Multimedia

Homework 2

Alessandro Trigolo

7 Giugno 2024

Indice

Elenco delle figure

Todo list

1 Introduzione

Fai introduzione

1.1 Utilizzo dello script

Le richieste dell'homework sono state soddisfatte attraverso uno script scritto con il linguaggio *Python* ed eseguito sul sistema operativo *Windows 10* in lingua italiana. Tutte le richieste sono state soddisfatte attraverso l'utilizzo del comando `ping` che si occupa di inviare delle richieste ICMP[1] ad una determinata destinazione. La risposta del suddetto comando diagnostico è il periodo impiegato dalla destinazione per rispondere, ovvero la **latenza**.

Prima di lanciare lo script, è necessario aprire il file *script.py* e impostare correttamente il percorso dello script modificando la costante `PATH_TO_SCRIPT`.

```
PATH_TO_SCRIPT = os.path.join("multimedia", "hw-2", "script")
```

In secondo luogo è possibile impostare le specifiche di progetto mediante delle costanti che indicano il nome del server su cui inviare le richieste, il numero di instanze da mandare ad ogni richiesta ping e l'incremento di lunghezza (step) che ci sarà tra la lunghezza del pacchetto durante l'iterazione i e la successiva $i + 1$.

```
# project specification
SELECTED_SERVER_CITY = "Atlanta"
INSTANCES = 30
STEP_BETWEEN_LENGTHS = 25
```

Tali impostazioni verranno poi brevemente processate con il seguente codice. In particolare si osserva che la costante `SELECTED_SERVER_CITY` verrà utilizzata per ottenere l'effettivo dominio del server e l'abbreviazione del nome della città. In secondo luogo la costante `STEP_BETWEEN_LENGTHS` verrà utilizzata per generare un vettore che parta da 10 e per arrivare a 1471 incrementa il suo valore di esattamente `STEP_BETWEEN_LENGTHS`.

```
SERVERS = {
    "Atlanta" : "atl.speedtest.clouvider.net",
    "New York City" : "nyc.speedtest.clouvider.net",
    "London" : "lon.speedtest.clouvider.net",
    "Los Angeles" : "la.speedtest.clouvider.net",
    "Paris" : "paris.testdebit.info",
    "Lillie" : "lille.testdebit.info",
    "Lyon" : "lyon.testdebit.info",
    "Aix-Marseille" : "aix-marseille.testdebit.info",
    "Bordeaux" : "bordeaux.testdebit.info"
}

# set choosen options
server = SERVERS[SELECTED_SERVER_CITY]
city = server.split(".")[0]
```

[1] ICMP sta per *Internet Control Message Protocol* è un protocollo utilizzato dai dispositivi di una rete per comunicare la presenza di problemi riguardanti la trasmissione dei dati

```
16 |     payload_lengths = range(10, 1471, STEP_BETWEEN_LENGTHS)
```

Usando le impostazioni dell'esempio, lo script — che sarà descritto nel capitolo 2 — manderà le richieste al server di Atlanta `atl.speedtest.clouvider.net`. Lo script poi, al fine di analizzare le performance di rete di tale server, per ogni valore di lunghezza all'interno della lista `payload_lengths`, effettureà 30 istanze di richiesta. In altre parole si faranno 30 richieste di lunghezza 10 byte al server di Atlanta, poi altre 30 di lunghezza $10 + 25 = 35$ byte, poi altre 30 di lunghezza $35 + 25 = 60$ bytes e così via fino ad arrivare alla lunghezza 1470 bytes, per un totale di 1770 richieste.

1.2 Parametri di progetto

All'interno dello script vegono utilizzati due tipi di comando `ping`. Il primo, utilizzato nella prima task, descritta nel paragrafo 2.1, è il comando ping di default di Windows: lo script funziona solo e soltanto se la lingua di Windows è in italiano altrimenti non è possibile analizzare correttamente le risposte del comando. Nella seconda task, esplorata nel paragrafo 2.2, viene utilizzato invece il comando `psping`, che è in lingua inglese. Al fine di riuscre ad utilizzare lo script è necessari avere l'eseguibile `psping.exe` all'interno della cartella da cui poi viene eseguito lo script (ovvero la cartella che, attraverso il percorso all'interno di `PATH_TO_SCRIPT`, permette di arrivare al file `script.py`). Si è scelto di utilizzare psping in quanto è più prestante del comando ping di default. Inoltre nel primo punto non si è utilizzato psping in quanto non è presente l'impostazione che permette di ridurre il Time-To-Live del pacchetto.

In secondo luogo, al fine di eseguire i comandi nel terminale, non si è utilizzata la funzione `os.system`, bensì la libreria `subprocess`, che permette di ottenere il risultato direttemane in una stringa senza avere la necessità di scriverlo sul file per poi accedere al file per leggerlo. Questo migliora le prestazioni computazionali dello script, in particolare nella seconda parte dove in genere viene inviata una mole elevata di pacchetti ICMP. È comunque possible scrivere su files i risultati dell'esecuziuone del comando, è sufficiente impostare a `True` la *flag* che si chiama `SAVE_TO_FILE`.

Specifica dettagliatamente i parametri utilizzati

- Los Angeles
- K = 250
- Dim = 10 → 1470, step di 1

2 Performance di rete

Dopo aver introdotto i parametri di progetto adottati, andremo ora ad analizzare le perfomrance di rete. In particolare ci occupiamo di analizzare il numero di connessione che debbono essere attraversate al fine di raggiungere il server a Los Angeles `la.speedtest.clouvider.net`. In secondo luogo, attraverso una massiccia fase di

richieste ICMP al server analizzeremo i tempi di latenza, estraendone i minimi, i massimi, la media e la devizione standard. Infine utilizzeremo i dati raccolti per stimare ed analizzare il throughput verso il server scelto.

2.1 Numero di link attraversati

Il primo valore che andremo a calcolare è il numero di connessioni — dette anche *links* — che ci separa dal server di Los Angeles. In particolare ci occuperemo di recuperare tale valore in due modi diversi. Il primo modo per farlo è tramite l'utilizzo del comando Windows `tracert` che permette di tracciare interamente il percorso che un pacchetto IP[2] compie al fine di raggiungere la destinazione.Il conamdo `tracert` si basa su una serie contigua di ping che sono inviati impostando il Time-To-Live a 1 e incrementandolo man mano. In questo modo, nel momento in cui il comando ping non scade, il TTL dello specifico messaggio ICMP corrisponde al numero di link che separa il sorgente dal destinatario. Invocando il comando `tarcert` sul server di Los Angeles `la.speedtest.clouvider.net`, si ottiene sul terminale la lista numerata di tutti i link attraversati, come mostrato nel frammento di codice seguente.

```
Traccia instradamento verso la.speedtest.clouvider.net [77.247.126.223]
su un massimo di 30 punti di passaggio:

1    <1 ms    <1 ms    <1 ms CLOUD-STORAGE [192.168.1.1]
2    36 ms    17 ms     8 ms 151.6.141.50
3    11 ms     7 ms     8 ms 151.6.141.34
4    14 ms    14 ms    14 ms 151.6.0.68
5    13 ms    15 ms    12 ms 151.6.1.182
6    15 ms    14 ms    14 ms mno-b3-link.ip.twelve99.net [62.115.36.84]
7    30 ms    29 ms    29 ms prs-bb1-link.ip.twelve99.net [62.115.135.224]
8   113 ms   113 ms   127 ms ash-bb2-link.ip.twelve99.net [62.115.112.242]
9   172 ms   172 ms   171 ms lax-b22-link.ip.twelve99.net [62.115.121.220]
10  173 ms   173 ms   173 ms clouvider-ic-355873.ip.twelve99-cust.net
    [213.248.74.63]
11  176 ms     *      203 ms 77.247.126.1
12  172 ms   171 ms   172 ms 77.247.126.223

Traccia completata.
```

Tale risultato si traduce nel seguente codice Python dove, dopo aver memorizzato il risultato del comando nella stringa `result`, quest'ultima viene suddivisa in righe e viene analizzata in modo tale da restituire l'ultimo numero del link, che coincide con il numero totale di connessioni attraversate per raggiungere il server. Mediante tale funzione si ottiene che il numero di links necessari per raggiungere il server di Los Angeles corrisponde a 12.

```
1    def get_links_from_tracert(server: str) -> int:
2
3        # get number of links from tracert
4        cmd = f"tracert {server}"
```

[2]IP è l'acronimo di *Internet Protocol*

```
        result = subprocess.run(cmd, shell=True,
            stdout=subprocess.PIPE, stderr=subprocess.PIPE,
            text=True)

        last_link_line = result.stdout.split("\n")[-4]

        return int(last_link_line.split(" ")[1])
```

Per contare il numero di connessioni, oltre all'utilizzo del comando `tracert`, è possibile impiegare direttamente il comando `ping`. In particolare, è sufficiente iterare attraverso una serie di ping, diminuendo ad ogni iterazione il parametro TTL (Time-To-Live), che rappresenta il numero massimo di link che il pacchetto può attraversare prima di essere eliminato. La seguente funzione in Python itera sui valori di TTL da 20 a 0. Ad ogni iterazione viene inviata una richiesta ping predefinita, con 4 pacchetti di dimensione di 32 byte. Quando il ping non ottiene una risposta, significa che il valore di TTL è diminuito al punto da non essere più sufficiente per raggiungere la destinazione finale. Pertanto, viene restituito il valore di TTL incrementato di 1, che è il minimo TTL necessario per raggiungere il server. Di fatto questo appriccio è l'esatto opposto dell'approccio che adotta `tarcert`: in questo caso infatti al posto di incrementare il TTL fino a che non si riceve una risposta, lo si decrementa fino a che il messaggio ICMP scade, non riuscendo quindi a raggiungere la destinazione.

```
    def get_links_from_ping(server: str) -> int:

        for ttl in range(20, 0, -1):

            cmd = f"ping {server} -i {ttl}"
            result = subprocess.run(cmd, shell=True,
                stdout=subprocess.PIPE, stderr=subprocess.PIPE,
                text=True)

            if "TTL scaduto durante il passaggio" in result.stdout:
                return (ttl + 1)
```

In tutti e due i casi, come risultato, il numero di connessioni che separa il sorgente dal server di Los Angeles `la.speedtest.clouvider.net` è 12.

2.2 Analisi del *Round Trip Time*

La seconda parte, volta ad analizzare il *RTT*, necessita uno script Python più complesso, anche volto a migliorare le prestazioni computazionali dello script. Si è quindi scelto di utilizzare un approccio **multi-threading** ed il comando `psping`, che è più prestante rispetto al comnue ping, inoltre permetto di ottenere delle latenze più precise (fino al centesimo di millisecondo).

Per permettere l'utilizzo di diversi thread, è necessario racchiudere le richieste ping in una funzione, chiamata `ping_server`. In questa funzione la richiesta di ping ha diverse proprietà specificate:

- `-n`, che specifica il numero di istaze per ogni richiesta ping;
- `-l`, indica la lunghezza del *payload* di ciascuna richiesta;

- ◇ `-i`, specifica l'intervallo di secondo tra un ping e l'altro; viene impostato a zero per aumentare la rapidità dello script;
- ◇ `-w`, che indica il numero di *warmup request* da fare prima di iniziare la sequenza di ping.

Dopo aver eseguito il comando, il risultato viene analizzato e ne sono estratte le latenze, che sono aggiunte in una lista (o vettore). Infine sono ritornati due valori, la lunghezza corrente e la lista con tutte le latenze.

```
    def ping_server(server, instances, length):

        cmd = f"psping -n {instances} -l {length} -i 0 -w 0
            {server}"
        result = subprocess.run(cmd, shell=True,
            stdout=subprocess.PIPE, stderr=subprocess.PIPE,
            text=True)

        millisecs_vector = []
        lines = result.stdout.split("\n")

        for line in lines:
            if "Reply from" in line:
                duration = float(line.split(": ")[1].split("ms")[0])
                millisecs_vector.append(duration)

        return length, millisecs_vector
```

La funzione descritta sopra viene utilizzata nel seguente script dove, mediante il pacchetto `ThreadPoolExecutor`, viene eseguita da ciascun thread. In questo modo molte più richieste sono mandate contemporaneamente migliorando notevolmente le prestazioni del programma. Senza l'utilizzo di un algoritmo multithreading, ottenere tempistiche di esecuzione superiori a una o due ore era inevitabile, anche con un numero limitato di istanze (ad esempio, 50 istanze con incrementi di 75). Infine, la coppia restituita dalla funzione `ping_server`, viene aggiunta al dizionario `stats` dove sono immagazzinate tutte latenze per poi essere analizzate.

```
    stats = {}

    # using ThreadPoolExecutor to imporve computational capabilites
    with concurrent.futures.ThreadPoolExecutor(max_workers=100) as
        executor:

        futures = []
        for length in payload_lengths:
            # submit a task to the executor to ping the server with the
                specified parameters
            future = executor.submit(ping_server, server,
                INSTANCES, length)
            futures.append(future)

        # iterate over completed futures as they become available
```

```
        for future in concurrent.futures.as_completed(futures):
            length, millisecs_vector = future.result()
            stats[length] = millisecs_vector

    # order stats by length
    stats = dict(sorted(stats.items()))
```

Al fine di calcolare tutti i massimi, i minimi, le medie e le varianze è sufficiente iterare lungo il dizionario creato. Lo seguente script si occupa proprio di questo, anche se in modo poco efficiente ma perlomeno leggibile.

```
    max_values = {}
    min_values = {}
    average_values = {}
    standard_deviations = {}

    for key, value in stats.items():
        max_values[key] = max(value)
        min_values[key] = min(value)
        average_values[key] = sum(value) / len(value)
        standard_deviations[key] = math.sqrt(sum((x -
            average_values[key]) ** 2 for x in value) / len(value))
```

Risultati

Dopo aver eseguito gli script, è quindi possibile mostrare tutte le latenze raccolte, i loro massimi, i minimi, le latenze medie per ciascuna lunghezza e la loro varianza. Sono presentati di seguito tre immagini contenenti i risultati delle richieste; cioascune delle tre immagini fa riferimento ad una diversa scelta di istanze e di step.

L'immagine 1 di seguito rappresenta tutte le latenza raccolte durante l'esecuzione dei ping con 20 istanze per richiesta e con uno step di 10 tra una lunghezza e l'altra, per un totale di 2940 ping effettuati in circa un minuti. Si può osservare che in genere tutte le latenze sono attorno ai 170 millisecondi, con alcuni picchi tra i 750 e gli 800 bytes. Non a caso, nei grafici rappresentati in figura 2, si osserva infatti che i picchi sono rispecchiati nella rappresentazione dei massimi (in alto a destra), delle media (in basso a sinistra) e delle varianze (in basso a destra). Si osserva inoltre che i minimi variano tra 168 e 170 millisecondi e che crescono al crescere della dimensione del pacchetto; questo aspetto sarà dettagliatamente esplorato nel prossimo paragrafo, dedicato al *throughput* (2.3).

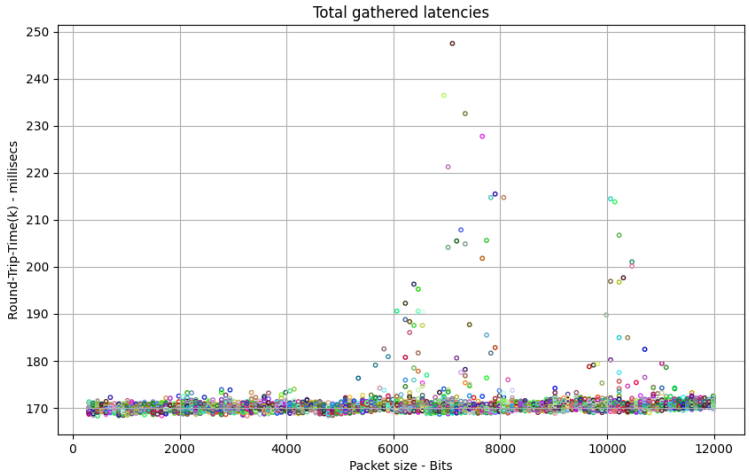

Figura 1: Tutte le ottenute con 20 istanze ed uno step di 10.

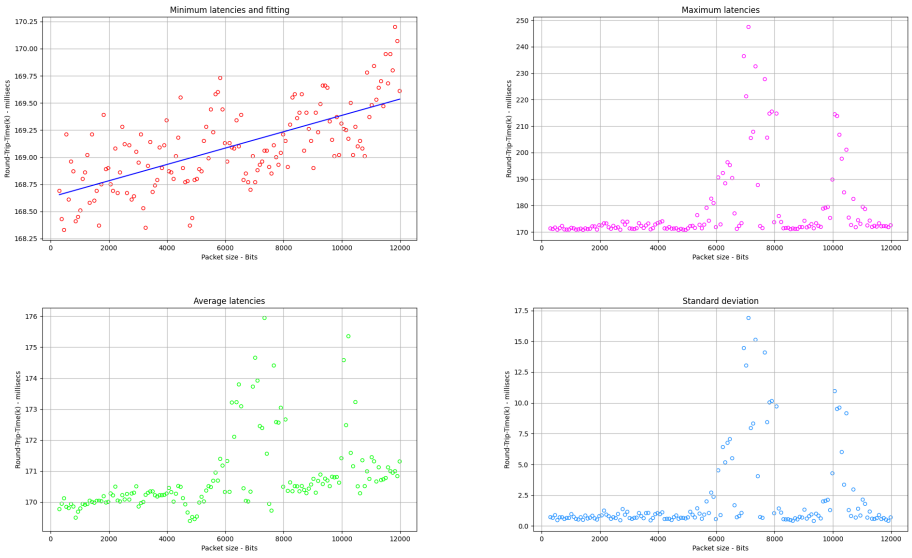

Figura 2: Grafici in riferimento all'immagine 1.

Aumentando il numero di ping effettuati, in particolare aumentonado il numero di istanze a 100 e riducendo l'incremento delle lunghezze a 2, otteniamo un grafico come quello della figura 3, che contiene 73100 valori di latenza, ottenuti in circa 6 minuti. Anche in questo caso possiamo notare dei picchi che sono distribuiti lungo tutte le lunghezze. In particolare, i più elevati si aggirano attorno a 600 bytes di lunghezza del payload. Osservando infatti i grafici nell'immagine 4, notiamo una forte varianza

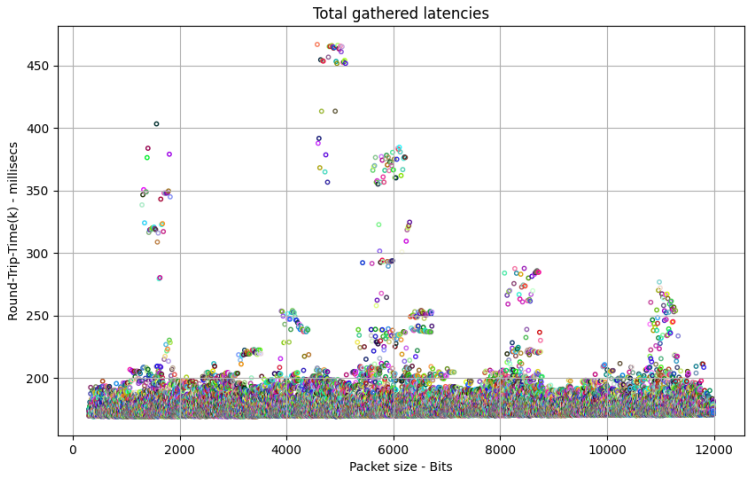

Figura 3: Grafici ottenuti con 100 istanze ed uno step di 2.

(fino a 30 millisecondi) proprio in queste zone. In questa immagine si rende ancora più evidente la crescita dei minimi e anche delle medie con il crescere della lunghezza dei pacchetti.

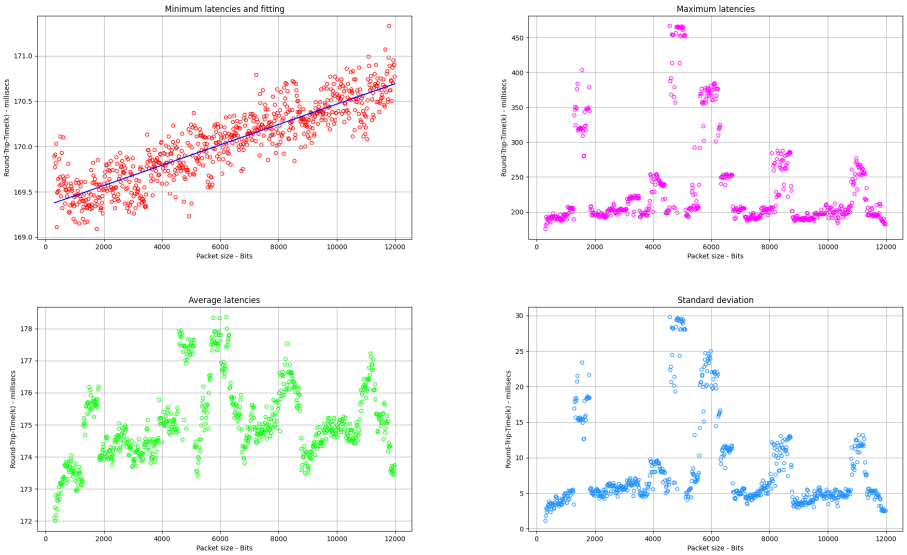

Figura 4: Grafici in riferimento all'immagine 3.

Infine, nell'immagine 5 sono state processate un totale di 365250 richieste in circa un quarto d'ora, risultato di 250 istanze per lunghezza ed uno step di 1 tra una lunghezza e l'altra. Si può infatti notare che il numero di dati in quest'ultimo grafico è molto più elevato rispetto ai due precedenti garantendo quindi che i dati analizzati sono più simili a quelli reali. In questo caso è interssante osservare che sia la media che la varianza sono elevate per la maggior parte delle lunghezze per decrescere solo quando il pacchetto è di dimensione ridotta oppure molto elevata (figura 6). Anche il grafico che rappresenta il minimo non segue più una retta definita (a differenza di quello mostrato nei grafici 4), ma dopo i 350 bytes si hanno minimi che arrivano fino a 178 millisecondi.

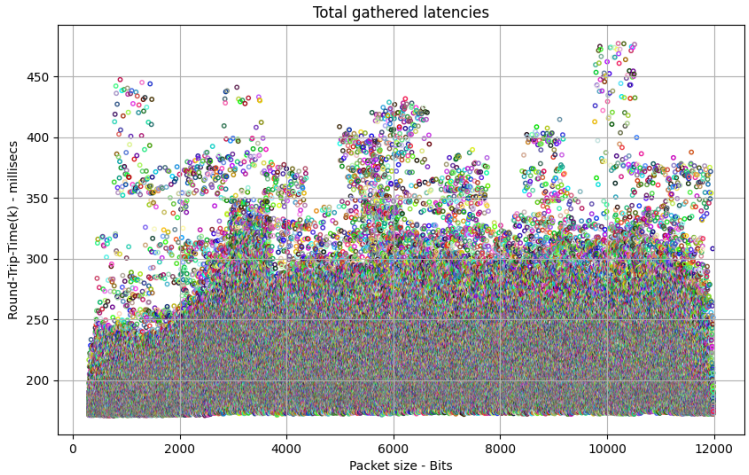

Figura 5: Grafici ottenuti con 250 istanze ed uno step di 1.

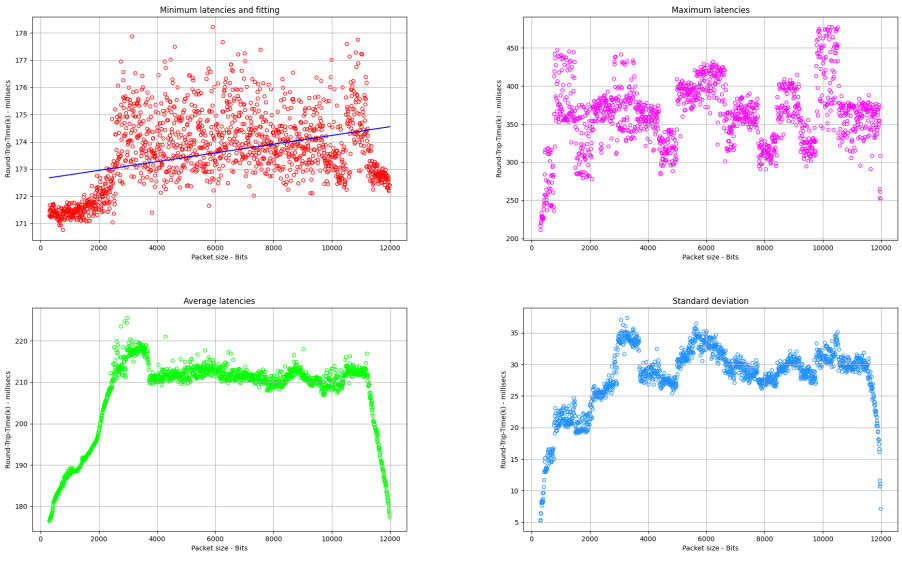

Figura 6: Grafici in riferimento all'immagine 5.

Osservando e comparando i tre grafici mostrati nelle figure si nota che, ragionevolemnte, maggiore è la mole di dati che viene analizzata e maggiori sono la varianza e la media in quanto si migliora l'accuratezza delle misure, avvicinandosi sempre di più ad un valore simile alla realtà.

2.3 Throughput

Attraverso tutti i dati raccolti è possibile riuscire a stimare il throughput. Si fornisce una breve spiegazione del ragionamento matematico che le supporta le scelte di calcolo del valore *throughput* $\mathcal{R}$.

Spiega throughput e indica l = ojfenowefowefn

Il RTT di un pacchetto k di lunghezza L ($\text{RTT}(L,k)$) è definito come la somma dei ritardi d_i causati dei collegamenti i-esimi ($i \in [1,n]$); tale latenza è a sua volta definita come la somma di 4 valori di latenza d:

$$d_i = d_{i,\text{processing}} + d_{i,\text{queue}} + d_{i,\text{transmission}} + d_{i,\text{propagation}}$$

Dove tali latenze indicano rispettivamente:

- $d_{i,\text{processing}}$, che indica il ritardo impiegato per controllare gli errori e calcolare l'instradamente del pacchetto — in genere è un ritardo trascurabile;
- $d_{i,\text{queue}}$, ovvero il tempo trascorso del pacchetto k all'interno di un buffer prima di essere elaborato — tale valore varia nel tempo, a seconda di quanto sia pieno il buffer, e verrà indicato con la notazione $q_i(k)$;
- $d_{i,\text{transmission}}$, è la durata del segnale che rappresenta il pacchetto — può essere matematicamente modellato come il rapporto tra la dimensione L del pacchetto k e dal throughput R_i del link i;
- $d_{i,\text{propagation}}$, rappresentato con τ_i, indica il tempo fisico di trasmissione del segnale lungo il medium.

Dunque il valore di d_i diventa:

$$d_i = q_i(k) + \frac{L}{R_i} + \tau_i$$

e il calcolo del Round Trip Tipe totale risulta quindi essere:

$$\begin{aligned}\text{RTT}(L,k) = \sum_{i=1}^{n} d_i &= \sum_{i=1}^{n} \left[q_i(k) + \frac{L}{R_i} + \tau_i \right] \\ &= \sum_{i=1}^{n} q_i(k) + L \sum_{i=1}^{n} \frac{1}{R_i} + \sum_{i=1}^{n} \tau_i\end{aligned}$$

A questo punto cerchiamo di riassumere le tre sommatorie come segue:

$$Q(k) = \sum_{i=1}^{n} q_i(k) \qquad \alpha = \sum_{i=1}^{n} \frac{1}{R_i} \qquad T = \sum_{i=1}^{n} \tau_i$$

E conseguentemente il calcolo del Round Trip Time diventa:

$$\mathrm{RTT}(L,k) = Q(k) + \alpha L + T$$

Per riuscire a ridurre completamente il Rounde Trip Time ad una retta, è necessario gestire la varibilità del ritardo di coda del pacchetto $Q(k)$. Si osserva che, con un elevato numero di istanze K — come quelle scelte in questo caso, dove $K = 250$ — la probabilità che almeno una volta i buffer dei link siano liberi aumenta. Di conseguenza, di tutti gli RTT, prendere la minima latenza per ogni pacchetto $k \in \{1, \ldots, K\}$, garantirebbe che $Q(k) = 0$. Si ottiene quindi un'approssimazione lineare del RTT:

$$RTT_{\min}(L) = \min_{k \in \{1,\ldots,K\}} \mathrm{RTT}(L,k) \approx \alpha L + T$$

A questo punto si hanno tutte le informazioni necessarie per ricavare il throughput $\mathcal{R}$ in quanto, attraverso tutti i dati ottenuti nel paragrafo 2.2, è possibile prendere il minimi RTT per ciascuna lunghezza $L \in \{1, \ldots, 1470\}$ e, attraverso una regressione lineare (detto anche interpolazione o *fitting*), è possibile ottenere sia il coefficiente α che la somma dei ritardi di trasmissione T. Una volta recuperato il coefficiente α, è sufficiente invertire la formula definita in precedenza per calcolare il throughput come segue:

$$\alpha = \sum_{i=1}^{n} \frac{1}{R_i} = \frac{n}{\mathcal{R}} \qquad \longrightarrow \qquad \mathcal{R} = \frac{n}{\alpha}$$

Si osserva che la relazione soprastante prende come ipotesi quella in cui tutti i link abbiano throughput uguali. Nell'eventualità in cui sia presente un link con throughput nettamente minore rispetto agli altri, si ha un effetto di collo di bottiglia — *bottleneck*. In questo caso, si ha una riduzione del throughput sia durante l'andata del pacchetto ping che al ritorno, ciò determina una riduzione del throughput globale, approssimabile con il calcolo del throughput nell'ipotesi in cui $n = 2$:

$$\mathcal{R}_{\text{bottleneck}} = \frac{n}{\alpha}\Big|_{n=2} = \frac{2}{\alpha}$$

I risultati matematici ottenuti si traducono in un numero estremamente ridotto di linee di codice. Prima di tutto è necessario creare la lista di lunghezza trasformata in bit, come menzionato all'inizio del paragrafo. In secondo luogo, tramite una regressione lineare, fornita dalla libreria `sklearn.linear_model`, è possibile recuperare il coefficiente α e quindi calcolare i due throughput: $\mathcal{R}$ ed $\mathcal{R}_{\text{bottleneck}}$.

```
1    payload_lengths_bit = [8 * (length + 28) for length in
         payload_lengths]
2
3    # use linear regression to retrive alpha, need to transform list into
         np.arrays
```

```
    reg = LinearRegression().fit(
        np.array(payload_lengths_bit).reshape(-1, 1), # transpose of
            payload lengths
        np.array(list(min_values.values()))
    )

    alpha = reg.coef_[0]

    # compute throughput
    throughput_identical_link = 2 * tracert_links / alpha
    throughput_bottleneck = 2 / alpha
```

Risultati

Eseguendo il codice descritto con le tre diverse istanze di esecuzione, è possibile calcolare i due casi di througput e analizzare meglio il *fit* sui minimi ottenuti per ciascuna lunghezza L. Iniziando dall'esecuzione con 20 istanze ed uno step di 10, mediante l'interpolazione lineare si ottiene che $\alpha \approx 7.5 \cdot 10^{-5}$ (figura 7) e quindi:

$$\mathcal{R} = 318511\,\text{bps} \approx 319\,\text{Kbps} \qquad \mathcal{R}_{\text{bottleneck}} = 26542\,\text{bps} \approx 27\,\text{Kbps}$$

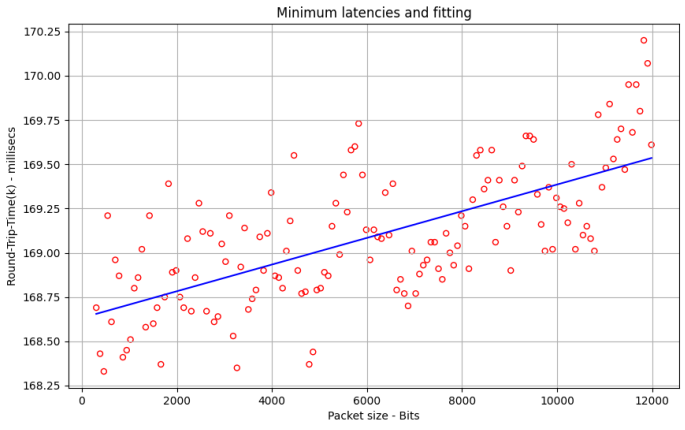

Figura 7: Regressione lineare del RTT minimo, in riferimento all'immagine 2.

Eseguendo lo script impostando $K = 100$ e lo step a 2 si giunge ad un coefficiente $\alpha \approx 1.12 \cdot 10^{-4}$ (immagine 8), maggiore rispetto a quello precedente che quindi riduce il valore del throughput a:

$$\mathcal{R} = 213508\,\text{bps} \approx 214\,\text{Kbps} \qquad \mathcal{R}_{\text{bottleneck}} = 17792\,\text{bps} \approx 18\,\text{Kbps}$$

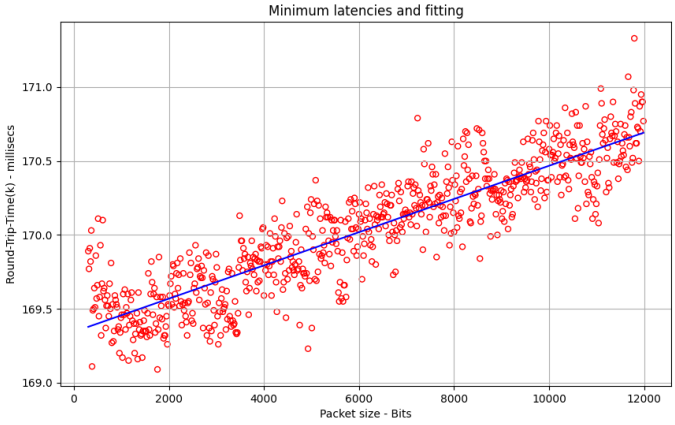

Figura 8: Regressione lineare del RTT minimo, in riferimento all'immagine 4.

Infine impostando le istanze ad un massimo di 250 e lo step minimo di 1, si ottiene che il valore del coefficiente $\alpha \approx 1.61 \cdot 10^{-4}$ è ancora maggiore (grafico 9), diminuendo ancora di più il valore del throughput:

$$\mathcal{R} = 149382\,\text{bps} \approx 149\,\text{Kbps} \qquad \mathcal{R}_{\text{bottleneck}} = 12448\,\text{bps} \approx 12\,\text{Kbps}$$

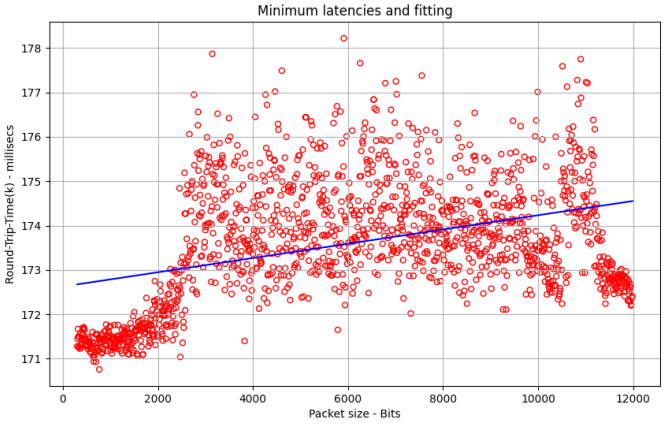

Figura 9: Regressione lineare del RTT minimo, in riferimento all'immagine 6.

3 Conclusioni

Fai le conclusioni

K	$\mathcal{R}$	$\mathcal{R}_{\text{bottleneck}}$
20	319 Kbps	27 Kbps
100	214 Kbps	18 Kbps
250	149 Kbps	12 Kbps