Assignment #3: Relazione

PROGRAMMAZIONE DI RETI

BAIARDI MARTINA
LOMBARDINI ALESSANDRO

TASK 1

Descrizione della FSM del Server

Il server attende nello stato *wait_connection* una richiesta di connessione da parte di un client sulla porta designata. A seguito di una richiesta, dopo aver instaurato una connessione, il server passa allo stato *wait_hello_message*, dove attende che venga inviato un hello message dal client.

Quando il client invia il messaggio, il server, a seguito della ricezione, ne controlla la validità:

1. Se l'esito risulta **positivo,** invia un messaggio di conferma al client e passa allo stato *wait_probe_message*, dove attende i messaggi di probe da parte del client.
2. Se l'esito risulta **negativo**, invia un messaggio di errore al client e chiude la connessione. Lo stato del server ritorna quello iniziale (*wait_connection)*.

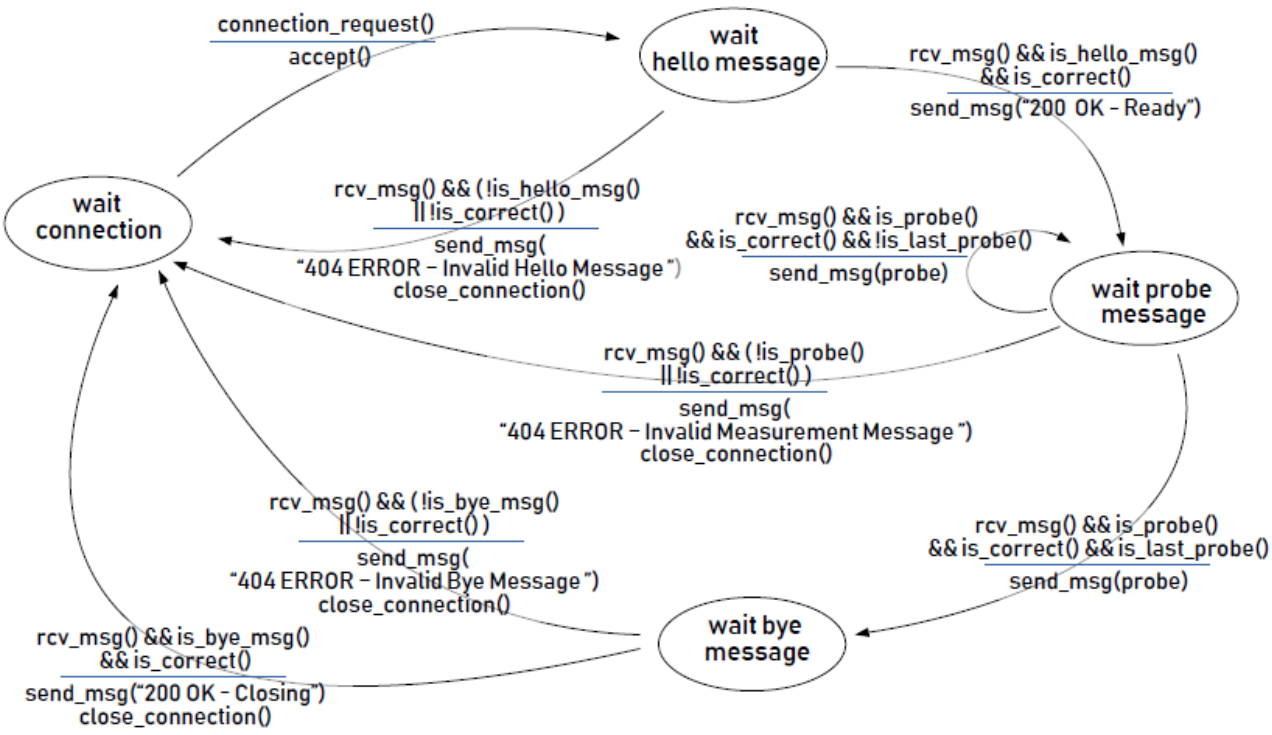
Nello stato *wait_probe_message*, all'arrivo di un nuovo messaggio, il server ne controlla la validità:

1. Se l'esito risulta **positivo,** invia indietro il messaggio ricevuto e attende di ricevere i messaggi di probe successivi;
2. Se l'esito risulta **negativo**, invia un messaggio di errore al client e chiude la connessione. Lo stato del server ritorna quello iniziale (*wait_connection)*.

Lo stato *wait_probe_message* non cambia finché il messaggio ricevuto, oltre ad essere valido, non è l'ultimo fra quelli previsti, ovvero non ne sono stati ancora inviati un numero pari alla quantità specificata nell'*hello message*. Una volta arrivati tutti i messaggi di probe validi, lo stato cambia in *wait_bye_message*, ovvero il server si mette in attesa del *bye message*. Quando il server riceve un messaggio, controlla che questo sia effettivamente il *bye message* atteso:

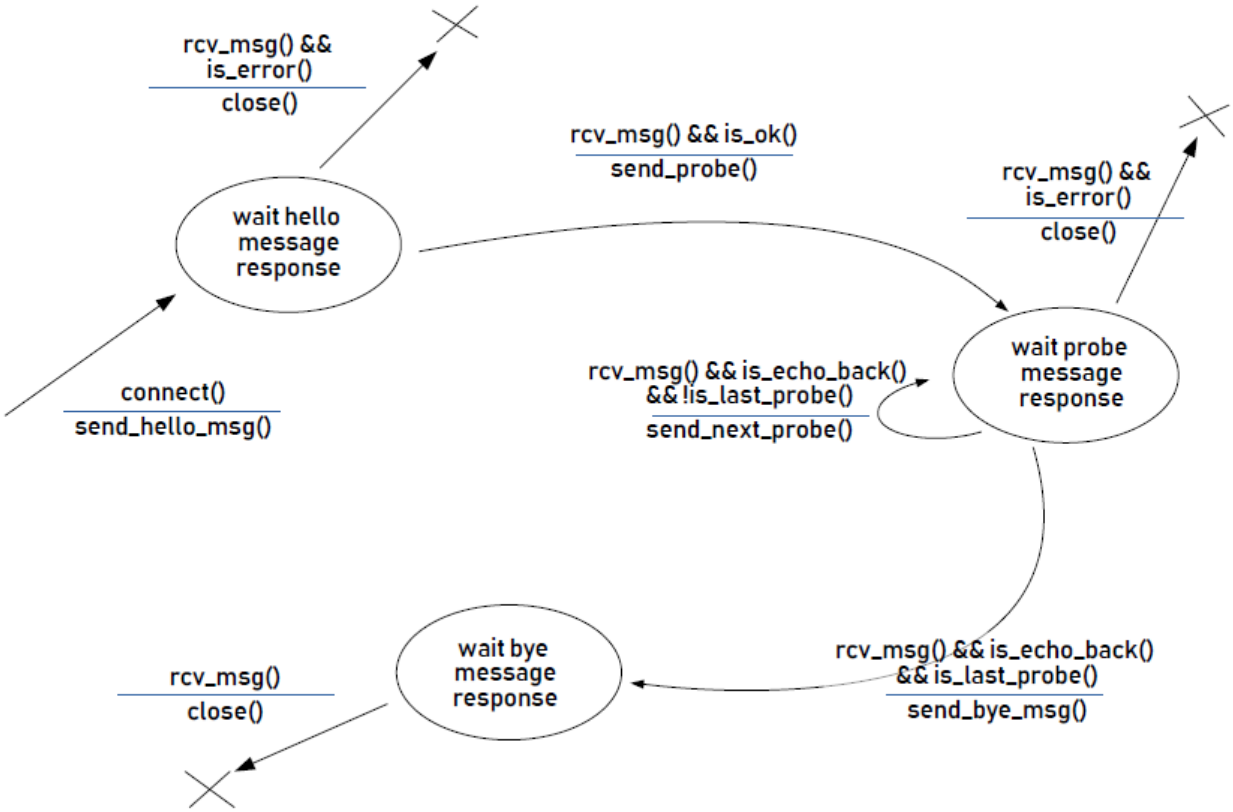
1. Se l'esito è **positivo**, invia un messaggio di conferma al client.
2. Se l'esito è **negativo**, invia al client un messaggio di errore.

Una volta inviata la risposta, il server procede con la chiusura della connessione e torna in ascolto di nuove richieste nello stato *wait_connection*.



Descrizione della FSM del Client

Il client instaura una connessione con il server e invia un *hello message*, posizionandosi nello stato di *wait_hello_message_response*. Una risposta **negativa** implica un errore, dunque chiude la connessione e l'applicativo termina; una risposta **positiva** prevede lo spostamento nello stato di *wait_probe_message_response* e il conseguente invio di un primo *probe message*. Se la risposta coincide con il messaggio inviato, il client può procedere con l'invio del probe message successivo, altrimenti viene terminato l'applicativo. Non appena sono stati inviati un numero di *probe message* pari al valore indicato nell'*hello message*, e per ciascuno quali ricevuto un echo back valido da parte del server, il client può cambiare stato in *wait_bye_message response*, inviando un *bye_message* al server. In caso arrivi un messaggio di conferma viene chiusa la connessione con successo; nel caso in cui il messaggio sia invece di errore, viene terminato il programma con un errore, chiudendo anche in questo caso la connessione.



TASK 2

Client

Il client si preoccupa di eseguire la richiesta per instaurare una connessione TCP con il server. Devono essere dunque forniti, all'atto dell'esecuzione, l'indirizzo IP e la porta su cui il server è in ascolto.

L'utente, per esprimere le proprie preferenze riguardo il servizio che vuole utilizzare, può modificare il file *init.conf*. Al suo interno devono essere presenti i dettagli riguardanti la misura da effettuare. Il formato della stringa contenuta al suo interno deve essere:

<measure_type><sp><n_probes><sp><msg_size><sp><server_delay>

measure_type: il tipo di misura che su vuole effettuare; può essere ***thput*** o ***rtt***.
n_probes: il numero di probe che desidera vengano inviati al server.
msg_size: dimensione del payload dei messaggi di probe che vengono inviati al server (in byte).
server_delay: tempo di attesa che il server deve attendere prima di rispondere ai probe message del client (in millisecondi)

Il client legge queste informazioni e ne valuta la validità, ovvero la possibilità di utilizzarle al fine di realizzare una misurazione: se corrette, le estrae popolando una struttura dati interna, la quale viene tenuta aggiornata durante le diverse fasi d'esecuzione, se errate, il client termina.

La struttura dati è la seguente:

```
/* Service structure definition */
typedef struct node {
  char measureType[6];    /* Service requested - RTT or THPUT */
  int nProbes;            /* Number of probe messages to send */
  int messageSize;        /* Size of probe message's payload */
  int serverDelay;        /* Delay of server probe message echo (in milliseconds) */
  int serverFD;           /* File descriptor of socket */
  int phaseNumber;        /* 1: Hello phase
                             2: Probe phase
                             3: Bye phase */
} serviceNode;
serviceNode service;
```

Dove *phaseNumber* è una rappresentazione numerica delle fasi in cui si trova il client attualmente.

Se il contenuto del file è valido, il client esegue una richiesta di connessione al server e invia un *hello message,* entrando poi in attesa della risposta da parte del server. Se la risposta è positiva si prosegue con l'invio dei probe message, altrimenti si chiude direttamente l'applicativo segnalando l'errore. Il *probe message* viene ricreato ad ogni ciclo di invio/ricezione, incrementando il suo **sequence number**, per permettere di far sapere al server lo stato di avanzamento dell'invio dei probe message. Il payload invece, il quale non è altro che una sequenza dello stesso carattere, viene creato della dimensione desiderata e riutilizzato ad ogni invio di probe.

A seguito dell'invio di un probe message viene fatto partire un timer, il quale viene stoppato non appena si riceve una risposta da parte del server. Se la risposta corrisponde esattamente al probe message inviato viene valutato il RTT e stampato (in millisecondi); se invece il messaggio di risposta non corrisponde, viene chiusa la connessione e terminato l'applicativo. Una volta inviati tutti i *probe message* viene calcolata una media dei loro RTT. Se nell'*hello message* è stata richiesta la misura del RTT, questo viene direttamente stampato; se invece il client ha richiesto un calcolo del *throughput*, questo viene calcolato dividendo la dimensione dei pacchetti inviati per la media dei RTT precedentemente calcolata effettuando le opportune conversioni (viene fornito in kilobits per secondo). Come dimensione dei pacchetti viene presa quella relativa all'ultimo messaggio inviato; non viene tenuto conto del fatto che diversi numeri di sequenza comportano una diversa lunghezza del messaggio.

Infine, viene inviato il *bye message* al server e se ne attende la risposta. Una volta ricevuta, sia nel caso in cui questa sia positiva o negativa, la connessione viene chiusa e l'esecuzione dell'applicativo terminata.

Server

Il server è sempre in attesa di connessioni TCP sulla porta indicata dall'utente in fase d'avvio. Possiede a sua volta una struttura dati simile a quella del client, con l'aggiunta di un campo.

```
/* Service structure definition */
typedef struct node {
   char measureType[6];          /* Service requested - RTT or THPUT */
   int nProbes;                  /* Number of probe messages to send */
   int messageSize;              /* Size of probe message's payload */
   int serverDelay;              /* Delay of server probe message echo (in milliseconds) */
   int connectionFD;             /* File descriptor of socket */
   int phaseNumber;              /* 0: Ready to accept connection |
                                    1: Ready to accept service request |
                                    2: Ready to echo back |
                                    3: Ready response to bye */
   int probeSequenceNumberAwaited;       /* Probe sequence number awaited: useful in phase 2 */
} serviceNode;
serviceNode service;
```

Il campo *probeSequenceNumberAwaited* contiene il valore del sequence number che il server si aspetta di ricevere nel prossimo probe message. Se non corrisponde con il valore contenuto nel messaggio ricevuto viene inviato un messaggio di

errore al client. Le informazioni della struttura dati vengono inizializzate nel momento in cui il server riceve un *hello message* da parte di un client, in quando contiene tutte le informazioni relative al servizio richiesto. Esistono tre pattern di messaggio diversi che il server può accettare: uno per l'*hello message*, uno per il *probe message* e uno per il *bye message*. In funzione del valore di *phaseNumber* il server verifica che il messaggio arrivato soddisfi il pattern relativo allo stato in cui si trova.

Una volta stabilita una connessione con un client, si passa allo stato di attesa di ricezione di un *hello message*. Una volta ricevuto, si valuta la sua sintassi e vengono memorizzati i valori dei suoi campi nell'apposita struttura dati. Se il controllo restituisce esito positivo viene inviato un messaggio di conferma al client e viene modificato lo stato, passando in quello di attesa di *probe message*, nel caso in cui invece l'esito sia negativo viene inviato al client un messaggio d'errore e la connessione viene chiusa, tornando allo stato iniziale.

Arrivato un messaggio di probe, questo viene controllato che sia come previsto: se il messaggio non soddisfa il pattern viene segnalato al client, la connessione viene chiusa e il server torna in attesa di una richiesta di connessione, se invece questo soddisfa i criteri del server, tra cui la sequenzialità del sequence_number, viene inviato al client lo stesso messaggio ricevuto. Una volta terminati i *probe message* previsti, si passa all'ultimo stato: l'attesa di un *bye message*.

Anche per il *bye message* si effettua un controllo quando viene ricevuto, e viene restituito al client un messaggio per fa sapere l'esito della valutazione del server, in questo caso viene in ogni caso chiusa la connessione.

Note

- Per gestire la ricezione di messaggi di dimensione variabile, abbiamo deciso di sfruttare l'allocazione dinamica: in questo modo possono essere ricevuti pacchetti di qualsiasi dimensione. Viene accettato, come valore di payload, un qualsiasi valore positivo maggiore di 0: dati i limiti di memoria assegnata ai processi, non ne è garantito il funzionamento per ordini di grandezza eccessivi.
- Entrambi gli applicativi prevedono, per la ricezione di un singolo messaggio, la possibilità di più letture da buffer e meccanismi per ricostruirlo pezzo per pezzo. Eventuali ritardi del protocollo TCP nella consegna di un pezzo di messaggio non creano quindi problemi. Il termine di un messaggio è definito da un carattere *\n,* se non presente entrambi gli applicativi non sono in grado di terminare la lettura da socket.
- Per 1K abbiamo inteso 1000 byte, e non 1024.
- Anche se non citato nelle specifiche, abbiamo preferito introdurre un messaggio di errore da parte del server per la *bye phase*. In questo modo il client, nel caso di *bye message* errato, può terminare con la consapevolezza che è avvenuto un errore lato server.
- Non sono previsti timer: se una delle due parti dovesse terminare la propria esecuzione nel mezzo di uno scambio di messaggi, l'altro capo non ha modo di terminare la propria esecuzione autonomamente o, nel caso del server, ritornare nello stato di attesa iniziale.
- Se il client invia un probe message in più, lato server questo viene trattato come *bye message* errato, in quanto quest'ultimo si trovava in attesa di un *bye message*.

Grafici delle misure ottenute

Le misure RTT presentano un andamento crescente rispetto all'incremento del numero di byte inviati. Le variazioni rimangono nell'ordine di grandezza di frazioni di millisecondo in quanto gli incrementi del numero di byte del payload non sono sufficientemente grandi per apprezzare differenze nell'ordine dei millisecondi.

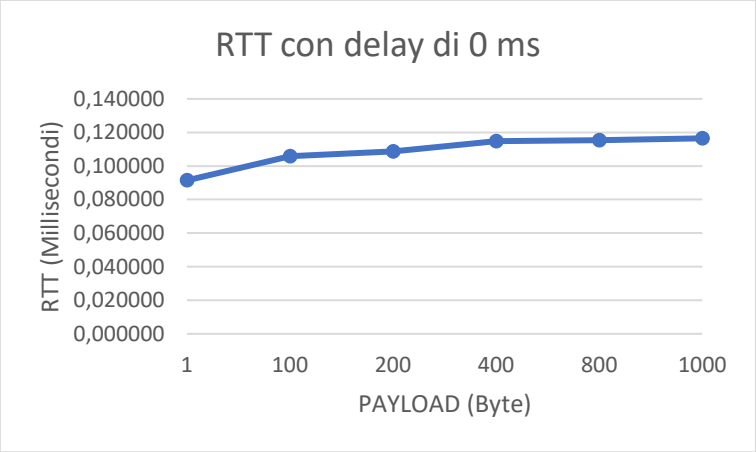
Il throughput tende invece a raddoppiare ogni volta che viene raddoppiato il numero di byte inviati; essendo il RTT poco suscettibile al variare di piccole quantità di byte, la divisione del calcolo del througput tende, al raddoppiarsi del numeratore, a restituire il doppio del valore precedente. È dunque proporzionale all'aumento del numero di byte inviati.

La misura deve ovviamente tenere conto del fatto che è stata realizzata utilizzando due processi in esecuzione sulla stessa macchina virtuale; di conseguenza i RTT analizzati risultano estremamente piccoli rispetto ad una comunicazione TCP nella rete Internet. Trattandosi di variazioni estremamente piccole, dipendono tutte estremamente dal sistema operativo e dallo scheduling dei suoi processi in esecuzione, infatti è possibile che si verifichino, all'aumentare dei byte, risultati completamente differenti dalle aspettative.

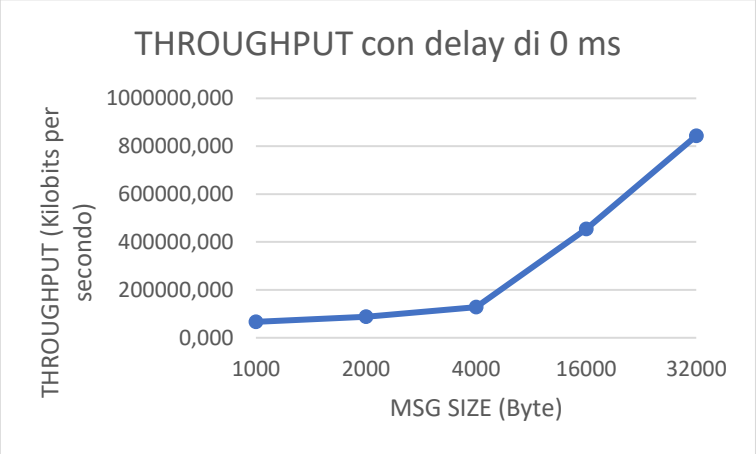
La tendenza generale dell'andamento del throughput e del RTT, tenendo conto delle condizioni descritte, viene evidenziata negli schemi sottostanti.

SERVER_DELAY := 0 ms

PAYLOAD (byte)	1	100	200	400	800	1000
RTT(millisecondi)	0,091450	0,105900	0,108650	0,114750	0,115350	0,116500

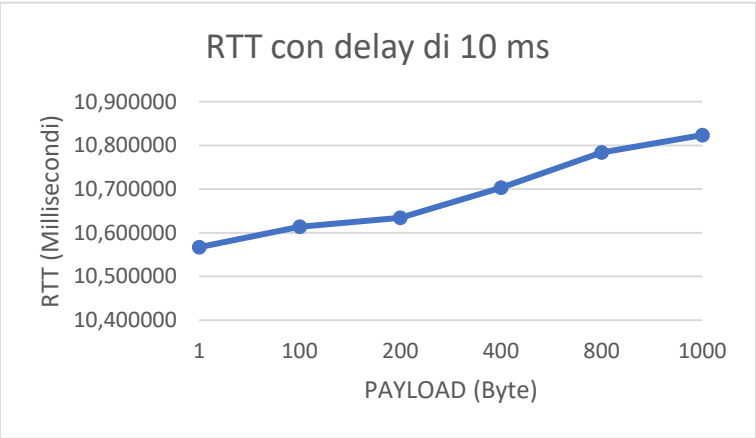


MSG SIZE (byte)	1000	2000	4000	16000	32000
THPUT(kbps)	66567,410	88394,382	127427,437	454635,218	843234,062

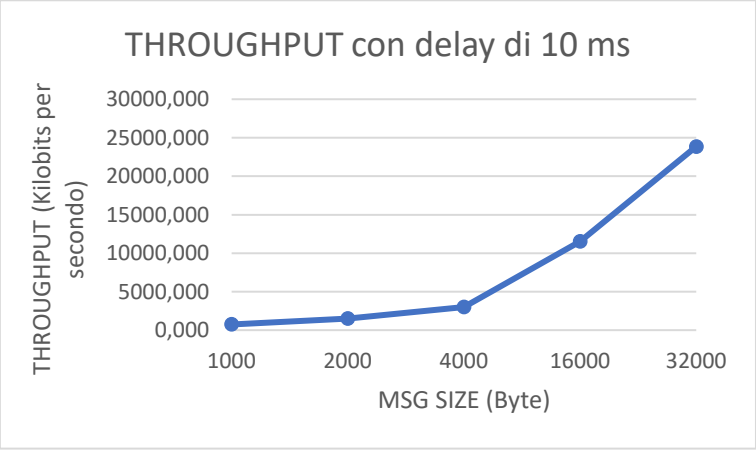


SERVER_DELAY := 10 ms

PAYLOAD (byte)	1	100	200	400	800	1000
RTT(milliseconds)	10,567000	10,614000	10,634500	10,702700	10,783700	10,823400

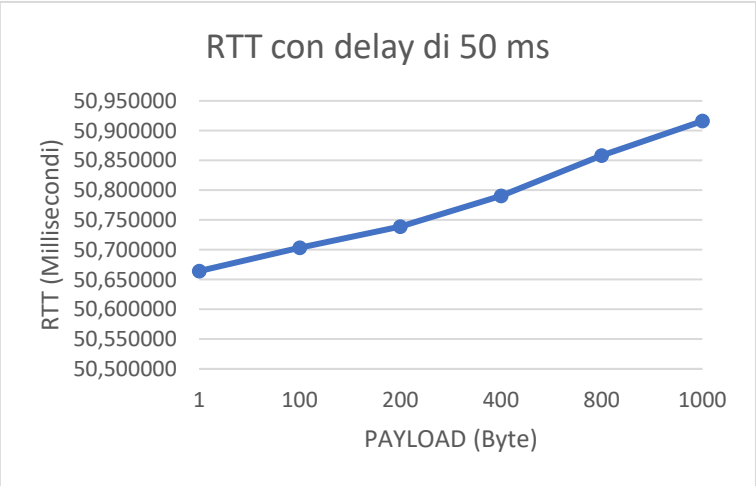


MSG SIZE (byte)	1000	2000	4000	16000	32000
THPUT(kbps)	737,390	1504,580	2999,090	11512,933	23858,700

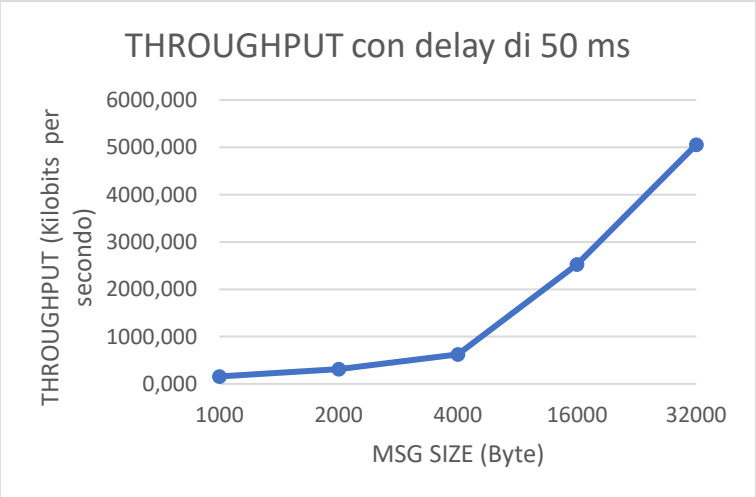


SERVER_DELAY := 50 ms

PAYLOAD (byte)	1	100	200	400	800	1000
RTT(milliseconds)	50,664249	50,703400	50,738700	50,790800	50,858100	50,916100

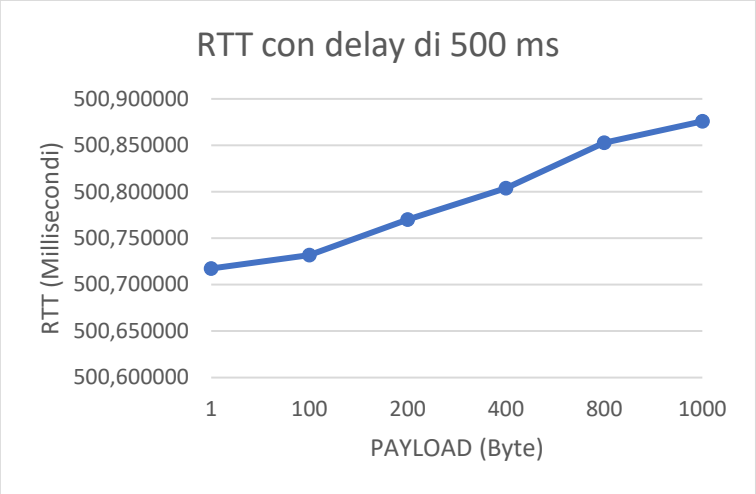


MSG SIZE (byte)	1000	2000	4000	16000	32000
THPUT(kbps)	157,980	312,826	620,914	2525,588	5056,864

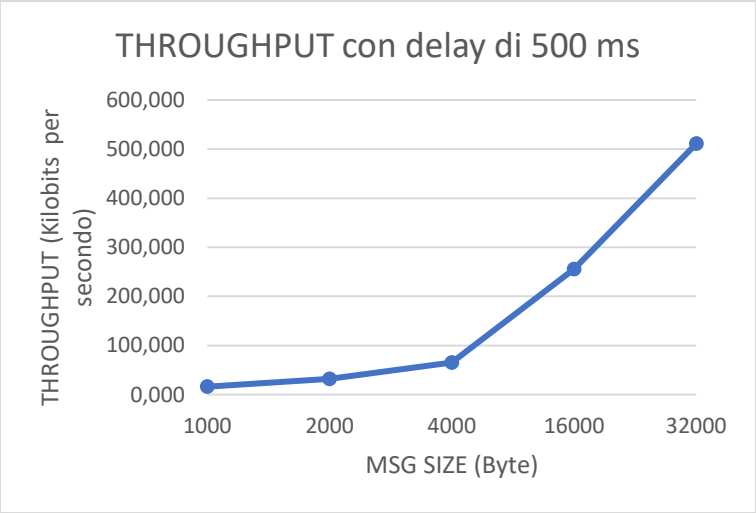


SERVER_DELAY := 500 ms

PAYLOAD (byte)	1	100	200	400	800	1000
RTT(milliseconds)	500,717419	500,731692	500,769900	500,803900	500,852800	500,875800



MSG SIZE (byte)	1000	2000	4000	16000	32000
THPUT(kbps)	16,073	32,031	64,949	255,439	510,990



Dai grafici con delay è possibile notare come l'RTT e throughput mantengono un andamento simile al caso inziale ma con ordine di grandezza diverso. Tanto più è alto il delay tanto più l'ordine di grandezza del throughput tende a diminuire, subendo infatti, a parità di numeratore, un aumento consistente del denominatore. L'RTT, scontatamente, mantiene l'ordine di grandezza del delay.