Fakulta informatiky a informačných technológií

Slovenská technická univerzite v Bratislave

# Zadanie 1: Analyzátor sieťovej komunikácie

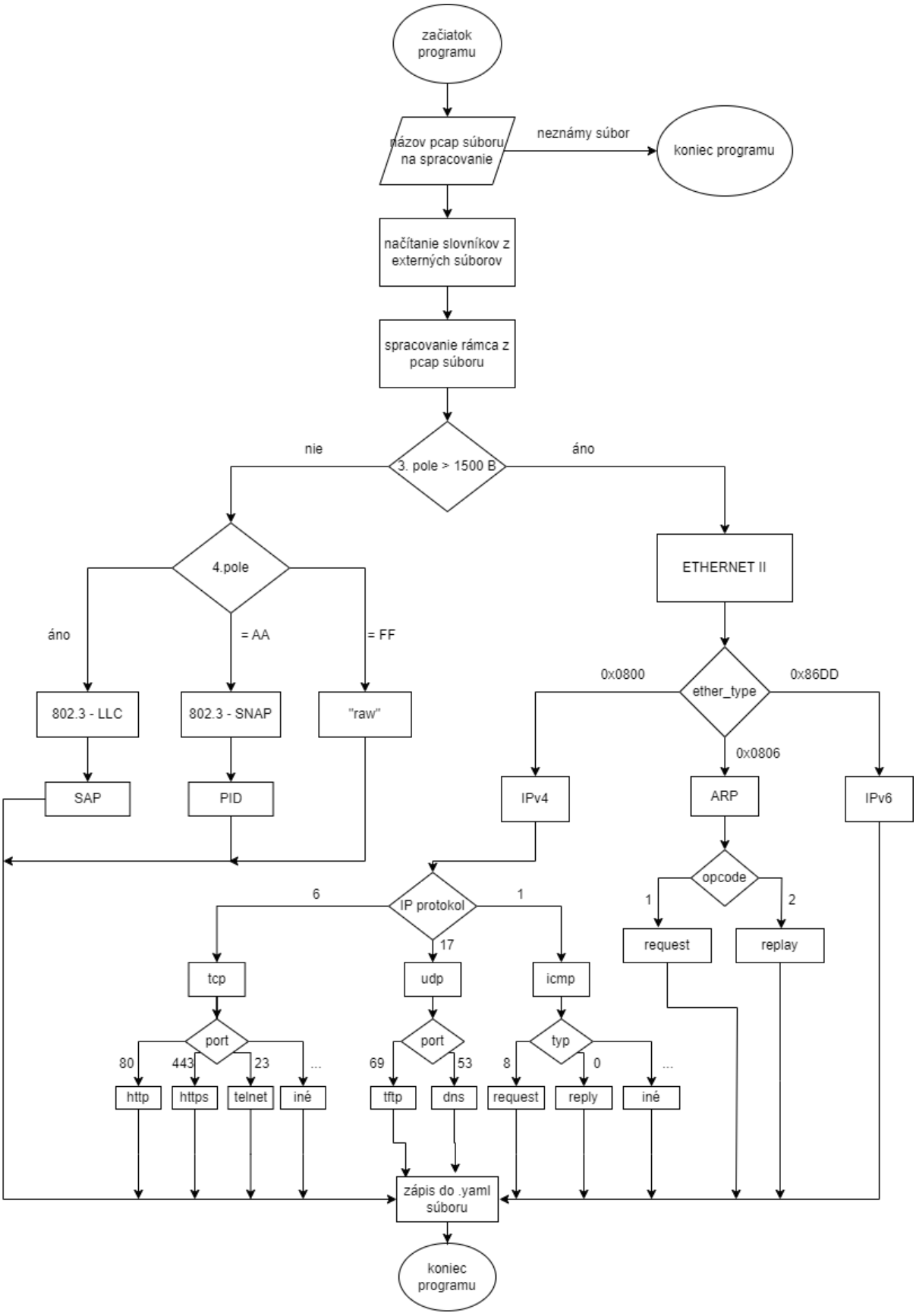
## Počítačové a komunikačné siete

Tomáš Brček

2023/2024

# Obsah

# Diagram spracovávania riešenia



# Navrhnutý mechanizmus analyzovania protokolov na jednotlivých vrstvách

## Linková vrstva

Pri analyzovaní typu rámca sa pozerám na na 3. pole, teda 13. a 14. bajt v poradí. Hodnotu týchto dvoch bajtov si premením z hexadecimálnej do desiatkovej sústavy. V prípade, že táto hodnota bude väčšia ako 1500B, ide o typ rámca **ETHERNET II**. Ak je hodnota týchto bajtov menšia ako spomínaná hodnota 1500B, pozerám sa do 4. poľa, teda na 15. bajt:

- ak v ňom nájdem hexadecimálnu hodnotu AA je to typ **IEEE 802.3 LLC + SNAP**
- ak v ňom nájdem hexadecimálnu hodnotu FF je to typ **IEEE 802.3 LLC**
- inak ide o typ **IEEE 802.3 RAW**

## Sieťová vrstva

**a. IEEE 802.3 LLC + SNAP**

- pri tomto type rámca ďalej určujeme PID
- ten sa nachádza na 21. a 22. bajte, nájdem tieto bajty a spojím ich dokopy
- následne sa pozriem do slovníka, v ktorom mám uložené možné hodnoty PID a ak sa tam táto hodnota nachádza, uložím ju do slovníka pre výpis do .yaml súboru

**b. IEEE 802.3 LLC**

- pri tomto type rámca ďalej určujeme SAP
- ten sa nachádza na 15. bajte
- následne sa pozriem do slovníka, v ktorom mám uložené možné hodnoty SAP a ak sa tam táto hodnota nachádza, uložím ju do slovníka pre výpis do .yaml súboru

**c. ETHERNET II**

- pri tomto type rámca sa opäť pozeráme na 13. a 14. bajt, spojím ich dokopy a pozriem sa do slovníka, ktorý som načítal z externého súboru
- ak sa takáto hodnota v slovníku nachádza, vypíše sa protokol do slovníka pre výpis do .yaml súboru
- najčastejšími protokolmi na sieťovej vrstve sú**: IPv4, IPv6** alebo **ARP**

## Transportná vrstva

Protokol na transportnej vrstve sa určuje pri type rámca **ETHERNET II** a pri hodnote ether_type: **IPv4**. Tento protokol je určený v IP hlavičke a nachádza sa na 24. bajte. Tento bajt si premením z hexadecimálnej sústavy do decimálnej, a pozriem sa do slovníka, v ktorom mám načítané dáta z externého súboru. Ak sa takáto hodnota vyskytuje v kľúčoch tohto slovníka, hodnota na danom kľúči sa zoberie ako protokol na transportnej vrstve.

Najčastejšími protokolmi na transportnej vrstve sú: **ICMP, TCP, UDP, ...**

## Aplikačná vrstva

Protokol na aplikačnej vrstve sa určuje na základe zdrojového a cieľového portu, ktoré sú určené v hlavičke transportnej vrstvy. Najprv si zistím zdrojový a cieľový port, následne sa pozriem do slovníkov **TCP** a **UDP**, kde mám uložené hodnoty známych portov. Ak aspoň jeden z portov sa vyskytuje medzi kľúčmi v slovníkoch TCP alebo UDP, hodnota tohto kľúča sa uloží ako protokol na aplikačnej vrstve.

Najčastejšími protokolmi na aplikačnej vrstve sú:

- **TCP***: ftp-data, ftp-control, ssh, telnet, dns, http, https*
- **UDP**: *dns, bootps, bootpc, tftp, rip, traceroute*

# Filtrovanie

Filtrovanie sa v mojej implementácii spúšťa zadaním argumentu pri spustení programu. Je potrebné zadať prepínač -p nasledovaný názvom protokolu, ktorý chceme vyfiltrovať.

Akceptované protokoly za prepínačom -p, sú:

**TCP**: *http, https, telnet, ssh, ftp-control, ftp-data*

**UDP**: *tftp*

**ICMP**

**ARP**

Príklad korektného spustenia filtrovania: **python3 main.py -p http**

Príklady nekorektného spustenia filtrovania: **python3 main.py -p**

**python3 main.py -p dns**

## Protokol nad TCP

Keď bol pri spustení zadaný ako argument jeden z protokolov nad **TCP** (*http, https, telnet, ssh, ftp-control, ftp-data*) je potrebné kontrolovať kompletnosť všetkých komunikácii. Kompletná komunikácia je taká komunikácia, ktorá obsahuje otvorenie a aj ukončenie spojenia.

Je potrebné vypísať všetky kompletné komunikácie a taktiež prvú nekompletnú komunikáciu.

Túto úlohu som riešil tak, že som si vytvoril triedu *TcpFilter*, do ktorej ukladám všetky rámce, pre konkrétnu komunikáciu, ktorá prebieha medzi dvoma IP adresami a dvoma portami. Po zadaní protokolu nad TCP, teda prechádzam všetky rámce a kontrolujem, či ich aplikačný protokol je zhodný so zadaným protokolom pri spustení programu. Ak sa nájde zhoda, pozriem sa do poľa *fil*, ktoré slúži na ukladanie všetkých filtrovaných komunikácii. Postupne prechádzam každú komunikáciu v tomto poli a zisťujem, či aj tento rámec patrí do danej komunikácie, slúži na to nasledujúca funkcia:

```
#definovanie, kedy sú dve triedy zhodné
def __eq__(self, TcpFilter):
    if TcpFilter.src==self.src and TcpFilter.dst==self.dst and TcpFilter.src_port==self.src_port and TcpFilter.dst_port==self.dst_port:
        return True
    elif TcpFilter.src==self.dst and TcpFilter.dst==self.src and TcpFilter.src_port==self.dst_port and TcpFilter.dst_port==self.src_port:
        return True
    return False
```

Keď zistí, že rámec patrí do tejto komunikácie, pridá ho do poľa rámcov. V prípade, že tam nepatrí, skontroluje ďalšiu komunikáciu v poli filtrovaných komunikácii. Ak nepatrí ani do jednej z nich, vytvorí sa nová komunikácia a pridá sa do spomínaného poľa.

Pri TCP komunikáciách je potrebné sa tiež pozerať na príznaky, ktoré sú nastavené pri jednotlivých rámcoch. Tie zisťujem vo funkcii *get_flags()* v *main.py* a následne ich aj ukladám, vo forme slovníka, do poľa *flags* v triede *TcpFilter*, pre konkrétnu komunikáciu.

```python
#funkcia, ktorá zisťuje príznaky nastavené v IP hlavičke
def get_flags(packet, o):
    hex_flag = "".join(packet[46+o:48+o])[1:]   #načítanie bytov s flagmi a odstránenie prvého znaku (veľkosť hlavičky)
    flags = {                                   #slovník všetkých príznakov
        'Reserved': 0,
        'Accurate ECN': 0,
        'Congestion Window Reduced': 0,
        'ECN-Echo': 0,
        'Urgent': 0,
        'ACK': 0,
        'PSH': 0,
        'RST': 0,
        'SYN': 0,
        'FIN': 0
    }

    #pole názvov príznakov v rovnakom poradí
    fs = ['Reserved','Accurate ECN','Congestion Window Reduced','ECN-Echo','Urgent','ACK','PSH','RST','SYN','FIN']
    i = 0
    #pre každé číslo vo flagoch
    for num in hex_flag:
        j = 0
        bin_num = f"{int(bin(int('0x' + num, 16)).replace('0b', '')):04d}"         #prepis reprezentácie príznakov z hexadecimálnej do binárnej sústavy
        while j < 4:
            if i == 0:
                j = 2                       #preskočenie prvých troch znakov pri príznaku Reserved
            flags[fs[i]] = int(bin_num[j])  #uloženie získaného čísla do prislúchajúceho príznaku
            i += 1
            j += 1

    return flags            #vrátenie slovníka s príznakmi
```

Keď prejdeme všetky pakety, je potrebné rozhodnúť, ktoré z týchto komunikácii sú kompletné a ktoré nie. Na to slúži funkcia *check_complete_com()* v triede *TcpFilter*. Najskôr sa pozriem, či je komunikácia otvorená (prebehol **3-way handshake**), možné situácie:

- *SYN, SYN+ACK, ACK*
- *SYN, SYN, ACK, ACK*

Následne kontrolujem ukončenie komunikácie (**4-way handshake**), možné situácie:

- *FIN+ACK, ACK, FIN+ACK, ACK*
- *FIN, ACK, FIN, ACK*
- *FIN+ACK, FIN+ACK*
- *FIN, RST*
- *RST*

Komunikácie s úspešným otvorením a ukončením komunikácie sa považujú za **kompletné**.

## Protokol nad UDP

Keď bol pri spustení zadaný ako argument protokol nad **UDP**, teda protokol *TFTP* je potrebné kontrolovať kompletnosť všetkých komunikácii. Kompletná komunikácia je taká komunikácia, kde veľkosť posledného datagramu s dátami je menšia ako dohodnutá veľkosť bloku pri vytvorení spojenia a zároveň odosielateľ tohto paketu prijme ACK od druhej strany.

Je potrebné uviesť všetky rámce a prehľadne ich uviesť v komunikáciách, nielen prvý rámec na UDP porte 69, ale identifikovať všetky rámce každej TFTP komunikácie a prehľadne ukázať, ktoré rámce patria do ktorej komunikácie.

Túto úlohu som riešil tak, že som si vytvoril triedu *TftpFilter*, do ktorej ukladám všetky rámce, pre konkrétnu komunikáciu, ktorá prebieha medzi dvoma IP adresami a dvoma portami. Po zadaní protokolu TFTP, teda prechádzam všetky rámce a kontrolujem, či ich aplikačný protokol je zhodný so zadaným protokolom pri spustení programu. Ak sa nájde zhoda, pozriem sa do poľa *fil*, ktoré slúži na ukladanie všetkých filtrovaných komunikácii. Postupne prechádzam každú komunikáciu v tomto poli

a zisťujem, či aj tento rámec patrí do danej komunikácie, slúži na to nasledujúca funkcia v triede *TftpFilter*:

```
#definovanie, kedy sú dve triedy zhodné
def __eq__(self, TFTP):
    if TFTP.src==self.src and TFTP.dst==self.dst and TFTP.src_port==self.src_port and TFTP.dst_port==self.dst_port:
        return True
    elif TFTP.src==self.dst and TFTP.dst==self.src and TFTP.src_port==self.dst_port and TFTP.dst_port==self.src_port:
        return True
    elif TFTP.src==self.dst and TFTP.dst==self.src and TFTP.dst_port==self.src_port:
        return True
    return False
```

Keď zistí, že rámec patrí do tejto komunikácie, pridá ho do poľa rámcov. V prípade, že tam nepatrí, skontroluje ďalšiu komunikáciu v poli filtrovaných komunikácii. Ak nepatrí ani do jednej z nich, vytvorí sa nová komunikácia a pridá sa do spomínaného poľa.

Pri TFTP komunikáciách je potrebné sa tiež pozerať na pole *opcode*, ktoré hovorí o informácií, ktorú aktuálny rámec prenáša. Tie zisťujem vo funkcii *get_opcode()* v *main.py* a následne ich aj ukladám do poľa *opcodes* v triede *TftpFilter*, pre konkrétnu komunikáciu.

Keď prejdeme všetky pakety, je potrebné rozhodnúť, ktoré TFTP komunikácie sú kompletné a ktoré nie. Na to slúži funkcia *check_complete_com()* v triede *TftpFilter*, ktorá prechádza všetky rámce v konkrétnej komunikácií a sleduje, či sa strieda blok s dátami a blok s ACK. Na konci skontroluje, či posledný blok s dátami má menšiu ako dohodnutú veľkosť, ak áno, skontroluje, či prišlo ACK z druhej strany. Vtedy považuje TFTP komunikáciu za kompletnú.

## Protokol ICMP

Keď bol pri spustení zadaný ako argument protokol **ICMP** je potrebné kontrolovať kompletnosť všetkých komunikácii. Kompletná komunikácia je určená na základe nasledujúceho princípu:

- identifikujeme dvojice IP source a IP destination
- identifikujeme v hlavičke polia Identifier a Sequence
- nová komunikácia je daná dvojicou IP adries a polom Identifier

Je potrebné identifikovať všetky typy ICMP správ

Túto úlohu som riešil tak, že som si vytvoril triedu *ICMP*, do ktorej ukladám všetky rámce, pre konkrétnu komunikáciu, ktorá prebieha medzi dvoma IP adresami a portami. Po zadaní protokolu ICMP, teda prechádzam všetky rámce a kontrolujem, či ich protokol je zhodný so zadaným protokolom pri spustení programu. Ak sa nájde zhoda, pozriem sa do poľa *fil*, ktoré slúži na ukladanie všetkých filtrovaných komunikácii. Postupne prechádzam každú komunikáciu v tomto poli a zisťujem, či aj tento rámec patrí do danej komunikácie, slúži na to nasledujúca funkcia v triede *ICMP*:

```
#definovanie, kedy sú dve triedy zhodné
def __eq__(self, ICMP):
    if self.id != -1 and ICMP.src==self.src and ICMP.dst==self.dst and ICMP.id==self.id:
        return True
    elif self.id != -1 and ICMP.src==self.dst and ICMP.dst==self.src and ICMP.id==self.id:
        return True
    return False
```

Keď prejdeme všetky pakety, je potrebné rozhodnúť, ktoré ICMP komunikácie sú kompletné a ktoré nie. Na to slúži funkcia *check_complete_com()* v triede *ICMP*, ktorá prechádza všetky rámce v konkrétnej komunikácií a sleduje, či sa striedajú správy **request** a **reply,** prípadne **Time exceeded**.

Za kompletné komunikácie sa považujú:

- *Echo request – Echo reply*
- *Echo request – Time exceeded*

```
#funkcia, ktorá kontroluje, či je komunikácia kompletná
def check_complete_com(self):
    req_sent = False
    rep_sent = False
    sequence = 0
    #for-cyklus, ktorý prechádza všetky rámce komunikácie
    for i, frame in enumerate(self.frames):
        #kontroluje striedanie Echo request a Echo reply / Time exceeded
        if i%2 == 0:
            if 'icmp_type' in frame and frame['icmp_type'].upper() == "Echo request".upper():
                sequence = frame['icmp_seq']
                req_sent = True
                rep_sent = False
            #v prípade nesprávneho poradia vráti False
            else:
                return False
        else:
            if req_sent and frame['icmp_type'].upper() == "Echo reply".upper() and sequence == frame['icmp_seq']:
                rep_sent = True
                req_sent = False
            elif req_sent and frame['icmp_type'].upper() == "Time exceeded".upper() and sequence == frame['icmp_seq']:
                rep_sent = True
                req_sent = False
            #v prípade nesprávneho poradia vráti False
            else:
                return False
    return True
```

## Protokol ARP

Keď bol pri spustení zadaný ako argument protokol **ARP** je potrebné kontrolovať kompletnosť všetkých komunikácii.

Túto úlohu som riešil tak, že som si vytvoril triedu *ArpFilter*, do ktorej ukladám všetky rámce, pre konkrétnu komunikáciu, ktorá prebieha medzi dvoma IP adresami a portami. Po zadaní protokolu ARP, teda prechádzam všetky rámce a kontrolujem, či ich protokol je zhodný so zadaným protokolom pri spustení programu. Ak sa nájde zhoda, pozriem sa do poľa *fil*, ktoré slúži na ukladanie všetkých filtrovaných komunikácii. Postupne prechádzam každú komunikáciu v tomto poli a zisťujem, či aj tento rámec patrí do danej komunikácie, slúži na to nasledujúca funkcia v triede *ArpFilter*:

```
#metóda kontrolujúca, či sú dve triedy zhodné
def is_same(self, ArpFilter, opcode):
    if opcode == "REQUEST" and self.dst == ArpFilter.dst:
        return True
    elif opcode == "REPLY" and self.dst == ArpFilter.src:
        return True
    return False
```

Keď zistí, že rámec patrí do tejto komunikácie, pridá ho do poľa rámcov, na základe jeho hodnoty *opcode*. Keď je **REQUEST**, pridá sa do poľa *requests* v triede *ArpFilter*, keď je **REPLY**, pridá sa do poľa *reply* v triede *ArpFilter*. V prípade, že tam nepatrí, skontroluje ďalšiu komunikáciu v poli filtrovaných

komunikácii. Ak nepatrí ani do jednej z nich, vytvorí sa nová komunikácia a pridá sa do spomínaného poľa.

Po prejdení všetkých rámcov sa kontroluje kompletnosť komunikácii. Pre každú **ARP request** správu, sa snaží nájsť **ARP reply**, s ktorým tvorí pár a tieto vypíše do kompletnej komunikácie. Keď *pre ARP request* správu nenájde pár, vloží ho do poľa *reqs*, ktoré na konci vráti aj s kompletnými komunikáciami, aj s prebytočnými *ARP reply* správami.

```
#funkcia, ktorá kontroluje, či je komunikácia kompletná
def check_complete_com(self):
    complete = []
    reqs = []
    #for-cyklus, ktorý prechádza všetky requesty
    for i, req in enumerate(self.requests):
        found_couple = False
        #for-cyklus, ktorý prechádza všetky reply
        for j, rep in enumerate(self.reply):
            if self.check_couple(req, rep):             #kontroluje, či tvoria pár
                found_couple = True
                complete.append(self.requests[i])
                complete.append(self.reply.pop(j))
        if not found_couple:        #ak nenájde pár pre req, vloží ho do poľa reqs
            reqs.append(req)

    #metóda vracia tri polia, pole kompletných komunikácii a polia requestov a replyov
    return complete, reqs, self.reply
```

## Fragmentovaný IP paket

Táto úloha je rozšírením úlohy ICMP filtra. Teda ako vstupný protokol je v argumente zadaný ICMP.

Keď veľkosť paketu presiahne hodnotu 1500B, rozdelí sa na viacero menších častí, teda fragmentov. V tejto úlohe je potrebné rozšíriť úlohu ICMP filtra, aby dokázala spracovávať aj takéto pakety.

Úlohu som vyriešil tak, že som si vytvoril dve triedy **Fragment** a **FragmentedICMP**. Do triedy Fragment ukladám rámce, ktoré spolu tvoria jednu fragmentovanú správu. Do triedy *FragmentedICMP* potom ukladám jednotlivé fragmenty, ktoré spolu tvoria jednu komunikáciu.

Takto prechádzam všetky pakety a kontrolujem, či sú fragmentované alebo nie. Keď nájdem paket, zistím, či je fragmentovaný, ak áno, vytvorím preň inštanciu triedy Fragment, do ktorej vložím všetky rámce tvoriace jednu ICMP správu. Následne kontrolujem, či sa už tento Fragment nachádza v nejakej komunikácií v poli *fil*. Ak sa nachádza, vložím ho do poľa *fragments* triedy *FragmentedICMP*. Ak sa nenachádza, vytvorím novú inštanciu triedy *FragmentedICMP* a vložím ju do poľa *fil*.

Vyhodnocovanie kompletnosti komunikácie prebieha rovnako, ako pri nefragmentovaných ICMP správach.

# Príklad štruktúry externých súborov pre určenie protokolov a portov

Všetky externé súbory mám uložené v adresári **Protocols**:

- *EtherType.txt* – hodnoty EtherType
- *ICMP_types.txt* – hodnoty ICMP správ
- *ip_protocols.txt* – IP protokoly
- *LLC.txt* – SAP pre IEEE 802.3 LLC
- *PID.txt* – PID pre IEEE 802.3 LLC + SNAP
- *TCP.txt* – TCP porty
- *UDP.txt* – UDP porty

**Príklad externého súboru (TCP.txt):**

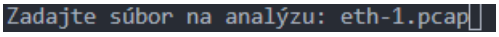
```
7:echo
19:chargen
20:ftp-data
21:ftp-control
22:ssh
23:telnet
25:smtp
53:dns
79:finger
80:http
110:pop3
111:sunrpc
119:nntp
137:netbios-ns
139:netbios-ssn
143:imap
179:bgp
389:ldap
443:https
445:microsoft-ds
514:syslog
1080:socks
17500:DB-LSP-DISC
```

**Príklad externého súboru (EtherType.txt):**

```
0200:XEROX PUP
0201:PUP AddrTrans
0800:IPv4
0801:X.75 Internet
0805:X.25 Level 3
0806:ARP
8035:Reverse ARP
809B:Appletalk
80F3:Appletalk AARP (Kinetics)
8100:IEEE 802.1Q VLAN-tagged frames
8137:Novell IPX
86DD:IPv6
880B:PPP
8847:MPLS
8848:MPLS with upstream-assigned label
8863:PPPoE Discovery Stage
8864:PPPoE Session Stage
88CC:LLDP
9000:ECTP
```

# Používateľské rozhranie

V prípade klasického spustenia programu sa vypíšu do externého súboru všetky spracované rámce z .pcap súboru. Názov tohto súboru je potrebné zadať do štandardného vstupu, keď sa to program opýta.




Po stlačení tlačidla ENTER začne hľadať tento súbor v adresári: *vzorky_pcap_na_analyzu*. Ak sa súbor s takýmto názvom nenájde, vypíše sa chybová hláška a program skončí. Inak sa začne tento súbor spracovávať a vypíše sa do súboru *packets.yaml* v tom istom adresári, kde sa nachádza aj hlavný program.

V prípade, že chceme filtrovať nejakú konkrétnu komunikáciu, musíme protokol zadať ako argument pri spúšťaní programu.

Príklad korektného spustenia filtrovania: **python3 main.py -p http**

Príklady nekorektného spustenia filtrovania: **python3 main.py -p**

**python3 main.py -p dns**

Ďalej program pokračuje klasickým spôsobom, spomínaným vyššie.

# Implementačné prostredie

Moje zadanie som programoval v programovacom jazyku *Python*. Tento jazyk som si zvolil preto, lebo mi viac vyhovuje ako jazyky C / C++. V mnohých veciach sa s ním lepšie pracuje a ľahšie sa spracovávajú reťazce a polia, čo bolo základom tohto zadania.

Ako IDE som použil *Visual Studio Code*, pretože sa mi páči jeho jednoduchosť a prehľadnosť.

# Zhodnotenie a možnosti rozšírenia

V tomto zadaní bolo našou úlohou navrhnúť a implementovať programový analyzátor Ethernet siete, ktorý analyzuje komunikácie v sieti zaznamenané v .pcap súbore.

V rámci tohoto zadania som navrhol a úspešne implementoval programový analyzátor Ethernet siete, ktorý je schopný analyzovať komunikácie v sieti zaznamenané v .pcap súbore.

Hlavné vlastnosti a funkcie môjho analyzátoru zahŕňajú:

- **spracovanie .pcap súborov**: program je schopný načítať a analyzovať .pcap súbory rôznych veľkostí a komplexnosti
- **rozpoznávanie protokolov**: analyzátor je schopný identifikovať rôzne sieťové protokoly, ako napríklad TCP, UDP, HTTP a mnohé ďalšie
- **analyzovanie komunikácie**: program poskytuje detailné informácie o prenášaných dátach, zahŕňajúc zdrojovú a cieľovú IP adresu, porty, dĺžku paketov a ďalšie dôležité údaje
- **možnosti filtrovania**: analyzátor poskytuje možnosti filtrovania komunikácie na základe zadaného protokolu

**Možnosti rozšírenia**

V budúcnosti by sa môj analyzátor dal rozšíriť o:

- filtrovanie ďalších protokolov
- filtrovanie podľa ďalších kritérií
- a podobne.