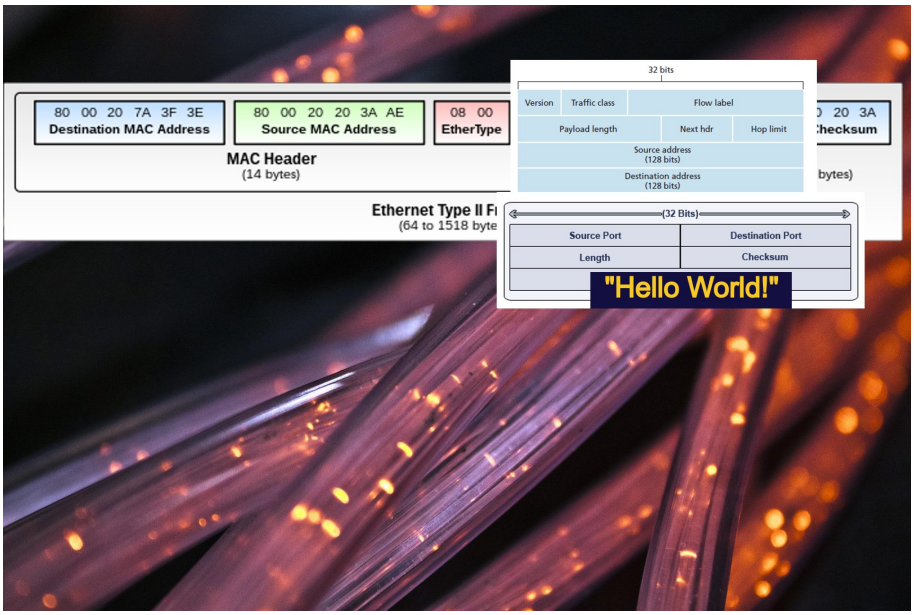
Packet Sniffer in Python

Alla scoperta delle reti informatiche

Matteo Bernini

https://github.com/mattebernini/

https://t.me/mattebernini



21 settembre 2022

Indice

1 Introduzione

2 La pila protocollare

Internet una delle tante possibili implementazioni di una inter-rete informatica, visto il successo che ha avuto e visto che presa come modello nel progettare reti informatiche si pu dire che sia la migliore delle implementazioni possibili. Questo complesso sistema di reti e sottoreti che vanno a formare un unica grande rete che connette tutto il mondo organizzata su pi livelli, l'idea quella che ogni livello fornisca dei servizi al livello superiore e ne riceva dal livello sottostante, cos che la manutenzione del sistema sia il pi semplice possibile (cambiando l'implementazione di un livello non devo alterare gli altri a patto di garantire i medesimi servizi al livello superiore). Questa organizzazione a livelli viene chiamata "pila protocollare" o "pila dei protocolli" e consiste in 5 livelli, ognuno di questi livelli aggiunge un header al dato che vogliamo inviare nel pacchetto di bit che invieremo fisicamente all'interno di Internet (partendo dall'alto, dal PC che vuole inviare una stringa "Hello World!":

1. Application: questo livello il livello a cui opera il programmatore, il livello pi alto, la cui unica preoccupazione decidere cosa inviare e a chi, in questo esempio il cosa la stringa sopracitata, il chi viene identificato da una stringa di 32 bit che identifica l'host ed un numero naturale che identifica il processo destinatario (indirizzo IP e porta). Il come delegato al livello sottostante, a questo livello basta decidere il protocollo di trasporto che si vuole usare, TCP o UDP.

2. Transport: il compito principale di questo livello quello di instaurare una comunicazione da processo a processo di due diversi host (mittente e destinatario), ci pu essere fatto principalmente tramite due diversi protocolli TCP e UDP (il primo affidabile, l'altro no ma pi veloce). A questo livello viene aggiunto un header alla stringa "Hello World!" che vogliamo inviare dove si specificano porta sorgente e porta destinataria pi altre informazioni a seconda del protocollo. A questo punto dell'incapsulamento abbiamo il "segment" costituito dall'header trasporto e il dato applicazione (la stringa).

3. Network: come dice il nome del livello la sua mansione quella di portare il segment dall'host mittente all'host destinatario tramite tutta la rete Internet, per fare ci viene aggiunto un header IP al segment che va a formare il datagram, il quale verr decodificato ad ogni router per fare in modo che il pacchetto venga indirizzato correttamente all'host destinatario[1].

4. Data-Link: fornisce al livello Network il servizio di trasportare il datagram tra le varie connessioni fisiche della rete, praticamente aggiungendo un header in testa e un trailer in coda al datagram completa il processo di incapsulamento del pacchetto formando il frame, header e trailer ovviamente dipendono dal protocollo.

5. Phisical: al pi basso livello si fornisce il servizio al Data-Link layer di trasportare fisicamente i bit.

[1] Questo meccanismo prende il nome di Routing.

3 Il viaggio della stringa "Hello World!"

Mettiamo il caso che io voglia inviare una stringa "Hello World!" tramite protocollo TCP (trasporto affidabile) al mio amico e che io conosca il suo indirizzo IP e la porta dalla quale in ascolto, come faccio?

```
# il mio programma
import socket
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sd:
    sd.connect((IP_amico, PORTA_AMICO))
    sd.send(bytes("Hello World!"))
```

Semplice, utilizzando un servizio fornito dal sistema operativo chiamato socket che mi permette di inviare dati dal livello applicazione, con 4 linee di codice ho inviato una stringa, lo potevo fare anche non conoscendo il funzionamento dei 4 livelli sottostanti della pila protocollare? Si, questo il grande vantaggio della pila! Detto questo per a noi interessa come abbia viaggiato pertanto:

- Sul nostro PC alla stringa viene aggiunto un header TCP formando il segment, dopodich viene creato il datagram aggiungendo l'header IP, infine il datagram viene racchiuso da head e trailer formando il frame.
- Il frame viaggia fisicamente fino al router di casa dove viene decapsulato fino al datagram, tramite il datagram il router decider il prossimo router che il pacchetto deve raggiungere per avvicinarsi all'host destinatario, incapsuler il datagram e invier il frame al prossimo router scelto fino a quando non si arriver all'host destinatario.
- Arrivati all'host destinatario (il PC del mio amico) il frame verr decapsulato fino al segment, dove sar indicata la porta di destinazione, a questo punto il sistema operativo del mio amico sapendo il processo associato alla porta far arrivare la stringa al mio amico (sempre grazie al socket).

Ovviamente il mio amico doveva scrivere un programma per ricevere la stringa:

```
# il programma del mio amico
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((LOCALHOST, PORTA_AMICO))
    s.listen()
    sd, addr = s.accept()
    with sd:
        str = sd.recv(1024)
        print(str)      # stampa "Hello World!"
```

4 Packet Sniffing

Il packet sniffing un metodo hacker che consente all'attaccante di leggere i pacchetti che passano in rete, per fini didattici noi applicheremo lo stesso metodo alla nostra scheda di rete, cos da poter leggere tutti i pacchetti che arrivano a livello Data-Link sul nostro PC e allo stesso tempo per evitare di commettere un reato (si, il packet sniffing reato ma noi sniffiamo i nostri stessi pacchetti quindi no).

4.1 Leggere frame Data-Link

Per poter leggere tutti i frame che arrivano al nostro PC utilizzeremo un tipo speciale di socket:

```
s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))
```

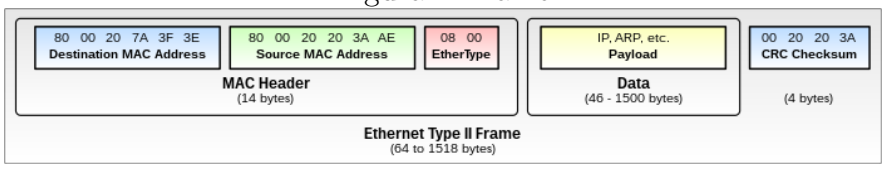
Oltre alla libreria socket per dovremo utilizzare anche la libreria struct per fare il parsing dei singoli byte dell'header. La funzione main sar strutturata pi o meno cos:

```
s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))
while True:
    raw_data, addr = s.recvfrom(65535)
    # raw_data contiene il frame, ma senza decapsulamento
    # sono soltanto un insieme di bit senza senso
    print(raw_data)     # stampa bit a caso
```

Per ogni livello scriveremo una funzione che decapsuler il pacchetto e che ci fornir la informazioni necessarie per decapsulare gli altri pacchetti di livello superiore.

4.2 Decapsuliamo il frame

Figura 1: frame



Come detto nei capitoli precedenti il frame il pacchetto di livello Data-Link, lo standard del formato dei frame Ethernet, tale standard ci dice che l'head formata da 6 bytes per l'indirizzo MAC[2] di destinazine, altri 6 per quello sorgente e 2 per il tipo di protocollo utilizzato a livello Network, nel trailer presente il checksum per fare error detection, in mezzo a header e trailer sta il datagram che ha una dimensione massima di 1500 byte.

```
def ethernet_head(raw_data):
    dest, src, proto = struct.unpack('! 6s 6s H', raw_data[:14])
    dest_mac = dest
    src_mac = src
    network_protocol = socket.htons(proto)
    datagram = raw_data[14:]
    return dest_mac, src_mac, network_protocol, datagram
    # il parsing viene fatto solo dell'header, il trailer ha poco
   significato
```

4.3 Decapsuliamo il datagram

Stiamo risalendo la pila protocollare, prima eravamo a livello Data-Link adesso siamo a livello Network, abbiamo decapsulato il frame e adesso conosciamo sia il datagram che il protocollo a

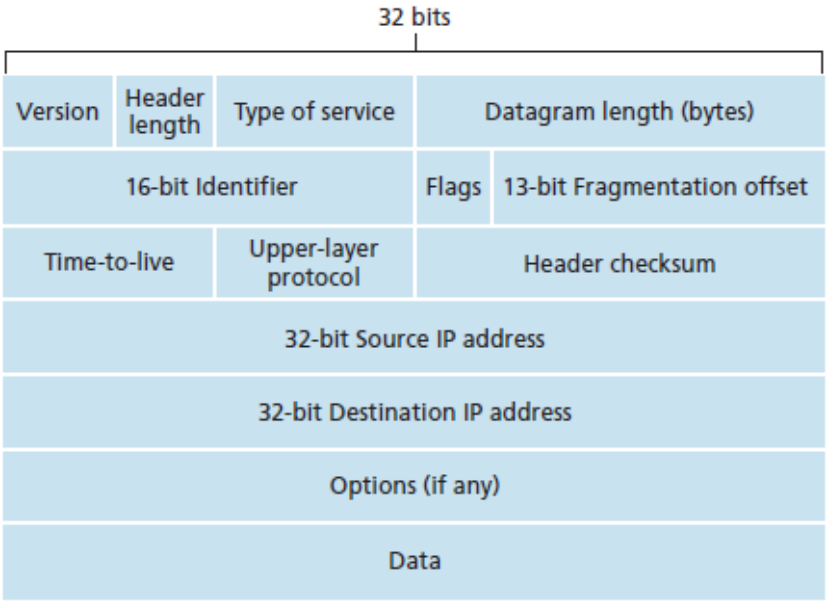
[2]Indirizzo fisico unico che identifica ogni scheda di rete al mondo, a livello Data-Link si utilizza questo indirizzo e non l'IP e la porta.

livello Network che il datagram usa. Per decapsulare il datagram dovremo quindi: individuare il tipo di protocollo e scrivere una funzione che faccia il parsing dell'header di tale protocollo per ogni protocollo livello Network. Per semplicit lo faremo soltanto di due protocolli, i pi famosi ed usati, IP versione 4 e versione 6. Dalla funzione di parsing del frame si pu notare che la terza variabile di ritorno restituisce il tipo, pertanto possiamo aggiornare il main cos:

```
raw_data, addr = s.recvfrom(1518)
eth = ethernet_head(raw_data)
stampa_eth_header(eth)
if eth[2] == 8:
    ipv4 = ipv4_head(eth[3])
elif eth[2] == 34525:
    ipv6 = ipv6_head(eth[3])
```

4.3.1 IPv4

Figura 2: datagram IPv4



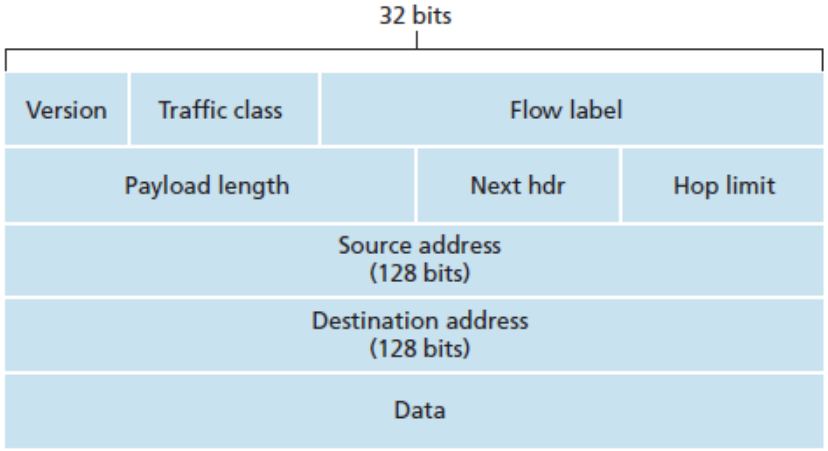
Il datagram composto di un header e di una sezione data che contiene il segment di livello Transport. L'header del datagram IPv4 diviso in righe da 4 byte, la prima riga contiene nel primo byte 4 bit per la versione (sempre 4) e altri 4 per la lunghezza dell'header (l'header contiene opzioni, quindi variabile), 1 byte di poco conto e 2 byte con la lunghezza del datagram (¡1500 byte). La seconda riga contiene informazioni relative alla frammentazione dei datagram pi grandi di 1500 in datagram pi piccoli. La terza riga ha nel 1o byte il numero massimo di hop[3] rimanenti, uno che definisce il protocollo di livello trasporto e due per il checksum dell'header. Le righe 4 e 5 sono rispettivamente per gli IP di mittente e destinatario. La sesta riservata alle opzioni (poco utilizzate).

[3]Un hop il passaggio di un pacchetto da un dispositivo di livello Network ad un altro (da host a router, da router a router, da router a host).

```
def ipv4_head(raw_data):
    version_header_length = raw_data[0]
    version = version_header_length >> 4
    header_length = (version_header_length & 15) * 4
    ttl, transport_protocol, src_IP, dst_IP = struct.unpack('! 8x B B 2x 4
   s 4s', raw_data[:20])
    segment = raw_data[header_length:]
    src_IP = get_ip(src_IP)
    dst_IP = get_ip(dst_IP)
    return version, header_length, ttl, transport_protocol, src_IP, dst_IP
   , segment
    # alcune informazioni sono omesse come la seconda riga o il checksum
```

4.3.2 IPv6

Figura 3: datagram IPv6



Il datagram della successiva versione 6 molto pi semplice, infatti questa nuova versione fu introdotta al fine di semplificare sia gli indirizzamenti di IPv4 che erano limitati a 32 bit che il protocollo stesso. I primi 2 byte sono uguali alla versione precedente, il resto della riga occupato dal flow label che un campo speciale per richiedere responsiveness (non ci interessa molto). Seconda riga abbiamo 2 byte per la lunghezza del campo data (la lunghezza dell'header fissata a 40 byte), 1 per il protocollo di livello trasporto e un altro per il numero massimo di hop. Seguono l'indirizzo su 128 bit di mittente e destinatario e il campo data che contiene il segment.

```
def ipv6_head(raw_data):
    version = raw_data[0] >> 4
    lenght, ttl, transport_protocol = struct.unpack('! 16s 8s 8s',
   raw_data[:20])
    src_IP, dest_IP = struct.unpack('! >QQ >QQ', raw_data[:8])
    data = raw_data[:40]
    return version, lenght, ttl, transport_protocol, src_IP, dest_IP, data
```

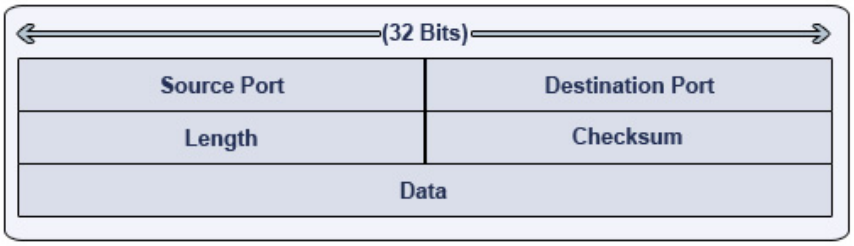
4.4 Decapsuliamo il segment

Dal decapsulamento del datagram abbiamo ottenuto, a prescindere dal protocollo, il segment e il protocollo di livello trasporto utilizzato, possiamo dunque semplicemente decodificare il segment come fatto per gli altri pacchetti:

```
# nel main
if ipv4[3] == 6:
    tcp = tcp_head(ipv4[6])
    stampa_tcp_header(tcp)
    print(tcp[10])
if ipv4[3] == 17:
    udp = udp_head(ipv4[6])
    stampa_udp_header(udp)
    print(udp[4])
```

4.4.1 UDP

Figura 4: segment UDP



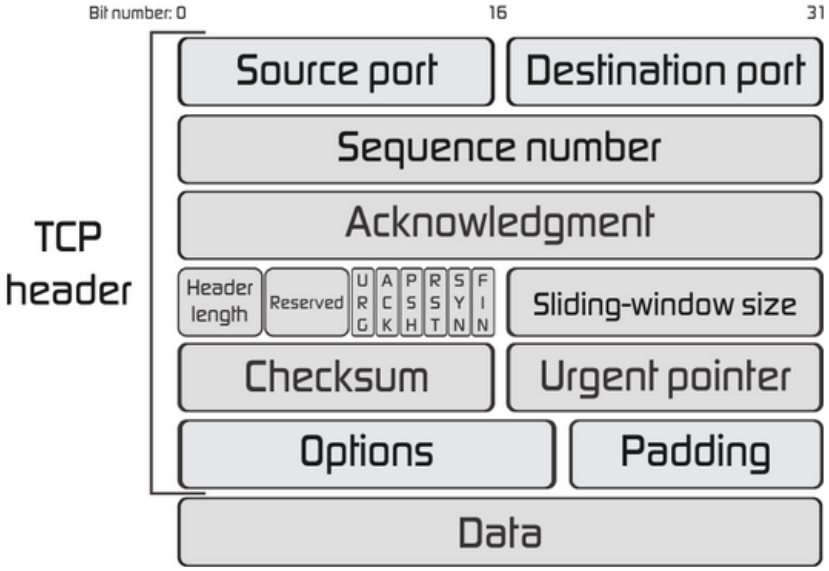
L'header molto semplice, infatti il pregio principale di questo protocollo appunto la semplicit e la velocit, costituito da 8 byte che indicano porta sorgente e destinataria (4 byte ciascuna), 4 byte che indicano la lunghezza del segment (header+data) e 4 per il checksum (usato solo per error detection).

```
def udp_head(raw_data):
    (src, dst, lenght, checksum) = struct.unpack('! H H H H', raw_data
    [:8])
    data = raw_data[8:]
    return src, dst, lenght, checksum, data
```

4.4.2 TCP

Il protocollo TCP il protocollo di trasporto affidabile, pertanto presenta un header pi complesso. Sempre suddiviso in righe da 32 bit costituito da una prima riga identica a quello UDP, nella seconda c' il sequence number che indica il primo byte che inviamo dal messaggio suddiviso (sempre perch c' la MTU di 1500 byte), nella terza l'ACK number che serve per comunicare al destinatario fino a dove ho ricevuto ci che mi ha mandato, nella quarta ho la lunghezza dell'header, i flag e la receive window (per implementare il controllo di flusso), nella quinta riga abbiamo il checksum e l'urgent data pointer (indica che da li inizia una sezione urgente).

Figura 5: segment TCP




```
def tcp_head(raw_data):
    (src_port, dest_port, sequence, acknowledgment, offset_reserved_flags,
    recv_window, checksum, urg_data_ptr) = struct.unpack('! H H L L H H H
   H', raw_data[:20])
    offset = (offset_reserved_flags >> 12) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_psh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1
    data = raw_data[offset:]
    return src_port, dest_port, sequence, acknowledgment, flag_urg,
   flag_ack, flag_psh, flag_rst, flag_syn, flag_fin, data, recv_window,
   checksum, urg_data_ptr
```

5 Leggere tutti i pacchetti

```
def main():
    s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))
    while True:
        raw_data, addr = s.recvfrom(1518)
        eth = ethernet_head(raw_data)
        stampa_eth_header(eth)
        if eth[2] == 8:
            ipv4 = ipv4_head(eth[3])
            stampa_ipv4_header(ipv4)
            if ipv4[3] == 6:
                tcp = tcp_head(ipv4[6])
```

```
                stampa_tcp_header(tcp)
                print(tcp[10])
            elif ipv4[3] == 17:
                udp = udp_head(ipv4[6])
                stampa_udp_header(udp)
                print(udp[4])
        elif eth[2] == 0x86DD:
            ipv6 = ipv6_head(eth[3])
            stampa_ipv6_header(ipv6)

    main()
```

Codice completo su github . Le funzioni di decapsulamento scritte nei capitoli precedenti ci permettono di leggere letteralmente tutti i pacchetti che ci arrivano e che mandiamo, possiamo dunque sfruttare questi concetti per tanti mini-progetti.

6 Parsing indirizzi IP

L'intenzione di questo capitolo quella di mostrarci quali sono gli indirizzi IP con i quali il nostro PC comunica, infatti decapsulando il datagram posso conoscere quali sono gli IP che inviano e ricevono pacchetti, per semplicit il programma pensato solo per IPv4. Gli indirizzi IPv4 (da ora chiamati solo IP per comodit) sono di due tipi: pubblici e privati, questo perch sono su 32 bit e quindi c'era bisogno di un modo per simulare pi indirizzi IP di quelli che in realt ci sono, gli indirizzi privati che il nostro sniffer intercetta sono quelli della nostra sottorete (se siete al wifi di casa sono gli indirizzi che vi mandano pacchetti e sono collegati al wifi), questi indirizzi sono del tipo 10.x.x.x o 192.x.x.x, inoltre sappiamo che tutti gli indirizzi del tipo 127.x.x.x sono di loopback (sono pacchetti mandati a noi stessi).

```
def ip_privato(ipv4):
    if ipv4.startswith("192.") or ipv4.startswith("10.") or ipv4.
   startswith("127."):
        return True
    else:
        return False
```

A questo punto sapendo riconoscere gli indirizzi pubblici e privati posso scartare i secondi, con un po' di Python List ecco il programma:

```
# libreria con le funzioni dei capitoli precedenti
import packet_sniffer as ps
import socket
import time

src_ip = []
dest_ip = []
public_ip = []
start = time.time()

s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))
while True:
    time.sleep(1)
    raw_data, addr = s.recvfrom(1518)
```

```
    eth = ps.ethernet_head(raw_data)
    if eth[2] == 8:
        ipv4 = ps.ipv4_head(eth[3])

        if ipv4[4] not in src_ip:
            src_ip.append(ipv4[4])
        if ipv4[5] not in dest_ip:
            dest_ip.append(ipv4[5])

        if ps.ip_privato(ipv4[4]) == False:
            public_ip.append(ipv4[4])
        if ps.ip_privato(ipv4[5]) == False:
            public_ip.append(ipv4[5])

    end = time.time()
    if (end - start) > 60:
        print("src IP:")
        print(src_ip)
        print("dest IP:")
        print(dest_ip)
        print("public IP:")
        print(public_ip)
        break
```

Il programma per un minuto ogni secondo campiona gli IP di source e dest nel datagram e li salva nell'apposita lista, dopo un minuto si ferma e stampa le 3 liste che ci interessano: IP sorgenti, destinatari e pubblici. Su whatismyipaddress.com/ip/ aggiungendo l'ip pubblico dopo "/ip/" possiamo sapere a chi appartiene e dove si trova il server.

6.1 ARP emulator

Il protocollo ARP (Address Resolution Protocol) un protocollo al confine tra il livello network e il livello data-link, infatti si occupa di tenere aggiornata una tabella che associa ad ogni indirizzo IP l'indirizzo MAC della scheda di rete a cui mandare effettivamente il frame. Con delle leggere modifiche al programma precedente possiamo ottenere una cosa simile, ovvero associare agli IP pubblici l'indirizzo MAC (stavolta usando i dizionari):

```
import packet_sniffer as ps
import socket
import time

start = time.time()
arp = {}

s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))
while True:
    time.sleep(1)
    raw_data, addr = s.recvfrom(1518)
    eth = ps.ethernet_head(raw_data)
    if eth[2] == 8:
        ipv4 = ps.ipv4_head(eth[3])

        if ipv4[4] not in arp:
```

```
            arp.update({ipv4[4] : eth[1]})
        if ipv4[5] not in arp:
            arp.update({ipv4[5] : eth[0]})


    end = time.time()
    if (end - start) > 60:
        print("IP\t\t\t\tMAC")
        for ip in arp.keys():
            print("{}\t->\t{}".format(ip, arp[ip]))
        break
```

7 Esecuzione

Per eseguire i programmi che fanno uso di socket di tipo RAW c' bisogno dei privilegi da root. Spostarsi da terminale sulla cartella contenente il programma che si vuole eseguire e digitare:

```
sudo python3 nome_programma.py

# se vogliamo salvare l'output su file
sudo python3 nome_programma.py > data.txt
```

Prima di eseguire il programma il terminale richieder la password utente.

Gli esempi mostrati nei capitoli precedenti sono solo alcuni dei possibili utilizzi di questo packet sniffer, il cui compito principale quello di far toccare con mano agli studenti i pacchetti che circolano in rete. Il codice completo disponibile su: https://github.com/mattebernini/my-packets-sniffer.

Riferimenti bibliografici

[1] Lezioni del prof Anastasi.

[2] Lezioni e laboratori del prof Pistolesi.