Názov zadania:	Komunikácia s využitím UDP protokolu
Univerzita:	Slovenská Technická Univerzita v Bratislave
Fakulta:	Fakulta informatiky a informačných technológií
Autor:	Filip Mojto
ID:	116253
Dátum:	15.11.2023

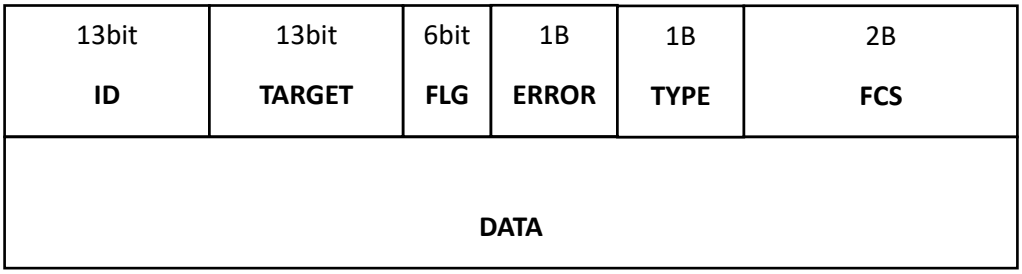
Tento dokument slúži ako dokumentácia a návrh funkcionality zadania 2 z predmetu PKS.

Návrh hlavičky protokolu

Dostupné **flagy** v časti **FLG**:

a) Bit 1 – **S**
b) Bit 2 – **A**
c) Bit 3 – **F**
d) Bit 4 – **C**
e) Bit 5 – **Q**
f) Bit 6 – **MF**

Bit *S* slúži ako iniciatíva začatia komunikácie s druhou stranou. *A* naznačuje, že žiadosť v rámci uvedenom v poli TARGET bola potvrdená. *F* slúži na vyjadrenie vôle ukončiť komunikáciu. *C* predstavuje kontrolnú správu, ktorú musí druhá strana potvrdiť. *Q* sa využije, ak jedna strana nejakým spôsobom po kontrolných správach nereaguje, kedy je nutné „násilím“ ukončiť komunikáciu.
FCS formou CRC slúži na kontrolu konzistentnosti prijatých rámcov.



Obrázok 1: Štruktúra hlavičky protokolu

ID predstavuje unikátne označenie rámca v komunikácii. Maximálny možný počet rámcov v jednej komunikácií je obmedzený na **8192** rámcov. Číslovanie ID začína od 1, pretože 0 slúži na označenie „žiadneho“ rámca, teda keď je v *TARGET* uvedená 0, nemá daný rámec žiadny cieľový rámec. Rámce musia byť označené pri začínaní aj končení komunikácie, pri potvrdzovaní rámcov, pri opätovných pokusoch odoslať správu či pri posielaní kontrolných správ. Takisto aj pri vytváraní postupnosti fragmentov.

ERROR predstavuje nejakú z definovaných chýb, ktoré zvyčajne iba dopĺňajú rámce s *ACK* = 0. Napríklad neplatné číslo fragmentu, prípadne chýbajúce fragmenty, atď... Pole *TYPE* predstavuje typ správy. Hodnota 0 v tomto prípade znamená poslanie bežnej **textovej správy**, 1 pre posielanie **textového súboru** a ďalšie hodnoty pre ďalšie súbory. Celkovo teda protokol podporuje 255 rôznych súborových prípon.

Výpočet CRC

Na zistenie zvyšku (teda CRC), budeme dáta deliť pomocou polynómu **CRC-16-CCIT**:

$$x^{16} + x^{12} + x^5 + 1$$

Toto delenie generuje 2^{16} možných CRC hodnôt, takže nám budú na CRC stačiť **2 bajty**.

Fungovanie protokolu ARQ

Zúčastnená strana v komunikácii bude mať pridelený **buffer**, ktorý bude udržovať prehľad o odoslaných paketoch, ktoré ešte neboli potvrdené druhou stranou. Veľkosť bufferu bude možné meniť dynamicky počas behu programu.

Ak druhá strana neodpovie, bude to buď znamenať, že rámec bol poškodený alebo že prijímateľ zlyhal a nereaguje. Preto pokus zopakuje niekoľko krát a potom pošle kontrolnú správu.

Metódy na udržiavanie spojenia

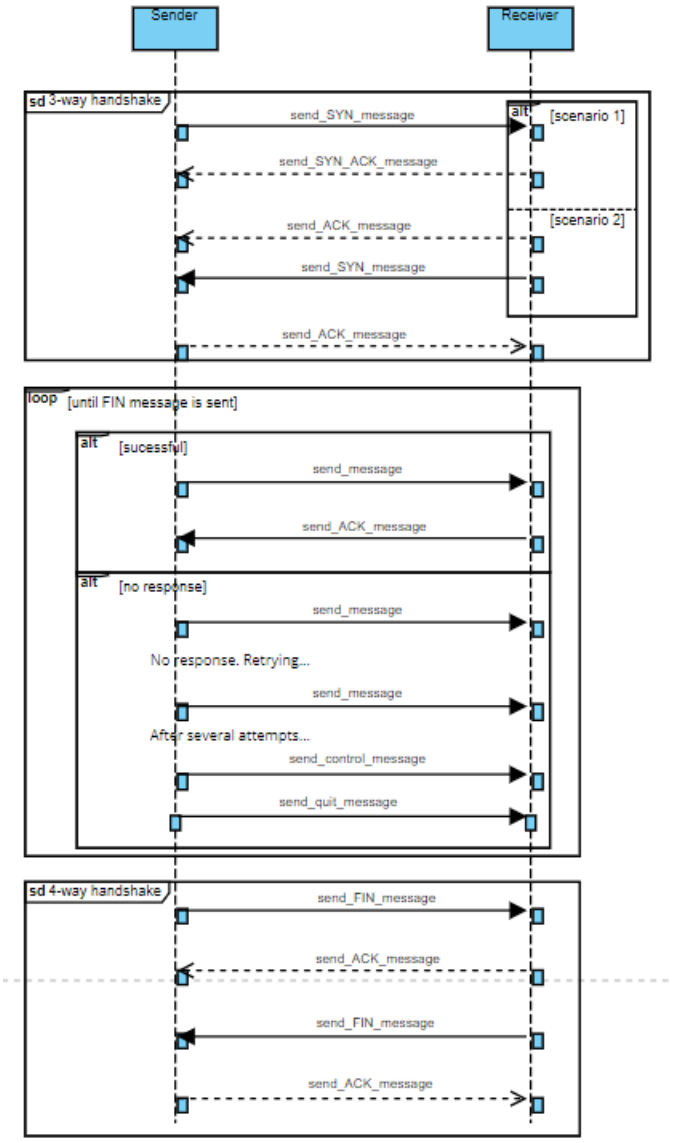
Na nadviazanie spojenia bude nutný **3-way handshake** a na ukončenie spojenia **4-way handshake**, tak ako je tomu pri TCP.

Počas komunikácie si budú zúčastnené strany navzájom posielať **kontrolné správy**, na ktoré keď nedostanú odpoveď v stanovenom čase, tak budú môcť usúdiť, že buď zlyhalo spojenie alebo je druhá strana proste neaktívna a budú môcť požadovať ukončenie tejto komunikácie.

Samotné strany si budú môcť určiť veľkosť kapacity prijatých, no nepotvrdených paketov, ktoré budú schopné prijať. Takto sa zabráni zbytočnému SPAM-ovaniu zúčastnených strán.

Diagram spracovávania komunikácie na oboch uzloch

Obrázok zachytáva 3 hlavné scenáre v komunikácii – **proces začatia**, **proces samotnej komunikácie** a **proces ukončenia**.

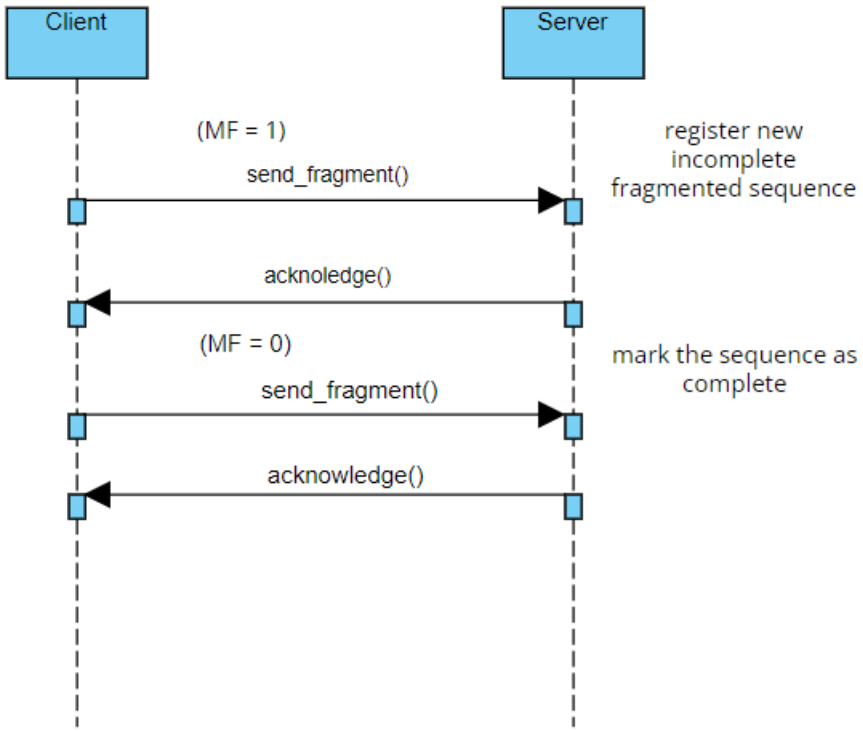


Obrázok 2: Diagram komunikácie v našom protokole

Len pripomíname, že diagram nezachytáva úplne všetky možné scenáre. Napríklad, je na odosielateľovi (entita **Sender**), aby komunikáciu započal pomocou SYN. Avšak proces ukončenia môže byť kľudne začatý aj na strane prijímateľa (entita **Receiver**). Rovnako tak pri procese odoslania bežnej či kontrolnej správy.

Príklad fragmentácie

Obrázok nižšie ukazuje, ako protokol pracuje s fragmentami. Pri odoslaní fragmentovanej správy sa vyhodnotí TARGET a ak je nulový, tak sa vytvorí nová sekvencia fragmentov. Ak nie, je tento fragment pridaný do takej sekvencie, ktorá obsahuje daný ID. Samozrejme, to, či sa ide o posledný fragment a možno sekvenciu vyhlásiť za kompletnú, je nutné signalizovať MF = 0.



Obrázok 3: Príklad fragmentácie

Popis jednotlivých častí kódu

Priložený kód obsahuje dva hlavné python súbory, ktoré reprezentujú obe strany komunikácie:

a) **server.py**
b) **client.py**

Server

Keď chceme spustiť server, stačí spustiť prvý zo súborov. Pomocou funcie *start()* sa automaticky spustí server, naviac sa ale spustí v novom vlákne. Táto funkcia sa každú sekundu pokúsi o načítanie dát zo socketu. Tu je nastavený *timeout*, aby sa tu nečakalo nekonečne dlho. Vždy ale skontroluje **signál 1**, či ho náhodou hlavné vlákno nenastavilo a nie je teda nutné skončiť čítanie. Ak je signál nastavený, skončí sa čítanie a nastaví sa **signál 2** pre možné pokračovanie hlavného vlákna vo vypínaní serveru. Používateľ môže za behu serveru použiť nasledovné príkazy:

a) **-h** pre zobrazenie pomoci,
b) **-s** pre uspatie konzoly na niekoľko sekúnd,
c) **-q** pre vypnutie servera a skončenie programu (metóda *quit()*).

Súperenie o kritickú oblasť nastáva pri použití príkazu na skončenie programu. Hlavné vlákno má totižto v úmysle vypnúť socket, z ktorého ale vedľajšie vlákno pravidelne číta dáta. Preto hlavné vlákno nastaví už spomínaný signál 1, aby signalizoval vôľu ukončiť program. Zvyšok pokračuje tak, ako je uvedené vyššie.

Klient

Keď chceme spustiť klienta, spustíme súbor *client.py*. V tomto prípade sa spustí iba 1 vlákno, ktoré čaká na príkazy zo strany užívateľa. Možné príkazy:

a) **-h** pre zobrazenie pomoci
b) **-i** pre začatie komunikácie so serverom (init_com())
c) **-m** pre odoslanie správy (send_message())
d) **-f** pre odoslanie súboru (send_file())
e) **-t** pre ukončenie komunikácie so serverom
f) **-q** pre vypnutie socketu a ukončenie programu

Začatie i ukončenie komunikácie sme už naznačili v diagrame. Príkaz -q nebude možné realizovať spolu s aktívnymi komunikáciami so serverom. Rámce bude možné aj **fragmentovať**, to sa bude využívať najmä v prípade posielania súboru.

Knižnice

Na prácu budeme používať nasledovné knižnice:

a) **socket**

b) **time**
c) **threading**
d) **enum**
e) **network_utils**
f) **utils**

Posledné dvé sú naše vlastné vytvorené knižnice. Prvá obsahuje rôzne triedy pre prácu so sieťou (*Header*, *IPv4*, *Communication*...) a druhá rôznu funkcionalitu, napríklad pre prácu s bitmi.