

Počítačové a komunikačné siete

## **Zadanie 2: komunikácia s využitím UDP protokolu**

Petra Miková

## Contents

Návrh riešenia.....	1
Vlastná hlavička .....	1
Sekvenčný diagram.....	2
Slovný opis fungovania programu .....	3
Server.....	3
Klient.....	3
Komunikácia .....	3
Keepalive (udržiavanie spojenia).....	3
ARQ metóda Selective Repeat.....	3
Checksum .....	4
Opis častí zdrojového kódu.....	4
Zmeny oproti návrhu.....	5
Sekvenčný diagram .....	<a href="#">5s</a>
Opis implementácie .....	6
Reprezentácia hlavičky (paketu).....	6
Client side .....	6
Server side .....	7
Pomocné funkcie .....	7
Použité knižnice .....	7
Testovací scenár.....	8
Wireshark .....	8
Záver .....	10

## Návrh riešenia

### Vlastná hlavička

Flag	Počet fragmentov	Poradie	CRC	Dáta
1B	3B	3B	4B	
Dáta				Názov súboru
<1461B				premenlivé

**Flag (1B):** udržiava informáciu o odosielanom packete

- 1: nadviazanie spojenia
- 2: keep alive
- 3: prenos dát (správa)
- 4: prenos dát (súbor)
- 5: správny packet s dátami
- 6: nesprávny packet s dátami (chyba v CRC)
- 7: ukončenie spojenia

**Počet fragmentov (3B):** udržiava informáciu o počte packetov s dátami, ktoré sa budú posilať  
Keďže chceme vedieť posilať 2MB súbor, musíme pre toto pole v hlavičke mať takú veľkosť, aby to bolo možné, aj keby klient chce dáta fragmentovať po jednom byte. Ak by to takto bolo, na prenesenie 2MB súboru by sme potrebovali  $2^{21}$  fragmentov – z čoho vyplýva že pre toto pole potrebujem 3B.

Toto pole je inicializované iba v inicializačnom packete o tom že sa budú prenášať dáta.

**Poradie (3B):** udržiava informáciu o poradí packetu odosielaného fragmentu

Toto pole je inicializované len pri prenose packetov s dátami.

**CRC (checksum) (4B):** udržiava informáciu o zvyšku po delení CRC metódou

Za použitia CRC-32 algoritmu bude táto hodnota o veľkosti 4B, takže toľko dáme aj pre toto pole do hlavičky.

**Dáta (<1461):** udržiava samotné odosielané dáta

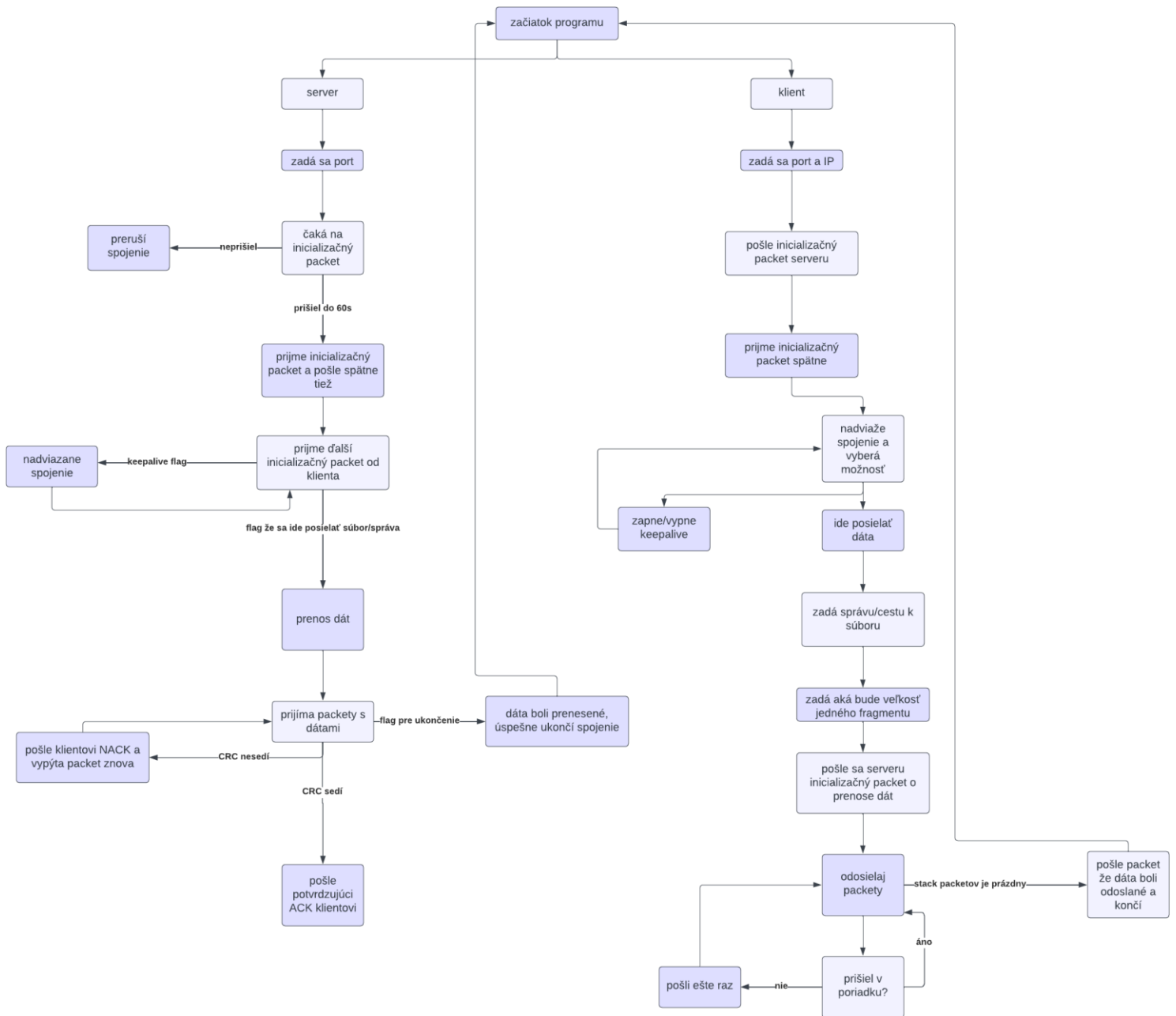
Keďže nechceme, aby došlo k fragmentácii na linkovej vrstve, maximálna veľkosť dát (teda jedného fragmentu) v packete musí byť 1500B – 20B (IP hlavička) – 8B (UDP defaultná hlavička) – 11B (moja hlavička) = 1461B. Tých 1500B je hodnota, ktorá predstavuje veľkosť dátovej časti v Ethernete.

**Názov súboru:** udržiava názov súboru ktorý sa bude posilať

Toto pole je inicializované len pri inicializačnom packete o tom že sa bude odosielať súbor.

## Sekvenčný diagram

V tejto časti ukážem sekvenčný diagram fungovania programu. Na oboch uzloch je taktiež užívateľ schopný spraviť zmenu rolí, tam teda proces po výmene zostáva rovnaký. Tento proces fungovania sa samozrejme pri finálnej verzii môže zmeniť.



## Slovný opis fungovania programu

### Server

Vytvorím si server socket a nabindujem doň zadaný port. Zároveň si zadefinujem timeout, dokedy čakám na inicializáciu spojenia zo strany klienta. Ak od klienta príde inicializačný packet na nadviazanie spojenia, prijem ho a pošlem mu ho späťne taktiež. Ak tento packet do 60 sekúnd nepríde, uzatvárať spojenie s chybou nadviazania spojenia. Inak teda čakám na ďalší inicializačný packet od klienta, kde nám môže prísť buď požiadavka o keepalive, zmena funkcie, ukončenie, alebo teda už samotný prenos dát. Ak sme dostali ukončovací packet, uzavrieme spojenie.

### Klient

Vytvorím si klient socket a vložím doň zadaný port a IP adresu. Pošlem serveru inicializačný packet a späťne prijem taktiež jeden od serveru, kde čakám na inicializáciu z jeho strany. Ak k nej teda došlo, nadväzujem spojenie a to zaslaním inicializačného packetu s možnosťami, ktoré som už spomenula vyššie. Ak teda iniciuje klient prenos dát, začína komunikácia.

### Komunikácia

Od klienta sa vypýta aký typ dát chce odosielať (správa/súbor), samotné dáta, a akú veľkosť fragmentu v jednom packete chce odosielať. Taktiež má možnosť si vybrať, či chce do prenosu zahrnúť aj chybné packety.

#### 1. Posielanie dát ARQ metódou Selective Repeat (client side)

V loope, ktorý končí až keď máme zásobník packetov prázdny, posielam fragmenty. Pre každý packet s fragmentom sa vypočíta CRC a posielam sa tento dátový packet serveru. Na základe odpovedí od serveru kontrolujeme, či všetky packety došli v poriadku, a ak nie, tak tie chybné posielame znova. Ak sa všetko úspešne odoslalo, pošleme serveru inicializačný packet o ukončení komunikácie. Metóda Selective Repeat bude opísaná hlbšie vo vlastnom odseku.

#### 2. Počúvanie dát zo strany servera (server side)

Znova v loope počúva a prijíma od klienta packety s dátami, kde rozhoduje či sú dobré alebo chybné (na základe toho že si na svojej strane vypočítava CRC a porovnáva ho s tým prijatým. Ak bol packet dobrý, posielam potvrdzujúci packet o úspešnom prenose fragmentu (ACK), a ak nie, posielam packet o chybnom packete (NACK) a žiada opätovné zaslanie. Ak server prijme ukončovací packet, pošle taktiež jeden späťne a ukončuje komunikáciu.

### Keepalive (udržiavanie spojenia)

Ak si klient vyberie možnosť keepalive, bude sa každých 15 sekúnd posielat packet s flagom keepaliveu serveru, ktorý pre udržanie spojenia musí odpovedať naspäť. Ak odpoveď nedôjde, klient dedukuje ukončenie spojenia. Tak isto na strane servera, ako som už vyššie spomínala, mám nastavený timeout pri začiatku spojenia, kde ak do 60 sekúnd nedôjde zo strany klienta inicializačný packet, server ukončuje spojenie. Spojenie môže do tretice zaniknúť ešte explicitným ukončením zo strany klienta.

### ARQ metóda Selective Repeat

Pri tejto metóde retransmittujeme len nesprávne packety, správne sa naďalej klasicky prijímajú a ukladajú do bufferu. Pre eventuálne prijatie fragmentovaných dát v správnom poradí si potrebujeme pre každý packet ktorý prenáša fragment dát udržiavať v hlavičke poradie fragmentu. Pri prijatí správneho packetu posielam server ACK, pri prijatí chybného posielam NACK, teda vieme jednoducho na strane klienta vidieť ktoré packety potrebujeme poslať znova. Samotná metóda funguje tak, že je zadefinované sliding window s určitou veľkosťou a poradie packetov usporiadane zostupne, a ak klient všetky packety z okna odoslal, server posielam ACK/NACK o tom packete, ktorý došiel ako prvý a

posúva okno o jeden index vľavo. Ak máme chybný packet, server pošle teda spomínaný NACK a klient pošle naspäť LEN tento jeden packet (na rozdiel od iných metód).

### Checksum

Checksum je v skratke kontrolný súčet packetu s dátami hovoriaci o tom, či nedošlo k chybe pri prenose. V mojej implementácii som sa rozhodla použiť metódu CRC. CRC pracuje na bitovej úrovni a využíva predom definovaný polynóm na vytvorenie checksumu takým spôsobom, že aj samotné dáta berie ako polynóm a týmto preddefinovaným ho vydolí. Modulo z tohto delenia sa ukladá ako checksum. Generuje sa z hlavičky a dát na strane klienta, ale aj na strane serveru, ktorý následne kontroluje a porovnáva svoj vypočítaný a prijatý checksum od klienta. Na základe tohto vie rozpoznať chybné packety. V implementácii budem chybné packety mockovať tak, že nejakým spôsobom pozmením výsledný checksum na strane klienta.

### Opis častí zdrojového kódu

Keďže je moja implementácia ešte len work in progress, priblížim aspoň jej hlavnú štruktúru. Na reprezentáciu svojej hlavičky používam triedu custom\_packet, v ktorej mám rovno aj metódu na spracovanie informácií do bytovej podoby. Celá implementácia je realizovaná v programovacom jazyku Python.

```
class custom_packet:
    def __init__(self, flag, number_of_fragments=None, fragment_order=None,
crc=None,
                data=None, filename=None):
        self.flag = flag
        self.number_of_fragments = number_of_fragments
        self.fragment_order = fragment_order
        self.crc = crc
        self.data = data
        self.filename = filename

    def __bytes__(self):
        flag_bytes = struct.pack('!B', self.flag)
        number_of_fragments_bytes = struct.pack('!BBB',
                                                self.number_of_fragments)
        if self.number_of_fragments is not None else b''
        fragment_order_bytes = struct.pack('!BBB', self.fragment_order) if
self.fragment_order is not None else b''
        crc_bytes = struct.pack('!I', self.crc) if self.crc is not None
else b''
        filename_bytes = self.filename.encode('utf-8') if self.filename is
not None else b''
        data_bytes = self.data if self.data is not None else b''

        return flag_bytes + number_of_fragments_bytes +
fragment_order_bytes + crc_bytes + filename_bytes + data_bytes
```

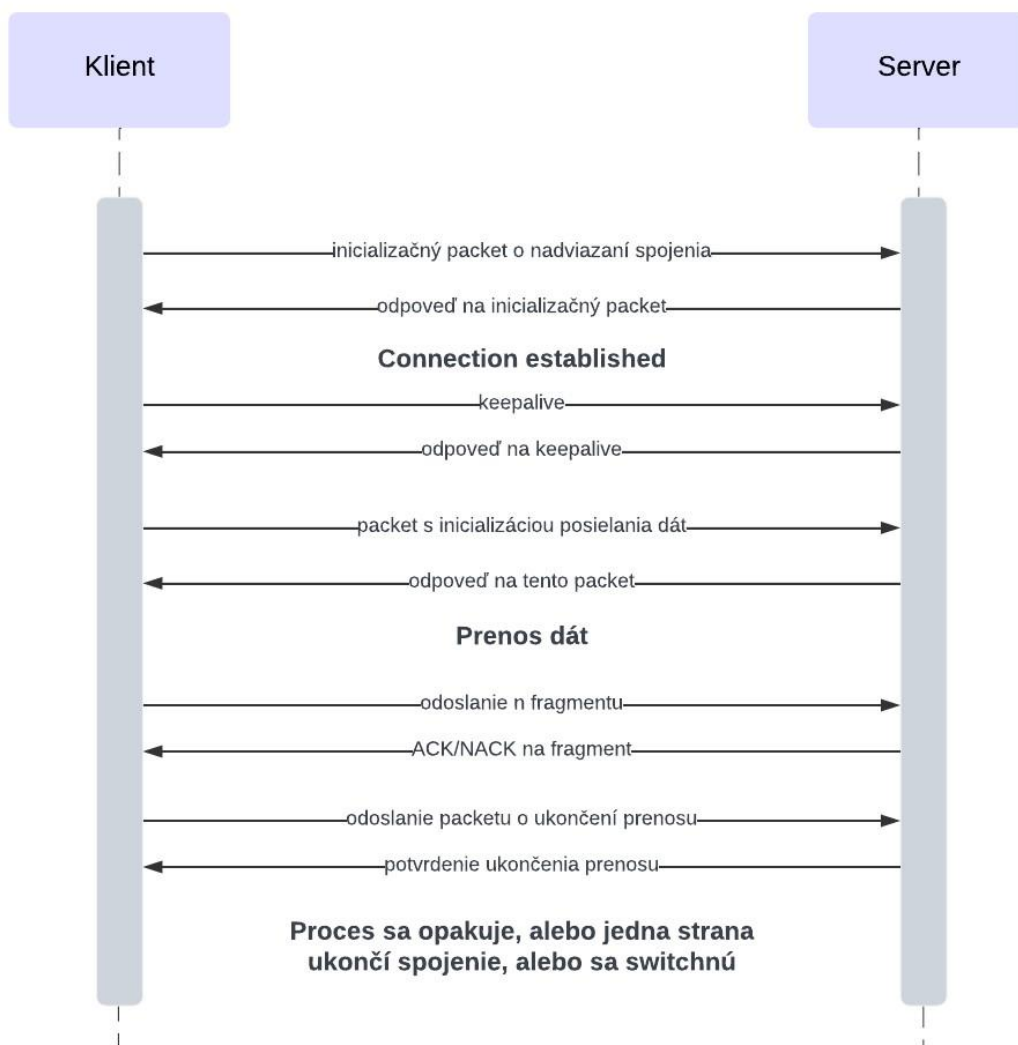
Čo sa týka zvyšku kódu, pri počiatočnom výbere klient/server sa spustí funkcia s obdobným menom ktorá je hlavná a v ktorej sa vykoná inicializácia spojenia. To ak je úspešné prechádza sa do podfunkcií ktoré už reprezentujú dekodovanie ďalších odoslaných packetov a následne aj funkcie pre prenos dát na oboch stranách. Z knižníc využívam pre hlavnú funkcionality socket. Podrobný opis metód a knižníc uvediem vo finálnej dokumentácii.

## Zmeny oproti návrhu

1. pridala som dva nové flagy a jeden pôvodný poupravila  
**7** – ukončenie prenosu dát  
Pridané:  
**8** – switch request  
**9** – ukončenie spojenia (vyžiadaním jednej zo strán)
2. v pôvodnom návrhu som mala keepalive odosielať každých 15 sekúnd, vo finálnej implementácii som to zmenila na **5 sekúnd** ako je v zadaní
3. pridaný správny **sekvenčný** diagram

## Sekvenčný diagram

V diagrame je zobrazený základný postup pri prenose dát. Ak ide o vykonanie switchu namiesto prenosu, tak sa z jednej strany pošle flag že sa má vykonať switch a druhá strana naň odpovedá. Tak isto ak chce jedna strana explicitne skončiť, pošle sa z jej strany packet s flagom o ukončení a druhá strana to prijme a skončí tiež. Prenos dát pomocou mnou zvolenej metódy ešte bude vyobrazený vo vlastnej časti.



## Opis implementácie

### Reprezentácia hlavičky (packetu)

Na reprezentáciu využívam classu `custom_packet`, ktorá už bola zobrazená v návrhu.

### Client side

Opis toho, aká logika sa využíva na strane klienta, už bol spomínaný v návrhu riešenia a táto logika sa nemenila. Opíšem teda aké funkcie vo svojej implementácii využívam a na čo slúžia.

#### 1. Funkcia `client`

Do tejto funkcie sa dostávame hneď po výbere možnosti z menu. Zadáva sa tam požadovaný port a IP a establishuje sa spojenie so serverom.

#### 2. Funkcia `client_after_init`

V tejto funkcii sa zapne `keepalive` až do doby pokým sa užívateľ nerozhodne poslať dáta. V tomto prípade sa odošlú dáta a `keepalive` sa po prenose znova naspäť zapne. Užívateľ má viacero možností: poslať správu, súbor, vykonať `switch`, alebo skončiť. Pri `switchi` sa zmena vykoná až keď server súhlasí taktiež, inak rola ostáva nezmenená.

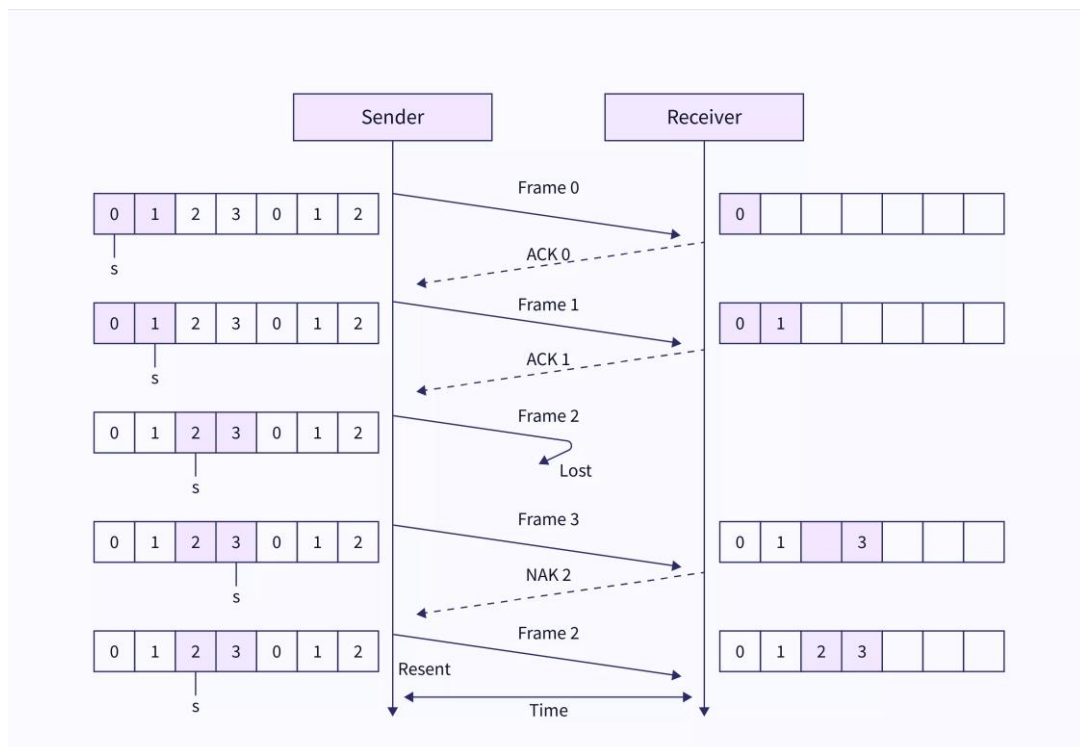
#### 3. Funkcia `data_process_client`

Tu sa zadáva veľkosť fragmentu a správa/nahrá sa súbor a vypočíta sa koľko fragmentov sa bude odosielať. Následne sa už prechádza do funkcie, kde sa celý prenos vykoná. Taktiež si vie užívateľ povedať, či a koľko chybných fragmentov chce odoslať.

#### 4. Funkcia `selective_repeat_arq`

V tejto funkcii sa vykonáva teda už samotný prenos dát pomocou metódy `Selective Repeat` s veľkosťou okna 4, ktorá sa samozrejme prispôsobí ak je fragmentov menej.

Na ukážku, `Selective Repeat` funguje takto:



Zdroj: <https://www.scaler.com/topics/computer-network/selective-repeat-arq/>



Prvotne sa teda zo strany klienta odšlú všetky fragmenty z okna, a po tomto sa okno o jedno posunie a už sa od servera očakáva ACK/NACK z úplne prvého odoslaného fragmentu. Teda napr sa odošlú prvotne 4 fragmenty s indexom 0,1,2,3 a server odpovedá na fragment 0 a následne sa okno posunie o jedno a klient odošle fragment s indexom 4 a očakáva odpoveď na 1tku. Ak dostane NACK, tak pošle hneď znova ten fragment ktorý bol chybný. Toto sa deje až dokým nám nedôjdu fragmenty na odoslanie, tie keď dôjdu tak už sa len počúva a prijímajú sa ACKY/NACKY, resp ak tuto dôjde NACK tak sa pripíše chybný fragment znova do buffera a pošle sa znova až dokým buffer fragmentov na odoslanie nie je prázdny. V tejto funkcii sa taktiež počíta CRC na celý packet, kde ak chceme poslať chybný fragment, vypočítané CRC sa vydeli dvoma.

#### 5. Funkcia flag\_check

Ide už len o pomocnú funkciu pre prenos dát kde sa handlujú flagy z prijatých packetov od servera a teda vykonávajú sa potrebné kroky opísane v predošlom odseku podľa toho či došiel ACK alebo NACK.

### Server side

Tak ako pri klientovi, aj tu opíšem funkcie z mojej implementácie pre stranu servera.

#### 6. Funkcia server

Do tejto funkcie sa prechádza hneď z menu. Zadá sa port a program sa snaží nadviazať spojenie s klientom. Ak sa to podarí, prechádza program ďalej, inak končí.

#### 7. Funkcia server\_after\_init

Tu sa počúva výber od klienta alebo má server možnosť ukončiť spojenie. Ak nekončí, prijíma a posíla odpovede na keepalive od klienta a zároveň čaká či sa klient rozhodne poslať dáta, ukončiť spojenie, alebo vykonať switch. Ak sa idú prenášať dáta, server pokračuje do funkcie kde sa budú tieto packety s dátami počúvať.

#### 8. Funkcia data\_process\_server

Tu sa teda prijme koľko fragmentov má dôjsť, a dokým nemáme počet prijatých korektných fragmentov rovných tomuto číslu, dovtedy server počúva a posíla odpovede. Vo while loope sa teda prijme packet od klienta s dátami a spraví sa check CRCčka, či to sedí. Ak CRC sedí, fragment sa prijme a odošle sa ACK. Ak nie, odošle sa NACK s požiadavkou odoslať fragment znova. Ak server prijme od klienta packet s flagom o ukončení prenosu dát, prechádza k spracovaniu odoslanej správy/súboru. Nakoniec sa správa vypíše, alebo sa vypíše cesta k prijatému súboru. V tejto časti má server taktiež možnosť si vypýtať switch rolí.

### Pomocné funkcie

V svojej implementácii taktiež používam zopár pomocných funkcií.

**parse\_packet:** tu sa rozparsujú bity z packetu do premenných na základe toho o aký packet sa jedná.

**unpack\_bytes:** ide len o pomocnú funkciu pre parsovanie

**is\_fragment\_correct:** počítanie CRC na strane klienta pre kontrolu správnosti fragmentu od klienta

**send\_keepalive:** funkcia ktorá sa spúšťa v threade po inicializácii spojenia, klient tam počúva odozvu servera na keepalive ktoré odoslal. Handluje sa tam neočakávané ukončenie spojenia ak server nepošle odozvu na keepalive.

### Použité knižnice

Používam copy pre vytvorenie deepcopy zopár arrayov, math pre zaokrúhlenie pri výpočte fragmentov ak nevyjde presné číslo, os pre ukladanie súboru a pracovanie s cestami, random pre výber chybného fragmentu, socket pre základnú funkcionality komunikácie, struct pre prácu s

Petra Miková  
ID: 120852

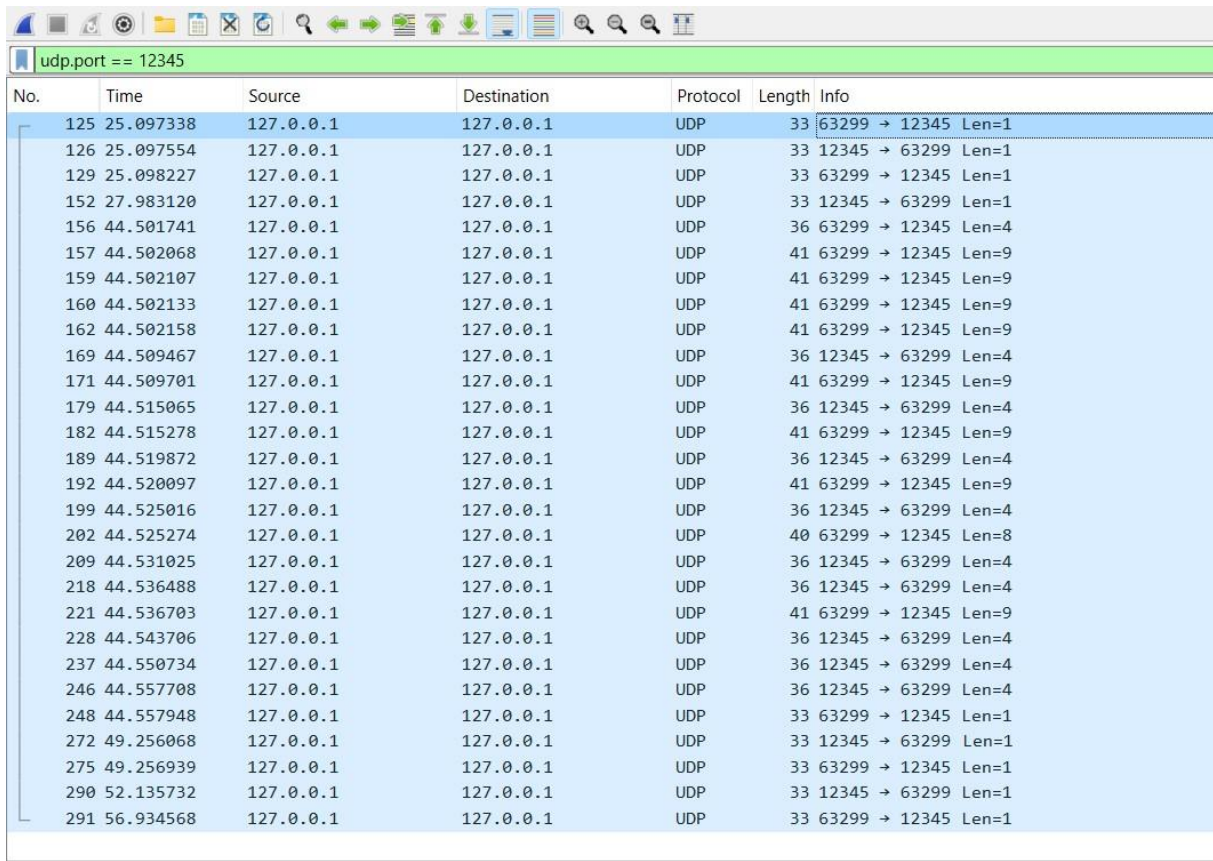
bytami, threading pre vytvorenie threadu na keepalive a taktiež time pre nastavenie intervalu pri keepalive, a z binascii využívam CRC32 funkciu.

## Testovací scenár

Vykonám test na localhoste s odoslaním správy Ahoj, fungujem! po fragmentoch o veľkosti 2 s tým, že jeden packet sa odošle chybný.

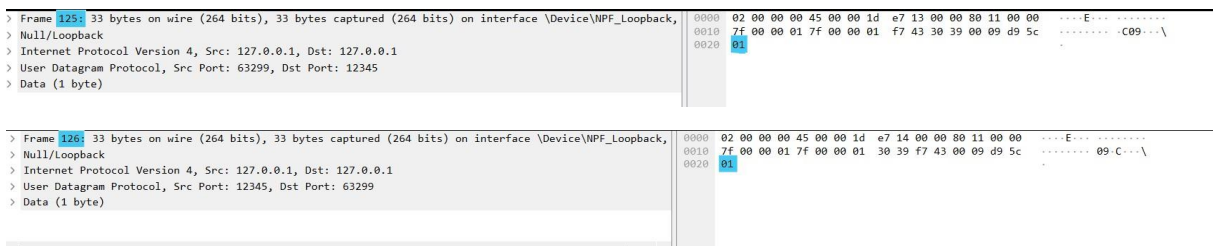
## Wireshark

Takto vyzerá celá komunikácia. Zvolila som port 12345.



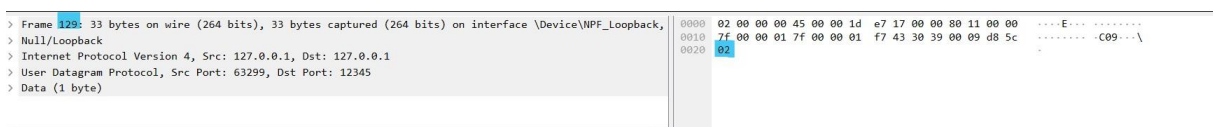
No.	Time	Source	Destination	Protocol	Length	Info
125	25.097338	127.0.0.1	127.0.0.1	UDP	33	63299 → 12345 Len=1
126	25.097554	127.0.0.1	127.0.0.1	UDP	33	12345 → 63299 Len=1
129	25.098227	127.0.0.1	127.0.0.1	UDP	33	63299 → 12345 Len=1
152	27.983120	127.0.0.1	127.0.0.1	UDP	33	12345 → 63299 Len=1
156	44.501741	127.0.0.1	127.0.0.1	UDP	36	63299 → 12345 Len=4
157	44.502068	127.0.0.1	127.0.0.1	UDP	41	63299 → 12345 Len=9
159	44.502107	127.0.0.1	127.0.0.1	UDP	41	63299 → 12345 Len=9
160	44.502133	127.0.0.1	127.0.0.1	UDP	41	63299 → 12345 Len=9
162	44.502158	127.0.0.1	127.0.0.1	UDP	41	63299 → 12345 Len=9
169	44.509467	127.0.0.1	127.0.0.1	UDP	36	12345 → 63299 Len=4
171	44.509701	127.0.0.1	127.0.0.1	UDP	41	63299 → 12345 Len=9
179	44.515065	127.0.0.1	127.0.0.1	UDP	36	12345 → 63299 Len=4
182	44.515278	127.0.0.1	127.0.0.1	UDP	41	63299 → 12345 Len=9
189	44.519872	127.0.0.1	127.0.0.1	UDP	36	12345 → 63299 Len=4
192	44.520097	127.0.0.1	127.0.0.1	UDP	41	63299 → 12345 Len=9
199	44.525016	127.0.0.1	127.0.0.1	UDP	36	12345 → 63299 Len=4
202	44.525274	127.0.0.1	127.0.0.1	UDP	40	63299 → 12345 Len=8
209	44.531025	127.0.0.1	127.0.0.1	UDP	36	12345 → 63299 Len=4
218	44.536488	127.0.0.1	127.0.0.1	UDP	36	12345 → 63299 Len=4
221	44.536703	127.0.0.1	127.0.0.1	UDP	41	63299 → 12345 Len=9
228	44.543706	127.0.0.1	127.0.0.1	UDP	36	12345 → 63299 Len=4
237	44.550734	127.0.0.1	127.0.0.1	UDP	36	12345 → 63299 Len=4
246	44.557708	127.0.0.1	127.0.0.1	UDP	36	12345 → 63299 Len=4
248	44.557948	127.0.0.1	127.0.0.1	UDP	33	63299 → 12345 Len=1
272	49.256068	127.0.0.1	127.0.0.1	UDP	33	12345 → 63299 Len=1
275	49.256939	127.0.0.1	127.0.0.1	UDP	33	63299 → 12345 Len=1
290	52.135732	127.0.0.1	127.0.0.1	UDP	33	12345 → 63299 Len=1
291	56.934568	127.0.0.1	127.0.0.1	UDP	33	63299 → 12345 Len=1

Prvé dva packety sú inicializačné s flagom 1:



Frame	Details	Hex	ASCII
125	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: 63299, Dst Port: 12345 Data (1 byte)	0000 02 00 00 00 45 00 00 1d e7 13 00 00 00 11 00 00 0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 09 d9 5c 0020 01	.....E..... .....C09....\
126	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: 12345, Dst Port: 63299 Data (1 byte)	0000 02 00 00 00 45 00 00 1d e7 14 00 00 00 11 00 00 0010 7f 00 00 01 7f 00 00 01 30 39 f7 43 00 09 d9 5c 0020 01	.....E..... .....09.C....\

Hneď na to sa začína posilať keepalive s flagom 2, v tomto prípade sa stihlo poslať raz až dokým si klient nevybral možnosť poslať správu.



Frame	Details	Hex	ASCII
129	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: 63299, Dst Port: 12345 Data (1 byte)	0000 02 00 00 00 45 00 00 1d e7 17 00 00 00 11 00 00 0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 09 d8 5c 0020 02	.....E..... .....C09....\

> Frame 152: 33 bytes on wire (264 bits), 33 bytes captured (264 bits) on interface \Device\NPF_{Loopback}, Null/Loopback	0000 02 00 00 00 45 00 00 1d e7 2e 00 00 80 11 00 00	-----E---% -f-----
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	0010 7f 00 00 01 7f 00 00 01 30 39 f7 43 00 09 d8 5c	-----C09---N
> User Datagram Protocol, Src Port: 12345, Dst Port: 63299	0020 02	-----
> Data (1 byte)		

Po tomto sa teda zo strany klienta odoslal packet s flagom 3, ktorý hovorí, že sa ide odosielať správa. Taktiež v sebe udržiava informáciu o tom koľko fragmentov sa bude posilať, v tomto prípade 8. Hneď po tomto začína už samotný prenos dát

Frame 156: 36 bytes on wire (288 bits), 36 bytes captured (288 bits) on interface \Device\NPF_{Loopback}, Null/Loopback	0000 02 00 00 00 45 00 00 20 e7 2f 00 00 80 11 00 00	-----E---% -f-----
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 0c d7 4e	-----C09---N
User Datagram Protocol, Src Port: 63299, Dst Port: 12345	0020 03 00 00 08	-----
Data (4 bytes)		

Prenesú sa teda prvotné 4 fragmenty z definovaného sliding window podľa Selective Repeat metódy.

> Frame 157: 41 bytes on wire (328 bits), 41 bytes captured (328 bits) on interface \Device\NPF_{Loopback}, Null/Loopback	0000 02 00 00 00 45 00 00 25 e7 30 00 00 80 11 00 00	-----E---% -0-----
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 11 b1 b9	-----C09---N
User Datagram Protocol, Src Port: 63299, Dst Port: 12345	0020 00 00 00 1b c9 36 8d 41 68	-----6A h
Data (9 bytes)		

> Frame 159: 41 bytes on wire (328 bits), 41 bytes captured (328 bits) on interface \Device\NPF_{Loopback}, Null/Loopback	0000 02 00 00 00 45 00 00 25 e7 32 00 00 80 11 00 00	-----E---% -2-----
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 11 b1 a9	-----C09---N
User Datagram Protocol, Src Port: 63299, Dst Port: 12345	0020 00 00 01 ff 02 34 ba 6f 6a	-----4 o j
Data (9 bytes)		

> Frame 160: 41 bytes on wire (328 bits), 41 bytes captured (328 bits) on interface \Device\NPF_{Loopback}, Null/Loopback	0000 02 00 00 00 45 00 00 25 e7 33 00 00 80 11 00 00	-----E---% -3-----
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 11 f3 31	-----C09---1
User Datagram Protocol, Src Port: 63299, Dst Port: 12345	0020 00 00 02 b0 19 3e ab 2c 20	-----> j
Data (9 bytes)		

> Frame 162: 41 bytes on wire (328 bits), 41 bytes captured (328 bits) on interface \Device\NPF_{Loopback}, Null/Loopback	0000 02 00 00 00 45 00 00 25 e7 35 00 00 80 11 00 00	-----E---% -5-----
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 11 ac ef	-----C09---LVhf u
User Datagram Protocol, Src Port: 63299, Dst Port: 12345	0020 00 00 03 a0 4c 56 68 66 75	-----
Data (9 bytes)		

Po tomto sa od severa prijíma prvý ACK na fragment 0. ACK nesie flag 5, NACK nesie flag 6.

> Frame 169: 36 bytes on wire (288 bits), 36 bytes captured (288 bits) on interface \Device\NPF_{Loopback}, Null/Loopback	0000 02 00 00 00 45 00 00 20 e7 3c 00 00 80 11 00 00	-----E---% -<-----
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	0010 7f 00 00 01 7f 00 00 01 30 39 f7 43 00 0c d5 56	-----C09---V
User Datagram Protocol, Src Port: 12345, Dst Port: 63299	0020 05 00 00 00	-----
Data (4 bytes)		

Okno sa posúva a posiela sa ďalší fragment, teraz číslo 4.

> Frame 171: 41 bytes on wire (328 bits), 41 bytes captured (328 bits) on interface \Device\NPF_{Loopback}, Null/Loopback	0000 02 00 00 00 45 00 00 25 e7 3e 00 00 80 11 00 00	-----E---% ->-----
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 11 66 b1	-----C09---f-
User Datagram Protocol, Src Port: 63299, Dst Port: 12345	0020 00 00 04 4f 31 dd d6 6e 67	-----01--n g
Data (9 bytes)		

Pokračuje sa rovnakým spôsobom (nebudem tu prikladať všetky screenshoty), až dokým nedôjde náhodou k prijatiu NACKu. V našom prípade teda chybný fragment bol 4.

> Frame 203: 36 bytes on wire (288 bits), 36 bytes captured (288 bits) on interface \Device\NPF_{Loopback}, Null/Loopback	0000 02 00 00 00 45 00 00 20 e7 64 00 00 80 11 00 00	-----E---% -d-----
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	0010 7f 00 00 01 7f 00 00 01 30 39 f7 43 00 0c d4 52	-----C09---R
User Datagram Protocol, Src Port: 12345, Dst Port: 63299	0020 06 00 00 04	-----
Data (4 bytes)		

Po prijatí tohto sa prijme ďalší ACK, a hneď na to sa posiela tento vyžiadany packet ktorý bol chybný.

> Frame 218: 36 bytes on wire (288 bits), 36 bytes captured (288 bits) on interface \Device\NPF_{Loopback}, > Null/Loopback > Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 > User Datagram Protocol, Src Port: 12345, Dst Port: 63299 > Data (4 bytes)	0000 02 00 00 00 45 00 00 20 e7 6d 00 00 80 11 00 00 .....E..... 0010 7f 00 00 01 7f 00 00 01 30 39 f7 43 00 0c d5 51 .....09:C...Q 0020 05 00 00 05
> Frame 221: 41 bytes on wire (328 bits), 41 bytes captured (328 bits) on interface \Device\NPF_{Loopback}, > Null/Loopback > Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 > User Datagram Protocol, Src Port: 63299, Dst Port: 12345 > Data (9 bytes)	0000 02 00 00 00 45 00 00 25 e7 70 00 00 80 11 00 00 .....E...%p..... 0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 11 5d 84 .....C09...]- 0020 00 00 04 9e 63 bb ad 6e 67 .....c...n g

Ďalej už len pokračuje posielanie ďalších fragmentov a prijímanie ACKov. Ak toto bolo dokončené, klient posla serveru upovedomenie s flagom 7, že sa prenos skončil.

> Frame 248: 33 bytes on wire (264 bits), 33 bytes captured (264 bits) on interface \Device\NPF_{Loopback}, > Null/Loopback > Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 > User Datagram Protocol, Src Port: 63299, Dst Port: 12345 > Data (1 byte)	0000 02 00 00 00 45 00 00 1d e7 8b 00 00 80 11 00 00 .....E..... 0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 09 d3 5c .....C09...\ 0020 07
--	--

Server mu odpovedá.

> Frame 272: 33 bytes on wire (264 bits), 33 bytes captured (264 bits) on interface \Device\NPF_{Loopback}, > Null/Loopback > Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 > User Datagram Protocol, Src Port: 12345, Dst Port: 63299 > Data (1 byte)	0000 02 00 00 00 45 00 00 20 e7 a3 00 00 80 11 00 00 .....E..... 0010 7f 00 00 01 7f 00 00 01 30 39 f7 43 00 0c d5 52 .....09:C...R 0020 07
--	---

Znova sa zapne a prijíma keepalive do ďalšieho výberu zo strany klienta.

> Frame 275: 33 bytes on wire (264 bits), 33 bytes captured (264 bits) on interface \Device\NPF_{Loopback}, > Null/Loopback > Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 > User Datagram Protocol, Src Port: 63299, Dst Port: 12345 > Data (1 byte)	0000 02 00 00 00 45 00 00 1d e7 a6 00 00 80 11 00 00 .....E..... 0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 09 d8 5c .....C09...\ 0020 02
> Frame 290: 33 bytes on wire (264 bits), 33 bytes captured (264 bits) on interface \Device\NPF_{Loopback}, > Null/Loopback > Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 > User Datagram Protocol, Src Port: 12345, Dst Port: 63299 > Data (1 byte)	0000 02 00 00 00 45 00 00 1d e7 b5 00 00 80 11 00 00 .....E..... 0010 7f 00 00 01 7f 00 00 01 30 39 f7 43 00 09 d8 5c .....09:C...\ 0020 02

Klient poslal ukončovací packet s flagom 9, server ho len prijme a spojenie sa uzatvára na oboch stranách.

> Frame 291: 33 bytes on wire (264 bits), 33 bytes captured (264 bits) on interface \Device\NPF_{Loopback}, > Null/Loopback > Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1 > User Datagram Protocol, Src Port: 63299, Dst Port: 12345 > Data (1 byte)	0000 02 00 00 00 45 00 00 1d e7 b6 00 00 80 11 00 00 .....E..... 0010 7f 00 00 01 7f 00 00 01 f7 43 30 39 00 09 d1 5c .....C09...\ 0020 09
--	--

## Záver

Zadanie hodnotím ako veľmi zaujímavé a náučné na pochopenie problematiky komunikácie po sieti. Podarilo sa mi naimplementovať všetko z požiadavok a funguje to ako na localhoste tak aj na dvoch PC, samozrejme pri vypnutí firewallu. Komunikáciu dokážem korektne odchytiť aj pomocou Wiresharku, kde pekne vidno presný postup spojenia a prenosu dát.