

# Multimedia

## Homework 2

Alessandro Trigo

7 Giugno 2024

## Indice

<b>1</b>	<b>Introduzione</b>	<b>4</b>
1.1	Utilizzo dello script . . . . .	4
1.2	Parametri di progetto . . . . .	5
<b>2</b>	<b>Performance di rete</b>	<b>5</b>
2.1	Numero di link attraversati . . . . .	5
2.2	Analisi del <i>Round Trip Time</i> . . . . .	7
2.3	Throughput . . . . .	13
<b>3</b>	<b>Conclusioni</b>	<b>14</b>

## Elenco delle figure

1	Tutte le ottenute con 20 istanze ed uno step di 10. . . . .	9
2	Grafici in riferimento all'immagine 1. . . . .	10
3	Grafici ottenuti con 100 istanze ed uno step di 2. . . . .	11
4	Grafici in riferimento all'immagine 3. . . . .	11
5	Grafici ottenuti con 250 istanze ed uno step di 1. . . . .	12
6	Grafici in riferimento all'immagine 5. . . . .	13

## Todo list

Fai introduzione . . . . .	4
indica che hai usato subprocess per invocare i cmd . . . . .	5
Di che nalle task 1 hai usato ping mentre enella task 2 haiu suato psiong . . . . .	5
Continua, capisci le formule di throughput . . . . .	13
Fai le conclusioni . . . . .	14

# 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<sup>1</sup> 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`.

```
1 | 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 istanze 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  $i + 1$ .

```
1 | # project specification
2 | SELECTED_SERVER_CITY = "Atlanta"
3 | INSTANCES = 30
4 | 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`.

```
1 | SERVERS = {
2 |     "Atlanta" : "atl.speedtest.clouvider.net",
3 |     "New York City" : "nyc.speedtest.clouvider.net",
4 |     "London" : "lon.speedtest.clouvider.net",
5 |     "Los Angeles" : "la.speedtest.clouvider.net",
6 |     "Paris" : "paris.testdebit.info",
7 |     "Lillie" : "lille.testdebit.info",
8 |     "Lyon" : "lyon.testdebit.info",
9 |     "Aix-Marseille" : "aix-marseille.testdebit.info",
10 |    "Bordeaux" : "bordeaux.testdebit.info"
11 | }
12 |
13 | # set choosen options
14 | server = SERVERS[SELECTED_SERVER_CITY]
15 | city = server.split(".")[0]
16 | payload_lengths = range(10, 1471, STEP_BETWEEN_LENGTHS)
```

Se si usano quindi le impostazioni mostrate nell'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`, effettuerà 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 vengono 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 riuscire ad utilizzare lo script è necessario 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 non primo punto non si è utilizzato `psping` in quanto non è presente l'impostazione che permette di ridurre il Time-To-Live del pacchetto.

- Los Angeles
- $K = 250$
- $\text{Dim} = 10 \rightarrow 1470$ , step di 1

indica che  
hai usato  
subprocess  
per invocare  
il cmd

Di che nelle  
task 1 hai  
usato ping  
mentre enel-  
la task 2  
hai usato  
psping

## 2 Performance di rete

Dopo aver introdotto i parametri di progetto adottati, andremo ora ad analizzare le performance 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 deviazione 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<sup>2</sup> compie al fine di raggiungere la destinazione. Il comando `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 `tracert` 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}"
5     result = subprocess.run(cmd, shell=True,
6                             stdout=subprocess.PIPE, stderr=subprocess.PIPE,
7                             text=True)
8
9     last_link_line = result.stdout.split("\n")[-4]
10
11     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-

<sup>2</sup>IP è l'acronimo di *Internet Protocol*

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 approccio 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.

```
1 | def get_links_from_ping(server: str) -> int:
2 |
3 |     for ttl in range(20, 0, -1):
4 |
5 |         cmd = f"ping {server} -i {ttl}"
6 |         result = subprocess.run(cmd, shell=True,
7 |                                 stdout=subprocess.PIPE, stderr=subprocess.PIPE,
8 |                                 text=True)
9 |
10 |        if "TTL scaduto durante il passaggio" in result.stdout:
11 |            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 comune `ping`, inoltre permette 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 istanze 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.

```
1 | def ping_server(server, instances, length):
2 |
```

```

3 |         cmd = f"psping -n {instances} -l {length} -i 0 -w 0
      {server}"
4 |         result = subprocess.run(cmd, shell=True,
      stdout=subprocess.PIPE, stderr=subprocess.PIPE,
      text=True)
5 |
6 |         millisecs_vector = []
7 |         lines = result.stdout.split("\n")
8 |
9 |         for line in lines:
10 |             if "Reply from" in line:
11 |                 duration = float(line.split(": ")[1].split("ms")[0])
12 |                 millisecs_vector.append(duration)
13 |
14 |         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.

```

1 |     stats = {}
2 |
3 |     # using ThreadPoolExecutor to improve computational capabilities
4 |     with concurrent.futures.ThreadPoolExecutor(max_workers=100) as
      executor:
5 |
6 |         futures = []
7 |         for length in payload_lengths:
8 |             # submit a task to the executor to ping the server with the
              specified parameters
9 |             future = executor.submit(ping_server, server,
              INSTANCES, length)
10 |            futures.append(future)
11 |
12 |            # iterate over completed futures as they become available
13 |            for future in concurrent.futures.as_completed(futures):
14 |                length, millisecs_vector = future.result()
15 |                stats[length] = millisecs_vector
16 |
17 |            # order stats by length
18 |            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.



```

1   max_values = {}
2   min_values = {}
3   average_values = {}
4   standard_deviations = {}
5
6   for key, value in stats.items():
7       max_values[key] = max(value)
8       min_values[key] = min(value)
9       average_values[key] = sum(value) / len(value)
10      standard_deviations[key] = math.sqrt(sum((x -
          average_values[key]) ** 2 for x in value) / len(value))

```

## Grafici

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; cioasune 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

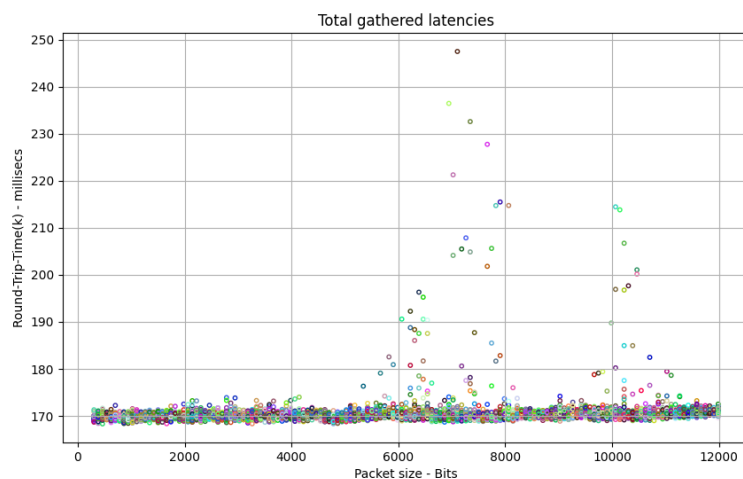


Figura 1: Tutte le ottenute con 20 istanze ed uno step di 10.

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).

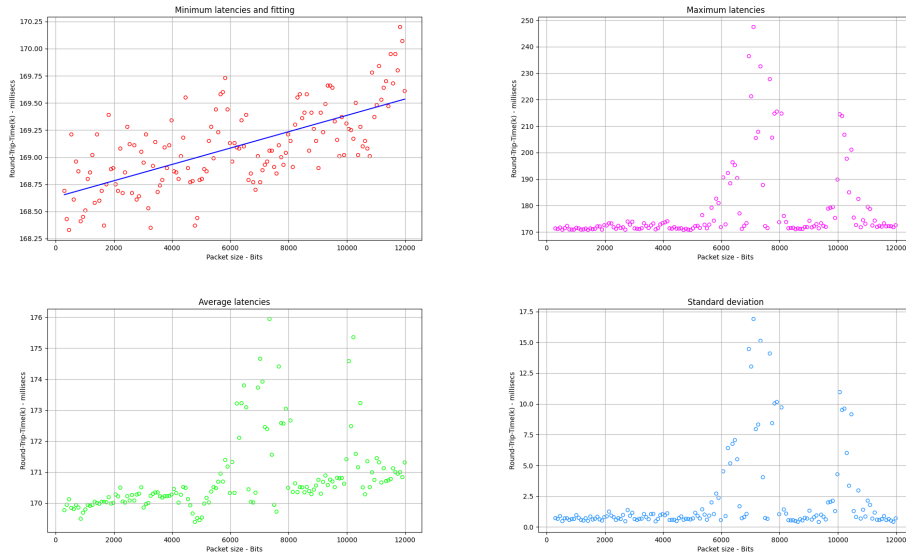


Figura 2: Grafici in riferimento all'immagine 1.

Aumentando il numero di ping effettuati, in particolare aumentonando 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 (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.

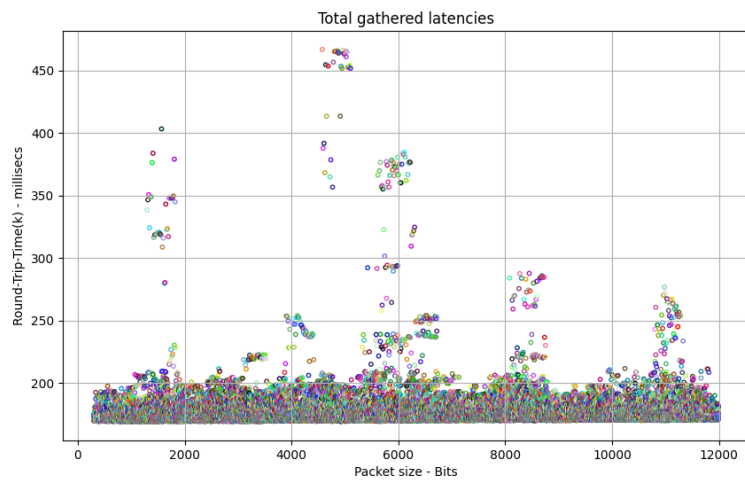


Figura 3: Grafici ottenuti con 100 istanze ed uno step di 2.

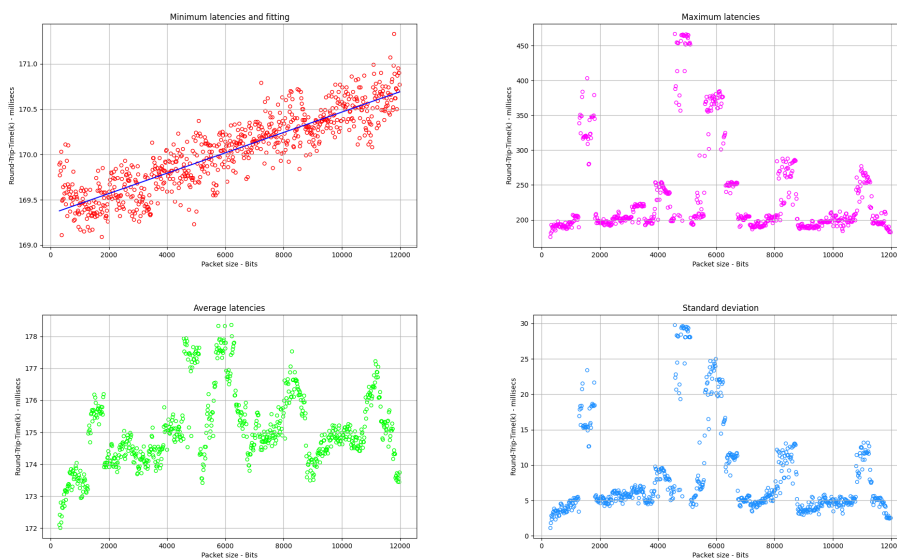


Figura 4: Grafici in riferimento all'immagine 3.

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 interessante osservare che sia la media che la

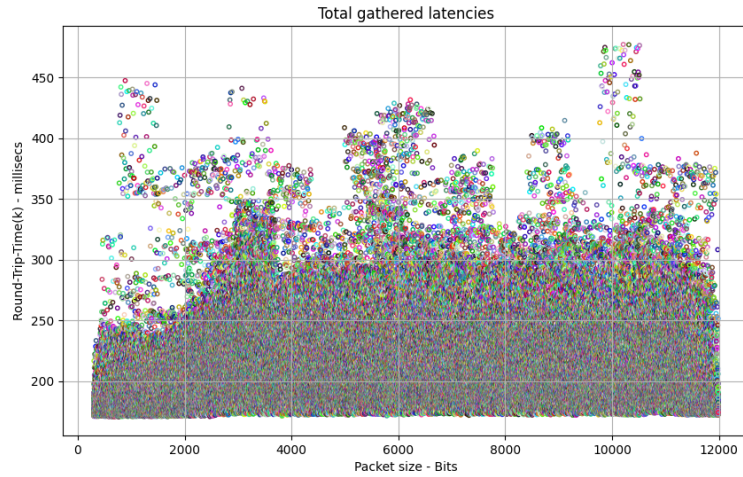


Figura 5: Grafici ottenuti con 250 istanze ed uno step di 1.

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.

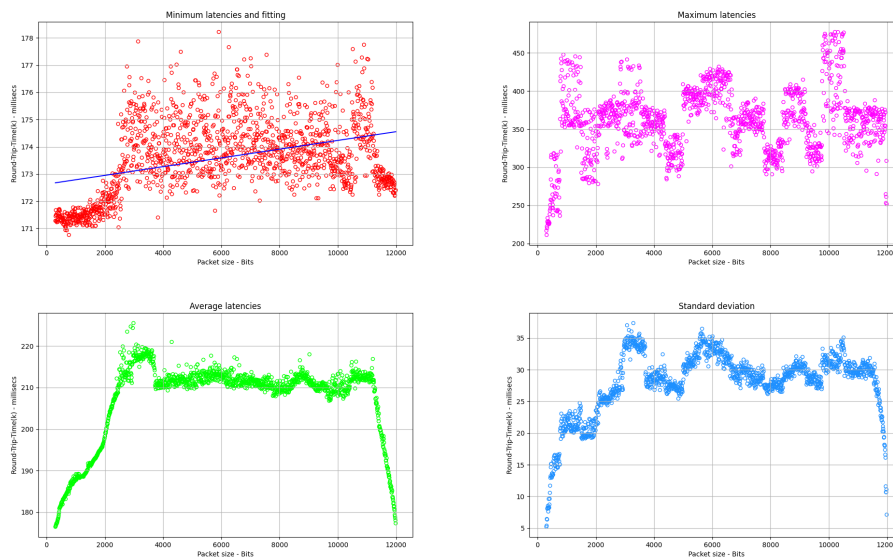


Figura 6: Grafici in riferimento all'immagine 5.

Osservando e comparando i tre grafici mostrati nelle figure si nota che, ragionevolmente, 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 quindi possibile effettuare una regressione lineare al fine di ottenere

```

1  payload_lengths_bit = [8 * (length + 28) for length in
    payload_lengths]
2
3  # use linear regression to retrieve alpha, need to transform list into
    np.array
4  reg = LinearRegression().fit(
5      np.array(payload_lengths_bit).reshape(-1, 1), # transpose of
        payload_lengths
6      np.array(list(min_values.values()))
7  )
8
9  alpha = reg.coef_[0]
10
11 # compute throughput
12 throughput_identical_link = 2 * tracers_links / alpha
13 throughput_bottleneck = 2 / alpha

```

Continua,  
capisci le  
formule di  
throughput

### 3 Conclusioni

Fai le conclusioni