Università di Roma Tor Vergata

INGEGNERIA

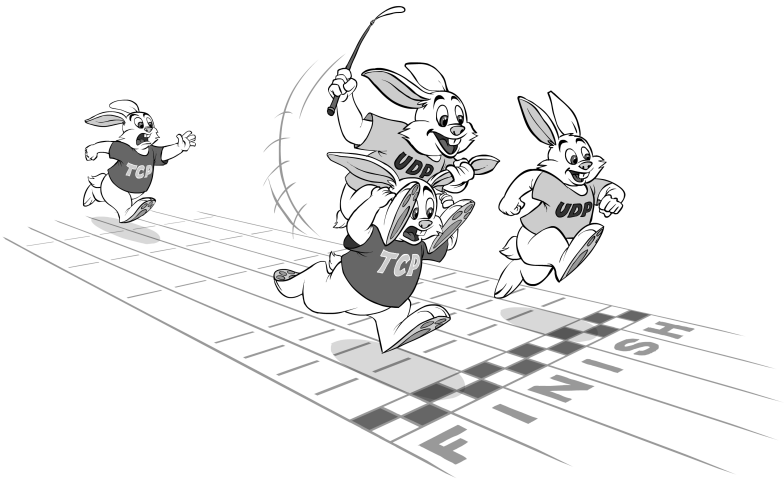
Università degli Studi di Roma Tor Vergata

PROGETTO DI INGEGNERIA DI INTERNET E WEB

TRASFERIMENTO FILE SU UDP

di

Chiaretti Riccardo – Di Cosmo Giuseppe – Nedia Salvatore



INDICE

Sommario

1 INTRODUZIONE

Il progetto che descriveremo di seguito è stato realizzato come parte dell'esame di Ingegneria di Internet e Web commissionato dal professore, responsabile di tale disciplina, Francesco Lo Presti.
La realizzazione del codice è stata effettuata tramite l'editor di testo Sublime Text in ambiente Linux come richiesto dalle specifiche, mentre la sua esecuzione ed i vari casi di test sono stati realizzati tramite la bash di Ubuntu.

2 DESCRIZIONE DEL PROBLEMA

L'intento del progetto è quello di ideare ed implementare in linguaggio C, usando l'API del socket di Berkley, un'applicazione client-server per il trasferimento affidabile di file che utilizzi a livello di trasporto il protocollo non orientato alla connessione UDP(User Datagram Protocol).
La comunicazione tra client e server deve avvenire tramite un protocollo che, oltre ad implementare l'affidabilità, deve prevedere la trasmissione di due tipi di messaggi:

- messaggi di comando: inviati dal client al server per effettuare determinate operazioni come l'upload di un file su server(comando PUT), il download di un file(comando GET) e la richiesta della lista dei file presenti sul server(comando LIST).
- messaggi di risposta: inviati dal server al client contenenti l'esito dell'esecuzione dei comandi.

Al fine di garantire un servizio affidabile il protocollo deve essere implementato a livello applicativo seguendo le regole del protocollo di tipo selective repeat(finestra di spedizione, timer di ritrasmissione, ecc..).

3 ARCHITETTURA DEL SISTEMA

3.1 LATO SERVER

L'architettura utilizzata lato server prevede inizialmente un solo processo in esecuzione che accetta connessioni attraverso una propria socket, ad una data porta pre-assegnata e tale fino alla fine della sua esecuzione. Una volta ricevuta una richiesta di connessione da parte di un client la gestione delle richieste di quest'ultimo, viene effettuata da un processo figlio
creato dal processo iniziale ogni volta che un nuovo client effettua una connessione.
Al momento della creazione il processo padre comunica al figlio tutte le informazioni necessarie per la corretta gestione del client, dopodiché torna ad accettare connessioni dalla propria socket. Nel momento in cui il client decide di terminare la connessione, per liberare risorse, il processo figlio termina, segnalandolo al padre, la sua esecuzione.

3.2 LATO CLIENT

Al momento dell'esecuzione il client è composto da un solo processo che tenta di effettuare la connessione, tramite la propria socket la cui porta è assegnata dal sistema operativo tra quelle disponibili, al server il cui indirizzo IP viene specificato da linea di comando. Dopo aver instaurato la connessione, tale processo crea un processo figlio il cui compito è quello di inviare pacchetti al server ed eseguire le routine di ogni comando.
Il processo padre durante l'esecuzione gestisce sia lo standard input attraverso il quale riceve i comandi digitati dall'utente, sia la ricezione di pacchetti che vengono poi opportunamente passati, tramite pipe, al figlio e gestiti differentemente in base al comando corrente(dato che i comandi sono sequenziali).

4 SCELTE PROGETTUALI

4.1 GESTIONE DELLA CONCORRENZA

Il server ed il client, di tipo concorrente, sono stati realizzati utilizzando un approccio multi-processo dato che, dovendo gestire ogni client indipendentemente dagli altri, non era necessario un criterio multi-thread con cui la comunicazione inter-process sarebbe stata meno complessa. Inoltre, per quanto riguarda il server, l'utilizzo di un unico processo sarebbe stato sconveniente poiché, dovendo effettuare il multiplexing dell'I/O tramite la funzione select()[1], la gestione contemporanea di troppi file descriptors avrebbe
potuto portare a rallentamenti e a spiacevoli comportamenti della select() stessa.

4.2 STRUTTURA DEL PACCHETTO UDP AFFIDABILE

Come mostra la Figura 1, il payload del segmento UDP a livello di trasporto è stato opportunamente modificato per riuscire ad implementare il trasferimento affidabile a livello applicativo; il nuovo payload contiene:

- Il campo **sequence number** e il campo **ack number** entrambi di 32 bit, utilizzati dal mittente e dal destinatario UDP per implementare il trasferimento dati affidabile.

[1] select(): funzione definita nella libreria <sys/select.h> che permette di monitorare più file descriptors.

4. Scelte progettuali

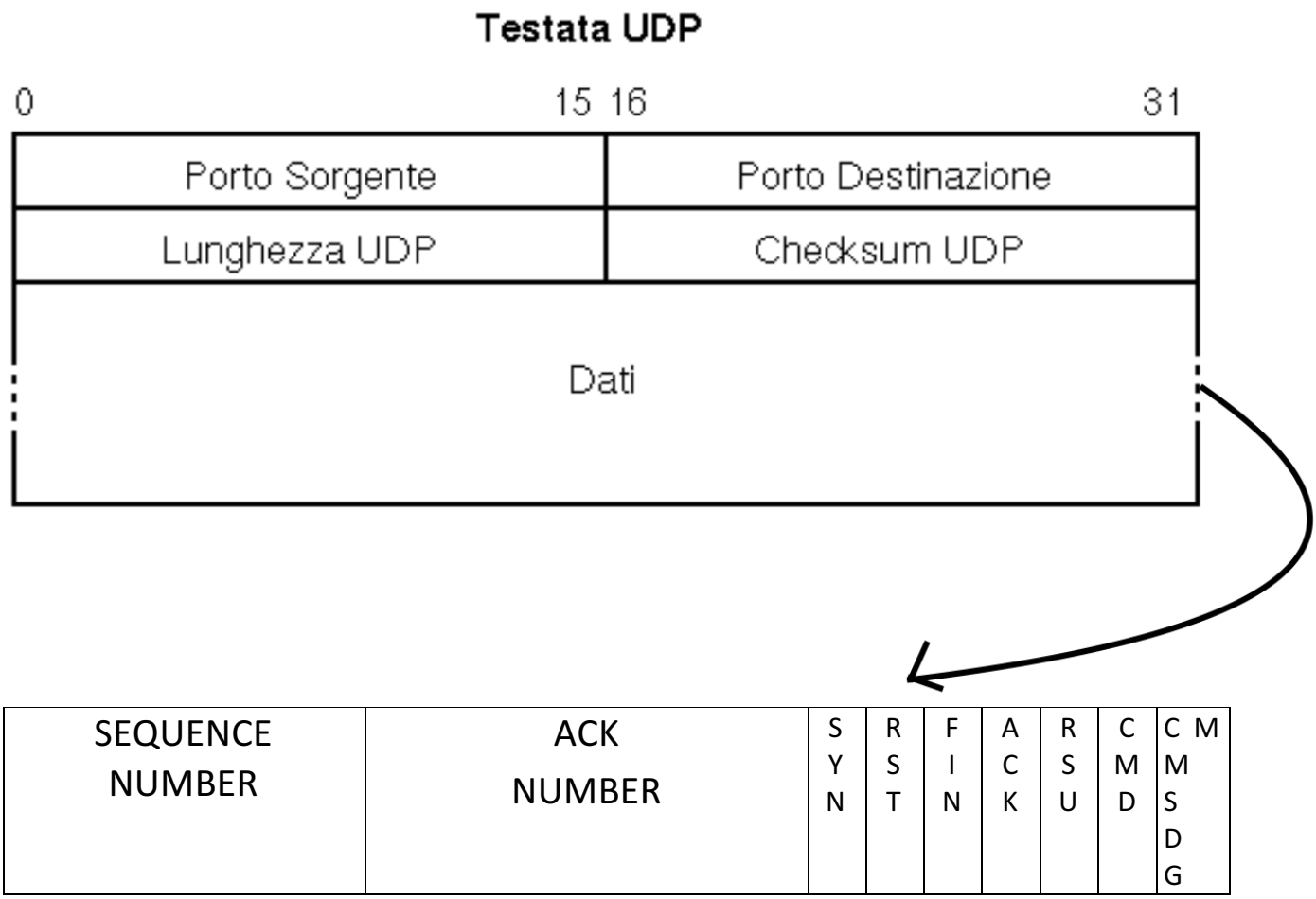


Figura 1: Struttura segmento UDP completo dell'affidabilità

- Il campo **flags** di 8 bit contenente:
  - Il bit *SYN:* utilizzato per stabilire la connessione tra client e server;
  - Il bit *RST:* impostato in situazioni di errore;
  - Il bit *FIN:* utilizzato nella chiusura della connessione;
  - Il bit *ACK:* indica che è stato ricevuto il riscontro di un pacchetto precedentemente inviato;
  - Il bit *RSU:* se impostato il client ha ricevuto un pacchetto di risposta ad uno dei comandi;
  - Il bit *CMD:* se impostato identifica un messaggio di richiesta di un comando;

- I bit *CDM_MSG(*2 bit*):* utilizzati per distinguere tra loro i tre comandi; se 01 è un messaggio di list, se 10 è un messaggio di get infine se 11 di put.

- Il campo payload di 1483 byte contenente i dati da trasportare

5 IMPLEMENTAZIONE

5.1 INSTAURAZIONE DELLA CONNESSIONE

Essendo UDP un protocollo di tipo connectionless, è stato implementato un meccanismo di connessione come in Figura 2.

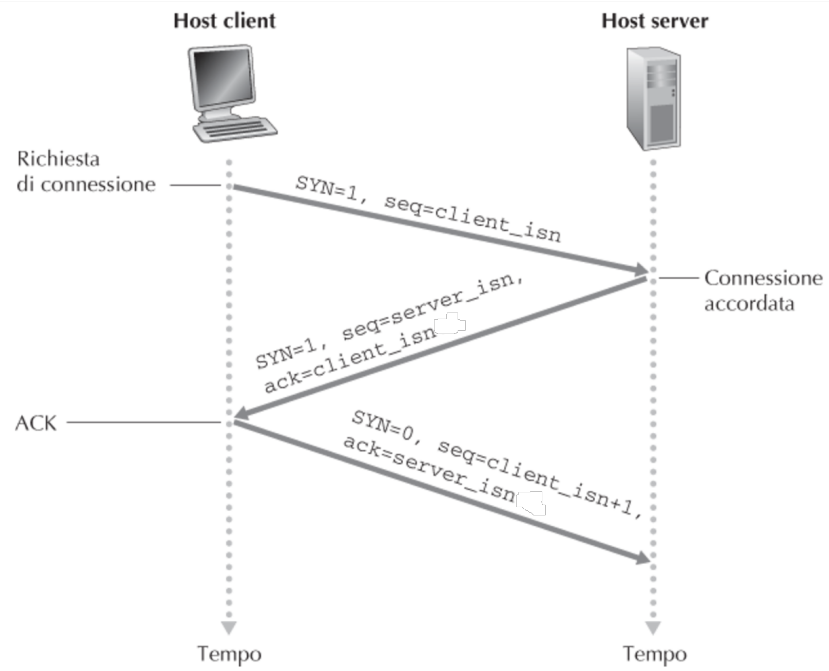


Figura 2: Instaurazione della connessione in UDP affidabile

Riprendendo il protocollo TCP, l'instaurazione della connessione avviene secondo un handshake a tre partendo dal client.

Lato client viene inviato un pacchetto di SYN con numero di sequenza generato casualmente, numero di ack impostato a 0 e bit di SYN uguale a 1.
Lato server, il processo padre, accetta richieste di connessione sulla sua socket("welcome socket"); una volta ricevuto un pacchetto, controlla se effettivamente esso corrisponde ad un SYN e in tal caso si assicura che tale client non sia già gestito da un processo figlio all'interno del sistema. Quest'ultimo controllo è necessario dato che, a causa della perdita di pacchetti all'interno della rete, se uno stesso client inviasse più volte un pacchetto di SYN il server allocherebbe risorse tante volte quanti sono i SYN ricevuti anche se si tratta dello stesso client.
Se i vari controlli sono andati a buon fine il processo padre costruisce un pacchetto di SYN-ACK, ne delega l'invio al figlio, e ritorna ad "attendere" la socket nel caso ricevesse altre richieste di connessione.
A questo punto entra in gioco il processo figlio che come prima operazione invia il pacchetto di SYN-ACK ricevuto dal padre; tale pacchetto ha il numero di sequenza generato casualmente, come numero di ack *client_isn*(Figura 2) e tra i bit di flag ha quello di SYN e quello di ACK impostati ad 1. Come terzo ed ultimo passo dell'handshake il client dopo aver ricevuto il SYN-ACK invia a sua volta un ACK che dopo esser ricevuto dal client conferma l'instaurazione della connessione.

5.2 CHIUSURA DELLA CONNESSIONE

Nel momento in cui il client decide di chiudere la connessione, invia al server un particolare pacchetto con bit di FIN impostato ad 1(pacchetto di FIN) la cui ricezione lato server comporta l'invio di un cosiddetto FIN-ACK(pacchetto in cui sia il bit di FIN che di ACK sono impostati ad 1). Nell'istante in cui invia il FIN-ACK, il server inizializza un timer, scaduto il

quale, invia un FIN(l'uso del timer è per permettere al client di ricevere giustamente l'ack).

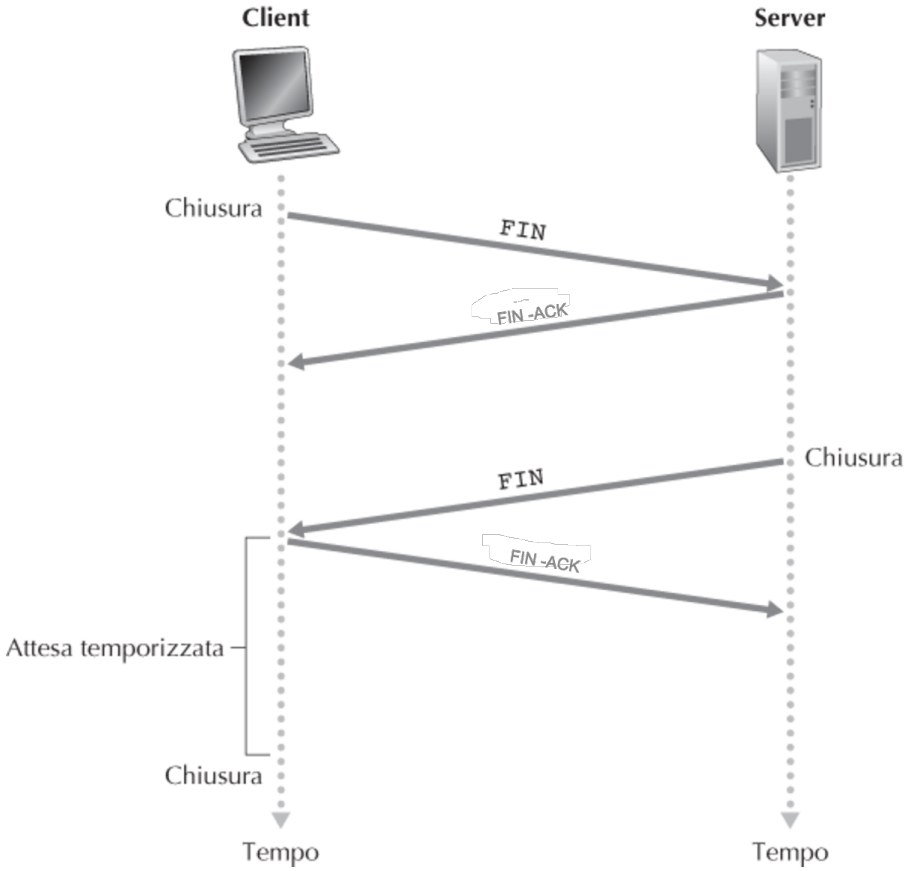


Figura 3: Chiusura della connessione in UDP affidabile

Quando il client riceve il FIN invia un FIN-ACK e dopo l'attesa di un breve periodo di tempo chiude la connessione liberando le risorse; lato server la chiusura avviene alla ricezione del FIN-ACK oppure al verificarsi della scadenza di un timer avviato dopo l'invio del FIN.

5.3 COMANDI PUT, GET E LIST

Uno dei compiti del client è quello di inviare messaggi di comando al server; in particolare, come citato nel Paragrafo 3.2, è il processo padre che dallo standard input legge comandi e li invia al figlio per la loro esecuzione.
Onde evitare race conditions sui file, nel momento in cui si effettua un'operazione su file, viene acquisito un lock che in base ai vari casi può essere in lettura, scrittura o entrambe.
Per organizzare i file presenti nel server si gestisce una serie di directory strutturate ad albero binario(vedi Figura 4) in cui la ricerca dei documenti presenti al suo interno avviene seguendo un algoritmo di ricerca binaria: ogni file che viene salvato dal server, viene inserito all'interno della cartella corrispondente all'iniziale del proprio nome.

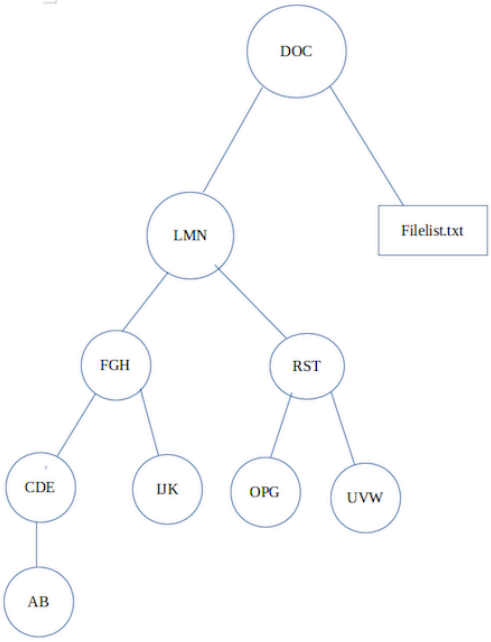


Figura 4: la figura mostra la disposizione ad albero delle cartelle

5.3.1 COMANDO PUT

Dopo aver ricevuto dallo standard input il comando di put, il client invia un pacchetto di comando con numero di ack uguale a 1, flag di CMD impostato ad 1 e di CMD_MSG impostato a 11; inoltre il payload contiene rispettivamente numero di pacchetti, precedentemente calcolati nella frammentazione del file, previsti per lo scambio, nome del file[2], e dimensione dell'ultimo pacchetto[3].
Il server, dopo aver ricevuto il pacchetto di (richiesta)put, si preoccupa di inizializzare tutte le strutture dati necessarie per lo scambio affidabile ed evitare che il ripetuto invio del pacchetto di put porti la ri-allocazione di risorse.
Come da figura il server poi invia un PUT-ACK il cui riscontro da parte del client comporta l'inizio dello scambio del file.
Al termine del trasferimento, quando il server ha ricevuto tutti i pacchetti, viene spedito un messaggio PUT-RSU con lo scopo di informare il client dell'esito dell'operazione.

[2] Con il quale il server salverà il file nel caso non esistesse altrimenti, invierà un pacchetto di errore.
[3] Tale informazione è necessaria per conoscere il numero preciso di bytes poiché l'ultimo pacchetto può avere dimensione minore rispetto alla massima lunghezza del payload.

5. Implementazione

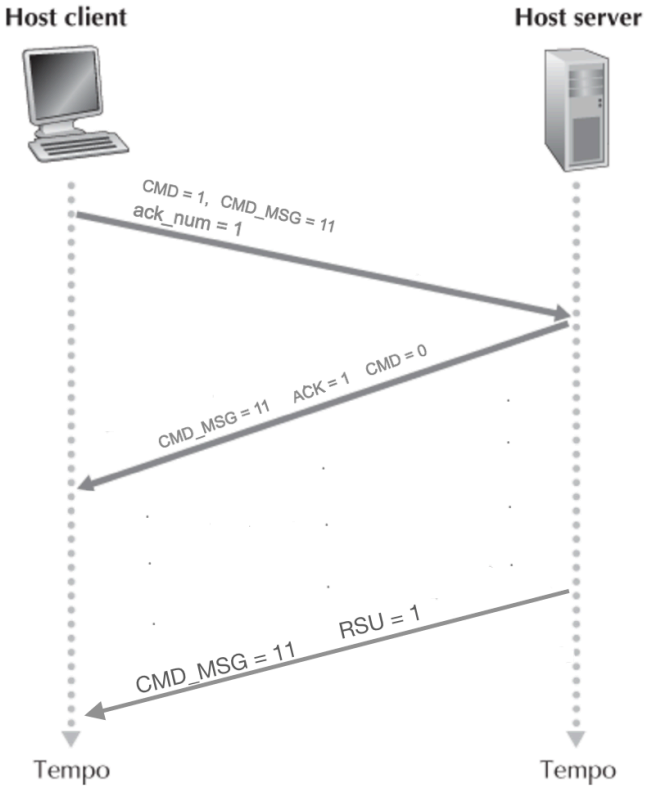


Figura 5: Scambio di pacchetti durante una PUT

TIMER	FINESTRA	PROB. DI PERDITA	TEMPO DI TRASFERIMENTO(s)
ADATTATIVO	3	0%	1.15
ADATTATIVO	3	50%	3.67
ADATTATIVO	3	80%	7.25
ADATTATIVO	80	0%	0.66
ADATTATIVO	80	50%	3.15
ADATTATIVO	80	80%	5.49
FISSO 3s	3	0%	3.5
FISSO 3s	3	50%	7.1
FISSO 3s	3	80%	30.86
FISSO 3s	80	0%	2.18
FISSO 3s	80	50%	6.82
FISSO 3s	80	80%	28.96
FISSO 5s	3	0%	4.06
FISSO 5s	3	50%	7.28
FISSO 5s	3	80%	38.72
FISSO 5s	80	0%	4.48
FISSO 5s	80	50%	6.6
FISSO 5s	80	80%	37.3

Figura 6: La figura mostra mediamente le prestazioni del comando PUT

5.3.2 COMANDO GET E LIST

Nel caso in cui il comando ricevuto sia GET il client ed il server iniziano(come da Figura 6) un handshake a 3 vie[4] necessario affinché entrambi allochino tutte le risorse utili per lo scambio.

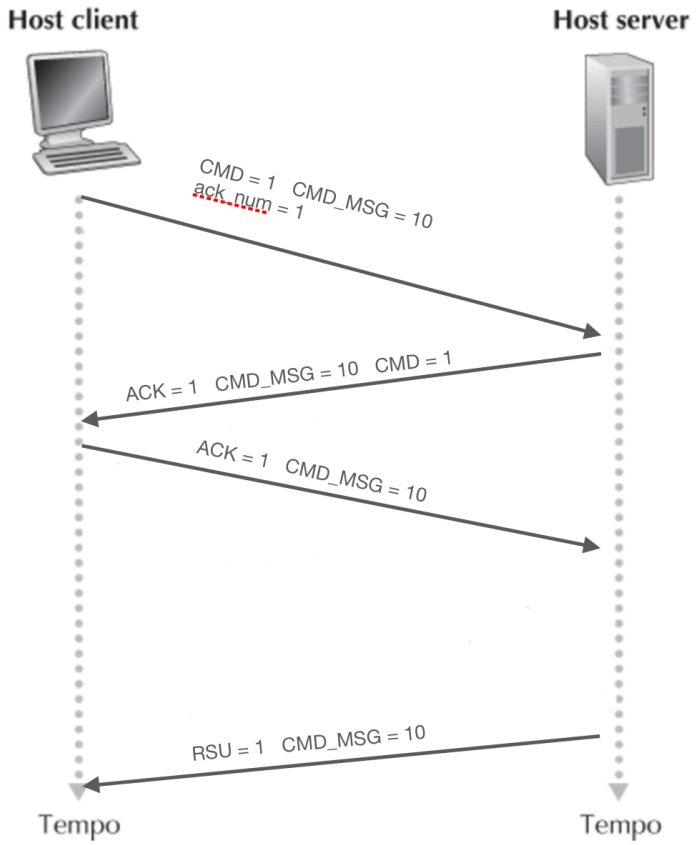


Figura 7: Scambio di pacchetti durante una GET

Se il server riceve una richiesta di un file che non possiede, viene inviato un opportuno messaggio di errore.

[4] A differenza del caso di put, in cui è il client ad essere il mittente, nella get è il server quindi prima di poter inviare i dati del file vero e proprio, deve accertarsi che il client abbia inizializzato tutte le strutture dati necessarie.

Dopo tale meccanismo viene avviato da parte del mittente il trasferimento del file richiesto. Completato l'invio di quest'ultimo, il server invia un messaggio di risposta contenete l'esito dell'operazione.

TIMER	FINESTRA	PROB. DI PERDITA	TEMPO DI TRASFERIMENTO(s)
ADATTATIVO	3	0%	0.3
ADATTATIVO	3	50%	1.5
ADATTATIVO	3	80%	2.57
ADATTATIVO	80	0%	0.53
ADATTATIVO	80	50%	0.66
ADATTATIVO	80	80%	1.9
FISSO 3s	3	0%	2.22
FISSO 3s	3	50%	6.45
FISSO 3s	3	80%	22.65
FISSO 3s	80	0%	3.5
FISSO 3s	80	50%	10
FISSO 3s	80	80%	20.32
FISSO 5s	3	0%	2.26
FISSO 5s	3	50%	10.65
FISSO 5s	3	80%	25.96
FISSO 5s	80	0%	7.14
FISSO 5s	80	50%	15.19
FISSO 5s	80	80%	23.02

Figura 8: La figura mostra mediamente le prestazioni del comando GET

Per quanto riguarda il comando di LIST la routine è la stessa del precedente con l'unica differenza che, a parte i flag che li contraddistinguono, il payload del primo pacchetto dell'handshake non contiene informazioni utili essendo il file richiesto sempre la lista dei documenti presenti nel server.

5. Implementazione

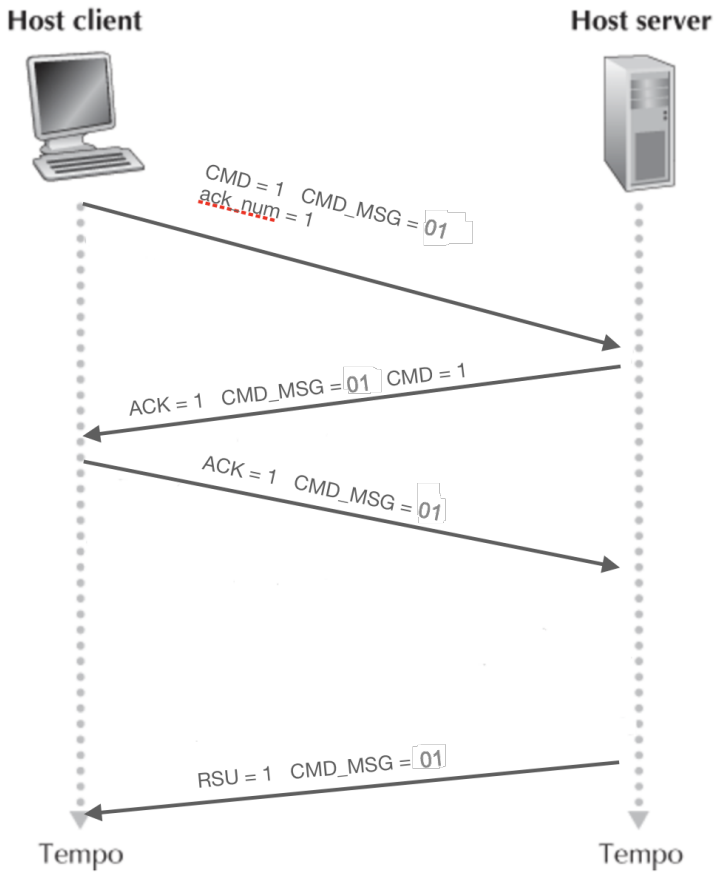


Figura 9: Scambio di pacchetti durante una LIST

TIMER	FINESTRA	PROB. DI PERDITA	TEMPO DI TRASFERIMENTO(s)
ADATTATIVO	3	0%	0.18
ADATTATIVO	3	50%	0.44
ADATTATIVO	3	80%	3.48
FISSO 3s	3	0%	0.29
FISSO 3s	3	50%	3.28
FISSO 3s	3	80%	10.01
FISSO 5s	3	0%	0.11
FISSO 5s	3	50%	10.21
FISSO 5s	3	80%	16.39

Figura 10: La figura mostra mediamente le prestazioni del comando LIST

5.4 TIMER DI RITRASMISSIONE

Per la ritrasmissione i timer sono stati implementati sfruttando le funzionalità della select(). Infatti ogni volta che avviene l'invio di un pacchetto viene assegnato ad esso un timer sfruttando le funzioni messe a disposizione dalla libreria <sys/timerfd.h>.
In particolare come primo passo, viene creato un file descriptor tramite la funzione timerfd_create() dopodiché gli viene assegnato un evento di timeout utilizzando la timerfd_settime() ed infine tale descrittore di file viene aggiunto all'insieme degli fd controllati dalla select().
Nel caso in cui il generico pacchetto venga riscontrato allora il timer ad esso associato verrà eliminato liberando risorse per il sistema operativo, altrimenti se si verifica un evento di timeout tale pacchetto verrà ri-inviato collegando ad esso un nuovo timer.
Oltre alla possibilità di utilizzare un periodo di timeout fisso, c'è anche quella di utilizzare un timer adattativo[5] che viene ricalcolato in base ai ritardi della rete; e formule utiizzate per effettuare ciò sono le seguenti:

```
clock_gettime(CLOCK_REALTIME, &now);/*Ottengo i secondi e nanosecondi correnti*/
long sample_rtt = now.tv_nsec - check_pkt->dyn_timer;
ext_rtt = (1 - 0.125) * ext_rtt + 0.125 * sample_rtt;
dev_rtt = (1 - 0.25) * dev_rtt + 0.25 * abs(sample_rtt - ext_rtt);
timeout = (ext_rtt + 4 * dev_rtt);
```

Figura 11: Implementazione timeout adattativo

Come da Figura 7:

- sample_rtt: rappresenta il tempo che intercorre tra l'istante dell'invio del pacchetto e quello della ricezione del suo riscontro;
- ext_rtt: è una media esponenziale ponderata dei valori di sample_rtt;
- dev_rtt: è invece il discostamento del valore sample_rtt dalla sua media(ext_rtt);

[5] Si noti che il timer adattativo è stato calcolato riprendendo la strategia adottata dal protocollo TCP

- timeout: infine questo valore sarà quello che verrà assunto dal timer adattativo.

5.5 TRASFERIMENTO AFFIDABILE

Come citato nel Paragrafo 2, il trasferimento affidabile è stato implementato seguendo le regole del protocollo di ripetizione selettiva; per questo sia lato client che lato server sono state utilizzate strutture dati e funzioni che permettessero la realizzazione di tale protocollo.

```
struct pkt_wndw{
    struct header *pkt;
    unsigned char ack;//0 if not acknowledged; 1 if already acknowledged
    int time_fd;
    long dyn_timer;
    unsigned char jumped;
};
```

Figura 12: Contenuto della struct pkt_wndw

La struttura dati su cui si fonda l'implementazione è la *struct pkt_wndw* contenente, come da Figura 11, oltre al pacchetto:

- ack: che può assumere valore 0 se il pacchetto non è stato inviato, 2 se è stato inviato ma non ancora riscontrato e 1 se è stato riscontrato;
- time_fd: è il file descriptor del timer associato a quel pacchetto(vedere paragrafo 5.4);
- dyn_timer: valore utilizzato solo nel caso in cui il timer è di tipo adattativo. Esso viene aggiornato alla ricezione dei riscontri ed è settato come periodo di timeout;
- jumped: variabile che può assumere valore 0 se il pacchetto è stato inviato mentre 1 se esso è stato perso a "causa della perdita della rete".

Lato mittente le funzioni utilizzate per effettuare l'invio dei pacchetti all'interno della finestra ed il riscontro dei tali sono rispettivamente *snd_pkt_in_wnd()* e *ack_sender()*.
Quando sono disponibili pacchetti da inviare che ricadono all'interno della finestra(Figura 12), ossia tra nextseqnum e snd_base+N[6], la funzione *snd_pkt_in_wnd()* si occupa di inviarli popolando opportunamente la *struct pkt_wndw*; se si verifica un timeout è tra i compiti del mittente quello di effettuare, tramite la funzione *re_send(),* il rinvio e l'associazione di un nuovo timer.

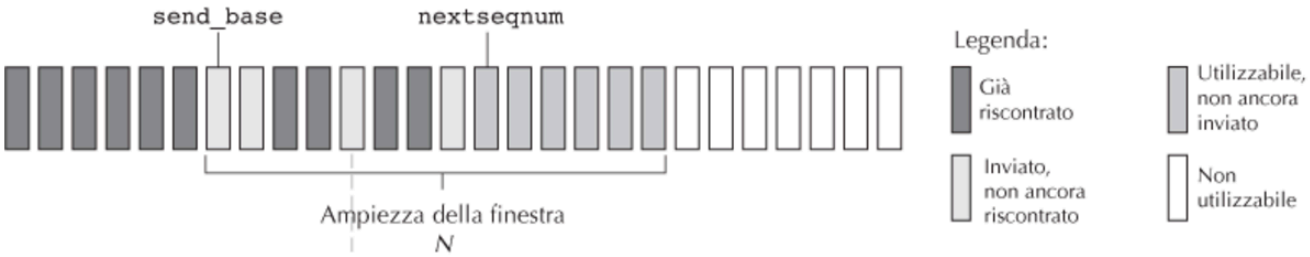


Figura 13:Visione del mittente sui numeri di sequenza

Alla ricezione di un ack, entra in gioco la funzione *ack_sender()* che si occupa di etichettare il pacchetto come ricevuto(ack = 1) ammesso che esso si trovi nella finestra. Inoltre, nel caso in cui il pacchetto riscontrato corrisponda con quello più vecchio di cui non sia stato ancora ricevuto il riscontro, viene spostata la base della finestra verso il pacchetto che non ha ricevuto ack con il più piccolo numero di sequenza.
Lato ricevente, dato che in un protocollo di tipo selective repeat le finestre del mittente e destinatario non sempre coincidono, viene creata a sua volta una finestra di ricezione come quella mostrata in Figura 13.

[6] nextseqnum è l'indice del prossimo pacchetto da inviare, snd_base è l'indice del più vecchio pacchetto che non è stato ancora riscontrato mentre N è l'ampiezza della finestra.

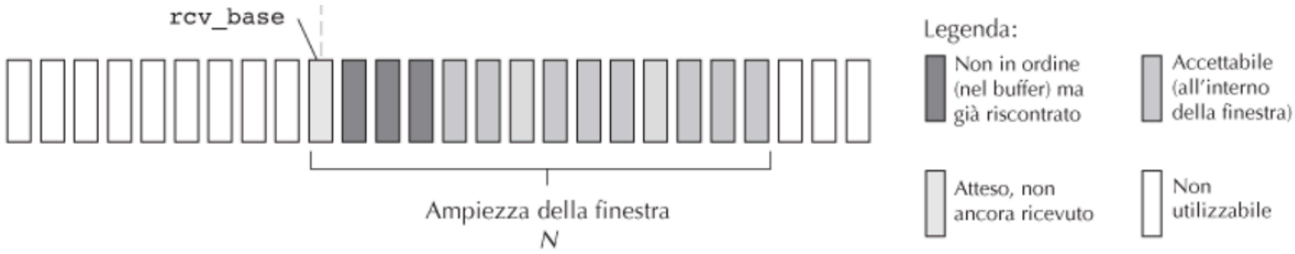


Figura 14: Visone del ricecente sui numeri di sequenza

La funzione utilizzata per riscontrare è *ack_reciver();* essa nel caso in cui venga ricevuto un pacchetto il cui numero di sequenza cade nell'intervallo [rcv_base, base+N-1] invia un ack selettivo salvando inoltre il contenuto dello stesso. Se viene riscontrato un pacchetto il cui numero di sequenza corrisponde all'indice rcv_base, allora la finestra di ricezione viene spostata in avanti.
Rispetto a quando un pacchetto sia già stato riscontrato, nel qual caso viene comunque generato un ack, quando ne viene ricevuto uno fuori finestra esso viene ignorato.

6 MANUALE DI INSTALLAZIONE, CONFIGURAZIONE ED ESECUZIONE

Per eseguire i programmi del client e del server, dopo essersi opportunamente spostati nella cartella UDP_affidabile/project, bisogna effettuare i seguenti passi:

1. Digitare a linea di comando make server e make client per creare gli eseguibili server_udp e client_udp;
2. Digitare ./server_udp per eseguire il server e ./client_udp <server IP address> per eseguire il client, dove <server IP address> deve essere opportunamente sostituita dall'indirizzo IP del server;

6. Manuale di installazione, configurazione ed esecuzione

I parametri configurabili nei due programmi sono:

- Finestra di spedizione/ricezione: per modificarla è necessario entrare nel file basic.h e cambiare il valore della costante SLIDING_WNDW;
- Probabilità di perdita: modificare la costante failure_prob nel file basic.h;
- Timer fisso: per attivarlo bisogna porre la variabile is_dyn = 0 in client.c/server.c, la variabile timeout = 0 in timer.c dopodiché impostare la variabile timer_fisso al valore desiderato;
- Timer dinamico: per scegliere il timer adattativo è necessario porre, a differenza del precedente caso, la variabile is_dyn = 1 in client.c/server.c, la variabile timeout in timer.c pari al valore desiderato e la variabile timer_fisso = 0;

Per eseguire i comandi ammessi(vedere Paragrafo 7 per ulteriori immagini dimostrative):

- LIST:
    - Digitare LIST e, dopo aver premuto Invio, verrà mostrata la lista dei file sul server.
- PUT:
    - Digitare PUT e premere Invio;
    - Successivamente digitare il nome del file di cui si vuole effettuare l'upload e premere Invio.
- GET:
    - Digitare GET e premere Invio;
    - Successivamente digitare il nome del file di cui si vuole effettuare il download e premere Invio.

- END:
  - Dopo aver digitato END e premuto invio, il client chiuderà la connessione col server.

7 ESEMPI DI FUNZIONAMENTO

7.1 APERTURA CONNESSIONE

Lato client:

```
riccardochiaretti@ubuntu:~/Desktop/UDP_affidabileBuono$ ./client 127.0.0.1
Ok, you are now connected!

Insert a command! Possible commands are: LIST, GET, PUT
```

Figura 15: connessione del client al server nel momento d'esecuzione del programma

Lato server:

```
riccardochiaretti@ubuntu:~/Desktop/UDP_affidabileBuono$ ./server
Connected with IP: 127.0.0.1 and port: 30399
```

Figura 16: il server ha instaurato una connessione con il client il cui indirizzo IP e di porta sono stampati a schermo

7.2 COMANDO LIST

Lato client:

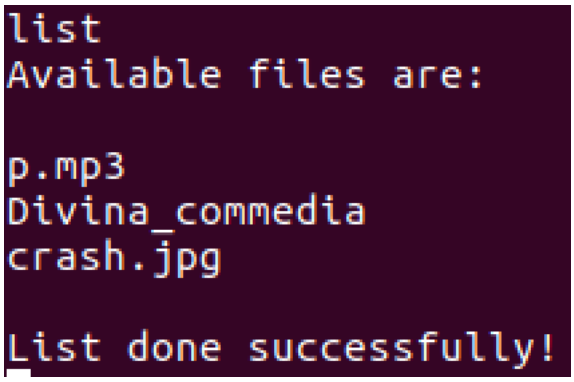


Figura 17: dopo aver digitato il comando list il client ottiene la lista dei file presenti nel server con un messaggio di risposta

Lato server:

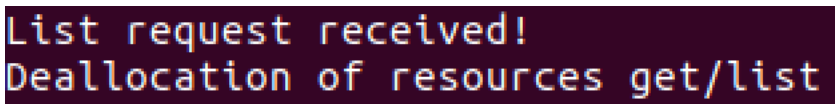


Figura 18: dopo aver opportunamente gestito il comando ricevuto, il server dealloca le risorse

7.3 COMANDO PUT

Lato client:

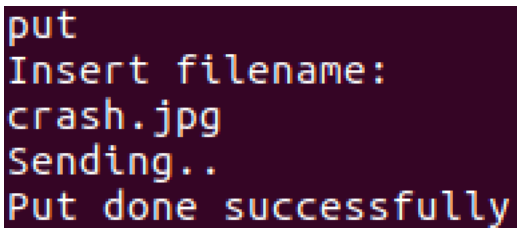


Figura 19: dopo aver digitato il comando PUT e il file da caricare, il client ottiene l'esito dell'operazione

```
put
Insert filename:
crash.jpg
Sending..
File already exists!
```

Figura 20: nel caso in cui il client tentasse di fare l'upload di un file già esistente nel server, ottiene un messaggio d'errore

```
put
Insert filename:
pippo.txt
Not valid input file, try again
```

Figura 21: opportuna gestione dell'errore nel caso in cui il client tentasse di fare l'upload di un file che non possiede

Lato server:

```
Put request received!
Deallocating resources put
```

Figura 22: dopo aver gestito la richiesta di PUT, il server dealloca le risorse

7.4 COMANDO GET

Lato client:

```
get
Insert filename:

crash.jpg
Sending..
Get done successfully!
```

Figura 23: dopo aver digitato il comando GET e il file da scaricare, il client ottiene l'esito dell'operazione

```
get
Insert filename:
pippo.txt
Sending..
File you require does not exist
```

Figura 24: messaggio d'errore nel caso in cui il client tentasse di effettuare il download di un file non presente nel server

Lato server:

```
Get request received!
Deallocation of resources get/list
```

Figura 25: dopo aver portato a termine il comando GET, il server dealloca le risorse

7.5 CHIUSURA DELLA CONNESSIONE

Lato client:

```
end
Connection closed!
```

Figura 26: dopo aver digitato il comando END ed effettuato l'handshake a tre vie il client chiude la connessione col server

```
end
Connection timed out! Try to re-connect!
```

Figura 27: quando il client non riceve più risposte dal server, chiude la connessione per timeout

Lato server:

Figura 28: indica che il client ha chiesto la chiusura della connessione, di conseguenza vienne chiusa anche dal server

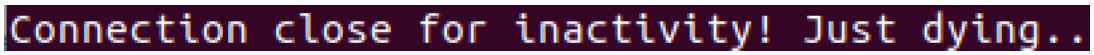


Figura 29: onde evitare memory leak dovuti alla mancata deallocazione di risorse, il server dopo un periodo di inattività chiude la connessione col client

8 LIMITAZIONI RISCONTRATE

Per quanto riguarda il timer dinamico è stata impostata una soglia minima e massima pari rispettivamente a 100000000 nanosecondi(0.1 secondi) e 900000000 nanosecondi(0.9 secondi).
Questo perché, siccome a livello implementativo era necessario lavorare in nanosecondi dato che i vari test sono stati effettuati su un solo host o al massimo tra host poco distanti, i tempi in gioco erano di un ordine molto minore dei secondi e nel caso in cui avesse assunto un valore molto alto, le funzioni di settaggio assumevano strani comportamenti.

```
if(timeout > 900000000) /*Limite superiore..*/
    timeout = 700000000;
if(timeout < 100000000)/*..ed inferiore per il timer adattativo*/
    timeout = 100000000;
```

Figura 30: limite minimo e massimo del timeout dinamico