

# TRASFERIMENTO FILE SU UDP

---

Progetto di Ingegneria di Internet e del Web 2018-2019

Realizzato da:

Camilli Michela

Falaschi Valentina

Lasco Giuseppe

Marcucci Marco

## Sommario

1	Il protocollo Selective Repeat.....	3
1.3	Implementazione Selective Repeat.....	4
1.2	Implementazione timer .....	7
1.3	Implementazione probabilità di perdita .....	9
1.4	Gestione concorrenza.....	9
2	Architettura Client-Server .....	11
3	Manuale d'uso .....	13
4	Esempi d'utilizzo .....	15
5	Test prestazionali.....	16
6	Bibliografia .....	21

# 1 Il protocollo Selective Repeat

Il protocollo Selective Repeat appartiene alla famiglia di protocolli di comunicazione affidabile, che gestiscono la ritrasmissione dei pacchetti, in caso di perdita di questi ultimi. In particolare, nel caso in esame, viene rinviato solo il pacchetto perso. Ogni qual volta una ricezione avvenga con successo, viene inviato un ack di conferma.

Per la realizzazione del trasferimento di dati affidabile è necessario gestire i seguenti meccanismi:

- Timer
- Probabilità di perdita
- Numero di sequenza
- Finestra
- Acknowledgement

Per quanto concerne la gestione del **timeout**, si è fatto riferimento alle formule discusse per TCP nel libro “Reti di calcolatori e Internet”. In particolare:

$$\text{EstimatedRTT} = (1-\alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}$$

$$\text{DevRTT} = (1-2\alpha) \times \text{DevRTT} + 2\alpha \times |\text{SampleRTT} - \text{EstimatedRTT}| \quad (*)$$

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}$$

(\*) Il valore  $\beta$ , originariamente presente nella formula, è stato sostituito con  $2\alpha$ , come raccomandato nel paragrafo a cui si fa riferimento.

TimeoutInterval viene raddoppiato ogniqualvolta si verifichi un timeout, per evitare che il timer relativo ad un pacchetto successivo scada prematuramente. D'altra parte, EstimatedRTT viene aggiornato alla ricezione di ogni pacchetto e TimeoutInterval viene ricalcolato di conseguenza.

Il timer associato a ciascun pacchetto scade a causa della **perdita** dello stesso. Per mostrare tale caratteristica, è stato necessario implementare un meccanismo di probabilità di perdita, che verrà meglio spiegato in seguito.

Client e Server hanno **finestre** distinte; ciò comporta che essi abbiano una visione diversa dello stato di trasmissione.

L'ampiezza della finestra è stata scelta uguale alla metà dello spazio dei **numeri di sequenza**; tale scelta risulta ottimale, evitando ambiguità nell'invio di pacchetti e nella ricezione degli ack.

Gli **acknowledgement** nell'algoritmo di selective repeat, sono, per l'appunto, selettivi e mantengono il numero di sequenza del singolo pacchetto. In particolare, viene mantenuto il riferimento al valore di *send\_base*, rappresentante il numero di sequenza del pacchetto in attesa di riscontro da più tempo. Alla ricezione dell'ack del pacchetto *send\_base*, la base della finestra del mittente si muove verso il pacchetto con il più piccolo numero di sequenza che non ha ancora ricevuto acknowledgement.

## 1.3 Implementazione Selective Repeat

Nell'implementazione del protocollo, pacchetti e ack sono stati incapsulati in due apposite strutture:

```
struct packet{
    int seq;
    int ack;
    int dim_data (*);
    char data[DIM_DATA_BLOK];
};
```

```
struct ack{
    int seq;
    int ack;
};
```

(\*) si è optato per un valore di *dim\_data* pari a 1300 byte.

In particolare, i dati vengono formattati nelle strutture dalle funzioni *make\_ack* e *make\_packet*.

Tutte le funzioni fondamentali alla realizzazione del protocollo di ripetizione selettiva sono contenute in SRprotocol.c. Queste sono:

- writeSR
- receiveSRack
- receiveSR
- receive\_routine

Le funzioni *writeSR* e *receiveSRack* vengono eseguite da due thread distinti e concorrenti tra loro nella gestione della finestra del protocollo.

### ***writeSR***

Tale funzione viene chiamata dalla *write\_transfer (\*)* e il suo ruolo fondamentale è quello di inviare pacchetti, lato mittente, sulla socket dedicata.

Svolge, inoltre, altri importanti compiti, tra cui:

- I. Creazione del timer del pacchetto;
- II. Gestione della probabilità di perdita di un pacchetto, tramite la funzione *get\_loss\_probability*;
- III. Gestione della finestra.

#### III.

Per quanto concerne la gestione della finestra, è stato usato, in fase di invio, un buffer ciclico di puntatori a *struct packet* di dimensione pari a *win\_size* (la dimensione della finestra scelta dall'utente). Questa scelta è derivata dal fatto che il mittente, a differenza del destinatario, invia pacchetti sequenzialmente e, pertanto, questi vengono inseriti all'interno della finestra già in ordine. Si è, quindi, mantenuta traccia dei valori *send\_base* ed *end\_base*, rappresentanti gli indici del buffer in cui si trovano, rispettivamente, il pacchetto che da più tempo è in attesa di ack e l'ultimo pacchetto inviato.

L'aggiornamento della finestra, che avviene ogniqualvolta venga inviato un pacchetto, si effettua, in modo atomico, modificando il parametro *end\_base*.

(\*) funzione contenuta in file transfer.c, nella quale viene fatta la *read* del file inserito dall'utente, calcolandone volta per volta i byte letti complessivamente.

### ***receiveSRack***

Tale funzione viene chiamata dalla *receive\_ack* (contenuta in file transfer.c), il suo compito è gestire la ricezione degli ack.

I passi di cui si compone sono:

- I. Si valuta se l'ack è riferito ad un pacchetto avente numero di sequenza nell'intervallo accettabile;
- II. Aggiornamento TimeoutInterval;
- III. Gestione finestra.

#### I.

Nel caso in cui il numero di sequenza dell'ack ricevuto sia contenuto nell'intervallo accettabile e corrisponda a *send\_base*, la finestra viene liberata dei pacchetti che hanno ricevuto l'ack (che vengono anche deallocati). Al contrario, se il numero di sequenza dell'ack non fosse *send\_base*, viene impostato a 1 il campo *ack* della struttura *packet*, ad indicare che è stato ricevuto il riscontro relativo a quello specifico pacchetto.

## II.

Viene eliminato il timer associato al pacchetto e viene calcolato il nuovo valore del TimeoutInterval (come discusso precedentemente).

## III.

Vengono aggiornati in maniera atomica i valori di send\_base e end\_base, nei casi in cui ci siano o meno altri pacchetti in attesa di ack.

### ***receiveSR***

La funzione *receiveSR* viene chiamata dalla *read\_transfer (\*)*, lato destinatario, e il suo compito principale è quello di ricevere pacchetti sulla socket dedicata.

La *receiveSR* chiama, quindi, la *receive\_routine*.

(\*)funzione contenuta in file\_transfer.c, nella quale, se il comando è di tipo *get*, viene creata la cartella associata al singolo Client, in cui verrà posto il file richiesto dall'utente.

### ***receive\_routine***

Questa funzione viene chiamata dalla *receiveSR*, e si occupa della scrittura dei pacchetti su file.

Tale operazione si compone di più passi specifici, in particolare:

- I. Si valuta se la sequenza del pacchetto considerato sia contenuta nell'intervallo accettabile;
- II. Si gestisce l'ultimo pacchetto e la chiusura della connessione;
- III. Invio degli ack sulla socket ad essi dedicata;
- IV. Gestione finestra.

## I.

Nel caso in cui il pacchetto ricevuto sia contenuto nell'intervallo accettabile e corrisponda a *send\_base*, vengono scritti su file, in ordine, tutti i pacchetti già ricevuti in maniera sequenziale(compreso *send\_base*). Gli indirizzi di tali pacchetti vengono, quindi, deallocati e viene aggiornato il valore di *send\_base*.

Nel caso il pacchetto non fosse *send\_base*, questo viene salvato nel buffer.

## II.

Nel caso in cui il pacchetto ricevuto sia l'ultimo, viene inviato un "pacchetto speciale" al mittente, che indica la chiusura della connessione.

### III.

In qualsiasi caso, viene inviato un ack al mittente, poiché quest'ultimo deve essere a conoscenza che il pacchetto è stato ricevuto correttamente.

### IV.

Per quanto concerne la gestione della finestra è stato usato, in fase di ricezione, un buffer ciclico di puntatori a *struct packet* di dimensione pari a  $2 \times win\_size$  (il doppio della dimensione della finestra scelta dall'utente).

Questa scelta deriva dal fatto che i pacchetti, in questo caso, potrebbero non arrivare in ordine; perciò, ad ogni numero di sequenza è riservato uno slot specifico nel buffer.

Si è ritenuto opportuno operare in questo modo, a discapito di un limitato spreco di memoria, per evitare complessità derivate dal riordino dei pacchetti, che avrebbero causato un sovra-utilizzo di tempo in CPU.

In questo caso, *send\_base* rappresenta il numero di sequenza del pacchetto atteso, cioè il pacchetto immediatamente successivo all'ultimo pacchetto scritto su file.

## 1.2 Implementazione timer

In Selective Repeat si utilizzano i timeout per cautelarsi contro la perdita di pacchetti. Ad ogni trasmissione viene associato un timer e al verificarsi dell'evento di timeout si procede alla ritrasmissione del pacchetto.

Nel programma un timer è rappresentato dalla seguente struttura:

```
struct timer {  
    timer_t * id_timer;  
    struct packet* pkt;  
    struct timer* next_timer;  
    struct protocolSR* protocol;  
    pthread_mutex_t* mutex;  
    int arrived;  
    long to;  
};
```

Il campo *id\_timer* rappresenta l'identificativo del timer.

Il mittente mantiene i timer associati ai pacchetti in volo all'interno di una lista collegata; il campo *next\_timer* ha, infatti, lo scopo di mantenere il riferimento al timer successivo della lista.

Il campo *arrived* viene settato a 0 nel caso in cui il pacchetto non sia ancora arrivato al destinatario, altrimenti viene posto a 1.

Il campo *to* identifica l'intervallo di timeout.

Il campo *mutex* indica il pthread mutex, la cui funzione verrà spiegata in seguito.

Le funzioni fondamentali alla gestione dei timer, contenute nel file timer.c, sono:

- `create_timer`
- `timer_handler`
- `to_calculate`

### ***create\_timer***

Questa funzione viene chiamata dalla *writeSR*, il suo scopo principale è quello di associare ad ogni pacchetto un proprio timer.

Passi fondamentali:

- I. Inizializzazione parametri della struttura timer attraverso l'ausilio della funzione *make\_timer*;
- II. Creazione timer POSIX;
- III. Inserimento timer all'interno della lista collegata.

#### II.

La creazione di un timer avviene in seguito alla chiamata della funzione *create\_timer*.

I parametri della funzione sono stati impostati in modo che, allo scadere di un timer, venga eseguita la funzione *timer\_handler*. Tale scelta ha permesso di gestire in maniera univoca i singoli timer associati a ciascun pacchetto.

### ***timer\_handler***

La seguente funzione, eseguita allo scadere di un timer, ha come scopo principale quello di rinviare i pacchetti.

Passi fondamentali:

- I. Si controlla se l'ack del pacchetto sia stato ricevuto;
- II. Si invia il pacchetto, nel caso in cui questo non sia ancora stato riscontrato.

#### I.

Prima di procedere con qualsiasi operazione, viene controllato il campo *arrived*: se questo si trova settato a 1, come specificato precedentemente, significa che il pacchetto è già stato riscontrato e quindi il timer viene cancellato, attraverso l'apposita funzione *delete\_timer*, e tutte le risorse relative ai timer vengono deallocate.

In caso contrario, quello in cui l'ack del pacchetto non sia ancora arrivato, viene aggiornato il valore del timer, in particolare il valore di *to*. Questa verifica è necessaria per via della



concorrenza tra i thread, poiché un ack può arrivare in contemporanea allo scadere del timer corrispondente.

### ***to\_calculate***

Tale funzione viene chiamata soltanto nel caso in cui l'utente scelga di utilizzare timer adattivo, cioè soltanto nel caso in cui il campo *flag\_timer* sia posto a 1.

Il suo scopo principale è calcolare i nuovi valori di timeout e per farlo sono state utilizzate le formule precedentemente descritte.

Nello sviluppo del progetto si è ritenuto opportuno usare i timer POSIX, anziché i timer UNIX o più semplicemente l'API `alarm()`, poiché i primi vanno incontro a numerose limitazioni, come suggerito dal libro *The Linux programming interface*, mentre l'API `alarm()` non permette di tenere attivi più timer contemporaneamente nell'ambito dello stesso processo.

## **1.3 Implementazione probabilità di perdita**

Per simulare la perdita dei messaggi in rete è stato implementato un meccanismo apposito.

In particolare, in `timer.c` è stata realizzata la funzione *get\_loss\_probability*. Questa prende come parametro la probabilità di perdita (*loss\_p*) selezionata dall'utente e la confronta con un numero casuale, ottenuto tramite la funzione `rand()` e normalizzato al valore `RAND_MAX`.

Se il valore casuale è inferiore a *loss\_p*, la funzione ritorna 1 e, quindi, si ha effettivamente perdita del pacchetto; in caso contrario, il pacchetto viene inviato con successo.

Tale funzione viene invocata in `SRprotocol.c` e in `timer.c`.

Nel primo caso è chiamata nella *writeSR* e, se si ha perdita, il pacchetto non viene inviato (viene, quindi, saltata la *sendto*).

Nel secondo caso lo stesso ragionamento viene effettuato quando scade il timer relativo ad un pacchetto; se *get\_loss\_probability* ritorna 1, il pacchetto non viene rinviato.

## **1.4 Gestione concorrenza**

Sono stati utilizzati, al fine di gestire la concorrenza, **semafori** e **pthread mutex** di tipo POSIX. Tale scelta è risultata necessaria per garantire atomicità nelle sezioni critiche condivise tra thread.

In particolare, si è dovuto ricorrere all'ausilio di strutture semaforiche nei seguenti casi:

### SRprotocol.c

Il pthread mutex, inizializzato nella *initializeSR*, è stato utilizzato sia nella *writeSR* che nella *receiveSRack* per fare in modo che i parametri del protocollo venissero aggiornati in maniera atomica.

Si è usato, inoltre, un semaforo con numero di token pari alla dimensione della finestra. Questo semaforo, allocato anch'esso nella *initializeSR*, ha permesso di poter gestire il processo di invio di pacchetti e di ricezione degli ack per un numero di questi in volo pari, per l'appunto, al valore di *win\_size*. Attraverso di esso si è potuto tener traccia della capacità disponibile nella finestra.

In particolare, la *sem\_wait* è stata inserita all'inizio della *writeSR*. La *sem\_post*, d'altra parte, è posizionata nella *receiveSRack*. Dunque dopo la ricezione dell'ack di *send\_base* viene liberato lo spazio nella finestra, consentendo di inviare nuovi pacchetti.

### timer.c e linked\_list.c

Del pthread mutex, inizializzato nella *initialize*, viene fatto il lock e l'unlock nella *timer\_handler* o nella *remove\_node\_and\_delete\_timer\_by\_seq*. Tale mutex è risultato necessario per gestire:

- il caso in cui il timer relativo ad un pacchetto scada in contemporanea all'arrivo del suo stesso ack;
- la possibilità che il thread che gestisce l'arrivo degli ack venga bloccato a causa della scadenza del timer.

Tale mutex serve, quindi, a gestire la sincronizzazione degli eventi di arrivo degli ack e scadenza del timer e a fare in modo che il thread che acquisisce il controllo termini il suo compito.

## 2 Architettura Client-Server

Per quanto concerne l'**instaurazione della connessione**, Client e Server sono organizzati nel modo seguente:

### Lato Client:

Il processo padre si connette con il Server creando un processo figlio che gestisce la comunicazione. Nel frattempo, il padre attende la morte del processo figlio per far sì che l'utente possa, una volta terminata la comunicazione, effettuare una nuova richiesta senza che la connessione venga chiusa.

### Lato Server:

Al lancio del programma, viene creato un processo delegato alla creazione di una socket di connessione. Tale processo si pone, quindi, in ascolto di nuove connessioni; appena un Client si connette sulla porta, il processo padre genera un processo figlio che si occupa della comunicazione: vengono create due socket, ciascuna di esse utilizzata da Client e Server sia lato mittente che destinatario, aventi due diversi numeri di porta che vengono comunicati al Client; a questo punto inizia la comunicazione. Nel mentre questa è attiva, il padre si mette in ascolto di una nuova connessione, in modo da creare un Server concorrente.

I **comandi inviati dall'utente** (oltre al comando di uscita *quit*) possono essere di tre tipi: *get*, *put* o *list*.

### ***get <file>***

#### Lato Client:

- Vengono inizializzati i parametri di protocollo e viene creata un'apposita directory in *Client\_repository*;
- Se il file richiesto esiste, viene chiamata la funzione *read\_transfer*.

#### Lato Server:

- Vengono inizializzati i parametri di protocollo;
- Vengono creati due thread: uno delegato a invocare la funzione *receive\_ack*, l'altro la *write\_transfer*;
- Si attende la terminazione del thread che esegue la *write\_transfer* e, in seguito, viene eliminato il thread che esegue la *receive\_ack*.

### ***put <file>***

#### Lato Client:

- Vengono inizializzati i parametri di protocollo;
- Viene ricevuto un messaggio dal Server, attraverso cui il Client viene sbloccato (meccanismo di sincronizzazione);
- Vengono creati due thread: uno delegato a invocare la funzione *receive\_ack*, l'altro la *write\_transfer*;

- Si attende la terminazione del thread che esegue la *write\_transfer* e, in seguito, viene eliminato il thread che esegue la *receive\_ack*.

#### Lato Server:

- Vengono inizializzati i parametri di protocollo;
- Viene inviato un messaggio di sincronizzazione, ad indicare che il Server è pronto a ricevere;
- Viene ricevuta, tra le informazioni, quella relativa alla dimensione del file;
- Viene invocata la *receiveSR*.

#### ***list***

#### Lato Client:

- Vengono inizializzati i parametri di protocollo e viene creata un'apposita directory in *Client\_repository*;
- Viene chiamata la funzione *read\_transfer*;
- Viene stampato a schermo il contenuto del file "list.txt";
- Viene rimossa la directory.

#### Lato Server:

- Viene invocato il comando *ls* della cartella *Server\_repository*, reindirizzato sul file "list.txt";
- Viene chiamata la funzione *get* (lato Server).

### 3 Manuale d'uso

Per compilare il programma, è sufficiente aprire un terminale dalla cartella del progetto ed eseguire i comandi *make client* e *make server*. Dopo di ciò, bisogna effettuare i seguenti passi:

- i. Nel terminale digitare il comando **`./server`**
  - In questo modo viene eseguita l'applicazione e il Server si pone in attesa di eventuali connessioni.
- ii. Aprire un nuovo terminale nella cartella e digitare un comando del tipo:  
**`./client <ip_server> <window_size> <loss_probability> <flag_timer> <timer_value_in_millisec>`**
  - **`./client <ip_server>`** -> viene eseguita l'applicazione e il Client si connette al Server tramite l'indirizzo ip specificato;
  - **`<window_size>`** -> grandezza della finestra che si vuole utilizzare;
  - **`<loss_probability>`** -> probabilità di perdita dei pacchetti. Essendo, per l'appunto, una probabilità, deve essere un numero  $p \in [0,1]$ ;
  - **`<flag_timer>`** -> flag che permette di scegliere se utilizzare un timeout fisso (flag posto a 0), oppure un timeout adattivo (flag posto a 1);
  - **`<timer_value_in_millisec>`** -> rappresenta il valore, in millisecondi, del timeout nel caso di timeout fisso o il valore di partenza del timeout nel caso adattivo.

Una volta effettuate tali operazioni, è richiesto all'utente l'inserimento di uno dei comandi discussi in precedenza. Vengono, quindi, assegnati i numeri di porta al Client e al Server e si inizia il trasferimento (se il comando non presenta errori). Quando quest'ultimo termina, Client e Server si riconnettono con nuovi numeri di porta.

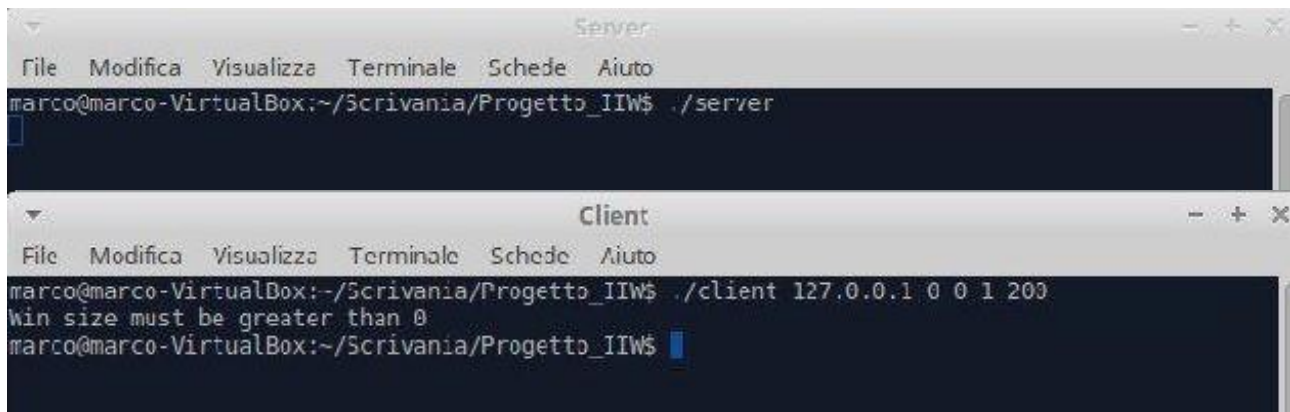
Se si volesse lanciare il programma con i file di prova allegati al progetto o con propri file, basta inserirli:

- Nella cartella `Server_repository` nel caso si volesse eseguire un comando di *get* oppure di *list*;
- Nella cartella `Client_repository` nel caso si volesse eseguire un comando di *put*;

Nel caso in cui si volessero utilizzare più Client contemporaneamente, è sufficiente aprire nuovi terminali e ripetere, per ciascuno di essi, il comando descritto nel punto ii.

In seguito sono mostrati alcuni esempi di lancio dell'applicazione, in cui vengono inseriti esplicitamente i valori di cui parlato in precedenza.

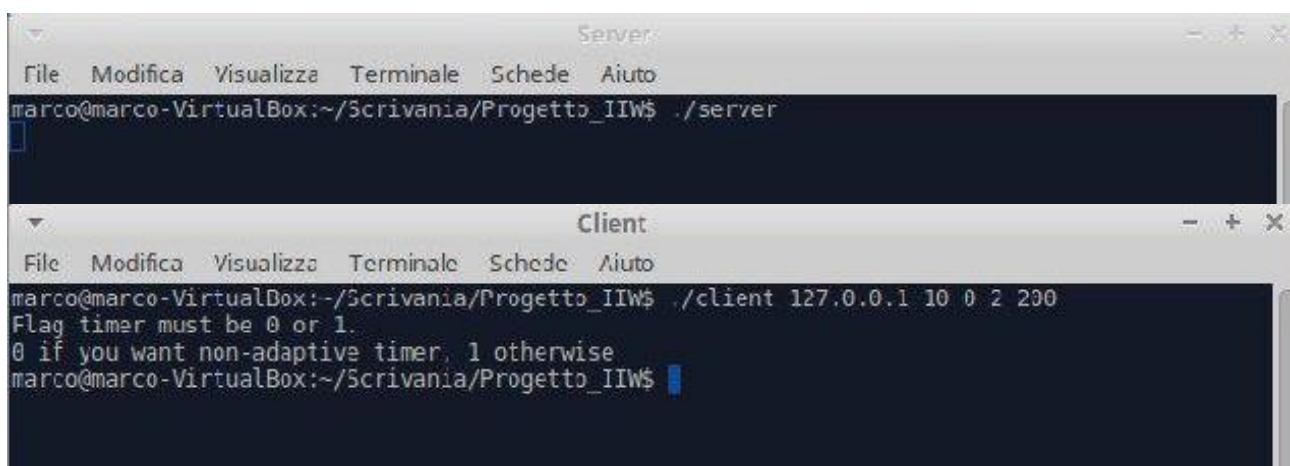
I. Lancio dell'applicazione con parametro <window\_size> errato



```
Server
File Modifica Visualizza Terminale Schede Aiuto
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./server

Client
File Modifica Visualizza Terminale Schede Aiuto
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./client 127.0.0.1 0 0 1 200
win size must be greater than 0
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$
```

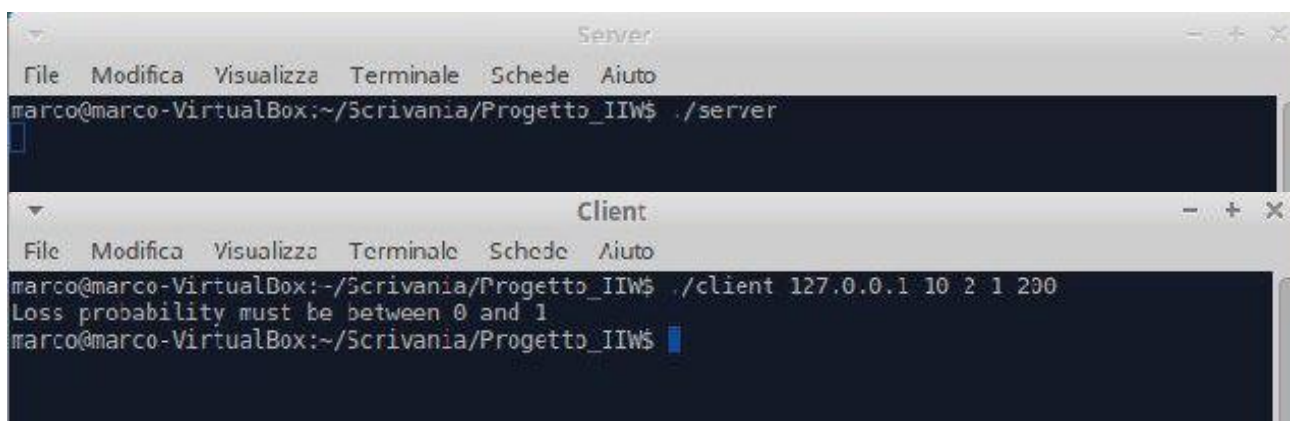
II. Lancio dell'applicazione con parametro <flag\_timer> errato



```
Server
File Modifica Visualizza Terminale Schede Aiuto
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./server

Client
File Modifica Visualizza Terminale Schede Aiuto
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./client 127.0.0.1 10 0 2 200
Flag timer must be 0 or 1.
0 if you want non-adaptive timer, 1 otherwise
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$
```

III. Lancio dell'applicazione con parametro <loss\_probability> errato



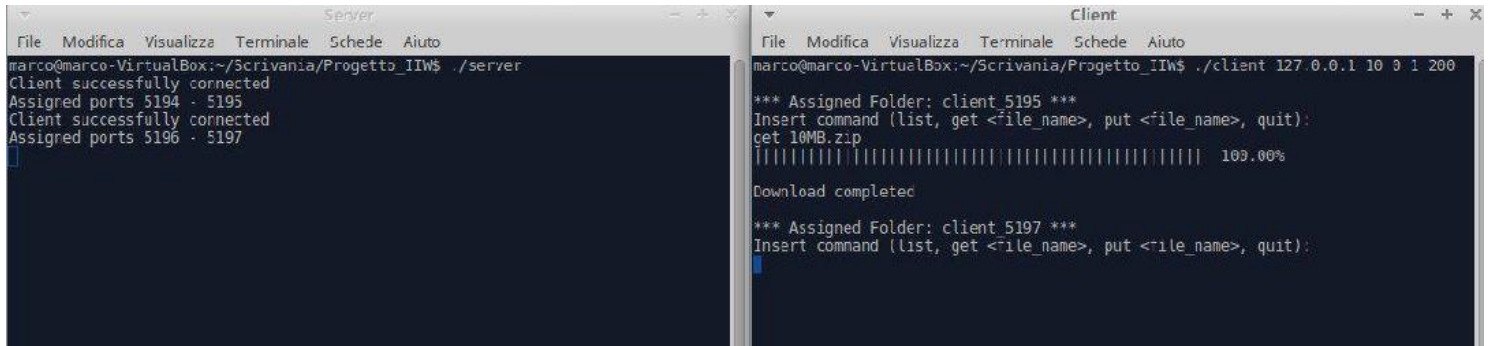
```
Server
File Modifica Visualizza Terminale Schede Aiuto
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./server

Client
File Modifica Visualizza Terminale Schede Aiuto
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./client 127.0.0.1 10 2 1 200
Loss probability must be between 0 and 1
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$
```

## 4 Esempi d'utilizzo

In questo paragrafo si vuole mostrare il funzionamento dell'applicazione nelle situazioni di maggiore interesse e, cioè, quando l'utente esegue i comandi di get, put e list.

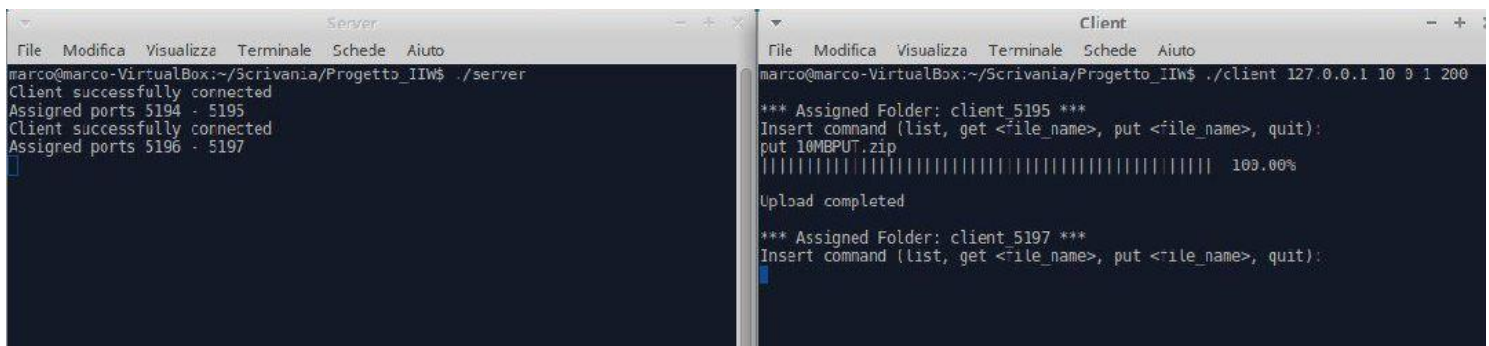
### I. Utilizzo del comando *get*



```
Server
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./server
Client successfully connected
Assigned ports 5194 - 5195
Client successfully connected
Assigned ports 5196 - 5197

Client
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./client 127.0.0.1 10 0 1 200
*** Assigned Folder: client 5195 ***
Insert command (list, get <file_name>, put <file_name>, quit):
get 10MB.zip
||||| 100.00%
Download completed
*** Assigned Folder: client 5197 ***
Insert command (list, get <file_name>, put <file_name>, quit):
```

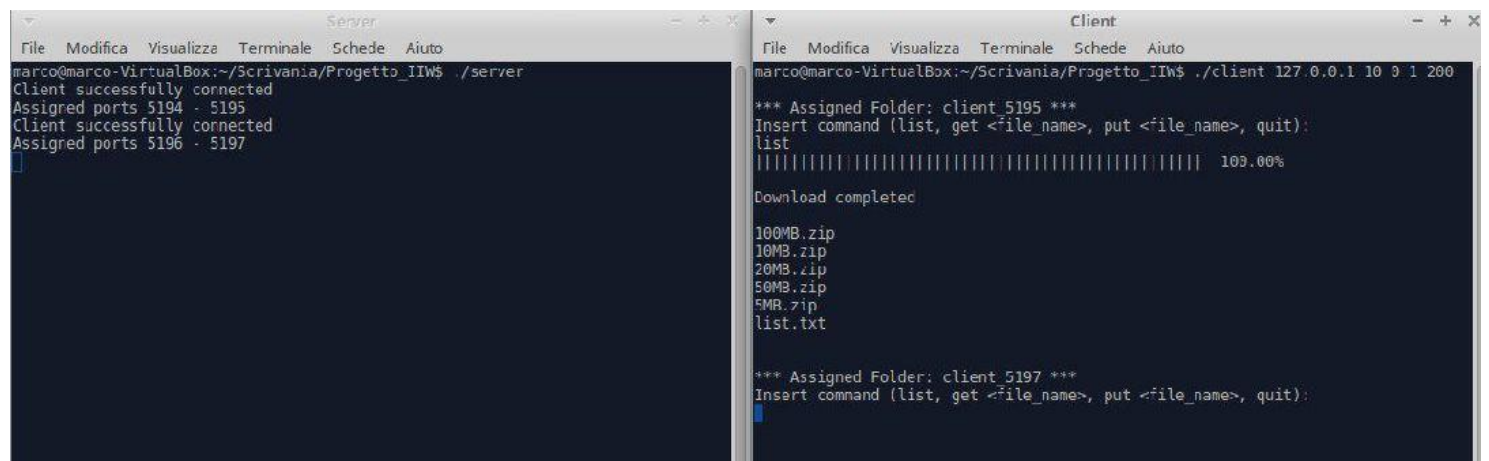
### II. Utilizzo del comando *put*



```
Server
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./server
Client successfully connected
Assigned ports 5194 - 5195
Client successfully connected
Assigned ports 5196 - 5197

Client
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./client 127.0.0.1 10 0 1 200
*** Assigned Folder: client 5195 ***
Insert command (list, get <file_name>, put <file_name>, quit):
put 10MBPUT.zip
||||| 100.00%
Upload completed
*** Assigned Folder: client 5197 ***
Insert command (list, get <file_name>, put <file_name>, quit):
```

### III. Utilizzo del comando *list*



```
Server
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./server
Client successfully connected
Assigned ports 5194 - 5195
Client successfully connected
Assigned ports 5196 - 5197

Client
marco@marco-VirtualBox:~/Scrivania/Progetto_IIW$ ./client 127.0.0.1 10 0 1 200
*** Assigned Folder: client 5195 ***
Insert command (list, get <file_name>, put <file_name>, quit):
list
||||| 100.00%
Download completed
100MB.zip
10MB.zip
20MB.zip
50MB.zip
5MB.zip
list.txt
*** Assigned Folder: client 5197 ***
Insert command (list, get <file_name>, put <file_name>, quit):
```

## 5 Test prestazionali

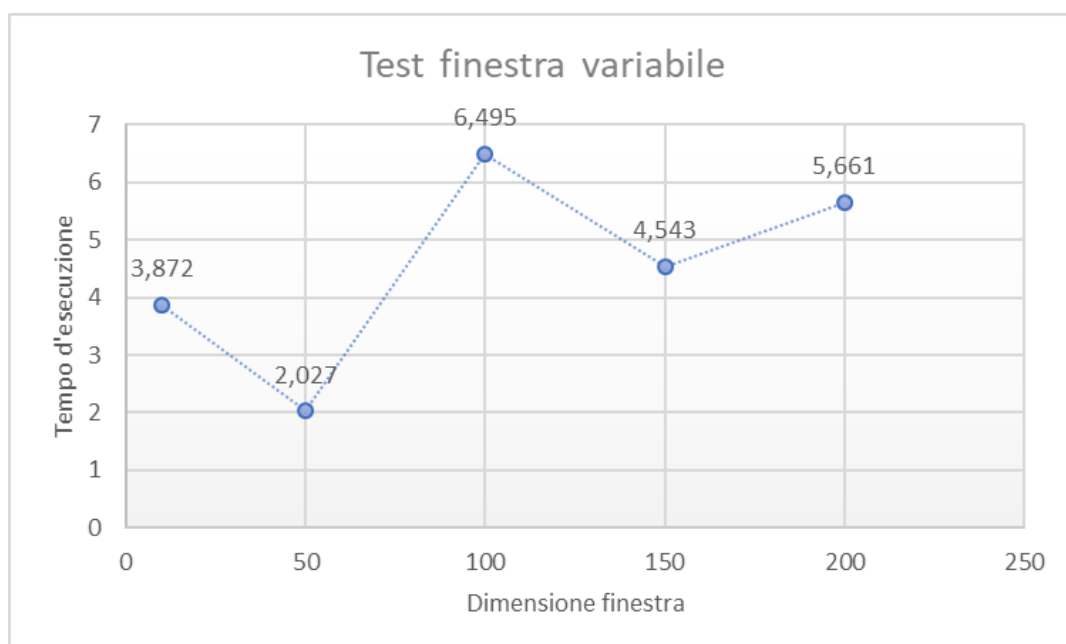
I Test eseguiti sono prettamente prestazionali; sono state, infatti, eseguite molteplici prove per misurare il tempo impiegato dal programma per portare a termine le operazioni scelte dall'utente.

Nelle tabelle di test si è fatto variare un singolo parametro, mantenendo costanti gli altri, in modo tale da analizzare gli effetti di quest'ultimo sui tempi di trasmissione dei dati. I valori variabili sono stati evidenziati in grassetto nelle tabelle e le unità di misura sono state inserite tra parentesi.

### TEST1

Dimensione finestra	Percentuale di perdita del pacchetto	Timer adattivo	Valore timeout (ms)	Dimensione file (MB)	Durata processo (sec)
<b>10</b>	0	1	200	10	<b>3,872</b>
<b>50</b>	0	1	200	10	<b>2,027</b>
<b>100</b>	0	1	200	10	<b>6,495</b>
<b>150</b>	0	1	200	10	<b>4,543</b>
<b>200</b>	0	1	200	10	<b>5,661</b>

Il TEST1 permette di valutare il tempo di trasmissione, al variare della dimensione della finestra. I risultati, come mostrato anche nel grafico seguente, non evidenziano un comportamento lineare al crescere della dimensione della finestra, ma variabile; in particolare, si è rilevata una notevole diminuzione della durata del processo per finestra di dimensione 50.





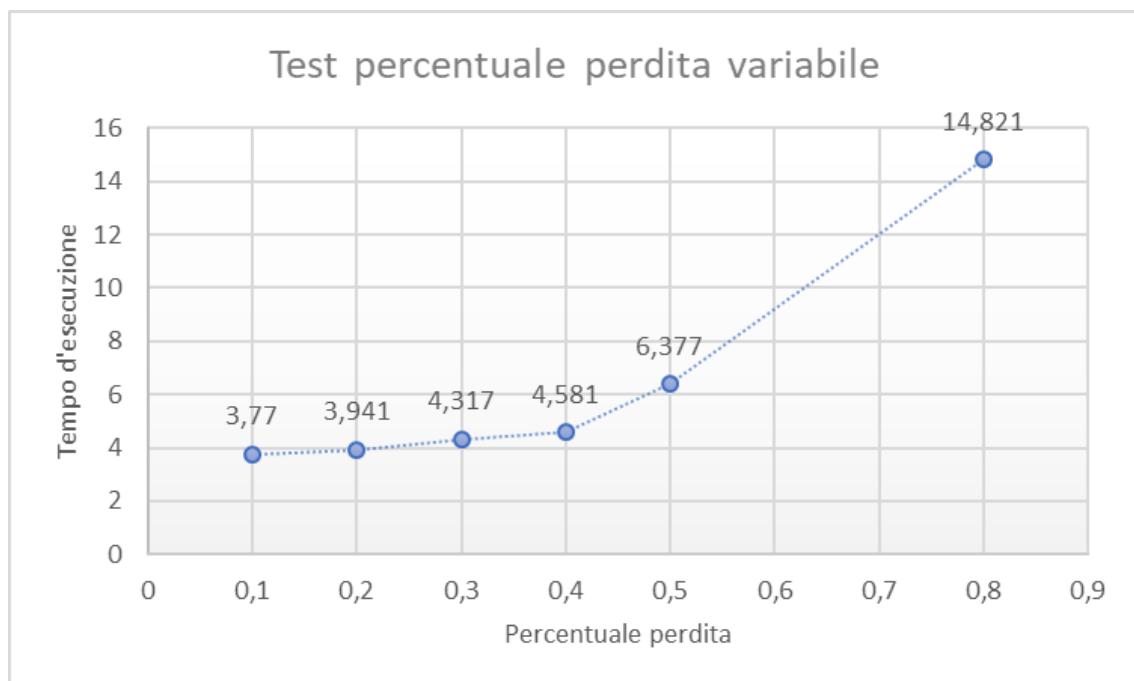
## TEST2

Dimensione finestra	Percentuale perdita del pacchetto	Timer adattivo	Valore timeout (ms)	Dimensione file (MB)	Durata processo (sec)
10	<b>0,1</b>	1	200	10	<b>3,770</b>
10	<b>0,2</b>	1	200	10	<b>3,941</b>
10	<b>0,3</b>	1	200	10	<b>4,317</b>
10	<b>0,4</b>	1	200	10	<b>4,581</b>
10	<b>0,5</b>	1	200	10	<b>6,377</b>
10	<b>0,8</b>	1	200	10	<b>14,821</b>

Il TEST2 ha lo scopo di mostrare la variazione delle prestazioni del programma all'utilizzo di diverse probabilità di perdita.

I dati riportati nella tabella mostrano, come atteso, un andamento crescente del tempo di esecuzione all'aumentare della percentuale. Si può notare, infatti, che, ad esempio, con percentuale di perdita dell'80% (quindi molto elevata), si ha un sensibile incremento nella durata del processo (evidente nel grafico sottostante).

Non è stata, in questo caso, stilata una tabella con timer non adattivo in quanto, già con bassa percentuale di perdita e con valore di timeout costante impostato a 200 ms, l'intervallo di tempo aumenta sensibilmente.

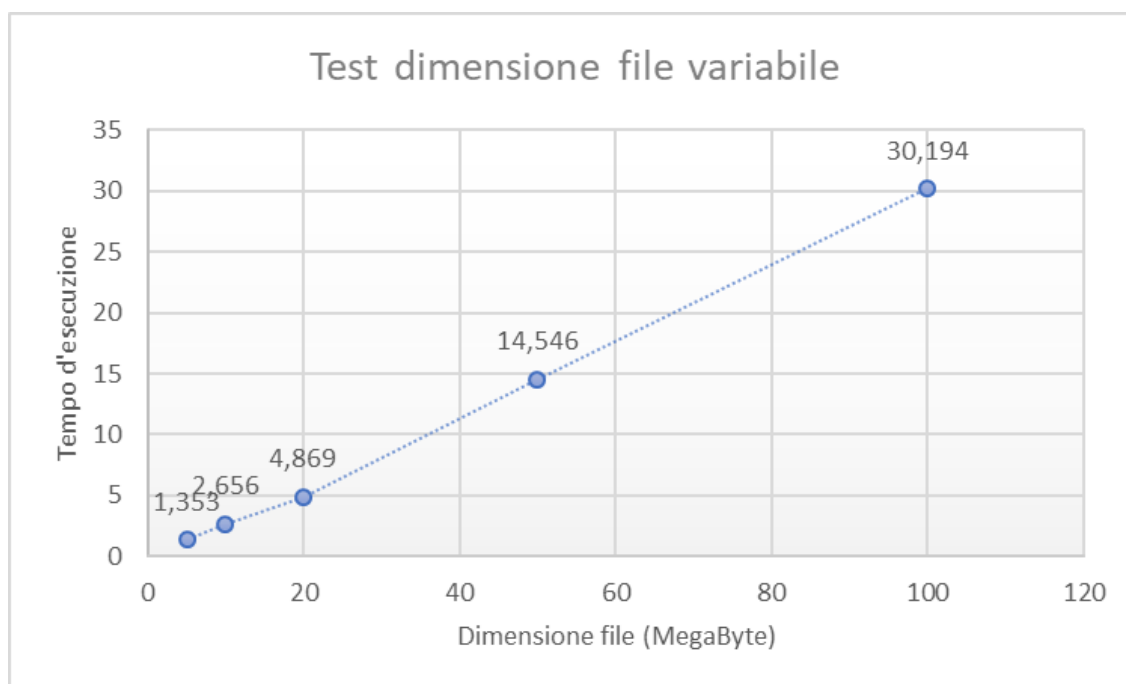


### TEST3

Dimensione finestra	Percentuale perdita del pacchetto	Timer adattivo	Valore timeout (ms)	Dimensione file (MB)	Durata processo (sec)
10	0	1	200	5 MB	1,353
10	0	1	200	10 MB	2,656
10	0	1	200	20 MB	4,869
10	0	1	200	50 MB	14,546
10	0	1	200	100 MB	30,194

Il TEST3 mostra le differenze di prestazioni del programma nel caso in cui vari la dimensione del file.

Come atteso, all'aumentare della dimensione del file, il tempo d'esecuzione del programma cresce linearmente.

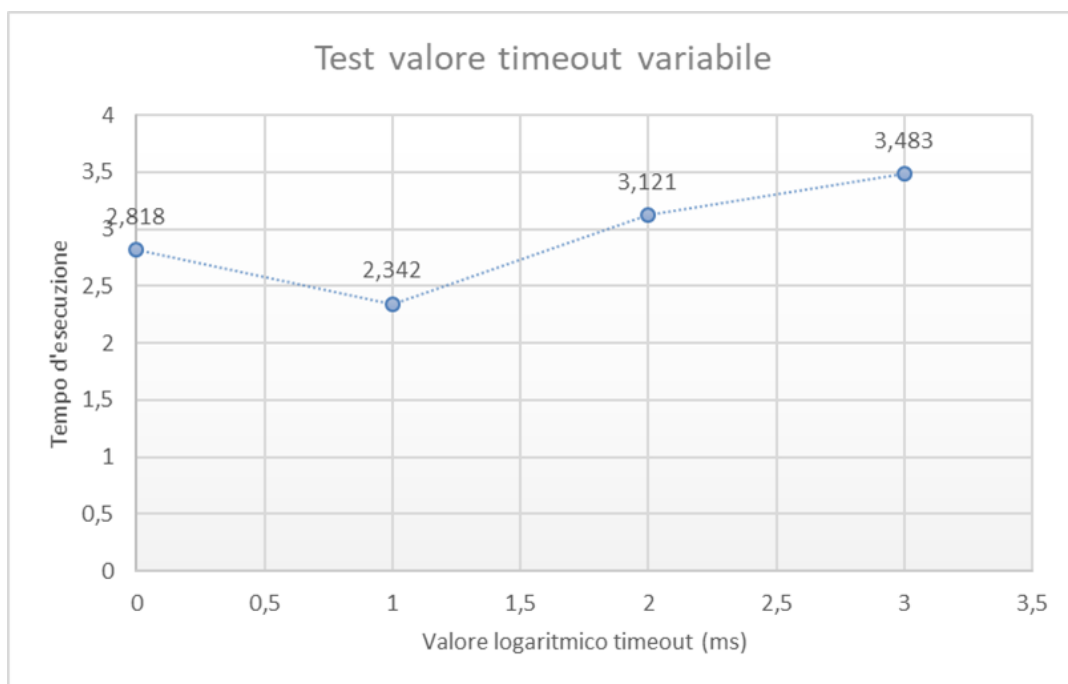


## TEST4

Dimensione finestra	Percentuale perdita del pacchetto	Timer adattivo	Valore timeout (ms)	Dimensione file (MB)	Durata processo (sec)
10	0	1	<b>1</b>	10	<b>2,818</b>
10	0	1	<b>10</b>	10	<b>2,342</b>
10	0	1	<b>100</b>	10	<b>3,121</b>
10	0	1	<b>1000</b>	10	<b>3,483</b>

Il TEST4 permette di osservare i diversi risultati riscontrati in seguito alle variazioni del valore di timeout. In questo caso, all'aumentare del valore di timeout, la durata del processo si mantiene approssimativamente costante.

Ciò risulta coerente alle aspettative in quanto, con timer adattivo, all'arrivo del primo ACK viene immediatamente calcolato il valore del timeout in base all' RTT. Risulta, quindi, quasi ininfluente il timeout di default.



Nei test visti in precedenza, si è preferito considerare solo il caso di timer adattivo in quanto ritenuto più importante in termini prestazionali. Il caso di timer non adattivo, infatti, avrebbe aumentato, anche in maniera significativa, i tempi di attesa.

I file di prova considerati, e su cui sono stati eseguiti i test, sono contenuti nella cartella di progetto e sono stati selezionati di dimensioni variabili per fare in modo che i test fossero il più possibile rilevanti. Il programma è stato provato, oltre che con tali file, anche con altri di varie tipologie: sono stati usati, infatti, file di testo, immagine, video e audio.

Oltre i test di tipo prestazionale, descritti in precedenza, per verificare la correttezza dei dati inviati, è stato utilizzato il comando sha256sum che restituisce un hash univoco del file.

I test sono stati eseguiti su una macchina virtuale con versione Ubuntu 18.10. Le caratteristiche della macchina virtuale sono: 6 core del processore (I7 8th gen.), 10 giga di RAM. Il programma è stato, inoltre, lanciato su una macchina virtuale con versione Xubuntu 16.04.3 LTS.

## 6 Bibliografia

- Reti di calcolatori e Internet – James F. Kurose, Keith W. Ross
- The Linux programming interface – Michael Kerrisk
- <https://stackoverflow.com/>