na string pomocou .decode() a vypíše, spolu s rôznymi štatistickými údajmi o prenose.

NACK

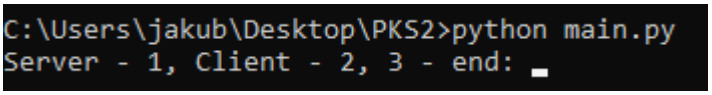
Po prijatí ktoréhokoľvek fragmentu sa kontroluje pomocou funkcie checksum() správnosť prijatého fragmentu. Pokiaľ je nesprávna, server nezaujíma akého typu správa je, odošle na ňu NACK.

Keep-Alive, FIN

Posledné typy sú administračné. Keep-Alive fragment bez dát, ktorý iba obnovuje časovač timeout-u. FIN hovorí serveru aby uzavrel socket, zrušil cyklus počúvania a vrátil sa do hlavného menu.

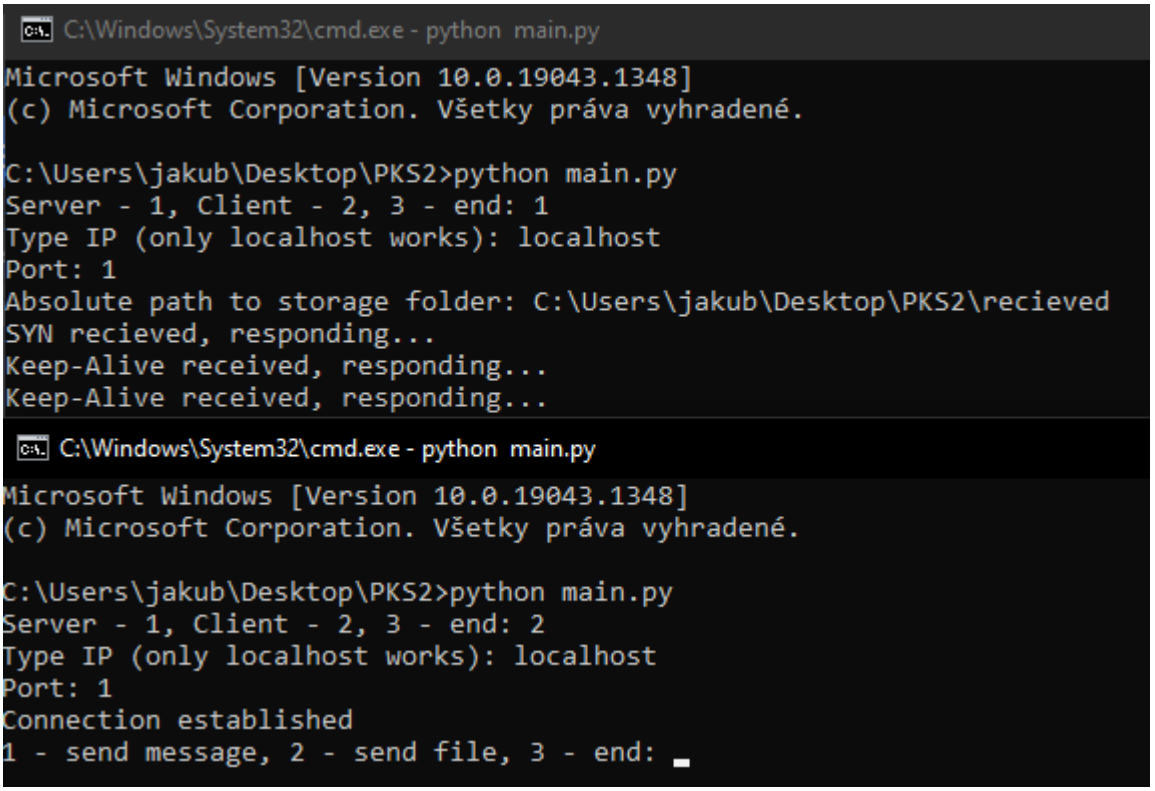
5. Spustenie programu

Užívateľské prostredie



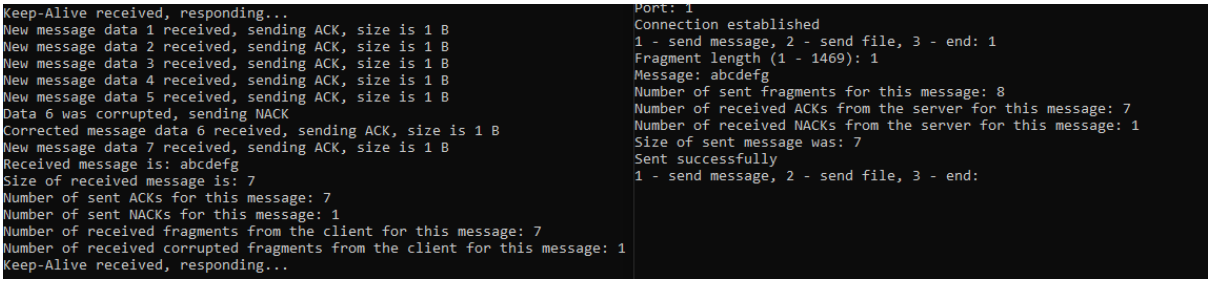
Obr.3 - Spustenie programu

Program sa spúšťa cez súbor main.py, ktorého je nutné spustiť na tom istom zariadení 2 inštancie. **Najprv sa zapína server, až potom klient.**



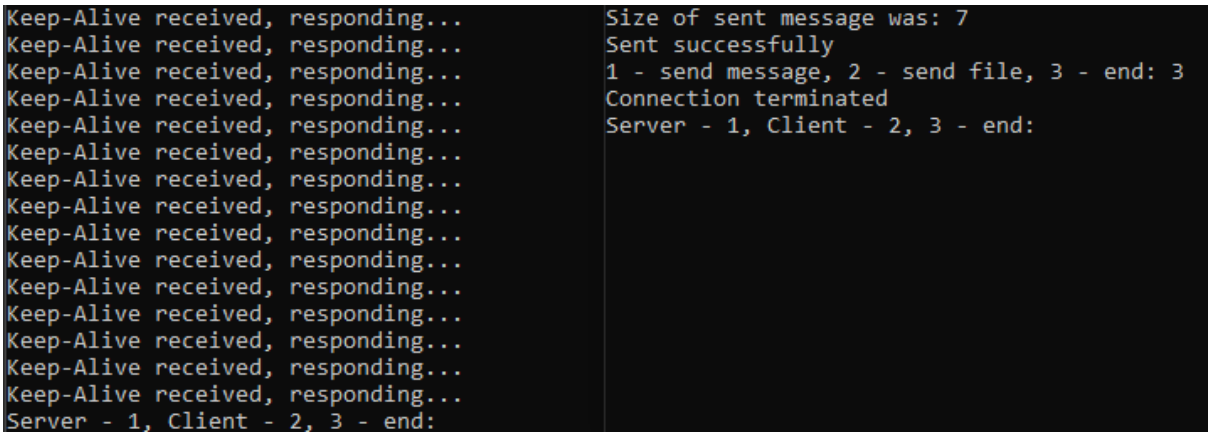
Obr.4 - Spustenie 2 inštancií a nadviazanie komunikácie

Obe strany programu si najprv vypýtajú IP adresu a port spojenia. Server ešte potrebuje priečinok, do ktorého bude vkladať prijaté súbory. Automaticky po zadaní potrebných údajov na oboch stranách nastane nadviazanie komunikácie a spustí sa Keep-Alive, ktorý sa opakuje každých 10 sekúnd.



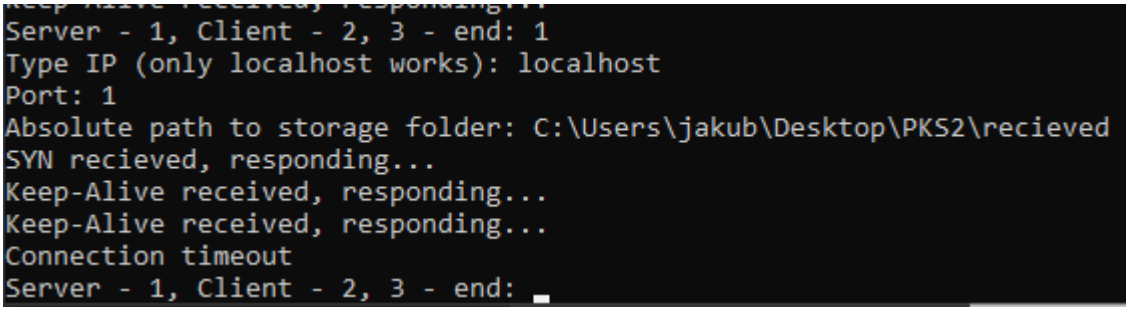
Obr.5 - Demonštrácia odoslania správy

Klient si po zadaní možnosti pre správu vypýta maximálnu veľkosť fragmentu (ak bude väčšia ako maximálna, tak ju zaokrúhli na maximálnu 1469 B). Na oboch stranách sa vypíše po úspešnom odoslaní správa o štatistických údajoch prenosu: Koľko bolo ACK, NACK, fragmentov a chybných fragmentov. Na servery vypisujem aj pre každý fragment údaje o ňom, vrátane jeho poradia.



Obr.6 - Ukončenie/SWAP

Po zadaní ukončenia sa spojenie uzavrie na oboch stranách, čo spôsobí návrat do hlavného menu.



Obr.7 - Server timeout

Ak sa vypne klient, ale server ostane zapnutý (alebo sa stratí veľké množstvo fragmentov), serveru vyprší po 20 sekundách spojenie, server sa dostane do hlavného menu.

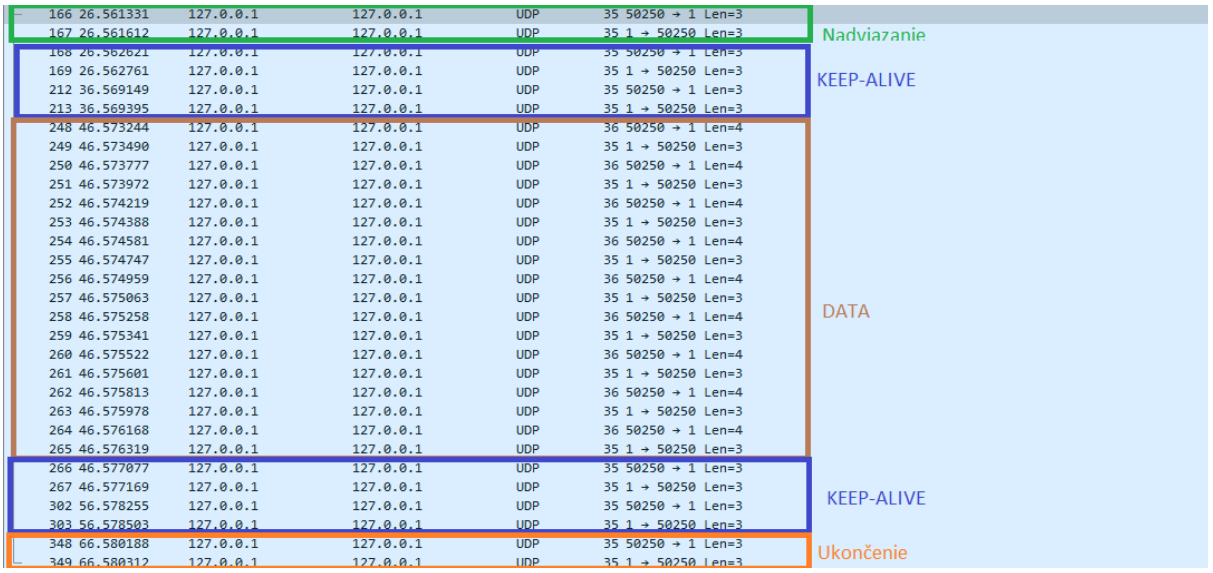
```
C:\Users\jakub\Desktop\PKS2>python main.py
Server - 1, Client - 2, 3 - end: 2
Type IP (only localhost works): localhost
Port: 1
Connection established
1 - send message, 2 - send file, 3 - end:
KEEP ALIVE FAILED, ANY INPUT WON'T MATTER
d
Resetting program, wait please...
Server - 1, Client - 2, 3 - end:
```

Obr.8 - Klient KA dead

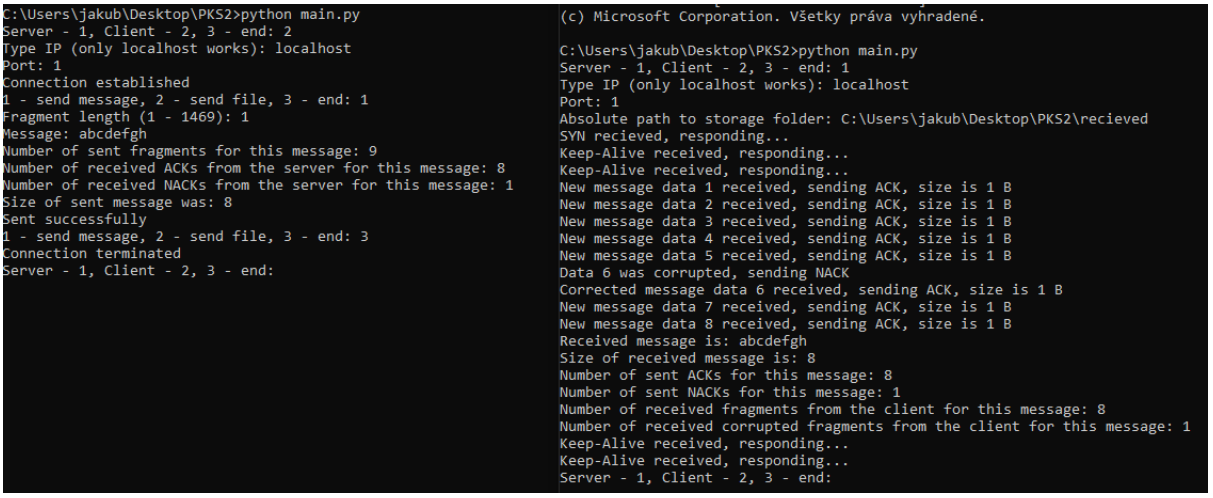
Ak sa server vypne, Keep-Alive ohlási stratu spojenia, čo vypne spojenie a program, vráti sa do hlavného menu. Toto nie je úplne správne riešenie, malo by to 5 krát opakovať odoslanie Keep-Alive, avšak po odpojení servera sa vyšle ICMP správa na server, ktorá vráti timeout, ktorý následne zapne Windows Error ConnectionResetError. Nazdávam sa, že na Linuxe tento problém nenastane. Ak chcem otestovať opakovanie Keep-Alive, stačí najprv zapnúť klient a potom server.

Wireshark

Funkčnosť komunikátora je možné overiť pomocou programu Wireshark. Pre demonštráciu pošlem cez komunikátor správu *abcdefgh* s veľkosťou fragmentu 1 B, pričom najprv je nutné nadviazať komunikáciu a po poslaní je nutné spustiť Keep-Alive a neskôr program riadne vypnúť a odpojiť spojenie.



Obr.9 - Wireshark vizualizácia



Obr.10 - CMD vizualizácia

6.Záver

Program bol vypracovaný vo vývojom prostredí PyCharm a Windows 10, avšak spúšťam ho cez CMD, kvôli rýchlejšiemu prenosu. Program som testoval na localhost, avšak bude fungovať aj na riadnej IP adrese. Zadanie je napísané v Python 3.9, s použitými knižnicami CRC, Socket a Threading.

Myslím si, že toto zadanie bolo jedno z najzaujímavejších zadaní na škole. Začal som si kvôli nemu uvedomovať, ako podstatné je TCP pre posielanie súborov a správ. Som spokojný s výsledkom, program je funkčný s pomerne veľkými optimalizáciami. Výstup je prehľadný a transparentný, užívateľ v ňom nájde všetko čo potrebuje.