

Počítačové a komunikačné siete

Zadanie 2

Komunikácia s využitím UDP protokolu

Martin Melíšek

Cvičenie: Štvrtok 10:00

Cvičiaci: Ing. Kristián Košťál, PhD.

2020/2021

Komunikácia s využitím UDP protokolu

Úvod

Zadaním tohto projektu je navrhnuť a implementovať program s použitím vlastného protokolu nad protokolom UDP transportnej vrstvy sieťového modelu TCP/IP. Program umožní komunikáciu medzi dvoma účastníkmi v lokálnej sieti. Účastníci komunikácie budú mať možnosť posilať **textové správy alebo ľubovoľný súbor** medzi počítačmi (uzlami). Program bude pozostávať z dvoch častí, medzi ktorými je možné prepínať. Vysielacia, čiže klientska zahajuje spojenie s prijímacou – serverovou časťou. Pri takejto komunikácii sa predpokladá, že v sieti dochádza k stratám dát a preto je potrebné navrhnuť protokol, tak aby spojenie bolo **spoľahlivé** a predišlo sa k strateným packetom. Počas komunikácie sa packety fragmentujú, tak aby neprevýšil maximálnu možnú veľkosť MTU pre Ethernet II a to 1500B. Po prenesení prvého súboru pri nečinnosti komunikátor automaticky odošle packet pre **udržanie spojenia** každých 10-60s, pokiaľ používateľ neukončí spojenie.

Analýza

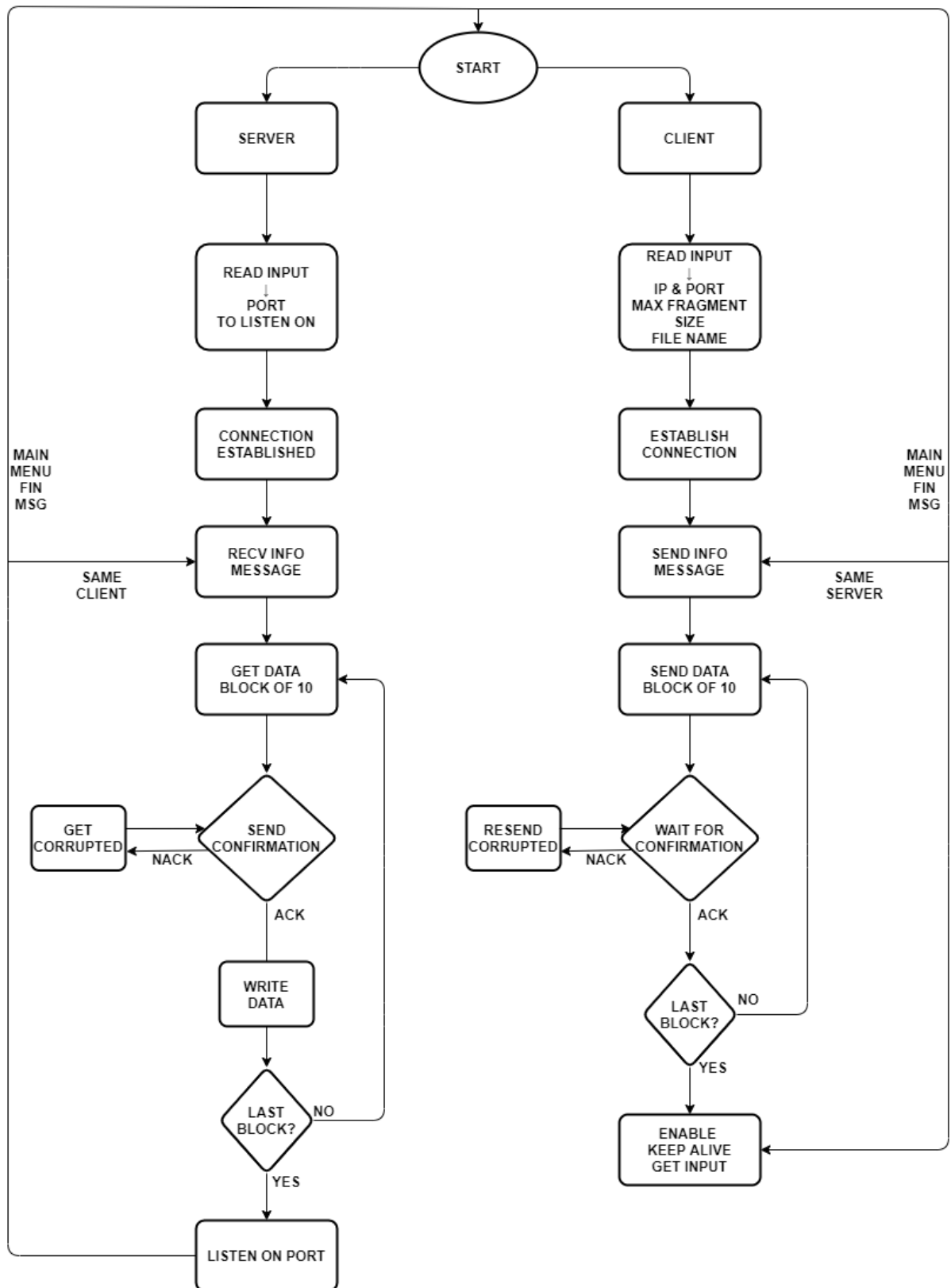
Komunikácia prebieha nad protokolom UDP, ktorý **nezaručuje spoľahlivé doručenie** všetkých packetov. Pre aplikácie, ktoré si nemôžu dovoliť oneskorenie pri opakovanom prenose chybných packetov a nevyžadujú overenie alebo opravu údajov je veľmi vhodným prostriedkom výmeny dát. UDP protokol je zameraný na rýchlosť a preto sa môže stať, že packety neprídu v správnom poradí alebo dokonca druhá strana nemusí byť pripravená prijať dáta, ktoré sú jej posielané. Tento fakt je potrebné vyriešiť implementovaním **vlastného protokolu**, ktorý bude vo vlastnej réžii riadiť výmenu správ medzi komunikujúcimi uzlami. K pôvodnej hlavičke packetu pridáme vlastnú na aplikačnej vrstve. V tejto hlavičke si budeme posilať potrebné informácie, ktoré nám pomôžu zabezpečiť komunikáciu, po stránke spoľahlivosti, tak aby prebehla v poriadku a bez strát.

Pri implementácii vlastného protokolu sa budeme snažiť priblížiť k niektorým prvkom, ktoré obsahuje **TCP** – transmission control protocol. Protokol TCP garantuje doručenie dát k príjemcovi a dodanie v rovnakom poradí ako boli odoslané. Na to aby všetko prebehlo bezproblémovo je potrebné nadviazať spojenie pomocou *three-way handshake* – vymeniť si informačné správy a zvoliť si náhodne vygenerované sekvenčné čísla. Až po úvodných troch packetoch je možné zahájiť dátovú komunikáciu, ktorá taktiež musí byť riadne ukončená.

Oba spomenuté transportné protokoly používajú kontrolný súčet (**checksum**) na overenie korektne prijatého packetu. Možnosť ako vypočítať checksum je viacero a rozličné protokoly môžu využívať inú techniku. Hlavnou pointou je, že pred odoslaním packetu resp. enkapsulácie sa vypočíta menšie množstvo dát, ktoré vznikne následkom určitej operácie na väčšom dátovom bloku. Po prijatí sa na druhej strane vykoná rovnaká operácia a vypočíta sa vlastný kontrolný súčet. Akonáhle sa vypočítaná hodnota nezhoduje s prijatou hodnotou, znamená to, že počas prenosu došlo k chybe.

Návrh protokolu

Blokový návrh – koncepcia fungovania riešenia



Obrázok 1: Blokový návrh programu

Obsah hlavičky

Hlavička bude predstavovať určitú časť dát, ktoré budeme odosielať v každom packete, ktorý odošleme príjemcovi. Jej forma sa počas komunikácie bude **meniť**. Vo vývoji, toho ako bude vyzeráť sa pozeráme hlavne na jej veľkosť a užitočnosť. Aby sme ušetrili redundantné dáta posielané pri inicializácii spojenia, rozdelíme hlavičku na viacero typov.

Typy hlavičky:

- **Informačná**
 - Neobsahuje časti posielaného súboru alebo textu, ale slúži na:
 - Nadviazanie spojenia
 - Nultý fragment pri inicializácii prenosu dát
 - Ostatné typy, ktoré obsahujú iba signalizačnú správu
- **Dátová**
 - Spolu so samotnými dátami posielame aj ďalšie potrebné informácie pre server

Signalizačné typy správ

Typ je 1B pole na začiatku hlavičky, ktoré obsahuje hlavnú informáciu, o význame packetu.

Názov	Skratka	Binárna hodnota	Decimálna hodnota
Otvorenie spojenia	SYN	00000001	1
Korektné prijatie dát	ACK	00000010	2
Chybné prijatie dát	NACK	00000100	4
Inicializácia prenosu	INIT	00001000	8
Dáta - Správa	DM	00010000	16
Dáta - Súbor	DF	00100000	32
Ukončenie spojenia	FIN	01000000	64
Keep Alive	KA	10000000	128

Tabuľka 1: Typy správ

Systém je navrhnutý tak, aby v prípade, že si to implementácia vyžaduje, je možné kombinovať jednotlivé typy správ a odosielať viacero na raz.

V programe bude každá hlavička vyzeráť nasledovne:

Type	Data	CRC
------	------	-----

Obrázok 2: Základná forma hlavičky

Na základe tohto konceptu budeme prispôbovať poličko Data vzhľadom na posielať typ správy. Čisto informačný typ správ, čiže: SYN, ACK, FIN, KA nepotrebujú žiadne ďalšie dáta a iba oboznamujú server alebo klienta o stave v akom sa spojenie nachádza.

SYN – Nadviazanie spojenia

ACK – Potvrdenie prijatia predch. správy

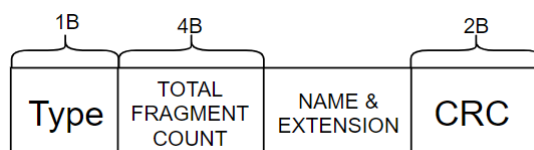
FIN – Uzavretie socketu a spojenia

KA – Keep Alive správa pre server

Typy správ, kde potrebujeme zaslať viacero informácií pre prehľadnosť komunikácie sú: INIT, NACK, DM, DF

- **INIT**

- V tomto nultom fragmente odosielame informácie o posielaných dátach.
- Celkový počet fragmentov - 4B
- Ak posielame správu nastavíme aj DM bit, ak súbor DF.
- Príklad prvého fieldu v hlavičke: 0010 1000 – INIT a DF; odosielame aj meno súboru.
- V prípade súboru zapíšeme aj názov a príponu. (V OS Windows je maximálna dĺžka názvu súboru 255 znakov)



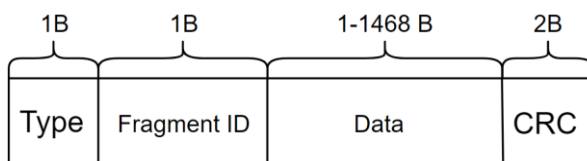
Obrázok 3: INIT hlavička

- **NACK**

- V prípade chybného alebo chýbajúceho packetu odošleme NACK, čo predstavuje informáciu pre druhú stranu o tom, že posledne posielaný packet neprišiel v poriadku.
- Pri výmene dát, zapisujeme do tela tejto správy indexy poškodených packetov.
- ID packetov zapisujeme do 2B čísla nasledovne:
 - 0-9. bit → Podľa toho, ktorý bit je nastavený vieme zistiť, ktorý packet z bloku neprišiel korektne.
 - 14-10. bit → Nevyužitý.
 - 15. bit → Obsahuje okamžitú informáciu o tom, že exaktne posledná správa neprišla v poriadku, posielala sa v prípade SYN,ACK,INIT chýb.

- **DM(Data Message) & DF (Data File)**

- Ich súčasťou je ID z konkrétneho bloku odosielaných packetov.
- Samotné dáta.



Obrázok 4: Dátová hlavička

Na základe týchto typov správ môžeme prehlásiť, že efektívna dĺžka hlavičky je iba 4B. Pre klienta to znamená, že môžeme poslať v jednom rámci $1500 - 20 - 8 - 4 = \mathbf{1468B}$ dát. (MAX MTU – IP Header – UDP Header – Custom Header). Všetky potrebné informácie na uloženie súboru pošleme v inicializačnej správe a už ich nemusíme znovu opakovanne posielat'. Inicializačná informačná správa je najdlhšia, ktorú odosielame, čo sa týka hlavičky, ale neobsahuje dáta. Server si udržiava iba počítadlo fragmentov, nakoľko vie, koľko ich má

očakávať. Už dopredu si z celkového počtu vypočíta, ktorý blok bude obsahovať menší počet fragmentov ako obvykle. Na serveri sa spojenie ukončí, ak mu príde **FIN** správa (generovaná klientom pri vstupe od používateľa) alebo nepríde **KA**.

ARQ Metóda (Automatic Repeat reQuest)

Z dôvodu definície UDP protokolu ako určitá forma nespoľahlivého kanála musíme zabezpečiť plynulé a správne prijatie odoslaných packetov. Vhodne zvolená ARQ metóda nám uľahčí nasledovné vyžiadanie nesprávne prijatých fragmentov. Pomocou kladných a negatívnych potvrdení o prijatí správy poskytneme **spätnú väzbu** odosielajúcej strane. Prijemca indikuje zaslaním typu správy **ACK**, že bezchybne prijal dátový rámec, ktorý mu bol doručený. Ak odosielateľ neobdrží potvrdenie než uplynie interne nastavený časovač, musí packet odoslať znovu alebo popripade spojenie preruší.

V tomto projekte využijeme posielanie dátových packetov po blokoch vo veľkosti 10.

Po potvrdení o úspešnom prijatí inicializačného packetu so základnými informáciami odosielateľ vždy posieľa fragmenty za sebou a indexuje ich od 0 po 9. Po poslednom odoslanom fragmente čaká na potvrdenie. Server prijíma a z každého fragmentu počíta checksum. Ak sa zhoduje, označí ho ako korektne prijatý. Po 10 fragmentoch zapíše prijaté dáta do súboru a vygeneruje **ACK** správu. V prípade, že niektorí z packetov nepríde alebo obsahuje poškodené dáta vygeneruje sa **NACK** správa s indexami packetov, kde nastala chyba. Odosielateľ obdrží túto správu a musí znovu odoslať konkrétne fragmenty z tohto bloku a zopakovať proces len s chybnými packetmi. Nakoľko v každom fragmente je jeho ID dokážeme ich na strane serveru zoradiť a tým pádom vyriešiť nedostatok UDP protokolu pokiaľ by nám prišli v nesprávnom poradí.

Celkovo môžeme tento proces označiť ako **blokovú metódu so selektívnym žiadaním**.

Pri takomto kontrolovaní packetov môže nastať viacero možných okrajových situácií, ktoré je potrebné vyriešiť. Medzi príklady môžeme zaradiť, to že id v packete, môže prísť nesprávne (poškodené) a preto musíme skontrolovať, aj tie packety ktoré prišli správne. Následne vykonáme porovnanie medzi týmito množinami, aby sme si nevyžiadali už správne obdržaný packet.

Checksum algoritmus

Možností ako vypočítať checksum je viacero, v tomto projekte sa budeme venovať **CRC – Cyclic Redundancy Check**. Odosielajúca strana, predtým než enkapsuluje packet, vydolí dáta + hlavičku polynómom n -tého stupňa, pričom na koniec dát do *footera* sa pripíše zvyšok po delení dĺžky n . Na druhej strane sa po prijatí dáta znovu vydolia dohodnutým rovnakým polynómom. Ak bude zvyšok rovnaký ako je zvyšok zapísaný v prijatom fragmente, znamená to, že nenastala žiadna chyba pri prenose dát. CRC algoritmus je vhodný pri odosielaní menšej veľkosti dát a preto nám bude stačiť **2B**, ktorú používa aj TCP. Využijeme všeobecne známy predefinovaný **CRC-16-CCITT** z importovaného modulu `crcmod`, ktorý je založený na polynóme: $x^{16} + x^{12} + x^5 + 1$. Používa sa vo viacerých well-known protokoloch ako napríklad *Bluetooth*.

Keep Alive Metóda

Pomocou **Keep Alive** signálu dokážeme udržiavať spojenie aktívne a nemusíme ho opätovne nadväzovať. Posielanie týchto správ sa aktivuje na klientskej časti programu, akonáhle sa odošle prvý súbor alebo textová správa a opakuje sa každých **20 sekúnd**. Pokiaľ nedorazí potvrdenie do určitého časového limitu (1 sekunda), klient opätovne odošle **KA** a ak ani v tomto prípade odpoveď nepríde spojenie sa prehlási za ukončené a uzavrie sa socket.

Odosielanie týchto správ by malo prebiehať na samostatnom vlákne/podprocese aby neblokovalo opätovné zadanie vstupu z klávesnice a teda odosielanie súborov klientom. Keep Alive správy budú vhodným prostriedkom aj pre server, ktorý po uplynutí časovača a neobdržaní **KA** správy ukončí spojenie a uzavrie socket.

Implementačné prostredie a použité knižnice

Program predstavuje konzolovú aplikáciu, vytvorenú v programovacom jazyku **Python v3.8.5 64bit**. Spúšťa sa cez Command Line, PowerShell alebo iný CLI nástroj. Počas celého vykonania programu používame nasledovné importované moduly:

- [socket](#)
 - poskytuje základné rozhranie na otvorenie spojenia, odoslanie a prijatie rámcov
- [struct](#)
 - zabezpečuje enkapsuláciu python objektov na fixnú veľkosť v packete
- [os](#)
 - získanie veľkosti posiadaného súboru
- [math](#)
 - zaokrúhlenie pri výpočte počtu fragmentov z veľkosti súboru
- [crcmod](#)
 - vytvorenie predefinovanej funkcie na výpočet CRC – použité na checksum.

Používateľské rozhranie

Aplikácia sa ovláda pomocou príkazov zadávaných z klávesnice do konzoly. Na začiatku má používateľ na výber nastaviť mód komunikácie – Klient alebo Server. Následne odpovedá na prompty programu a zadáva IP + port servera, maximálnu veľkosť fragmentu a ak chce odoslať súbor, tak jeho názov alebo priamo textovú správu. Na strane servera zadá port na akom má server počúvať, prípadne ešte aj cestu kam sa súbor uloží. Po úspešnom prenesení dát, na klientskej časti, buď zadá cestu k novému súboru, prepne mód alebo môže ukončiť aplikáciu.

Zmeny oproti návrhu

INIT Správy pre prenos súboru

V časti servera došlo k zmene pri prijatí prvej INIT správy. Ak je prvá INIT správa typu ("**INIT**", "**DF**"), tak to znamená, že bude nasledovať rozfragmentovaný názov súboru. Tento názov sa prenáša rovnako ako **správa** a teda server posiela ACK alebo NACK na základe, toho či sa podarilo prijať všetky dáta z bloku. Následne sa už klasicky pokračuje v prenesení dát zo súboru. V INIT správach sa nachádza iba počet fragmentov nasledujúcich dát. INIT, s DM typom ostáva rovnaký.

Zjednotenie veľkostí hlavičiek do dvoch typov

V návrhu sme mohli pozorovať vlastné hlavičky pre INFO, INIT, NACK a DATA. Teraz už existujú len dva typy, ktoré vyzerajú nasledovne:

- DATA hlavička
 - 1B typ - DF, DM, NACK, INIT
 - 4B pole, ktoré obsahuje údaje na základe typu

Typ	Údaj
INIT	Počet fragmentov dát.
DF, DM	Celkové ID posieleného fragmentu
NACK	Počet znovuvyžiadaných fragmentov

Tabuľka 2: 4B pole dátovej hlavičky

- 1 až 1465B - DATA

Typ	Údaj
INIT	Nevyužitý
DF, DM	Dáta správy/súboru
NACK	ID žiadaných fragmentov

Tabuľka 3: Dátové pole

- INFO hlavička, ktorá **NEOBSAHUJE** nepoužité polia z dátovej hlavičky
 - 1B typ – SYN, ACK, KA, FIN

CRC ostalo pre obe hlavičky 2B.

Zvýšenie prehľadnosti a upustenie od zbytočných komplikácií udržiavania viacerých typov hlavičiek.

Veľkosť a informácie v poli fragment ID

Na základe feedbacku prišlo k prechodu z relatívneho ID na **absolútne** a teda potrebujeme pre toto číslo 4B. Posielame v ňom exaktné poradové číslo posieleného fragmentu. Hlavným dôvodom je prípad, keby prišiel dátový paket pre server oneskorene, nebolo by možné na strane prijímača rozoznať, že ide o paket, ktorý sa nemá zapisovať do súboru a má ho zahodiť. Takto už vie, že obsahuje ID, ktoré už zapísal a môže ho zahodiť.

Ukladanie údajov zo súboru na strane servera

Keďže posielame fragment id s absolútnou cestou, neodstraňujeme z RAM fragmenty staršie ako veľkosť bloku a teda môže prijímač ukladať do pamäte všetky dáta a zapísať ich až po prijatí celého súboru.

Maximálna veľkosť dátového paketu

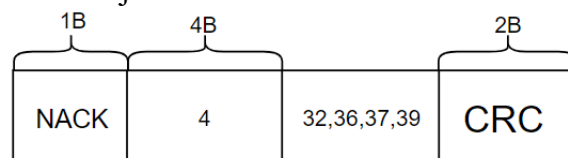
Nakoľko sa zväčšilo pole fragment ID v dátovom pakete z 1B na 4B, maximálna veľkosť dátového paketu klesla o 3B z 1468B na **1465B**.

Keep Alive cyklus

Kratšia doba pre Keep Alive cyklus, ktorý beží na vlastnom vlákne, z **20 sekúnd** prechod na **10 sekúnd**. Dôvodom je fakt, že na to aby KA vlákno nepoužívalo navyše veľké množstvo zdrojov CPU musí byť uspané. **Uspanie** jedného vlákna znamená, že sa nebude v nekonečnom cykle kontrolovať, či môže odoslať správu a ak používateľ chce opätovne poslať súbory/ukončiť spojenie, musí počkať dokým sa toto vlákno opäť zobudí a skontroluje stav spojenia.

Formát NACK správ

NACK správy, rovnako ako dátové správy, obsahujú **absolútne ID fragmentov**, ktoré boli poškodené alebo neprišli vôbec. Dĺžka tejto správy nie je na rozdiel od návrhu ohraničená veľkosťou 2B, ale môže byť akokoľvek dlhá, pretože NACK správy sa nemusia fragmentovať. Inšpirujeme sa protokolom *FTP*, ktorý posla rôzne príkazy a s nimi aj potrebné dáta. Príkazy môžeme vnímať v našom protokole ako signalizačné správy a konkrétne príkaz *FTP PASV* obsahuje určité údaje vo svojom tele. Tieto dáta sú kódované ako ASCII znaky – čísla oddelené čiarkami. Preto typ NACK obsahuje celkový počet žiadaných fragmentov a v dátovej časti ich čísla sú oddelené znakom čiarky. Príklad:



Obrázok 5:Príklad formátu NACK správy

Implementácia

Základnou myšlienkou fungovania programu je možnosť nastavenia programu ako klient (odosielateľ) alebo server (prijemca). Komunikácia prebieha pomocou **half-duplex**, čiže v jednom okamihu daný uzol buď, prijíma alebo vysiela. Po prijatí všetkých vstupných informácií sa inicializujú triedy pre oba uzly, nadviaže spojenie a začne každý uzol vykonávať svoju jednoznačnú činnosť. Server počúva a klient odosiela dáta.

Používateľské rozhranie, vstupné údaje a interakcia

Na správne fungovanie programu je potrebné si od užívateľa vypýtať nasledovné údaje:

Client

- IPv4 Adresa servera v štandardizovanej forme, akceptuje sa aj slovo „localhost“, ak sa jedná o komunikáciu dvoch procesov na rovnakom PC.
- Port na akom server počúva, rozsah: 1 – 65535. Odporúča sa zadať port vyšší než 1024, ale je to len odporúčanie.
- Maximálna veľkosť fragmentov dát, rozsah: 1 – 1465. K tejto hodnote je implicitne pripočítaná maximálna veľkosť hlavičky a nepríde k fragmentácii na linkovej vrstve.
- Výber, toho čo sa odosiela:
 - Správa
 - Samotná správa, akejkolvek dĺžky dané CLI prostredie ponúka. Akonáhle užívateľ stlačí **ENTER** správu už nie je možné zmeniť.
 - Súbor
 - Cesta k súboru – absolútna alebo relatívna. Podporované sú '/' '/' '/' '\' '\' v kombinácii s '.' a '..'
 - Predpokladá sa, že užívateľ má právo na čítanie tohto súboru.
- % chybných paketov pre simuláciu chyby alebo -X, čo predstavuje pokazenie konkrétneho paketu (ak sa taký odosiela).

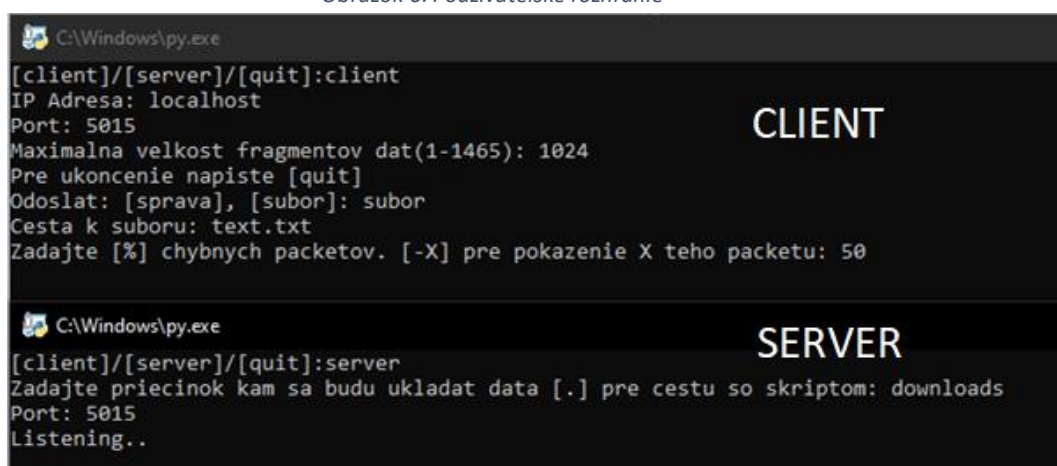
Server

- Priečinko, kam sa budú ukladať prijaté správy – Absolútna cesta alebo relatívna. Systémové skratky alebo premenné prostredia, ako %appdata% alebo %UserProfile% nie sú podporované. Predpokladá sa, že používateľ má právo na zápis do tohto priečinku.
 - Ak neexistuje, objaví sa *prompt*, či si ho užívateľ želá vytvoriť.
 - Nie – skript sa vráti na výber iného priečinku.
 - Áno – vytvorí sa v ceste obsahujúcu skript.
- Port, na ktorom server počúva, rozsah: 1 – 65535, opäť sa neodporúča používať well known porty..

Je potrebné najprv nastaviť server a až potom klienta.

Na to aby komunikácia vôbec prebehla musí sa usporiadaná dvojica (IP,PORT) zhodovať s IP adresou zariadenia, na ktorom je spustený server a portom na ktorom počúva. Najprv sme nastavili server: ukladanie do priečinka downloads počúvanie na porte 5015. Klient rovnaká IP&port, veľkosť dátových fragmentov 1024, posiela sa súbor text.txt a je 50% šanca na pokazenie dátového segmentu

Obrázok 6: Používateľské rozhranie



```
C:\Windows\py.exe
[client]/[server]/[quit]:client
IP Adresa: localhost
Port: 5015
Maximalna velkost fragmentov dat(1-1465): 1024
Pre ukoncenie napiste [quit]
Odoslat: [sprava], [subor]: subor
Cesta k suboru: text.txt
Zadajte [%] chybných paketov. [-X] pre pokazenie X teho packetu: 50

C:\Windows\py.exe
[client]/[server]/[quit]:server
Zadajte priecinok kam sa budu ukladat data [.] pre cestu so skriptom: downloads
Port: 5015
Listening..
```

Opis mechanizmu odosielania a prijímania dát

Na úvod je vhodné povedať, že cieľom pri implementácii a programovaní funkčnosti tohto programu je navrhnuť riešenie, ktoré sa bude snažiť zjednotiť funkcie využité oboma uzlami a praktizovať pritom **DRY** (Don't Repeat Yourself) princíp. Program obsahuje jednu abstraktnú spoločnú triedu s názvom **Uzol**, od ktorej obe konkrétne triedy **Client** & **Server** dedia. Táto trieda má atribúty a metódy, ktoré využijeme aj pri odosielaní, tak aj pri prijímaní dát.

Inicializácia

Pred začatím odosielania sa vytvorí trieda **Constants** a trieda **Crc**, ktorých inštancie putujú ako parameter pre naše dva uzly. Konštanty obsahujú veľkosti hlavičiek, určité nemenné hodnoty a všetky typy hlavičiek načítané z externého súboru **.JSON**. Tento jednoduchý súbor pozostáva z čísel a korešpondujúcich názvov z [tabuľky č.1](#). Crc obsahuje len dve metódy, výpočet a kontrolu crc checksumu. Kontrolné výpisy na informácie a debugovanie sú riešené cez custom **Logger**, s defaultnou hodnotou integrity na výpis všetkého, čo sa v programe deje. Následne po prijatí vstupných údajov sa začína prenos.

Nadviazanie spojenia

Nadviazanie prebieha klasickým spôsobom prebratým od protokolu TCP a to three-way handshake cez (SYN), (SYN,ACK), (ACK) správy. Po otvorení socketu sa nastaví **timery** na oboch uzloch a server si zapíše **adresu klienta**. Nadviazanie spojenia je potrebné vykonať iba raz.

```
def nadviaz_spojenie(self):
    try:
        self.send_simple("SYN", self.target)
        self.recv_simple(("SYN", "ACK"), self.recv_buffer)
        self.send_simple("ACK", self.target)
    except CheckSumError:
        self.logger.log("Poskodeny packet, chyba pri nadviazani spojenia", 5)
        raise
    except socket.timeout:
        self.logger.log("Cas vyprsal pri inicializacii", 5)
        raise
    self.logger.log("Spojenie nadviazane.\n", 1)
```

Tu si môžeme všimnúť využitie **loggeru** na výpis, **custom exception** v podobe **ChecksumError**, ktorá je raisnutá v prípade, že správa dorazí poškodená.

Obrázok 7: Nadviazanie spojenia

Prijatie & odoslanie signalizačných správ bez dát

Procedúry v triede **Uzol**,

- `send_simple(typ, target)`,
- `recv_simple(expected_typ, buffer)`,

slúžia ako *wrapper* pre globálne prijímanie **INFO** správ → segmentov, ktoré neobsahujú dáta. Odosielanie je jednoduché, vytvorí sa typ na základe vstupu a vypočíta sa checksum. Na uloženie polí v hlavičke na konkrétne veľkosti sa používa metóda z modulu **struct** → `struct.pack(format, data)`. Formát je nastavený na oboch uzloch rovnako a jednotlivé polia sa **NEZAROVNÁVAJÚ**. Hlavička veľkosti napr. **7B**(pozostávajúca z 1B typ, 4B int 2B short) je **skutočne 7B**. Toto je riešené pomocou operátora `" = "` v parametri formát, ktorý vypne zarovňovanie a použije natívne zarovnanie systému (Little alebo Big Endian)

zariadenia, pre ktoré je tento program navrhnutý. Windows OS – Little Endian.

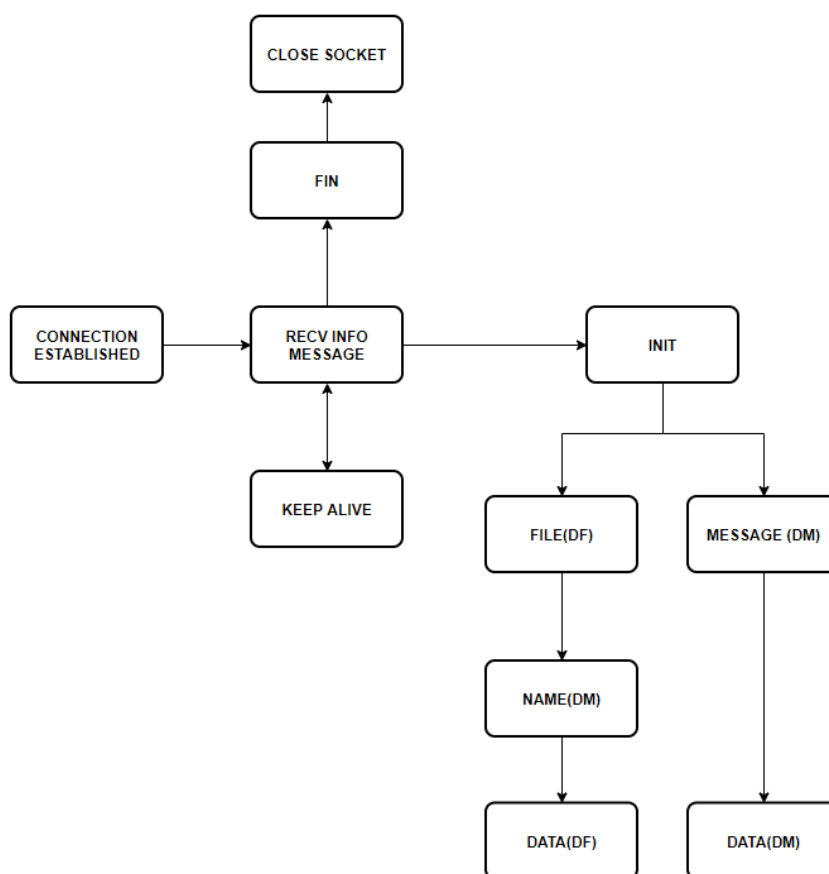
Pri prijímaní sa „rozbaľujú“ jednotlivé polia hlavičky, najprv checksum a typ. Ak nesedí očakávaný typ, alebo checksum, tak sa správa okamžite zahodí a ignoruje, ak je to potrebné vyvolá sa CheckSumError.

Prijatie info správy

Po nadviazaní komunikácie server prechádza do hlavnej slučky prijímania dát `recv_data()`. V tejto procedúre potrebujeme prijať INIT packet, ktorý server informuje o tom, či sa bude posielat' súbor alebo správa. V inite pri správe je straight-forward počet fragmentov správy, po ktorej nasledujú samotné bloky dát. INIT pri súbore je potrebné rozdeliť na posielanie názvu súboru a obsahu prečítaného z daného súboru. Ak je maximálna veľkosť fragmentu menšia ako dĺžka názvu súboru fragmentujeme aj tieto správy. Je to jeden krok navyše, kde vykonáme rovnaký postup ako pri posielaní správy, len s obsahom názvu & prípony súboru.

`recv_info()`, slúži aj ako rózcestie pre správy rôznych typov:

- INIT – pokračovanie v prijatí dát
- KA – udržiavanie spojenia otvorené
- FIN – ukončenie spojenia



Obrázok 8: Rózcestie na serveri po nadviazaní spojenia

Odoslanie info paketu

Klient posiela informačný INIT packet v `send_info(typ, pocet)`. Tu je vhodné si priblížiť ako funguje spoločná procedúra na odosielanie všeobecných dátových packetov.

V super triede `Uzol` procedúra,

`send_data(typ, hdr_info, hdr_struct, raw_data, chyba)`

dostane ako parametre:

- `typ` – DF/DM, čo je v tomto prípade `tuple` jedného alebo dvoch `stringov`
- `hdr_info` – 4B `int` hodnota obsahujúca pri init správach počet fragmentov, alebo ID fragmentu
- `hdr_struct` – do akej podoby - formátu sa má uložiť hlavička – `string`
- `raw_data` – `None` alebo dáta prečítané zo súboru/správy enkódované na `bytes` formát
- `chyba` – zadaná používateľom, ak je > 0 , jedná sa o percentuálnu šancu na poškodenie odosiadaného dátového paketu. Ak je to záporná hodnota, je to ID konkrétne poškodeného paketu.

Ďalší priebeh je už klasický, formátujeme hlavičku do veľkosti akú potrebujeme, vypočítame checksum a dáta odošleme serveru.

Prevod typu na 1B číslo a naspäť:

- Pri odosielaní:
 - Logická operácia OR nastaví bity na základe vstupu.
- Pri prijímaní:
 - Logická operácia AND skontroluje, ktoré bity sú nastavené.

Prijatie a spracovanie segmentov na serveri

Prijatie info správy signalizuje prechod do hlavnej funkcie prijímania fragmentov dát. `recv_fragments(typ, pocet_fragmentov)`.

Server používa triedu `FragmentController` na uchovanie všetkých potrebných počítadiel a hodnôt na správne prijatie posielaných fragmentov. Na základe prijatého počtu fragmentov, ktoré má očakávať a veľkosti bloku si dokáže vypočítať, ktorý bude posledný blok a koľko bude obsahovať fragmentov. Jednoducho povedané, prijímame fragmenty a po každom bloku posielame ACK/NACK správy. Prijaté dáta vždy najprv skontrolujeme, či prišli v poriadku (pomocou výpočtu CRC) a potom spracujeme prijatý fragment v procedúre `process_fragment(recvd_data, expected_type, f_controller)`

returns 0 if OK else 1

V prípade problému alebo erroru v podobe prijatia neočakávaného typu alebo detekcie **RETRANSMISSION** funkcia vráti `err. code 1`, čo vykazuje to, že fragment nebol správne spracovaný a neráta sa do počtu fragmentov súboru, čiže pochopiteľne nebude ani zapísaný do výstupu. Detekcia znovu prijatia už obdržaného paketu funguje, tak že sa skontroluje miesto, na ktoré má patriť a ak sa tam už nachádza iný, tento nový sa zahodí. Ak toto miesto nie je prázdne znamená, to že v minulosti sme už prijali fragment správne a prehlásili o ňom, že je v poriadku. Opätovne prijaté segmenty sa nerátajú do bloku a nemusia byť potvrdené.

Kontrola bloku a žiadanie chýbajúcich/poškodených segmentov

Po každom pakete kontrolujeme, či to nie je posledný z bloku, resp. posledný celkovo. Ak sme prijali 10 fragmentov alebo vypršal timer, skontrolujeme, či máme všetkých 10 alebo toľko, koľko očakávame.

- Máme všetky a spokojne môžeme odoslať ACK správu o potvrdení bloku, taktiež, je potrebné resetovať blokové počítadlá v fragment controlleri.
- Pakety chýbajú/prišli pokazené a je potrebné si ich opätovne vyžiadať.

Pokiaľ nám nesedia počty (alebo vypršal časovač), vo funkcii `obtain_corrupted(..)` skonštruujeme **NACK** správu a pomocou vyššie spomínanej procedúry `send_data(..)` ju odošleme. Ako vyzerá a čo obsahuje sme si už hovorili v predchádzajúcich kapitolách na [obrázku 5](#) a v tabuľkách [2](#) a [3](#). Nakoľko poznáme počet žiadaných paketov, vieme aj kedy ukončiť prijímanie. Rovnako ako v hlavnom cykle, zahadzujeme chybné / poškodené / opakované fragmenty. Tu je možné, že správa ani na druhý krát nepríde v poriadku, preto po vypršaní časovača, znovu pošleme NACK správu. Tento proces skúsime zopakovať viackrát a keď sa to ani na druhý, či tretí krát nepodarí spojenie ukončíme, pretože môžeme prehlásiť, že komunikácia je nestabilná a nespoľahlivá.

Takýmto spôsobom prijímame všetky doručené segmenty a na konci prijatý objekt zapíšeme do želaného súboru alebo správu vypíšeme na štandardný output.

V tomto momente sa server prepne do pasívneho modu a počúva na porte s nadviazaným spojením na KA, INIT, FIN správy podobne ako je to na [obrázku 8](#).

Odosielanie segmentov z klienta

Hlavná procedúra odosielania dát `send_fragments(typ, pocet_fragmentov)` číta časti zo súboru alebo z používateľom zadanej správy. Maximálne máme načítaných naraz iba 10 segmentov (maximálna veľkosť bloku) dát. V cykle opakujeme posielanie cez procedúru `send_data(...)` pokým neobdržíme potvrdenie prijatia všetkých segmentov, resp. blokov. Po každom odoslanom segmente, kontrolujeme, či sa nejedná o posledný z daného bloku alebo posledný celkovo. Následne prerušíme odosielanie a čakáme na doručenie **ACK/NACK** správy. V prípade ACK signálu prehlásime blok ako potvrdený a načítame do pamäte ďalší s tým, že predchádzajúci z pamäte odstránime. Ak obdržíme NACK signál, v procedúre `send_corrupted(data, nack_ids, pocet_fragmentov)` zistíme, ktoré segmenty je potrebné opätovne odoslať a znova odosielame. Ak by bolo potrebné za sebou odosielať rovnaký segment viac krát (v rámci jedného bloku) nezarátame ho do celkového počtu v bloku a v tomto prípade môžeme poslať aj viac ako je prednastavená hodnota 10 správ bez potvrdenia.

Simulácia poškodeného packetu

Pri odosielaní dátových packetov dokážeme simulovať chybu siete, čiže stratu dát. Možností ako vytvoriť takúto chybu je viacero, ale v tomto projekte použijeme upravenie CRC hodnoty, tak, že ju **zvýšime o 1**. Ak po vypočítaní pôvodnej crc, serveru príde iné číslo ako si vypočíta on sám znamená to, že packet je poškodený a zahodí ho.

Toto riešenie by malo byť nepriestrelné, pretože iné možnosti simulácie chyby predstavujú explicitné upravenie dát, hlavičky alebo prepísanie hodnoty crc. Kebyže si zvolíme napríklad, metódu nastavenia hodnoty CRC na 0, môže sa stať, že pri CRC16 sa vypočítaná hodnota UŽ bude rovnať 0 a teda segment nepoškodíme. Táto šanca (**0.001526%**) samozrejme nie je vysoká, ale tento prípad môže nastať. Rovnako nám nič nezaručuje fakt, že ak by sme zmenili dáta, CRC hodnota by nebola rovnaká.

Keep Alive

Finálna podoba Keep Alive ostáva taká aká bola popísaná v návrhu. Pomocou viacnitiťovosti posielame na ďalšom vlákne Keep Alive signály a na hlavnom vlákne, čítame vstup od usera. Po odoslaní dát užívateľ dostane ďalší prompt od programu, s otázkou, či chce opakovať prenos ďalšieho súboru/správy na rovnaký server alebo ukončiť spojenie. Na pozadí sa posielajú KA správy aby spojenie ostalo otvorené. Ak si vyberie rovnaký server, tak opäť musí zadať všetky údaje, okrem IP adresy a portu.

Jedna nevýhoda tohto vlastného vlákna je tá, že aby nevyužívala všetky dostupné prostriedky, musí byť uspaná na minimálne 1 sekundu. V našom skripte využívame **ThreadPoolExecutor** z modulu *concurrent.futures*. Tomuto konateľovi submitneme našu funkciu a spustíme príkaz sleep na začatie KA cyklu. Dôležitý fakt je, že keď sme mu už raz funkciu poslali, nie je možné ju prerušiť. Musíme aj v hlavnom vlákne počkať na terajšie „dospanie“, kedy sa skontroluje atribúta triedy **Client** → **self.ka**, ktorú KeepAlive vlákno vyhodnotí ako *False* a vypne odosielanie KeepAlive packetov, čím vyjde z funkcie. Toto je jediný spôsob ako môžeme predísť nekonečnému cyklu aby využíval náš skript všetky CPU prostriedky.(pri 8vláknovom CPU) 12.5% z CPU time.

V konzole sa vypisujú všetky poslané KA a prijaté ACK packety, odkedy bolo Keep Alive zapnuté. Tieto dva výpisy každých 10 sekúnd môžu byť rušivé a preto je v prompte ešte aj možnosť toggle-núť tieto print-outy aby neprekážali pri zadávaní vstupu.

```
[Rovnaký] target / [toggle] KA spam: / [fin] ukonci spojenie
POSLAL:['KA']
Velkost ODOSLANEHO fragmentu: 3, Data: 0
PRIJAL:['ACK']
Velkost PRIJATEHO fragmentu: 3, Data: 0
rovnaky
Maximalna velkost fragmentov dat(1-1465): 1024
Pre ukoncenie napiste [quit]
Odoslat: [sprava], [subor]: sprava
Zadajte spravu: Testujem.
Zadajte [%] chybných packetov. [-X] pre pokazenie X teho packetu: 0_
```

Obrázok 9: Prompt po odoslaní súboru

Ukončenie spojenia

Klient po každom poslaní správy / súboru dostáva na konci ponuku, či nechce ukončiť spojenie správou FIN. Ak si zvolí túto možnosť, okamžite sa odošle FIN, čo v našom programe je skôr RST z TCP, resp. na ukončenie stačí, aby klient poslal FIN správu a socket sa uzavrie na oboch stranách. Následne je potrebné opäť si zvoliť, ktorú časť bude daná inštancia programu reprezentovať a nadviazať spojenie znovu.

Doplnenie použitých knižníc a popis zdrojových súborov

Okrem knižníc zahrnutých v návrhu na [strane 7](#), sme použili nasledovné doplňujúce knižnice:

- [time](#)
 - Uspenie threadu – `time.sleep(n)` a timer – `time.time()` v ňom.
- [concurrent.futures](#)
 - Vytvorenie dvoch threadov na spustenie KA
- [json](#)
 - parsovanie dát z externého .json súboru
- [random](#)
 - ak si zvolíme simulovať poškodenie paketov, náhodná hodnota figuruje vo výpočte toho, či sa má segment pokaziť alebo nie

V programe, taktiež používame niektoré vlastné „moduly“, z ktorých importujeme použité funkcie.

Štruktúra zdrojových súborov

- `main.py`
 - Hlavný zdrojový súbor, tu sa program spúšťa.
 - Obsahuje vytvorenie hlavných tried a obdržanie vstupu od používateľa.
- `uzol.py`
 - Abstraktná super trieda, od ktorej klient a server dedia.
- `client.py`
 - Implementácia klientskej časti programu.
- `server.py`
 - Implementácia serverovej časti programu.
- `fragment_controller.py`
 - Združuje dôležité počítadlá, a kontroly blokov pri prijímaní dát.
- `logger.py`
 - Zaznamenáva všetky info výpisy a reguluje ich intenzitu.
- `utils.py`
 - Výpočet CRC, Konštanty, `CHKSumError` a spracovanie vstupu.
- `constants.json`
 - Externý súbor, v ktorom sú uložené typy hlavičiek.

Wireshark Dissector

V rámci testovania a overenia správneho fungovania programu bol vyvinutý vlastný dissector v programovacom jazyku Lua pre filtrovanie navrhnutého komunikačného protokolu v programe Wireshark. Wireshark na porte 5015 rozoznáva protokol, ktorý používa skript s názvom: `PKS_M` tzv. M Protocol. V pridelených poliach dokáže vypísať o aký typ správy sa jedná, koľko fragmentov sa bude odosielať, ID posielaných fragmentov, veľkosť payloadu a ID znovu-vyžiadaných segmentov v NACK správach.

Protocol	Length	Info
PKS_M	35	56426 → 5015 Len=3 Type: SYN
PKS_M	35	5015 → 56426 Len=3 Type: SYN,ACK
PKS_M	35	56426 → 5015 Len=3 Type: ACK
PKS_M	39	56426 → 5015 Len=7 Type: INIT,DF
PKS_M	35	5015 → 56426 Len=3 Type: ACK
PKS_M	48	56426 → 5015 Len=16 Type: DM ID: 0
PKS_M	40	5015 → 56426 Len=8 Type: NACK - 0
PKS_M	48	56426 → 5015 Len=16 Type: DM ID: 0
PKS_M	35	5015 → 56426 Len=3 Type: ACK
PKS_M	39	56426 → 5015 Len=7 Type: INIT,DF
PKS_M	35	5015 → 56426 Len=3 Type: ACK
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 0
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 1
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 2
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 3
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 4
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 5
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 6
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 7
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 8
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 9
PKS_M	46	5015 → 56426 Len=14 Type: NACK - 1,3,6,7
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 1
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 3
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 6
PKS_M	1504	56426 → 5015 Len=1472 Type: DF ID: 7
PKS_M	35	5015 → 56426 Len=3 Type: ACK

Obrázok 11: Filter v akcií

```

PKS_M      179 56426 → 5015 Len=147 Type: DF ID: 1730
PKS_M      35 5015 → 56426 Len=3 Type: ACK
PKS_M      35 56426 → 5015 Len=3 Type: KA
PKS_M      35 5015 → 56426 Len=3 Type: ACK
PKS_M      35 56426 → 5015 Len=3 Type: KA
PKS_M      35 5015 → 56426 Len=3 Type: ACK
PKS_M      35 56426 → 5015 Len=3 Type: FIN
PKS_M      35 5015 → 56426 Len=3 Type: ACK

```

Obrázok 10: Wireshark filter → Keep Alive

PKS_M Data
Typ: 32 (DF)
ID posielaného fragmentu: 1106
Data Payload: (1465 bytes)
CRC: 0xf2d8

0020	20	52	04	00	00	1a	f3	0f	b5	c5	75	73	23	a9	00	9e	R...	..us#...
0030	4f	3c	66	a9	34	cc	a6	ac	3e	4c	3e	40	ea	47	3c	75	0<f.4...	>L>@.G<u
0040	aa	ec	98	39	ec	7b	d4	af	2e	c5	20	1e	6a	30	c8	f8	...9. {...	. . .j0..

Obrázok 12: Pohľad na dáta

Záver a zhrnutie

V závere tejto práce môžeme potvrdiť, že sa nám podarilo splniť cieľ implementácie vlastného protokolu, v ktorom vystupujú dva uzly medzi, ktorými prebieha komunikácia výmenou segmentov po internetovej sieti.

Aplikovali sme model client-server a vytvorili tzv. „chat“, v ktorom posielame dáta z odosielača na prijímač. Program umožňuje **odoslať správu alebo celý súbor**. Na zabezpečenie **spoľahlivej komunikácie** sme sa inšpirovali existujúcimi protokolmi (najmä TCP). Všetky dáta musia prísť správne a po **blokokoch segmentov veľkosti 10** posielame zo servera **potvrdenie** vo forme ACK. Signály a typy správ boli dôležitým prvkom na zabezpečenie plynulej komunikácie a propagovania stavu, v akom sa spojenie nachádza pre oba uzly.

Počas komunikácie **dokážeme simulovať chybu** a stratu dát. Program sa vie vysporiadať so zle prijatými údajmi nakoľko, dáta sú kontrolované pomocou **CRC checksumu**. Ak sa táto vypočítaná suma nezhoduje so sumou prijatou alebo segment neprišiel vôbec skonštruujeme ďalší signál s typom **NACK**, ktorý pošleme späť. Takýmto spôsobom si **opätovne vyžiadame poškodené dáta**. Po úspešnom odoslaní údajov udržiavame spojenie aktívne cez **Keep Alive signalizačné správy**, ktoré sa posielajú v cykloch po 10 sekúnd. Ukončenie spojenia je signalizované typom FIN, kedy sa na oboch stranách uzavrie socket.

Na overenie funkčnosti tohto protokolu a programu používame vlastný prispôbosený analyzátor paketov implementovaný v jazyku Lua a zavedený do programu Wireshark.

Celkový výsledok odzrkadľuje čas venovaný tomuto projektu a opäť raz, môžeme využiť nadobudnuté znalosti o sieťovej komunikácii v zložitejších riešeniach v ďalších pokračovaní tohto alebo iného predmetu, či v praxi.

Martin Melišek 2.12.2020