Ingegneria di Internet e del Web - A.A. 2018/19

Trasferimento file su UDP

Progetto B

Antonangeli Mattia, Trombetti Lisa

Sommario

1. Introduzione

Lo scopo del progetto è quello di progettare e implementare in linguaggio C un'applicazione Client-Server per il trasferimento affidabile di file che impieghi UDP come protocollo di strato di trasporto.

Il sistema deve permettere l'utilizzo di tre comandi principali:

- List: visualizzazione sul client dei file disponibili sul server
- Get: download di un file dal server
- Put: upload di un file sul server

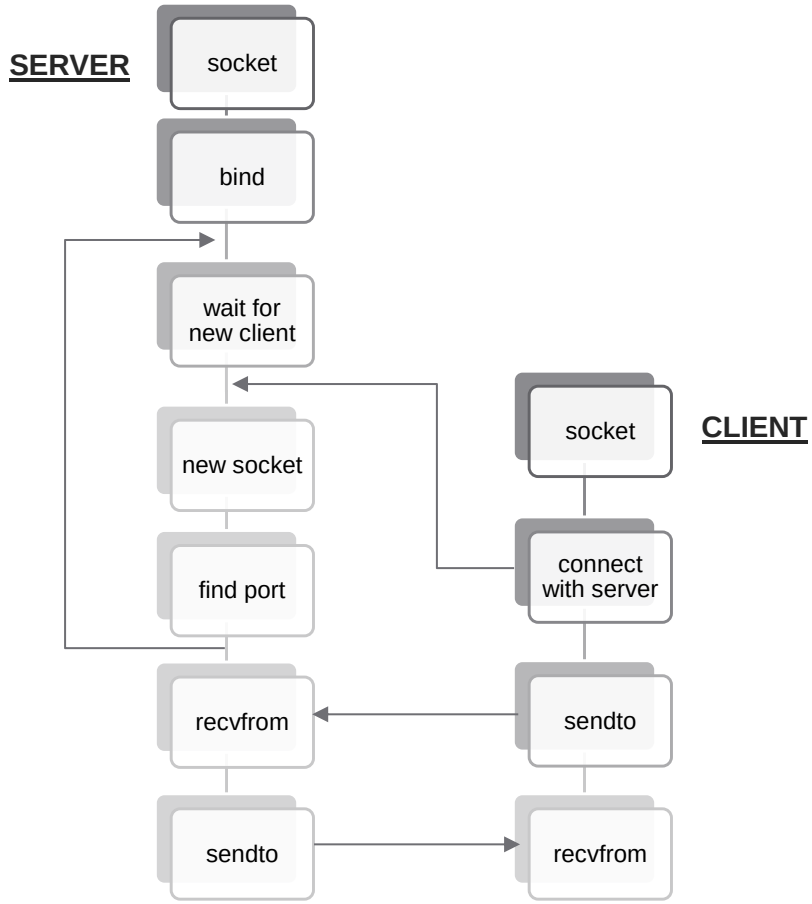
Il sistema è stato sviluppato in ambiente Unix, testato su un dispositivo con processore Intel Core I3-4005U e con SO basato su Ubuntu 18.04.

2. Ciclo server/client

Per il codice relativo al client/server fare riferimento ai file "client_udp.c" e "server_udp.c" contenuti rispettivamente nelle cartelle "client" e "server".

Tutte le funzioni nominate di seguito sono contenute nella libreria "sr_udp.c".

2.1. Connessione iniziale



2.1.1. Lato server

Dal punto di vista del server, allo start si effettuano le seguenti operazioni:

1) Il server crea una nuova socket eseguendo il bind() con l'opzione INADDR_ANY per accettare i pacchetti da una qualsiasi delle sue interfacce di rete.
2) Il server si mette quindi in attesa sulla porta di default.
3) Quando riceve una richiesta di connessione viene creata una nuova socket e si cerca una porta libera su cui far connettere il nuovo client utilizzando la funzione "find_port()".
4) Viene richiamata la funzione "connect_server()" che si occupa di inviare al client i parametri della comunicazione:
    - numero di porta
    - dimensione della finestra di ricezione
    - timeout
    - flag del timeout adattivo
5) Viene quindi creato un nuovo processo figlio tramite fork() che si occuperà delle richieste del client.
6) Il server torna ad attendere nuove connessioni.

I parametri di connessione, la porta di default e il numero totale di connessioni concorrenti che si vuole supportare possono essere modificati direttamente nel file "server_udp.c" (tramite le costanti alle righe 18-22).

Per evitare che rimangano aperte indefinitamente delle connessioni è stata inserita un'attesa di 3 min allo scadere della quale se il client non ha inviato nessun comando il processo figlio ad esso associato viene terminato.

2.1.2. Lato client

Dal punto di vista del client, allo start si effettuano le seguenti operazioni:

1) Il client crea una nuova socket.
2) Viene richiamata la funzione "connect_client" che si occupa di contattare il server sulla porta di default e di ricevere i parametri della connessione (vedi 2.1.1. Lato server, punto 4).
3) A questo punto è pronto a comunicare con il server e rimane in attesa di un nuovo comando in input.

NOTA: Se il server non in esecuzione, il client ritenta la connessione per 5 secondi, dopodiché il processo viene terminato dopo aver mostrato un messaggio di errore.

2.2. Chiusura

La chiusura della connessione può avvenire in 3 modi:

1) Input del comando quit da tastiera (q)
2) Terminazione tramite interrupt (Control + C)
3) Timeout, se non si inserisce un nuovo comando per più di 3 min

Il client richiama la funzione "quit_conn" che si occupa di inviare al server una richiesta di chiusura della comunicazione.

3. Trasferimento file affidabile

I messaggi che vengono scambiati fra client e server hanno tutti la stessa struttura:

```
struct message{
        char *cmd;          // comando

        char *mess;         // contenuto del messaggio

}
```

Per inviare un messaggio viene usata la funzione “send_mess” che estrapola dalla struttura message i dati da inviare trasformandoli in una stringa del tipo comando+messaggio. Viceversa, la funzione “recv_mess” prende i dati ricevuti e li divide per popolare la struttura message passata in input.

Le funzioni “getFunc”, “putFunc” del server e rispettivamente “get_file”,”put_file” del client fanno uso delle funzioni elencate di seguito.

3.1. Funzione upload_file

*int upload_file(int sd, struct sockaddr_in addr, FILE * fd, int N, int start_timeout, int adapt, int dim)*

- Descrittore della socket: sd
- Indirizzo a cui inviare i dati: addr
- Descrittore del file da inviare: fd
- Dimensione della finestra di invio: N
- Timeout iniziale: start_timeout
- Flag per il timeout adattivo: adapt
- Dimensione del file: dim

La funzione “upload_file” implementa il protocollo Selective Repeat dalla parte del sender e permette l’utilizzo sia di un timeout fisso che di uno adattivo, specificato tramite la flag adapt.

I messaggi inviati sono strutturati come segue:

cmd → numero_di_sequenza=dimensione

mess → dati letti dal file

In una fase iniziale vengono inviati una serie di messaggi con numero di sequenza compreso fra [0,send_base+N] (la send_base è inizialmente posta a 0), finchè o non si trova la fine del file oppure si arriva al massimo numero di pacchetti che possono essere contemporaneamente in volo.

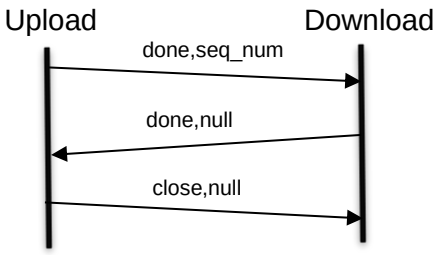
Per ogni nuovo messaggio che deve essere inviato viene creato un thread a cui viene passata la struttura “thread_arg” che contiene informazioni sull’indirizzo a cui devono essere inviati i dati, la struttura message ed il valore del timeout.

Il thread tenterà ripetutamente di inviare il messaggio con uno sleep pari al timeout fra un “send_mess” e quello successivo.

A questo punto la funzione si mette in attesa di ricevere qualcosa dal destinatario.

Alla ricezione di un ack si controlla che il suo sequence number si trovi in [send_base,send_base+N] e, se presente in quel range, si invia un segnale di terminazione al thread associato tramite pthread_cancel(tid).

Se il sequence number dell'ack ricevuto coincide con la send_base viene fatta scorrere in avanti la finestra rendendo possibile l'invio di nuovi pacchetti.



Quando la fine del file viene raggiunta si dà inizio ad un handshake finale come mostrato nella figura a sinistra.

Viene quindi ritornato il valore 0 per segnalare che il trasferimento è andato a buon fine.

Nel caso in cui la comunicazione con il receiver venga interrotta e non arrivi nessun messaggio per troppo tempo vengono eliminati i thread e si ritorna -1 per segnalare l'errore.

3.1.1. Timeout adattivo

Per implementare un timeout che sia in grado di adattarsi ai ritardi presenti sulla rete è stata creata la funzione "get_timeout" che prende in input il valore dell'RTT misurato e una struttura "adaptive_tm":

```
struct adaptive_tm{
        int est_rtt;

        int dev_rtt;

};

int get_timeout(adaptive_tm *atm, int new){

        atm->est_rtt = (1-ALPHA)*(atm->est_rtt) + ALPHA*new;

        atm->dev_rtt = (1-BETA)*(atm->dev_rtt) + BETA*abs(new - atm->est_rtt);

        return (atm->est_rtt + 4*(atm->dev_rtt));

}
```

Per stimare il valore dell'RTT è stata utilizzata la funzione gettimeofday(struct timeval *tv, struct timezone *tz) della libreria time.h.

3.2. Funzione download_file

*int download_file(int sd, struct sockaddr_in addr, FILE *fd, int N)*

- Descrittore della socket: sd
- Indirizzo da cui ricevere i dati: addr
- Descrittore del file su cui scrivere i dati ricevuti: fd
- Dimensione della finestra di ricezione: N

Come per la funzione precedente anche “download_file” implementa il protocollo Selective Repeat, ma dal lato del receiver.

I pacchetti inviati dalla funzione “upload_file” sono riassemblati in ordine per ottenere il file inviato o, nel caso del comando list, la lista corretta dei file presenti sul server.

Inizialmente la funzione rimane in attesa di un messaggio dal sender.

Alla ricezione di un messaggio, viene verificato il contenuto del campo cmd:

- err: viene stampato su schermo il messaggio di errore presente nel campo mess e la funzione ritorna -1
- done: segnala che l’altro lato ha trovato la fine del file
- messaggio informativo: se il numero di sequenza si trova in [recv_base,recv_base+N) viene inviato l’ack e viene salvato il contenuto del campo mess, se invece si trova in [recv_base-N,recv_base) viene reinviato soltanto l’ack. In ogni altro caso il pacchetto viene semplicemente scartato

Se il numero di sequenza dell’ack ricevuto è uguale alla recv_base viene fatta scorrere la finestra di ricezione andando a scrivere sul file il contenuto dei messaggi precedentemente salvati.

Quando la fine del file viene raggiunta anche dal lato del receiver viene inviato un messaggio con campo cmd=done e si attende di ricevere un messaggio di close per terminare la comunicazione.

Anche in questo caso se la comunicazione con il sender viene interrotta e quindi non si riceve nessun messaggio per troppo tempo viene stampato su schermo un messaggio di errore e la funzione ritorna -1.

3.3. Funzione listFunc

*int listFunc(int sd, struct sockaddr_in client, int N, int start_timeout, int adapt, char *path)*

- Descrittore della socket: sd
- Indirizzo del client: client
- Dimensione della finestra di invio: N
- Timeout iniziale: start_timeout
- Flag per il timeout adattivo: adapt
- Path della cartella di file del server: path

La funzione “listFunc” ha il compito di inviare al client la lista dei file presenti sul server per cui è possibile il download.

L’implementazione è essenzialmente uguale a quella di “upload_file”, ma i messaggi che vengono inviati contengono i nomi dei file presenti nella cartella il cui path viene passato come input.

Per interagire con la directory sono state usate le funzioni della libreria <dirent.h>:

- *opendir(const char* name)*: apertura della directory
- *readdir(DIR *dirp)*: lettura del contenuto della directory

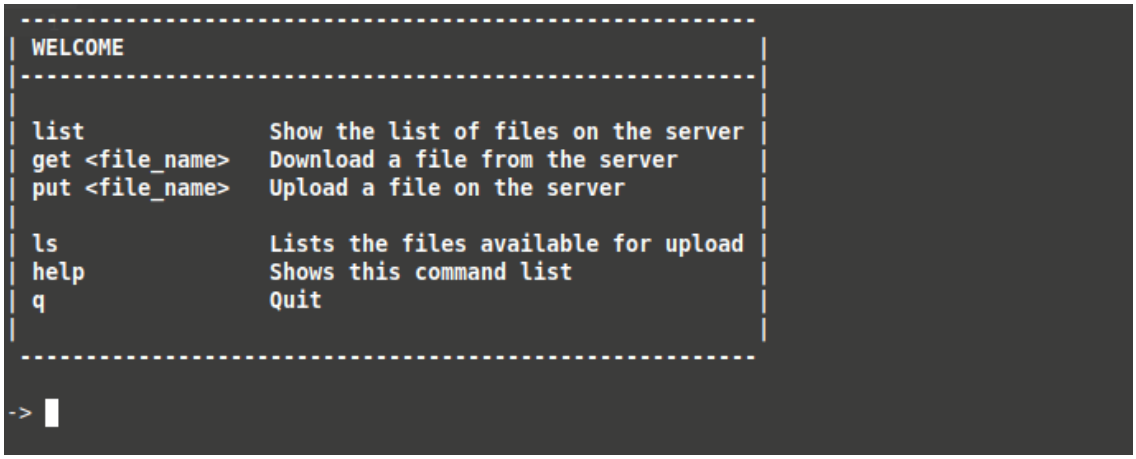
4. Istruzioni per l’uso

Passi necessari all’avvio del programma:

1. Accedere alla cartella ftUDP
2. Aprire la shell
3. Digitare il comando “make” o “make debug” nel caso in cui si intenda testare l’efficienza del sistema
4. Avviare il server con “./serverF”
5. Avviare il client con “./clientF”

4.1. Interfaccia utente

Dopo aver avviato sia il server che il client ciò che si presenta all’utente (client) è un’interfaccia come segue:

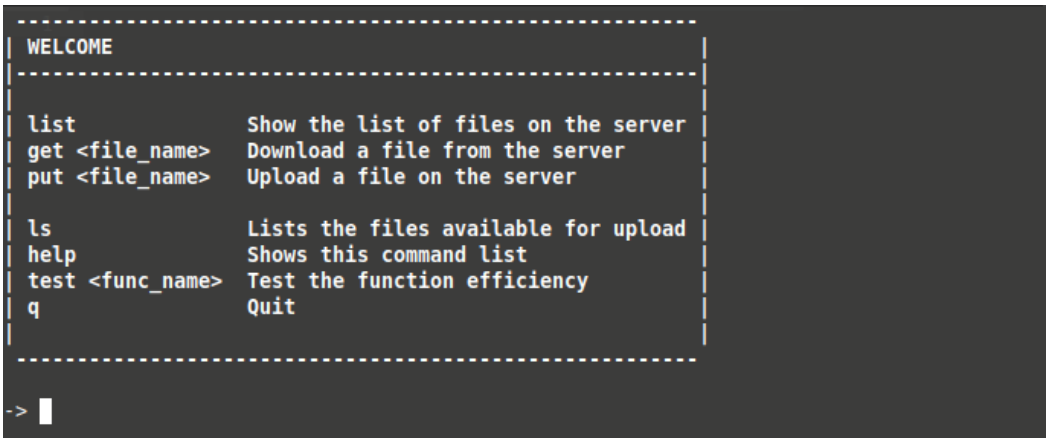


Oltre ai tre comandi principali (list, get, put) l’utente può:

- Visualizzare la lista dei file disponibili per l’upload (comando ls)
- Visualizzare nuovamente la lista dei comandi (comando help)
- Uscire dal programma (comando q)

È possibile modificare il path in cui vengono salvati i file scaricati tramite la costante PATH alla riga 14 del file “client_udp.c”.

4.2. Modalità debug



Compilando con il comando “make debug” nell’interfaccia utente (client) appare la possibilità di eseguire il comando test che permette di richiamare la funzione “file_transfer_debug” per effettuare una prova delle funzioni list, get, put . È necessario passare come parametro il nome della funzione che si intende testare.

A questo punto è possibile scegliere sia il numero di ripetizioni del test che il file su cui operare nel caso di comando get o put.

4.3. Esempi di funzionamento

Comando list:

```
-> list

----- Server File list -----

[BD]0245805.rar
.directory
bg.jpg
Mattia.gif
prova.txt
video.mp4
Lisa.gif
abcd.txt
index.jpeg

----------------------------

->
```

Comando get:

```
-> get prova

Error opening file: File not found

-> get prova.txt


Download of file 'prova.txt' complete

->
```

Comando put:

```
-> put prova.txt

Error opening file: File already exists, please rename it before trying again!

-> put prova2.txt


File successfully uploaded to the server!

->
```

Server:

```
Server started successfully. Waiting for clients...

[Port:50001] - Connected with new client
[Port:50001] - Command list
[Port:50001] - Command get
[Port:50001] - File 'index.jpeg' successfully sent
[Port:50002] - Connected with new client
[Port:50002] - Command list
[Port:50002] - Command put

Error opening file (put): File exists

[Port:50002] - Error receiving file
[Port:50002] - Command put
[Port:50002] - File 'prova2.txt' successfully received
[Port:50001] - Command quit
[Port:50001] - Closing connection with client
[Port:50001] - Connected with new client
[Port:50001] - Command list
[Port:50002] - Command quit
[Port:50002] - Closing connection with client
```

5. Prestazioni

In fase di test è stato utilizzato, per il trasferimento, un file di dimensione 1 MB. I test sono stati effettuati con variazioni di:

- Tipo di timeout (normale / adattivo);
- Timeout iniziale (10000 / 20000 / 30000);
- Probabilità di errore (0 / 0,125 / 0,25 / 0,5);
- Dimensione finestra di ricezione (1 / 3 / 5 / 10 / 20);

I valori ottenuti sono stati ricavati effettuando la media aritmetica dei tempi di 1000 sessioni di list / get / put.

Per vedere l'andamento dei tempi di trasmissione e ricezione, sono stati considerati, separatamente, i tempi in caso di:

- Timeout fisso (10000 ms), relativi alla dimensione della finestra N = 10, con tutte le probabilità
- Timeout fisso (10000 ms), relativi alla probabilità P = 0.125, con tutte le dimensioni delle finestre.
- Variazione del timeout in caso di TO fisso e adattivo, con N = 10 e P = 0,125

I valori ottenuti nelle tabelle sono espressi in microsecondi.

5.1. List

Var. finestra	N = 1	N = 3	N = 5	N = 10
P = 0.125 T = 10000 fisso	171503,6	135121,6	112466,5	55554,88

Var. prob.	P = 0	P = 0,125	P = 0,25	P = 0,5
N = 10 T = 10000 fisso	15152,99	55554,88	207371,1	437302,2

Var. T. Fisso	T = 10000	T = 20000	T = 30000
N = 10 P = 0.125	55554,88	185663,2	238771,4

Var. T. Adattivo	T = 10000	T = 20000	T = 30000
N = 10 P = 0.125	86959,31	199802,2	311502,3

Si può notare che il tempo, ovviamente, diminuisce all'aumentare della finestra e aumenta all'aumentare della probabilità di errore. Inoltre, dato che nei nostri test il list inviava pochi messaggi, il timeout adattivo impiega più tempo del timeout fisso poiché ha bisogno di inviare più messaggi per adattarsi all'effettivo ritardo di rete. Ovviamente il tempo aumenta all'aumentare del timeout iniziale.

5.2. Get

Var. finestra	N = 1	N = 3	N = 5	N = 10	N = 20
P = 0.125 T = 10000 fisso	1168714	831703,2	670828,6	579793,9	339906,5

Var. prob.	P = 0	P = 0.125	P = 0.25	P = 0.5
N = 10 T = 10000 fisso	110879,3	579793,9	977265	2159163

Var. T. Fisso	T = 10000	T = 20000	T = 30000
N = 10 P = 0.125	579793,9	889450,9	1265725

Var. T. Adattivo	T = 10000	T = 20000	T = 30000
N = 10 P = 0.125	475458,1	464810,1	479795,5

Si può notare che si comporta come il List al variare di N, P, e T, ma il timeout adattivo è differente: adattando il timeout si nota che per tutti e tre i valori di T il tempo è simile.

5.3. Put

Var. finestra	N = 1	N = 3	N = 5	N = 10	N = 20
P = 0.125 T = 10000 fisso	1189365	842627,9	690906,3	496387,4	369610,2

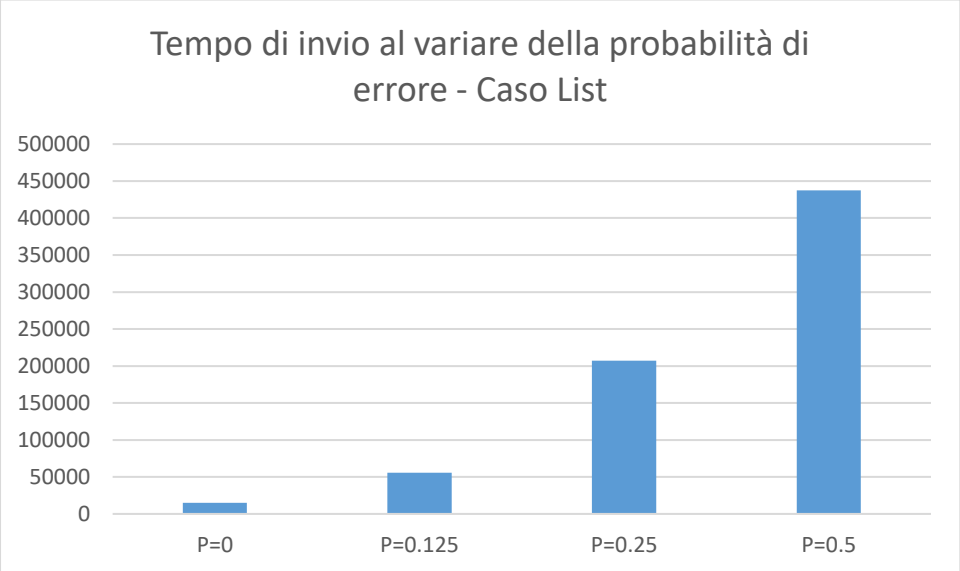
Var. prob.	P = 0	P = 0.125	P = 0.25	P = 0.5
N = 10 T = 10000 fisso	160262,4	496387,4	828637,9	2318396

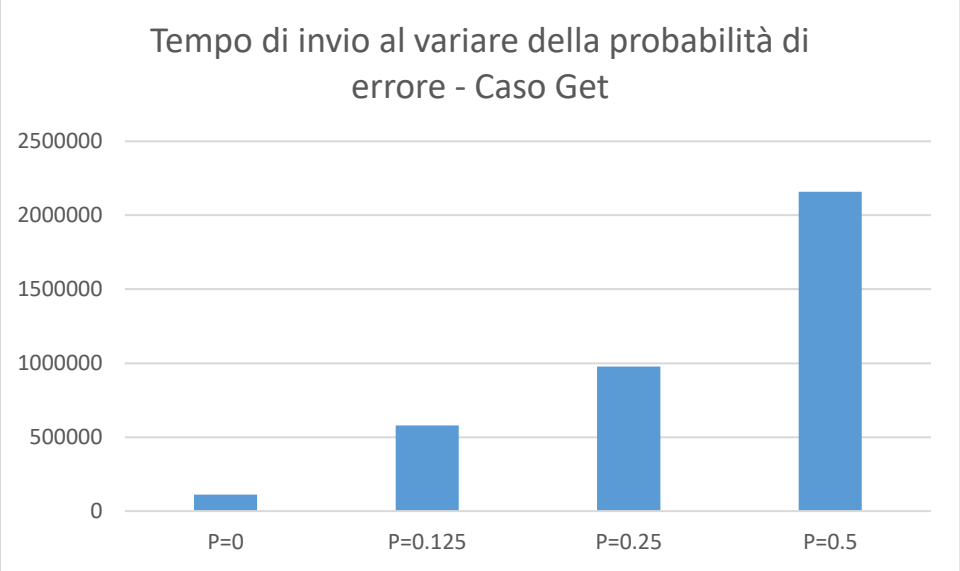
Var. T. Fisso	T = 10000	T = 20000	T = 30000
N = 10 P = 0.125	496387,4	966866,8	1487836

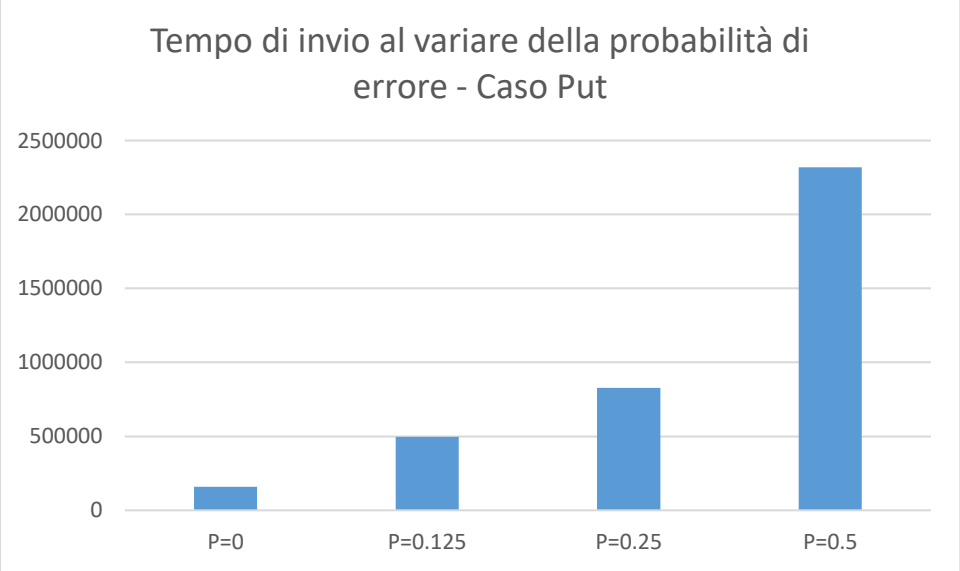
Var. T. Adattivo	T = 10000	T = 20000	T = 30000
N = 10 P = 0.125	438179,5	496912,7	454558,9

Essendo la funzione speculare del Get, i tempi tra i due sono simili.

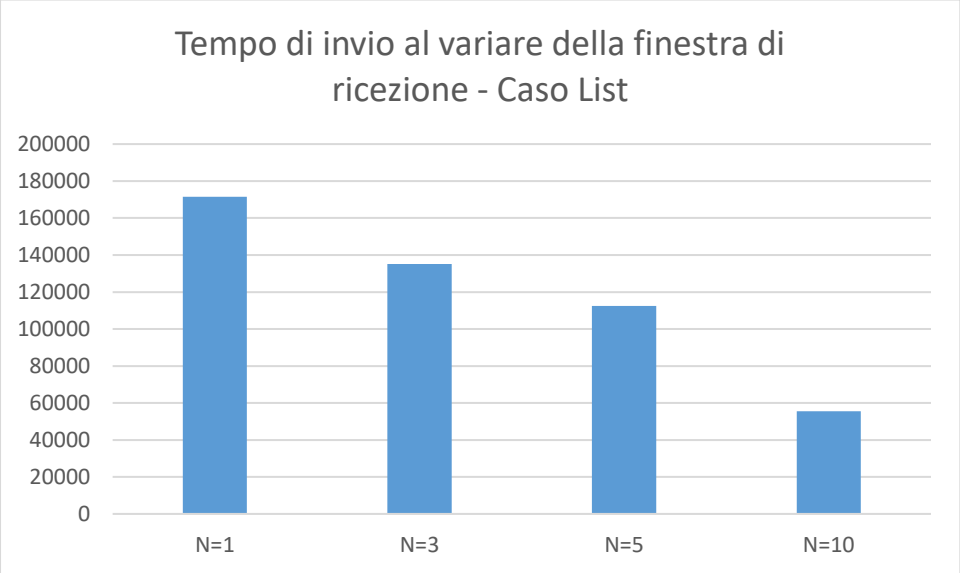
5.4. Test al variare della probabilità di errore

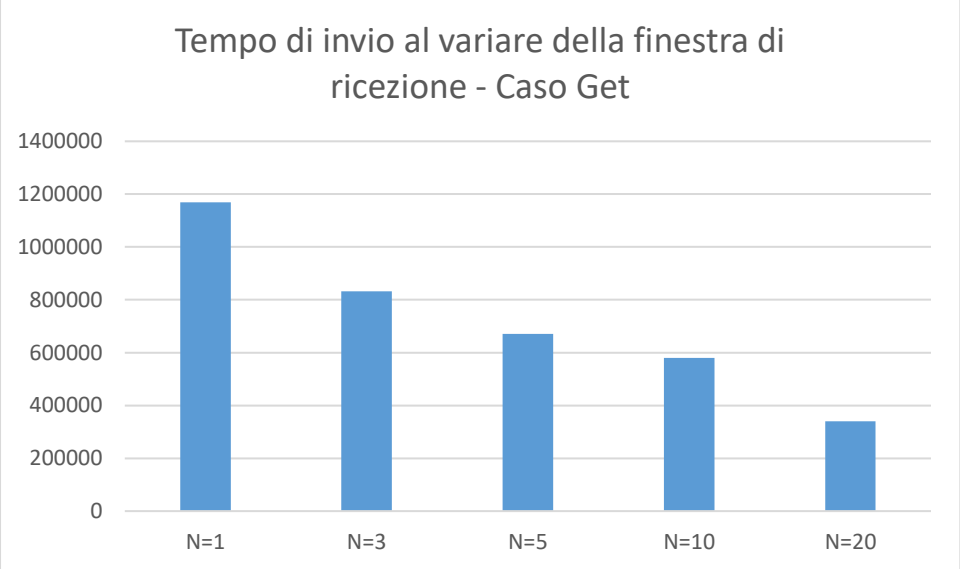


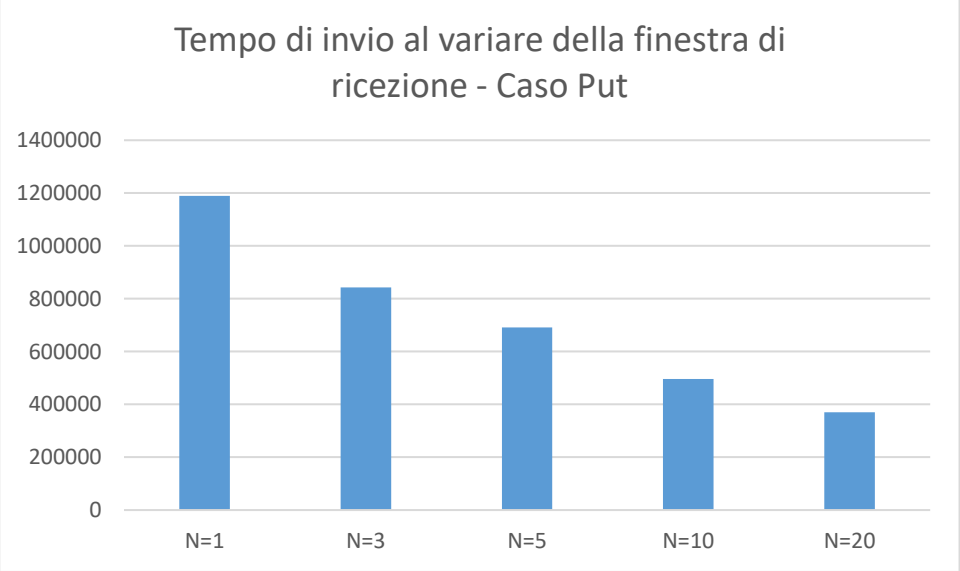




5.5. Test al variare della dimensione della finestra







5.6. UDP vs TCP file transfer

SR UDP	TCP
110879,3	18532,67

Anche nel caso di TCP è stato trasferito un file di dimensione 1 MB. Si può notare dai valori della tabella sopra che il TCP è molto più veloce del Selective Repeat UDP (un ordine di grandezza)

È stato considerato, per il tempo di SR UDP, il test con P=0, N=10, T=10000.

6. Conclusioni

In questo progetto è stato riscontrato che l'utilizzo del timeout adattivo rende più efficiente lo scambio di file, rispetto al caso di timeout fisso. Come previsto, la probabilità di perdita influisce negativamente sulle prestazioni a differenza della dimensione della finestra, che invece aumenta la velocita di invio. Le prestazioni che alla fine siamo stati in grado di ottenere si distanziano solo di un ordine di grandezza rispetto a quelle del TCP.