

# Ingegneria di Internet & Web 2017/2018

## Progetto B: Trasferimento file su UDP

Marco Lanciotti  
Andrea Lombardo  
Valeria Polli



Rome, January 30, 2019.

## Contents

<b>1</b>	<b>INTRODUZIONE</b>	<b>4</b>
1.1	Processi client-server . . . . .	4
1.2	Trasferimento UDP . . . . .	5
1.3	Selective Repeat . . . . .	7
<b>2</b>	<b>ARCHITETTURA DEL SISTEMA</b>	<b>9</b>
2.1	Server . . . . .	9
2.2	Client . . . . .	10
<b>3</b>	<b>SPECIFICHE IMPLEMENTATIVE</b>	<b>11</b>
3.1	Instaurazione della connessione . . . . .	11
3.2	Gestione Concorrenza . . . . .	12
3.2.1	Specifiche Concorrenza . . . . .	12
3.3	List: Visione file disponibili . . . . .	13
3.3.1	Specifiche LIST . . . . .	13
3.4	Get/Put → Download/Upload File . . . . .	13
3.4.1	GET <FILENAME> : . . . . .	13
3.4.2	PUT <FILENAME> : . . . . .	14
3.5	Trasferimento file . . . . .	14
3.5.1	Principio Buffer Circolare: . . . . .	15
3.5.2	Scambio di messaggi . . . . .	17
3.6	Disconnessione client-server . . . . .	19
3.6.1	COMANDO <EXIT> . . . . .	19
3.6.2	SEGNALI SIGINT,SIGQUIT . . . . .	19
<b>4</b>	<b>ESEMPI DI FUNZIONAMENTO</b>	<b>20</b>
4.1	Schermata iniziale . . . . .	20
4.2	Messaggi di comando . . . . .	21
4.2.1	Comando “list” . . . . .	21
4.2.2	Comando “get” . . . . .	22
4.2.3	Comando “put” . . . . .	23
4.2.4	Comando “exit” . . . . .	24
4.2.5	Invio segnale . . . . .	24
4.3	Messaggi di risposta . . . . .	25
<b>5</b>	<b>PIATTAFORME UTILIZZATE</b>	<b>26</b>
<b>6</b>	<b>PIATTAFORME UTILIZZATE</b>	<b>27</b>
<b>7</b>	<b>VALUTAZIONI PRESTAZIONALI</b>	<b>28</b>
7.1	Test prestazionali . . . . .	28



# 1 INTRODUZIONE

Il progetto realizzato ha come obiettivo principale quello di implementare un' applicazione client-server che utilizzi UDP come servizio di trasporto.

## 1.1 Processi client-server

Il sistema realizzato è formato da due tipi di moduli: **il client e il server**, generalmente eseguiti su macchine differenti collegati alla rete. Il server si occupa di tutte quelle operazioni necessarie a realizzare un servizio mentre il client ha la facoltà di inviare richieste formulate da un utente, in seguito ad una connessione, al server. Tuttavia, in generale, ogni processo invia e riceve messaggi attraverso un'interfaccia tra il livello di applicazione e il livello di trasporto, chiamata socket; di conseguenza il livello di trasporto nell'host di ricezione non trasferisce i dati a un processo, ma piuttosto ad una socket che svolge la funzione di intermediario. Ciascun host ha la possibilità di istanziare questo canale di comunicazione, ossia la socket, la quale gli permette di comunicare con il server con cui ha intenzione di instaurare una connessione.

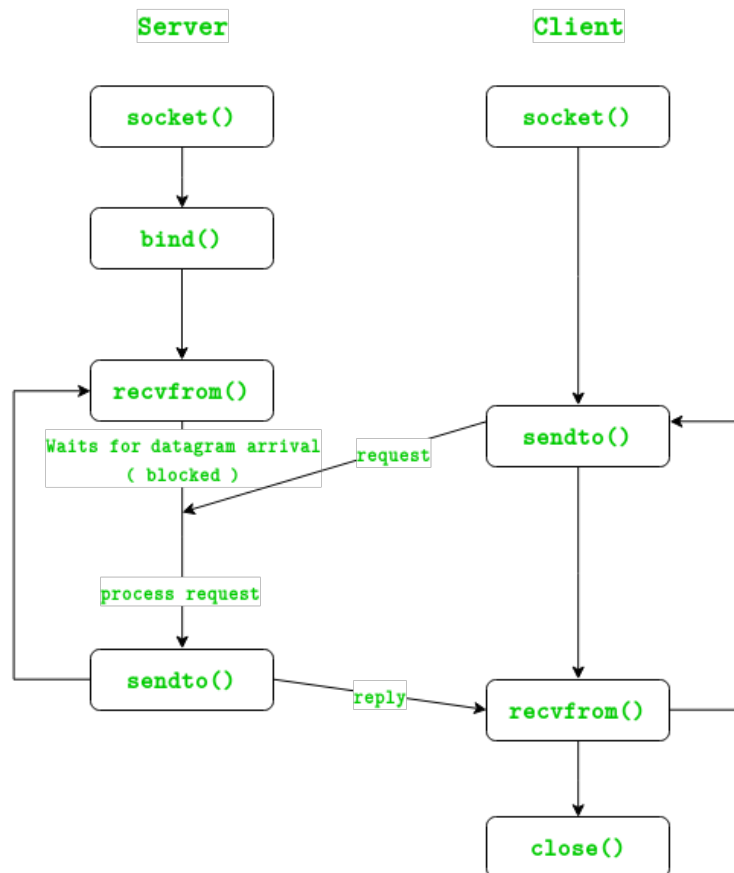


Figure 1.1: UDP Connection

Come richiesto, abbiamo fatto uso delle API (Application Programming Interface) del socket di Berkeley per quanto riguarda l'interazione con il sistema operativo.

Una socket in questo contesto ha un ciclo di vita molto semplice: apertura, lettura/scrittura, chiusura. La socket è un'interfaccia locale all'host, controllata dal sistema operativo, creata/posseduta dall'applicazione tramite la quale il processo applicativo può inviare/ricevere messaggi a/da un altro processo applicativo (remoto o locale). Il meccanismo di comunicazione tra client e server, nel contesto di questo progetto, avviene attraverso lo scambio di messaggi di comando e di messaggi di risposta.

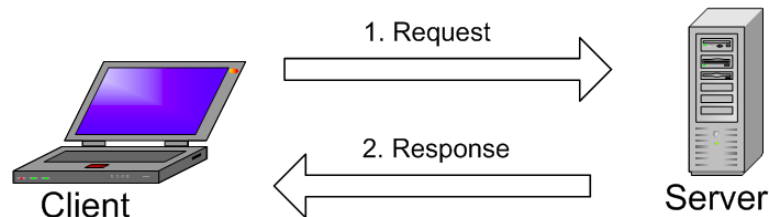


Figure 1.2: Scambio messaggi comando risposta tra Client-Server

Come detto, al fine di poter realizzare un meccanismo di comunicazione tra client e server è necessario prima instaurare una connessione. La connessione in ambito UDP, come nel nostro caso, ha bisogno di poche operazioni, in generale istanziare una socket ed effettuare una operazione di `bind()`, nel lato del server. Tuttavia per ogni client che vuole instaurare una connessione con il server ci sarà bisogno di definire una porta univoca per server e client. Dopo aver stabilito una connessione tramite una porta prestabilita, il client è in grado di effettuare richieste al server tramite l'utilizzo dei seguenti comandi:

- **list**: inviando questo messaggio al server, quest'ultimo risponde al client inviandogli un elenco di file presenti al suo interno;
- **get <filename>**: messaggio che il client invia al server per ottenere un file. Il server risponde inviandogli il file desiderato, nel caso in cui lo possieda;
- **put <filename>**: messaggio che il client invia al server per inviargli un file. Questo file verrà prelevato dalla cartella "Upload" del client, per poi essere inviato al server;
- **exit**: messaggio utilizzato per terminare la comunicazione con il server;

Per quanto riguarda invece i messaggi di risposta, si tratta di messaggi che vengono inviati al client in risposta al verificarsi di una determinata situazione, ad esempio l'invio di un comando errato.

## 1.2 Trasferimento UDP

UDP è un protocollo di trasporto in cui non esiste handshaking tra le entità di invio e di ricezione e proprio per questo motivo offre un servizio non orientato alla connessione. UDP, a differenza di TCP svolge le funzioni essenziali per un protocollo di trasporto:

- **funzione di multiplexing/demultiplexing per l'indirizzamento dei dati;**
- **una verifica degli errori mediante la checksum (inserita in un campo d'intestazione);**

A parte questi servizi UDP non aggiunge altro di rilevante riguardo al protocollo di rete IP che effettuerà un tentativo di consegna in modalità best-effort. UDP è un protocollo di tipo connectionless (senza connessione) infatti non garantisce né il riordinamento dei pacchetti né la ritrasmissione di quelli persi; di conseguenza viene definito un protocollo **NON AFFIDABILE**. A discapito di ciò è usato frequentemente per la trasmissione di contenuti multimediali. Dopo aver prelevato i messaggi dal processo applicativo, UDP aggiunge il numero di porta di origine e di destinazione per multiplexing e demultiplexing, aggiunge due ulteriori piccoli campi, per poi passare il segmento ottenuto al livello di rete. Quest'ultimo inserisce il segmento in un datagramma IP ed effettua un tentativo di consegna; in caso di successo UDP utilizza il numero di porta di destinazione per consegnare i dati al processo applicativo corretto.

Alcune applicazioni fanno uso di UDP per i seguenti motivi:

- **Controllo più fine su quali dati inviare e quando;**
- **Assenza di connessione.** Non essendoci handshake tra gli elementi coinvolti, UDP non introduce ritardo nello stabilire una connessione e non conserva lo stato di quest'ultima. Ricordiamo ad esempio che il DNS sfrutta UDP proprio per questo motivo;
- **Spazio ridotto per l'intestazione del pacchetto** (solo 8 byte);

Vediamo ora la struttura di un segmento UDP:

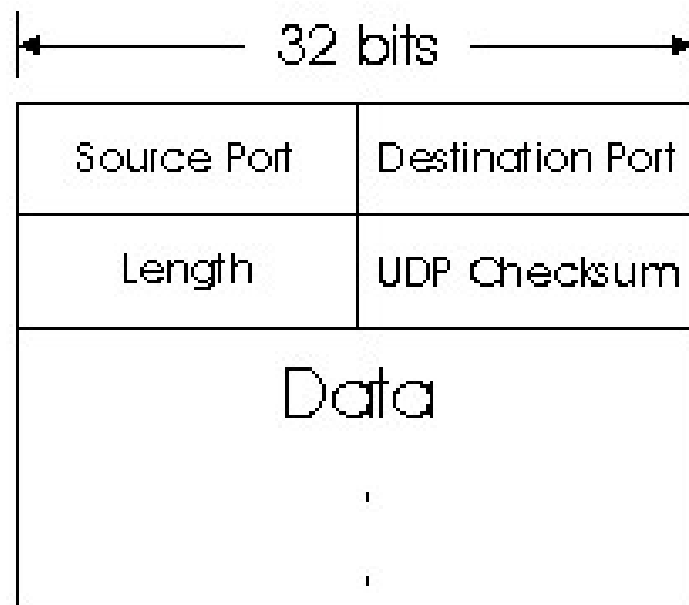


Figure 1.3: Struttura Segmento UDP

L'intestazione, o **HEADER**, di un generico UDP presenta solo quattro campi di due byte ciascuno:

1. **Source Port:** numero di porta di sorgente;
2. **Destination Port:** numero di porta di destinazione;
3. **Length:** numero di byte del segmento UDP;
4. **Checksum:** parametro utilizzato dal ricevente per verificare se i bit del segmento UDP sono stati alterati durante il trasferimento, ossia se vi sono degli errori. Tale campo viene messo a disposizione poiché non c'è garanzia che tutti i collegamenti tra origine e destinazione controllino gli errori;
5. **Data:** I restanti byte del pacchetto sono occupati dal settore Data, ossia il contenuto effettivo da inviare.

### 1.3 Selective Repeat

L'idea cardine su cui si basa questo progetto è quello di realizzare un applicativo di rete client-server in cui lo scambi di messaggi avvenga tramite un servizio di comunicazione non affidabile (UDP). Tuttavia, al fine di garantire l'affidabilità di **UDP**, è stato necessario, come richiesto, l'implementazione a livello applicativo del protocollo **SELECTIVE REPEAT**.

Nella ripetizione selettiva, il mittente ritrasmette soltanto i pacchetti per i quali non ha ricevuto un **ACK**, a differenza, ad esempio di **GBN**, in cui per un solo pacchetto perso si ritrasmettono tutti i successivi già inviati nel pipeline.

Dunque tale protocollo evita le ritrasmissioni e fa ritrasmettere al mittente solo i pacchetti su cui vi sono sospetti di errore. Questa forma di ritrasmissione a richiesta costringe il destinatario a inviare dei messaggi di acknowledgement specifici per i pacchetti ricevuti correttamente. Dal punto di vista implementativo mittente e ricevente hanno a disposizione un buffer e ad ogni pacchetto viene associato un numero di sequenza.

Di seguito riportiamo brevemente il comportamento di mittente e destinatario:

- Il mittente avrà a disposizione una finestra con  $N$  pacchetti provenienti dall'applicativo, quest'ultimi potranno essere inviati in maniera concorrenziale. Ognuno di questi pacchetti avrà associato un timer utile nella ritrasmissione del pacchetto qualora non sia stato ricevuto alcun **ACK** nell'intervallo di tempo definito per l'appunto dal timer. Ora, invece, nel caso il mittente riceva un **ACK** tale pacchetto verrà classificato come "Ricevuto", inoltre, se l'**ACK** ricevuto risulta essere il primo della finestra, quest'ultima posizionerà il suo indice sul pacchetto con il più piccolo numero di sequenza a non aver ricevuto ancora segnale di acknowledgement. Lo spostamento della finestra consente il caricamento di altri pacchetti con il loro conseguente invio.
- Il destinatario qualora riceva un pacchetto all'interno della sua finestra di ricezione, invierà un **ACK** selettivo. Ora se il pacchetto non era stato già ricevuto viene inserito nel buffer e qualora presentasse un numero di sequenza uguale alla base della finestra di ricezione, verrà consegnato al livello superiore insieme a tutti i pacchetti ricevuti con numeri di sequenza consecutivi presenti nel buffer. Così facendo si garantisce il giusto ordinamento nella consegna dei pacchetti.

## Selective repeat: finestre sender, receiver

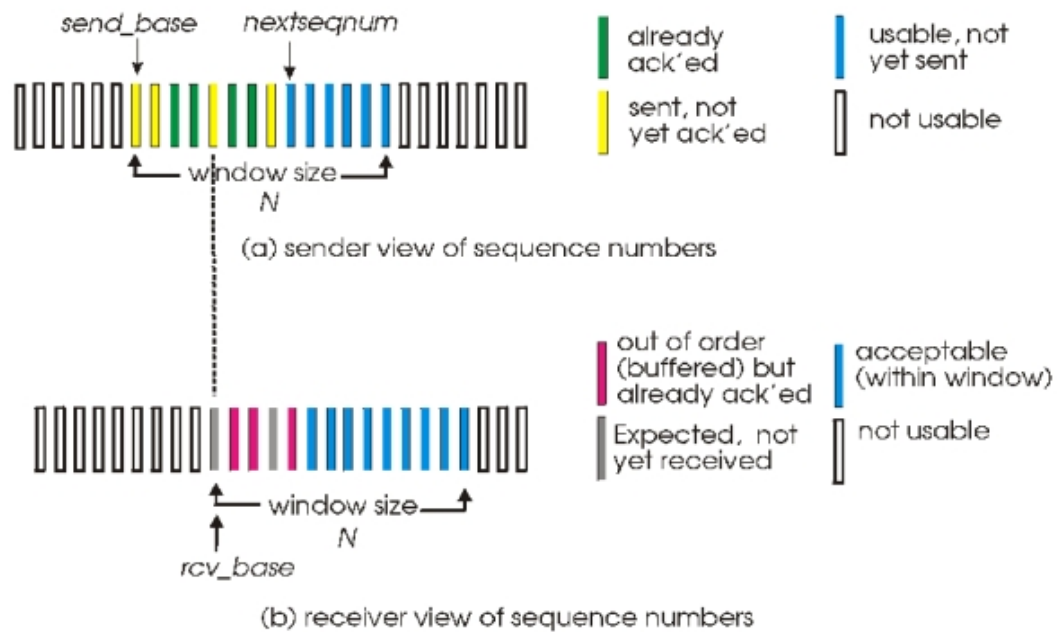


Figure 1.4: Selective Repeat Protocol



## 2 ARCHITETTURA DEL SISTEMA

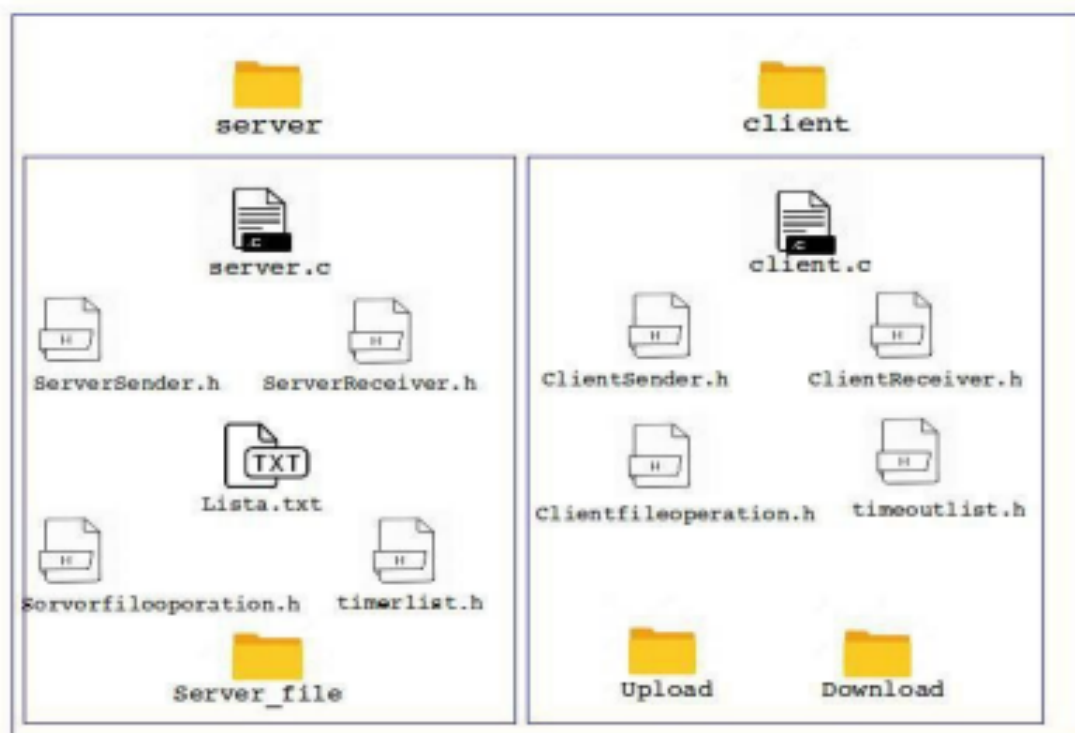


Figure 2.1: Architettura Sistema

### 2.1 Server

Innanzitutto come possiamo notare abbiamo un file “server.c”, il quale risponde in modo concorrente a tutte le richieste provenienti dai vari client. Successivamente abbiamo un file di testo chiamato “Lista.txt”, contenente i nomi di tutti i file posseduti dal server. Troviamo inoltre una cartella “Server\_file” con tutti i file posseduti dal server e infine abbiamo inserito una serie di header file, contenenti operazioni utilizzate per i seguenti scopi:

- **ServerSender.h:** gestire l’invio di file;
- **ServerReceiver.h:** gestire la ricezione di file provenienti da un client;
- **timerlist.h:** gestire tutti gli utenti impegnati in una conversazione privata;
- **Client.h:** gestire una lista collegata, utilizzata per realizzare la funzione del timeout;
- **serverfileoperation.h:** lavorare con i file (apertura, creazione, chiusura, ecc. . . );

## 2.2 Client

Riguardo l'architettura client, è presente un file principale, chiamato `Client.c`. Anche qui abbiamo diversi header file:

- **ClientSender.h**: gestire l'invio di file verso il server;
- **ClientReceiver.h**: gestire la ricezione di file provenienti da un server;
- **timeoutlist.h**: gestire una lista collegata, utilizzata per realizzare la funzione del timeout;
- **clientfileoperation.h**: lavorare con i file (apertura, creazione, chiusura, ecc. . . ).

Infine abbiamo due cartelle:

- **Download**: contiene i file che possono essere inviati ad un server tramite il comando “put filename.txt”;
- **Upload**: contiene i file ricevuti dal server tramite la richiesta “get filename.txt”;

### 3 SPECIFICHE IMPLEMENTATIVE

Dopo aver visto nel capitolo precedente l'architettura del nostro progetto, ci addentriamo ora nella descrizione della sua implementazione e delle scelte da noi effettuate in fase di sviluppo.

#### 3.1 Instaurazione della connessione

Una connessione tra un client ed un server si instaurerà nel momento in cui, il server dopo essersi messo in ascolto su una determinata porta, riceva una richiesta da parte di un client, caratterizzato anch'esso da una porta e un'indirizzo ip predefinito. **Ovviamente affinché si possa stabilire data connessione si deve rispettare il seguente vincolo: la porta di ascolto lato server e di connessione lato client deve essere la stessa.** Dunque ogni qualvolta un client si voglia collegare con il nostro server, si accoppierà ad esso tramite una porta predefinita, o di default. Tuttavia al server da noi realizzato, avendo la facoltà di accettare richieste da più client contemporaneamente, è stata assegnata la competenza di poter gestire porte differenti associate ai vari client. Ora il server, per il principio suddetto, assegnerà sequenzialmente, partendo dalla porta di default, una data porta al client richiedente. Quest'ultimo, ricevuta la porta, instaurerà una nuova connessione con il server, anch'esso messosi in ascolto su quella porta. Ovviamente anche nella gestione della porta a cui connettersi è stata garantita l'affidabilità, infatti il server in seguito all'invio del messaggio contenente la porta avrà bisogno di un messaggio di ACK (di conferma) da parte del client. Tanto è vero che il client nel momento in cui riceverà la porta in modo corretto avrà l'onere di segnalare la ricezione al server tramite ACK. Ora garantita l'affidabilità, il server si predisporrà, come su citato, alla ricezione di messaggi.

In fin dei conti, dal punto di vista implementativo, sia il server come il client avranno bisogno di istanziare un canale di comunicazione tramite la funzione socket. Tuttavia bisogna osservare come nell'utilizzo di detta funzione siano stati usati dei parametri appositi per l'utilizzo del protocollo UDP:

1. **SOCK\_DGRAM**: Supporta i datagrammi, ossia dei messaggi non affidabili;
2. **IPPROTO\_UDP**: campo formale specifico per il protocollo UDP;

Successivamente entrambi dovranno definire una struttura in cui si specificheranno la porta che verrà utilizzata, un campo predefinito `AF_INET` che, in concomitanza con il parametro `SOCK_DGRAM` sopra menzionato, ci permetterà a tutti gli effetti di instaurare una trasmissione UDP. Inoltre il server, a differenza del client, effettuerà la `bind()` così da far sapere al SO a quale processo verranno inviati i dati ricevuti dalla rete. Fatto ciò si metterà in attesa con l'intenzione di ricevere dati.

## 3.2 Gestione Concorrenza

Come detto in precedenza, Il server ha la competenza di offrire un servizio a più client simultaneamente. Per raggiungere tale scopo si è deciso di strutturare il server tramite il principio del multithreading. La nostra scelta è virata sul multithreading piuttosto che sul multiprocesso poiché, come sappiamo, nei server di rete, il cui carico lavorativo è molto elevato, esiste un limite di multiprogrammazione. Il server superato tale limite rifiuta richieste da parte dei client per evitare uno scenario in cui il server non riesce a servire nemmeno le richieste attive. L'utilizzo dei thread, inoltre, è molto meno oneroso a livello di utilizzo di risorse rispetto ad un approccio multiprocesso. Infatti un applicativo multithread presuppone la creazione di un unico processo con la conseguente partizione dello stack tra i vari thread. In fin dei conti ogni thread, come un processo, avrà il suo stack privato allocato tuttavia sullo stack fisico del `main_process`. Da questo punto di vista, l'approccio multiprocesso presuppone l'allocazione di maggior memoria e un maggior dispendio di risorse sulla macchina lavorante. Dal punto di vista di performance effettive è noto che utilizzare dei thread ci permette:

- un minor tempo di context switch;
- maggior efficienza del caching;
- minor tempo nella creazione e nella terminazione di un thread;
- miglior efficienza nella comunicazione e nella sincronizzazione dato che la comunicazione interprocesso richiede l'intervento del kernel, per motivi di protezione, con il conseguente impatto sulle prestazioni.

N.B: In genere risulta più conveniente implementare server concorrenti tramite multithreading, con creazione dinamica di un nuovo thread che si fa carico della gestione di una nuova richiesta di servizio, rispetto alla soluzione in cui ogni nuova richiesta determini l'attivazione di un nuovo processo.

### 3.2.1 Specifiche Concorrenza

Dal punto di vista implementativo abbiamo deciso di creare un thread-server per ogni client effettuante connessione. Oltretutto abbiamo preferito associare ad ogni thread una struttura privata, al fine di poter gestire le risorse in maniera ottimale e sicura. Infatti in questo modo ogni client avrà una sua struttura dedicata sul server con socket privata e porta privata. Inoltre l'utilizzo di una struttura privata ci ha permesso di controllare efficientemente la concorrenza tanto che tale scelta ci ha evitato l'utilizzo di innumerevoli risorse bloccanti quali semafori o mutex che altresì sarebbero stati doverosi in presenza di risorse comuni non private.

### 3.3 List: Visione file disponibili

Il client una volta stabilita la connessione con il server avrà la possibilità di richiedere un servizio. Il primo servizio che andiamo ad analizzare è il comando **LIST** il quale restituirà al client richiedente una lista dei file disponibili appartenenti al server. In questo modo il client visualizzerà il nome del file che in seguito potrà richiedere tramite il comando **GET <FILENAME>**

#### 3.3.1 Specifiche LIST

##### LATO CLIENT:

Nell'esecuzione del comando list il client, dopo aver inviato tramite sendto() il messaggio list al server, si predisporrà a ricevere una duplice risposta da parte del server. Infatti il client effettuerà la sendto() di richiesta, prima si preparerà per la ricezione del numero di file disponibili nel server e in seguito, avuta tale informazione, otterrà, uno per volta, tutti i nomi dei file nel server. Il client,ottenuti tali messaggi di risposta, si occuperà di restituire all'utente una risposta per la domanda formulata: farà visionare la lista dei file.

##### LATO SERVER:

Nell'esecuzione del comando list il server, dopo aver ricevuto tramite rcvfrom() il messaggio list da parte del client, pianificherà tutte quelle operazioni utili a fornire la giusta risposta al client.

1. Prima di tutto il server dovrà ricostruire il path-assoluto del file denominato da noi "LISTA.TXT" in cui sono presenti i nomi dei file disponibili. Al fine di raggiungere questo obiettivo abbiamo realizzato una funzione obtain\_path() la quale a seconda del comando dal client effettuato restituirà il path dell'istruzione richiesta. Si prenda anche in considerazione che tramite l'obtain\_path() si è diversificato il path da ottenere a seconda del run dell'applicazione o da terminale o dall'ide CLION;
2. Dopo di che il server partizionerà il contenuto del file ("LISTA.TXT") tramite funzione read\_file\_list() la quale simultaneamente ci fornirà l'esatto numero di file presenti sul server e un buffer di stringhe in cui in ogni singolo elemento sarà esattamente il nome di un file;
3. Dapprima Il server invierà il numero di file e successivamente, uno per volta, il nome del file.

### 3.4 Get/Put → Download/Upload File

#### 3.4.1 GET <FILENAME> :

Il client una volta stabilita la connessione con il server avrà la possibilità di richiedere un servizio. In questo paragrafo ci occuperemo di analizzare il funzionamento del comando **GET**. Quest' ultimo avrà la funzionalità di mettere a disposizione del client la facoltà di scaricare, tramite server, il file richiesto. Condizione necessaria e sufficiente affinché si verifichi ciò è l'assoluta presenza del file sia nella cartella **SERVER\_FILE** associata al server sia nel noto file **LISTA.TXT** che tiene traccia di tutti gli eventuali file adoperabili.

### 3.4.2 PUT <FILENAME> :

Il client una volta stabilita la connessione con il server avrà la possibilità di richiedere un servizio, come descritto nel caso della GET, ma anche di offrirlo come vedremo ora con la **PUT**. Infatti, in questo paragrafo ci occuperemo di analizzare il funzionamento del comando PUT. Quest'ultimo avrà la funzionalità di mettere a disposizione del client la facoltà di uploadare il file richiesto e quindi permettere al server di memorizzare il file immesso da client nella lista dei suoi file disponibili. Così facendo il Server, qualora un altro Client in futuro richiedesse tale file, ne permetterà facilmente il download. Condizione necessaria e sufficiente affinché si verifichi ciò è l'assoluta presenza del file nella cartella **UPLOAD** associata al Client. Fatto ciò il Server potrà aggiornare il file **LISTA.TXT** e memorizzarvi il nuovo file inviatogli.

### 3.5 Trasferimento file

Nel realizzare i due comandi sopra citati ci siamo incontrati nella difficoltà di capire come trasferire il file nel modo corretto. In questa sezione ci siamo posti, dunque, come obiettivo principale l'analizzare le scelte effettuate nel realizzare un trasferimento di dati che si dimostrasse affidabile senza tuttavia compromettere l'efficienza e la velocità.

Nel trasferire un file le maggiori responsabilità sono state assegnate al lato mittente dell'applicativo. Anche in questa circostanza, come già visto in precedenza riguardo all'instaurazione della connessione, la nostra scelta implementativa è ricaduta sull'utilizzo di thread operanti in concorrenza.

Ad ognuno di codesti thread è stata assegnata una funzionalità/responsabilità differente; così facendo si ha avuto la possibilità di suddividere il lavoro in maniera proporzionale, rendendo il tutto il più efficiente possibile. Dunque si è deciso di creare 3 thread così disposti:

- **Thread1** gestore dei pacchetti ha assolto la funzionalità di leggere i dati dal file, inserirli in un pacchetto a cui è associato un numero di sequenza ed inviarli.
- **Thread2** gestore del messaggio di acknowledgement
- **Thread3** - Main\_Thread - gestore del timeout ha provveduto a gestire le ritrasmissioni dei pacchetti "persi".

Data la presenza dei thread si è deciso anche in questa circostanza di instaurare una struct privata per gestire i thread. Tuttavia, a differenza di come è stata gestita la connessione, in questo contesto si è preferito utilizzare un'unica struct per gestire l'invio dei pacchetti. Infatti abbiamo inserito all'interno di questo struct un array di strutture associate alla gestione di un pacchetto. Per l'appunto ciascun pacchetto scambiato tra server e client è rappresentato come una struct `snd_pack`, i cui campi sono:

- **data:** un buffer per contenere l'informazione da trasportare;
- **acked:** un booleano che permette di capire se il pacchetto è stato riscontrato o meno;
- **seqnum:** un intero relativo al numero di sequenza (da 0 ad N);
- **finished:** un booleano per indicare l'ultimo pacchetto in cui è stato scomposto il file da inviare;

- **retransmitted:** booleano che permette di capire se il pacchetto è stato ritrasmesso.

In questo modo, data la presenza di un'unica struct tutti i thread lavorando in simbiosi tra loro sono a conoscenza della perdita, ritrasmissione o ricezione di un dato pacchetto identificato dal suo seq\_num.

Ci teniamo particolarmente a sottolineare che la dimensione di un pacchetto, così come il valore di  $N$ , sono dei valori costanti, che è possibile opportunamente modificare a proprio piacimento.

Se nel lato mittente si è preferito usare un approccio multithreading invece, per quanto riguarda il lato ricevente, abbiamo adoperato un approccio mono-processo costituito da un unico flusso di esecuzione responsabile di operazioni di ricezione, invio dell'ACK e scrittura su file.

Abbiamo deciso di utilizzare i thread al fine migliorare le prestazioni del nostro sistema, rendendolo scalabile tramite lo svolgimento di più operazioni in parallelo.

In generale, ciascun pacchetto scambiato tra server e client è rappresentato come una struct `snd_pack`, i cui campi sono:

- un buffer per contenere l'informazione da trasportare;
- un booleano;
- un intero relativo al numero di sequenza (da 0 ad  $N$ );
- un booleano per indicare ;

Ci teniamo particolarmente a sottolineare che la dimensione di un pacchetto, così come il valore di  $N$ , sono dei valori costanti, che è possibile opportunamente modificare a proprio piacimento.

### 3.5.1 Principio Buffer Circolare:

Per l'implementazione della ripetizione selettiva abbiamo deciso di fare uso di un buffer circolare di dimensione  $N$  e di alcune variabili, che abbiamo denotato con **W**, **WS**, **WR** per tenere conto della posizione delle finestre.

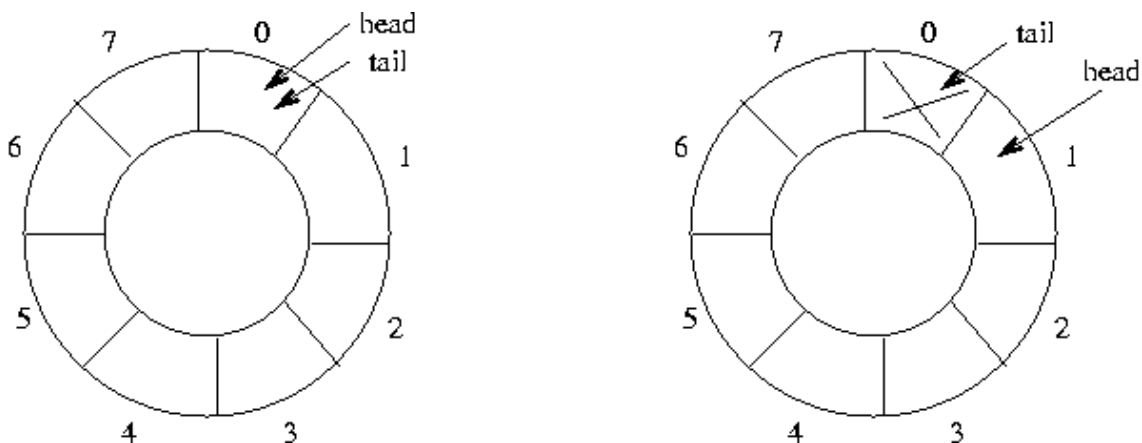


Figure 3.1: Buffer Circolare

Ciascun pacchetto, ha associato il proprio numero di sequenza alla posizione che questo occupa nel buffer prima di essere inviato e dopo essere stato ricevuto. Lo scorrimento della finestra è stato reso, nel nostro caso, incrementando le variabili associate al buffer circolare.

Tale buffer circolare deve essere utilizzato in modo esclusivo dai thread che implementano il lato mittente, poiché è necessaria la bufferizzazione dei pacchetti inviati.

Qualora un pacchetto venisse perso, oppure venisse perso l'ACK ad esso relativo, questo dovrà essere ritrasmesso dal thread incaricato, il quale lo andrà a prelevare dal buffer in cui si trova.

Il thread gestore dei riscontri, ovvero dei messaggi di acknowledgement, libererà il buffer, quando questo pacchetto non dovrà più essere utilizzato. Generalmente la bufferizzazione è utile ed importante anche dal lato ricevente, poiché è probabile che i pacchetti non arrivino con il giusto ordine.

Un problema con cui ci siamo dovuti confrontare è stato come capire e gestire la terminazione di una trasmissione. Dopo un'attenta riflessione, abbiamo affidato tale compito ad una procedura finale, in cui il thread che si occupa di leggere i pacchetti e bufferizzarli, realizza di aver terminato il proprio compito e notifica tale evento anche al thread in attesa di ricevere gli ack.

Quest'ultimo avvia, da questo momento in poi, una procedura che consiste nell'inviare un particolare pacchetto con un numero di sequenza al di fuori della dimensione dei buffer di invio e di ricezione.

A questo punto, il destinatario nota un pacchetto con un numero di sequenza maggiore o uguale a N e capisce di aver ricevuto tutto il file; all'ultimo riscontro ricevuto, il thread gestore dei messaggi di acknowledgement, termina il proprio lavoro.

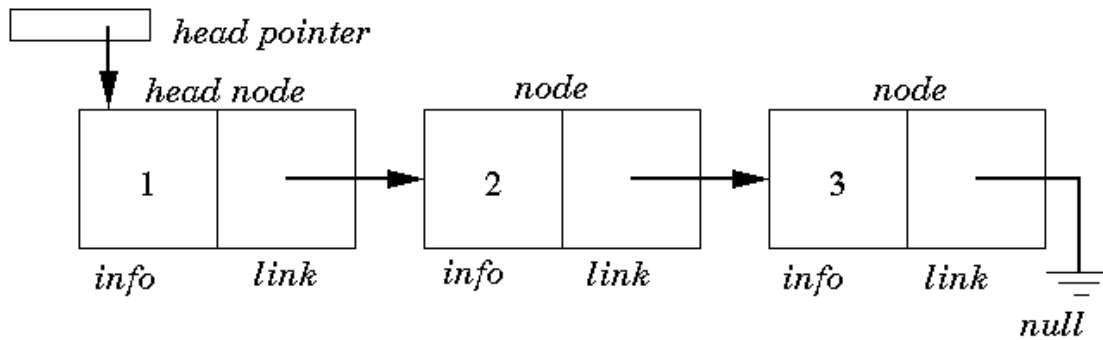
L'ultimo ack, quello finale, sarà inviato dal ricevitore una volta, o tre, a seconda se si tratti del client o del server. Un'altra caratteristica molto importante del protocollo Selective Repeat, è quella che ad ogni pacchetto abbiamo associato un timer, alla cui scadenza avviene una nuova trasmissione.

Il motivo di questa scelta risiede nel fatto che il pacchetto potrebbe essersi perso in rete, oppure potrebbe essersi perso l'ACK a esso associato.

Andando a soffermarci sulle nostre scelte implementative, abbiamo deciso di gestire il timeout facendo uso di una lista collegata come struttura di appoggio. Ciascun elemento di questa lista, chiamato `node_t`, è una struct che presenta i seguenti campi:

- un intero `seq_num`, relativo al numero di sequenza del pacchetto;
- un booleano `finished`, per identificare l'ultimo pacchetto della sequenza;
- una struct `timeval`;
- un puntatore al nodo successivo.





## A Linked List

Figure 3.2: A Linked List

Quando un pacchetto viene inviato, viene aggiunto come ultimo di questa lista collegata, controllata dal thread gestore del timeout. Quest'ultimo si occupa di confrontare costantemente il tempo di volo del primo elemento della lista, ovvero il pacchetto che da più tempo si trova "in volo", con il valore del timeout specificato sempre come variabile globale. Per quanto riguarda invece il thread gestore dell'ACK, se riceve ACK di pacchetti che però non hanno ancora avuto riscontro, scorre la lista collegata alla ricerca dei giusti nodi, azzerava i loro timer e infine li elimina.

### 3.5.2 Scambio di messaggi

Risolto il problema del trasferimento affidabile di un file grazie all'introduzione del multithreading e all'utilizzo del buffer circolare, si è posto nuovamente davanti a noi il problema del trasferimento affidabile. Infatti non doveva essere reso affidabile solo il trasferimento dei file ma anche un qualsiasi scambio di informazione tra client e server; dunque anche l'invio di un semplice comando presupponeva l'affidabilità e la garanzia di ricezione.

Tuttavia abbiamo pensato che realizzare un trasferimento affidabile per un flusso di pochi byte seguendo la stessa procedura tortuosa svolta per i file fosse una cosa poco intelligente, abbiamo optato per una strategia più banale e a cui qualcuno già aveva pensato per noi. Infatti la nostra decisione nel realizzare ciò è ricaduta nel ricreare un meccanismo noto come "three-way-handshake" utilizzato già dal protocollo TCP per l'instaurazione della connessione.

Scovata la nostra strategia risolutiva abbiamo sviluppato una forma di scambio messaggi basato su diversi tentativi e timer di attesa associati ricreando così per l'appunto un handshake che potesse assicurare sia al client che al server l'arrivo dei propri messaggi. L'invio e la ricezione di questi messaggi è possibile grazie alle funzioni `sndMsg()` e `rcvMsg()` al cui interno vengono sfruttate due `sendto()` e due `receivefrom()`.

La prima fase per instaurare la connessione inizia con l'invio del pacchetto INFO e si conclude d'altro canto con l'invio dell'ACK da parte del destinatario. La seconda fase, in sostanza, se va tutto liscio si conclude con la ricezione dell'ACK da parte del mittente. Tuttavia il destinatario non può avere la certezza che l'ACK inviato sia stato ricevuto correttamente. Per ovviare a ciò si è pensato di introdurre un ulteriore scambio di messaggi oltre la necessità di impostare il timeout sulle `receivefrom()` del mittente e del destinatario. Dunque ora analizziamo il caso in cui il mittente non riceva l'ACK, anche in conseguenza alla perdita del pacchetto; se dovesse succedere ciò scadrà il timer associato alla receive-

from() e per tre volte cercherà di rinviare il pacchetto INFO fino a ricevere il relativo oppure terminando con messaggio di errore.

Il destinatario ,invece, ogni volta che riceve un messaggio invia l'ACK e imposta il timeout della receivefrom() su un valore che consenta la ritrasmissione multipla da parte del mittente; anche in questo caso se scade timer il destinatario lo notificherà terminando con un messaggio di errore.

Successivamente il mittente per avvisare della ricezione dell'ACK elaborerà un messaggio di tipo FINACK; infatti ora se il destinatario riceve un messaggio di FIN capirà che il mittente ha ricevuto l'ACK e cercherà di informarlo nuovamente con un ACK.

Ora, come prima, se il mittente non riceverà l'ACK(FINACK) del destinatario, tenterà per tre volte di recapitare il messaggio. Se alla fine mittente riceve il FINACK lo scambio si considera avvenuto correttamente e smette di inviare FIN. Il destinatario finché riceverà messaggi FIN, continuerà ad inviare FINACK e nel momento che non riceverà più nulla e scadrà il timer, controllerà in una variabile se ha ricevuto almeno un messaggio FIN; in tal caso considera lo scambio avvenuto con successo ritornando il messaggio INFO ricevuto inizialmente.

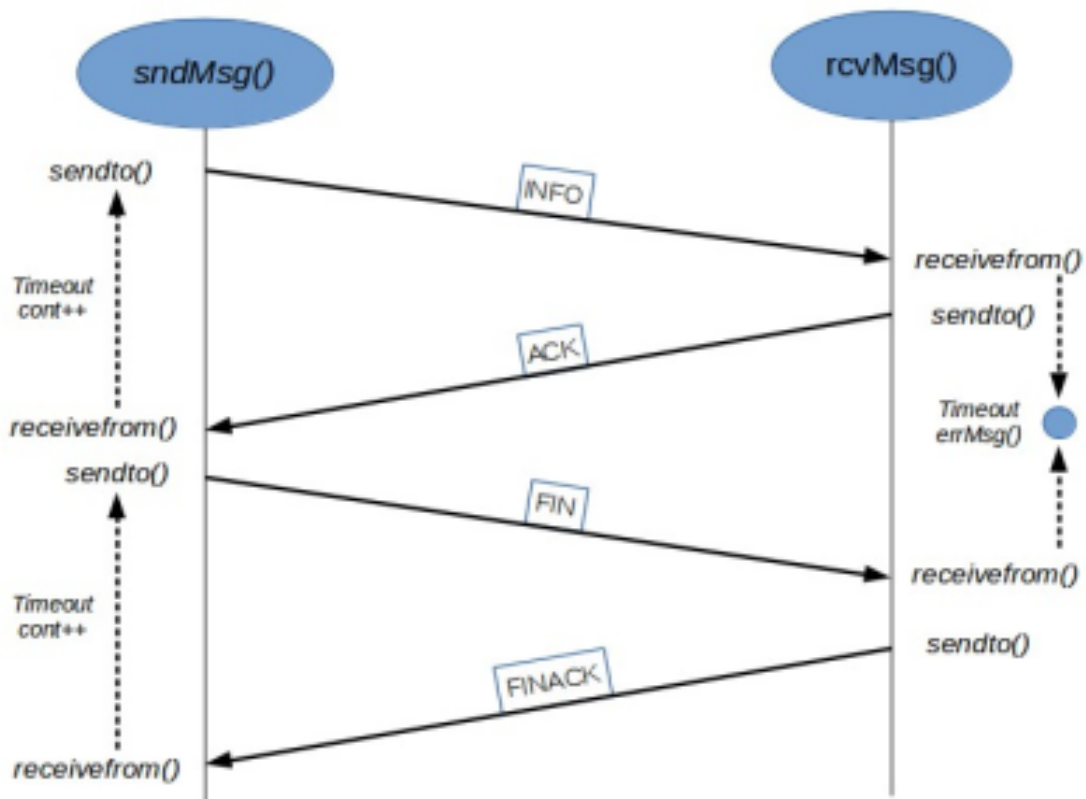


Figure 3.3: Scambio di Messaggi

### 3.6 Disconnessione client-server

Un Client per disconnettersi dal server associato, ossia dal thread i-esimo chi si occupa della sua gestione, può optare per due strade differenti:

1. La prima casistica consiste nell'immettere da tastiera il comando **exit**, presente nella schermata dei comandi disponibili per l'applicativo;
2. La seconda scelta si riversa sull'invio e di conseguenza gestione di uno dei seguenti segnali: **SIGINT**,**SIGQUIT**.

Tuttavia la seconda strada percorribile può essere testata lanciando l'applicativo da terminale.

**OSS: - SIGINT(ctrl+C) - SIGQUIT(ctrl+\)**

#### 3.6.1 COMANDO <EXIT>

Nel momento in cui un Client decida di disconnettersi avrà la possibilità di farlo inviando il comando exit al Server. Quest'ultimo riconosciuto il comando dapprima chiuderà il canale di comunicazione(socket) istanziata per il corrispettivo Client e successivamente rilascerà tutte le risorse associate allo specifico Client terminando il thread ad esso associato. Inoltre Il Server, il quale tiene traccia di tutti gli utenti connessi simultaneamente, aggiornerà tale statistica. Fatto ciò Il Client terminerà con successo stampando messaggio di terminazione con successo.

#### 3.6.2 SEGNALI SIGINT,SIGQUIT

Nel momento in cui un Client decida di disconnettersi avrà la possibilità di farlo inviando un segnale SIGINT o SIGQUIT al Server. Il Client catturerà uno di questi segnali e si occuperà della loro gestione richiamando un handler. Tale handler avrà la funzione di incrementare una variabile globale detta "segnale" , settata per default a 0. Una volta incrementata il flusso esecutivo entrerà in una porzione di codice che avrà la gestione della terminazione del Client. Infatti il Client avviserà il Server con un messaggio di notifica e quest'ultimo ricevuto tale avvertimento seguirà le stesse procedure descritte nel paragrafo 3.5.2.

## 4 ESEMPI DI FUNZIONAMENTO

In questo capitolo ci soffermiamo su alcuni esempi atti ad evidenziare come avviene la comunicazione tra client e server, supponendo una probabilità di perdita dei pacchetti pari al 5%. Abbiamo deciso di inserire, nelle sezioni sottostanti, alcune stampe in modo da illustrare le tipologie di messaggi scambiati (messaggi di comando e messaggi di risposta).

### 4.1 Schermata iniziale

Nel momento in cui l'applicazione viene avviata si prospetta all'utente un menù di comandi disponibili tra i quali scegliere. Nella schermata sottostante forniamo una dimostrazione di ciò che l'utente si troverà davanti al run del nostro applicativo. Inoltre daremo la possibilità all'utente di controllare il suo indirizzo IP e il numero di porta tramite la quale si è instaurata la connessione con il server.

```
Linked server ip: 127.0.0.1 8890

+-----+
|ELENCO  COMANDI:|
+-----+
| 1) list: elenco dei file presenti nel Server|
| 2) get <Filename>: Download del file      |
| 3) put <Filename>: Upload del file         |
| 4) exit                                   |
+-----+
ENTER MESSAGE:
```

Figure 4.1: Schermata iniziale

#### WARNING:

Ci teniamo a precisare che qualora l'utente non selezioni un comando tra quelli qui sotto elencati oppure erroneamente non immetta il comando nel modo corretto (GET E PUT) verrà avvisato dal server tramite un messaggio di WARNING.

1. gestione di un comando nullo o troppo breve;

```
ENTER MESSAGE: |
ATTENZIONE: Hai inserito un comando non valido. Riprova
```

Figure 4.2: Comando nullo o troppo breve

2. Comando non riconosciuto tra quelli implementati;

```
ENTER MESSAGE:      comando errato
Attenzione : comando non valido!
Linked server ip: 127.0.0.1 8890
```

Figure 4.3: Comando non valido

## 4.2 Messaggi di comando

### 4.2.1 Comando “list”

Tramite l'utilizzo del comando “list”, un client richiede al server la lista di file disponibili al download. Il client una volta effettuata la richiesta si metterà in attesa di ricevere il numero di file presenti all'interno del file “LISTA.txt”, tenente traccia dei file disponibili, e successivamente riceverà e mostrerà a video i nomi dei file. Il server al fine di elaborare tale richiesta scansionerà il file “LISTA.txt”, terrà conto del numero dei file presenti e del nome degli effettivi file. Fatto ciò invierà tali informazioni in due tranches separate:

1. Numero dei file;
2. File disponibili;

```
ENTER MESSAGE:      list
Number File 6
+-----+
1) Decamerone.txt
+-----+
2) OrlandoFurioso.txt
+-----+
3) Divina.txt
+-----+
4) Vuoto.txt
+-----+
5) Autori_Progetto.tx
+-----+
6) Bibbia.txt
+-----+
Linked server ip: 127.0.0.1 8890
```

Figure 4.4: Funzionamento Comando List

### 4.2.2 Comando “get”

Il client, inviando il messaggio “get nomefile.txt”, chiede al server di inviargli il file specificato. Il server, ricevuto tale comando, dopo essersi assicurato di possedere effettivamente il file, inizia il trasferimento dell’informazione, altrimenti viene restituito al client un messaggio di errore. Il file ricevuto da quest’ultimo, viene inserito in una cartella chiamata “upload” del client e prelevato dalla cartella “download” del server.

1. Mostra il caso in cui il download è avvenuto con successo;

```
ENTER MESSAGE:      get Bibbia.txt
Current path -> /home/marco96/Scrivania/IIW2_final/Client/Download/Bibbia.txt
Ho ricevuto il file Bibbia.txt
Linked server ip: 127.0.0.1 8890
```

Figure 4.5: Download avvenuto con successo

2. Mostra il caso in cui c’è il tentativo di eseguire il download di un file non presente in Lista.txt;

```
ENTER MESSAGE:      get File11.tssas
Il file richiesto non è presente nella lista dei file disponibili del Server
Linked server ip: 127.0.0.1 8890
```

Figure 4.6: File non presente in Lista.txt;

3. Mostra il caso in cui c’è il tentativo di eseguire la get senza un file specificato;

```
ENTER MESSAGE:      get
ATTENZIONE: Devi inserire un file
Linked server ip: 127.0.0.1 8890
```

Figure 4.7: get senza nessun file

4. Mostra il caso in cui c’è il tentativo di eseguire la get con un file di dimensione zero → VUOTO ;

```
ENTER MESSAGE:      get Vuoto.txt
Attenzione: file richiesto è vuoto riprova con un altro file!
Linked server ip: 127.0.0.1 7778
```

Figure 4.8: get con file Vuoto

### 4.2.3 Comando “put”

Tramite l'utilizzo del comando “put nomefile.txt” il client può inviare un determinato file di testo al server. Tale file viene prelevato dalla cartella “upload” del client e inviato nella cartella “Server\_file” del server. Infine, viene aggiornato il file di testo dell'elenco dei file(Lista.txt), aggiungendo il nome del nuovo file ricevuto.

1. Mostra il caso in cui l'upload è avvenuto con successo;

```
ENTER MESSAGE:      put Autori_Progetto.txt
filepath -> /home/marco96/Scrivania/IIW2_final/cmake-build-debug
/home/marco96/Scrivania/IIW2_final/Client/Upload/
FILESIZECLIENT->45
buf_size -> 45
mandati_size -> 8
PASS01: INIZIALIZZO STRUCT SND_THREAD_INFO
fd: 17
sock_fd: 14
byte letti: 45
senderManager: send_base:0,next_tosend:1
byte letti: 0
ackManager: send_base:0, next_tosend:1
struct timer t2->tv_sec: 1547576314
Ricevuto ACK 0, Timeout = 919
SampleRTT:922,EstimatedRTT:115,DevRTT:201,TIMEOUT:919

101
File sent: Autori_Progetto.txt
Linked server ip: 127.0.0.1 7778
```

Figure 4.9: Upload avvenuto con successo

2. Mostra il caso in cui c'è il tentativo di eseguire l'upload di un file non presente nella cartella Upload;

```
ENTER MESSAGE:      put Ciccio.txt
filepath -> /home/marco96/Scrivania/IIW2_final/cmake-build-debug
/home/marco96/Scrivania/IIW2_final/Client/Upload/
Il file non è presente nella cartella Upload quindi non posso mandarlo
Linked server ip: 127.0.0.1 7778
```

Figure 4.10: File non presente in Upload

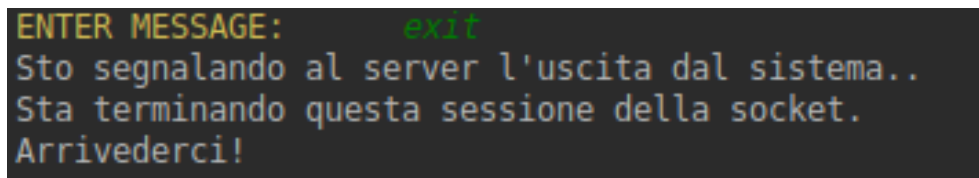
3. Mostra il caso in cui c'è il tentativo di eseguire la put senza un file specificato;

```
ENTER MESSAGE:      put
ATTENZIONE: Devi inserire un file
Linked server ip: 127.0.0.1 7778
```

Figure 4.11: put senza nessun file

#### 4.2.4 Comando “exit”

Nel momento in cui il client immette il comando exit si predisporrà alla terminazione della connessione. Il client, scelto il comando exit, invierà tale messaggio al server che si occuperà di disconnettere il thread i-siemo responsabile della gestione del relativo Client. Fatto ciò deallocherà tutte le risorse ad esso associate permettendo così al Client di scollegarsi. Infine, il client stamperà un messaggio di terminazione e di disconnessione dal server.

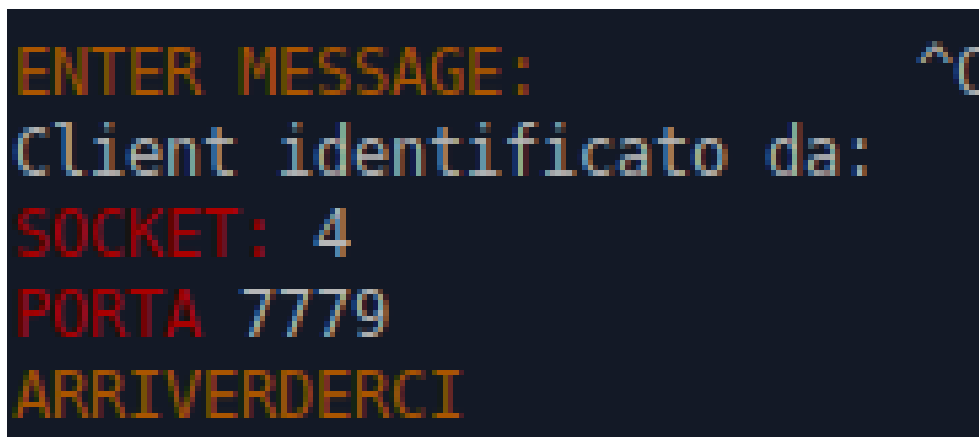


```
ENTER MESSAGE:      exit
Sto segnalando al server l'uscita dal sistema..
Sta terminando questa sessione della socket.
Arrivederci!
```

Figure 4.12: Disconnessione dal Server

#### 4.2.5 Invio segnale

La gestione del segnale è una sotto-casistica del comando exit, o meglio, esso ci permette in modo medesimo di disconnettersi dal server. Il client, una volta catturato il segnale (SIGINT/SIGQUIT), invierà un messaggio di avvertimento al server che inizierà in questo modo tutte le procedure di disconnessione già discusse nel comando exit.



```
ENTER MESSAGE:      ^C
Client identificato da:
SOCKET: 4
PORTA 7779
ARRIVERDERCI
```

Figure 4.13: Esempio Disconnessione Segnale SIGINT



### 4.3 Messaggi di risposta

Nei paragrafi precedenti ci siamo soffermati sulle risposte che il Client in riceve dal Server in seguito all'immissione di un comando. Analogamente il Server come reagirà a tali richieste?

#### 1. LIST:

```
OSSERVA: La porta del client a cui si fa riferimento è 7778 la cui socket è 4;  
Data: list  
Current_path -> /home/marco96/Scrivania/IIW2_final/Server/Lista.txt  
Number_file 6  
TEMPO TOTALE : 0.069900
```

Figure 4.14: Risposta del Server alla list

#### 2. GET:

```
Data: get Autori_Progetto.txt  
Inviato pacchetto 0  
senderManager: send_base:0,next_tosend:1  
,Scaduto timer pacchetto 0, rinviato  
Ricevuto ACK 0, Timeout = 2000000  
SampleRTT:0,EstimatedRTT:0,DevRTT:0,TIMEOUT:2000000  
  
Ho ricevuto tutti gli ack  
  
Inviato pacchetto 101  
Ricevuto ACK 101  
Ho stampato il file Autori_Progetto.txt
```

Figure 4.15: Risposta del Server alla get

#### 3. PUT:

```
Data: put Autori_Progetto.txt  
file descriptor -> 5  
size -> 45  
Ricevuto pacchetto 0  
Scritto pacchetto 0  
Ricevuto pacchetto 101  
Ho ricevuto tutti i pacchetti  
File già presente in lista. Non devo aggiornare la lista dei file disponibili su Server  
Ho ricevuto il file Autori_Progetto.txt
```

Figure 4.16: Risposta del Server alla put

#### 4. EXIT:

```
OSSERVA: La porta del client a cui si fa riferimento è 7778 la cui socket è 4;  
Data: exit  
  
Sta terminando questa sessione la cui porta è 7778 e la socket di riferimento è 4 .
```

Figure 4.17: Risposta del Server alla exit

## 5 PIATTAFORME UTILIZZATE

Per quel che riguarda la fase di sviluppo e il testing del nostro sistema abbiamo deciso di installare ed utilizzare sulle nostre macchine Linux, la piattaforma CLion messa a disposizione da JetBrains e ottenuta sfruttando la licenza studenti valida per un anno. Si tratta di una piattaforma intuitiva e semplice da utilizzare. In tutto il percorso lavorativo abbiamo sfruttato il meccanismo di debug contenuto al suo interno, per poter risolvere problemi di varia natura. Di seguito riportiamo le caratteristiche dei calcolatori da noi utilizzati.

### Piattaforma 1 - M

Nome	Asus <u>VivoBook</u> A551LN
Processore	Intel® Core™ i 7-4510U CPU @ 2.00GHz × 4
Memoria	11,6 <u>GiB</u>
Sistema Operativo	Ubuntu 18.04.01 <u>LTS</u>
Disco	515,8

### Piattaforma 2 - V

Nome	Toshiba-Satellite-Pro-L300
Processore	Intel® Core™ 2 Duo CPU T5870 @ 2.00GHz × 2
Memoria	2,8 <u>GiB</u>
Sistema Operativo	Ubuntu 16.04 <u>LTS</u>
Disco	243,0 <u>GB</u>

### Piattaforma 3 - A

Nome	<u>DELL Inspiron-15-3552</u>
Processore	Intel® <u>Celeron®</u> CPU N3060 @ 1.60GHz × 2
Memoria	3.8 <u>GiB</u>
Sistema Operativo	Ubuntu 14.04 <u>LTS</u>
Disco	480,7 <u>GB</u>

## 6 PIATTAFORME UTILIZZATE

Dopo un'attenta analisi del codice prodotto, riportiamo di seguito le limitazioni da noi riscontrate:

- Qualora si riscontrasse un' interruzione del download, i file non correttamente scaricati devono essere rimossi manualmente dal client o dal server;
- In caso di “get” oppure “post” di un file che possiede lo stesso nome di un file già presente, il risultato sarà un file dal contenuto corrotto, a meno che il contenuto di entrambi non sia lo stesso;
- Il numero massimo di thread attivi, e quindi di client che possono comunicare con il server è pari a 50, valore che però può essere modificato;
- Non ci siamo posti nella situazione in cui due client vogliono nello stesso momento mettersi in contatto con il server, di conseguenza non sappiamo come il nostro sistema potrebbe rispondere;

## 7 VALUTAZIONI PRESTAZIONALI

### 7.1 Test prestazionali

I test prestazionali da noi eseguiti sono mirati a vedere come variano le prestazioni del nostro sistema al variare di alcuni parametri ritenuti significativi. Nelle pagine successive abbiamo inserito dieci casi di test; per ognuno sono presenti due tabelle, nella prima abbiamo una breve descrizione del test effettuato, mentre nella seconda vengono riportati tutti i valori da noi ottenuti su ciascuna piattaforma.

#### 7.1.1: Test 1

Nome test	Connessione del client $i$ -esimo con l' $i$ -esimo thread del server.
Descrizione	Il seguente test è volto a calcolare il tempo impiegato dal client $i$ -esimo per connettersi con l' $i$ -esimo thread del server. I valori ottenuti sono stati calcolati considerando i primi dieci thread.
Unità di misura tempo	microsecondi
Note	Media pesata sui primi 10 valori

#### Risultati

Thread $i$	Tempo ottenuto M	Tempo ottenuto V	Tempo ottenuto A
1	0,07253	0,20238	0,965900
2	0,03960	0,14038	0,103000
3	0,04400	0,115940	0,164900
4	0,03360	0,212600	0,1135900
5	0,04110	0,1062200	0,965900
6	0,04940	0,1314200	0,913200
7	0,03780	0,1344200	0,941400
8	0,04390	0,1141700	0,817700
9	0,04510	0,1205200	0,8508
10	0,04600	0,1845800	0,105120

### 7.1.2: Test 2

Nome test	Test variazione probabilità $p$ in funzione dei thread attivi
Descrizione	Il seguente test è volto a calcolare come varia il tempo impiegato per svolgere una chiamata "get", facendo variare la probabilità di perdita $p$ . I risultati sono riportati in termini di un solo client in esecuzione, oppure cinque.
Unità di misura tempo	microsecondi
Note	Parametri utilizzati: $N=100$ , $w_s = 50$ , $w_r = 50$ . Comando: "get" File trasferito : "Bibbia.txt" Media pesata sui primi 5 valori.

### Risultati

$p$	Thread $i$	Tempo ottenuto M	Tempo ottenuto V	Tempo ottenuto A
0.01	1	31,3658	41,90323	47,30943
0.01	5	40,2000	46,76390	53,42672
0.05	1	35,12840	48,02370	53,90921
0.05	5	45,84100	52,18420	63,10231
0.1	1	49,6938	57,28809	68,0088
0.1	5	53,9180	68,63420	78,8355
0.5	1	61,92485	75,02839	85,21176
0.5	5	64,11100	80,01180	89,9841

### 7.1.3: Test 3

Nome test	Test variazione probabilità $p$ in funzione dei thread attivi
Descrizione	Il seguente test è volto a calcolare come varia il tempo impiegato per svolgere una chiamata "put", facendo variare la probabilità di perdita $p$ . I risultati sono riportati in termini di un solo client in esecuzione, oppure cinque.
Unità di misura tempo	microsecondi
Note	Parametri utilizzati: $N=100$ , $w_s = 50$ , $w_r = 50$ . Comando: "put" File trasferito : "Bibbia.txt" Media pesata sui primi 5 valori.

### Risultati

$p$	Thread $i$	Tempo ottenuto M	Tempo ottenuto V	Tempo ottenuto A
0.01	1	36,49700	43,281602	44,16626
0.01	5	41,83821	46,902648	48,42671
0.05	1	46,75100	52,016253	50,2109
0.05	5	48,00300	55,102634	62,2109
0.1	1	52,669200	55,019320	78,254075
0.1	5	56,14260	57,103639	79,10802
0.5	1	61,591340	62,018364	88,200425
0.5	5	67,13450	64,274011	89,12123

#### 7.1.4: Test 4

Nome test	Test variazione dimensione finestra N
Descrizione	Il seguente test è volto a calcolare come varia il tempo impiegato per svolgere una chiamata " <u>get</u> ", facendo variare la dimensione della finestra N. Con probabilità di perdita 0.01
Unità di misura tempo	microsecondi
Note	Comando: " <u>get</u> " File trasferito : " <u>Bibbia.txt</u> " Media pesata sui primi 5 valori.

#### Risultati

N	<u>wr</u>	<u>ws</u>	<u>Thread</u> -i	Tempo ottenuto M	Tempo ottenuto V	Tempo ottenuto A
1000	250	250	1	11,92374	13,91600	14,3075
1000	250	250	5	19,10982	21,01739	20,8612
200	100	100	1	28,64374	31,19531	37,3747
200	100	100	5	34,26980	37,02163	42,4481
20	10	10	1	80,41996	84,10361	87,6402
20	10	10	5	89,39510	87,03204	90,0088
50	25	25	1	49,30208	47,29026	58,3602
50	10	10	5	51,64588	53,02836	65,3014

#### 7.1.5: Test dimensione della finestra N con parametri $p = 0.05$ , media pesata in 5 comando "put", file "Bibbia.txt" Test 5

Nome test	Test variazione dimensione finestra N
Descrizione	Il seguente test è volto a calcolare come varia il tempo impiegato per svolgere una chiamata " <u>put</u> ", facendo variare la dimensione della finestra N.
Unità di misura tempo	microsecondi
Note	Probabilità $p = 0.05$ Comando: " <u>put</u> " File trasferito : " <u>Bibbia.txt</u> " Media pesata sui primi 5 valori.

## Risultati

N	<u>wr</u>	<u>ws</u>	<u>Thread</u> -i	Tempo ottenuto M	Tempo ottenuto V	Tempo ottenuto A
1000	250	250	1	13,84223	15,46811	18,46951
1000	250	250	5	16,98522	23,22688	22,78212
200	100	100	1	27,23210	30,19531	34,37472
200	100	100	5	32,21231	34,87821	39,44811
20	10	10	1	77,44236	84,46829	83,19531
20	10	10	5	80,92313	87,46860	87,98522
50	25	25	1	50,32313	51,97531	55,32313
50	10	10	5	54,24698	56,64826	60,46985

### 7.1.6: Test 6

Nome test	Test tempo impiegato dal comando " <u>list</u> "
Descrizione	Il seguente test è volto a calcolare il tempo impiegato dal comando <u>list</u> in funzione del numero di <u>thread client</u> in esecuzione: 1, 2 oppure 5.
Unità di misura tempo	microsecondi
Note	Media pesata sui primi 10 valori.

## Risultati

<u>Thread i</u>	Tempo ottenuto M	Tempo ottenuto V	Tempo ottenuto A
1	0.03275	0,02241	0.11725
2	0.03399	0,04912	0.07909
5	0.03075	0,02133	0,10416



### 7.1.7: Test 7

Nome test	Test tempo impiegato dal comando " <u>get</u> "
Descrizione	Il seguente test è volto a calcolare il tempo impiegato dal comando " <u>get</u> " in funzione del numero di <u>thread client</u> in esecuzione: 1, 2 oppure 5.
Unità di misura tempo	microsecondi
Note	Probabilità di perdita $p = 0.05$ Media pesata sui primi 10 valori. File trasferito : " <u>Bibbia.txt</u> "

### Risultati

<u>Thread i</u>	Tempo ottenuto M	Tempo ottenuto V	Tempo ottenuto A
1	24,63752	40,8467	31,3713
2	26,14314	31,0147	36,9222
5	29,03473	38,9764	49,37092

### 7.1.9:Test 9

Nome test	Test tempo impiegato dal comando " <u>get</u> " con variazione della dimensione del pacchetto
Descrizione	Il seguente test è volto a calcolare il tempo impiegato dal comando " <u>get</u> " al variare della dimensione del singolo pacchetto.
Unità di misura tempo	microsecondi
Note	Probabilità di perdita $p = 0.05$ Media pesata sui primi 5 tentativi. File trasferito : " <u>Bibbia.txt</u> "

### Risultati

Dimensione Pacchetto	Tempo ottenuto M	Tempo ottenuto V	Tempo ottenuto A
500	39,16260	43,01734	44,22672
1000	35,81230	38,19284	38,90921
4000	22,04748	24,03390	25,7531

### 7.1.10: Test 10

Nome test	Test tempo impiegato dal comando "post" con variazione della dimensione del pacchetto
Descrizione	Il seguente test è volto a calcolare il tempo impiegato dal comando "post" al variare della dimensione del singolo pacchetto.
Unità di misura tempo	microsecondi
Note	Probabilità di perdita $p = 0.05$ Media pesata sui primi 5 tentativi. File trasferito : <u>"Bibbia.txt"</u>

### Risultati

Dimensione Pacchetto	Tempo ottenuto M	Tempo ottenuto V	Tempo ottenuto A
500	35,73000	38,01736	40,9636
1000	32,69098	34,02735	36,2109
4000	21,332420	26,01264	28,1082

## 8 MANUALE D'USO

Il capitolo finale di questa relazione lo dedichiamo ad una semplice guida all'installazione del nostro sistema, il cui codice sorgente può essere trovato nel CD allegato. Seguendo i passi sottoelencati è possibile eseguire il codice sul proprio dispositivo:

- **prelevare il file compresso “IIW” e spostarlo all'interno del proprio file system;**
- **decomprimere la cartella;**
- **aprire la cartella “codice”;**
- **aprire due terminali separati e posizionarsi nella giusta cartella;**
- **con il primo terminale compilare il file “server.c” all'interno della cartella “server”;**
- **con il secondo terminale compilare il file “client.c” all'interno della cartella “client”;**
- **eseguire infine entrambi.**