Reliable File Transfer

**Progetto di Ingegneria di Internet e Web 2024/25**

**Travaglini Sara**



**Indice**

1. [**Introduzione implementazione** 3](#_Toc198225141)

[1.1 Tipologie di richieste del Client 3](#_Toc198225142)

[1.2 Tipologie di errore 3](#_Toc198225143)

1. [**Comunicazione Client-Server** 4](#_Toc198225144)

[2.1 Connessione iniziale 4](#_Toc198225145)

[2.2 Chiusura della connessione 5](#_Toc198225147)

1. [**Trasferimento affidabile dei File 5**](#_Toc198225152)

[3.1 Sender Selective Repeat 5](#_Toc198225153)

3.2 Receiver Selective Repeat…………………………………………………………………….............7

1. [**Scelte progettuali** 7](#_Toc198225154)

[4.1 Server multiprocess 7](#_Toc198225155)

[4.2 Timeout adattivo 8](#_Toc198225156)

[4.3 Configurazione parametri del sistema 8](#_Toc198225157)

[4.4 Struttura del codice 9](#_Toc198225158)

[4.5 Gestione file 9](#_Toc198225159)

1. [**Installazione ed esempi di utilizzo** 9](#_Toc198225160)

[5.1 Installazione 10](#_Toc198225161)

[5.2 Interfaccia utente di avvio 10](#_Toc198225162)

[5.3 Interfaccia del server 11](#_Toc198225163)

[5.4 Esempi di utilizzo 11](#_Toc198225164)

1. [**Prestazioni del sistema** 13](#_Toc198225165)

[6.1 Simulazione *List* 13](#_Toc198225166)

[6.2 Simulazione *Get* 13](#_Toc198225169)

[6.3 Simulazione *Put* 14](#_Toc198225172)

6.4 Osservazioni generali simulazioni ..………………..…………..………………….………….15

1. [**Manuale d’uso** 16](#_Toc198225174)

[7.1 Configurazione 16](#_Toc198225175)

[7.2 Compilazione ed Esecuzione 16](#_Toc198225176)

**8. Conclusioni……………………………………………………………………………………………......….17**

# Introduzione implementazione

Il progetto consiste nel realizzare un'applicazione su un'architettura Client-Server, con il server in grado di gestire in contemporanea le richieste di molteplici client, per il trasferimento affidabile di file utilizzando il protocollo di trasporto UDP. L’applicazione è stata sviluppata in linguaggio C utilizzando le socket API Berkley.

Il sistema è stato progettato e testato interamente in ambiente Unix.

### 1.1 Tipologie di richieste del Client

I comandi che il client deve poter eseguire sono:

1. **List**: consente di ottenere dal server la lista dei file contenuti nella cartella di download;
2. **Get**: consente di trasferire sul client (in una apposita cartella) un file presente nella cartella di download del server;
3. **Put**: consente di trasferire sul server (nella cartella di download) un file presente nella cartella locale del client.

Ricevuta la richiesta il server ne verifica la validità:

-nel caso non sia valida informa il client con un messaggio contente un codice di errore specifico

### 1.2 Tipologie di errore

|  |  |
| --- | --- |
| *Comando* | *Errore* |
| GET | Error opening file: File not found |
| PUT | Error opening file: File already exist please rename it before trying again |

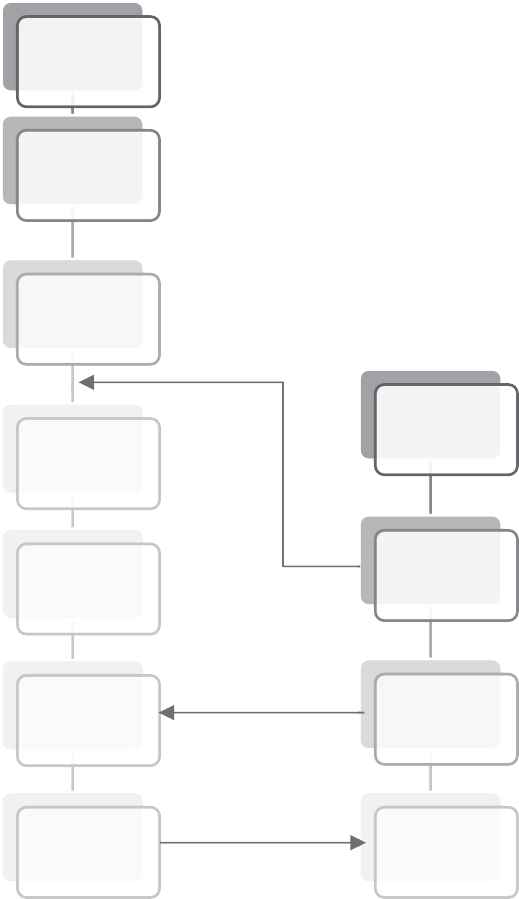
- nel caso il comando sia valido il server invia un messaggio di risposta al client notificando il successo dell’operazione richiesta.

# 2. Comunicazione Client-Server

# 2.1 Connessione iniziale

## Una volta avviato, il server crea una nuova socket, chiama la funzione *bind()* per farsi assegnare un indirizzo e si mette in attesa di richieste di connessione da parte dei client. Quando un nuovo client chiede di connettersi al server invia un messaggio di connessione al quale il server, se in quel momento è disponibile, risponde creando una connessione col client richiedente e si mette in attesa di un comando da eseguire.

**SERVER**



socket()

bind()

Wait for new client

**CLIENT**

socket()

socket()

Connect server

Find port

.

rdt\_send()

rdt\_rcv()

rdt\_send()

rdt\_rcv()

**Connessione lato Server**

Le operazioni eseguite dal server al momento dell’avvio del sistema sono:

1. Creazione di una nuova socket effettuando la chiamata *bind()* con l’opzione *INADDR\_ANY* per accettare pacchetti da una qualsiasi delle sue interfacce di rete;
2. Il server si mette in attesa di richieste di connessione da parte di un client sulla porta di default;
3. Appena arriva una nuova richiesta di connessione viene creata una nuova socket;
4. Il server cerca una porta libera alla quale far connettere il client chiamando la funzione della libreria *connection.h “find\_port()”;*
5. Invoca poi la chiamata alla funzione “*connect\_server”,* appartenente sempre al modulo *connection.h*, che si occupa di inviare al client i parametri per la comunicazione:
   * PORT\_NUMBER;
   * Dimensione della finestra W;
   * TIMEOUT;
   * Flag timeout adattivo ADAP;
6. Viene creato un nuovo processo figlio con la chiamata *fork()* il quale gestirà le richieste del client;
7. Il server si rimette in attesa di nuove connessioni da parte dei client.

# 2.2 Chiusura della connessione

**Connessione lato Client**

Le operazioni eseguite dal client al momento dell’avvio sono:

1. Creazione di una nuova socket;
2. Invocazione della funzione, dichiarata nella libreria *connection.h,* “*connect\_client”*, la quale ha il compito di contattare il server sulla porta di default e di ricevere i parametri della connessione;
3. Il client a questo punto è pronto a comunicare con il server, mostrando il menù di benvenuto con la funzione “*say\_hello”;*
4. Si mette in attesa di un comando di input da parte dell’utente.

### Il sistema può gestire tre modalità per la chiusura della connessione:

### Chiamata della funzione del modulo *connection.c* “close\_conn()” tramite l’input dell’utente tramite comando da tastiera (tasto ‘q’);

### Input da tastiera dell’utente tramite il comando di interrupt ‘Ctrl-C’;

### Terminazione per via di esaurimento del TIMEOUT, nessun comando per più di 3 minuti.

### 3. Trasferimento affidabile dei File

Il trasferimento dati sfrutta il protocollo con pipeline ***selective repeat***, realizzato su socket UDP, al fine di garantire una comunicazione affidabile che possa usare la banda disponibile anche in presenza di tempi di latenza non trascurabili.

La comunicazione è di tipo monodirezionale quindi si possono etichettare i due host come sender e come receiver.

I messaggi scambiati tra client e server sono incapsulati all’interno di una struttura *“message”* contenente un campo comando ed un campo payload.

Per inviare un messaggio è stata implementata la funzione *“rdt\_send”*, definita nel file header *reliable\_com.h*, la quale estrapola dalla struct *“message”* i dati da inviare trasformandoli in una stringa ‘comando + payload’.

Di contro, è stata sviluppata la funzione *“rdt\_rcv”,* definita anch’essa nel modulo *reliable\_com.h*, la quale partiziona i dati ricevuti per popolare la struttura *“message”* passata come input.

# Sender Selective Repeat

Per realizzare la logica del sender del protocollo Selective Repeat è stata implementata la funzione:

***int upload\_sender (int sd, struct sockaddr\_in addr, FILE \*fd, int N, int start\_timeout, int adapt, int dim);***

* *int sd:* socket descriptor;
* *struct sockaddr\_in addr:* indirizzo del receiver;
* *FILE \*fd:* file descriptor del file da inviare;
* *int N:* dimensione della finestra del sender;
* *int start\_timeout:* timeout iniziale;
* *int adapt:* flag del timeout adattivo;
* *int dim:* dimensione del file in byte;

**Funzionamento:**

Per implementare la logica del sender si è scelto di costruire il pacchetto di invio utilizzando la struttura *“message”,* assegnando al campo di comando il numero di sequenza del pacchetto e, al campo payload i dati letti dal file dato in input alla funzione.

Nella fase iniziale, si imposta il valore di *send\_base* a 0, e vengono inviati una serie di pacchetti con numero di sequenza nell’intervallo numerico [0, *send\_base + N*] finché non si raggiunge la fine del file da inviare oppure si arriva al numero massimo di pacchetti che possono essere contemporaneamente in volo.

Ogni volta che la funzione deve inviare un nuovo pacchetto, genera un nuovo thread al quale viene passata la struttura *“thread\_arg”,* definita nel modulo *types.h*, la quale contiene informazioni sull’indirizzo del receiver, il pacchetto realizzato con la struttura *“message”* ed il valore del timeout.

Utilizzando la funzione *“rdt\_send()”* ogni thread tenterà ripetutamente di inviare un pacchetto con uno sleep impostato al valore del timeout fra un invio e il successivo.

La funzione si mette in attesa di ricevere un riscontro dal receiver.

All’arrivo di un messaggio di *ACK* viene controllato se il *sequence\_number* riscontrato si trovi all’interno dell’intervallo [*send\_base, send\_base + N*].

Se il controllo ha avuto esito positivo, si invia un segnale di terminazione al thread associato al pacchetto col relativo *sequence\_number* riscontrato usando la chiamata di sistema *pthread\_cancel(tid).*

Se al contrario il *sequence\_number* riscontratocoincide con il valore posseduto da *send\_base,* viene fatta scorrere in avanti la finestra rendendo possibile l’invio di nuovi pacchetti.

**sender**

**receiver**

Nel momento in cui viene raggiunta la fine del file da inviare, si avvia

una fase di handshake di chiusura.



done,seq\_num done,null close,null

Se il trasferimento completo del file ha avuto successo la funzione

ritorna il valore 0 e viene mostrato un messaggio all’utente

di avvenuto upload del file.

Nel caso in cui la comunicazione con il destinatario venga interrotta

e non si riesca a far recapitare nessun pacchetto per troppo tempo, vengono

eliminati tutti i thread coinvolti nell’invio del file, la funzione ritorna il valore -1 e viene stampato a schermo un messaggio di errore nell’upload.

***int list\_sender (int sd, struct sockaddr\_in client, int N, int start\_timeout, int adpt, char \*path);***

* *int sd:* socket descriptor;
* *struct sockaddr\_in client:* indirizzo del client;
* *int N:* dimensione della finestra del sender;
* *int start\_timeout:* timeout iniziale;
* *int adapt:* flag del timeout adattivo;
* *char \*path:* puntatore al path della cartella dei file contenuti dal server;

**Funzionamento:**

La funzione ha il compito di inviare al client la lista dei nomi dei file presenti nella cartella dei file disponibili per il download, posseduti dal server.

La struttura è la stessa della funzione *“upload\_sender()”*  con l’unica differenza nel contenuto dei pacchetti inviati dai thread. Si è utilizzata anche in questo caso la struttura *“message”* per costruire la parte dati del pacchetto impostando come vuoto il campo di *comando* e inserendo nel campo *payload* il nome del file che il thread sta consegnando al client.

Per interagire con la cartella dei file del server si è fatto uso delle funzioni presenti nella libreria standard *<dirent.h>:*

* *opendir (const char \*name);* per l’apertura della directory;
* *readdir (DIR \*dirp);* per la lettura del contenuto della cartella;

# Receiver Selective Repeat

La logica del receiver è stata implementata nella funzione:

***int download\_rcv (int sd, struct sockaddr\_in addr, FILE \*fd, int N);***

* *int sd:* socket descriptor;
* *struct sockaddr\_in addr:* indirizzo da cui ricevere i dati;
* *FILE \*fd:* file descriptor del file da inviare;
* *int N:* dimensione della finestra del sender;

**Funzionamento:**

la funzione ha lo scopo di riassemblare ed ordinare i pacchetti inviati dalla funzione *“upload\_sender”.* Il risultato finale sarà il file inviato ricostruito nella sua interezza se la il client ha invocato il comando *put*, oppure la lista dei file presenti sul server se il client ha chiamato il comando *list.*

Alla ricezione di un nuovo messaggio da parte del sender, la funzione controlla il contenuto del campo *comando* del pacchetto ricevuto.

* Se è stata ricevuta la stringa ***err***: vuol dire che si è verificato un errore nell’invio del pacchetto, e la funzione quindi stampa a schermo il messaggio di errore presente nel campo *payload*.
* Se è stata ricevuta la stringa ***done****:* è un messaggio di segnalazione da parte del sender di aver raggiunto la fine del file (o la fine della lista dei file).

Altrimenti, se è presente un messaggio informativo, la funzione controlla il *sequence\_number.*

Se si trova all’interno dell’intervallo [*recv\_base, recv\_base + N*], memorizza il contenuto del campo *payload* in un buffer circolareed invia un messaggio di *ACK.*

Se il *sequence\_number,* si trova nell’intervallo [*recv\_base – N, recv\_base*) la funzione reinvia soltanto un messaggio di *ACK* poiché il pacchetto era già stato ricevuto.

Se il *sequence\_number* non rientra in nessuno dei due intervalli suddetti, il pacchetto viene scartato.

Nel momento i cui viene rilevato che il *sequence\_number* coincide con *recv\_base* viene fatta scorrere la finestra di ricezione, si scrivono tutti i blocchi consecutivi bufferizzati sul file, si libera il buffer circolare di ricezione e si incrementa *recv\_base* per far scorrere la finestra*.*

Se viene trovato un pacchetto con campo *payload* vuoto (pacchetto di chiusura), si invia un messaggio *done* e ci si mette in attesa del messaggio di *close* da parte del sender per terminare la comunicazione.

Come nell’implementazione del sender Selective Repeat, anche il receiver in caso di interruzione della comunicazione con il sender, o in caso non si riesca a ricevere alcun messaggio in troppo tempo, imposta come valore di ritorno -1 e il sistema stampa a schermo un messaggio di errore.

# Scelte progettuali

# 4.1 Server multiprocess

Per ottenere la concorrenza del server si è scelto di utilizzare un approccio multi-processo. Il processo padre si occupa di ricevere le richieste da parte dei client e generare per ognuna un nuovo processo figlio.

Ogni processo figlio ha il compito di gestire la comunicazione e l’interazione il client che ne ha fatto scaturire la sua generazione.

**Funzionamento del server:**

All’avvio, il processo principale crea un socket UDP, abilita l’opzione di riutilizzo dell’indirizzo e lo “binda” sulla porta nota (*PORT*), in modo da restare in ascolto di qualunque client che desideri avviare una comunicazione. A questo punto stampa su schermo un messaggio di avvio e si prepara a ricevere richieste.

Il cuore del server è un ciclo infinito in cui, usando la primitiva rdt\_rcv(), all’arrivo di nuovi pacchetti provenienti da client che hanno intenzione di connettersi, controlla se il contenuto del campo *comando* è "*conn*", in tal caso capisce che un nuovo client vuole instaurare una connessione affidabile. A questo punto, per non bloccare l’accettazione di ulteriori client durante il setup, viene creato una seconda socket UDP, su cui sarà instradata la vera comunicazione affidabile. Viene assegnata dinamicamente una porta libera (tramite la funzione find\_port), e se questa operazione fallisce – ad esempio perché il server è già saturo di connessioni – si risponde con un messaggio di errore e si abbandona il tentativo.

Quando la negoziazione avviene con successo – ossia il server e il client hanno scambiato i messaggi handshake necessari per accordarsi sui parametri di finestra (W), timeout (T) e timeout adattivo (ADAP) – il processo padre genera un nuovo figlio tramite fork(). Da quel momento, il processo figlio si occupa esclusivamente di dialogare con quel client: ascolta i comandi *(list, get, put, quit*) sul socket dedicato e li elabora invocando le funzioni list\_sender, getFunc o putFunc, tutte implementate a loro volta usando Selective Repeat per garantire l’affidabilità su UDP. Durante ciascuna operazione di trasferimento file, il figlio mantiene un timeout per rilevare l’inattività del client: se non arriva nulla per tre minuti, chiude la connessione e termina.

Nel frattempo, il processo padre, chiude la copia del socket ormai gestito dal figlio, torna subito ad ascoltare nuove richieste di connessione sul socket originale cosicché possa servire altri client senza ritardi. Quando il figlio riceve il comando "*quit*" o supera il timeout, esso effettua una chiusura ordinata del socket, stampa un messaggio di log e termina con exit(0), rilasciando tutte le risorse allocate.

**Vantaggi del server multiprocesso**

L’adozione di un modello multiprocesso porta diversi vantaggi in questo contesto. Innanzitutto, ogni client è isolato in un processo distinto: eventuali errori o crash nel gestore di uno non compromettono l’intera applicazione. Inoltre, su sistemi multicore, i processi figli possono essere schedulati in parallelo, aumentando il throughput complessivo del server e riducendo la latenza percepita dai singoli client. Dal punto di vista della gestione della memoria, ciascun processo figlio mantiene il proprio spazio, evitando la complessità di sincronizzazione tipica di un modello multithreaded. Infine, nel contesto di Selective Repeat, dove il controllo fine di timeout, buffering e ritrasmissioni è piuttosto articolato, disporre di un’istanza separata per ogni client semplifica enormemente la logica: non serve mantenere strutture dati condivise né preoccuparsi di lock o race conditions.

# 4.2 Timeout adattivo

Il protocollo *selective repeat*prevede che ogni pacchetto inviato abbia un proprio timeout, il quale una volta scaduto porti alla ritrasmissione del relativo pacchetto; per implementare un sistema che facesse questo è stata creata la funzione *“get\_timeout”* presente nel modulo *protocol.h*. La funzione realizza in poche righe il cuore del meccanismo di calcolo del timeout adattivo, ispirato al celebre algoritmo di Jacobson/Karels utilizzato in TCP.

**Funzionamento:**

Ogni volta che arriva una notifica di avvenuta ricezione (o di ritrasmissione) con un certo valore misurato di round‐trip time (RTT), la funzione aggiorna due stime interne:

– **stima media dell’RTT (*est\_rtt*):** viene applicata una media mobile esponenziale (EWMA), in cui il parametro ALPHA determina quanto peso dare all’ultima misura rispetto al valore già calcolato.

La formula:

**est\_rtt = (1 - ALPHA) \* est\_rtt + ALPHA \* RTT**

Essa riduce l’influenza di eventuali picchi dovuti a jitter, favorendo la stabilità della stima nel tempo.

– **deviazione standard approssimata dell’RTT (*dev\_rtt*)**: parallelo alla stima media, anche la variabilità dell’RTT viene monitorata con un EWMA sui valori assoluti delle differenze tra la misura attuale e la stima. Usando un coefficiente BETA, la formula:

**dev\_rtt = (1 - BETA) \* dev\_rtt + BETA \* | (RTT - est\_rtt) |**

mantiene una “memoria” della dispersione dell’RTT, fondamentale per adattare il timeout alle condizioni reali di rete.

A questo punto, il timeout vero e proprio viene calcolato come:

**Timeout = est\_rtt + 4 \* dev\_rtt**

ovvero si aggiunge alla stima media un margine pari a quattro volte la deviazione, così da coprire con alta probabilità i casi in cui la rete subisca ritardi imprevisti. In pratica, se la rete è instabile, *dev\_rtt* cresce e il timeout si allunga, mentre in condizioni stabili il margine si restringe, riducendo inutili attese prima di ritrasmettere un pacchetto.

I valori usati per i parametri sono:

**ALPHA =1/8;**

**BETA = 1/4;**

Questi parametri sono quelli usati di solito da TCP, ed essendo potenze di due rendono più rapide le operazioni di moltiplicazione che vengono implementate sotto forma di shift binario.

# 4.3 Configurazione parametri del sistema

Per la personalizzazione dei parametri utilizzati nella comunicazione Client-Server è stato creato un apposito file header di configurazione e dichiarazione delle strutture principali utilizzate dal sistema *types.h.*

Le costanti configurabili sono:

* *MAX:* Dimensione massima della PDU;
* *PROB*: Probabilità di perdita di un pacchetto;
* *ALPHA:* Costante per il calcolo di *est\_rtt;*
* *BETA:* Costante per il calcolo di *dev\_rtt;*
* *PORT:* Porta di default del server;
* *NUMPORT:* Numero massimo di connessioni concorrenti;
* *W:* Dimensione della finestra di spedizione/ricezione;
* *T:* Timeout in microsecondi;
* *ADAP:* Flag per il timeout adattivo;

Il sistema permette anche la modifica del Path della cartella dei file locale del client e del server. Per farlo bisognerà modificare la costante *PATH* definita rispettivamente in *“client.c”* e in *“server.c”*.

# 4.4 Struttura del codice

Nel nostro progetto il sorgente è suddiviso in dieci moduli principali, ognuno corredato dal proprio header per isolare responsabilità e interfacce:

* ***Makefile***: orchestratore della build, definisce le regole di compilazione e gli obiettivi per generare gli eseguibili client e server, gestendo dipendenze e flags del compilatore.
* ***connection.c*** / ***connection.h***: si occupa della selezione dinamica di porte libere e del three-way iniziale per la connessione client-server e la negoziazione dei parametri di connessione quali la dimensione della finestra e timeout tra client e server.
* ***debug.c*** / ***debug.h***: fornisce funzioni di debug e di test di funzionamento e valutazione delle prestazioni dei vari comandi gestiti dall’applicazione.
* ***file\_ops.c*** / ***file\_ops.h***: implementa le operazioni di I/O su file, tra cui la lista dei file a disposizione (*list\_sender*), l’invio (*getFunc*) e la ricezione (*putFunc*) di singoli file, interfacciandosi con il protocollo di trasferimento.
* ***protocol.c*** / ***protocol.h***: contiene le strutture dati e i comandi di alto livello del protocollo come il calcolo del timeout adattivo e la funzione di inoltro ripetuto di pacchetti da parte dei thread.
* ***reliable\_com.c*** / ***reliable\_com.h***: realizza le primitive *rdt\_send()* e *rdt\_rcv()* che garantiscono la consegna affidabile su UDP, con gestione di ritrasmissioni, ACK e timeout.
* ***utils.c*** / ***utils.h***: raccoglie piccole utility trasversali, ad esempio conversioni di rete <-> host order, funzioni di supporto per la gestione di stringhe e buffer, e stampe a schermo di messaggi si successo o di errore.
* ***client.c***: programma applicativo lato client, che presenta un’interfaccia testuale all’utente, invia i comandi (*list*, *get*, *put*, *quit*) e invoca le API del modulo *connection*.*h* e *reliable\_com.h* per interagire col server.
* ***server.c***: demone UDP multiprocesso che, dopo il *bind()* sulla porta nota, riceve richieste di connessione genera processi figli per ogni client e utilizza *file\_ops.h* e *reliable\_com.h* per gestire in parallelo i trasferimenti.

La scelta di modulare il codice porta a numerosi vantaggi in termini di manutenibilità, riusabilità e chiarezza architetturale. Innanzitutto, ogni componente ha una singola responsabilità ben definita che semplifica la comprensione e il debug. La separazione delle interfacce in header dedicati consente una compilazione incrementale: cambiando un modulo non è necessario ricompilare l’intero progetto, riducendo i tempi di build. Inoltre, una struttura modulare favorisce il testing unitario, ogni modulo può essere isolato e validato in modo indipendente, e agevola l’estensione futura.

### 4.5 Gestione file

La cartella del server contiene file comuni accessibili ad ogni client, dato che non vi è nessuna forma di autenticazione (come da direttive).

I client su host differenti avranno invece cartelle contenenti diversi file.

# Installazione ed esempi di utilizzo

# 5.1 Installazione

Per installare ed avviare l’applicazione è necessario:

1. Accedere alla cartella *“RFT”;*
2. Aprire la shell di comando;
3. Compilare l’applicazione usando il comando *“make”* oppure *“make debug”* se si intende testare l’efficienza del sistema;
4. Avviare il server con il comando da shell *“./server”;*
5. Avviare il client con il comando da shell *“./client”;*

# 5.2 Interfaccia utente di avvio

Dopo aver avviato sia il server che il client il sistema mostra all’utente la seguente interfaccia di menù:

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Nel menù principale oltre ai comandi principali (*list, get, put*), sono mostrati anche le richieste:

* *ls:* visualizza la lista dei file disponibili per l’upload;
* *help:* visualizza nuovamente la schermata home di menù;
* *q:* comando per uscire dal programma.

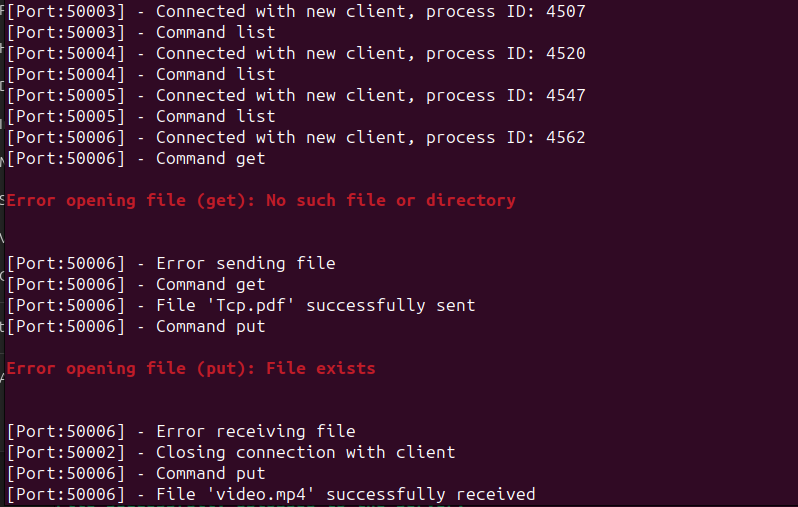
Eseguendo, invece, in ***modalità debug***, verrà mostrata la schermata:

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

In questo menù di interfaccia utente è presente anche il comando *“test”,* il quale se invocato permette di effettuare una prova e test di velocità delle funzioni *list,* *get* e *put.* Per eseguirlo è necessario passare il nome del file su cui si intende effettuare il test. La funzione poi chiederà all’utente il numero di ripetizioni del test.

# 5.3 Interfaccia del server

****

Durante l’esecuzione del sistema il server mostra tutte le interazioni che avvengono con i vari client che si connettono riportando anche i comandi che ognuno da in input e l’esito delle relative richieste effettuate.

# 5.4 Esempi di utilizzo

* Comando ***list***

**Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.**

Una volta che l’utente ha dato in input il sistema tramite scrittura da tastiera il comando *“list”,* verrà mostrata su terminale la lista dei file presenti nella cartella del server disponibili per il download all’interno della cartella del client.

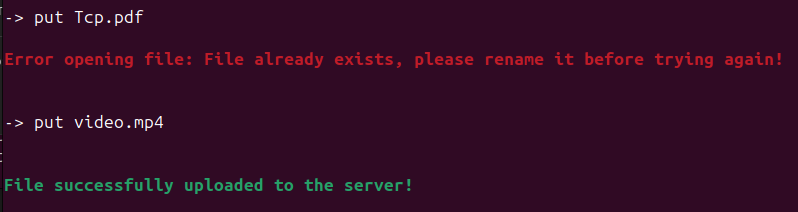
* Comando ***get***

***Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.***

Se l’utente dà in input al sistema il comando *“get”* seguito dal nome del file che si intende scaricare dalla cartella del server a quella del client, l’applicazione mostrerà a schermo l’esito del download.

* Comando ***put***

******

Se invece si ha intenzione di caricare nella cartella del server un file presente nel folder locale del client, l’utente dovrà digitare da tastiera il comando ***“put”*** seguito dal nome del file. Per visualizzare i file pronti per l’upload basterà chiamare la funzione ***“ls”*** la quale mostrerà a schermo tutti i file presenti nella cartella del client.

Al termine dell’operazione di upload il sistema mostrerà sul terminale un messaggio di esito della transazione.

# Prestazioni del sistema

Al fine di simulare un reale impiego del sistema ed avere valori affidabili della misurazione delle prestazioni sono stati effettuati test con più client che inoltrano la medesima richiesta su file differenti cosicché si riesca a valutare la risposta del server nel caso di più richieste in contemporanea.

I test sono effettuati in locale il che comporta una probabilità di perdita esterna all’applicazione praticamente nulla oltre ad un RTT molto basso.

Il file utilizzato nelle misurazioni è di dimensione 1Mb.

I test sono stati eseguiti su una macchina dotata di processore AMD Ryzen 5 3500U con Sistema Operativo Ubuntu-18.04.

# 6.1 Simulazione *List*

* Test 1 effettuato facendo variare la finestra e mantenendo fisso il valore della probabilità di perdita dei pacchetti e impostando il timeout fisso a 10 secondi.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test 1 | N = 1 | N = 3 | N = 5 | N = 10 |
| P = 0.125  T = 10000µs | 171503,6µs | 135121,6µs | 112466,5µs | 55554,88µs |

* Test 2 effettuato facendo variare il valore della probabilità di perdita di un pacchetto mantenendo costante il numero della finestra e il impostando il timeout fisso a 1 secondo.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test 2 | P = 0 | P = 0,125 | P = 0,25 | P = 0,5 |
| N = 10  T = 10000µs | 15152,99µs | 55554,88µs | 207371,1µs | 437302,2µs |

### Test 3 effettuato facendo variare il tempo del timeout fisso e mantenendo costanti la grandezza della finestra e la probabilità di perdita di un pacchetto.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test 3 | T = 10000µs | T = 20000µs | T = 30000µs |  |
| N = 10  P = 0.125 | 55554,88µs | 185663,2µs | 238771,4µs |

# Test 4 effettuato impostando il tempo di timeout a adattivo e studiando il comportamento del sistema mantenendo costante la grandezza della finestra e la probabilità di perdita di un pacchetto. I parametri “*T*” riportati in tabella sono i valori dei timeout iniziali scelti per il test.

|  |  |  |  |
| --- | --- | --- | --- |
| Test 4 | T = 10000µs | T = 20000µs | T = 30000µs |
| N = 10  P = 0.125 | 86959,31µs | 199802,2µs | 311502,3µs |

# 6.2 Simulazione *Get*

* Test 1 effettuato facendo variare la grandezza della finestra di spedizione/ricezione e mantenendo fisso il valore del timeout impostato a 10sec e il valore della probabilità di perdita di un pacchetto.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test 1 | N = 1 | N = 3 | N = 5 | N = 10 | N = 20 |
| P = 0.125  T = 10000µs | 1168714µs | 831703,2µs | 670828,6µs | 579793,9µs | 339906,5µs |

* Test 2 effettuato facendo variare il valore della probabilità di perdita di un pacchetto mantenendo costante il numero della finestra e il impostando il timeout fisso a 1 secondo.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test 2 | P = 0 | P = 0.125 | P = 0.25 | P = 0.5 |  |
| N = 10  T = 10000µs | 110879,3µs | 579793,9µs | 977265µs | 2159163µs |

### Test 3 effettuato facendo variare il tempo del timeout fisso e mantenendo costanti la grandezza della finestra e la probabilità di perdita di un pacchetto.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test 3 | T = 10000µs | T = 20000µs | T = 30000µs |  |
| N = 10  P = 0.125 | 579793,9µs | 889450,9µs | 1265725µs |

# Test 4 effettuato impostando il tempo di timeout a adattivo e studiando il comportamento del sistema mantenendo costante la grandezza della finestra e la probabilità di perdita di un pacchetto. I parametri “*T*” riportati in tabella sono i valori dei timeout iniziali scelti per il test.

|  |  |  |  |
| --- | --- | --- | --- |
| Test 4 | T = 10000µs | T = 20000µs | T = 30000µs |
| N = 10  P = 0.125 | 475458,1µs | 464810,1µs | 479795,5µs |

# 6.3 Simulazione *Put*

* Test 1 effettuato facendo variare la finestra e mantenendo fisso il valore della probabilità di perdita dei pacchetti e impostando il timeout fisso a 1 secondo.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test 1 | N = 1 | N = 3 | N = 5 | N = 10 | N = 20 |
| P = 0.125  T = 10000µs | 1189365µs | 842627,9µs | 690906,3µs | 496387µs | 369610,2µs |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test 2 | P = 0 | P = 0.125 | P = 0.25 | P = 0.5 |  |
| N = 10  T = 10000µs | 160262,4µs | 496387,4µs | 828637,9µs | 2318396µs |

* Test 2 effettuato facendo variare il valore della probabilità di perdita di un pacchetto mantenendo costante il numero della finestra e il impostando il timeout fisso a 1 secondo.

### Test 3 effettuato facendo variare il tempo del timeout fisso e mantenendo costanti la grandezza della finestra e la probabilità di perdita di un pacchetto.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test 3 | T = 10000µs | T = 20000µs | T = 30000µs |  |
| N = 10  P = 0.125 | 496387,4µs | 966866,8µs | 1487836µs |

* Test 4 effettuato impostando il tempo di timeout a adattivo e studiando il comportamento del sistema mantenendo costante la grandezza della finestra e la probabilità di perdita di un pacchetto. I parametri “T” riportati in tabella sono i valori dei timeout iniziali scelti per il test.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test 4 | T = 10000µs | T = 20000µs | T = 30000µs |  |
| N = 10  P = 0.125 | 438179,5µs | 496912,7µs | 454558,9µs |

## **6.4 Osservazioni generali simulazioni**

## Dalle misurazioni condotte emerge con chiarezza che, per tutti e tre i comandi (*list, get, put*), l’aumento della dimensione della finestra (*N*) comporta un miglioramento lineare del throughput fino a un certo punto (in particolare da *N = 1* a *N = 10*), dopodiché l’effetto benefico si attenua a causa del sovraccarico di gestione dei buffer. Al crescere della probabilità di perdita (*P*), i tempi di esecuzione aumentano in modo esponenziale, evidenziando l’importanza di dimensionare correttamente la grandezza della finestra in funzione della qualità di rete. Infine, l’adozione del timeout adattivo riduce drasticamente il numero di ritrasmissioni non necessarie in ambienti con jitter elevato, rendendo i trasferimenti più stabili e prevedibili rispetto a un timeout fisso.

# 7. Manuale d’uso

# 7.1 Configurazione

All’utente è data la possibilità di cambiare la posizione e il nome delle directory di default per i file del server e del client (quelle di default si chiamano “client\_src/files/” e “server\_src/files/”).

Per farlo dovrà recarsi all’interno del modulo *“client.c”* e *“server.c”* rispettivamente,nei quali troverà definita la variabile *PATH.*  Modificando la stringa assegnata alla variabile si ridefinirà il percorso file delle cartelle locali di client e server rispettivamente.

La modifica dei campi richiede il riavvio dell’applicazione e ricompilazione.

#### Server

Aprendo con un editor di testo il file *“types.h”* verranno visualizzati i seguenti campi relativi alla configurazione del server:

* *PORT = '50000'* questo campo identifica la porta pubblica utilizzata dal processo padre del server.

Questo valore deve essere una porta non utilizzata da applicazioni che condividano lo

stesso indirizzo IP dell’applicazione (in genere compresa tra 1024 e 65535).

Questo parametro è condiviso col processo client in quanto le due porte devono coincidere.

* *NUMPORT = '15000'* questo campo identifica il numero massimo di connessioni concorrenti gestite dal server.
* *W = '5'* questo campo identifica la dimensione in pacchetti della finestra di spedizione/ricezione.

Deve essere un valore intero positivo.

* *T = '10000'* questo campo identifica il timeout di ritrasmissione di un pacchetto (espresso in microsecondi) in caso di timeout fisso, o il valore iniziale del timeout in caso di timeout adattativo.
* *ADAP = '1'* valore flag che stabilisce la tipologia di timeout utilizzato e può assumere solo i seguenti valori:

“0” timeout fisso

“1” timeout adattativo

* *PROB = '0.125'* questo campo identifica la probabilità di perdita di ogni pacchetto (espressa in percentuale).

Può assumere un valore tra 0 e 100.

#### Client

Se l’utente volesse mutare i parametri impostati di default per il client, si dovrà recare nel modulo *“client.c”,* nel quale potrà modificare i seguenti valori:

* *IP = '127.0.0.1'* questo campo identifica l’indirizzo di rete del server a cui il client invia la richiesta. In questo caso è stato impostato di default l’indirizzo di *local-host*.
* *PATH = 'client\_src/files/'* questo campo identifica il percorso file della cartella locale del client.

## **Compilazione ed Esecuzione**

1. Recarsi nella cartella download e decomprimere e spostare nella directory che si preferisce il file ZIP *“RFT.zip”.*
2. Una volta che ci si trova all’interno della cartella decompressa lanciare il terminale.
3. Digitare il comando di compilazione *“make”.*
4. Avviare il server lanciando il comando *“./server”.*
5. Ritornare nuovamente all’interno della cartella che *RFT* e lanciare un nuovo terminale per il client.
6. Avviare il client con il comando *“./client”*.
7. Lanciare i comandi desiderati dal client:

* *list:* visualizza la lista dei nomi dei file presenti nella cartella locale del server;
* *get<file\_name>: avvia* il download del file specificato dalla directory del server;
* *put<file\_name>:* avvia l’upload del file specificato sulla directory del server;
* *ls:* visualizza la lista dei file presenti nella cartella del client;
* *help:* visualizza di nuovo menù di avvio;
* *q:* termina la comunicazione col server.

Si ha anche la possibilità di forzare la chiusura sia del client che del server lanciando il comando di interrupt da terminale *“CTRL+C”.*

# Conclusioni

In conclusione, il progetto ha dimostrato con successo la fattibilità di un’applicazione client‑server basata su UDP in cui l’affidabilità dei trasferimenti file è garantita attraverso l’implementazione del protocollo Selective Repeat. L’architettura modulare e multiprocesso adottata ha permesso di isolare chiaramente le responsabilità – dall’inizializzazione delle connessioni e del three-way handshake, fino alle primitive *rdt\_send()/rdt\_rcv*() e alle operazioni di I/O sui file – agevolando la manutenzione, il testing e la futura estendibilità del codice. I test di performance hanno evidenziato come l’aumento della dimensione della finestra di trasmissione riduca sensibilmente i tempi di trasferimento, nei limiti imposti dal ritardo di rete e dalla probabilità di perdita, confermando l’efficacia del timeout adattivo basato su stima RTT e deviazione. Infine, l’approccio multiprocesso ha garantito un’elevata robustezza e scalabilità, sfruttando al meglio le risorse di sistema e isolando eventuali fault su singole connessioni.

**Possibili sviluppi futuri** includono l’integrazione di meccanismi di autenticazione e cifratura per la sicurezza dei dati, l’introduzione di checksum nei messaggi per ulteriore tutela dell’integrità, e la creazione di un’interfaccia grafica che renda l’applicazione più fruibile anche a utenti non esperti.