Università di Messina — CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

Hands-On #4

Socket Raw di livello 3: leggere e decodificare l'header IP

Last Update: March 13, 2026

Scenario

Nell'HO precedente hai costruito uno sniffer che cattura i frame Ethernet e ne stampa i MAC address e l'EtherType. Sai quindi che quando l'EtherType vale `0x0800` il payload del frame contiene un pacchetto IPv4.

Ma cosa c'è esattamente in quel payload? Un header IP con informazioni fondamentali: indirizzi sorgente e destinazione, TTL, protocollo di trasporto (TCP, UDP, ICMP...), lunghezza totale.

In questo HO cambiamo approccio e usiamo `AF_INET + SOCK_RAW` — i raw socket di **livello 3**. A questo livello il kernel gestisce per noi il livello Ethernet, e ci consegna direttamente il pacchetto IP grezzo. Impareremo a leggere ogni campo dell'header, a convertire correttamente gli indirizzi IP in stringhe leggibili, e a distinguere i protocolli di trasporto incapsulati.

Concetti chiave da tenere a mente

AF_INET + SOCK_RAW vs AF_PACKET + SOCK_RAW

Nell'HO#2 usavamo `AF_PACKET`: scendevamo al livello 2 e vedevamo tutto, header Ethernet incluso. Ora usiamo `AF_INET`: il kernel decapsula il frame Ethernet per noi e ci consegna il pacchetto a partire dall'header IP.

Famiglia	Punto di accesso	Primo byte ricevuto
AF_PACKET	Livello 2 (Ethernet)	byte 0 dell'header Ethernet
AF_INET	Livello 3 (IP)	byte 0 dell'header IP // questo HO

Per aprire un raw socket a livello 3 si usa `socket()` con `AF_INET` e `SOCK_RAW`. Il terzo argomento non è più un EtherType ma un **numero di protocollo IP** (campo `protocol` dell'header IPv4). Le costanti sono definite in `<netinet/in.h>`:

Costante	Valore	Protocollo
IPPROTO_ICMP	1	ICMP — riceve solo pacchetti ICMP
IPPROTO_TCP	6	TCP — riceve solo pacchetti TCP
IPPROTO_UDP	17	UDP — riceve solo pacchetti UDP
IPPROTO_RAW	255	solo trasmissione, non riceve nulla

[!] IPPROTO_RAW non riceve nulla

`IPPROTO_RAW` è pensato esclusivamente per la *trasmissione* di pacchetti con header costruito a mano (`IP_HDRINCL`). Se lo usi per aprire un socket in ricezione, `recvfrom()` si blocca per sempre senza ricevere mai nulla — nessun errore, nessun output. Per ricevere

pacchetti usa sempre una costante specifica come `IPPROTO_ICMP` o `IPPROTO_TCP`.

La struttura dell'header IPv4

L'header IP ha una lunghezza minima di 20 byte (senza opzioni). Ogni campo ha una posizione e una dimensione fissa:

Byte 0	Byte 1	Byte 2-3	Byte 4-5
Ver + IHL	DSCP/ECN	Lunghezza totale	Identificazione
Byte 6-7	Byte 8	Byte 9	Byte 10-11
Flags + Offset	TTL	Protocollo	Checksum header
Byte 12-15	Byte 16-19		
IP Sorgente	IP Destinazione		
Byte 20+: Opzioni (se IHL > 5) + Payload			

In C questa struttura è rappresentata da `struct iphdr` definita in `<netinet/ip.h>`:

```c
struct iphdr {
    uint8_t  ihl      : 4;  /* IP Header Length (in parole da 32 bit ) */
    uint8_t  version  : 4;  /* versione IP (4)                          */
    uint8_t  tos;           /* Type of Service / DSCP                   */
    uint16_t tot_len;       /* lunghezza totale pacchetto (byte)        */
    uint16_t id;            /* identificazione (frammentazione)         */
    uint16_t frag_off;      /* flags + offset di frammentazione         */
    uint8_t  ttl;           /* Time To Live                             */
    uint8_t  protocol;      /* protocollo livello superiore             */
    uint16_t check;         /* checksum dell'header                     */
    uint32_t saddr;         /* indirizzo IP sorgente                    */
    uint32_t daddr;         /* indirizzo IP destinazione                */
};
```

Il campo IHL e il payload IP

Il campo `ihl` (IP Header Length) indica la lunghezza dell'header in **parole da 32 bit** (4 byte). Il valore minimo è 5, che corrisponde a $5 \times 4 = 20$ byte. Se `ihl > 5` sono presenti opzioni IP.

Per trovare dove inizia il payload (TCP, UDP, ICMP...):

```c
struct iphdr *ip = (struct iphdr *)buf;
int header_len   = ip->ihl * 4;        /* lunghezza header in byte */
unsigned char *payload = buf + header_len; /* inizio payload */
```

Convertire un indirizzo IP in stringa

Gli indirizzi `saddr` e `daddr` sono `uint32_t` in **network byte order**. Per stamparli in formato `a.b.c.d` si usa `inet_ntoa()` oppure `inet_ntop()` (preferibile perchè thread-safe):

```c
#include <arpa/inet.h>

struct in_addr src;
src.s_addr = ip->saddr;            /* gia' in network byte order */
printf("SRC: %s\n", inet_ntoa(src));

/* oppure con inet_ntop (thread-safe, preferibile): */
char src_str[INET_ADDRSTRLEN];
inet_ntop(AF_INET, &ip->saddr, src_str, sizeof(src_str));
printf("SRC: %s\n", src_str);
```

[!] `inet_ntoa()` non è thread-safe

`inet_ntoa()` usa un buffer statico interno: chiamarla due volte nella stessa `printf()` sovrascrive il risultato della prima chiamata. Usa sempre `inet_ntop()` quando devi stampare sorgente e destinazione nella stessa istruzione.

Codice di partenza

Il programma seguente funziona già correttamente: compilalo, eseguilo con `sudo` e genera traffico ICMP con `ping -c 3 localhost` in un secondo terminale. Osserva l'output, poi passa agli esercizi che ti chiederanno di migliorarlo passo per passo.

Listing 1: ip_sniffer_base.c — funzionante, da migliorare

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/socket.h>     /* socket(), recvfrom()
                            */
#include <netinet/ip.h>     /* struct iphdr
                             */
#include <netinet/in.h>     /* IPPROTO_ICMP, struct sockaddr_in
          */
#include <arpa/inet.h>      /* inet_ntoa()
                               */

#define BUFSIZE 65536

int main(void)
{
    /* AF_INET + SOCK_RAW + IPPROTO_ICMP:
       il kernel consegna tutti i pacchetti ICMP in arrivo,
       gia' decapsulati dal frame Ethernet.
       Il buffer inizia direttamente dall'header IP. */
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sock < 0) { perror("socket"); return 1; }

    printf("In ascolto (solo ICMP)... premi Ctrl+C per uscire\n\n")
        ;

```

```c
    unsigned char buf[BUFSIZE];
    int pkt = 0;

    while (1) {
        ssize_t n = recvfrom(sock, buf, BUFSIZE, 0, NULL, NULL);
        if (n < 0) { perror("recvfrom"); break; }

        /* I primi byte del buffer sono l'header IP */
        struct iphdr *ip = (struct iphdr *)buf;

        /* tot_len e' in network byte order: serve ntohs() */
        printf("Pacchetto #%d  (%d byte)\n",
               ++pkt, ntohs(ip->tot_len));

        /* inet_ntoa() usa un buffer statico interno:
           chiamarla due volte nella stessa printf sovrascrive
           il primo risultato. Per ora usiamo due printf separate.
              */
        struct in_addr s, d;
        s.s_addr = ip->saddr;
        d.s_addr = ip->daddr;
        printf("  SRC: %s\n", inet_ntoa(s));
        printf("  DST: %s\n", inet_ntoa(d));

        printf("  TTL: %d  Proto: %d\n\n",
               ip->ttl, ip->protocol);
    }

    close(sock);
    return 0;
}
```

▷ Come generare traffico di test

In un secondo terminale, mentre il programma è in ascolto:

```
ping -c 3 localhost        # ICMP su loopback, visibile subito
ping -c 3 8.8.8.8          # ICMP verso internet (richiede rete)
```

Il programma attuale vede solo ICMP. Gli esercizi ti porteranno ad estenderlo per ricevere anche TCP e UDP.

Esercizi

Esercizio 1 — Stampa leggibile del protocollo

Il campo `ip->protocol` viene stampato come numero intero. Scrivi una funzione che lo converte in una stringa leggibile e usala al posto del `%d`:

Listing 2: Prototipo da implementare

```c
/*
 * Restituisce il nome del protocollo IP dato il suo numero.
 * Esempio: proto_name(1) -> "ICMP"
 *          proto_name(6) -> "TCP"
 */
const char *proto_name(uint8_t protocol);
```

La funzione deve riconoscere almeno ICMP (1), TCP (6), UDP (17). Per tutti gli altri valori restituire `"sconosciuto"`.
L'output atteso diventa:

```
  TTL: 64   Proto: ICMP (1)
```

Esercizio 2 — Decodifica completa dell'header IP

Estendi il programma per stampare **tutti** i campi significativi dell'header IP. L'output per ogni pacchetto deve essere simile a questo:

```
--- Pacchetto #1 ---
  Versione  : 4
  IHL       : 20 byte (5 parole da 32 bit)
  Lunghezza : 84 byte
  TTL       : 64
  Protocollo: ICMP (1)
  SRC       : 127.0.0.1
  DST       : 127.0.0.1
  Flags     : DF
  Frag off  : 0
```

Per estrarre i **flag** e l'**offset di frammentazione** dal campo `frag_off` (che li contiene compattati insieme):

Listing 3: Estrazione flag e offset

```c
uint16_t fo = ntohs(ip->frag_off);

int flag_df = (fo >> 14) & 1;   /* Don't Fragment */
int flag_mf = (fo >> 13) & 1;   /* More Fragments */
int offset  = fo & 0x1FFF;      /* Fragment Offset (in unit. da
    8 byte) */
```

▷ Suggerimento

`ihl` contiene il numero di *parole da 32 bit*, non i byte. Moltiplica per 4 per ottenere i byte. Il valore minimo è 5 (= 20 byte); se `ihl < 5` il pacchetto è malformato e va scartato.

Challenge — Naviga nel payload: decodifica l'header ICMP

Dopo l'header IP si trova il payload — nel caso di ICMP, l'header ICMP. Scrivi una funzione che calcola il puntatore corretto al payload tenendo conto delle eventuali opzioni IP:

Listing 4: Prototipo da implementare

```c
/*
 * Restituisce puntatore all'inizio del payload IP.
 * Scrive in *payload_len la sua lunghezza in byte.
 * Ritorna NULL se il pacchetto e' malformato o troppo corto.
 */
unsigned char *ip_payload(const unsigned char *buf,
                          ssize_t n,
                          size_t *payload_len);
```

Usa la funzione per stampare, per ogni pacchetto ICMP, i campi dell'header ICMP (definito in `<netinet/ip_icmp.h>`):

```c
struct icmphdr {
    uint8_t  type;      /* tipo: 8=Echo Request, 0=Echo Reply */
    uint8_t  code;      /* codice (di solito 0) */
    uint16_t checksum; /* checksum dell'header ICMP */
    union {
        struct { uint16_t id; uint16_t sequence; } echo;
        /* altri campi... */
    } un;
};
```

L'output atteso per un `ping`:

```
  ICMP type=8 (Echo Request)  id=1234  seq=1
```

[!] Attenzione

Controlla sempre che `n >= ip->ihl * 4` prima di calcolare il puntatore al payload. Un pacchetto troncato potrebbe avere `n` minore della lunghezza dichiarata dall'header — accedere oltre il buffer sarebbe comportamento indefinito.

Consegna: inviare zip con report e sorgenti al docente (`roberto.marino@unime.it`)
Il primo che mi da la soluzione della challenge... avrà un regalo da parte del docente :)