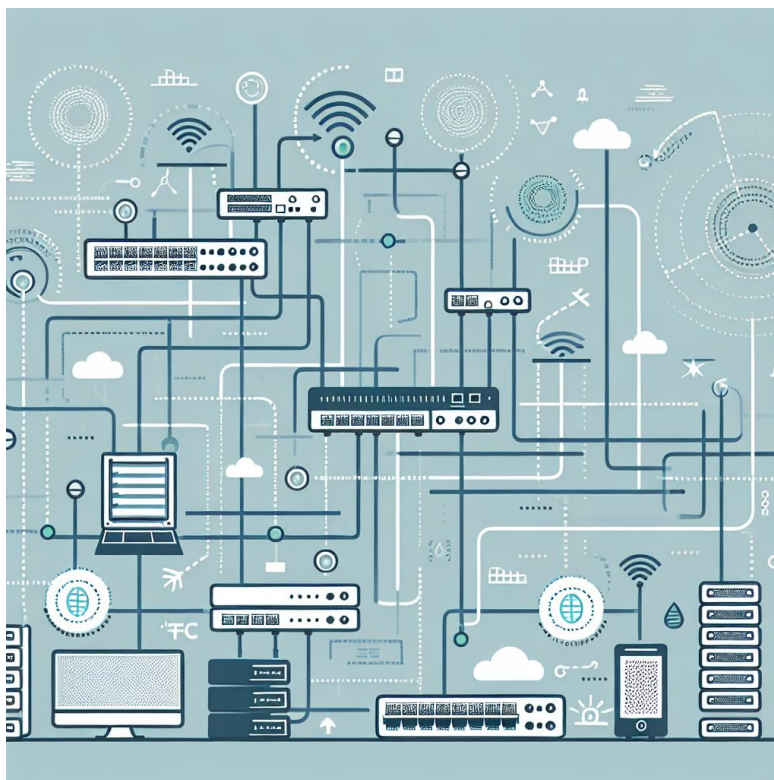


Analisi del traffico ICMP e TCP con Scapy e Wireshark

Relazione per il corso di Programmazione di Reti



Presentato da:
Bartocetti Enrico, 0001115097

Indice

1	Introduzione	2
2	Pacchetti ICMP Echo	3
2.1	Generazione dei pacchetti con Scapy	3
2.2	Analisi del traffico con Wireshark	3
2.2.1	ICMP Echo Request	4
2.2.2	ICMP Echo Reply	5
3	Segmenti TCP SYN	6
3.1	Generazione dei pacchetti con Scapy	6
3.2	Analisi del traffico con Wireshark	6
3.2.1	Richiesta TCP con SYN attivo	7
3.2.2	Risposta TCP con SYN ACK attivi	8
3.2.3	Ritrasmissioni TCP con SYN ACK attivi	9
4	Segmenti TCP con flag personalizzati	10
4.1	Generazione dei pacchetti con Scapy	10
4.2	Analisi del traffico con Wireshark	11
4.2.1	Tutti i flag attivi	11
4.2.2	Three-way handshake	11
5	Commenti finali	13
5.1	Differenze tra ICMP e TCP	13

Capitolo 1

Introduzione

Obiettivo

Creare e inviare pacchetti ICMP e TCP con Scapy, catturarli con Wireshark e analizzare i risultati.

Requisiti minimi

- Inviare pacchetti ICMP Echo (ping) e TCP SYN verso un host
- Catturare i pacchetti in Wireshark e salvarli in .pcap
- Analizzare: IP di origine/destinazione, porte, checksum, TTL

Estensioni

- Inviare un pacchetto TCP con flag personalizzati
- Visualizzare e spiegare le differenze tra ICMP e TCP
- Generare un semplice report HTML o PDF con gli screen e analisi

Output atteso

- Script Python con Scapy (`/src/script.py`)
- File .pcap della cattura (presenti nella cartella `/wireshark`)
- Relazione con screenshot e commenti tecnici

Capitolo 2

Pacchetti ICMP Echo

2.1 Generazione dei pacchetti con Scapy

Si vuole inviare un pacchetto ICMP Echo Request (ping) a un host. Per l'invio ho utilizzato il metodo `.sr()` della libreria `scapy`, specificando il tipo ICMP. Riporto nel Listing 2.1 la funzione che ho scritto per poter generare il pacchetto del ping, che richiede come parametro l'indirizzo IP oppure l'hostname del destinatario. La funzione stampa anche il risultato dell'operazione.

```
1 import scapy.all as scapy
2
3 def send_ping(destination):
4     print("----- INVIO PING A", destination, " -----")
5     res = scapy.sr(scapy.IP(dst=destination)/scapy.ICMP(), timeout=4)
6     print("--- RISULTATI: ---")
7     for r in res:
8         r.show()
9     print()
```

Listing 2.1: Funzione python per la generazione di un pacchetto ICMP Echo Request

2.2 Analisi del traffico con Wireshark

Ho mandato il ping a `google.com`, che è stato risolto nell'indirizzo IP `216.58.204.238`. Per la cattura del traffico di rete ho utilizzato `wireshark` con il filtro `host 216.58.204.238` per catturare solo i pacchetti da / per l'host indicato. Nel file `wireshark/icmp_echo.pcap` è presente il risultato della cattura.

Riporto in seguito il dettaglio della richiesta e della risposta.

2.2.1 ICMP Echo Request

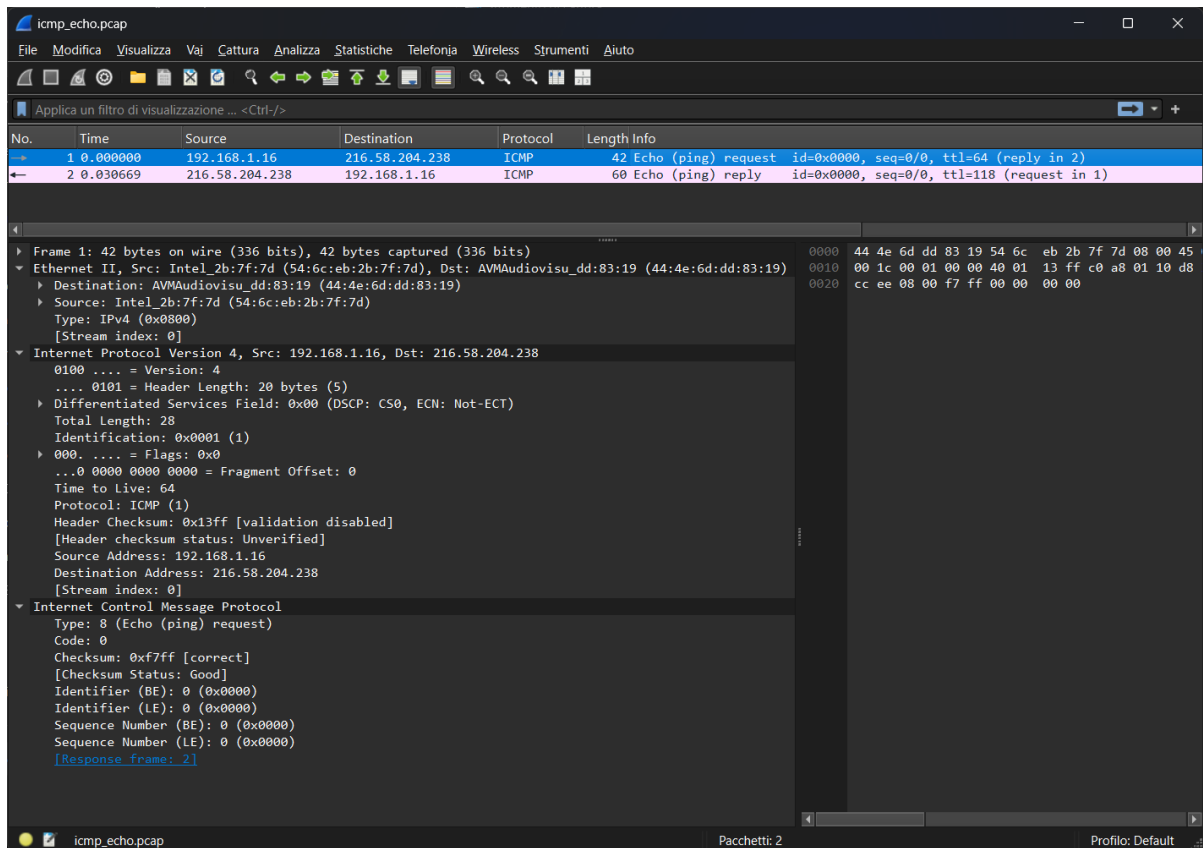


Figura 2.1: Dettaglio Wireshark dell'Echo Request

Il pacchetto parte dal mio PC connesso alla rete di casa, infatti nella PCI del livello IP troviamo il campo **Source Address**: 192.168.1.16, ovvero l'IP privato del mio PC nella mia rete. Nel campo **Destination Address**: 216.58.204.238 troviamo l'IP del destinatario della richiesta, ovvero l'host verso cui ho inviato il pacchetto ping. Il campo **Time To Live** è settato al livello massimo (64) visto che il pacchetto è stato catturato non appena generato.

Nella parte dati del pacchetto IP viene trasportato il pacchetto ICMP: possiamo confermarlo leggendo il campo **Protocol**: ICMP nell'header del pacchetto IP. All'interno della parte riservata al protocollo ICMP viene specificato che il pacchetto è di tipo **Echo Request**.

Notiamo che non sono presenti riferimenti a nessuna porta: questo perché IP e ICMP sono protocolli dell'Internet Layer della suite TCP/IP, quindi non c'è la necessità di comunicare tramite una porta con un livello superiore (ovvero con un protocollo di trasporto).

Risulta infine evidente che sia il pacchetto IP sia quello ICMP sono dotati di un proprio checksum, che nel caso dell'IP non viene verificato mentre nell'ICMP è corretto.

2.2.2 ICMP Echo Reply

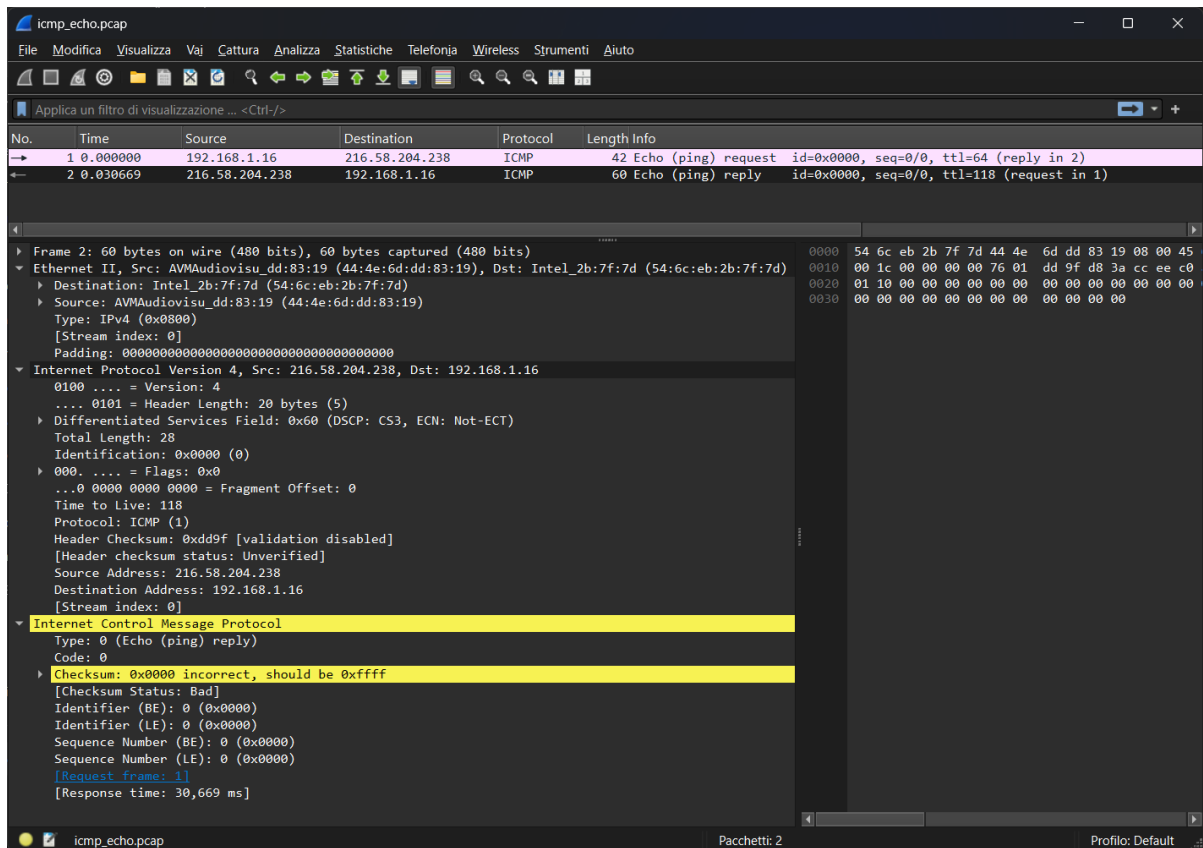


Figura 2.2: Dettaglio Wireshark dell'Echo Reply

Questa cattura presenta varie differenze rispetto a quella precedente. Quella più ovvia è l'inversione degli indirizzi IP nei campi **Destination Address**: 192.168.1.16 e **Source Address**: 216.58.204.238, visto che si tratta del pacchetto di risposta (quindi generato dal destinatario del ping).

Poiché si tratta della risposta al ping iniziale, IP ci dice che il protocollo utilizzato è sempre **Protocol**: ICMP, mentre nella parte riservata al protocollo ICMP viene specificato che si tratta di una **Echo Reply**.

In questo caso il campo **TTL** è settato a 118: possiamo presupporre che alla generazione del pacchetto sia stato 128 (la potenza del 2 più vicina a 118), ma è stato decrementato di un'unità per ogni router che il pacchetto ha attraversato. Utilizzando il comando `tracert 216.58.204.238` ho poi verificato che per raggiungere il destinatario dal mio host sono necessari 10 salti.

Come prima il checksum dell'header IP è presente ma non verificato. Nel caso di ICMP invece, si ha che il checksum calcolato è diverso da quello riportato nel pacchetto. Ho provato a ripetere varie volte il ping, ottenendo sempre lo stesso risultato: potrebbe quindi essere che l'host destinatario sceglie di non calcolare il checksum lasciandolo a 0. Il contenuto informativo del pacchetto sembra comunque essere corretto nei campi che sono di nostri interesse.

Capitolo 3

Segmenti TCP SYN

3.1 Generazione dei pacchetti con Scapy

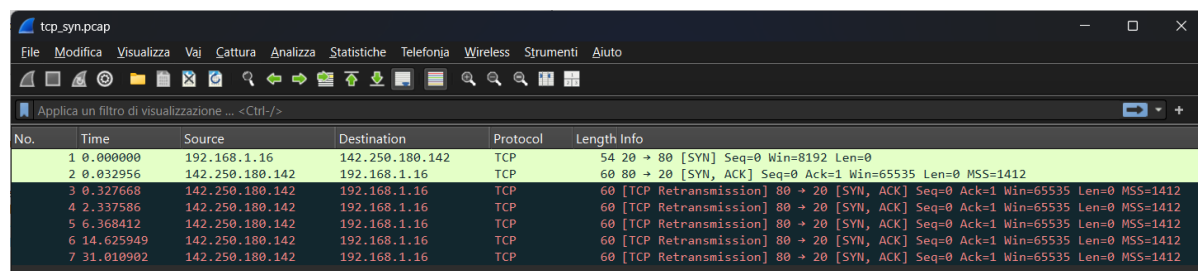
Si vuole inviare un segmento TCP con il flag **SYN** a un host: ciò corrisponde alla richiesta di instaurazione di una connessione.

Come nel capitolo precedente, ho utilizzato il metodo `.sr()` della libreria `scapy`, specificando il tipo TCP, con il flag **S** attivo (ovvero SYN). Riporto nel Listing 3.1 la funzione che ho scritto per poter generare il pacchetto contenente il segmento TCP, che richiede come parametro l'indirizzo IP (o l'hostname) e la porta del destinatario. La funzione stampa anche il risultato dell'operazione.

```
1 import scapy.all as scapy
2
3 def send_tcp_syn(destination, port):
4     print("----- INVIO SYN A", destination, ", PORTA", port, "-----")
5     res = scapy.sr(scapy.IP(dst=destination)/scapy.TCP(dport=port, flags
6     = "S"), timeout=4)
7     print("--- RISULTATI: ---")
8     for r in res:
9         r.show()
10    print()
```

Listing 3.1: Funzione python per la generazione di un segmento TCP con flag **SYN** attivo

3.2 Analisi del traffico con Wireshark



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.16	142.250.180.142	TCP	54	20 → 80 [SYN] Seq=0 Win=8192 Len=0
2	0.032956	142.250.180.142	192.168.1.16	TCP	60	80 → 20 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412
3	0.327668	142.250.180.142	192.168.1.16	TCP	60	[TCP Retransmission] 80 → 20 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412
4	2.337586	142.250.180.142	192.168.1.16	TCP	60	[TCP Retransmission] 80 → 20 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412
5	6.368412	142.250.180.142	192.168.1.16	TCP	60	[TCP Retransmission] 80 → 20 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412
6	14.625949	142.250.180.142	192.168.1.16	TCP	60	[TCP Retransmission] 80 → 20 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412
7	31.010902	142.250.180.142	192.168.1.16	TCP	60	[TCP Retransmission] 80 → 20 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412

Figura 3.1: Pacchetti catturati da Wireshark

Visto che un TCP SYN dà il via al three way handshake, per considerare la connessione instaurata è necessario che l'host da cui parte la richiesta confermi con un ACK finale.

Nel nostro caso ciò non avviene, infatti notiamo che il server ritrasmette periodicamente il suo SYN ACK (vedi Figura 3.1), credendo che quelli precedenti siano stati perduti durante la trasmissione.

Il file della cattura contenente i pacchetti sniffati su cui sono stati effettuati gli screenshot, è il file `/wireshark/tcp_syn.pcap`.

3.2.1 Richiesta TCP con SYN attivo

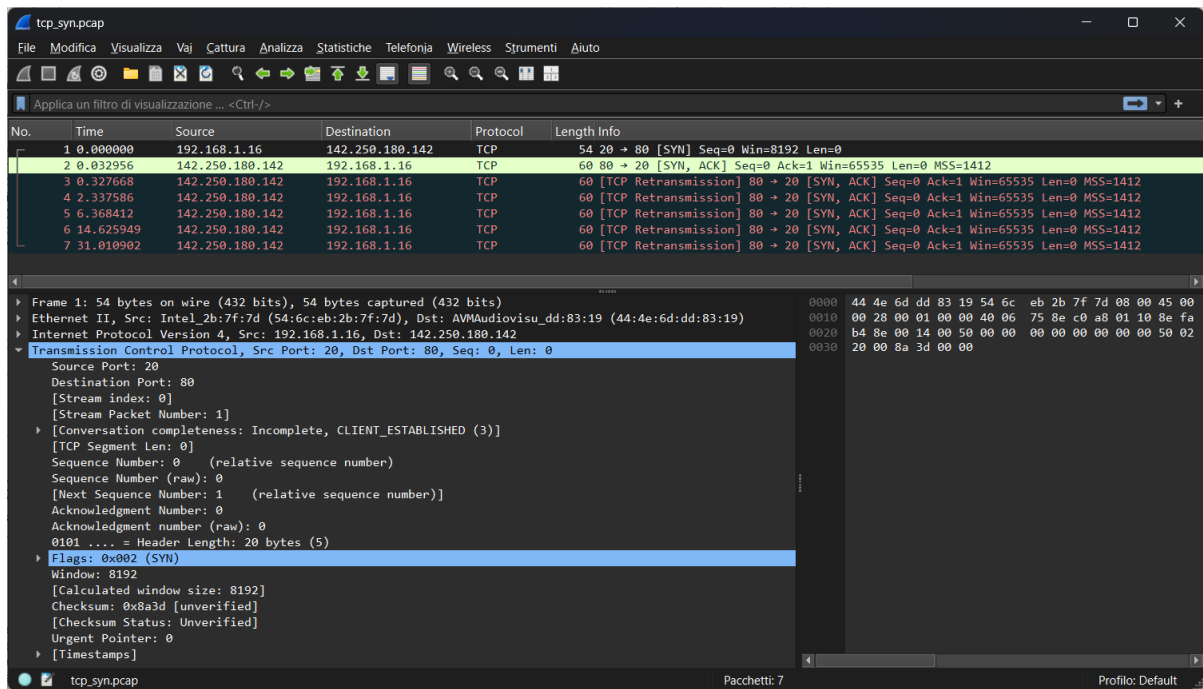


Figura 3.2: Dettaglio del pacchetto trasmesso, con flag SYN attivo

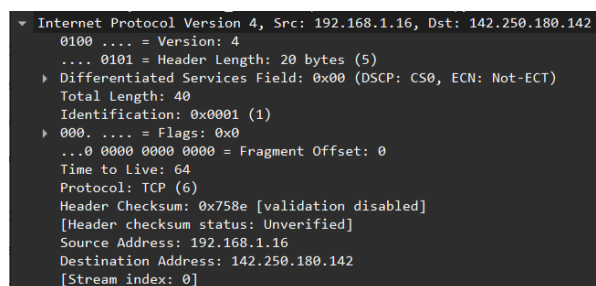


Figura 3.3: Dettaglio dell'header IP

Controlliamo innanzitutto l'header del protocollo IP (vedi Figura 3.3). Come nell'analisi precedente (vedi sezione 2.2.1) il **Source Address** e il **Time To Live** sono rimasti invariati; è rimasta disabilitata anche la validazione dell'header. Il **Destination Address** contiene un indirizzo IP diverso rispetto al precedente, nonostante io abbia mandato il pacchetto sempre a `google.com`: questo è successo perché il server DNS ha ritenuto opportuno risolvere lo stesso nome di dominio con un altro indirizzo, probabilmente per poter distribuire meglio il carico sui vari server. Grazie al campo **Protocol: TCP** notiamo che nel payload del pacchetto viene trasportato un segmento del protocollo TCP,

che andremo ad analizzare (il cui contenuto viene riportato in Figura 3.2.)

Per essere sicuro di comunicare con un server TCP ho scelto come **Destination Port: 80**: questo perché 80 è il numero di porta well-known attribuito ai server HTTP, e l'HTTP è un protocollo applicativo che utilizza il protocollo TCP a livello di trasporto. Il campo **Source Port: 20** ci dice che scapy ha scelto di utilizzare la porta 20 TCP del mio PC per la comunicazione: probabilmente viene scelta di default quando non si specifica una porta sorgente. Questo non dovrebbe però accadere visto che la porta 20 corrisponde alla well-known port per il trasferimento dei dati del protocollo applicativo FTP; avrebbe invece dovuto utilizzare una porta libera (nel range 49152 - 65535) assegnata dal sistema operativo, visto che i client quando instaurano una connessione non hanno bisogno di utilizzare una porta nota. Anche in questo caso il checksum riportato non viene verificato.

3.2.2 Risposta TCP con SYN ACK attivi

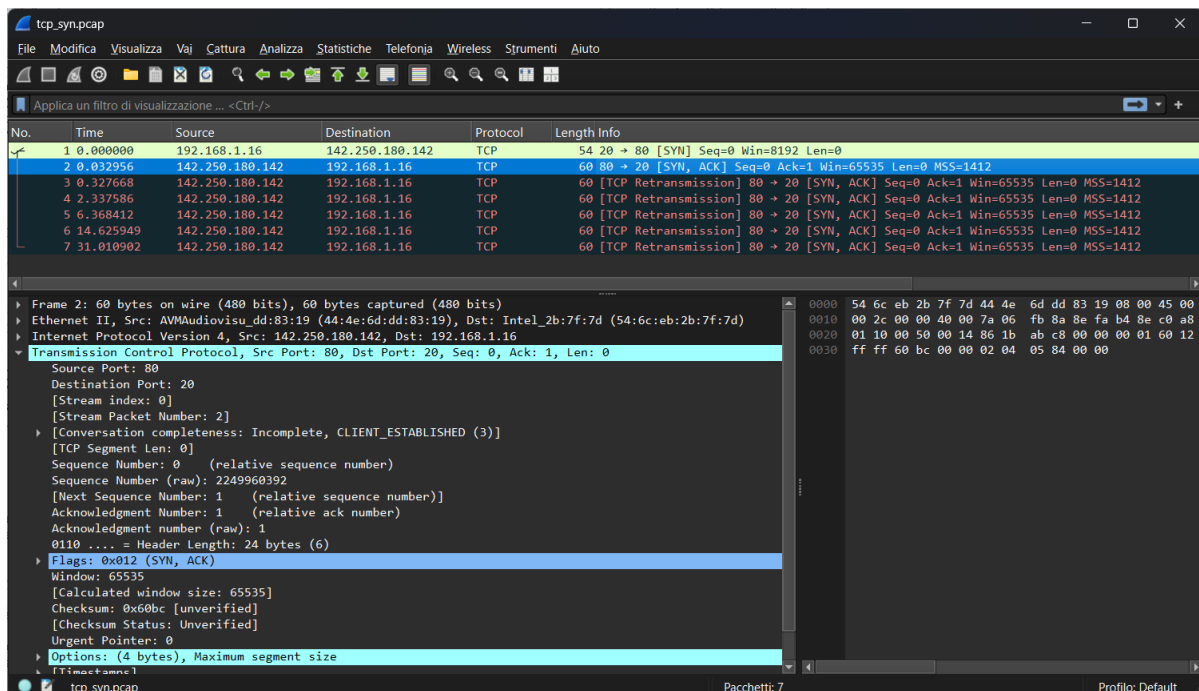


Figura 3.4: Dettaglio del pacchetto ricevuto con flag SYN e ACK attivi

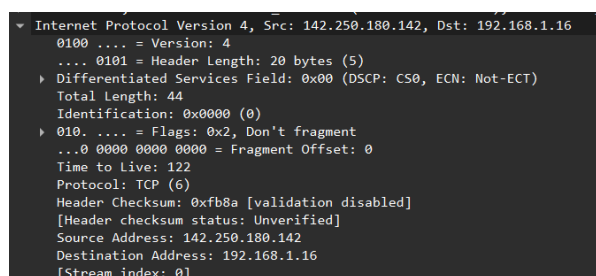


Figura 3.5: Dettaglio dell'header IP

Visto che si tratta del pacchetto di risposta alla mia richiesta di connessione (Figura 3.5), gli indirizzi IP riportati sono gli stessi ma scambiati tra mittente e destinatario.

Notiamo sempre che il TTL è stato decrementato di qualche unità, che il protocollo della PDU trasportata nei dati del pacchetto IP è il protocollo TCP e che il checksum non è stato verificato.

Passando all'analisi del protocollo TCP (Figura 3.4), trattandosi della risposta, anche qua le porte sorgente e destinatario sono scambiate. Poiché questo segmento è il secondo step del three way handshake, vediamo che sono attivi i flag **SYN** e **ACK**: il server ci sta confermando tramite l'ACK di aver ricevuto la nostra richiesta di connessione, e contemporaneamente chiede al mio host di aprire la connessione inviando il SYN. L'ACK number è impostato a 1 per confermare al mio PC di aver ricevuto correttamente tutti i segmenti con numero di sequenza fino allo 0, quindi il prossimo che si aspetta è il segmento con numero di sequenza a 1. Il numero di sequenza del server invece parte da 2249960392, ovvero un numero generato in maniera casuale per evitare di avere segmenti dispersi per la rete con lo stesso sequence number che potrebbero essere ricevuti quando non hanno più nessun senso, dando origine a vari errori.

3.2.3 Ritrasmissioni TCP con SYN ACK attivi

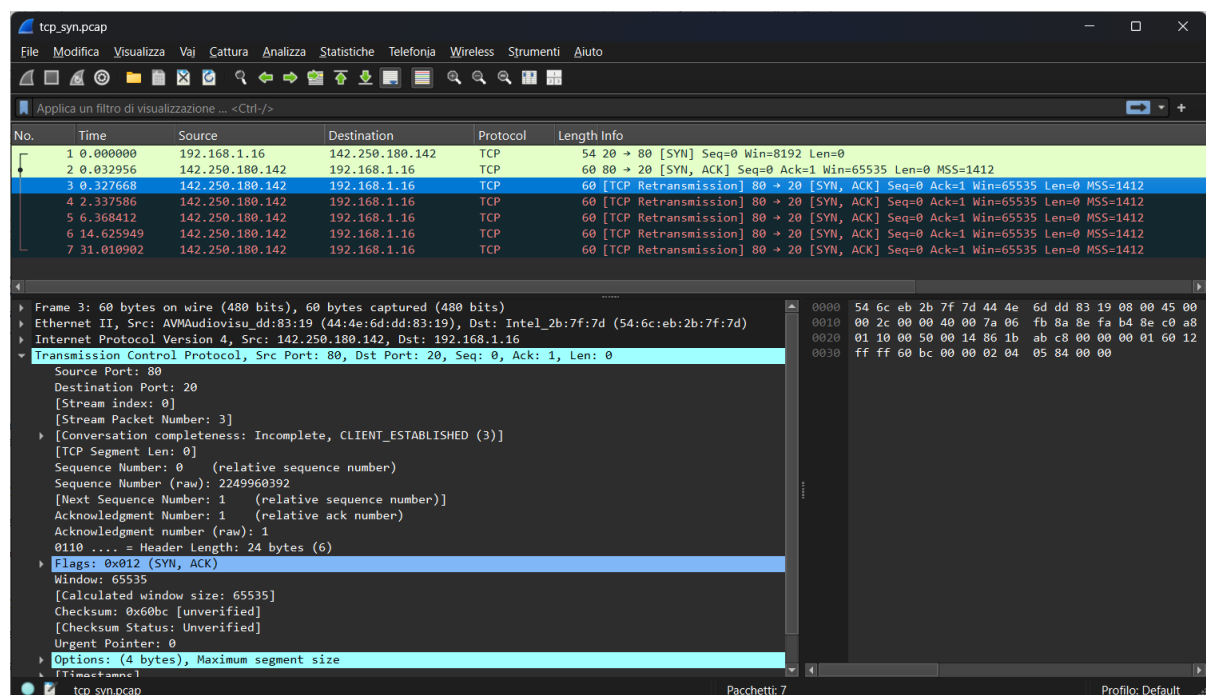


Figura 3.6: Dettaglio del pacchetto ricevuto in seguito alla ritrasmissione, con flag SYN e ACK attivi

Il codice python contenuto nel Listing 3.1 invia un TCP SYN e riceve l'eventuale risposta, senza però effettuare nient'altro. Il three way handshake si considera concluso (e quindi la connessione è stabilita) solo dopo la conferma finale data dall'host richiedente all'host destinatario con un segmento con flag ACK attivo, segmento che il mio host non invia al server. Per questo motivo il server crede che il suo SYN ACK non sia mai arrivato a destinazione, causando così la ritrasmissione dello stesso (notiamo infatti che tutto il contenuto del segmento è uguale a quello precedente, vedi Figura 3.4) allo scadere del tempo di timeout.

Capitolo 4

Segmenti TCP con flag personalizzati

4.1 Generazione dei pacchetti con Scapy

La generazione di segmenti TCP attivando flag personalizzati è resa molto semplice dalla libreria scapy: è sufficiente creare una stringa contenente un carattere per ogni flag scelto, come riportato nel Listing 4.1. Ad esempio per attivare i flag FIN e ACK bisogna settare `flags="FA"`.

Nella funzione che ho scritto, lascio scegliere all'utente quali flag vuole abilitare.

```
1 import scapy.all as scapy
2
3 def send_tcp_flags(destination, port):
4     print("----- FLAG DISPONIBILI: -----")
5     print("F - FIN")
6     print("S - SYN")
7     print("R - RST")
8     print("P - PSH")
9     print("A - ACK")
10    print("U - URG")
11    print("E - ECE")
12    print("C - CWR")
13    print("\nDigitare i caratteri dei flag da attivare (ad esempio \"SR\"
14    \" per i flag SYN e RST)\n>", end=" ")
15    flgs = input().upper()
16    print("----- INVIO FLAG", flgs, "A", destination, ", PORTA", port,
17    "-----")
18    res = scapy.sr(scapy.IP(dst=destination)/scapy.TCP(dport=port, flags
19    =flgs), timeout=4)
20    print("--- RISULTATI: ---")
21    for r in res:
22        r.show()
23    print()
```

Listing 4.1: Funzione python per la generazione di un segmento TCP con flag attivabili a scelta

4.2 Analisi del traffico con Wireshark

4.2.1 Tutti i flag attivi

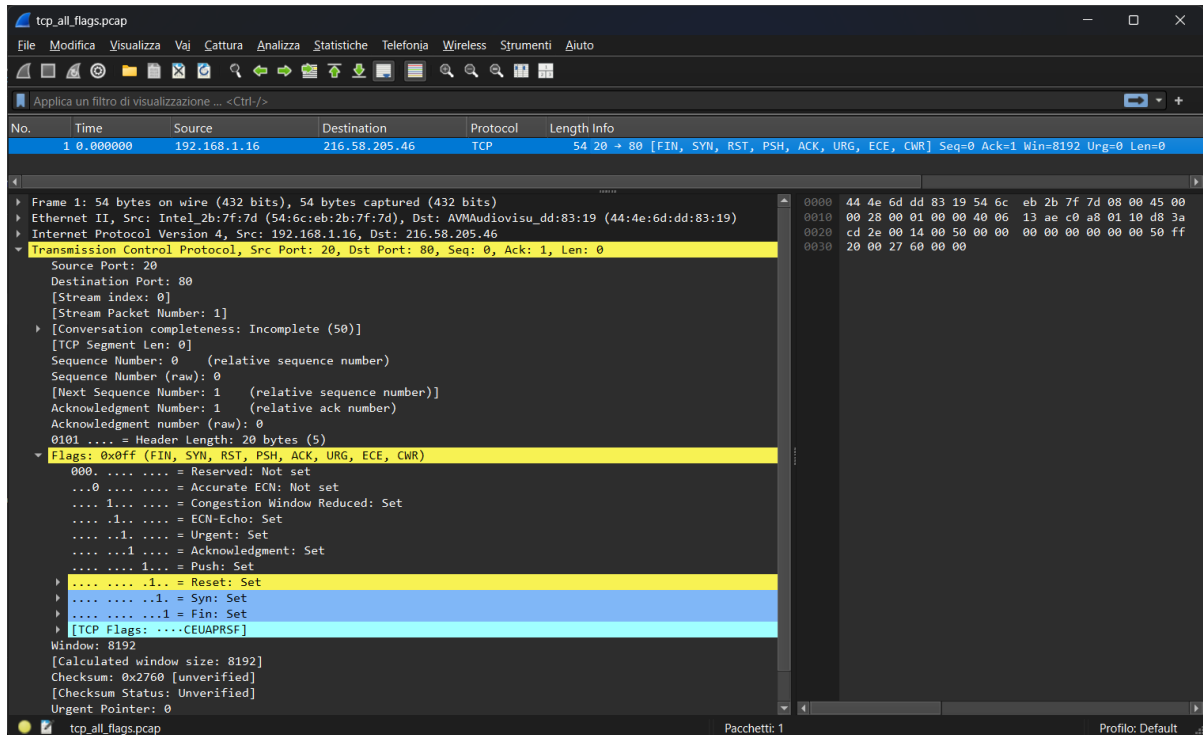


Figura 4.1: Dettaglio del pacchetto inviato con tutti i flag attivi

Tramite la funzione sopra descritta, ho inviato un segmento TCP al server HTTP `google.com:80` abilitando tutti i flag possibili. Questa operazione non ha ovviamente alcun senso, infatti dalla cattura si vede che non ho ricevuto nessuna risposta dal server. Il pacchetto è salvato nel file `/wireshark/tcp_all_flags.pcap`.

4.2.2 Three-way handshake

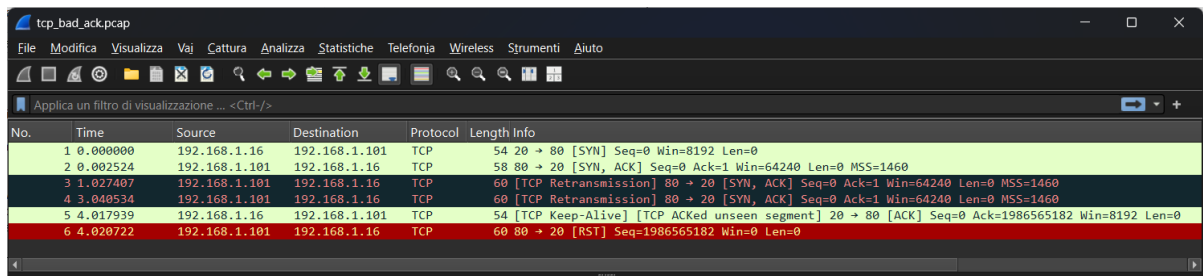


Figura 4.2: Sequenza dei pacchetti inviati nel tentativo di simulare un three-way handshake

In questa prova ho cercato di simulare un three-way handshake con un mio server HTTP: riporto punto per punto la spiegazione di ogni pacchetto riferendomi al numero progressivo riportato dalla cattura di wireshark.

1. Ho generato un segmento con il flag SYN attivo, per far partire il three-way handshake
2. Il server conferma la sua disponibilità ad aprire la connessione inviando un segmento con SYN e ACK attivi
3. Poiché stavo generando i pacchetti a mano con lo script, ho impiegato un po' di tempo per generare l'ACK di risposta, causando la scadenza del tempo di timeout del server (ovvero credeva che il suo pacchetto SYN ACK si fosse perso per la rete)
4. Come il punto 3
5. Questo segmento contiene l'ACK che ho generato, che avrebbe dovuto terminare il three-way handshake confermando l'apertura della connessione
6. Visto che il pacchetto generato al punto 5 contiene il flag di ACK ma ha sequence number e ack number errati, il server risponde con un segmento contenente il flag RST per resettare la connessione poiché si sono verificati questi errori nella numerazione dei segmenti

Il risultato della cattura di whireshark è salvato nel file `/wireshark/tcp_bad_ack.pcap`

Capitolo 5

Commenti finali

5.1 Differenze tra ICMP e TCP

ICMP e TCP sono due protocolli completamente differenti: il primo appartiene al **livello 3** del modello ISO/OSI (livello rete del TCP/IP), il secondo al **livello 4** ISO/OSI (livello di trasporto del TCP/IP). Mentre ICMP è utilizzato per ottenere informazioni diagnostiche dalla rete (abbiamo visto che possiamo eseguire il ping verso un host per verificarne la raggiungibilità), TCP è utilizzato per garantire affidabilità (controllo di flusso, rilevazione di errore, controllo della congestione, ecc...) instaurando una connessione basata su un livello per sua natura inaffidabile, ovvero il livello rete. Abbiamo infine notato che nel segmento TCP abbiamo indicate le porte del mittente e del destinatario, mentre nei pacchetti ICMP no: questo è dovuto al fatto che TCP è un protocollo di livello superiore rispetto ad IP, quindi le porte sono le SAP (Service Access Point) che mettono in comunicazione il livello 3 con il livello 4.