|  |
| --- |
| Fakulta informatiky a informačných technológií |
| Komunikácia s využitím UDP protokolu – PKS 2022/2023 |
| Slovenská technická univerzita |

|  |
| --- |
| Michal Kilian  8.12.2022 |

OBSAH

1. ÚVOD – ZADANIE
   1. VÝBER PROGRAMOVACIEHO JAZYKU A PROSTREDIA
2. ŠTRUKTÚRA KÓDU
3. ŠTRUKTÚRA HLAVIČKY VLASTNÉHO PROTOKOLU
4. OPIS POUŽITEJ METÓDY KONTROLNEJ SUMY (CHECKSUM)
5. FUNGOVANIE ARQ
6. METÓDY PRE UDRŽANIE SPOJENIA (KEEP ALIVE)
7. FUNKČNOSŤ PROGRAMU
8. ZÁVER
9. ÚVOD - 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 pozostáva z 2 častí, vysielacej a prijímacej (v mojom prípade sú to časti *Client* a *Server*).

* 1. VÝBER PROGRAMOVACIEHO JAZYKU A PROSTREDIA

Na toto zadanie som si zvolil programovací jazyk *Python 3.10.9.* a vypracoval som ho v prostredí *PyCharm 2022.2.2*.

1. ŠTRUKTÚRA KÓDU

Moje vypracovanie tohto zadania sa skladá z 8 modulov, sú nimi:

**main.py** – Hlavný modul, ktorý spúšťa program. Nachádza sa v ňom hlavný *while* cyklus, z ktorého je možné ďalej pokračovať ako *Server* alebo *Client*.

**server.py** – Modul, v ktorom sa nachádza trieda *Server* a spolu s ním aj všetky potrebné funkcie na jeho inicializáciu.

**client.py** – Modul, v ktorom sa nachádza trieda *Client* a spolu s ním aj všetky potrebné funkcie na jeho inicializáciu.

**comm\_funcs.py** – Modul, v ktorom sa nachádzajú funkcie potrebné pri posielaní alebo prijímaní packetu.

**packet\_creator.py** – Modul, v ktorom sa nachádza trieda *PacketCreator*, ktorá je zodpovedná za korektné zostavenie packetu pri jeho odosielaní.

**packet\_translator.py** – Modul, v ktorom sa nachádza trieda *PacketTranslator*, ktorá je zodpovedná za zistenie informácií o packete pri jeho prijímaní.

**packet\_data.py** – Modul, v ktorom sa nachádza dátová trieda *PacketData*, ktorá je potrebná pri ukladaní informácií o packete.

**indexes.py** – Modul, ktorý obsahuje dôležité hodnoty využité pri prijímaní a prekladaní packetu.

1. ŠTUKTÚRA HLAVIČKY VLASTNÉHO PROTOKOLU

Štruktúra sa oproti návrhu riešenia takmer nezmenila, jediná zmena spočíva v pridaní ďalšieho dátového typu, ktorým je *FILE-HEADER*. Je to v podstate označenie packetu, ktorý obsahuje informácie o súbore, ktorý bude poslaný a prijatý. Konkrétne obsahuje názov súboru a veľkosť súboru, oddelené separátorom definovaným v module *indexes.py*. Samotná štruktúra však má takúto podobu:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type of bytes** | **Values they can acquire and their meaning** | | | | **Byte length** |
| Commtype | 0 –> NO-COMMTYPE | 1 -> SYN | 2 -> ACK | 3 -> FIN | 1 |
| Datatype | 0 -> NO-DATATYPE | 1 -> TEXT | 2 -> FILE | 3 -> FILE-HEADER | 1 |
| Fragnum | 32 bit Unsigned Integer (0 - 2147483647) | | | | 4 |
| Fragtotal | 32 bit Unsigned Integer (0 - 2147483647) | | | | 4 |
| Checksum | 32 bit Unsigned Integer (0 - 2147483647) | | | | 4 |
| Data | Any bytes | | | | Any |

Ako vidíme v tabuľke, hlavička má teda spolu 14 bajtov:

* **commtype** – hodnota 1. bajtu, keďže jeho hodnota môže byť len 0-3, ukladá sa sem číslo v jeho bajtovej forme
* **datatype** – hodnota 2. bajtu, keďže jeho hodnota taktiež môže byť len 0-3, ukladá sa sem číslo v jeho bajtovej forme
* **fragnum** – hodnota 3.-6. bajtu, ukladá sa sem číslo fragmentu v poradí po prevedení do jeho binárnej podoby
* **fragtotal** – hodnota 7.-10. bajtu, ukladá sa sem celkový počet fragmentov po prevedení do jeho binárnej podoby
* **checksum** – hodnota 11.-14. bajtu, ukladá sa sem kontrolná suma packetu, ktorá sa počíta z celej hlavičky a dát, okrem miesta pre túto hodnotu
* **data** – hodnota 15.-x. bajtu, môže obsahovať bajty textu alebo súboru, ich veľkosť je obmedzená používateľom, ktorý si volí veľkosť fragmentu

1. OPIS POUŽITEJ METÓDY KONTROLNEJ SUMY (CHECKSUM)

Vypočítanie kontrolnej sumy je vitálne pre overenie, že prijímač prijal packet presne v takej istej podobe, v akej opustil vysielača. Ako je jasné zo štruktúry protokolu, kontrolná suma je uložená na 11.-14. bajte. Pri prijímaní packetu sa teda prevedie táto hodnota z jej 32 bitovej binárnej podoby do desiatkovej číselnej sústavy. Následne je potrebné prerátať túto hodnotu znova, čo sa deje vo funkcii *get\_checksum()*:

Obrázok, na ktorom je text

Automaticky generovaný popis

Kontrolná suma sa teda znova ráta pomocou funkcie *crc32* z knižnice *zlib*, z hlavičky bez kontrolnej sumy a dát. Následne, ak máme obidve tieto hodnoty, porovnáme ich:

Obrázok, na ktorom je text

Automaticky generovaný popis

Ak sme zistili, že sa tieto hodnoty nerovnajú, musel teda nastať prípad, že sa aspoň jeden bajt packetu pri prechádzaní sieťou zmenil. Prijímač tohto packetu teda neposiela ACK, ale naopak, posiela FIN, a tým dáva naspäť najavo vysielaču, že správa neprišla v poriadku. Týmto si prijímač vyžiada opätovné poslanie packetu a tentokrát bude dúfať v správne prejdenie packetu po sieti.

Ak sa však tieto hodnoty rovnajú, čo je žiadaný scenár, nastáva prijímačom poslanie ACK-u, a tým teda prijímač informuje vysielača, že packet prišiel v poriadku a vysielač môže pokračovať v posielaní ďalších fragmentov alebo správ.

1. FUNGOVANIE ARQ

Ako metódu *ARQ (Automatic repeat request)* som si vybral metódu *Stop&Wait*. V mojom programe funguje v podstate tak, že klient nepošle nasledujúci fragment správy alebo súboru dovtedy, dokým mu nepríde *ACK* od prijímača. Týmto si vieme ozrejmiť, že aktuálne poslaný packet prešiel po sieti v poriadku a môžeme pokračovať v posielaní. Ak *ACK* nepríde do piatich sekúnd, fragment sa pošle znova. Ak sa to zopakuje trikrát, nastane *Timeout Error*. Túto metódu si môžeme pozrieť na príklade z kódu, napríklad v klientovej funkcii *send\_file()*:

  
Obrázok, na ktorom je text

Automaticky generovaný popis

Najskôr treba však nastaviť *Python socketu* takzvaný *timeout*:



Tento *timeout* teda použijeme pri zachytení *Timeout Erroru*. Takto vieme povedať, či prišiel žiadaný *ACK* do piatich sekúnd, alebo nie.

1. METÓDY PRE UDRŽANIE SPOJENIA (KEEP ALIVE)

Udržanie spojenia je taktiež dôležitou súčasťou fungovania prepojenia medzi vysielačom a prijímačom. Táto metóda sa nezmenila oproti návrhu riešenia, vykonáva sa vo funkcii klienta *client\_keep\_alive()*:

Obrázok, na ktorom je text

Automaticky generovaný popis  
Obrázok, na ktorom je text

Automaticky generovaný popis

Táto funkcia sa zavolá v momente, ako sa nadviaže spojenie so serverom a vykonáva sa na samostatnom thread-e:

Obrázok, na ktorom je text

Automaticky generovaný popis

Jej trvanie je zastavené, ak klientov *SYN* nie je zodpovedaný serverovým *ACK*-om po 5 sekúnd trikrát v rade.

1. FUNKČNOSŤ PROGRAMU

Teraz si prejdeme celú funkčnosť programu, spolu aj s ukážkou packetov posielaných po sieti vo Wiresharku.

Program sa začína spustením hlavného cyklu s názvom *main\_loop()*, v ktorom máme 3 možnosti:



Po zadaní príkazu:

**server** – sa nás program spýta na našu IP adresu a port, na ktorom chceme počúvať. V zátvorkách sú predvolené možnosti, ktoré budú účinné, ak nič nezadáme. Po úspešnom zadaní IP adresy a portu sa zobrazí správa, že na našej IP adrese a porte sme začali úspešne počúvať.

Obrázok, na ktorom je text

Automaticky generovaný popis

**client** – nás program vyzve na zadanie IP adresy a portu serveru, s ktorým chceme nadviazať spojenie a poslať mu správu alebo text. Po úspešnom zadaní týchto údajov sa pokúšame pripojiťna tento konkrétny server, a to poslaním *SYN*-u:



Ak sa nám trikrát nedostane *ACK* zo servera, zobrazí sa táto správa a vrátime sa do hlavného menu:

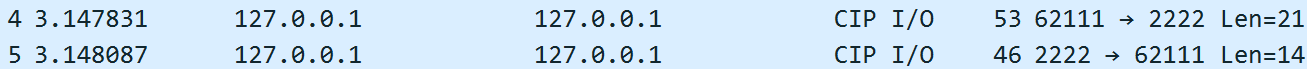
Obrázok, na ktorom je text

Automaticky generovaný popis

Ak sa nám však podarí nadviazať spojenie, t.j. dostaneme *ACK* zo servera:



Začne sa udržiavať spojenie rovnakými *SYN*-mi a *ACK*-mi:



**exit** – po zadaní tohto príkazu sa vypne programu

Následne, ak sme sa teda rozhodli použiť príkaz *client* a spojenie so serverom bolo nadviazané, dostaneme sa do cyklu s názvom *command\_loop()*:

Obrázok, na ktorom je text

Automaticky generovaný popis  
Obrázok, na ktorom je text

Automaticky generovaný popis

Tu máme teda k dispozícii 2 príkazy, ktorými sú:

**send** – po vybratí tohto príkazu nás program vyzve nasledovnými požiadavkami, takto bude vyzerať menu odosielania textu:

Obrázok, na ktorom je text

Automaticky generovaný popis

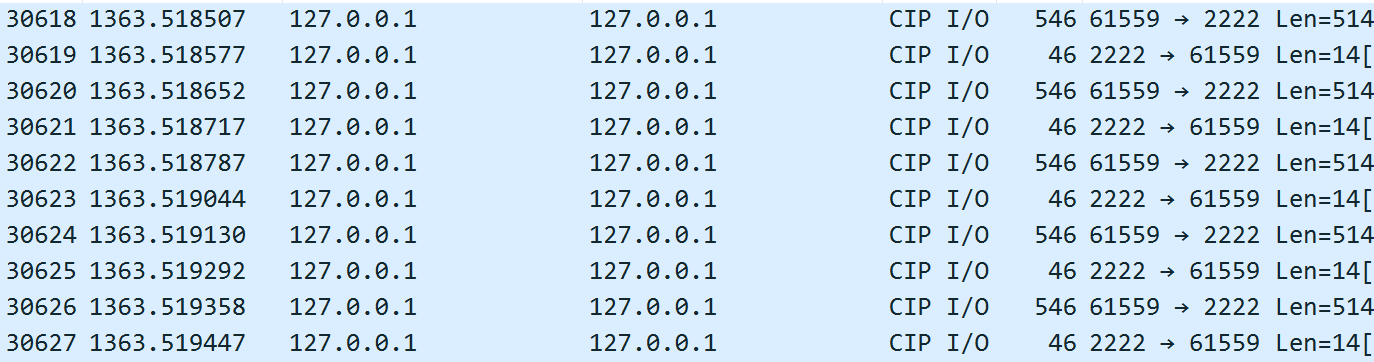
A takto bude vyzerať menu odosielania súboru:

Obrázok, na ktorom je text

Automaticky generovaný popis

Program sa teda spýta na zadanie konkrétneho textu alebo absolútnej cesty k súboru, ktorý má byť serveru poslaný a maximálnu veľkosť fragmentu, ktorý má byť po sieti poslaný, pričom tento fragment môže mať minimálnu veľkosť 5 bajtov a maximálnu veľkosť 8000B.

Po úspešnom zadaní týchto údajov sa správa rozloží na jednotlivé fragmenty a začne sa posielať, pričom sa využíva *ARQ* metóda *Stop&Wait*, takže klient čaká na *ACK* zo servera a až po jeho prijatí odošle ďalší fragment. Vo Wiresharku posielanie fragmentov súboru, každého s veľkosťou 500 bajtov vyzerá takto:



Odosielanie textu a súboru sa vykonáva vo funkciách *send\_text()* a *send\_file()*. Najskôr si prejdeme funkciu *send\_text():*

Obrázok, na ktorom je text

Automaticky generovaný popis

Obrázok, na ktorom je text

Automaticky generovaný popis

Najskôr sa teda vypočíta veľkosť bajtov textu a tá sa rozloží po veľkosti fragmentov. Tým sa vypočíta číslo fragmentu a celkový počet fragmentov a tieto hodnoty sa následne spolu s ďalšími informáciami o packete odošlú prijímačovi.

Ako vidíme, pri prijatí *FIN*-u sa rovnako pri *Timeout Errore* inkrementuje hodnota premennej *timeout* a ak je jej hodnota 3, tak sa vypíše správa, že server trikrát neodpovedal a preto posielanie skončilo.

Funkcia *send\_file()* je veľmi podobná ako vyššie predvedená funkcia *send\_text()*, líši sa však v získavaní bajtov súboru:

Obrázok, na ktorom je text

Automaticky generovaný popis  
Obrázok, na ktorom je text

Automaticky generovaný popis  
Obrázok, na ktorom je text

Automaticky generovaný popis

Po zavolaní sa tejto funkcie sa teda najskôr zistí, či takýto súbor existuje. Potom sa zistí jeho veľkosť funkciou *os.path.getsize()* z knižnice *os*. Tento súbor sa potom otvorí, jeho bajty o veľkosti jednotlivého fragmentu sa začnú čítať a v takejto forme sa odosiela. Rovnako, ak nastane *Timeout Error* trikrát alebo prijímač pošle naspäť *FIN* trikrát, odosielanie sa ukončí.

Tak, vysvetlili sme si odosielanie textu a súboru z klientovej strany a teraz prejdeme na ich prijímanie zo strany servera.

Server má teda zapnuté počúvanie na jeho IP adrese a určitom porte. Toto počúvanie a reagovanie na prichádzajúce packety sa odohráva vo funkcii servera s názvom *listening\_loop()*:

Obrázok, na ktorom je text

Automaticky generovaný popisObrázok, na ktorom je text

Automaticky generovaný popisObrázok, na ktorom je text

Automaticky generovaný popis

Ako si môžeme všimnúť, server reaguje na rozličné packety rozličnými spôsobmi:

* Ak je *status\_code* 200, znamená to, že kontrolná suma vypočítaná na strane servera sa zhoduje s tou vypočítanou na strane klienta a server posiela naspäť *ACK*.
* Ak serveru príde *FIN*, znamená to, že klient zadal príkaz *change*, ktorý aj pre klienta aj pre server znamená, že sa ukončí ich činnosť a vrátia sa do hlavného menu, v ktorom si môžu znova vybrať ich pokračovanie v programe. Týmto je zaručená možnosť zmeny z vysielaču na prijímač a naspäť.
* Ak serveru príde komunikačný typ *SYN* a dátový typ *NO-DATATYPE*, znamená to, že sa klient snaží nadviazať spojenie a preto mu odpovedáme *ACK*-om, pretože toto spojenie prijímame. Ak ešte nemáme tohto klienta uloženého v *clients\_connected*, tak vypíšeme správu, že sme boli spojení a uložíme si ho.
* Ak príde dátový typ *TEXT*, prechádzame do funkcie *handle\_text\_fragment()*:

Obrázok, na ktorom je text

Automaticky generovaný popis

Táto funkcia vypíše správu, že fragment bol úspešne prijatý a okrem toho zisťuje, či už fragmenty končia, alebo ešte nie, podľa toho, či sa číslo fragmentu rovná celkovému počtu fragmentov. Vracia dekódované dáta textu a booleovskú hodnotu *more\_fragments*.

* Ak príde dátový typ *FILE\_HEADER*, prechádzame do funkcie handle\_file\_header():

Obrázok, na ktorom je text

Automaticky generovaný popis

Táto funkcia oddelí cestu k súboru a jeho veľkosť pomocou separátora a vracia názov súboru, cestu k nemu a jeho veľkosť.

* Ak príde dátový typ *FILE*, pokračujeme vo funkcii *handle\_file\_fragment()*:

Obrázok, na ktorom je text

Automaticky generovaný popis

Táto funkcia zoberie dáta konkrétneho packetu, vypíše, že fragment bol prijatý bez chýb a vracia dáta packetu a hodnotu *more\_fragments*.

* Ak je *status\_code* 400, znamená to, že kontrolná suma vypočítaná na strane servera sa nezhoduje s tou na strane klienta, preto server posiela *FIN*.

Nakoniec sa pozrieme na hodnotu *more\_fragments* a podľa nej pokračujeme ďalej. Ak sme prijali už všetky fragmenty textu, vypíšeme informácie o správe vo funkcii *print\_text\_info()*. Ak sme prijali všetky fragmenty súboru, prechádzame do funkcie *reconstruct\_file()*:

Obrázok, na ktorom je text

Automaticky generovaný popis

V nej len vytvoríme súbor, alebo otvoríme, ak už bol vytvorený v adresári Downloads a zapisujeme do neho prijaté dáta, ktoré máme uložené v *Python dictionary* s názvom *file\_fragments*. Po zrekonštruovaní súboru ho uložíme do priečinku Downloads a vypíšeme o ňom informácie.

Týmto sme ukončili všetko, čo sa týka posielania a prijímania súboru aj zo strany servera, aj zo strany klienta.

Ešte sa však pozrieme na príkaz **change**:

* tento príkaz zadáva klient, a po jeho zadaní sa aj klient, aj server, ku ktorému je pripojený, vrátia do hlavného menu

Ako posledné si pozrieme príklad pripojenia sa na server a poslanie mu súboru. Server teda začne najskôr počúvať:

Obrázok, na ktorom je text

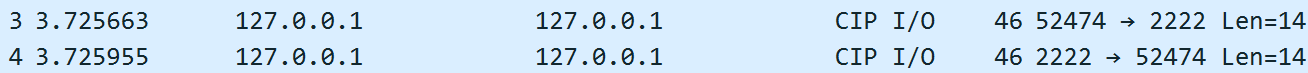
Automaticky generovaný popis

Potom zapneme klienta, ktorý sa na tento server pripojí:

Obrázok, na ktorom je text

Automaticky generovaný popis

Vidíme to aj vo Wiresharku:



My, ako klient, sme teda na porte 52474 a server je na porte 2222. Teraz mu pošleme súbor s veľkosťou fragmentu 500 bajtov. V konzole vidíme na strane klienta niečo takéto:

Obrázok, na ktorom je text

Automaticky generovaný popis

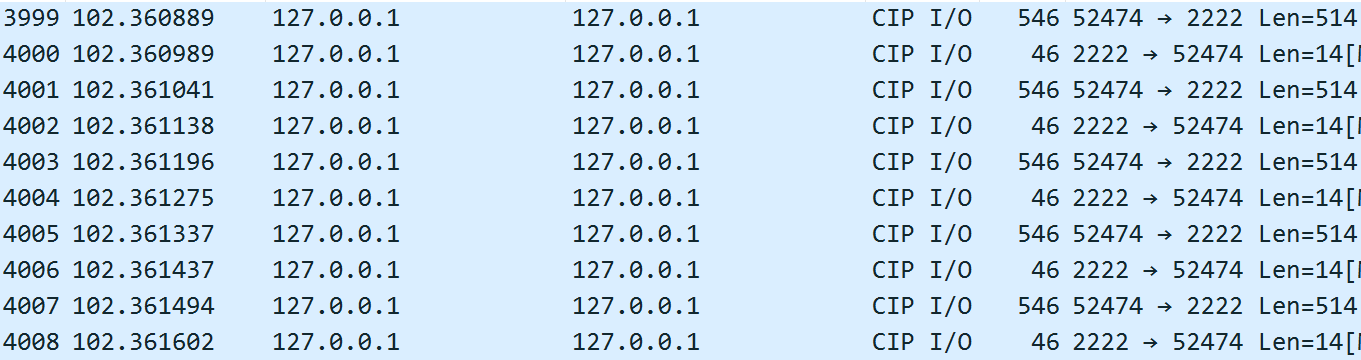
A na strane serveru niečo takéto:

Obrázok, na ktorom je text

Automaticky generovaný popis

Vidíme teda, že sa poslalo celkovo 2459 fragmentov s veľkosťou 500 bajtov. Taktiež na obidvoch stranách vidíme názov súboru, absolútnu cestu k súboru (na strane klienta je to cesta, kde sa súbor nachádzal pri posielaní a na strane serveru je to cesta, kam sa nám uložil, teda obidve cesty k súboru sú platné), tiež vidíme na obidvoch stranách aj veľkosť súboru.

Pozrieť si to môžeme aj vo Wiresharku:



Toto je samozrejme len určitá časť zo všetkých fragmentov odoslaných po sieti prijímačovi. Medzi týmito fragmentami sa však občas nachádzajú aj správy na udržanie spojenia.

Tak a týmto sme ukončili kapitolu Funkčnosť programu a prechádzame do poslednej časti – záveru.

1. ZÁVER

Ak by som mal zhodnotiť toto zadanie, hodnotím ho ako pomerne náročné, celkovo mi dalo zabrať, hlavne pre mňa nový koncept *Python socket* knižnice, thread-ov a pracovanie s nimi. Ale určite som prehĺbil svoje poznatky o sieťach a vytvorenie a používanie vlastného protokolu mi prišlo ako veľmi náučná činnosť.

Michal Kilian

Student ID: 116211