

# **Università degli studi di Roma Tor Vergata**

Dipartimento di Ingegneria Informatica



Progetto finale del corso di  
Ingegneria di Internet e Web

## **Trasferimento file su UDP**

**Gruppo:**

Mauro Ficarella 0202804  
Valentina Sisti 0215192  
Martina Turbessi 0219834

Roma, 16/07/2019



# Indice

<b>1 INTRODUZIONE.....</b>	<b>5</b>
1.1 Specifiche del software.....	6
1.1.1 Comunicazione .....	6
1.1.2 Server.....	6
1.1.3 Client.....	7
1.1.4 Trasferimento affidabile .....	7
<b>2 RICHIAMI TEORICI.....</b>	<b>8</b>
<b>3 ARCHITETTURA DEL SISTEMA E SCELTE PROGETTUALI.....</b>	<b>9</b>
3.1 Connessione .....	9
3.2 Concorrenza.....	9
3.3 Affidabilità UDP .....	10
3.3.1 Pacchetto UDP affidabile.....	10
3.3.2 Finestra di spedizione .....	11
3.3.3 Ripetizione selettiva.....	11
3.3.4 Timer.....	11
<b>4 IMPLEMENTAZIONE .....</b>	<b>12</b>
4.1 Apertura e chiusura della connessione.....	12
4.1.1 Server.....	12
4.1.2 Client.....	13
4.2 List .....	14
4.2.1 Prestazioni .....	14
4.3 Get (download).....	15
4.3.1 Download affidabile .....	15
4.3.2 Prestazioni .....	16
4.4 Put (upload) .....	17
4.4.1 Upload affidabile .....	17
4.4.2 Prestazioni .....	17
<b>5 MANUALE DELL'INSTALLAZIONE, CONFIGURAZIONE ED UTILIZZO .....</b>	<b>18</b>
5.1 Eseguire il software .....	18
5.2 Configurazione dell'applicativo .....	18
5.3 Utilizzo .....	18

<b>6 ESEMPI DI UTILIZZO .....</b>	<b>19</b>
<b>6.1 Connessione del server.....</b>	<b>19</b>
<i>6.1.1 Connessione riuscita del server.....</i>	<i>19</i>
<i>6.1.2 Eccesso di connessioni del client.....</i>	<i>19</i>
<i>6.1.3 Gestione connessioni del server.....</i>	<i>20</i>
<i>6.1.4 Chiusura della connessione del server .....</i>	<i>20</i>
<b>6.2 Connessione del client.....</b>	<b>21</b>
<i>6.2.1 Connessione riuscita del client.....</i>	<i>21</i>
<i>6.2.2 Connessione non riuscita del client.....</i>	<i>21</i>
<i>6.2.3 Chiusura connessione del client .....</i>	<i>21</i>
<b>6.3 Controllo dell'input.....</b>	<b>22</b>
<i>6.3.1 Lato server.....</i>	<i>22</i>
<i>6.3.2 Lato client .....</i>	<i>22</i>
<b>6.4 Servizio di listing .....</b>	<b>23</b>
<i>6.4.1 Lato client .....</i>	<i>23</i>
<b>6.5 Servizio di download di un file.....</b>	<b>24</b>
<i>6.5.1 Inizio del download lato client .....</i>	<i>24</i>
<i>6.5.2 Inizio del download lato server.....</i>	<i>25</i>
<i>6.5.3 Fine del download lato client .....</i>	<i>25</i>
<i>6.5.4 Fine del download lato server.....</i>	<i>25</i>
<i>6.5.5 Ricezione pacchetti lato client .....</i>	<i>26</i>
<i>6.5.6 Invio dei pacchetti lato server.....</i>	<i>26</i>
<b>6.6 Servizio di upload di un file.....</b>	<b>27</b>
<i>6.6.1 Inizio dell'upload lato client .....</i>	<i>27</i>
<i>6.6.2 Inizio dell'upload lato server.....</i>	<i>27</i>
<i>6.6.3 Fine dell'upload lato client .....</i>	<i>28</i>
<i>6.6.4 Fine dell'upload lato server.....</i>	<i>28</i>
<i>6.6.5 Invio dei pacchetti lato client .....</i>	<i>28</i>

## 1. Introduzione

Questo progetto nasce con lo scopo di progettare ed implementare, in linguaggio C utilizzando l'API del socket di Berkley, un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione, usando socket di tipo "SOCK\_DGRAM" e come protocollo di trasporto il protocollo UDP (User Datagram Protocol).

Il software è stato realizzato parallelamente su tre macchine, due con kernel UNIX (MacOS 10.14.4) (Figura 1 e 2) e l'altra con kernel Linux (Ubuntu 18.10) (Figura 3). In tutti e tre i casi si è fatto uso di CLion come editor di testo e ambiente di sviluppo.

Di seguito sono riportate le specifiche tecniche dei tre computer su cui è avvenuta la progettazione, la creazione e il test del software.



*Figura 1*



*Figura 2*



*Figura 3*

## **1.1 Specifiche del software**

Il software deve permettere:

- Connessione Client – Server senza autenticazione;
- All’utente di poter vedere la lista di file disponibili su Server;
- All’utente di poter scaricare uno tra i file presenti sul Server;
- All’utente di poter caricare un file sul Server;
- Il trasferimento dei file (download – upload) in modo affidabile;
- La possibilità di configurare la dimensione della finestra di spedizione “N”, la probabilità di perdita dei messaggi “p” e la durata di timeout “T” (il timeout deve poter essere o fisso o additivo);
- L’esecuzione del Client ed del server nello stesso spazio utente senza richiedere privilegi di root;
- Al Server di essere in ascolto su una porta di default (configurabile).

### **1.1.1 Comunicazione**

La comunicazione all’interno dell’applicativo deve supportare una duplice tipologia di messaggi:

- Messaggi di comando scambiati dal Client verso il Server per richiedere l’esecuzione delle operazioni;
- Messaggi di risposta scambiati dal Client verso il Server in risponso all’esecuzione delle operazioni.

### **1.1.2 Server**

Il Server, di tipo concorrente, dovrà fornire le seguenti funzionalità:

- L’invio del messaggio di risposta al comando “list” del Client, ovvero il messaggio contenente la lista dei nomi dei file presenti sul Server e disponibili per il download da parte del Client;
- L’invio del messaggio di risposta al comando “get” (download), ovvero il messaggio contenente il file richiesto dal Client oppure, in caso quest’ultimo non dovesse essere disponibile, un opportuno messaggio di errore;
- La ricezione di un messaggio “put” (upload), ovvero la ricezione del messaggio dal Client contenente un nuovo file da caricare sul Server.

Ogni operazione del Server sarà accompagnata da messaggi di avvenuto completamento, o eventualmente di insuccesso, della stessa.

### **1.1.3 Client**

Il Client, di tipo concorrente, dovrà fornire le seguenti funzionalità:

- l'invio del messaggio di richiesta "list" al Server per ottenere la lista dei nomi dei file disponibili per il download.
- l'invio del messaggio di richiesta "get" (download) al Server per ottenere il file scelto tra la lista dei file ricevuta.
- la ricezione del file demandato al Server o, in caso di richiesta non completabile, la ricezione di un messaggio d'errore.
- l'invio del messaggio di "put" (upload) per effettuare il caricamento di un file sul Server.

Ogni operazione del client sarà accompagnata da messaggi di avvenuto completamento, o eventualmente di insuccesso, della stessa.

### **1.1.4 Trasferimento affidabile**

Sapendo che lo scambio di messaggi del protocollo UDP avviene usando un servizio di comunicazione non affidabile, al fine di garantire l'affidabilità della trasmissione in ricezione e spedizione è necessario che, da parte del Client e del Server, venga costruito a livello applicativo un meccanismo che renda affidabile il protocollo UDP.

La richiesta è quella di implementare il protocollo Selective Repeat con finestra di spedizione di dimensione "N".

Per testare un'eventuale perdita di messaggi durante una trasmissione, è necessario simulare avvenimenti di tal genere poiché è alquanto improbabile che vengano persi pacchetti in rete, tra Client e Server, quando questi sono eseguiti sullo stesso host.

Si assume, dunque, che il mittente scarti i messaggi con una probabilità pari a "p".

## 2. Richiami teorici

Si focalizzi l'attenzione sui protocolli che una rete TCP/IP mette a disposizione del livello applicativo; il primo è “UDP” (User Datagram Protocol), l'altro è “TCP” (Transmission Control Protocol).

UDP, quello preso in considerazione nel caso in questione, fornisce alle applicazioni un protocollo non orientato alla connessione e non affidabile; non garantisce quindi né l'arrivo dei pacchetti a destinazione, né la ritrasmissione in caso di perdita d'informazione, né la corretta sequenza; non è orientato alla connessione poiché dialoga in modo quasi diretto con IP, prendendo i messaggi dal processo applicativo per poi aggiungere numero di porta di origine e di destinazione (più altri piccoli campi); in seguito passa il segmento al livello di rete; non è affidabile poiché il protocollo trasforma il servizio di consegna IP “tra sistemi periferici” in quello di consegna “tra processi in esecuzione sui sistemi periferici” che avviene in modalità “best-effort”, ovvero il protocollo “fa del suo meglio” per recapitare il messaggio senza darci tuttavia certezze sull'avvenuta riuscita dell'invio.

Le socket UDP, a differenza delle socket TCP, non supportano la comunicazione di tipo “stream” (ovvero una comunicazione in cui si ha a disposizione un flusso di dati che può esser letto un po' alla volta), bensì una comunicazione di tipo “datagram”, in cui i dati arrivano in singoli blocchi e devono essere letti integralmente. Per questo motivo, quando vengono aperte queste socket UDP, bisogna aprirle con la funzione socket impostando per il tipo di socket il valore “SOCK\_DGRAM”.

Tutto ciò che avviene nella comunicazione tramite queste socket è la trasmissione di un pacchetto da un client ad un server e viceversa.

Nel nostro caso, avendo la necessità di utilizzare il protocollo UDP per l'applicativo e dovendo garantire l'affidabilità della trasmissione, non possiamo affidarci ai servizi offerti dal protocollo bensì dobbiamo elaborare e progettare dei meccanismi diversi per soddisfare la richiesta.

I meccanismi che possiamo implementare all'interno del nostro applicativo, per garantirci l'affidabilità della trasmissione dei messaggi, sono molteplici; nel nostro caso la logica che andremo ad implementare è quella della “ripetizione selettiva”.

Questo protocollo permette di ritrasmettere solo quei messaggi che non hanno ricevuto un riscontro di avvenuta ricezione (riscontro che viene inviato dal ricettore al mittente).

Avendo assunto che il protocollo rientra nella macro categoria dei protocolli con pipeline, per velocizzare il meccanismo di ritrasmissione introduciamo una finestra che ci permette il controllo del messaggio di avvenuta ricezione dei pacchetti, solo se inviati all'interno della stessa; una volta eseguito questo controllo, se presenti, rispediamo i pacchetti persi; nel momento in cui abbiamo ricevuto tutti i riscontri degli invii positivi, scorriamo la finestra ai successivi messaggi in attesa di invio.

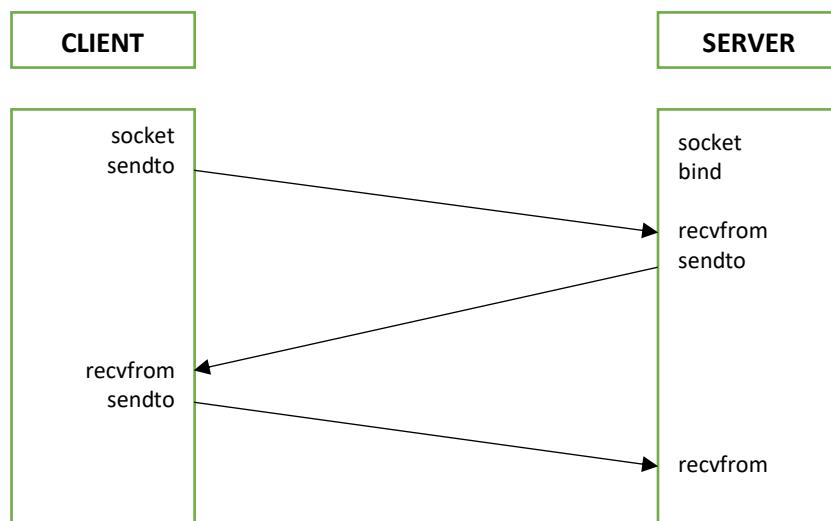
Per non rimanere bloccati nell'attesa di un riscontro d'invio che tarda ad arrivare o che, nel peggior caso, non arriverà mai, perché il pacchetto è stato definitivamente perso in rete, si introduce, nella logica di un meccanismo di ritrasmissione, il concetto di “timeout” allo scadere del quale il pacchetto inviato è considerato perso e quindi da rinviare.

### 3. Architettura del sistema e scelte progettuali

#### 3.1 Connessione

La connessione del client e del server del nostro applicativo avviene secondo lo standard delle API delle socket di Berkeley, configurate in modo tale che la connessione segua le direttive del protocollo UDP.

Essendo UDP un protocollo “connection-less”, non si esegue alcun tipo di hand-shaking tra le parti che vogliono entrare in collegamento. Una struttura generica di un server UDP (come in figura subito sotto) prevede, una volta creato il socket, la chiamata alla funzione “bind” per mettersi in ascolto dei dati (unica parte in comune con TCP); la ricezione dei dati dal client avviene attraverso la funzione “recvfrom” mentre una eventuale risposta sarà inviata tramite la funzione “sendto”; da parte del client invece una volta creato il socket non sarà necessario connettersi con “connect” ma si potrà effettuare direttamente una richiesta inviando un pacchetto con la funzione “sendto” e si potrà leggere una eventuale risposta con la funzione “recvfrom”.



*Figura 4: Schema di interscambio dei pacchetti per una comunicazione via UDP*

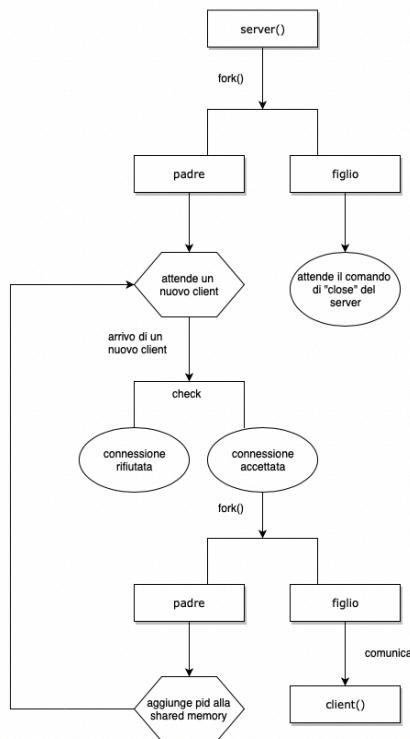
#### 3.2 Concorrenza

La concorrenza dell'applicativo viene assicurata da un meccanismo multiprocesso ottenuto mediante delle “fork()”.

La prima fork() viene eseguita nel main del server, dopo aver creato la connessione con cui i client possono collegarsi allo stesso. Il processo figlio di questa fork si mette in attesa del comando di “close” che chiude la connessione del server (questo argomento verrà affrontato largamente nel paragrafo 4.1); il processo padre invece procede con l'esecuzione vera e propria del server: qui verrà eseguita, da parte del padre, nel momento in cui un client si aggiungerà all'applicativo, un'ulteriore fork(); in seguito a questa nuova fork il relativo processo figlio si occuperà di instaurare la connessione e di procedere ai meccanismi per la comunicazione con il client, mentre il relativo processo padre invece si occuperà di inserire i processi figli nella shared memory.

Questa memoria condivisa è stata creata per raccogliere tutti i “pid” dei processi figli man mano che vengono creati per gestire i vari client; nel primo slot di tale memoria viene mantenuto il contatore dei client attualmente connessi all’applicativo, e come viene utilizzato questo dato nell’ambito della chiusura delle connessioni lato server verrà approfondito nel paragrafo 4.1.

Infine ogni processo ha una propria gestione dei segnali, che permettono di dirigere e gestire correttamente i flussi di ogni processo e di ogni client.



*Figura 5: Flowchart client – server concorrenziali*

### 3.3 Affidabilità UDP

Per riuscire ad implementare un meccanismo che rendesse affidabile UDP abbiamo opportunamente modificato il pacchetto che viene inoltrato sulla socket affinchè lo si possa inserire all’interno della finestra di spedizione, rinviare in caso di perdita (a causa di problemi di rete o di timeout) e gestire presso il destinatario per ricostruire correttamente il file ricevuto.

#### 3.3.1 Pacchetto UDP affidabile

Il pacchetto UDP da noi usato è stato associato a livello implementativo ad una struttura contenente il numero di sequenza del pacchetto (32 byte), il contenuto del pacchetto (1024 byte) ed un campo di ACK (1 intero).

Il campo del numero di sequenza è importantissimo poiché assicura di riuscire a rinviare il pacchetto che è stato perso e quindi di rendere affidabile la trasmissione; il campo del contenuto del pacchetto è il corpo del messaggio che si vuole mandare con le informazioni che si stanno scambiando tra server e client o viceversa; il campo ACK è necessario per verificare l’avvenuta ricezione del pacchetto, e in caso contrario,

controllando questo campo, ci si accorge di essere in una situazione di errore e si attua il meccanismo di ripetizione selettiva per porvi rimedio.

### 3.3.2 Finestra di spedizione

Dal momento che abbiamo utilizzato un protocollo “Selective Repeat”, è stato indispensabile l’utilizzo di una finestra di spedizione: nello specifico abbiamo utilizzato una finestra di dimensione N, un numero finito e personalizzabile.

A livello implementativo abbiamo utilizzato dei “cicli for” per creare una finestra di spedizione che lavorasse in modo analogo sia nel mittente che nel ricevente. Tale finestra verrà incrementata ognqualvolta che tutti i pacchetti saranno ricevuti correttamente durante la prima o le successive trasmissioni.

### 3.3.3 Ripetizione selettiva

Grazie alla finestra di spedizione e al campo contenente il numero di sequenza incluso nella nostra versione modificata di pacchetto UDP, siamo riusciti ad implementare la ripetizione selettiva. Questo meccanismo ci consente di rispedire solo i singoli pacchetti persi contenuti nella finestra prima di continuare la trasmissione, evitando così il rinvio di tutti i pacchetti della finestra (anche quelli correttamente ricevuti), a differenza di altri protocolli di rinvio: così facendo si risparmia molto tempo e si evitano spiacevoli errori, visto che i pacchetti correttamente ricevuti, durante un successivo rinvio, potrebbero incorrere in ulteriori problemi di invio e complicare notevolmente la trasmissione.

### 3.3.4 Timer

Il timer è un altro meccanismo che consente di ottenere l’affidabilità nella trasmissione. Lo scadere di un timer senza la ricezione di un messaggio di “ACK” segnala la perdita di un pacchetto e quindi si ha la necessità di rinviare tutti i pacchetti che subiscono questa problematica.

Nell’applicativo il timer può essere di due tipologie:

- Fisso: viene impostato un lasso di tempo (che resta invariato) durante il quale all’applicativo è concesso di restare in attesa del riscontro di avvenuta ricezione dell’invio del pacchetto;
- Adattivo: in tal caso il valore del timer viene ricalcolato ogni volta in base ai ritardi causati dalla rete. Per il ricalcolo del timer adattivo sono utilizzate le formule tipiche del protocollo TCP, mostrate anche nel seguente snippet:

```

75  void get_adaptative_timeout(double iniziale, double finale) {
76      double old_estimated = estimatedrtt;
77      double sample_rtt = (finale - iniziale);
78      estimatedrtt = ((1 - alfa) * old_estimated) + (alfa * (sample_rtt));
79      devrtt = ((1 - beta) * (devrtt)) + (beta * (fabs(((sample_rtt) - estimatedrtt))));
80      send_file_timeout = estimatedrtt + (4 * (devrtt));
81      printf("VALORI DEL TIMEOUT: %f, devrtt: %f, estimatedrtt: %f, sample_rtt: %f.\n", send_file_timeout, devrtt,
82             estimatedrtt, sample_rtt);
83  }

```

*Figura 6: Calcolo timeout additivo*

## 4. Implementazione

In questa sezione vengono approfondite le scelte implementative operate per realizzare il software e si entrerà più nel dettaglio circa il funzionamento dei servizi di list, get (download) e put (upload) offerti dall'applicativo e riguardo le modalità con cui esso apre e chiude la connessione lato server e lato client.

### 4.1 Apertura e chiusura della connessione

In questo paragrafo viene illustrato come è stata gestita la logica implementativa riguardante l'apertura e la chiusura delle varie connessioni. Le varie procedure, cui faremo riferimento, sono state rese possibili, oltre dalle scelte implementative descritte, da un'accurata logica di gestione di segnali che i processi associati a server e client catturano e gestiscono.

#### 4.1.1 Server

- Apertura della connessione: il server apre la connessione creando la socket associata alla porta scelta di default per l'inizio delle "conversazioni" dell'applicativo. Nel nostro caso è la porta 8080.

Tale socket viene creata tramite la chiamata dell'apposita funzione "create\_socket" e restituisce il file descriptor associato alla socket creata sulla porta.

Dopo l'apertura della connessione, il server si pone in attesa del pacchetto che un nuovo client invierà nel momento in cui si collegherà per la prima volta al nostro applicativo. Quando il server riceverà questo pacchetto di "benvenuto" da parte del client, aggiornerà il contatore del numero di client connessi e riserverà una porta per il nuovo client; tale porta sarà inviata tramite messaggio al client, il quale si occuperà di chiudere la connessione precedentemente aperta col server e di aprirne una nuova sulla porta che gli è stata appena comunicata. La stessa procedura verrà eseguita lato server, tramite un processo figlio che chiuderà la socket precedentemente aperta e ne aprirà una nuova sul canale associato al nuovo client. Così facendo server e client si riservano un loro "spazio" (connessione) per interagire tramite gli scambi di messaggi.

Tutto ciò ha un limite, nel senso che accadrà solo se il server avrà la possibilità di accettare nuove connessioni, in quanto abbiamo aggiunto un controllo sul numero di connessioni attualmente attive sul server, le quali devono esser minori del numero di connessioni che il server è in grado di accettare. Tale controllo ha scopo puramente dimostrativo nel caso in cui si volesse porre un limite all'aggiunta di client. Nel nostro caso si è considerato tale limite uguale a 5.

Nel caso in cui la connessione vada a buon fine, il server si metterà in attesa di richieste da parte del client; altrimenti si avrà un messaggio di notifica nel server e un messaggio di errore nel client al quale non sarà concessa la connessione.

- Chiusura della connessione: la chiusura della connessione lato server è gestita da un processo figlio, che permette al padre di rimanere in attesa dei messaggi di "benvenuto" dei client affinchè possa concedergli (o meno) la connessione.

Il processo figlio è in ascolto, tramite la funzione “`child_exit`”, di ciò che avviene nello “standard input” del server ed opera solo nel caso in cui venga inserito il comando “`close`”.

La funzione “`child_exit`” si occupa di recuperare i “pid” dei processi figli salvati sulla “shared memory” ogniqualvolta che un client si connette al nostro applicativo. I processi figli si occupano di mettere in comunicazione il server con i vari client connessi.

Quando si decide di spegnere il server, allora anche tutte le connessioni con i client (lato server) devono esser chiuse; a tale scopo si esegue un “ciclo for” tramite il quale si chiudono le connessioni e si “uccidono” i processi figli, associati ai client, presenti nella “shared memory”. Il primo elemento di quest’ultima è dedicato al numero di client attualmente connessi, mentre negli altri slot vengono inseriti i “pid” associati agli stessi.

Una volta chiuse tutte le connessioni verso i client, può essere chiusa anche quella associata alla porta di default del server; questo avvenimento viene notificato con un messaggio sul server.

Quando viene chiusa una connessione sul server, i client non avranno una notifica immediata di tale avvenimento, bensì si accorgeranno della chiusura del server solo nel momento in cui andranno ad eseguire una nuova richiesta, momento in cui arriverà anche al client un messaggio che notificherà del fatto che il server non è più in funzione.

Arrivati a questo punto, l’applicativo ha terminato la sua esecuzione, le risorse sono state rilasciate e le porte chiuse.

#### 4.1.2 Client

- Apertura della connessione: l’apertura della connessione avviene quando viene eseguito il client, ovvero quando il primo o l’n-esimo client prova a connettersi al nostro applicativo, inviando, mediante una socket inizialmente aperta sulla porta di default, un messaggio di “benvenuto” al server; attraverso quest’ultimo possiamo effettuare i vari controlli sull’effettiva possibilità di accordare o meno la connessione ai vari client precedentemente menzionati.  
Nel caso in cui i controlli andassero a buon fine e la connessione fosse concessa, si troverà stampato a schermo il menu del client e si avrà la possibilità di iniziare ad usare il software; altrimenti verrà restituito un messaggio di errore.
- Chiusura della connessione: il comando da inserire per chiudere la connessione di un client con il server è “`exit`”. In questa circostanza, prima di chiudere effettivamente la connessione lato client verso il server, si invia un messaggio a quest’ultimo per notificare l’imminente disconnessione di uno dei suoi client; successivamente ci si occuperà di chiudere effettivamente il client in questione. La procedura termina con un messaggio sul client che comunica l’effettiva chiusura dello stesso e con un messaggio sul server che informa della chiusura del canale di connessione e la terminazione della trasmissione dei pacchetti con quello specifico client.

## 4.2 List

Il comando list implementato nel nostro applicativo restituisce, una volta che il client si è connesso al server, la lista dei file presenti sul server.

Tale lista è un documento localizzato sia nel client nel server; nel caso del client, contiene tutti i nomi dei file di cui è possibile fare l’”upload”; nel caso del server invece contiene tutti i nomi dei file di cui è possibile fare il “download”.

Nel momento in cui viene eseguito un upload, viene fatta una scrittura di tale file nel server con la contestuale aggiunta del relativo nome nella lista appartenente al server; quando invece viene eseguito un download, la scrittura avviene nel client con la contestuale aggiunta del relativo nome nella lista, questa volta appartenente al client.

Per quanto riguarda l’aggiornamento del documento riguardante la lista dei file, verrà approfondito nei prossimi paragrafi.

Nell’applicativo in questione, quando un client si collega al server, si mette in “audit” dello “standard input”, in attesa dell’inserimento di un qualsiasi comando. In questo paragrafo verrà trattato il caso in cui il carattere inserito sia proprio “list”; tale comando viene inviato al server tramite messaggio. Difatti, il server, una volta che è stato fatto partire, è in “audit” del client; in questo caso è in ascolto del primo messaggio inviato dal client, che rappresenta la richiesta che lo stesso client vuole venga soddisfata dal server.

Il server, nel caso in questione, riceve il messaggio del client contenente il comando “list”; a questo punto, mediante la funzione “serverList” (che a sua volta contiene la funzione “get\_list”) apre, in sola lettura, il documento dove sono presenti i nomi dei file, e tramite una “read”, ne legge il contenuto e lo salva in un “buffer”. Tale buffer viene inviato al client. Una volta che è stato ricevuto il messaggio, la lista contenente i documenti presenti sul server viene stampata a schermo, in modo che l’utente possa essere sempre al corrente dei file che sono disponibili o meno, per l’eventuale download degli stessi.

### 4.2.1 Prestazioni

In questa sezione verranno mostrate le prestazioni nel nostro applicativo nel caso in cui venga erogato il servizio di list (prestazioni ottenute dalla media matematica effettuata su un campione di 10 tentativi per ogni caso esaminato).

Dimensione finestra	Probabilità di perdita	Timer	Tempo
3	20%	Adattativo	0.000018
3	20%	0.1s	0.000020
3	75%	Adattativo	0.000021
3	75%	0.1s	0.000025
85	20%	Adattativo	0.000014
85	20%	0.1s	0.000016
85	75%	Adattativo	0.000019
85	75%	0.1s	0.000022

## 4.3 Get (download)

Il comando “download” implementato nel nostro applicativo, una volta che il client si è connesso al server, permette di scaricare il file che verrà indicato dall’utente, tra quelli resi disponibili per il download all’interno del server. Il comando in questione deve essere inserito nel client quando esso è in “audit” dallo standard input. Una volta fatto questo, il client invierà un messaggio al server che racchiude il servizio del quale vuole usufruire, quindi in questo caso contenente una richiesta di tipo GET. Il server, essendo in ascolto, alla ricezione di questo messaggio avvierà l’apposita procedura volta allo scopo di inviare il file al client richiedente. Per prima cosa, al client viene mostrato un elenco dei file disponibili nel server, e questo viene fatto utilizzando le funzioni implementate per il comando “list”; in questo modo il client potrà scegliere ed inserire correttamente il nome del file che ha intenzione di scaricare. Avendo ricevuto la lista dei file e avendo poi inviato al server il nome del file che vuole scaricare, il client è pronto per ricevere il documento. Prima dell’effettivo invio dei pacchetti da parte del server al client, questi si scambiano un ulteriore messaggio di utility; infatti, quando il server riceve il nome del file che dovrà inviare al client, invierà un altro messaggio contenente, questa volta, la lunghezza del file (attraverso le chiamate di sistema “open” e “lseek”), in modo tale che il destinatario possa prepararsi al meglio per la ricezione del file, allocando le strutture necessarie. La procedura preliminare per il comando di download terminerà nel momento in cui il server, una volta suddiviso il file in pacchetti e posto ciascuno di essi nelle apposite strutture di supporto, inizierà ad inviare in modo affidabile i pacchetti al destinatario.

### 4.3.1 Download affidabile

Lato server, l’invio affidabile del file avviene quando il server inizia ad inviare i pacchetti contenuti nella finestra di invio di riferimento, attraverso la funzione “start\_sending\_pkct”. La procedura sarà diversa dipendentemente dal fatto che il resto della divisione tra la quantità di pacchetti da inviare e la dimensione della finestra restituisca un valore uguale o diverso da zero. La diversità di questa procedura riescede solamente nei parametri che verranno poi passati alla “send\_packet”, ovvero la funzione predisposta all’invio dei pacchetti all’interno della finestra corrente di spedizione. Una volta eseguito l’invio di tali pacchetti, questa funzione si occuperà anche di controllare che sia stato ricevuto correttamente l’ACK di tutti i pacchetti appena inoltrati. In caso contrario verranno rinviati i pacchetti che non hanno ricevuto riscontro di ricezione da parte del client. Il controllo dell’avvenuto riscontro di ricezione o meno da parte del client viene effettuato nella “check\_packet\_sended\_of\_window”, nella quale ci si accernerà che i flag di ACK dei pacchetti inviati all’interno della finestra di spedizione siano settati positivamente; in caso contrario la funzione si occuperà del rinvio dei singoli pacchetti persi.

Il campo di ACK di un determinato pacchetto viene settato mediante la seguente procedura: inizialmente, quando si invia un pacchetto al client, il campo ACK è settato negativamente, ovvero come non ricevuto. Il server si mette quindi in attesa dell’eventuale risposta di avvenuta ricezione del pacchetto da parte del client. Tale risposta viene racchiusa all’interno di un messaggio che viene inviato al server, il quale, una volta ricevuto tale messaggio, avrà la certezza che il client abbia ricevuto il pacchetto in questione e di conseguenza setterà positivamente il campo di ACK.

Nel caso in cui il client, invece, a seguito dell'invio di un pacchetto da parte del server, non trasmetta nessun messaggio di avvenuta ricezione a quest'ultimo, la procedura di settaggio del campo di ACK ad un valore positivo non avrà luogo; di conseguenza il server si ritroverà con uno o più pacchetti con campo ACK non modificato e considererà tali pacchetti come andati persi. Per questo motivo, la funzione dovrà occuparsi anche del rinvio di tali pacchetti persi. A tale scopo, è necessaria l'implementazione di un timer, associato ad ogni pacchetto; lo scadere di tale timer indicherà la necessità di dover rispedire il pacchetto a cui era associato. Nella "send\_packet", allora, oltre all'implementazione dell'invio affidabile dei pacchetti, viene effettuato anche il calcolo del tempo di spedizione per ogni pacchetto. Questo calcolo viene ovviamente effettuato solo nel caso in cui venga scelto di utilizzare un timer di tipo adattivo, piuttosto che un timer fisso.

Lato client, la gestione dei pacchetti ricevuti avviene in maniera più semplice rispetto a quella dei pacchetti inviati dal server. Il destinatario, essendosi preparato a dovere alla ricezione dei pacchetti, grazie alle informazioni contenute nei messaggi preliminari di utility che sono state descritte all'inizio del paragrafo, dovrà limitarsi a gestire i pacchetti che riceverà, inserendoli nelle apposite strutture di supporto che permettono la ricezione affidabile ed inviando un feedback di avvenuta ricezione ogni volta che riceve un pacchetto. Una volta che tutti i pacchetti del file saranno stati inviati e tutti quanti avranno ricevuto un riscontro positivo da parte del client, il file sarà arrivato correttamente a destinazione; il client allora potrà finalmente notificare al server l'avvenuta ricezione dell'intero file, che adesso sarà disponibile per essere acceduto localmente. La procedura di download può considerarsi conclusa quando il documento "file\_list", all'interno del client, viene aggiornato con il nome del file appena scaricato.

### 4.3.2 Prestazioni

In questa sezione verranno mostrate le prestazioni nel nostro applicativo nel caso in cui venga erogato il servizio di download (prestazioni ottenute dalla media matematica effettuata su un campione di 10 tentativi per ogni caso esaminato).

La dimensione della finestra è scelta di 3 ed 85, la probabilità di perdita è scelta del 20% e 75%, mentre il timer è adattativo o fisso (0.1s).

Le prestazioni inoltre riguardano il download di un file da circa 3MB.

Dimensione finestra	Probabilità di perdita	Timer	Tempo
3	20%	Adattativo	0.075810
3	20%	0.1s	0.141078
3	75%	Adattativo	0.209705
3	75%	0.1s	0.585360
85	20%	Adattativo	0.066611
85	20%	0.1s	0.126621
85	75%	Adattativo	0.199589
85	75%	0.1s	0.559410

## 4.4 Put (upload)

In questa sezione viene analizzato il comando di “upload” che permette di caricare, una volta che il client si è connesso al server, il file che viene specificato.

Tale comando viene esplicato nel menu delle operazioni disponibili nel client e viene richiesto quando il client è in ascolto (in modo tale da poterlo fornire).

Come per il servizio di download il client, una volta ricevuto il comando di “upload”, si occupa di inviare al server un messaggio affinché anch’esso si prepari per fornire il servizio adeguato. Il server mette quindi in funzione la procedura per il caricamento di un file del client. Anche in questo caso, prima che si vada avanti, la procedura inizia con la lista di file disponibili mostrata a schermo in modo tale che l’utente possa scegliere correttamente il file da caricare.

Inoltre, di nuovo, affinché ci si trovi pronti per l’upload del file, vengono allocate le risorse e le strutture richieste per il caricamento dello stesso.

Il file dunque viene spacchettato secondo le specifiche dell’applicativo e viene inviato; al termine dell’invio, il documento contenente la lista di files del server verrà aggiornato con il nome del file appena caricato da parte del client.

### 4.4.1 Upload affidabile

La logica di implementazione dell’upload è speculare a quella del download. Anche per l’affidabilità viene sfruttata la stessa logica della finestra di invio, degli ACK, del timer e del “pacchetto UDP affidabile”.

Per una più semplice comprensione, compattezza, semplicità ed immediatezza del codice sono state riutilizzate proprio le stesse funzioni, con le opportune modifiche, affinché gli interpreti di questo scambio di messaggi, sempre in modalità affidabile, fossero invertiti.

### 4.4.2 Prestazioni

In questa sezione verranno mostrate le prestazioni nel nostro applicativo nel caso in cui venga erogato il servizio di upload (prestazioni ottenute dalla media matematica effettuata su un campione di 10 tentativi per ogni caso esaminato).

La dimensione della finestra è scelta di 3 ed 85, la probabilità di perdita è scelta del 20% e 75%, mentre il timer è adattativo o fisso (0.1s).

Le prestazioni inoltre riguardano l’upload di un file da circa 3MB.

Dimensione finestra	Probabilità di perdita	Timer	Tempo
3	20%	Adattativo	0.068107
3	20%	0.1s	0.140957
3	75%	Adattativo	0.203175
3	75%	0.1s	0.809115
85	20%	Adattativo	0.063034
85	20%	0.1s	0.133554
85	75%	Adattativo	0.188764
85	75%	0.1s	0.758582

## 5. Manuale per l'installazione, configurazione ed utilizzo

### 5.1 Eseguire il software

È necessario eseguire:

- Server → per eseguire il server è necessario navigare all'interno della cartella UDP\_RELIABLE/Server e poi digitare make -f Makefile
- Client → per eseguire il client è necessario navigare all'interno della cartella UDP\_RELIABLE/Client e poi digitare make -f Makefile

### 5.2 Configurazione dell'applicativo

Nel programma si possono modificare i seguenti parametri:

- Probabilità di perdita: "LOSS\_PROBABILITY" è il parametro che ci permette di perdere i pacchetti con la probabilità richiesta;
- Timeout adattivo: "ADAPTATIVE\_TIMEOUT" è il parametro che ci permette:
  - o Di utilizzare un timeout di tipo adattivo se il parametro è settato a 1;
  - o Di utilizzare un timeout di tipo fisso se il parametro è settato a 0;
- Tempo di timeout per la ricezione di un pacchetto: "RECV FILE TIMEOUT" è il parametro che ci permette di settare il tempo di attesa che la socket deve attendere prima di considerare un pacchetto inviato e a cui non è stato corrisposto nessun ACK come perso;
- Tempo di timeout per l'invio di un pacchetto: "SEND FILE TIMEOUT" è il parametro che ci permette di settare il tempo di attesa che la socket pazienta prima di considerare un pacchetto inviato e a cui non è stato corrisposto nessun ACK come perso;
- Numero di porta: "PORT" è il parametro che ci permette di modificare il numero di porta su cui avviene la prima comunicazione client – server;
- Dimensione della finestra di spedizione: "WINDOW\_SIZE" è il parametro che ci permette di modificare la dimensione della finestra di spedizione mediante la quale avviene l'invio dei pacchetti.

### 5.3 Utilizzo

Nel programma si possono eseguire i seguenti comandi:

- All'interno del client:
  - o list → inserendo il comando "list" e premendo invio verrà mostrata la lista dei file presenti nel server;
  - o upload → inserendo il comando "upload" e premendo invio verrà mostrata la lista dei file presenti nel client che possiamo caricare sul server; successivamente, inserendo il nome del file di cui ci interessa fare l'upload e premendo invio, partirà il caricamento dello stesso;
  - o download → inserendo il comando "download" e premendo invio verrà mostrata la lista dei file presenti nel server che possiamo scaricare; successivamente, inserendo il nome del file di cui ci interessa fare il download e premendo invio, verrà avviato lo scaricamento dello stesso;

- exit → inserendo il comando “exit” e premendo invio verranno iniziate le procedure per la chiusura della connessione del client verso il server;
- All’interno del server:
  - close → inserendo il comando “close” e premendo invio verranno iniziate le procedure per la chiusura della connessione del server verso tutti i client.

## 6. Esempi di utilizzo

Questa ultima sezione è dedicata alla descrizione del funzionamento dell’applicativo tramite delle istantanee nei vari casi d’uso.

### 6.1 Connessione al server

In questo paragrafo viene mostrato come il server apre la connessione, notifica un eventuale numero eccessivo di client connessi (ai quali quindi non è possibile concedere la connessione), gestisce le nuove connessioni, le disconnessioni dei client e la chiusura della connessione stessa del server, anche se in presenza di client connessi, ai quali verrà notificata immediatamente la chiusura del server e con cui verrà successivamente chiusa la connessione.

#### 6.1.1 Connessione riuscita del server

Il server si avvia e concede la possibilità ai nuovi client di connettersi all’applicativo.

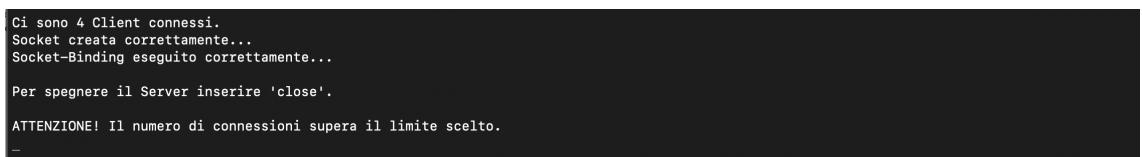


```
Last login: Sun Jun 16 21:04:34 on console
[MBP-di-Mauro:~ mauroficorella$ cd ClionProjects/UDP_RELIABLE/
[MBP-di-Mauro:UDP_RELIABLE mauroficorella$ cd Server
[MBP-di-Mauro:Server mauroficorella$ make
gcc -Wall -Wextra -o serverUDP serverUDP.o ../server_functions.o ..../common_functions.o
./serverUDP
Socket creata correttamente...
Socket-Binding eseguito correttamente...
Per spegnere il Server inserire 'close'.
```

*Figura 7: Connessione del server*

#### 6.1.2 Eccesso di connessioni del client

Il server riceve una richiesta di connessione da parte di un client che vorrebbe collegarsi all’applicativo, ma il numero di connessioni consentite così facendo verrebbe superato, quindi blocca questa azione e notifica ciò con un messaggio a schermo.

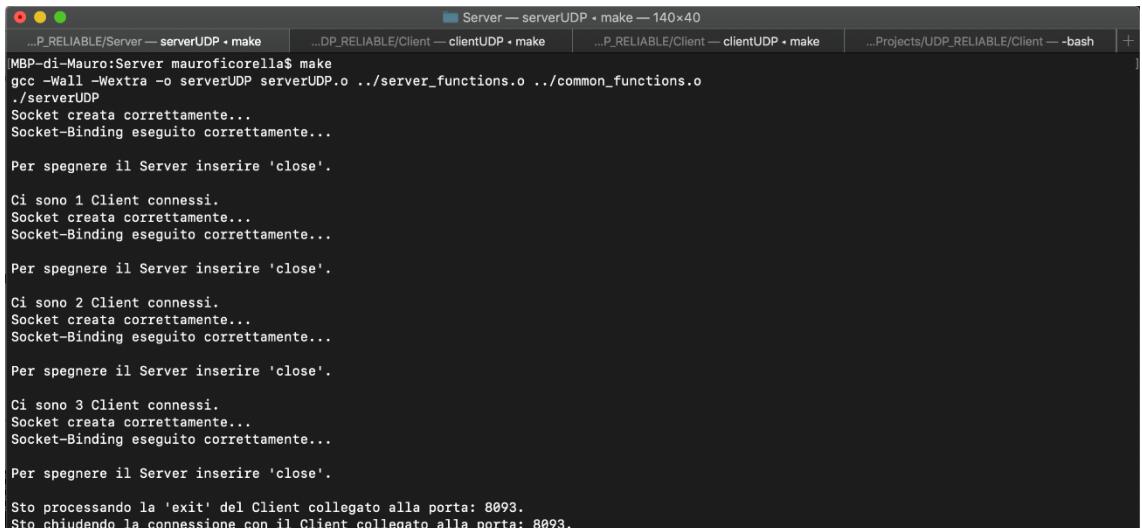


```
Ci sono 4 Client connessi.
Socket creata correttamente...
Socket-Binding eseguito correttamente...
Per spegnere il Server inserire 'close'.
ATTENZIONE! Il numero di connessioni supera il limite scelto.
```

*Figura 8: Connessione del client negata dal server*

### 6.1.3 Gestione connessioni del server

Il server notifica la connessione o la disconnessione di client all'interno del software.



```
Server — serverUDP • make — 140x40
...P_RELIABLE/Server — serverUDP • make      ...P_RELIABLE/Client — clientUDP • make      ...P_RELIABLE/Client — clientUDP • make      ...Projects/UDP_RELIABLE/Client — -bash +1
[MBP-di-Mauro:Server mauroficorella$ make
gcc -Wall -Wextra -o serverUDP serverUDP.o ../server_functions.o ../common_functions.o
./serverUDP
Socket creata correttamente...
Socket-Binding eseguito correttamente...

Per spegnere il Server inserire 'close'.

Ci sono 1 Client connessi.
Socket creata correttamente...
Socket-Binding eseguito correttamente...

Per spegnere il Server inserire 'close'.

Ci sono 2 Client connessi.
Socket creata correttamente...
Socket-Binding eseguito correttamente...

Per spegnere il Server inserire 'close'.

Ci sono 3 Client connessi.
Socket creata correttamente...
Socket-Binding eseguito correttamente...

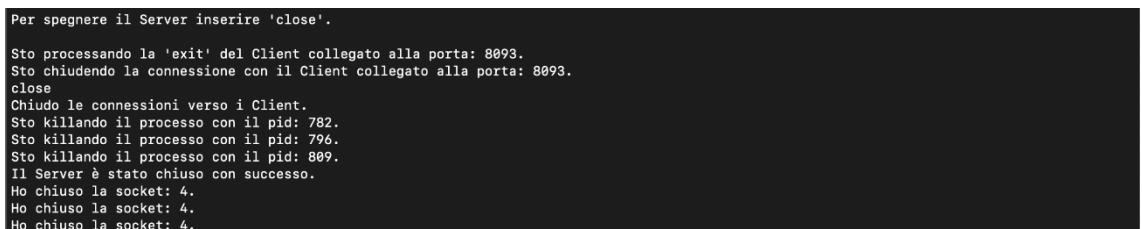
Per spegnere il Server inserire 'close'.

Sto processando la 'exit' del Client collegato alla porta: 8093.
Sto chiudendo la connessione con il Client collegato alla porta: 8093.
```

Figura 9: Gestione delle connessioni del server

### 6.1.4 Chiusura della connessione del server

Spegnimento del server tramite l'inserimento del comando “close” (anche con client eventualmente connessi).



```
Per spegnere il Server inserire 'close'.

Sto processando la 'exit' del Client collegato alla porta: 8093.
Sto chiudendo la connessione con il Client collegato alla porta: 8093.
close
Chiudo le connessioni verso i Client.
Sto killando il processo con il pid: 782.
Sto killando il processo con il pid: 796.
Sto killando il processo con il pid: 809.
Il Server è stato chiuso con successo.
Ho chiuso la socket: 4.
Ho chiuso la socket: 4.
Ho chiuso la socket: 4.
```

Figura 10: Chiusura delle connessioni del server

## 6.2 Connessione del client

In questo paragrafo viene mostrato come il client si collega al server, come il server notifica al client che non può collegarsi all'applicativo e come il client chiude la connessione verso il server.

### 6.2.1 Connessione riuscita del client

Il client richiede (e ottiene) la possibilità di collegarsi al server.

```
MBP-di-Mauro:Client mauroficorella$ make
gcc -Wall -Wextra -o clientUDP clientUDP.o ../client_functions.o ../common_functions.o
./clientUDP
Socket creata correttamente...
Inserisci un comando tra:
1) exit
2) list
3) download
4) upload
```

*Figura 11: Connessione del client*

### 6.2.2 Connessione non riuscita del client

Quando il client non riesce a collegarsi al server le cause possono essere molteplici: la connessione non viene concessa poiché il server ha smesso di funzionare o poiché il server ha raggiunto il limite massimo di connessioni.

```
list
Il tempo impiegato è: 0.000011
ATTENZIONE! Il Server non è più in funzione.
```

*Figura 12: Connessione non riuscita del client*

```
ATTENZIONE! Impossibile collegarsi al Server, limite di connessioni superato.
make: *** [client] Error 1
MBP-di-Mauro:Client mauroficorella$
```

*Figura 13: Connessione non riuscita del client*

### 6.2.3 Chiusura connessione del client

Disconnessione del client dall'applicativo.

```
MBP-di-Mauro:Client mauroficorella$ make
gcc -Wall -Wextra -o clientUDP clientUDP.o ../client_functions.o ../common_functions.o
./clientUDP
Socket creata correttamente...
Inserisci un comando tra:
1) exit
2) list
3) download
4) upload
Socket creata correttamente...
exit
Il Client sta chiudendo la connessione...
Client disconnesso.
MBP-di-Mauro:Client mauroficorella$
```

*Figura 14: Chiusura connessione del client*

## 6.3 Controllo dell'input

In questo paragrafo viene mostrato come il client ed il server possiedano una logica di controllo dell'input per i comandi che vengono inseriti volta per volta.

### 6.3.1 Lato server

Logica di controllo dell'input lato server.

```
MBP-di-Mauro:Server mauroficorella$ make
gcc -Wall -Wextra -o serverUDP serverUDP.o ../server_functions.o ../common_functions.o
./serverUDP
Socket creata correttamente...
Socket-Binding eseguito correttamente...

Per spegnere il Server inserire 'close'.

--
```

*Figura 15: Gestione input lato server*

### 6.3.2 Lato client

Logica di controllo dell'input lato client.

```
MBP-di-Mauro:Client mauroficorella$ make
gcc -Wall -Wextra -o clientUDP clientUDP.o ../client_functions.o ../common_functions.o
./clientUDP
Socket creata correttamente...
Inserisci un comando tra:
1) exit
2) list
3) download
4) upload
Socket creata correttamente...
--
```

*Figura 16: Gestione input lato client*

## 6.4 Servizio di listing

In questo paragrafo viene mostrato come il client richiede ed ottiene la lista dei file presenti sul server.

### 6.4.1 Lato client

```
Socket creata correttamente...
Inserisci un comando tra:
1) exit
2) list
3) download
4) upload
Socket creata correttamente...
list
Il tempo impiegato è: 0.000010
Lista dei file disponibili nel Server:
5maggio.txt
hallelujah.txt
prova.txt
```

*Figura 17: List del client*

## 6.5 Servizio di download di un file

In questo paragrafo viene mostrato come il client richiede ed ottiene il servizio di download di un file presente sul server (la stampa della lista dei file presenti nel server aiuta nella scelta del file da scaricare). Qualora non venisse introdotto un nome corretto, la richiesta di inserire il nome di un file da scaricare verrà riproposta all'utente finché non verrà inserito un nome corretto.

### 6.5.1 Inizio del download lato client

In questa sezione viene mostrato come, lato client, avviene la richiesta del servizio di download e l'inizio della ricezione dei pacchetti in modalità affidabile.

```
Inserisci il nome del file da scaricare...
prova.txt
Numero pacchetti da ricevere: 9.
Sto creando il file prova.txt...
Ho creato il file.
Inizio a ricevere i pacchetti in modalità affidabile...
sono nel while
offset : 9           count :0           packet_count : 9           w_size: 85
counter: 0
sono nel ciclo con offset
sono nel for
offset : 9           count :0           packet_count : 9           w_size: 9
counter: 0
Sto inviando l'ACK: 0.
          Pacchetto ricevuto numero di seq: 0.
sono nel for
offset : 9           count :1           packet_count : 9           w_size: 9
counter: 1
Sto inviando l'ACK: 1.
          Pacchetto ricevuto numero di seq: 1.
sono nel for
offset : 9           count :2           packet_count : 9           w_size: 9
counter: 2
Sto inviando l'ACK: 2.
          Pacchetto ricevuto numero di seq: 2.
sono nel for
offset : 9           count :3           packet_count : 9           w_size: 9
counter: 3
Sto inviando l'ACK: 3.
```

*Figura 18: Inizio del download lato client*

### 6.5.2 Inizio del download lato server

In questa sezione viene mostrato come, lato server, avviene la richiesta e l'inizio dell'invio dei pacchetti in modalità affidabile.

In questo specifico caso, la trasmissione è stata eseguita utilizzando il timeout adattativo. In figura si possono vedere i valori calcolati tramite la formula mostrata in precedenza.

```
Sto processando la 'download' del Client collegato alla porta: 8091.  
Il file prova.txt è stato richiesto.  
PACKET COUNT :6 packet_count = 6 seq= 0 offset= 6 .  
packet_count= 6 seq= 0 offset= 6 .  
Probabilità di perdita: 20 percento.  
time: 0.100000  
Sto inviando il pacchetto 0.  
sto per ricevere  
VALORI DEL TIMEOUT: 0.000123, devrtt: 0.000027, estimatedrtt: 0.000015, sample_rtt: 0.000123.  
Ho ricevuto l'ack del pacchetto 0.  
time: 0.005000
```

*Figura 19: Inizio del download lato server*

### 6.5.3 Fine del download lato client

In questa sezione viene mostrato come, lato client, avviene la fine del servizio di download richiesto dal client stesso.

```
Sto inviando l'ACK: 8.  
Pacchetto ricevuto numero di seq: 8.  
Il tempo impiegato è: 0.000650  
Il tempo impiegato è: 0.000699  
Ricezione terminata correttamente.  
Scrivo il file...  
File scritto correttamente.  
Aggiorno il file list...  
File aggiornato correttamente  
Operazione di download completata con successo.  
—
```

*Figura 20: Fine del download lato client*

### 6.5.4 Fine del download lato server

In questa sezione viene mostrato come, lato server, avviene la fine del servizio di download richiesto dal client.

```
Sto inviando il pacchetto 8.  
sto per ricevere  
VALORI DEL TIMEOUT: 0.000191, devrtt: 0.000031, estimatedrtt: 0.000066, sample_rtt: 0.000105.  
Ho ricevuto l'ack del pacchetto 8.  
Sto entrando nella check per l'eventuale ritrasmissione di pacchetti persi.  
File inviato correttamente.  
—
```

*Figura 21: Fine del download lato server*

### 6.5.5 Ricezione pacchetti lato client

In questa sezione viene mostrato come, lato client, avviene la ricezione dei pacchetti del file richiesto mediante il servizio di download demandato dal client stesso.

```
offset : 9           count :6           packet_count : 9           w_size: 9
counter: 6
Sto inviando l'ACK: 6.
          Pacchetto ricevuto numero di seq: 6.
sono nel for
offset : 9           count :7           packet_count : 9           w_size: 9
counter: 7
Sto inviando l'ACK: 7.
          Pacchetto ricevuto numero di seq: 7.
sono nel for
offset : 9           count :8           packet_count : 9           w_size: 9
counter: 8
Sto inviando l'ACK: 8.
          Pacchetto ricevuto numero di seq: 8.
```

*Figura 22: Ricezione pacchetti lato client*

### 6.5.6 Invio dei pacchetti lato server

In questa sezione viene mostrato come, lato server, avviene l'invio dei pacchetti del file richiesto mediante il servizio di download demandato dal client.

```
Sto inviando il pacchetto 2724.
sto per ricevere
il pacchetto è andato perso, ack: 2724 non ricevuto
VALORI DEL TIMEOUT: 0.001076, devrtt: 0.000219, estimatedrtt: 0.000199, sample_rtt: 0.000580.
time: 0.005000
Sto inviando il pacchetto 2725.
sto per ricevere
VALORI DEL TIMEOUT: 0.000984, devrtt: 0.000202, estimatedrtt: 0.000178, sample_rtt: 0.000029.
Ho ricevuto l'ack del pacchetto 2725.
time: 0.005000
Sto inviando il pacchetto 2726.
sto per ricevere
VALORI DEL TIMEOUT: 0.000895, devrtt: 0.000184, estimatedrtt: 0.000159, sample_rtt: 0.000028.
Ho ricevuto l'ack del pacchetto 2726.
time: 0.005000
```

*Figura 23: Invio dei pacchetti lato server*

## 6.6 Servizio di upload di un file

In questo paragrafo viene mostrato come il client richiede ed ottiene il servizio di upload di un file presente nel client stesso (la stampa della lista dei file del client aiuta nella scelta del file da caricare).

Qualora non venisse introdotto un nome corretto, la richiesta di inserire il nome di un file da caricare verrà riproposta all'utente finché non viene inserito un nome corretto.

### 6.6.1 Inizio dell'upload lato client

In questa sezione viene mostrato come, lato client, avviene la richiesta del servizio di upload e l'inizio dell'invio dei pacchetti in modalità affidabile. In questo caso specifico la trasmissione è stata eseguita usando il timeout adattativo. In figura sono mostrati i valori calcolati con la formula mostrata in precedenza.

```
Inserisci il nome del file da caricare...
prova.txt
Sto aprendo il file prova.txt...
Numero di pacchetti da caricare: 10.
Inizio il caricamento del file...
packet_count= 10           seq= 0           offset= 10      .
packet_count= 10           seq= 0           offset= 10      .
Probabilità di perdita: 20 percento.
time: 0.005000
Sto inviando il pacchetto 0.
VALORI DEL TIMEOUT: 0.000350, devrtt: 0.000059, estimatedrtt: 0.000114, sample_rtt: 0.000245.
Ho ricevuto l'ack del pacchetto 0.
time: 0.005000
Sto inviando il pacchetto 1.
VALORI DEL TIMEOUT: 0.000301, devrtt: 0.000046, estimatedrtt: 0.000116, sample_rtt: 0.000124.
Ho ricevuto l'ack del pacchetto 1.
time: 0.005000
```

Figura 24: Inizio dell'upload lato client

### 6.6.2 Inizio dell'upload lato server

In questa sezione viene mostrato come, lato server, avviene l'inizio della ricezione del file che il client sta scaricando.

```
Sto processando l' 'upload' del Client collegato alla porta: 8091.
Sto ricevendo il file prova.txt...
Lunghezza file: 9216.
Numero pacchetti da inviare: 10.
Sto creando il file prova.txt...
Ho creato il file.
Inizio a ricevere i pacchetti in modalità affidabile...
sono nel while
offset : 10           count :0           packet_count : 10           w_size: 85
counter: 0
sono nel ciclo con offset
sono nel for
offset : 10           count :0           packet_count : 10           w_size: 10
counter: 0
Sto inviando l'ACK: 0.
          Pacchetto ricevuto numero di seq: 0.
sono nel for
offset : 10           count :1           packet_count : 10           w_size: 10
counter: 1
Sto inviando l'ACK: 1.
          Pacchetto ricevuto numero di seq: 1.
```

Figura 25: Inizio dell'upload lato server

### 6.6.3 Fine dell'upload lato client

In questa sezione viene mostrato come, lato client, avviene la fine del servizio di upload richiesto dal client.

```
Sto inviando il pacchetto 9.  
VALORI DEL TIMEOUT: 0.000204, devrtt: 0.000027, estimatedrtt: 0.000097, sample_rtt: 0.000069.  
Ho ricevuto l'ack del pacchetto 9.  
Sto entrando nella check per l'eventuale ritrasmissione di pacchetti persi seq = 10.  
Il tempo impiegato è: 0.000600  
File caricato correttamente, operazione di caricamento completata con successo.
```

*Figura 26: Fine dell'upload lato client*

### 6.6.4 Fine dell'upload lato server

In questa sezione viene mostrato come, lato server, avviene la fine del servizio di upload richiesto dal client.

```
Sto inviando l'ACK: 9.  
Pacchetto ricevuto numero di seq: 9.  
Il tempo impiegato è: 0.000602  
Ricezione terminata correttamente.  
Scrivo il file...  
File scritto correttamente.  
Aggiorno file_list...  
File aggiornato correttamente.  
Operazione di upload completata con successo.
```

*Figura 27: Fine dell'upload lato server*

### 6.6.5 Invio dei pacchetti lato client

In questa sezione viene mostrato come, lato client, avviene l'invio dei pacchetti del file richiesto mediante il servizio di upload demandato dal client.

```
Sto inviando il pacchetto 4.  
VALORI DEL TIMEOUT: 0.000225, devrtt: 0.000027, estimatedrtt: 0.000115, sample_rtt: 0.000095.  
Ho ricevuto l'ack del pacchetto 4.  
time: 0.005000  
Sto inviando il pacchetto 5.  
VALORI DEL TIMEOUT: 0.000206, devrtt: 0.000023, estimatedrtt: 0.000114, sample_rtt: 0.000104.  
Ho ricevuto l'ack del pacchetto 5.  
time: 0.005000  
Sto inviando il pacchetto 6.  
VALORI DEL TIMEOUT: 0.000204, devrtt: 0.000023, estimatedrtt: 0.000110, sample_rtt: 0.000086.  
Ho ricevuto l'ack del pacchetto 6.
```

*Figura 28: Invio dei pacchetti lato client*