PKS Zadanie 2 – Komunikácia s využitím UDP protokolu

Adam Gábor
AIS ID: 116174
cvičenie: utorok 18:00
cvičiaci: Ing. Matej Janeba

Obsah:

1. Znenie zadania

Študent by mal byť vedieť filtrovať segmenty aplikácie pomocou nástroja Wireshark a vedieť identifikovať typy správ navrhnutého protokolu počas demonštrovania svojej aplikácie na cvičení.

Úloha 1 - Návrh programu a komunikačného protokolu

Hodnotí sa prehľadnosť a zrozumiteľnosť odovzdanej dokumentácie ako aj kvalita navrhovaného riešenia. 5 bodov získa študent, ktorý má v dokumentácií uvedené všetky podstatné informácie o fungovaní jeho programu. Korektne navrhnutú štruktúru hlavičky vlastného protokolu, opis použitej metódy kontrolnej sumy a fungovania ARQ, metódy pre udržanie spojenia, diagram spracovávania komunikácie na oboch uzloch (sekvenčný), popis jednotlivých častí zdrojového kódu (knižnice, triedy, metódy, ...). Podčiarknuté požiadavky sú minimálne.

Úloha 2 - Príprava

Nastaviť a overiť konektivitu medzi 2 uzlami, spustiť na oboch uzloch Wireshark. Spustiť zachytávanie vo Wireshark a nastaviť filter na zobrazenie iba komunikácie vlastného programu. Každý prenos sa kontroluje aj cez Wireshark.

Úloha 3 - Nastavenie IP a port

Body získa študent, ktorého program umožňuje nastaviť na prijímajúcom uzle port, na ktorom počúva a na vysielajúcom uzle IP adresu a port prijímača.

Úloha 4 - Prenos súboru menšieho ako nastavená veľkosť fragmentu

Body získa študent, ktorého program umožňuje úspešne preniesť súbor menší ako je nastavená veľkosť fragmentu podľa pokynov cvičiaceho.

Úloha 5 - Simulácia chyby pri prenose súboru a správy

1 body získa študent, ktorého program umožňuje úspešne preniesť súbor/správu pri simulácii chyby prenosu a má korektne implementovanú detekciu chyby a ARQ metódu. Korektným použitím komplexnejšej ARQ metódy je možné získať ďalší 3b. Príklady ARQ metód su medzi zdrojmi nižšie. Vašim návrhom na zlepšenie sa nebránime. Jednotlivé metódy sú hodnotené nasledovne:

- Stop & wait + 0b (vrátane jedného ACK pre skupinu segmentov)
- Go Back-N + 1b
- Selective Repeat + 2b
- Vylepšená Go Back-N/Selective Repeat + 3b (dynamický window sliding, optimalizácia počtu ACK správ, ...)

Úloha 6 - Prenos 2MB súboru

Body získa študent, ktorého program umožňuje úspešne preniesť súbor s veľkosťou 2MB pri nastavení veľkosti fragmentu podľa pokynov cvičiaceho a uložiť ho ako rovnaký súbor, zobrazuje absolútnu cestu k súboru a počet fragmentov spolu s ich veľkosťou.

Úloha 7 - Udržiavanie spojenia

2 body získa študent, ktorého program po prenesení súboru udržuje spojenie pomocou vlastných signalizačných správ a zobrazí informáciu, ak bolo spojenie prerušené. Korektným použitím komplexnejšej metódy pre udržanie spojenia je možné získať ďalší 1b.

Úloha 8 - Finálna dokumentácia a kvalita spracovania

Hodnotí sa prehľadnosť a zrozumiteľnosť odovzdanej dokumentácie ako aj kvalita pracovania celového riešenia. 3 body získa študent, ktorý má v dokumentácií uvedené všetky podstatné informácie o fungovaní jeho programu vrátane zmien, ktoré nastali v implementácii oproti návrhu. Dokumentácia tiež musí obsahovať aspoň 1 ukážku testovacieho scenáru (ideálne ako screeny z Wireshark).

Úloha 9 - Doplnená funkčnosť (doimplementácia) priamo na cvičení.

1 bod získa študent, ktorý doimplementuje úlohu v jej plnom rozsahu a predvedie jej funkčnosť bez toho, aby program padal alebo vyhadzoval akékoľvek chybové hlášky súvisiace s touto úlohou.

Kompletné znenie zadania: https://github.com/fiit-ba/pks-course/tree/main/202324/assignments/2_communication_over_udp

2. Realizácia projektu

Projekt som sa rozhodol realizovať v programovacom jazyku **Python** v IDE Pycharm 2023.2 s použitím knižnice **Socket**. Projekt testujem na svojich dvoch laptopoch, MSI GF 63 a MacBook Air 2019. Obe zariadenia fungujú ako klient aj server, a teda medzi nimi je možná výmena správ a odosielanie súborov oboma smermi.

3. Návrh protokolu

Vlastný protokol, ktorý budem vytvárať, bude vnoreným protokolom nad transportným protokolom UDP. Takýto protokol mi na rozdiel od jednoduchého UDP pomôže **lepšie vykonávať základné funkcionality** sieťovej komunikácie napríklad vďaka možnosti nadviazať spojenie, detegovať straty či lepšie reagovať na možné chyby v komunikácii medzi klientom a serverom.

3.1. Návrh hlavičky protokolu

Hlavička môjho protokolu sa bude v odoslanom packete (resp. datagrame) nachádzať za hlavičkou protokolov Ethernet II, IPv4 a UDP.

Byte č.	1-2	3-4	5-6	7-10	11 - koniec
	ID (vlajka)	počet fragmentov	poradie fragmentu	cyclic redundancy check	dáta

Tabuľka 1: štruktúra hlavičky môjho protokolu

ID (decimálne)	ID (hexadecimálne)	kód a význam
01	00 01	SYN - začiatok spojenia
02	00 02	ACK - prijatie správnych dát
03	00 03	NACK - prijatie chybných dát
04	00 04	DATA - odoslanie dát

05	00 05	KA - keep-alive správa
06	00 06	SW - požiadavka na zmenu roly klient/server
07	00 07	FIN - ukončenie spojenia

Tabuľka 2: hlavné typy ID správ v mojom protokole

Keďže je veľkosť mojej hlavičky 10B, dokopy bude v jednom packete možné prepraviť 1500 (bez Eth II hlavičky) - 20B (IPv4 hlavička za predpokladu najkratšej možnej dĺžky) - 8B (UDP hlavička obsahujúca napríklad porty) - 10B (hlavička môjho protokolu) = **1462 bajtov**, a táto veľkosť bude pre účely fragmentácie najväčšou možnou veľkosťou odoslaného "balíčka".

Čo sa týka maximálnej veľkosti odoslaného súboru, vďaka veľkosti časti hlavičky, do ktorej ukladám informácie o **fragmentácii**, ktorá je 2B, mi protokol umožňuje vytvoriť maximálne 65025 fragmentov, každý o maximálnej veľkosti dát 1462 bajtov. Celková maximálna veľkosť odoslaného fragmentovaného súboru pomocou môjho protokolu je teda približne 95 miliónov bytov ≈ 90 megabytov. Takýto počet fragmentov mi komfortne umožňuje poslať 2MB súbor, ktorý zadanie vyžaduje.

Pre účely **jednoduchosti výpočtov** a prácu s párnymi číslami som hlavičku zväčšil z 9B na 10B - pridal som jeden bajt do údaju o type správy. Toto rozhodnutie síce zväčšuje hlavičku, avšak takisto potenciálne ponúka možnosť vytvorenia tisícov **špecializovaných ID / vlajok**, ako napríklad samostatná vlajka pre každý odoslaný typ (koncovku) súboru či typ správy.

3.2. Návrh metódy CRC

Typ CRC, ktorý som pre implementáciu v zadaní zvolil, je **CRC-32**. Tento typ cyclic redundancy check je považovaný za veľmi bezpečný a pre účely môjho protokolu aj veľmi ľahko implementovateľný v Pythone vďaka funkcii crc32() z knižnice zlib. Ako napovedá aj názov, CRC32 generuje 32-bitový kontrolný súčet (checksum) a používa štandardizovaný polynóm s hexadecimálnou hodnotou 0x04C11DB7. Tento polynóm určuje, ktorý byte v dátach je XOR-ovaný aktuálnou hodnotou CRC.

3.3. Návrh ARQ metódy a metódy pre udržanie spojenia

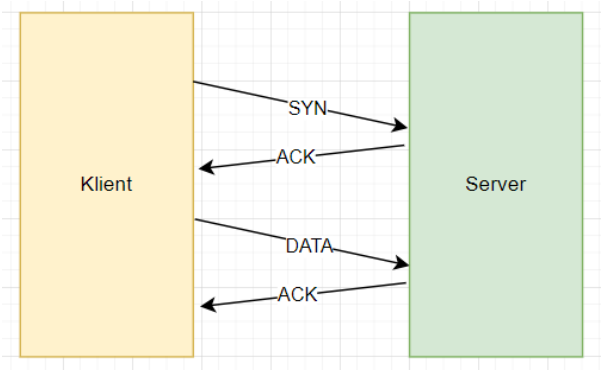
Pre môj program som si počiatočne vybral ARQ metódu stop & wait. V rámci tejto metódy posiela odosielajúca strana dáta a potom čaká, až kým jej zo strany prijímateľa nepríde potvrdenie o korektnom prijatí nepoškodenej správy. V prípade, že také potvrdenie po určitom čase nepríde, odosielateľ správu odošle znova.

Hlavička programu mi umožňuje moje rozhodnutie v priebehu implementácie programu zmeniť, keďže môžem pridať dva bity, zbaviť sa dodatočného bitu zo začiatku hlavičky určeného pre typ správy a tri bity priradiť poradovému číslu (sequence number), čo by mi umožnilo implementovať napríklad ARQ metódu Selective Repeat.

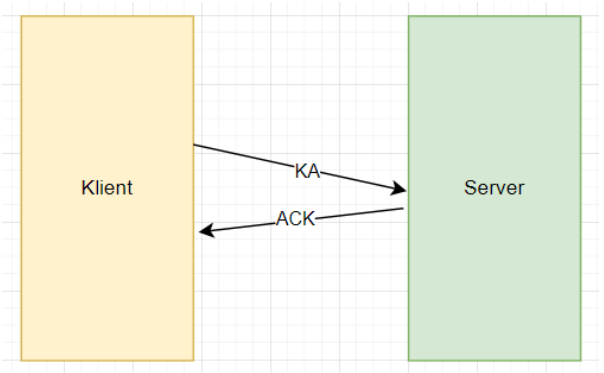
Pre udržanie spojenia som zvolil metódu keep-alive. Správu s ID keep-alive bude klient posielať serveru každých 10 sekúnd. Server následne žiadosť o udržanie spojenia potvrdí. V prípade, že takúto správu po dobu jednej minúty klient nepošle, bude spojenie prerušené.

3.4. Diagram spracovania komunikácie

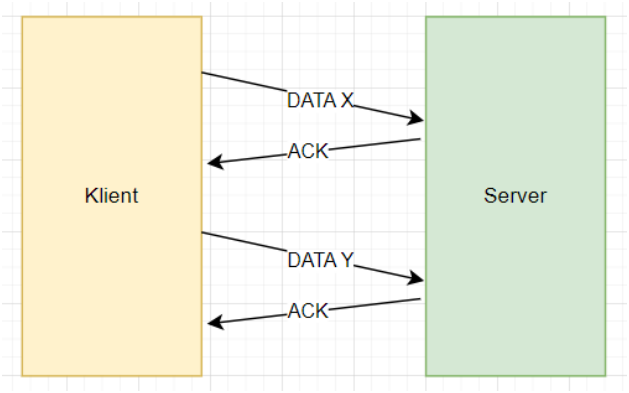
Obr. 1: Inicializácia spojenia a poslanie dát:



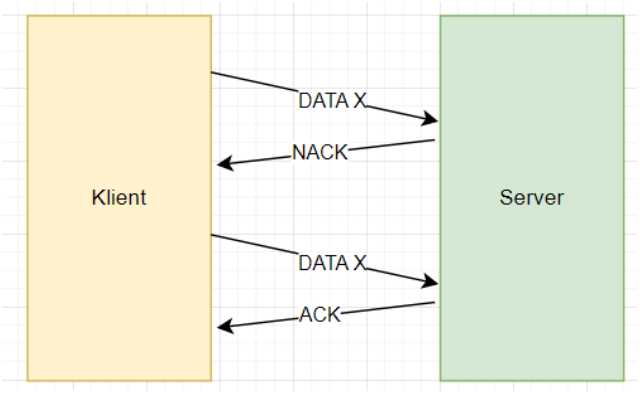
Obr. 2:Udržanie spojenia:



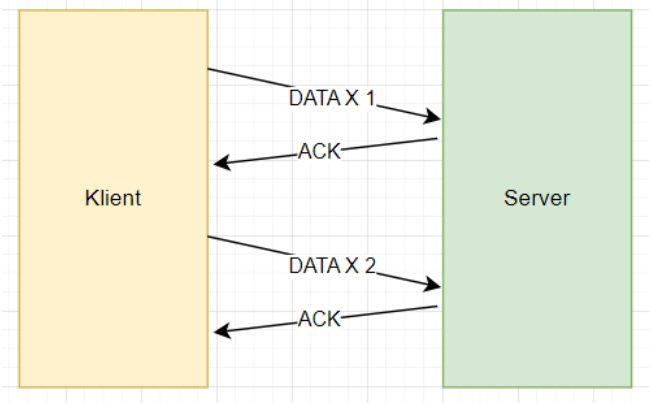
Obr. 3: Prijatie dvojice správ bez problémov:



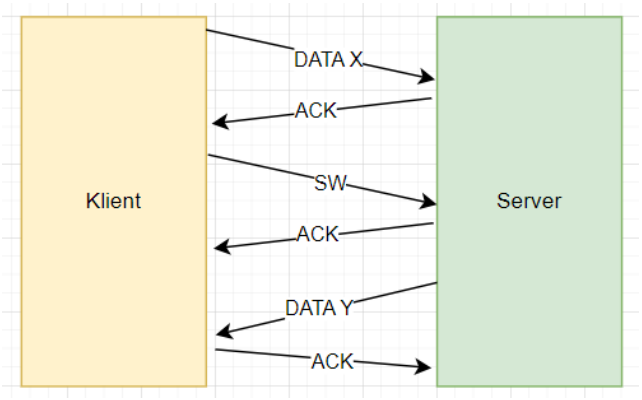
Obr. 4: Prijatie správy s problémom, následné znovuodoslanie správy a jej prijatie:



Obr. 5:Prijatie fragmentovanej správy:

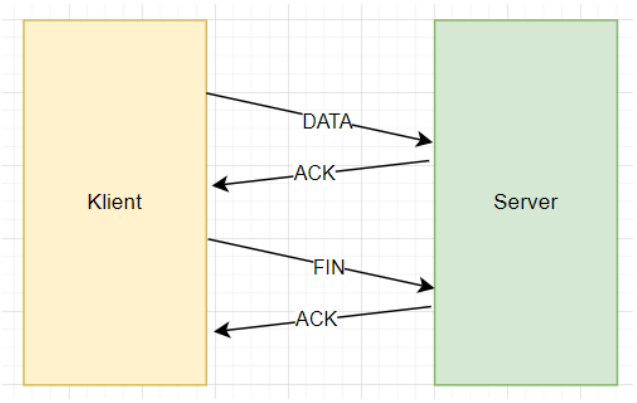


Obr. 6:Zmena úlohy odosielateľ / prijímateľ:



*pozícia klienta a servera sa po zmene úloh pochopiteľne vymení

Obr. 7: Ukončenie spojenia:



4. Implementácia zadania

4.1. Fungovanie programu

Program funguje ako CLI aplikácia. Obe zariadenia najprv program nastavia, nadviažu spojenie a potom je im umožnená jednosmerná komunikácia. Klient má možnosť serveru posielať správy, súbory, fragmentované správy a takisto zmeniť roly, takže bude pôvodný server môcť komunikovať s pôvodným klientom. Klient môže program na oboch zariadeniach ukončiť. Klient posiela každých 10 sekúnd serveru keep-alive správu. V prípade nedoručenia keep-alive správy v určitom časovom rozsahu server spojenie aj program ukončí.

Program je otestovaný a fungujúci na operačných systémoch Windows, MacOS aj Linux. Program môže prijímať cesty k súborom vo formáte s lomítkom (/) a takisto aj spätným lomítkom (\), teda má na všetkých troch operačných systémoch 100%-tnú funkcionalitu.

4.1.1. Zmeny oproti návrhu

V porovnaní s prvotným návrhom som v implementácii spravil viacero zmien. Veľkou zmenou bola **zmena hlavičky**, a to z 10 bytov iba na 8. Dôvodom bolo uvedomenie, že keďže využívam ARQ metódu Stop & Wait, hlavička **nemusí** niesť informáciu o počte a poradí fragmentov, lebo sa vždy posielajú zaradom (fragment číslo n sa nepošle, kým nie je fragment číslo n-1 prijatý a uložený). Z toho dôvodu nemôže zamenenie poradia fragmentov nastať, a tak mi stačí v hlavičke zadávať iba veľkosť doposiaľ odoslanej časti súboru, na čo som vyčlenil 3 byty a čo mi umožňuje poslať súbor s maximálnou veľkosťou približne **16 megabytov**. Z hlavičky som taktiež odstránil jeden byte pre vlajku, keďže dva byty sú pre zakódovanie 9 stavov, ktoré používam, nepotrebné.

Čo sa týka vlajok samotných, pridal som dve vlajky, ktoré používam pri prenášaní súboru a fragmentovaných správ. Ide o vlajky **FILE** a **EOF**, ktoré značia posielanie časti súboru a ukončenie posielania súboru. Ich funkcionalita je popísaná v kapitolách nižšie.

Zvyšok kódu plní svoju funkciu presne podľa môjho návrhu.

```
FLAGS = {
    'SYN': '01',
    'ACK': '02',
    'NACK': '03',
    'MSG': '04',
    'FILE': '05',
    'KA': '06',
    'SW': '07',
    'FIN': '08',
    'EOF': '09'
}
```

Obr. 8 - slovník vlajok zakódovaných v hlavičke

4.1.2. Inicializácia a ovládanie programu

Na začiatku programu musia oba uzly nastaviť potrebné dáta - v prípade serveru to je **IP adresa**, **port**, na ktorom počúva, a **priečinok**, kde sa uložia prijaté súbory. V prípade klienta sú to všetky tieto informácie a navyše aj IP servera, na ktorý sa pripojí.

Po pripojení (SYN-ACK handshaku) čaká konzola donekonečna na vstup klienta. Vstupom môže byť textová správa alebo špeciálna inštrukcia.

```
Enter your IP address: 10.10.46.154
Enter the port number: 50000
Enter the directory for saving files (press enter for default):
Enter 's' to act as server, anything else for client:
Enter the target IP address: 10.10.9.122
```

Obr. 9: Inicializácia programu zo strany klienta pripájajúceho sa na IP 10.10.9.122

4.2. Dekódovanie a kódovanie hlavičky

Oba uzly odosielajú a prijímajú datagramy, a teda musia oba uzly vedieť zakódovať a dekódovať hlavičku. Na to využívam funkcie **encode_header()** a **decode_header()**. Funkcia kódujúca hlavičku na prvý byte umiestni vlajku podľa inštrukcií zo slovníka, na ďalšie tri v prípade odosielania súboru jeho veľkosť v bytoch a na posledné 4 byty návratovú hodnotu z funkcie **crc32()**, ktorú mi sprostredkúva knižnica zlib. Funkcia, ktorá hlavičku prijatej správy dekóduje, z nej potom vlajku vyčíta a zráta nový checksum, ktorý porovná s tým v hlavičke. Pri rátaní checksumu beriem do úvahy iba dáta za mojou hlavičkou, nie celý packet.

```python
def encode_header(flag, message=b'', size=0):
    flag_hex = FLAGS[flag]
    flag_bytes = bytes.fromhex(flag_hex)
    size_bytes = size.to_bytes( length: 3, byteorder='big')
    checksum = compute_checksum(message)
    return flag_bytes + size_bytes + checksum.to_bytes( length: 4, byteorder='big')

5 usages
def decode_header(header, message, simulate_error=False):
    flag_bytes = header[:1]
    flag_hex = flag_bytes.hex()
    size = int.from_bytes(header[1:4], byteorder='big')
    received_checksum = int.from_bytes(header[4:], byteorder='big')
    calculated_checksum = compute_checksum(message)

    if simulate_error and message.decode('utf-8') == 'test' and random.random() < 0.9:
        calculated_checksum += 1 #naschval zvysi checksum o 1, aby nebol spravny

    if received_checksum != calculated_checksum:
        print("Checksum mismatch in header.")
        return None, size

    for name, value in FLAGS.items():
        if value == flag_hex:
            return name, size
    return None, size
```

Obr. 10: Funkcie na kódovanie a dekódovanie hlavičky, funkcia decode_header navyše

obsahuje možnosť simulovať nesprávny checksum

4.3. Klient mód

Kód módu klienta spočíva v dvoch úlohách na dvoch vláknach. Jednou z nich je čakanie na vstup z konzoly, kde užívateľ zadáva správy a príkazy, a tou druhou je **odosielanie keep-alive** správ. Keďže implementujem ARQ metódu Stop & Wait, po každom odoslanom pakete klient čaká na "ACK" zo strany serveru. V prípade, že správu s touto vlajkou nedostane, alebo dostane správu s inou vlajkou, odošle správu **znova**. Program pozná štyri druhy špeciálnych správ od užívateľa:

file	umožní klientovi vybrať súbor a veľkosť fragmentu, takisto umožní poslať fragmentovanú správu
switch	vymení rolu klienta a serveru
end	ukončí program na oboch uzloch
test	úmyselne vypočíta chybný checksum
testR	úmyselne čaká s odpoveďo

Tabuľka 3: typy a funkcie špeciálnych vstupov klienta

Keď chce užívateľ v roli klienta poslať **fragmentovaný súbor alebo správu**, zvolí špeciálnu inštrukciu "file". Po tom, čo túto inštrukciu zvolí, sa ho program opýta na cestu k súboru, resp. znenie správy. Keď chce užívateľ namiesto súboru odoslať správu, začne ju výkričníkom (teda vstup bude napr. !ahoj Janko). V takomto prípade program vie, že **nemusí** otvárať žiadne súbory a do bufferu namiesto bajtov súboru ukladá bajty znakov. Po špecifikovaní tela fragmentovaných správ užívateľ zadáva aj veľkosť jednotlivého fragmentu. Maximálna možná veľkosť fragmentu je **1464** (1500 - IP hlavička - UDP hlavička - moja hlavička), minimálna veľkosť je 1 byte. Odoslanie inštrukcie s posielaním fragmentov spustí funkciu **send_file()**, ktorá súbor (resp. správu) rozdelí na fragmenty žiadanej veľkosti a postupne ich posiela. Aj pri posielaní fragmentov platí, že užívateľ najprv čaká na potvrdenie, že bol predošlý odoslaný fragment prijatý, a až potom odosiela ďalší. Po odoslaní posledného fragmentu posiela klient špeciálnu správu s vlajkou **"EOF" = end of file**, ktorá serveru signalizuje, že nemusí čakať na ďalšie byty a súbor môže uložiť. V správe EOF je zároveň prenesený aj názov, pod ktorým sa súbor na serveri uloží.

```python
def client_mode(local_ip, target_ip, port, save_directory):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    client_socket.settimeout(5)

    def send_keep_alive():
        while keep_alive_running:
            try:
                client_socket.sendto(encode_header('KA'), (target_ip, port))
                time.sleep(10)
            except Exception as e:
                print(f"Error sending keep-alive: {e}")
                break

    global keep_alive_running
    keep_alive_running = True
    keep_alive_thread = threading.Thread(target=send_keep_alive, daemon=True)
    keep_alive_thread.start()

    client_socket.sendto(encode_header('SYN'), (target_ip, port))
    print("SYN sent. Waiting for ACK.")
```

```python
while True:
    try:
        data, _ = client_socket.recvfrom(1024)
        flag, _ = decode_header(data[:8], data[8:])

        if flag == 'ACK':
            print("ACK received. Connection established.")
            break
        else:
            print("Unexpected flag received. Resending SYN.")
            client_socket.sendto(encode_header('SYN'), (target_ip, port))

    except socket.timeout:
        print("Timeout waiting for ACK. Resending SYN.")
        client_socket.sendto(encode_header('SYN'), (target_ip, port))
```

Obr. 11: Funkcia pre klientsky uzol, cyklus nadväzujúci spojenie

```
while True:
    message = input("Enter message to send, 'file' to send a file, 'end' to close, 'switch' to switch roles: ")
    if message == 'file':
        file_path = input("Enter the file path: ")
        chunk_size = int(input("Enter the chunk size (1-1466): "))
        chunk_size = max(1, min(chunk_size, 1466))  #velkost fragmentu od 1 do 1466
        send_file(client_socket, file_path, target_ip, port, chunk_size)
    elif message == 'end':
        keep_alive_running = False
        keep_alive_thread.join()
        client_socket.sendto(encode_header('FIN'), (target_ip, port))
        print("FIN sent. Closing connection.")
        break
    elif message == 'switch':
        keep_alive_running = False
        keep_alive_thread.join()
        client_socket.sendto(encode_header('SW'), (target_ip, port))
        print("SW sent. Switching to server mode.")
        client_socket.close()
        server_mode(local_ip, port, save_directory)
        break
    else:
        message_bytes = message.encode('utf-8')
        while True:
            client_socket.sendto(encode_header( flag: 'MSG', message_bytes) + message_bytes, (target_ip, port))
            try:
                ack_data, _ = client_socket.recvfrom(1024)
                ack_flag, _ = decode_header(ack_data[:8], ack_data[8:])
                if ack_flag == 'ACK':
                    print("ACK received for the message.")
                    break
                elif ack_flag == 'NACK':
                    print("NACK received. Resending the message.")
                    continue
            except socket.timeout:
                print("Timeout waiting for ACK/NACK. Resending the message.")
```

Obr. 12: Cyklus čakajúci na vstup užívateľa a odosielanie správ / špeciálnych datagramov

4.4. Server mód

Uzol, ktorý funguje ako server, sa **inicializuje ako prvý**. Potom čaká na to, kým sa k nemu klient pripojí a zobrazuje správy a súbory, ktoré mu posiela. Server takisto čaká na keep-alive správy, na ktoré odpovedá ACK-om. V prípade, že keep-alive správu od klienta po dobu 120 sekúnd nedostane, ukončí spojenie aj program.

```python
def server_mode(server_ip, port, save_directory):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind((server_ip, port))

    last_keep_alive_time = time.time()
    file_buffer = b''
    message_buffer = b''
    receiving_file = False
    receiving_message = False
    fragment_counter = 0

    while True:
        try:
            server_socket.settimeout(60)
            data, addr = server_socket.recvfrom(1472)
            header = data[:8]
            message = data[8:]

            flag, size = decode_header(header, message, simulate_error=True)
            if flag is None:
                server_socket.sendto(encode_header('NACK'), addr)
                continue
```

```python
        except socket.timeout:
            current_time = time.time()
            if current_time - last_keep_alive_time > 60:
                print("Keep-alive message timeout. Closing connection.")
                break
```

Obr. 13: Nastavenie serverového uzla a funkcionalita keep-alive

Pri prijímaní **súboru** alebo **fragmentovanej správy** server ukladá prijaté dáta do bufferu až kým nedostane správu EOF. V tomto prípade dáta uloží / vypíše, a takisto vypíše aj ich špecifiká ako je veľkosť súboru alebo celkový počet prijatých datagramov.

```python
if flag == 'FILE':
    fragment_counter += 1
    print(f"Receiving fragment {fragment_counter}...")
    if not receiving_file and not receiving_message:
        file_buffer = b''
        message_buffer = b''
        #zisti ci ide o subor alebo spravu
        if data[:8].endswith(b'!'):
            receiving_message = True
        else:
            receiving_file = True

    if receiving_message:
        message_buffer += message
    else:
        file_buffer += message

    server_socket.sendto(encode_header('ACK'), addr)
elif flag == 'EOF':
    eof_message = message.decode('utf-8')
    if eof_message.startswith('!'):
        print(f"Complete message received: {eof_message[1:]}")
        message_buffer = b''
    else:
        # It's a file, save it
        file_name = eof_message.split('\\')[-1].split('/')[-1]  #zisti meno suboru, riesi / aj \ pre vsetky OS
        full_path = os.path.join(save_directory, file_name) if save_directory else file_name
        with open(full_path, 'wb') as file:
            file.write(file_buffer)
        print(
            f"EOF received. File transmission complete, file name: {file_name}, total size: {size} bytes in {fragment_counter} chunks
        file_buffer = b''
    fragment_counter = 0
    receiving_file = False
    server_socket.sendto(encode_header('ACK'), addr)
```

Obr. 14: Kód pre prijímanie fragmentovaných správ a súborov

```python
elif flag == 'KA':
    last_keep_alive_time = time.time()
    print("Keep-alive message received")
    server_socket.sendto(encode_header('ACK'), addr)
elif flag == 'FIN':
    print("FIN received. Closing connection.")
    server_socket.sendto(encode_header('ACK'), addr)
    break
elif flag == 'SW':
    print("Switching roles. This node is now a client.")
    server_socket.sendto(encode_header('ACK'), addr)
    server_socket.close()
    client_mode(server_ip, addr[0], port, save_directory)
    break
elif flag == 'MSG':
    print(f"Received message: {message.decode('utf-8')}")

    if message.decode('utf-8') == "test":
        print("TEST MESSAGE RECEIVED")
        time.sleep(7)

    server_socket.sendto(encode_header('ACK'), addr)
```

Obr. 15: Prijímanie špeciálnych typov správ a takisto bežných textových správ

5. Testovanie riešenia

Program som testoval na dvojici laptopov prepojených pomocou **Wi-Fi pripojenia** v rovnakej sieti. Odoslané aj prijaté datagramy som zachytával pomocou programu Wireshark.

Vyskúšal som "bežné používanie", prerušenie spojenia na oboch stranách a rôzne sekvencie správnych aj chybových vstupov.

Program **neošetruje** fatálne chyby terminujúce bežiaci program v konzole, akým je napríklad snaha poslať datagram na neexistujúcu / nedostupnú IP adresu, čo má za následok to, že po odpojení serveru častokrát na strane pripojeného klienta zlyhá.

5.1. Chybovosť datagramu a prerušenie spojenia

Program reaguje vďaka mechanizmu **keep-alive** na prerušenie spojenia zo strany klienta. Kým dôjde počítadlo keep-alive k nule a kým sa spojenie zo strany serveru zruší, klient má stále šancu pripojiť sa k serveru znova a pokračovať v komunikácii. V prípade, že sa preruší spojenie na strane serveru, program na strane klienta kvôli fatálnej chybe programovacieho jazyka zlyhá.

Mechanizmus **Stop & Wait** zaisťuje, že v prípade, že klientovi nepríde potvrdenie o prijatí datagramu serverom v určitom čase (v prípade môjho programu 5 sekúnd), klient pošle datagram s rovnakým obsahom znova. Túto skutočnosť dokáže môj program simulovať odoslaním špeciálnej inštrukcie **"test"**, po ktorej doručení server istú dobu čaká, kým doručenej potvrdí. Doručenie nesprávnych dát vie server simulovať nastavením vlajky **simulate_error** v kóde serveru na True.

Server má v takomto prípade 90% šancu, že checksum doručenej správy vyráta s chybou a odošle klientovi **správu o doručení poškodených dát**, čo prinúti klienta poslať správu znova.

5.2. Testovací scenár - výstup softvéru Wireshark

V tomto testovacom scenári som vyskúšal všetky funkcionality programu. Klienta som pripojil k serveru, vyskúšal poslať textovú správu, potom súbor s veľkosťou fragmentov 1400. Potom som roly vymenil a z nového klienta poslal súbor s veľkosťou fragmentov 200. Následne som spojenie ukončil. Tu je "premávka" programu, ktorý som zachytil pomocou softvéru Wireshark a vytvorenia vlastného stĺpca "FLAG" po vyfiltrovaní portu, ktorý som zadal pri vstupe:

udp.port == 50000

No.	Time	Source	Destination	Proto	FLAG	Info
136	2.134214	10.10.46.154	10.10.9.122	UDP	01…	51535 → 50000 Len=8
138	2.145424	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
266	5.871710	10.10.46.154	10.10.9.122	UDP	04…	51535 → 50000 Len=14
267	5.874941	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
493	12.149608	10.10.46.154	10.10.9.122	UDP	06…	51535 → 50000 Len=8
494	12.151860	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8

Obr. 16: Nadviazanie spojenia (01 - SYN, 02 - ACK), odoslanie správy (04 - MSG, 02 - ACK) a keep-alive datagramu (06 - KA, 02 - ACK)

972	19.759557	10.10.46.154	10.10.9.122	UDP	05…	51535 → 50000 Len=1408
974	19.763100	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
975	19.763179	10.10.46.154	10.10.9.122	UDP	05…	51535 → 50000 Len=1408
976	19.763727	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
977	19.763778	10.10.46.154	10.10.9.122	UDP	05…	51535 → 50000 Len=1408
978	19.767155	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
979	19.767234	10.10.46.154	10.10.9.122	UDP	05…	51535 → 50000 Len=1408
980	19.768401	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
981	19.768554	10.10.46.154	10.10.9.122	UDP	05…	51535 → 50000 Len=1408
982	19.771017	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
983	19.771112	10.10.46.154	10.10.9.122	UDP	05…	51535 → 50000 Len=1408
984	19.771558	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
985	19.771628	10.10.46.154	10.10.9.122	UDP	05…	51535 → 50000 Len=952
986	19.774135	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
987	19.774258	10.10.46.154	10.10.9.122	UDP	09…	51535 → 50000 Len=54
988	19.774683	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8

Obr. 17: Odosielanie súboru v datagramoch veľkosti 1400 + 8 hlavička, na konci odoslanie cesty k súboru s vlajkou 09 - EOF

1695	32.163983	10.10.46.154	10.10.9.122	UDP	07…	51535 → 50000 Len=8
1696	32.168474	10.10.9.122	10.10.46.154	UDP	02…	50000 → 51535 Len=8
1697	32.170551	10.10.9.122	10.10.46.154	UDP	06…	59873 → 50000 Len=8

Obr. 18: Vykonanie inštrukcie switch datagramom s vlajkou 07 - SW, potvrdenie a následné prehodenie, Ďalší keep-alive bol už poslaný bývalým serverom na vopred stanovený komunikačný port

5928	47.501122	10.10.9.122	10.10.46.154	UDP	05…	59873 → 50000 Len=208
5929	47.501299	10.10.46.154	10.10.9.122	UDP	02…	50000 → 59873 Len=8
5930	47.501897	10.10.9.122	10.10.46.154	UDP	05…	59873 → 50000 Len=208
5931	47.502045	10.10.46.154	10.10.9.122	UDP	02…	50000 → 59873 Len=8
5932	47.503399	10.10.9.122	10.10.46.154	UDP	05…	59873 → 50000 Len=208
5933	47.503521	10.10.46.154	10.10.9.122	UDP	02…	50000 → 59873 Len=8
5934	47.503962	10.10.9.122	10.10.46.154	UDP	05…	59873 → 50000 Len=208
5935	47.504116	10.10.46.154	10.10.9.122	UDP	02…	50000 → 59873 Len=8
5936	47.506056	10.10.9.122	10.10.46.154	UDP	05…	59873 → 50000 Len=176
5937	47.506236	10.10.46.154	10.10.9.122	UDP	02…	50000 → 59873 Len=8
5938	47.506564	10.10.9.122	10.10.46.154	UDP	09…	59873 → 50000 Len=14
5939	47.507759	10.10.46.154	10.10.9.122	UDP	02…	50000 → 59873 Len=8
6113	52.180219	10.10.9.122	10.10.46.154	UDP	06…	59873 → 50000 Len=8
6114	52.180378	10.10.46.154	10.10.9.122	UDP	02…	50000 → 59873 Len=8
6500	62.178766	10.10.9.122	10.10.46.154	UDP	08…	59873 → 50000 Len=8
6501	62.179004	10.10.46.154	10.10.9.122	UDP	02…	50000 → 59873 Len=8

Obr. 19: Odosielanie datagramov veľkosti 200 + 8 hlavička, následné doručenie datagramu s vlajkou 09 - EOF, potom doručenie jedného keep-alive datagramu a napokon špeciálna inštrukcia s vlajkou 08 - FIN, odpoveď 02 - ACK a koniec programu na oboch uzloch

V druhej časti simulácií som simuloval odoslanie správy “test”, pri ktorej má server 90% šancu spočítať checksum s chybou, a takisto “testR”, pri ktorej server úmyselne nechá klienta preposlať datagram so správou:

102	5.760168	10.10.46.154	10.10.9.122	UDP	04…	52647 → 50000 Len=12
103	5.804109	10.10.9.122	10.10.46.154	UDP	03…	50000 → 52647 Len=8
104	5.804361	10.10.46.154	10.10.9.122	UDP	04…	52647 → 50000 Len=12
105	5.806703	10.10.9.122	10.10.46.154	UDP	03…	50000 → 52647 Len=8
106	5.806909	10.10.46.154	10.10.9.122	UDP	04…	52647 → 50000 Len=12
107	5.809244	10.10.9.122	10.10.46.154	UDP	03…	50000 → 52647 Len=8
108	5.809447	10.10.46.154	10.10.9.122	UDP	04…	52647 → 50000 Len=12
109	5.811829	10.10.9.122	10.10.46.154	UDP	03…	50000 → 52647 Len=8
110	5.811982	10.10.46.154	10.10.9.122	UDP	04…	52647 → 50000 Len=12
111	5.814693	10.10.9.122	10.10.46.154	UDP	03…	50000 → 52647 Len=8
112	5.814848	10.10.46.154	10.10.9.122	UDP	04…	52647 → 50000 Len=12
113	5.817276	10.10.9.122	10.10.46.154	UDP	02…	50000 → 52647 Len=8

Obr. 20: Odoslanie správy “test” dĺžky 4 + 8 hlavička, s piatimi nasledujúcimi odpoveďami 03 - NACK signalizujúcimi príjem poškodených dát, dáta boli preposielané, až kým neprišlo z druhej strany ACK

282	3.817354	10.10.46.154	10.10.9.122	UDP	04…	63804 → 50000 Len=14
283	3.820135	10.10.9.122	10.10.46.154	UDP	02…	50000 → 63804 Len=8
425	5.716906	10.10.46.154	10.10.9.122	UDP	04…	63804 → 50000 Len=13
746	10.727009	10.10.46.154	10.10.9.122	UDP	04…	63804 → 50000 Len=13
1324	19.723998	10.10.9.122	10.10.46.154	UDP	02…	50000 → 63804 Len=8

Obr. 21: Odoslanie bežnej správy (04 - MSG, 02 - ACK) a následne odoslanie špeciálnej testovacej správy "testR", po ktorej server naschvál s odpoveďou ACK čaká, aby ju klient znovupreposlal (znovuodosielacia doba je 5s. bez odpovede)

6. Záver

Môj program plní všetky požiadavky dané zadaním. Funguje rýchlo a spoľahlivo. Klient dokáže serveru okrem textových správ posielať aj súbory, a to do veľkosti približne 16 megabytov. Program efektívne pracuje s miestom a správne simuluje komunikáciu medzi dvoma uzlami. Dokáže simulovať rôzne chyby ako nedoručenie správy o prijatí datagramu alebo prijatie narušeného datagramu.

Program nedokáže správne zvládať všetky chybové hlášky vyhodené programovacím jazykom Python a takisto neumožňuje uzlu pracujúcemu ako server iniciovať výmenu rolí. Tieto možnosti neboli implementované z dôvodu vysokej náročnosti a nedostatočného množstva času, a takisto preto, že by výrazne narušili fungovanie dovtedy fungujúceho kódu.

7. Zdroje

Zdroje použité k pochopeniu konceptu protokolov na transportnej vrstve:

- PKS 04: Internet Protocol (IP), ICMP, ARP UDP, TCP
- PKS 06: Transport Layer
- PKS 07: TCP riadenie toku dát
- https://github.com/fiit-ba/pks-course/tree/main/202324/

Zdroje použité pri návrhu:

- https://chat.openai.com/chat/ - *využité k pochopeniu fungovania rôznych ARQ metód a CRC metódy CRC-32*

Zdroje použité k programovaniu v jazyku Python:

- https://realpython.com/intro-to-python-threading/ - *využité pri programovaní vlákien*
- *https://docs.python.org/3/tutorial/index.html - využité pri programovaní dátových štruktúr*

Zdroje použité pri programovaní komunikátora:

- https://chat.openai.com/chat/ - *využité k programovaniu pomocných funkcií ako calculate_checksum(), decode_header() a encode_header() a niektorých častí veľkých funkcií, najmä vláknach pri keep-alive metóde*

Zdroj použitý pri tvorbe diagramov, pre účely testovania a tvorbe a dokumentácie:

- https://app.diagrams.net/
- https://www.wireshark.org/