Štefan Dudaško, AIS ID: 96835
Cvičenie pondelok: 17:00

Štefan Dudaško,
Fakulta Informatiky a informačných technológii

PKS - Zadanie 2
Komunikácia s využitím UDP protokolu

Štefan Dudaško, AIS ID: 96835
Cvičenie pondelok: 17:00

Zadanie

Navrhnite a implementujte program s použitím vlastného protokolu nad protokolom UDP (User Datagram Protocol) transportnej vrstvy sieťového modelu TCP/IP. Program umožní komunikáciu dvoch účastníkov v lokálnej sieti Ethernet, teda prenos textových správ a ľubovoľného súboru medzi počítačmi (uzlami).

Program bude pozostávať z dvoch častí – vysielacej a prijímacej. Vysielací uzol pošle súbor inému uzlu v sieti. Predpokladá sa, že v sieti dochádza k stratám dát. Ak je posielaný súbor väčší, ako používateľom definovaná max. veľkosť fragmentu, vysielajúca strana rozloží súbor na menšie časti - fragmenty, ktoré pošle samostatne. Maximálnu veľkosť fragmentu musí mať používateľ možnosť nastaviť takú, aby neboli znova fragmentované na linkovej vrstve.

Ak je súbor poslaný ako postupnosť fragmentov, cieľový uzol vypíše správu o prijatí fragmentu s jeho poradím a či bol prenesený bez chýb. Po prijatí celého súboru na cieľovom uzle tento zobrazí správu o jeho prijatí a absolútnu cestu, kam bol prijatý súbor uložený.

Program musí obsahovať kontrolu chýb pri komunikácii a znovuvyžiadanie chybných fragmentov, vrátane pozitívneho aj negatívneho potvrdenia. Po prenesení prvého súboru pri nečinnosti komunikátor automaticky odošle paket pre udržanie spojenia každých 10-60s pokiaľ používateľ neukončí spojenie. Odporúčame riešiť cez vlastne definované signalizačné správy.

Program musí mať nasledovné vlastnosti (minimálne):

1. Program musí byť implementovaný v jazykoch C/C++ alebo Python s využitím knižníc na prácu s UDP socket, skompilovateľný a spustiteľný v učebniach. Odporúčame použiť python modul socket, C/C++ knižnice sys/socket.h pre linux/BSD a winsock2.h pre Windows. Iné knižnice a funkcie na prácu so socketmi musia byť schválené cvičiacim. V programe môžu byť použité aj knižnice na prácu s IP adresami a portami:
   arpa/inet.h
   netinet/in.h

2. Program musí pracovať s dátami optimálne (napr. neukladať IP adresy do 4x int).
3. Pri posielaní súboru musí používateľovi umožniť určiť cieľovú IP a port.
4. Používateľ musí mať možnosť zvoliť si max. veľkosť fragmentu.
5. Obe komunikujúce strany musia byť schopné zobrazovať:
   a. názov a absolútnu cestu k súboru na danom uzle,
   b. veľkosť a počet fragmentov.
6. Možnosť simulovať chybu prenosu odoslaním minimálne 1 chybného fragmentu pri prenose súboru (do fragmentu je cielene vnesená chyba, to znamená, že prijímajúca strana deteguje chybu pri prenose).
7. Prijímajúca strana musí byť schopná oznámiť odosielateľovi správne aj nesprávne doručenie fragmentov.
8. Možnosť odoslať 2MB súbor a v tom prípade ich uložiť na prijímacej strane ako rovnaký súbor, pričom používateľ zadáva iba cestu k adresáru kde má byť uložený.

Štefan Dudaško, AIS ID: 96835
Cvičenie pondelok: 17:00

Analýza

UDP je protokol pracujúci na transportnej vrstve. Je často porovnávaný s protokolom TCP. Kľúčovým rozdielom medzi TCP a UDP je ten, že UDP je nespoľahlivý, a teda, nezasiela potvrdenie o prijatí správ ani dáta nekontroluje. Využíva sa najmä v komunikáciách, kde nepotrebujeme byť uistený o tom, že bol paket prijatý a vieme fungovať, aj keď s občasnými výpadkami niektorých paketov, napríklad hovor cez Skype. Na druhej strane však disponujeme rýchlejším prenosom dát a optimálnejším rozdelením dát do rámcov, keďže potrebujeme menej bytov na zostavenie hlavičky, keďže nepotrebujeme toľko informácii v hlavičke.

My budeme využívať tento protokol ako protokol na transportnej vrstve, ktorý obalí náš navrhnutý a implementovaný protokol. Taktiež implementujeme aj funkcionalitu znovuvyžadovania chybných fragmentov, aby sme zabezpečili očakávanú komunikáciu. Počas celej komunikácie budeme vysielať Keep alive správy, ktoré, keď nedostanú odpoveď, zatvoria spojenie, aby sa nestalo, že budeme môcť mať pripojené aj nekomunikujúce zariadenia.

Pre ilustráciu si ešte uvedieme, ako vyzerá UDP hlavička:

Source port	Destination port
Length	Checksum

UDP aj s prenášanými dátami (datagramami), vyzerá nasledovne:

UDP header	data

Keďže chceme predísť ďalšiemu fragmentovaniu na linkovej vrstve, potrebujeme nastaviť maximálnu veľkosť prenášaného rámca na „najväčšiu najmenšiu“ možnú veľkosť, aby sme dosiahli optimálne riešenie.

IPv4 protokol nás limituje na veľkosť 16 bytov, a teda 65 536 – 1 B => 65 535 B.
Od tejto veľkosti musíme ešte odrátať veľkosť UDP hlavičky (8 B) a veľkosť IP hlavičky (20 B), dostaneme sa teda na veľkosť 65 507 B dát, ktoré môžeme posielať. No keďže Ethernetové rámce dokážu prenášať informáciu s najväčšou možnou veľkosťou 1500 B, (MTU) ak by sme odoslali väčší rámec, bol by buď fragmentovaný alebo by nedorazil, sme limitovaný aj tak na veľkosť fragmentu maximálne 1500B, od ktorých musíme odrátať veľkosť našej hlavičky, ktorá je 14B (ďalej je popísaný výpočet), fragmenty, ktoré budeme prenášať budú mať veľkosť minimálne 14B a maximálne 1456B. (1500 – (20 + 8 + 14)) {IP header, UDP header, náš header}

Návrh riešenia

Na to, aby sme implementovali náš protokol, ktorý bude spĺňať požiadavky podľa zadania, vyskladáme si informácie, ktoré potrebujeme pri komunikácii posielať a na záver uvedieme finálny protokol, ktorý v ďalšej časti implementujeme. Ešte je dobré spomenúť, že riešenie budeme implementovať v jazyku Python a spoliehame sa na to, že tento jazyk je na

vyššej úrovni, taktiež programujeme ~~objektovo~~ funkcionálne, a teda neriešime zoraďovanie atribútov v hlavičke.

Polia v hlavičke:

1. **Poradie fragmentu** – budeme indexovať od 1. Toto poradie sa však berie do úvahy až pri rámcoch, kde sa zasiela požadovaná komunikácia, a teda nie pri nadväzovaní spojenia.
2. **Signalizačná správa** – Počas celej komunikácie je potrebné si zasielať nejaké „metadata“, ktoré nám pomôžu zistiť, čo sa práve v komunikácii deje.
   Rozlišujeme tieto signalizačné správy:
   - **1** – Inicializácia spojenia
   - **2** – Potvrdenie o začatí spojenia
   - **3** – Požiadavka o ukončenie spojenia
   - **4** - Potvrdenie ukončenia spojenia
   - **5** - Keep alive
   - **6** – Potvrdenie o prijatí fragmentu správne
   - **7** - Potvrdenie o prijatí fragmentu nesprávne
   - **8** ~~– Znovuvyžiadanie dát (ak v poslednom bloku neboli správne doručené dáta, ak neboli správne doručené v predošlých blokoch, len sa dopošlú pri ostatných)~~
     Riešime cez správu 7
   - **9** – Názov súboru aj s príponou
   - **10** – stdin – posiela sa len správa
   - **11** – posielanie dát
   - **12** – keep alive ACK
3. **Veľkosť fragmentu**
4. **Počet fragmentov**
5. **CRC**
6. **Dáta**

Návrh hlavičky v B (vizuálne, ideme po poradí, v akom sme zadefinovali polia hlavičky):

Naša hlavička					
2B	2B	4B	2B	4B	44-1456B
Poradie f.	Sign. správa	Veľkosť f.	Počet f.	CRC	Dáta

Štefan Dudaško, AIS ID: 96835
Cvičenie pondelok: 17:00

Príklad komunikácie

Na začiatku behu programu sa v prompte opýtame, ako sa chceme prihlásiť, či ako server alebo klient. Ak si vyberieme server, nastavíme Port. Po zapnutí si inštancii aplikácie sa v používateľskom rozhraní konzoli opýtame, na základnú konfiguráciu: IP servera, port servera, čo chceme, ako klient, posielať, veľkosť fragmentu. Ak sa jedná o stdin, napíšeme správu, ak chceme vysielať súbor, vyberieme si možnosť posielania súboru a napíšeme cestu k súboru.

Potom vysielame správu o žiadosti o inicializáciu spojenia, ak dostaneme ACK od servera, môžeme posielať dáta.

Ak posielame súbor a jeho veľkosť je väčšia ako 1457B, rozdelíme ho na fragmenty. Ak ho rozdelíme napríklad na 35 fragmentov, najprv pošlem 15, server spraví na všetky CRC kontrolu, ak je nejaký fragment poškodený, tak server pošle na klienta signalizačnú správu s tým, že si vypýta poškodené fragmenty.

Klient ich pošle na server spolu s ďalšími fragmentami, aby ich bolo dokopy 15. Server ich znova skontroluje a takto cyklí, ak boli v prvej várke poškodené napríklad fragmenty s poradovými číslami {4, 6, 8}, v ďalšom bloku 15 fragmentov odosielame fragmenty: {4, 6, 8, 16, 17, 18 … 27}, povedzme, že v ňom sú všetky dáta správne, pošleme fragmenty {28, 29, 30, … 35}, server pošle odpoveď, že fragment 30 je nesprávny, posielajúca strana pošle fragment s poradovým číslom 30 znova. Server to potvrdí a pošle ACK. Počas celej komunikácie sa stále posielajú keep alive správy, kde každá Keep alive správa prichádza s ACK od druhej strany. Ak chce strana ukončiť komunikáciu, posiela sa správa o ukončení komunikácie aj s ACK od druhej strany.

Chybné dáta simulujeme tak, že zoberieme dáta, vypočítame nad nimi CRC, až potom tie dáta pozmeníme a tak pošleme, potom si prijímajúca strana tiež vypočíta CRC a ten sa nebude zhodovať.

Na konci komunikácie sa, podľa na začiatku získanej správy o type súboru vyskladá súbor podľa poradových čísiel fragmentov, taktiež sa nastaví jeho meno a prípona podľa prvej správy o type súboru (sign. 9). Ak sa jedná o stdin (sign. 10), tak vyskladáme správu a vypíšeme stdout. Ak sme poslali súbor a komunikáciu sme neukončili, stále udržiavame komunikáciu cez Keep alive správy. Ak nedostaneme odpoveď, ukončíme spojenie.

Štefan Dudaško, AIS ID: 96835
Cvičenie pondelok: 17:00

Zmeny po začiatočnom návrhu a riešenie

V tejto časti popíšeme, čo sme pridali alebo upravili od počiatočného návrhu a v krátkosti popíšeme funkcie, ktoré sa počas behu programu vykonávajú, nebudeme ich popisovať do hĺbky, keďže sme sa kód snažili písať samopopisujúco s komentármi pri zložitejších alebo väčších celkoch.

KEEP ALIVE – správanie tejto správy sme zachovali, ako sme ho navrhli, ešte doplním, že túto správu sme implementovali ako thread, a teda komunikujúce strany môžu navzájom komunikovať a zároveň si preposielať keep alive správy. Server vysiela správu a očakáva od klienta odpoveď, ak ju nedostane v stanovenom čase, komunikácia je prerušená. Tento čas sme nastavili na 60 sekúnd, server vysiela keep alive správu každých 10 sekúnd. 60 sekúnd si môžeme zmeniť aj v configu ako ako aj niektoré ďalšie hodnoty.

CRC Kontrola – vypočítame u klienta, server si to prepočíta a porovná, ak nesedia, posiela sa správa, že server potrebuje správu preposlať. CRC vypočítavame polynomiálnou metódou na 32 bitoch. Používame modul crcmod, ktorá operáciu robí za nás. Táto funkcia je veľmi dobrá pre obrovské správy, je veľmi dobrá pri burst chybách, taktiež pri výmene bitu, atď. Chybu simulujeme tak, že prepočítame CRC pre správu a potom do nej vložíme string „ERROR“ a tak ju pošleme. To, do ktorého fragmentu chceme vložiť chybu špecifikujeme v súbore config.

Výmena komunikujúcich strán – program je spúšťaný cez funkciu main, ktorý je daemonom pre celý program. Po preposlaní súboru dostávame možnosť strany vymeniť. Pri výmene strán očakávame, že oba strany odpovedia na otázku o výmene strán s kladnou odpoveďou, keďže neriešime situáciu 2 serverov alebo 2 klientov.

Hlavné funkcie a súbory:

Client.py

Hlavný súbor pre prácu s vysielajúcou stranou, obsahuje základné funkcie na posielanie správ, kde vytvárame header – pribalíme všetky potrebné údajové byty, ku ktorým prípadne pribalíme správu, zaserializujeme ju do bytov a tak posielame.

Client_behaviour() – volá sa z mainu a volá funkcionalitu klienta, je to riadiaca funkcia, v cykle promptuje príkazy, ktoré chceme vykonávať, spúšťa po prvej priatej správe keep alive správy a volá funkciu handle_client_request_to_send_data(), ktorá sa stará o komunikáciu medzi klientom a serverom.

handle_client_request_to_send_data() – spracuváva celú komunikáciu, na začiatku vykoná výmenu inicializačných správ so serverom, neskôr príjma buď stdin alebo súbor, správu rozdelí do fragmentov a posiela po blokoch, ak odošle **k** fragmenvo, čaká na ACK od servera, ak dostane kladné ACK, posiela ďalej, ak záporné, posiela ďalej + do daného bloku pribalí chabnú správu až pokiaľ nedočíta celú správu.

Štefan Dudaško, AIS ID: 96835
Cvičenie pondelok: 17:00

Server.py

Hlavný súbor pre prácu s príjmajúcou stranou, taktiež má základné funkcie na posielanie packetu obdobné klientovi s obdobnou funkcionalitou, taktiež funkciu pre posielanie Keep alivu, ten kotrnolujeme, či dostávame aj od klienta (ACK), ak nie, ukončujeme komunikáciu.

Server_behaviour() – tiež ju voláme z mainu, riadi server a volá funkciu handle_server_responses()

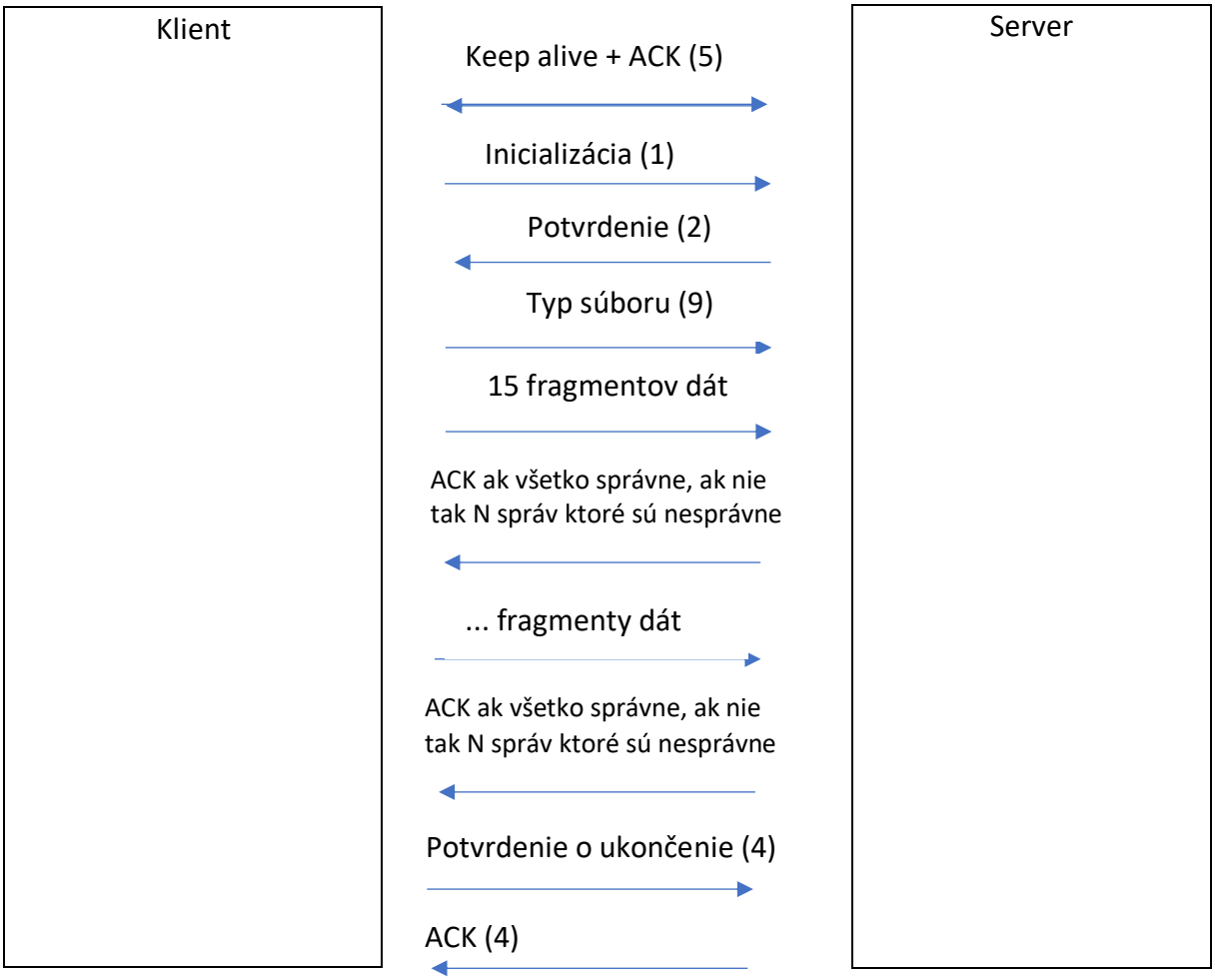
handle_server_responses() – spracuváva celú komunikáciu, na začiatku pri nadväzovaní spojenia posiela ACK na klienta po prijatí správy o požiadanie o komunikácie, po nadviazaní komunikácie očakáva, že dostane názov súboru alebo stdin, ďalej už len príjma správy a po každých **k** fragmentoch posiela správu o tom, v akom stave dané fragmenty dostal. Po každom prijatom fragmente si ukladáme do poľa fragmentov s CRC mismatchom, ak mal chybný mismatch, poradové číslo toho fragmentu, ktorý potom vyžiadame. Ak server prijal celý blok správne, posiela len ACK, ak bol nejaký fragment nesprávny, posielame správu, že bol ten fragment nesprávne prijatý.

Zhodnotenie

Zadanie bolo zaujímavé, plné chýb, ktoré sa dosť zložito odhaľujú, ale na druhej strane pomáhajú lepšie pochopiť, ako komunikácia prebieha a čo sa v komunikácii deje, kde sa môže komunikácia zaseknúť, a kde naopak posielame viacej dát, ako server očakáva. Zadanie sme riešili funkcionálne, čo som považoval za dobrý nápad na začiatku, keďže posielame a spracovávame dáta selektívne. Ale lepšie a prehľadnejšie by to bolo pravdepodobne objektovo. Po tomto zadaní mám oveľa lepšiu predstavu, ako prebieha komunikácia a výmena správ medzi komunikujúcimi správami.

Štefan Dudaško, AIS ID: 96835
Cvičenie pondelok: 17:00

Vizualizácia



Štefan Dudaško, AIS ID: 96835
Cvičenie pondelok: 17:00

Vizualizícia fungovania programu: