Ingegneria di Internet e Web – A.A. 2018/19

Progetto B: trasferimento file su UDP

Marco Paesano (0175163)

Ingegneria Informatica

Indice

Requisiti del progetto

Lo scopo del progetto è quello di implementare in linguaggio C POSIX un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione, cioè che utilizzi UDP come protocollo di trasporto.

Tra i requisiti dell'applicazione vi sono:

- connessione client-server senza autenticazione
- comando *list* per la visualizzazione sul client dei file disponibili sul server
- comando *get* per il download di un file dal server
- comando *put* per l'upload di un file sul server
- trasferimento affidabile dei file

La trasmissione affidabile dei file deve avvenire utilizzando il protocollo *selective repeat* con finestra di spedizione N.

La durata dei timeout per la ritrasmissione devono poter essere fissi o, a scelta dell'utente, adattivi e calcolati dinamicamente in base all'evoluzione dei ritardi di rete.

La perdita dei messaggi in rete deve essere simulata assumendo che il mittente scarti un messaggio con probabilità P.

La comunicazione tra client e server deve avvenire tramite messaggi di comando, inviati dal client verso il server per la richiesta di un servizio, e messaggi di risposta, inviati dal server verso il client come risposta ad una richiesta.

Descrizione dell’architettura del sistema e delle scelte progettuali

L’applicazione si compone di un server ed uno o più client.

Il server, una volta mandato in esecuzione, rimarrà in ascolto per un’eventuale richiesta di connessione da parte di un client. Una volta arrivata la richiesta, il server procederà ad aprire la connessione completando con il client un handshake a tre vie, dopodiché rimarrà in attesa di ricevere messaggi di comando da parte del client.

Ogni client avrà la propria connessione verso il server e quest’ultimo potrà servire le richieste in modo concorrente.

Il client, una volta mandato in esecuzione, cercherà di contattare il server per effettuare l’handshake ed aprire così la connessione. Nel caso in cui non dovesse riuscire a contattare il server per un tempo di 60 secondi, il client terminerà la sua esecuzione.
Una volta stabilita la connessione, il client sarà pronto a inviare messaggi di comando verso il server.

All’apertura della connessione, sia il client che il server inizializzano le proprie strutture dati e in particolare scelgono in modo casuale un numero di sequenza con cui identificare ogni messaggio inviato e ricevuto.

I comandi supportati dal sistema sono:

Nome	Codice	Descrizione
list	1	Richiede al server una lista dei file di cui è possibile fare il download
mylist	-	Stampa a schermo la lista dei file contenuti nella directory del client e di cui è possibile fare l’upload verso il server
get	2	Richiede al server il download di uno specifico file
put	3	Effettua l’upload di uno specifico file verso il server
help	-	Stampa a schermo la lista dei comandi
quit	fin = 1	Chiude la connessione e l’esecuzione del processo chiamante

Si è scelto di non inviare il comando sotto forma di stringa ma come codice numerico per semplificarne il riconoscimento e la gestione.

Dopo l’immissione da parte dell’utente di uno dei comandi, viene creato un apposito messaggio di richiesta che viene spedito al server. Il client si pone quindi in attesa di un messaggio di ack specifico per quel messaggio.
Se questo non arriva entro lo scadere di un timer, il messaggio di richiesta viene ritrasmesso finché non viene ricevuto il relativo ack.

Da notare che si è scelto di mantenere distinti i messaggi contenenti dati e i messaggi di ack, sia per semplificare il sistema, sia perché in ogni momento solo uno tra il client e il server sta inviando dati mentre l’altro li riceve.
Inoltre gli ack non sono cumulativi ma sono specifici per un dato messaggio, così da poter ritrasmettere, nel caso di perdita, unicamente il messaggio che è andato perso.

Il server, dopo aver ricevuto il messaggio di richiesta e inviato il relativo ack, prepara i dati richiesti o si prepara a riceverne a seconda del tipo di comando.

Nel primo caso, la lista o il file richiesto vengono suddivisi in chunks di dimensione non superiore alla massima dimensione del payload di un singolo messaggio. Questi chunks vengono quindi incapsulati all’interno di messaggi, che vengono inseriti in una coda di invio. Da questa coda vengono quindi selezionati i messaggi da inviare.
In un dato momento, durante la fase di invio, vi saranno al più N messaggi in volo in attesa di ricevere il proprio ack. Finché il messaggio inviato con numero di sequenza minore non avrà ricevuto l’ack, nessun altro messaggio potrà essere inviato, cioè la finestra di spedizione non potrà “scorrere”.
Ad ogni messaggio in volo viene associato un timer che alla scadenza provoca una ritrasmissione. Il timer può essere sia fisso che adattivo e calcolato dinamicamente in base ai ritardi di rete.

Alla ritrasmissione di un messaggio il valore del timer viene raddoppiato, il ché fornisce un rudimentale controllo della congestione così come avviene per il protocollo TCP. Si è scelto però di limitare di default a 2s il valore massimo per i timer, così da non rendere eccessivamente lunga l’esecuzione nel caso di valori di P particolarmente alti.

Nel caso in cui il file richiesto non esista, il server invierà un unico messaggio con un codice di errore numerico.

Il client, nel frattempo, sarà in attesa di ricevere messaggi. Appena ricevuto un messaggio, se non si tratta di un ack, verrà immediatamente inviato un apposito messaggio di ack in risposta. Se il messaggio ricevuto non è una vecchia ritrasmissione allora verrà inserito in una coda di ricezione.

I messaggi all’interno della coda di ricezione vengono serviti solo se sono nel corretto ordine rispetto ai numeri di sequenza, altrimenti si aspetta che arrivino i messaggi necessari a colmare i “buchi” nella sequenza.

Una volta ricevuto l’ultimo messaggio, il file richiesto sarà completo e verrà stampato un messaggio a schermo o, nel caso sia stata richiesta la lista dei file, verrà stampata la lista stessa.
Nel caso in cui sia il client a voler inviare un file, come prima cosa viene inviato un messaggio contenente il tipo di comando e il nome del file di cui si vuole fare l’upload, in modo che il server possa crearne un’esatta copia nella sua directory. Dopodiché il client suddividerà il file in chunks e li invierà con le stesse modalità descritte per il server. Quest’ultimo riceverà i messaggi, che saranno inseriti in una coda di ricezione, e invierà i relativi ack.
Una volta ricevuto correttamente il file, il server invierà un messaggio con un codice a conferma del successo, o meno, dell’operazione.

Infine, se l’utente immette il comando *quit*, il client invierà un messaggio di chiusura della connessione. Il server risponderà con un ack e poi a sua volta con un messaggio di chiusura.
Il client, una volta ricevuto la conferma di chiusura da parte del server, aspetterà per 30 secondi prima di terminare l’esecuzione.
Il server, una volta ricevuto l’ultimo ack, chiuderà la connessione con quel particolare client e rimarrà in attesa di aprire ulteriori connessioni con altri client.

Descrizione dell’implementazione

I messaggi sono stati implementati come *structure* avente i seguenti campi:

Nome	Tipo	Descrizione
syn	char	Flag per l’inizializzazione di una connessione
ack	char	Flag per indicare che il messaggio è un ack per il messaggio con numero di sequenza *ack_num*
fin	char	Flag per comunicare la fine della connessione
startfile	char	Flag per indicare che il messaggio è il primo della sequenza
endfile	char	Flag per indicare che il messaggio è l’ultimo della sequenza
cmd_t	char	Flag per indicare il tipo di comando a cui il messaggio è legato
ecode	enum	Flag che può assumere i valori {*success = 1, clierror = 2, serverror = 3*} che indicano rispettivamente un’operazione terminata con successo o un errore lato client/server
seq	unsigned long	Numero di sequenza associato al messaggio
ack_num	unsigned long	Numero di sequenza del messaggio a cui è indirizzato l’ack
data_size	unsigned long	Dimensione dei dati contenuti nel messaggio, al massimo pari a PAYLOAD_SIZE
file_size	unsigned long	Dimensione totale del file che si vuole inviare
data	array di char	Payload del messaggio

I messaggi da inviare o ricevuti vengono inseriti rispettivamente in una coda di invio o una coda di ricezione, implementate come linked list.
Gli inserimenti all’interno della coda di ricezione avvengono in modo ordinato rispetto al numero di sequenza, così da minimizzare l’arrivo e la gestione non ordinata dei pacchetti, tramite la funzione *insert_sorted*.
Gli inserimenti nella coda di invio avvengono tramite la funzione *append*.

I parametri principali del sistema, tra cui T, P ed N, sono contenuti in *myUDP.h*.

Il client e il server condividono alcune funzioni all'interno di *common.c*, che sono:

Nome	Descrizione
read_line	Legge una stringa da standard input e alloca un buffer per contenerla tramite la funzione *getline*
split_line	Divide una stringa in tokens tramite la funzione *strtok*
reset_msg	Azzera tutti i campi di un messaggio
send_ack	Invia un ACK a destinazione con probabilità (1-P) tramite la funzione *sendto*
print_queue	Stampa a schermo il contenuto della coda di invio/ricezione ed è tipicamente usata nella modalità di debug
insert_sorted	Inserisce un nuovo nodo all'interno della coda di invio/ricezione, ordinato in base al numero di sequenza del messaggio, scartando i duplicati
append	Inserisce un nuovo nodo alla fine della coda di invio/ricezione e ritorna un puntatore all'ultimo elemento
delete_node	Rimuove un nodo dalla coda di invio/ricezione, identificato dal numero di sequenza del messaggio che si vuole eliminare
search_node_by_seq	Ricerca in tempo lineare un nodo all'interno della coda di invio/ricezione, identificato dal numero di sequenza del messaggio
search_node_to_serve	Ricerca in tempo lineare un nodo all'interno della coda di invio/ricezione il cui campo *index* non sia ancora stato impostato
queue_size	Restituisce la grandezza della coda di invio/ricezione
timespec_normalize	Normalizza il valore di una variabile timespec
timespec_from_double	Converte il valore di una variabile double, tipicamente in secondi, in una variabile timespec normalizzata
timespec_to_double	Converte una variabile timespec in una variabile double
timespec_add	Somma due variabili di tipo timespec normalizzate
timespec_sub	Sottrae due variabili di tipo timespec normalizzate
str_cut	Rimuove *len* bytes da una stringa
rand_value	Ritorna un valore casuale compreso tra 0 e 1
sigint_handler	Stampa a schermo un messaggio quando viene ricevuto un segnale di SIGINT

Il server è stato implementato come un programma multi-processo e multi-threaded.

Il processo padre si occupa di rimanere in ascolto per eventuali richieste di connessioni da parte dei client su un socket *listensd*. Alla ricezione di un messaggio di SYN da parte di un client, viene generato un nuovo processo che si occuperà di chiudere il socket di ascolto e gestire la particolare istanza di connessione con quel client.

Il processo figlio genera un numero di sequenza random, generato usando la funzione *rand* con seme pari all'ID del thread, così che connessioni diverse possano avere numeri di sequenza diversi. A quel punto può essere completato l'handshake.

Le funzioni proprie del server sono:

Nome	Descrizione
accept_connection	Usata dal processo padre per ricevere messaggi di SYN. Il processo rimane in attesa di input tramite la funzione *select*. Alla ricezione di un messaggio di SYN viene creato un socket per la connessione e viene inizializzata la variabile *expected_seq*
complete_handshake	Crea un messaggio di SYN-ACK e lo invia al client con probabilità (1-P) finché non viene ricevuto il relativo ACK
send_message	Invia un messaggio generico che non sia un ACK con probabilità (1-P) e gestisce le ritrasmissioni
send_list	Scansiona la directory del server, salva la lista dei file in essa contenuti in un buffer, divide la lista in chunks che vengono incapsulati in messaggi che vengono inseriti nella coda di invio, infine invoca la creazione di N thread per l'invio dei messaggi
send_file	Apre il file richiesto, se esiste, lo divide in chunks che vengono incapsulati in messaggi che vengono inseriti nella coda di invio, infine invoca la creazione di N thread per l'invio dei messaggi. Se il file non esiste invia un unico messaggio con un codice d'errore
recv_msg	Riceve i messaggi che, se non sono ritrasmissioni, vengono inseriti nella coda di ricezione. Inoltre viene invocata la funzione *send_ack*
msg_handler	Serve i messaggi ricevuti. Nel caso di ack, ne viene segnalato l'arrivo così da interrompere ulteriori ritrasmissioni dovute allo scadere del timer. Per i messaggi che non sono ack vengono eseguite operazioni diverse in base al tipo di comando del messaggio

Il processo figlio si divide in vari threads, che lavorano in parallelo e in concorrenza. In particolare, il thread principale si occupa di creare N ulteriori threads per servire i messaggi arrivati tramite la funzione *msg_handler* e di gestire la ricezione dei messaggi attraverso la funzione *recv_msg*. All'occorrenza, cioè quando ci sono messaggi da inviare, vengono creati fino a N threads di invio, che una volta svolto il loro compito tramite la funzione *send_message* terminano l'esecuzione.

La sincronizzazione tra i vari threads avviene tramite un ampio uso di pthread_mutexes:

Nome	**Descrizione**
index_mutex	Per sincronizzare la selezione degli indici tra i sending threads
rec_index_mutex	Per sincronizzare la selezione degli indici tra gli handling threads
mutexes	Per sincronizzare i sending threads e gli handling threads all'arrivo degli ack
acked_mutexes	Per rendere più robusta la segnalazione di arrivo di un ack
rec_mutex	Per sincronizzare l'accesso alla coda di ricezione
snd_mutex	Per sincronizzare l'accesso alla coda di invio
exp_mutex	Per sincronizzare l'aggiornamento della variabile *expected_seq*
wr_mutex	Per sincronizzare la scrittura su file

La segnalazione di determinate condizioni avviene tramite le funzioni *pthread_cond_wait*, *pthread_cond_timedwait*, *pthread_signal* e *pthread_broadcast*.

I timer relativi ad ogni messaggio inviato sono stati implementati attraverso la *pthread_cond_timedwait*, che permette al thread chiamante di attendere che una certa condizione si verifichi finché il timer, rappresentato da una variabile timespec, non scada.

Le condizioni usate sono:

Nome	Descrizione
index_cond	Per segnalare il rilascio di un indice, che è quindi selezionabile da un altro thread di invio
rec_cond	Per segnalare l'arrivo di un messaggio
exp_cond	Per segnalare l'avvenuto aggiornamento della variabile *expected_seq*
sb_cond	Per segnalare l'avvenuto aggiornamento del puntatore *send_base*
ack_cond	Per segnalare ad un thread di invio l'arrivo dell'ack che stava aspettando

Il timeout adattivo è stato implementato così come avviene nel protocollo TCP. Se non ci sono state ritrasmissioni, si calcola un campione del RTT, una stima per il RTT, calcolato come $estimatedRTT = (1-\alpha) * estimatedRTT + \alpha * sampleRTT$, che prende quindi in considerazione la vecchia stima, e infine la variabilità tra il campione e la stima, calcolata come $devRTT = (1-\beta) * devRTT + \beta * |sampleRTT - estimatedRTT|$.
L'intervallo di timeout viene quindi calcolato come:
$TimeoutInterval = EstimatedRTT + 4 * DevRTT$.

La variabile *expected_seq*, inizializzata nella funzione *accept_connection*, indica il numero di sequenza del messaggio che ci si aspetta di ricevere e ha due funzioni fondamentali, serve cioè a distinguere un messaggio che è già stato ricevuto da uno mai ricevuto prima e inoltre permette di distinguere quale messaggio è nel corretto ordine rispetto al flusso di bytes ricevuti. Quando una nuova sequenza di bytes viene ricevuta, il primo messaggio della sequenza, che avrà quindi il flag *startfile* pari a 1, diventerà il receive base e l'handling thread, una volta servito il messaggio, aggiornerà il valore di *expected_seq* al valore pari al numero di sequenza del messaggio servito. Il messaggio che sarà servito successivamente dovrà quindi avere un numero di sequenza pari alla somma di *expected_seq* più la dimensione dei dati trasportati. In questo modo è possibile sequenzializzare l'ordine con cui vengono serviti i messaggi rispettando le caratteristiche del selective repeat.

In modo analogo, per i sending threads esiste un puntatore *send_base* che punta al messaggio con numero di sequenza minore che ancora non ha ricevuto l'ack. In questo modo, finché il thread responsabile dell'invio del messaggio puntato da *send_base* non riceve la segnalazione di arrivo dell'ack relativo, nessun altro thread può inviare messaggi "al di fuori" della finestra di spedizione. Una volta arrivato l'ack, il thread aggiornerà il

puntatore *send_base* così da farlo puntare al messaggio successivo nella sequenza, facendo “scorrere” la finestra.

Sia i sending threads che i receiving threads utilizzano un sistema di indici per la selezione univoca dei messaggi da inviare/servire. Quando un messaggio viene inserito nella coda di invio o nella coda di ricezione, gli viene associato automaticamente un indice pari a -1, che sta a significare che quel particolare messaggio non è ancora stato selezionato da nessun thread. Quando un thread seleziona un messaggio dall’interno di una coda tramite la funzione *search_node_to_serve,* l’indice di quel nodo viene aggiornato con il valore dell’indice associato a quello specifico thread, così che nessun altro thread possa selezionarlo nuovamente.

Il client invece è stato implementato come un programma a singolo processo, multi-threaded.

Le funzioni proprie del client sono:

Nome	**Descrizione**
open_connection	Invia un messaggio di SYN al server, dopodiché rimane in attesa per 60 secondi per una segnalazione (da parte di un handler) dell’arrivo di un messaggio di SYN-ACK
send_message	Invia un messaggio generico che non sia un ACK con probabilità (1-P) e gestisce le ritrasmissioni
send_file	Apre il file richiesto, se esiste, lo divide in chunks che vengono incapsulati in messaggi che vengono inseriti nella coda di invio, infine invoca la creazione di N thread per l’invio dei messaggi
send_cmd	Aspetta di ricevere input da tastiera, che, se risulta essere un comando valido, viene inviato al server
recv_msg	Riceve i messaggi che, se non sono ritrasmissioni, vengono inseriti nella coda di ricezione. Inoltre viene invocata la funzione send_ack
msg_handler	Serve i messaggi ricevuti. Nel caso di ack, ne viene segnalato l’arrivo così da interrompere ulteriori ritrasmissioni dovute allo scadere del timer. Per i messaggi che non sono ack vengono eseguite operazioni diverse in base al tipo di comando del messaggio
print_mylist	Stampa a schermo il contenuto della directory del client

Come il server, anche il client sceglie un numero di sequenza casuale tramite la funzione *rand* e seme dato dall’ID del thread.

Il thread principale del client si occupa di eseguire la funzione *send_cmd*. C'è poi un thread che esegue la funzione *recv_msg*, che si occupa di creare N threads per servire i messaggi arrivati tramite la funzione *msg_handler* e di gestire la ricezione dei messaggi.
Infine, come per il server, quando ci sono messaggi da inviare, vengono creati ulteriori N threads di invio, che una volta svolto il loro compito tramite la funzione *send_message* terminano l'esecuzione.

La gestione dei threads, della sincronizzazione, dell'invio e della ricezione dei messaggi, è pressoché identica a quella implementata per il server.

Il client ha a disposizione alcuni mutexes aggiuntivi:

Nome	Descrizione
open_mutex	Per sincronizzare l'arrivo del messaggio di SYN-ACK
close_mutex	Per sincronizzare l'arrivo del messaggio di FIN
prog_mutex	Per sincronizzare la stampa della barra del progresso

Il client ha inoltre a disposizione alcune condizioni aggiuntive:

Nome	Descrizione
open_cond	Per segnalare l'arrivo del messaggio di SYN-ACK
close_cond	Per segnalare l'arrivo del messaggio di FIN
prog_cond	Per segnalare la fine della stampa della barra del progresso

Sia per il server che per il client, la perdita dei messaggi è stata implementata attraverso l'uso della funzione *rand*. In particolare, per sapere se un messaggio è perso oppure può essere inviato, viene valutata la seguente condizione: *rand (mod RAND_MAX) > P.*

Descrizione delle limitazioni riscontrate

Il sistema implementato si è rivelato funzionante con prestazioni soddisfacenti, considerata la natura didattica del progetto, ma le sue limitazioni in un ipotetico uso reale sono evidenti.

Prima di tutto, il sistema è stato progettato per svolgere particolari funzioni per un particolare tipo di applicazione, il ché non lo rende flessibile ad altri tipi di utilizzo. A questo problema è legato anche l'overhead dovuto ad alcuni campi del protocollo, che in generale sarebbero superflui ma che sono stati mantenuti per rendere più semplice la gestione dei pacchetti.

Inoltre le performance potrebbero essere migliorate. Per esempio si potrebbe implementare un algoritmo più efficiente per la ricerca dei nodi all'interno delle strutture dati utilizzate o direttamente sostituire le strutture dati utilizzate per minimizzare i tempi di ricerca dei nodi.

Potrebbe poi essere ripensata la modalità di invio dei messaggi così da permettere l'invio diretto, senza dover prima inserire i messaggi da inviare in una struttura dati ad-hoc.

È da notare che in tutti i casi, visto che sia il server che il client sono multithreaded, le prestazioni dipendono anche dal particolare scheduling del sistema operativo.

Piattaforma utilizzata per lo sviluppo ed il testing

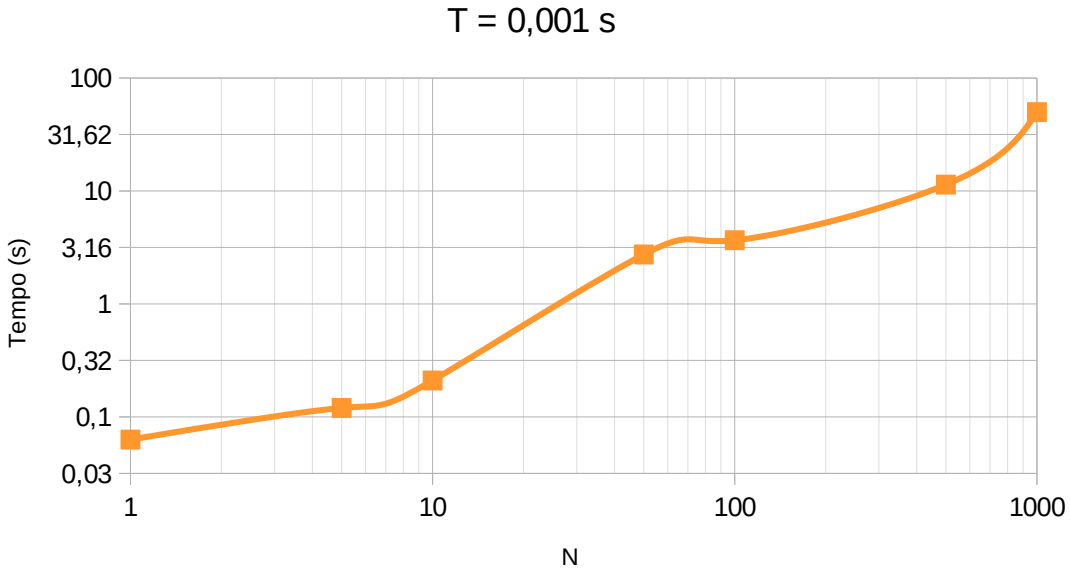
- **OS:**
  - Fedora 31, Kernel Linux 5.6.15
- **IDE:**
  - Vim
- **Hardware:**
  - 4x Intel Core i5-3320M @ 2.60 GHz
  - 16GB di ram @1600 MHz

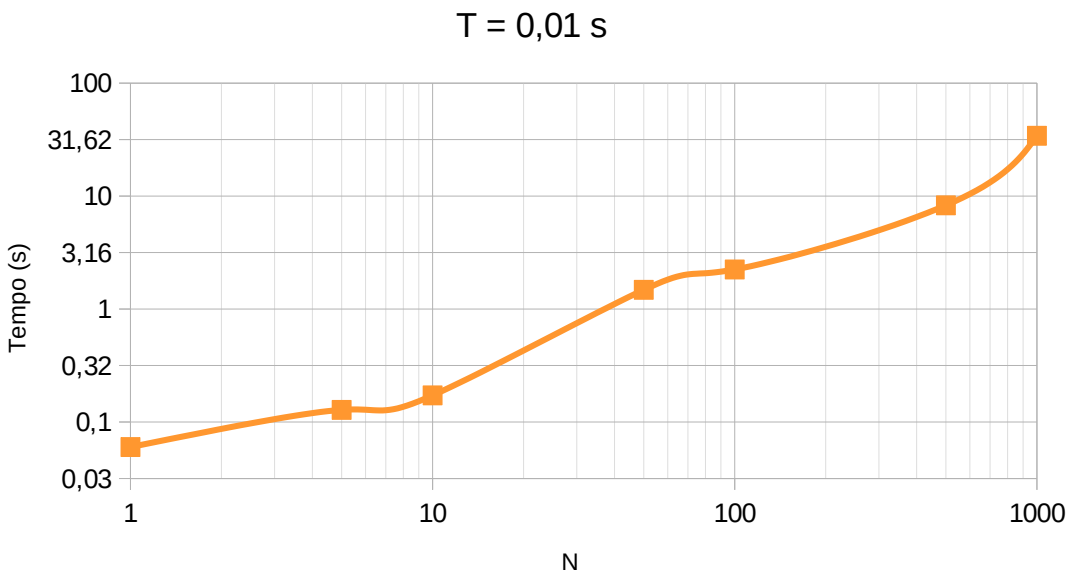
Valutazione delle prestazioni al variare dei parametri

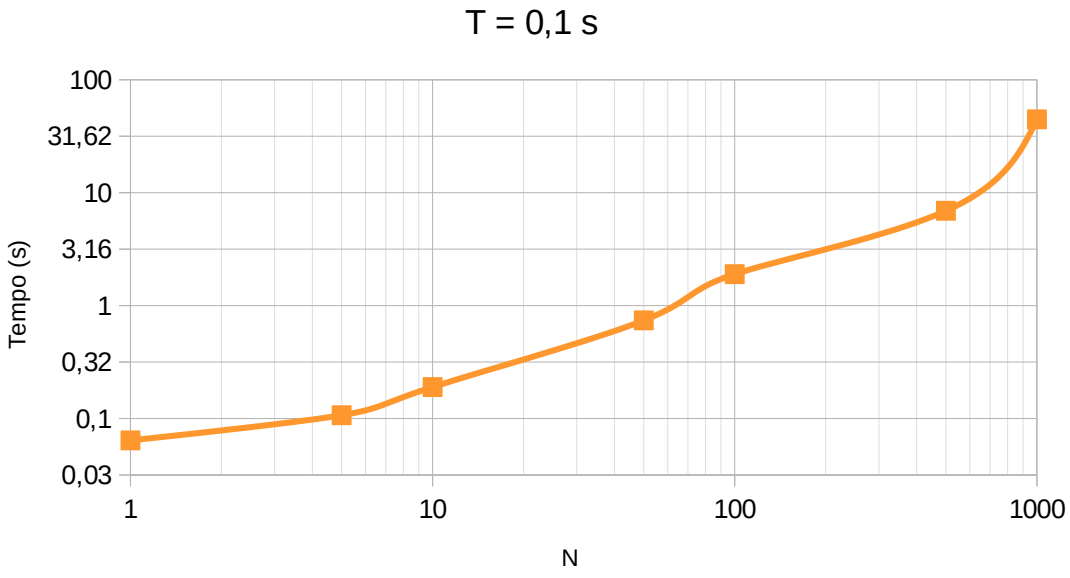
Le prestazioni sono state valutate in termini di tempo trascorso tra l'invio di una richiesta al server e la ricezione dell'ultimo messaggio che completa la transazione.
Per la misurazione dei tempi è stata usata la funzione *clock_gettime* della libreria *time*. In tutti i casi si è utilizzata una dimensione del payload trasportato da ogni pacchetto pari a 512 bytes e una dimensione del file da trasmettere pari a 512KB.
Per ogni test sono state effettuate tre prove con gli stessi parametri e i tempi sono il risultato della media tra i tre tempi registrati.

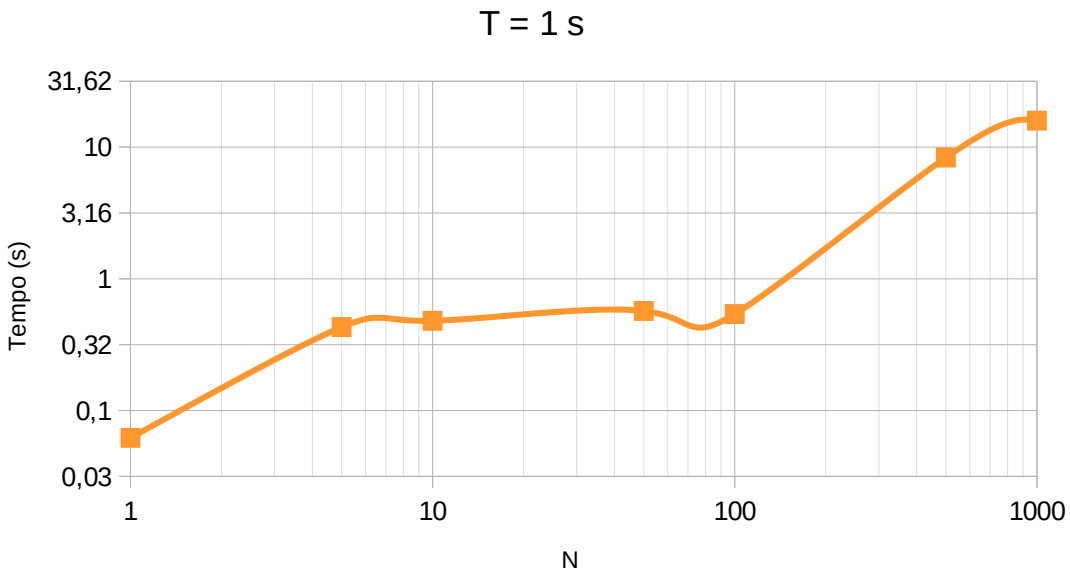
Prestazioni nel caso di timeout fisso (comando *get*)

- P = 0%



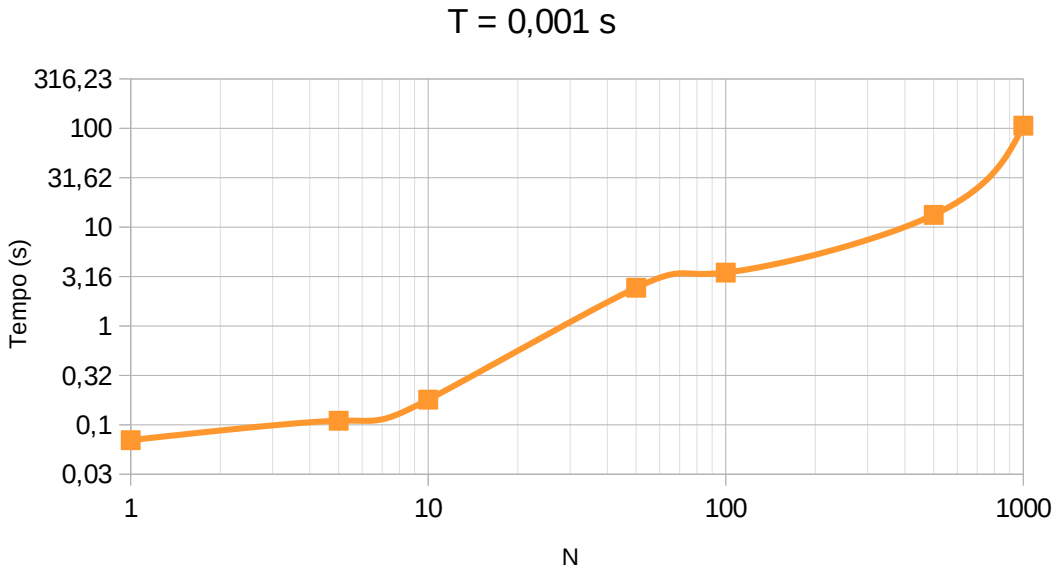

T = 0,01 s
100
31,62
10
3,16
1
0,32
0,1
0,03
Tempo (s)
1
10
100
1000
N

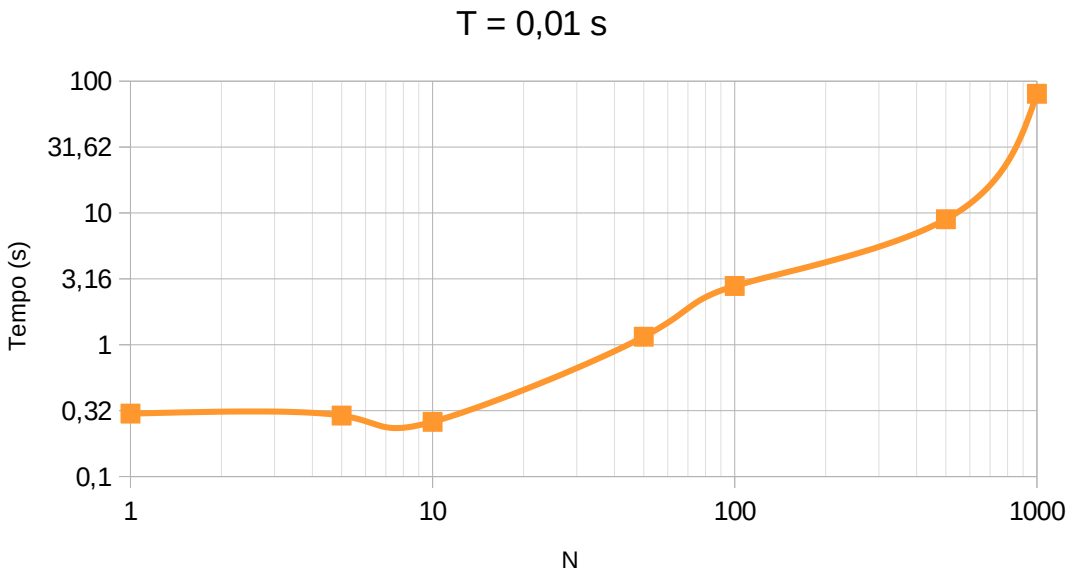

T = 0,1 s
100
31,62
10
3,16
1
0,32
0,1
0,03
Tempo (s)
1
10
100
1000
N

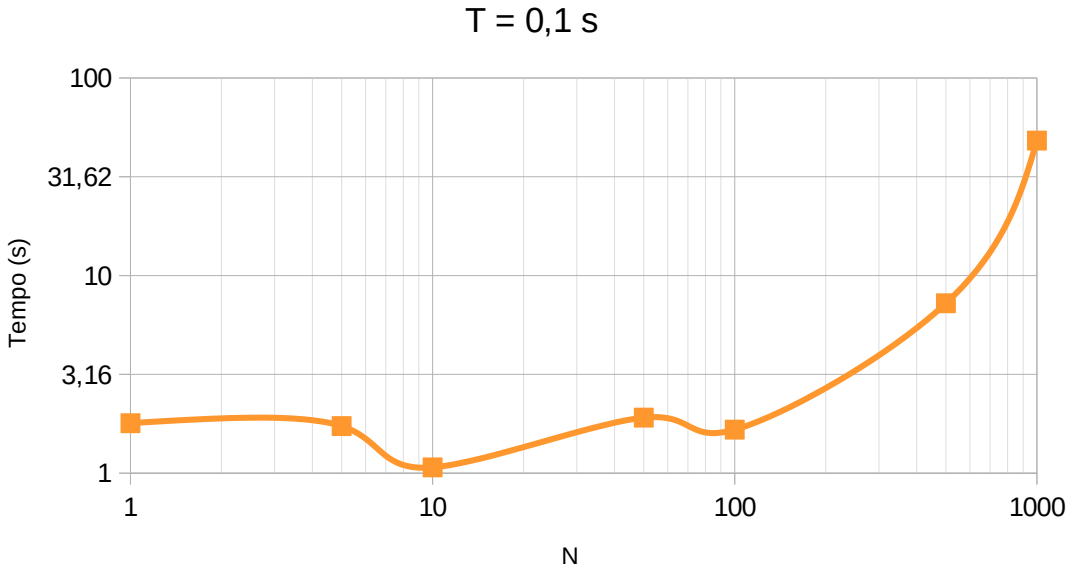


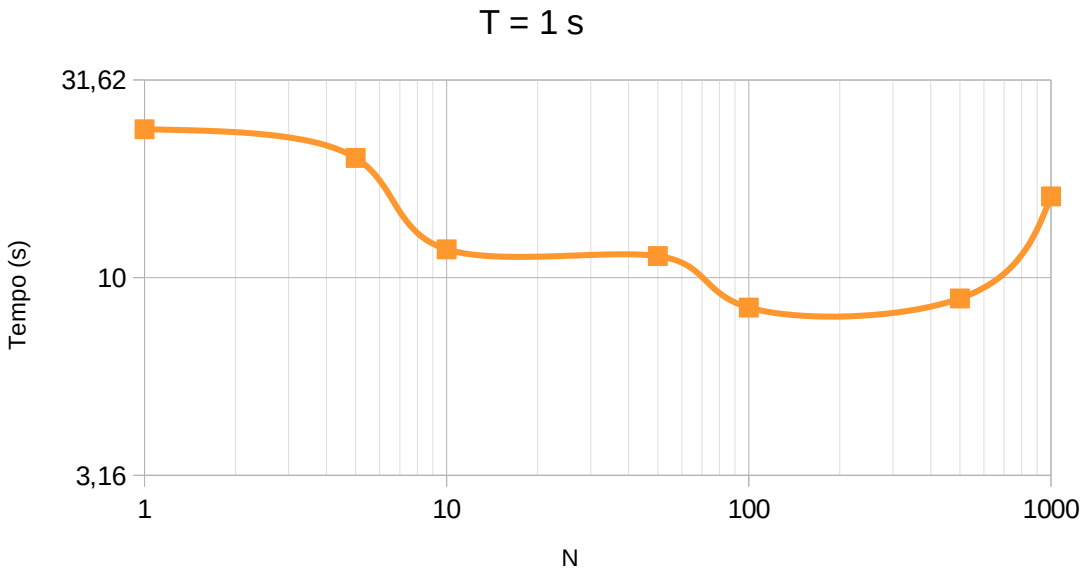
Nel caso P=0% non si ha perdita di pacchetti, anche se possono comunque esserci delle ritrasmissioni. Le performance peggiorano all'aumentare della finestra di spedizione, a causa dell'overhead dovuto alla sincronizzazione tra i threads, e sono pressoché indipendenti dal valore di T.

- P = 1%



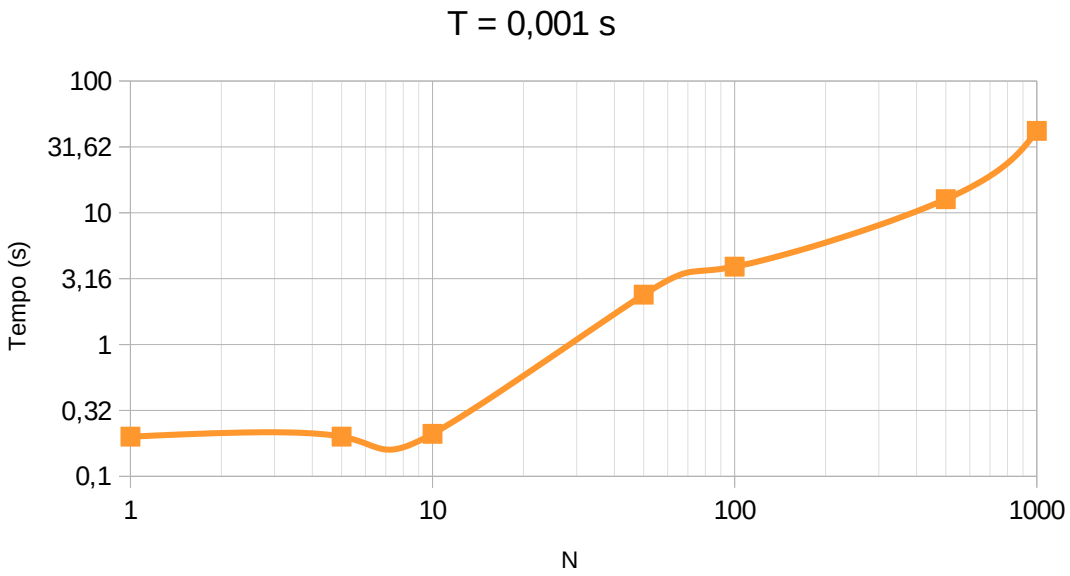

T = 0,01 s
100
31,62
10
3,16
1
0,32
0,1
Tempo (s)
1
10
100
1000
N

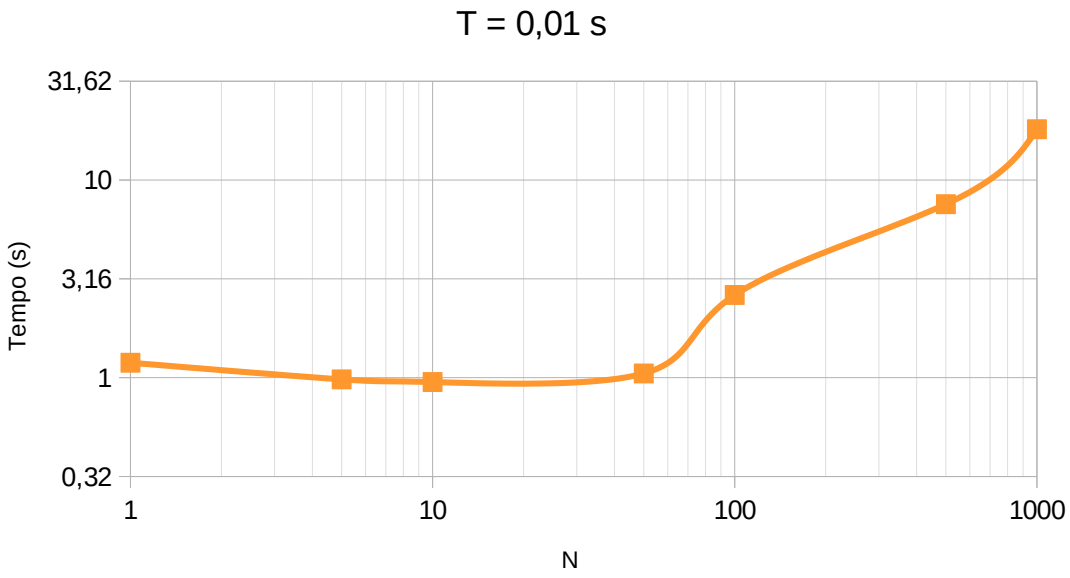

T = 0,1 s
100
31,62
10
3,16
1
Tempo (s)
1
10
100
1000
N

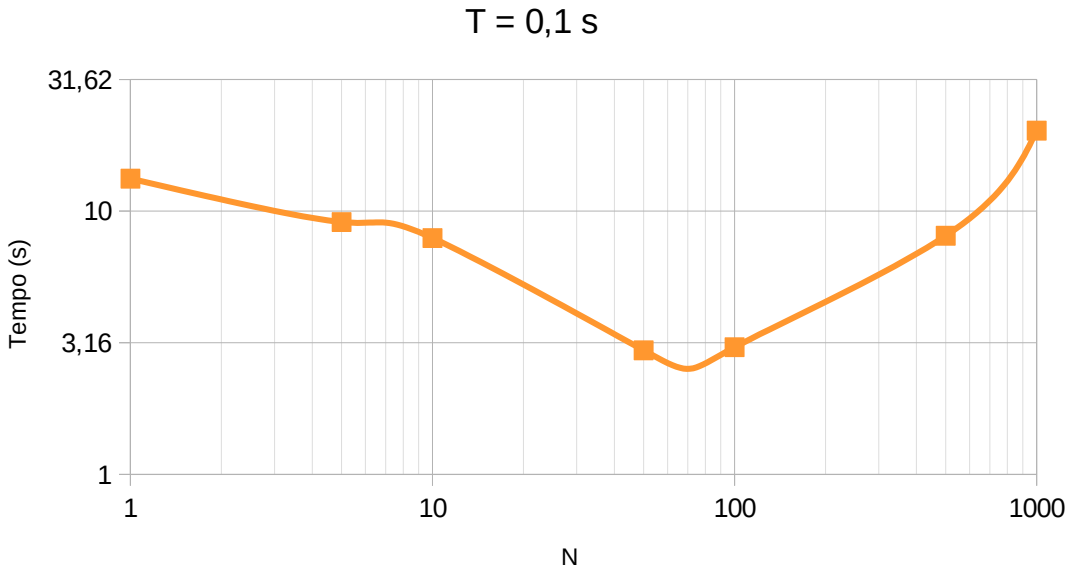


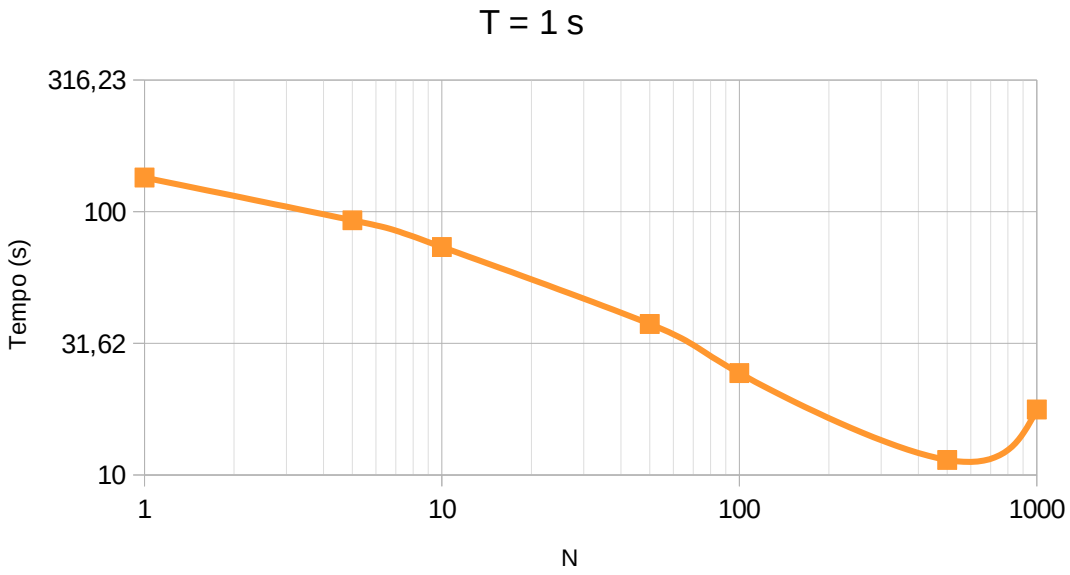
Nel caso P=1%, per valori bassi di T le performance sono identiche al caso senza perdita. Per valori di T maggiori si nota un leggero miglioramento delle performance per N>10.

- P = 5%



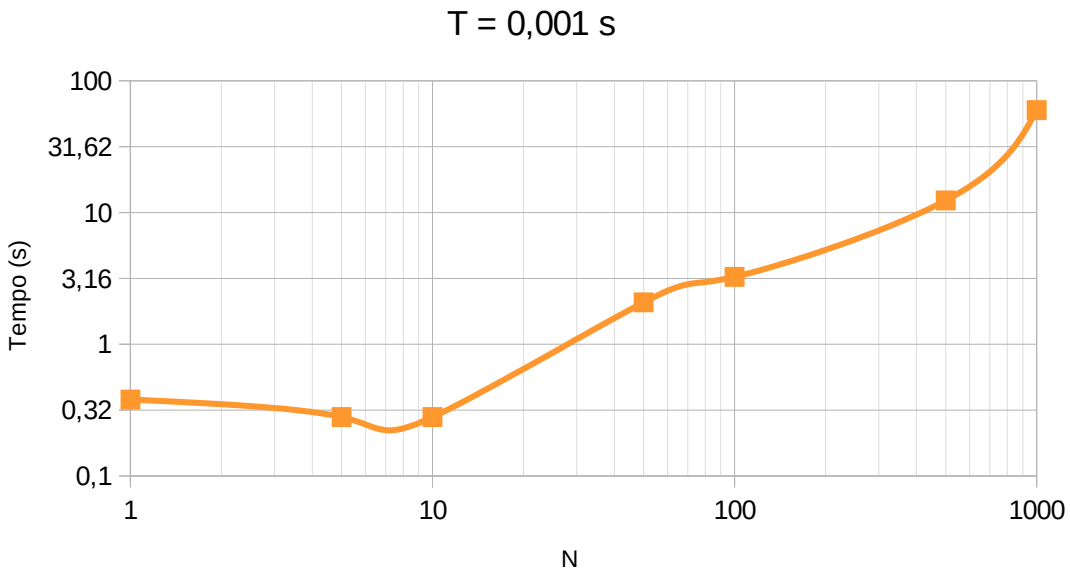

T = 0,01 s
31,62
10
3,16
1
0,32
Tempo (s)
1
10
100
1000
N

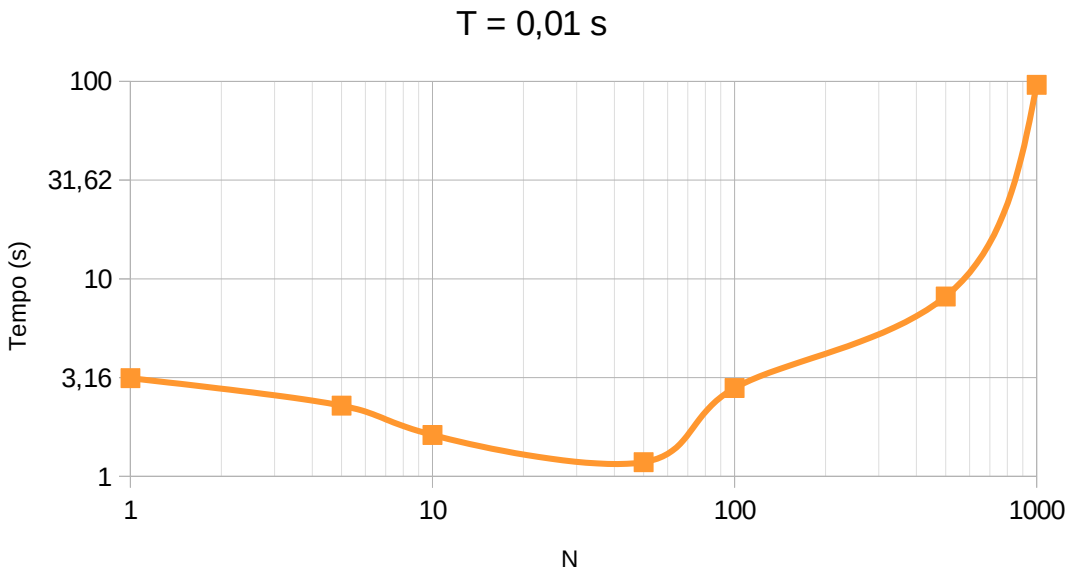

T = 0,1 s
31,62
10
3,16
1
Tempo (s)
1
10
100
1000
N

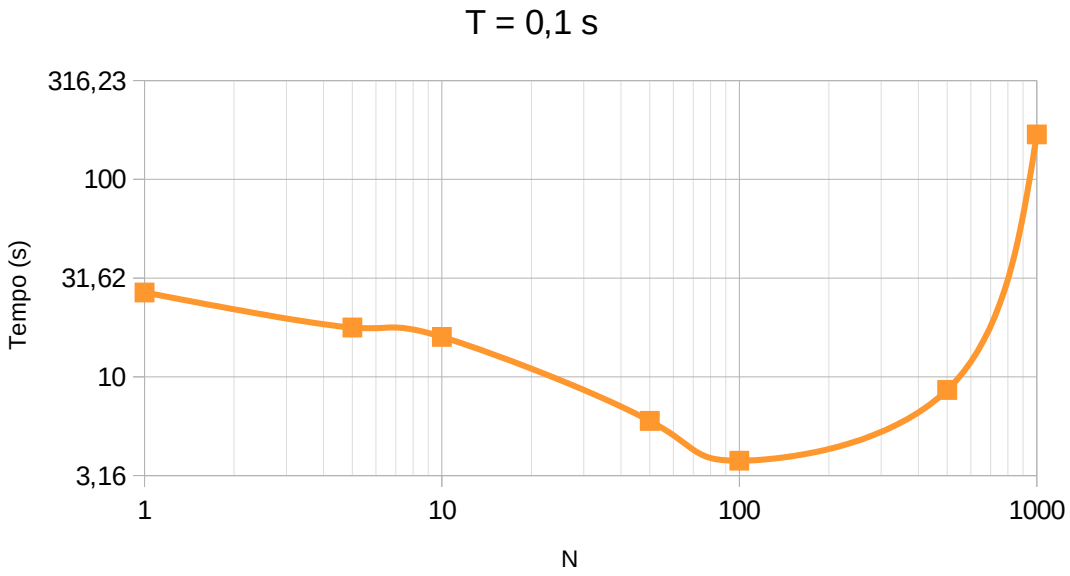


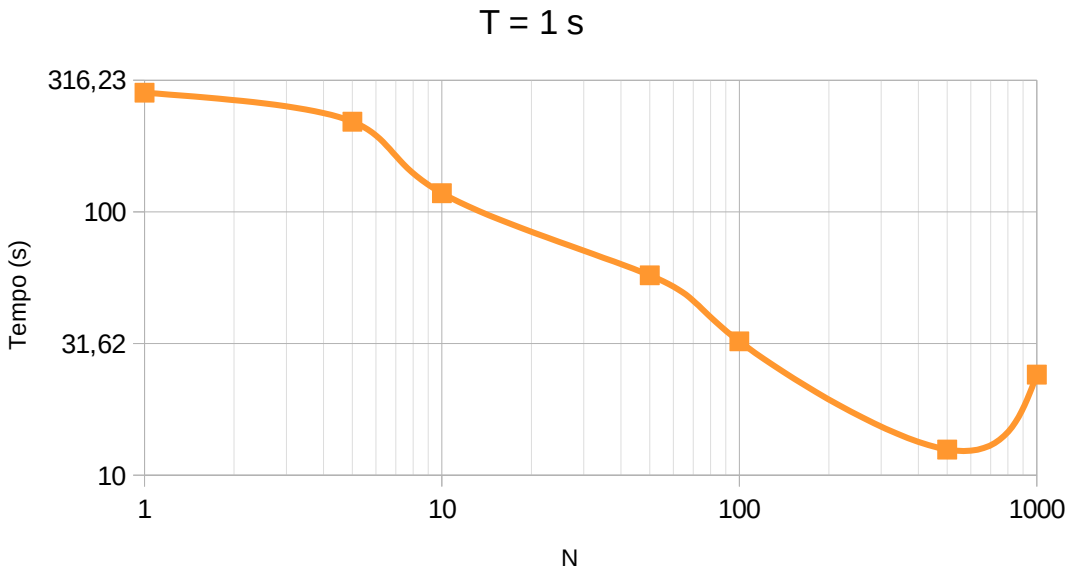
Nel caso P=5% è ancora più evidente che per valori di T bassi le performance migliori si hanno con valori piccoli di N, mentre per valori di T maggiori si hanno performance migliori con finestre di spedizione più ampie.

- P = 10%



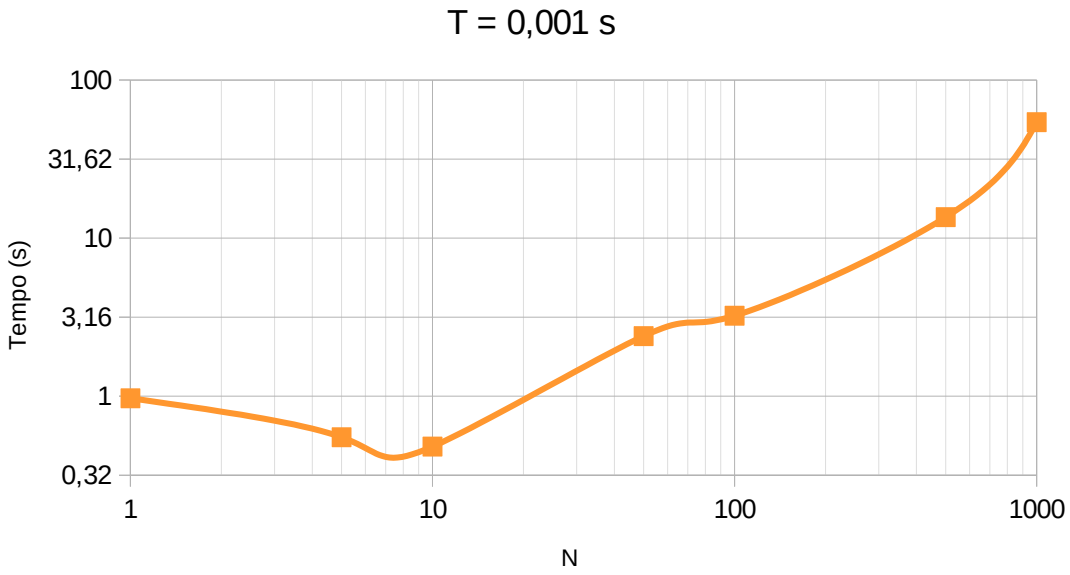

T = 0,01 s
100
31,62
10
3,16
1
Tempo (s)
1
10
100
1000
N

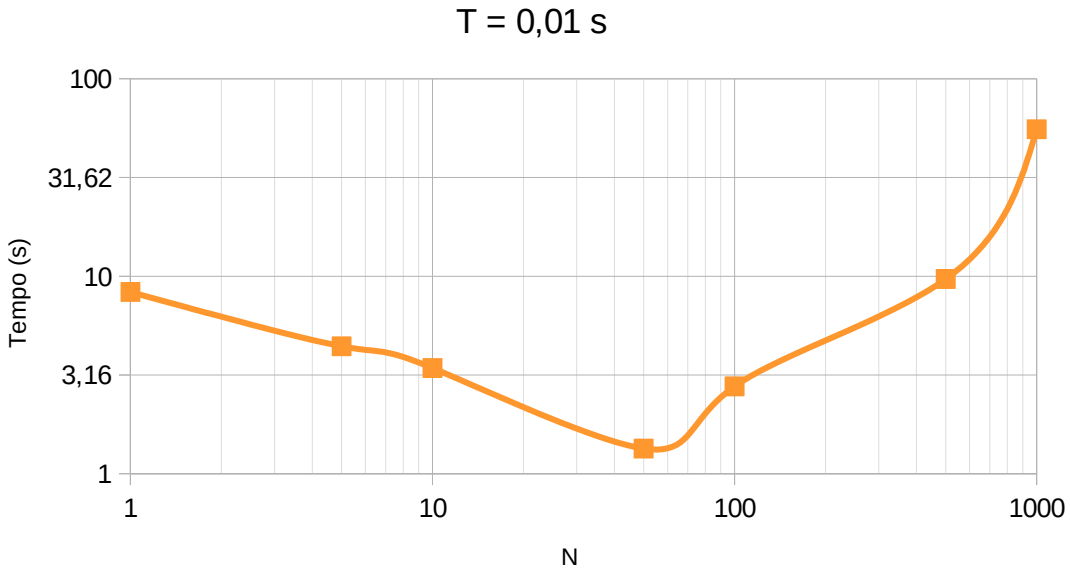

T = 0,1 s
316,23
100
31,62
10
3,16
Tempo (s)
1
10
100
1000
N

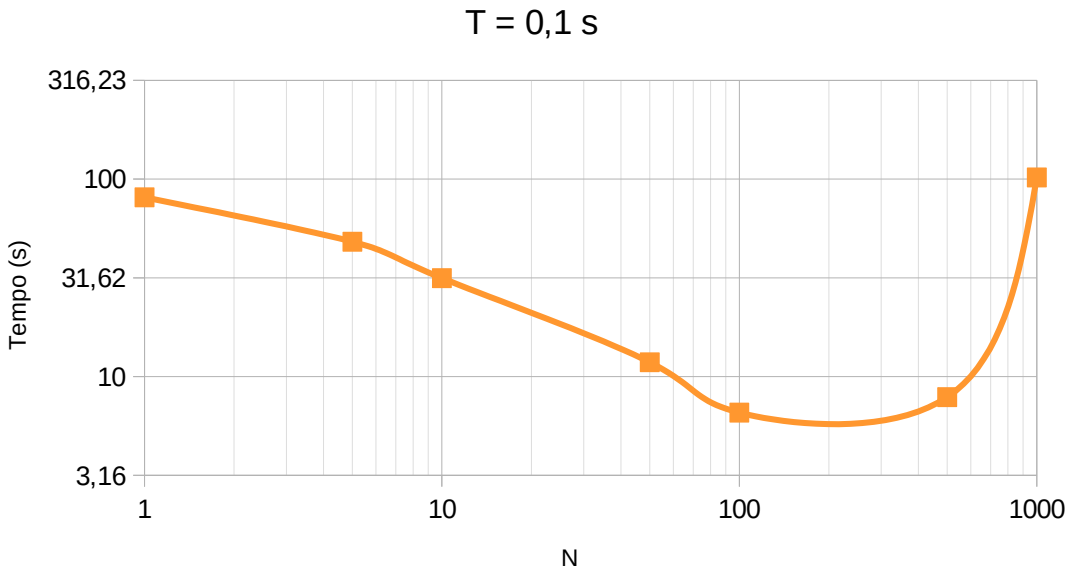


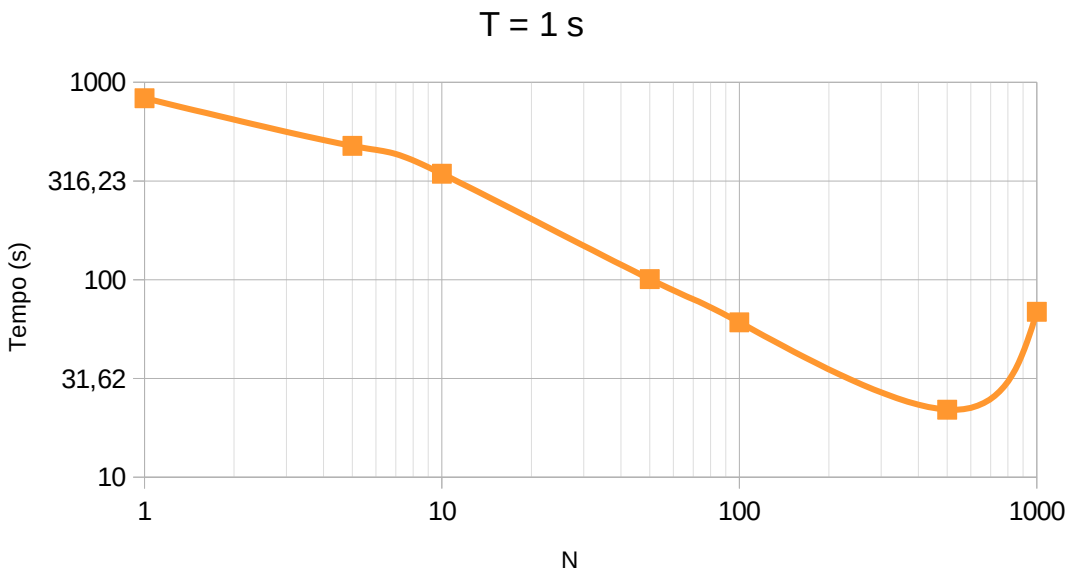
Per il caso P=10% si ha un andamento analogo al caso precedente.
I vantaggi di parallelizzare i tasks attraverso il multithreading si hanno con valori di T relativamente alti.

- P = 20%



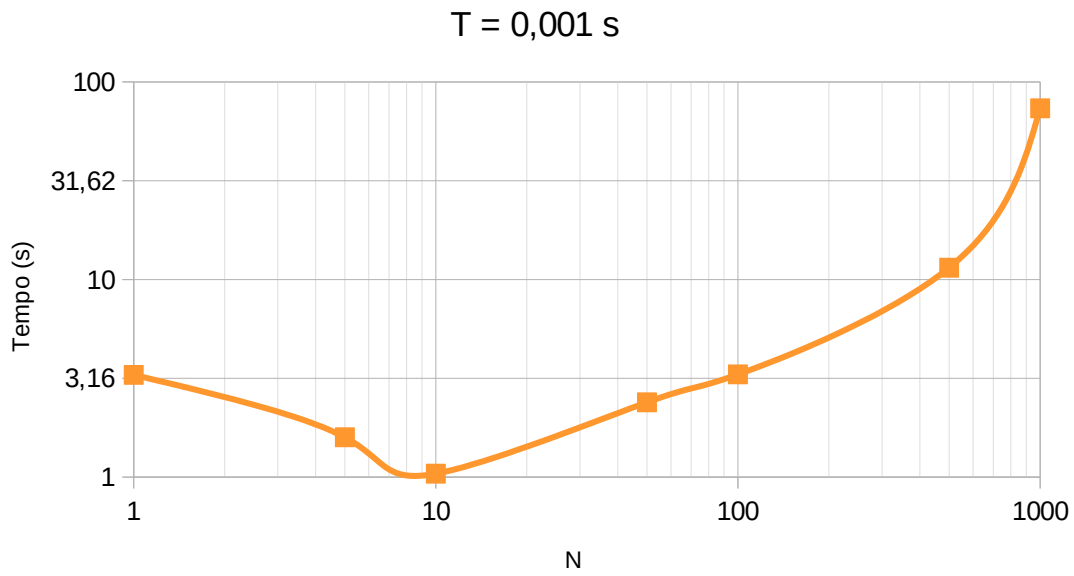

T = 0,01 s
100
31,62
10
3,16
1
Tempo (s)
1
10
100
1000
N

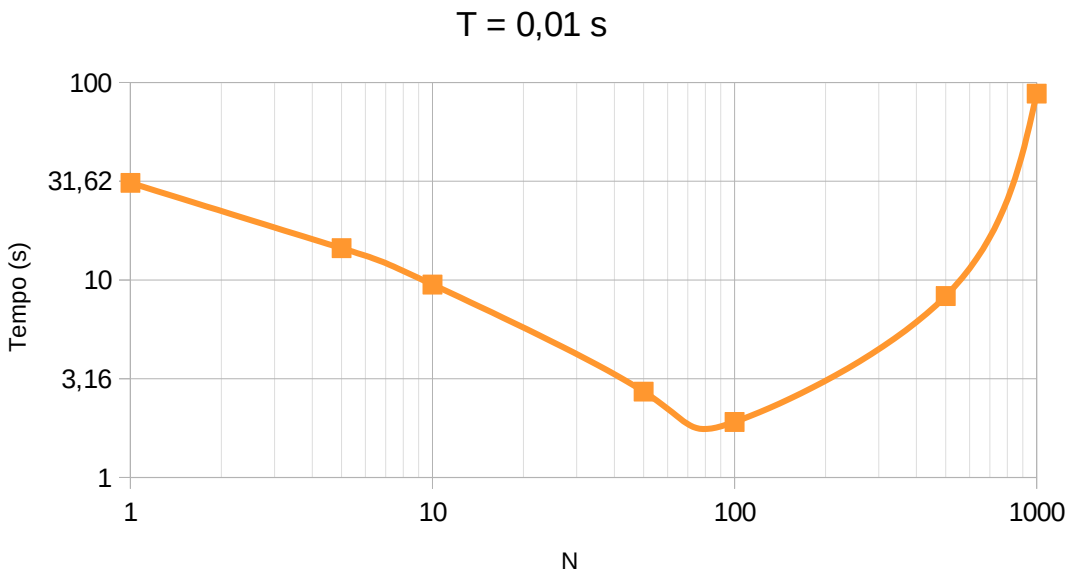

T = 0,1 s
316,23
100
31,62
10
3,16
Tempo (s)
1
10
100
1000
N

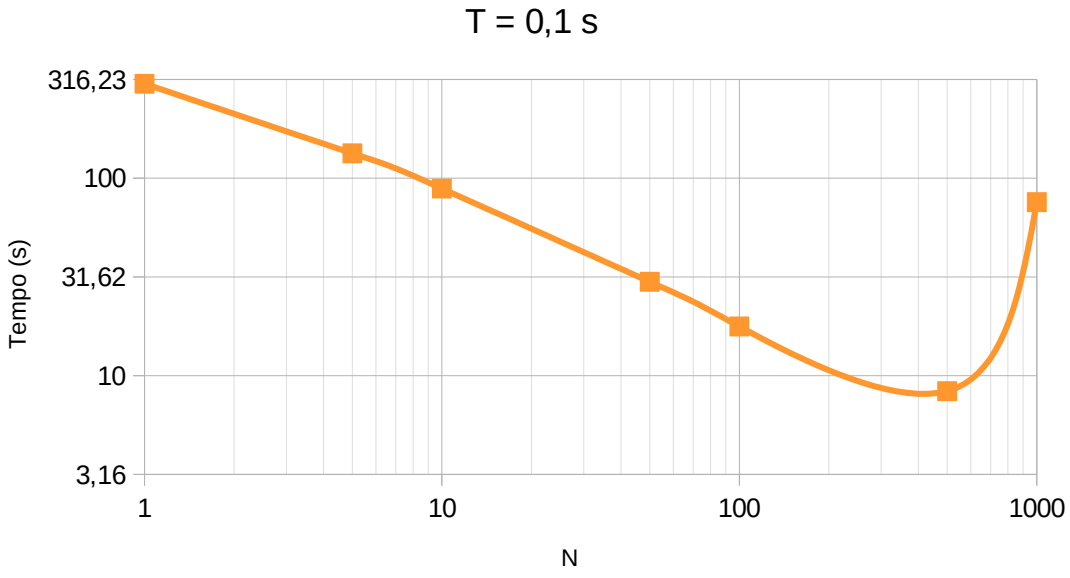


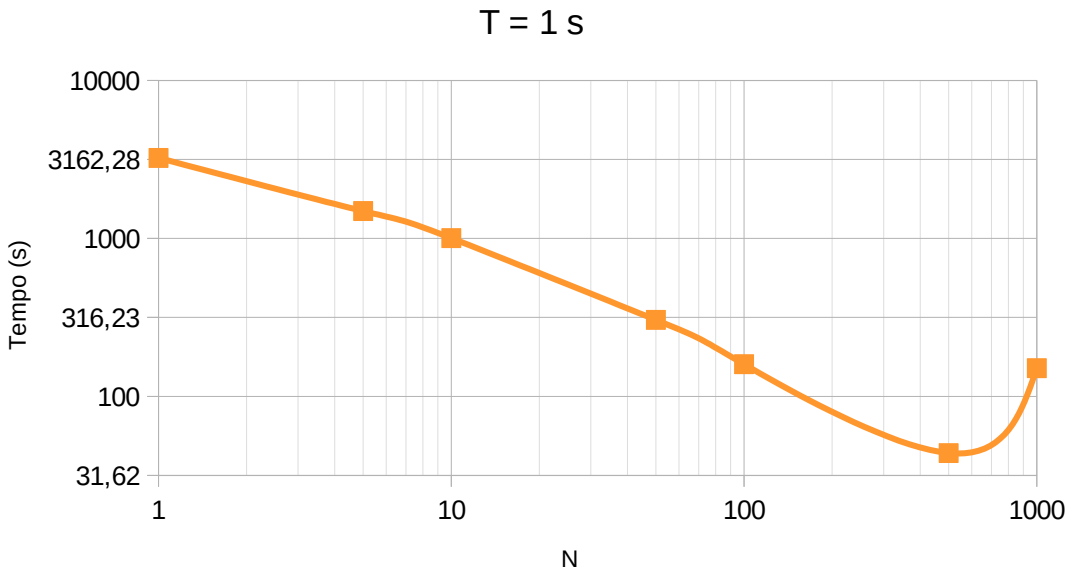
Per P=20% l'andamento è ancora analogo ai casi precedenti. Chiaramente le prestazioni migliori si hanno con valori di T bassi.

- P = 40%




T = 0,01 s
100
31,62
10
3,16
1
Tempo (s)
1
10
100
1000
N


T = 0,1 s
316,23
100
31,62
10
3,16
Tempo (s)
1
10
100
1000
N

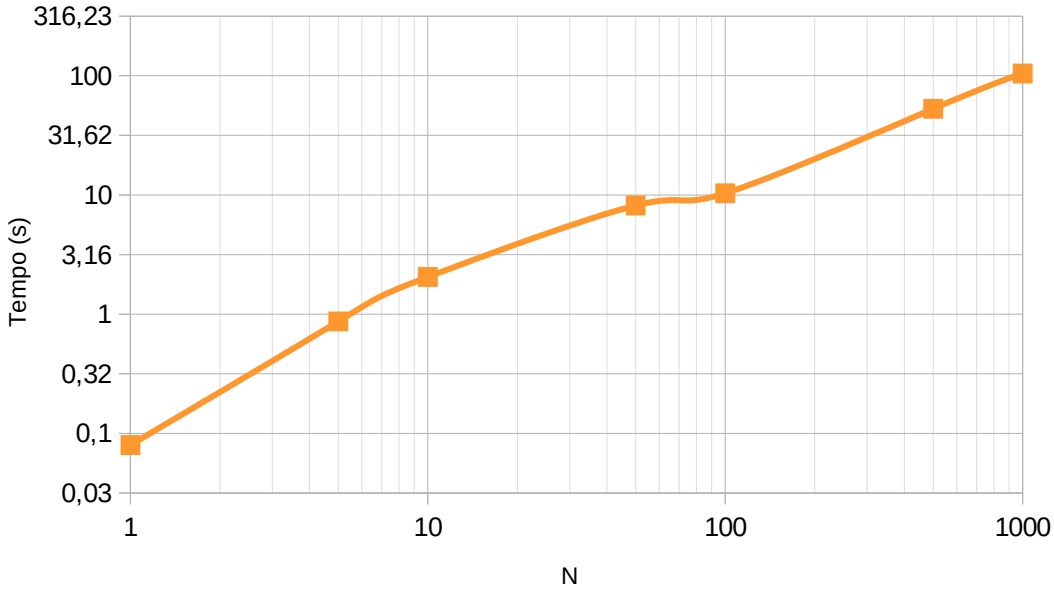


Per P=40% e valori di T relativamente alti si hanno prestazioni proibitive, così come ci si aspettava.

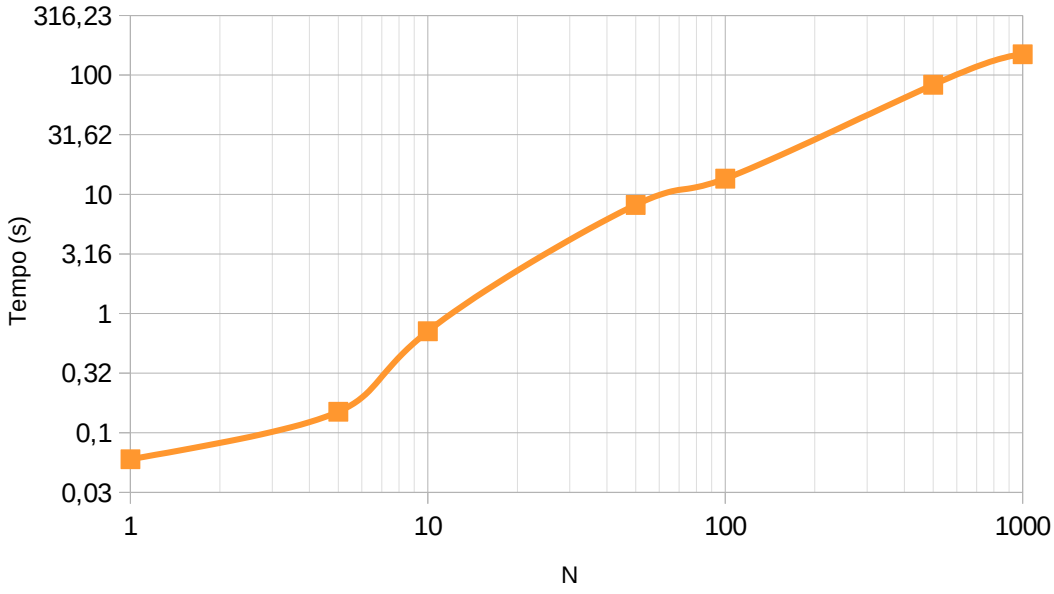
In definitiva, per valori bassi di P le prestazioni dipendono meno strettamente dal valore del timer e peggiorano all'aumentare di N. Aumentando la probabilità di perdita si hanno le prestazioni migliori con valori bassi di T e finestre di spedizione di grandezza media, cioè il miglior compromesso tra parallelizzazione dei tasks e overhead dovuto alla sincronizzazione tra i threads.

Prestazioni nel caso di timeout adattivo (comando get)

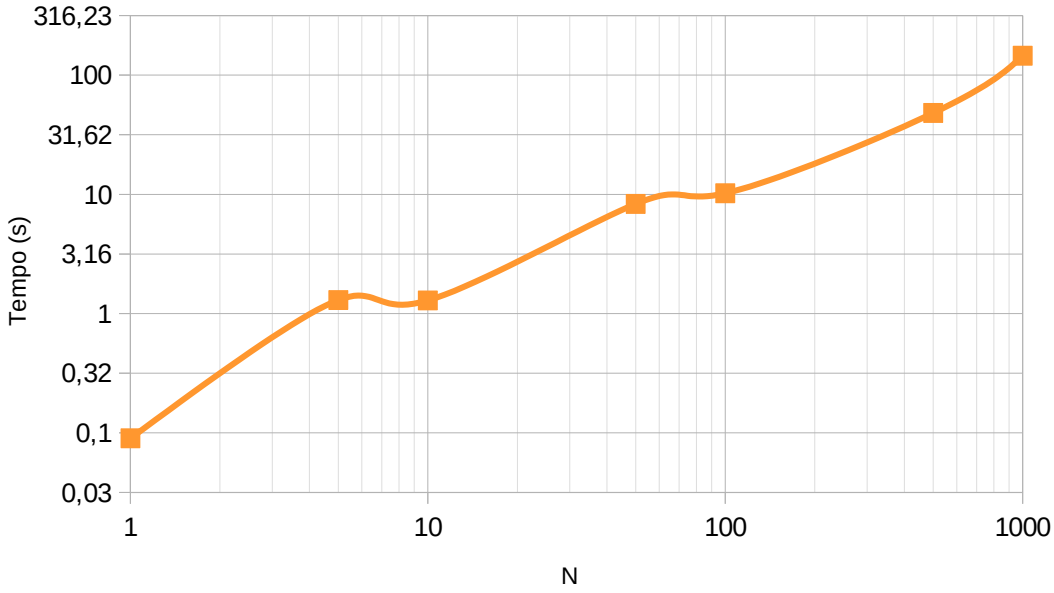
- P = 0%



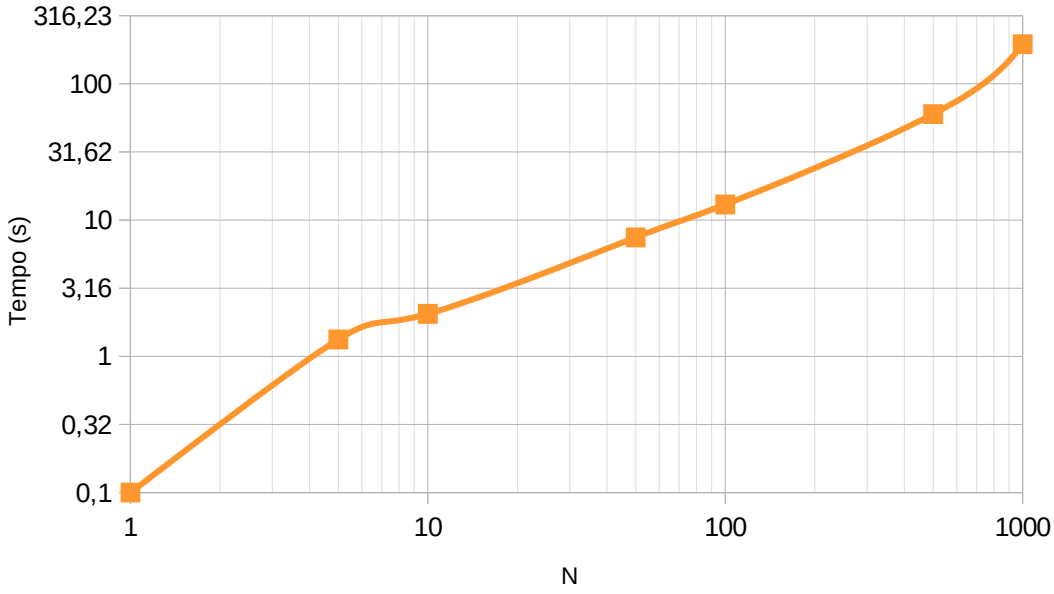
- P = 1%



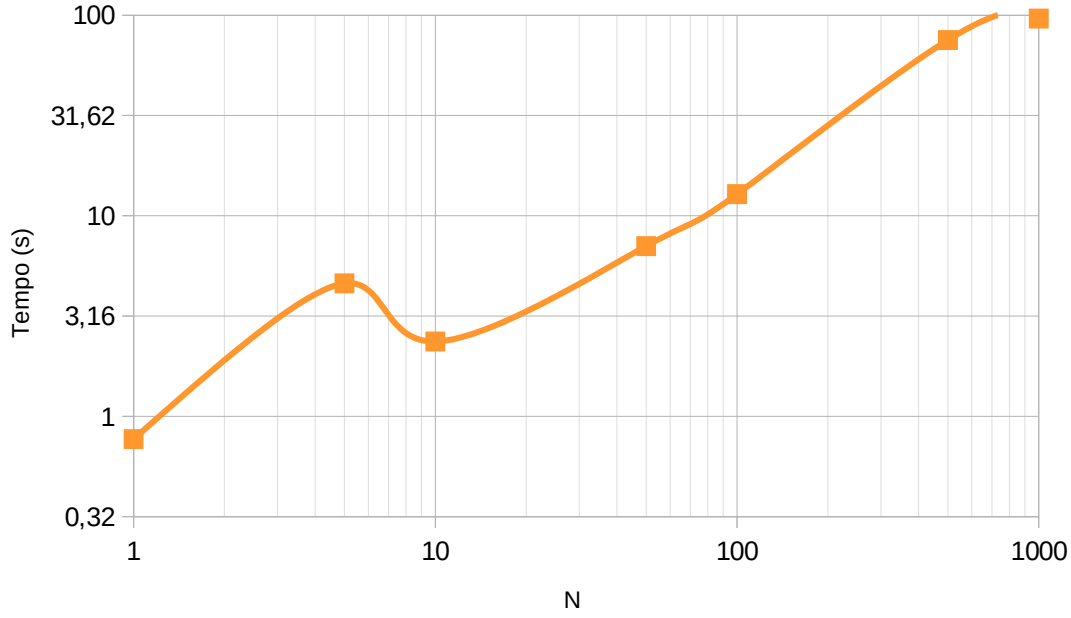
- P = 5%


316,23
100
31,62
10
3,16
1
0,32
0,1
0,03
Tempo (s)
1
10
100
1000
N

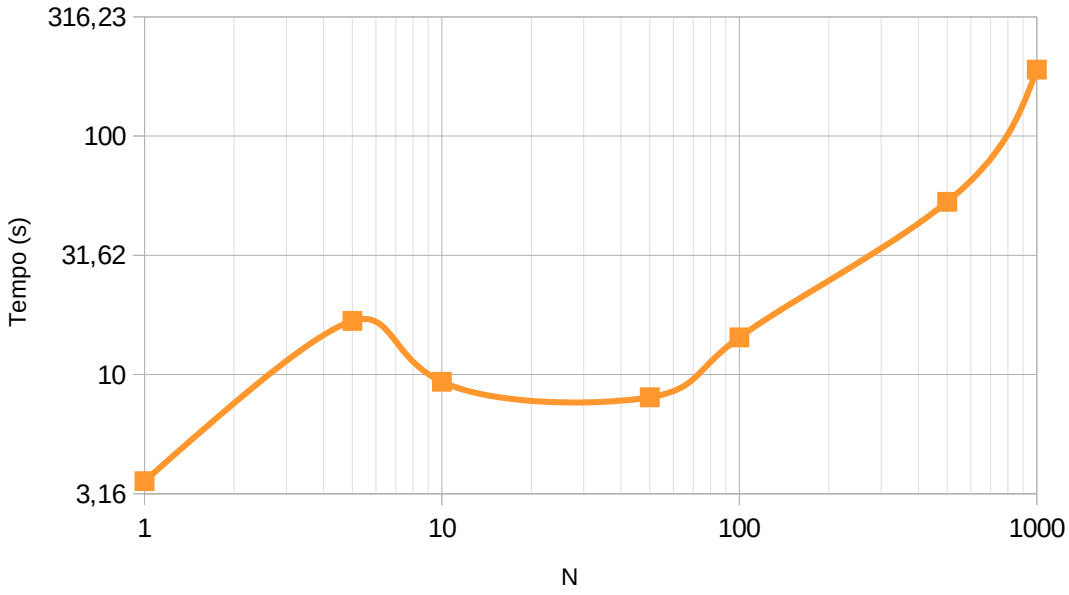
- P = 10%


316,23
100
31,62
10
3,16
1
0,32
0,1
Tempo (s)
1
10
100
1000
N

- P = 20%



- P = 40%



Con timeout adattivo, e quindi con valori di T tipicamente nell'ordine dei millisecondi o meno, l'andamento delle prestazioni non varia molto all'aumentare della probabilità di perdita dei pacchetti. Il caso migliore nella maggior parte dei test si è avuto per N=1, cioè la parallelizzazione tramite multithreading ha portato più svantaggi che vantaggi.

Manuale per l’installazione, la configurazione e l’esecuzione del sistema

Il codice sorgente e gli header dell’applicazione sono contenuti nella cartella *my-UDP*. Nel seguito si assumerà che i comandi utilizzati siano eseguiti all’interno della directory del progetto.

Configurazione

Se si vogliono configurare i parametri del protocollo, in particolare i valori di T, N e P, è necessario modificarli manualmente all’interno dei file sorgente.
In particolare, i valori di N, P e T si trovano rispettivamente nelle righe 6, 7 e 8 del file *myUDP.h*.

Da notare inoltre che il valore massimo di timeout è regolato da MAX_TIMEOUT_INTERVAL che per default è impostato a 2s.

Compilazione

Se si vuole utilizzare la versione con timeout fissi, il comando da usare sarà:

```
$ make
```

Se si vuole utilizzare la versione con timeout adattivi, il comando da usare sarà:

```
$ make adaptive
```

Per eliminare gli eseguibili, il comando da usare sarà:

```
$ make clean
```

Esecuzione

Una volta terminata la fase di compilazione, per eseguire l'applicazione è conveniente aprire due diverse istanze di shell, così da poter visualizzare all'evenienza i messaggi del server e del client contemporaneamente.

Nella prima finestra di shell verrà eseguito il server con il comando:

```
$ ./server <numero di porta arbitrario>
```

Il numero di porta può essere scelto arbitrariamente nell'intervallo [1024-65535].

Nella seconda finestra di shell verrà eseguito il client con il comando:

```
$ ./client 127.0.0.1 <numero di porta del server>
```

Da notare che l'ordine di esecuzione è importante. Se venisse eseguito prima il client, chiaramente questo non riuscirebbe a contattare il server e terminerebbe la propria esecuzione dopo 60s.

Una volta stabilita la connessione verrà stampata a schermo la lista dei comandi.

Da notare che all'interno delle cartelle rispettivamente del client e del server sono già contenuti diversi file di varie dimensioni per testare le funzionalità del sistema.

Per terminare l'esecuzione, nel caso del client è possibile utilizzare sia il comando *quit*, che chiude la connessione e dopo 30 secondi termina il processo principale, oppure il segnale *SIGINT* (ctrl+c). Nel caso del server è necessario utilizzare il segnale SIGINT (ctrl+c) per terminare il processo principale.

Esempi di funzionamento

1. Esecuzione del server (in attesa di ricevere un messaggio di SYN)

```
[marco@fedora-thinkpad my-UDP-v2]$ ./server 7777
```

2. Esecuzione del client (invio del SYN e ricezione del SYN-ACK)

```
[marco@fedora-thinkpad my-UDP-v2]$ ./client 127.0.0.1 7777

N = 10
T = 1.000 s (adaptive timer)
P = 10%

Sending SYN message...
Connection established with server.

Insert one of the following requests to the server:
- list
- get <filename>
- put <filename>
- mylist
- help
- quit
```

3. Connessione avvenuta con successo (lato server)

```
[marco@fedora-thinkpad my-UDP-v2]$ ./server 7777

Received SYN message from 127.0.0.1 on port 54149
Spawning a connection socket...

Connection established with:
Client address  127.0.0.1
Client port     54149
```

4. Comando *list*

```
list

LIST:
1M.txt
1k.txt
2k.txt
4k.txt
8k.txt
16k.txt
32k.txt
64k.txt
128k.txt
256k.txt
512.txt
512k.txt
ibm1401.jpg
lorem_ipsum.txt
newton.jpg

Total time elapsed: 0.000 s
```

5. Comando *get*

```
get 1M.txt

Progress: [#################### 100.00%]

Download completed!
Total time elapsed: 3.090 s
```

6. Comando *put*

```
put newton.jpg

Progress: [#################### 100.00%]

File uploaded correctly!
Total time elapsed: 9.048 s
```

7. Comando *mylist*

```
mylist

MY LIST:
1M.txt
hello.c
movies.jpg
newton.jpg
```

8. Comando *get* per un file inesistente

```
get 1M


Error: the file requested doesn't exist. Please, try again.
```

9. Comando *put* per un file inesistente

```
put newton


Error: the file to send doesn't exist.
```

10. Comando *quit* (lato client)

```
quit

Closing connection...
```

11. Comando *quit* (lato server)

```
Connection closed with client on port 54149
```