Relazione progetto B IIW – AA 2016/2017

Membri del gruppo:

- Milia Cristian 0217898
- De Turris Daniele 0218324
- Dello Vicario Mario 0215190

Indice:

Architettura del server

Il server è organizzato seguendo le linee guida del preforking, ed è organizzato in 3 componenti fondamentali:

- main process
- pool_handler thread
- pool di processi

Il main process è il processo incaricato di avviare il server e tutte le sue componenti.
Una volta completato ciò si occuperà unicamente di accettare le richieste sulla porta di ascolto di default.
Ad ogni richiesta aggiunge un messaggio in una coda contenente le informazioni relative al client che l'ha contattato, la stessa coda su cui sono in ascolto i processi del pool.

Il pool_handler è un thread, creato dal main process in fase di bootstrap, incaricato di mantenere in uno stato coerente il pool di processi, e di occuparsi della loro morte.
In particolare, in modo ciclico, controlla in modo esclusivo (mtx_prefork) quale sia il numero di processi liberi e disponibili per una nuova richiesta, ed in caso sia inferiore al numero designato (impostato a 4), ne crea di nuovi; una volta completato ciò esegue una waitpid() non bloccante, di

modo da non lasciare traccia di eventuali processi del pool morti, e esonerando da questo lavoro il main process.

Il pool, è un gruppo di processi, che vanno ad implementare effettivamente il preforking.
Invece che optare per un preforking fortemente controllato, con un thread libero di uccidere un processo (con problemi relativi allo stato, se impegnato o meno), abbiamo deciso di orientarci verso una soluzione più flessibile, basata sull'autogestione del singolo processo, il pool_handler è incaricato solo di crearne di nuovi in caso non ce ne siano a sufficienza.
Ogni singolo processo esegue il child_job()),che può essere schematizzato come:

-controllo se ci sono già abbastanza processi liberi, e nel caso mi "suicido"
-in modo esclusivo leggerò dalla coda di messaggi condivisa con il main process la prima richiesta
-soddisfo la richiesta
-in caso abbia già soddisfatto troppe richieste mi "suicido"

Tutto ciò viene eseguito in modo ciclico da ogni processo incaricato di gestire richieste.

```
void child_job() {
    char done_jobs=0;
    unlock_signal(SIGALRM);
    if(close(main_sockfd)==-1){//chiudi il socket del padre
        handle_error_with_exit("error in close socket fd\n");
    }
    for(;;){
        lock_sem(&(mtx_prefork->sem));//semaforo numero processi
        if(mtx_prefork->free_process>=NUM_FREE_PROCESS){
            unlock_sem(&(mtx_prefork->sem));
            exit(EXIT_SUCCESS);
        }
        mtx_prefork->free_process+=1;
        unlock_sem(&(mtx_prefork->sem));
        lock_sem(mtx_queue_child);
        value=(int)msgrcv(msgid,&request,sizeof(struct msgbuf)-sizeof(long),0,0);
        unlock_sem(mtx_queue_child);//non è un problema prendere il mutex e bloccarsi in coda
        lock_sem(&(mtx_prefork->sem));
        mtx_prefork->free_process-=1;
        unlock_sem(&(mtx_prefork->sem));
        reply_to_syn_and_execute_command(request,mtx_file);//soddisfa la richiesta
        done_jobs++;//incrementa numero di lavori svolti
        if(done_jobs>MAX_PROC_JOB){
            exit(EXIT_SUCCESS);
        }
    }
    return;
}
```

Per quanto riguarda la gestione delle socket, mentre il main process esegue la bind in fase di bootstrap, e rimane sempre in ascolto sulla porta di default, i processi del pool si comportano in maniera ben diversa:
innanzitutto, non esegue la bind su una porta prestabilita, ma su di una assegnata dal S.O., inoltre questa viene creata alla gestione di una richiesta, e liberata alla fine.
In questo modo, cambiando continuamente il numero di porta, possiamo ridurre al minimo i messaggi che un client (che magari ha perso un fin_ack e che quindi continua a mandare fin in continuazione) possa interferire con la gestione di richiesta completamente nuova da parte di un altro client. [file Server.c]

Architettura del Client

Il Client è organizzato sulla base del modello “fork”. All’avvio il main process è incaricato della creazione del thread waitpid(), per poi entrare in uno stato ciclico nel quale legge i comandi digitati dall’utente.
In particolare per ogni comando digitato, in caso questo sia valido, il processo effettuerà una fork() creando un processo figlio incaricato di soddisfare la richiesta corrispondente al comando.
L’azione spiegata è illustrata nel codice seguente, nel quale “command” corrisponde alla prima parola digitata, mentre “filename” alla seconda.

```
} else if (!is_blank(filename) && strncmp(command,"get",3) == 0) {
    //fai richiesta al server del file
    if ((pid = fork()) == -1) {
        handle_error_with_exit("error in fork\n");
    }
    if (pid == 0) {
        client_get_job(filename,mtx_file);
    }
} else if (strncmp(command,"list",4) == 0) {//list
    if ((pid = fork()) == -1) {
        handle_error_with_exit("error in fork\n");
    }
    if (pid == 0) {
        client_list_job();
    }
}
```

Il processo appena creato si occuperà della creazione di una socket e della successiva connessione al Server.
La connessione prevede un massimo di 10 tentativi (oltre i quali si suppone che il Server sia irraggiungibile) nei quali il Client invia periodicamente il messaggio di SYN.
In caso di risposta molto ritardata da parte del Server, potrebbe capitare la ricezione di due o più messaggi SYN_ACK, dei quali soltanto il primo verrà preso in considerazione (salvataggio del numero di porta del processo del Server incaricato della propria richiesta) , mentre gli altri verranno ignorati.
A seguito la richiesta verrà soddisfatta (con successo o meno) portando alla morte del processo stesso.
Successivamente il thread waitpid (il cui codice è riportato di seguito) si occuperà di liberare le risorse dedicate al processo. [file Client.c]

```
void *thread_job(void *arg) {//thread che esegue waitpid dei processi del client
    (void) arg;

    block_signal(SIGALRM);
    while (1) {
        while ((waitpid(-1, NULL,WNOHANG)) > 0);
        pause();
    }
    return NULL;
}
```

Comunicazione affidabile tra host su UDP

Due host,dopo aver stabilito una semplice connessione con segmenti di syn e syn_ack,per scambiarsi messaggi in maniera affidabile, hanno bisogno di ritrasmettere i pacchetti che vengono persi nella rete.
Per questo,per comunicare in maniera affidabile tramite ritrasmissioni, si fa uso di un thread ritrasmettitore. Nel programma,infatti,vi sono due thread dedicati esclusivamente alla trasmissione:uno per l'invio dei pacchetti e la ricezione degli ack;l'altro per la gestione delle ritrasmissioni. Entrambi i threads condividono una memoria condivisa e una lista dinamica spesso ad accesso esclusivo per mezzo di mutex; la memoria condivisa contiene tutte le variabili per implementare il protocollo selective_repeat,la lista dinamica contiene informazioni sui pacchetti.
I due threads hanno responsabilità ben diverse tra loro. Il primo thread,colui che invia pacchetti e riceve ack, ha il compito di inserire nella lista dinamica il pacchetto che ha appena inviato,aggiornare le variabili della finestra di trasmissione e aggiornare le variabili della finestra di ricezione quando riceve un ack. Il secondo thread,invece ,ha il compito di andare alla ricerca nella lista dinamica di pacchetti non ancora riscontrati,verificare se è scaduto il timer ,per poi eventualmente ritrasmetterli;se esiste un timer prossimo alla scadenza ,di un pacchetto non ancora riscontrato ,il thread dorme per il tempo restante e verifica subito dopo se il pacchetto è da ritrasmettere,dopodiché passa al pacchetto successivo. [file communication.c]

Lista dinamica

La lista dinamica è fondamentale per implementare le ritrasmissioni,infatti,ogni nodo della lista contiene le informazioni di ogni pacchetto inviato:numero di sequenza,timer che indica quando il pacchetto scadrà,puntatore al nodo precedente e puntatore al nodo successivo.
La lista ha una caratteristica molto importante,essa è ordinata in maniera crescente rispetto alla scadenza dei timer dei pacchetti. Questa proprietà implica che ,per sapere se un pacchetto è da ritrasmettere,non è necessario scansionare ogni volta tutta la lista,ma basta prendere in considerazione solo il primo nodo.
L'ordinamento viene garantito ad ogni inserimento di un nodo,quindi la lista in ogni istante risulta essere ordinata. L'ordinamento ,inoltre,è facilitato da un fatto molto interessante:inviando due pacchetti ,è molto probabile che il secondo scadrà dopo il primo,perciò si è pensato bene di ordinare la lista iniziando la scansione dalla coda,facendo si che,soprattutto nel caso di timer statico,l'algoritmo di ordinamento risulta molto efficiente.
Il numero massimo di nodi della lista non è definibile,ma fortunatamente i thread inseriscono e eliminano i nodi da essa in maniera abbastanza alternata,in modo tale da bilanciarne la lunghezza. [file dynamic_list.c]

Timer adattivo

Per il calcolo del nuovo valore di timeout,in modalità timer adaptive, si è fatto uso di due variabili: Sample_RTT e estimated_RTT. Il sample_RTT viene calcolato dall'istante di invio (o ritrasmissione) di un pacchetto all'istante di ricezione del relativo ack. L'estimated_RTT,invece,viene calcolato come media mobile esponenziale ponderata dei valori di sample_RTT.

estimated_RTT = (0,875)*estimated_RTT + (0,125)*sample_RTT

Per arrivare all'effettivo valore di timeout si moltiplica per due il valore di estimated_RTT,questo per lasciare timeout ad un valore più grande del valore stimato evitando così ritrasmissioni inutili. [file timer.c]

Timeout = 2*estimated_RTT

Selective repeat

Per implementare l'algoritmo di selective repeat sono state definite 3 strutture:
una struttura che definisce il formato di un elemento della finestra di ricezione,una struttura che definisce il formato di un elemento della finestra di trasmissione e una memoria condivisa,utilizzata dai due thread, che contiene tutte le variabili necessarie a soddisfare le richieste client-server.

Campi principali finestra ricezione,finestra trasmissione e memoria condivisa:

```
struct window_rcv_buf{//elemento della finestra di ricezione
	char received;//ricevuto 1==si 0==no
char*payload;//dati pacchetto
};
struct window_snd_buf{//elemento della finestra di trasmissione
	char acked;//riscontrato 1==si 0==da riscontrare 2==vuoto
	char*payload;//dati pacchetto
};
struct shm_sel_repeat{
	struct addr addr;//indirizzo dell'host
	int pkt_fly;//numero pacchetti in volo (va da 0 a w-1)
	pthread_mutex_t mtx;//mutex
	struct window_snd_buf *win_buf_snd;//finestra trasmissione (va da 0 a 2w-1)
	struct window_rcv_buf *win_buf_rcv;//finestra ricezione (va da 0 a 2w-1)
	int dimension;//dimensione del file/list
	int window_base_rcv;//sequenza di inizio finestra ricezione(va da 0 a 2w-1)
	int window_base_snd;sequenza di inizio finestra trasmissione(va da 0 a 2w-1)
	int seq_to_send;//prossimo numero di sequenza da dare al pacchetto (va da 0 a 2w-1)
	int byte_readed;//byte letti e riscontrati lato sender
	int byte_written;//byte scritti e riscontrati lato receiver
	int byte_sent;//byte inviati con pacchetti senza contare quelli di ritrasmissione
	int fd;//file descriptor
	struct node* head;//puntatore alla testa della lista dinamica
	struct node *tail;//puntatore alla coda della lista dinamica
};
```

Come si può vedere dalla funzione "send_message_in_window",un qualsiasi messaggio inviato con protocollo selective repeat viene memorizzato in finestra,inserisce nella lista dinamica le informazioni del pacchetto e aggiorna le variabili seq_to_send,pkt_fly e byte sent.

Quando si riceve un ack in finestra,invece, viene chiamata la funzione "rcv_ack_in_window" che segna messaggio come riscontrato prova a vedere se l'ack ricevuto è uguale all'indice di inizio finestra (window_base_snd) e in tal caso fa sliding window aggiornando byte

riscontrati,window_base_snd e diminuendo pacchetti in volo(che non può mai essere maggiore di Window).

Similmente,quando si riceve un messaggio in finestra ,viene chiamata la funzione rcv_msg_in_window che ,dopo aver mandato il relativo ack,segna messaggio come ricevuto e prova a vedere se il messaggio ricevuto è uguale all'indice di inizio finestra (window_base_rcv) ,in tal caso fa sliding window aggiornando byte ricevuti e window_base_rcv.

In realtà in base al tipo di messaggio inviato e in base al tipo di messaggio ricevuto (pacchetto contenente file,pacchetto contenente lista o messaggio) viene chiamata una funzione specifica che gestisca il caso. [file communication.c]

Macchine a stati

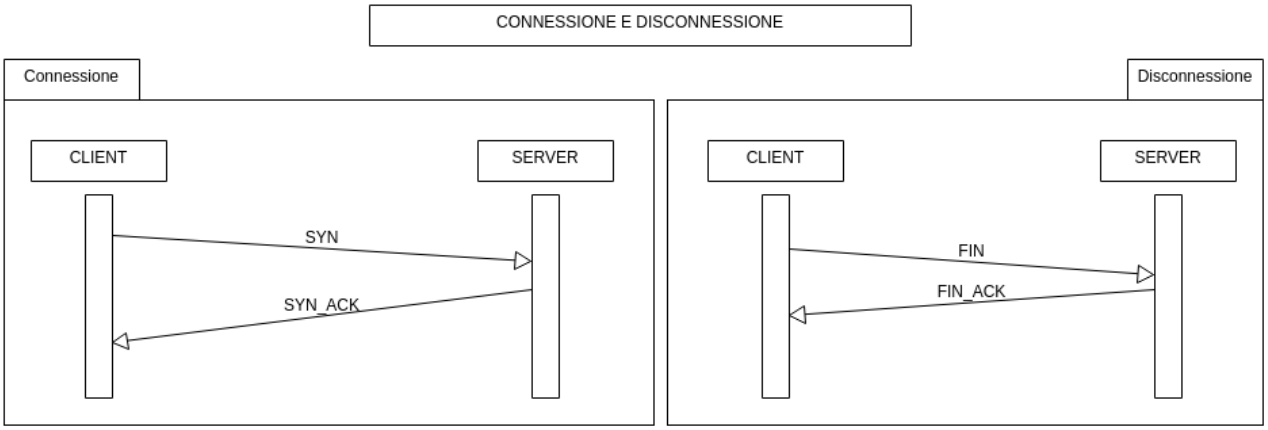
Logicamente ogni richiesta client-server può essere suddivisa nelle fasi di inizio,trasferimento dati e fine.Per suddividere le richieste nelle 3 fasi sono state impiegate macchine a stati. Il nome dello stato coincide con il nome della funzione in esecuzione e le azioni intraprese in uno stato sono l'invio di pacchetti ,che possono essere di tipo diverso da stato a stato, e la ricezione di particolari pacchetti che fanno transire in un nuovo stato.

Un esempio è riportato a seguire.

```
int rcv_list(struct temp_buffer temp_buff,struct shm_sel_repeat *shm) {
    alarm(TIMEOUT);
    send_message_in_window(temp_buff,shm, START,"START" );
    while (1) {
        if (recvfrom(shm->addr.sockfd, &temp_buff,...) !=-1) {//bloccati finquando non ricevi file
            // o altri messaggi
            if (temp_buff.command == SYN || temp_buff.command == SYN_ACK) {
                continue;//ignora pacchetto
            } else {
                alarm(0);
            }
            if (temp_buff.seq == NOT_A_PKT && temp_buff.ack != NOT_AN_ACK) {//se è un ack
                ...
                alarm(TIMEOUT);
            }
            ...
            }else if (seq_is_in_window(shm->window_base_rcv, shm->param.window, temp_buff.seq)) {
                //se non è un ack e è in finestra
                ...
                alarm(TIMEOUT);
            }
        }
        if (great_alarm_client == 1) {
            printf(RED"Server is not available,request list\n"RESET);
            ...
            alarm(0);
            pthread_exit(NULL);
        }
    }
}
```

Gestione della connessione

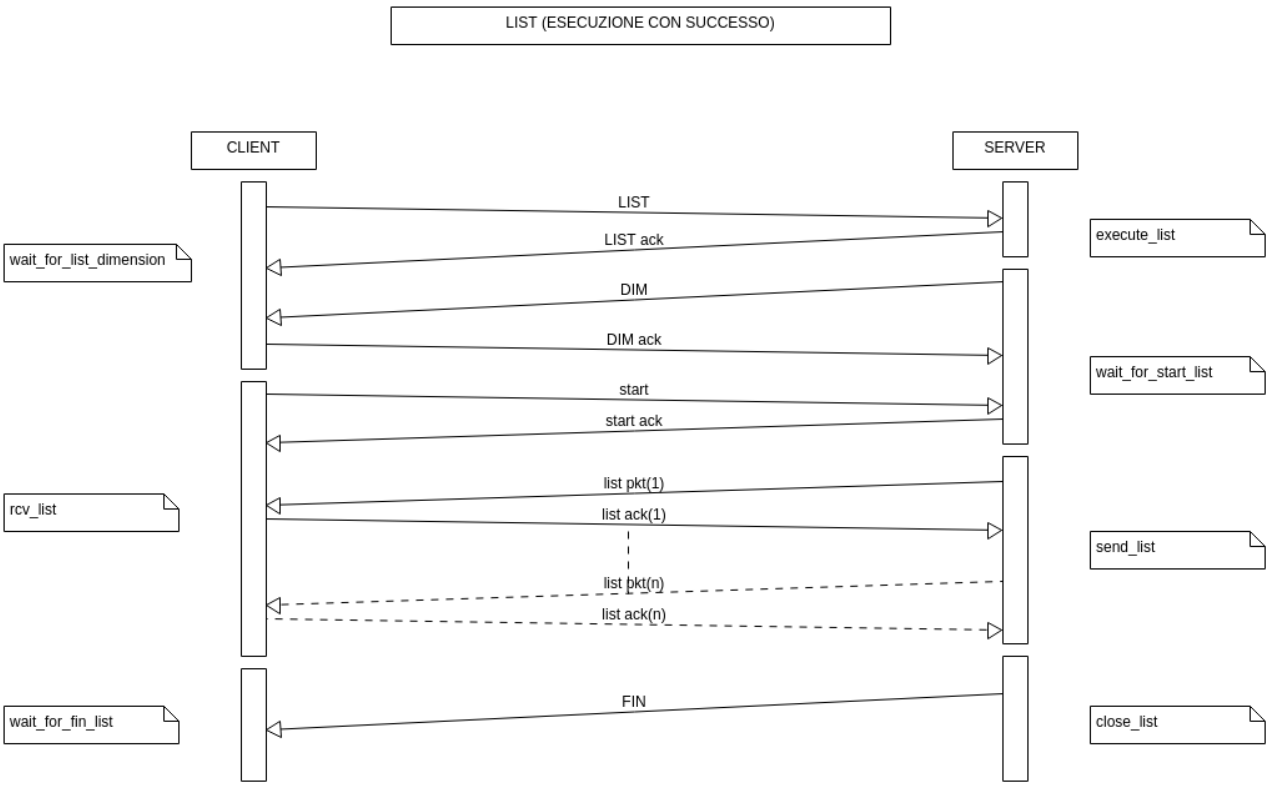
Nell' immagine che segue viene riportato un esempio di connessione tra Client e Server, in modo tale da mostrare al lettore l'ordine cronologico dei messaggi che vengono scambiati per stabilire la connessione.
In particolare ricordiamo che la connessione nello specifico avviene tra il Client ed un processo figlio del server che accoglie la richiesta.

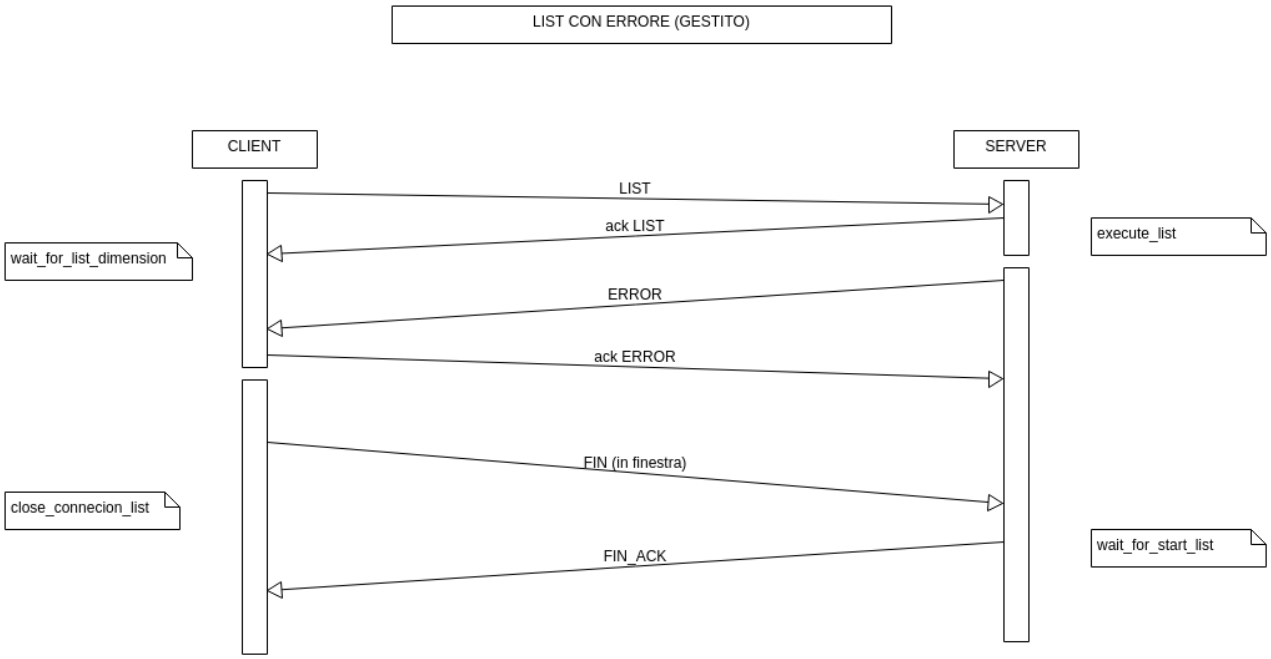


Schemi di esecuzione

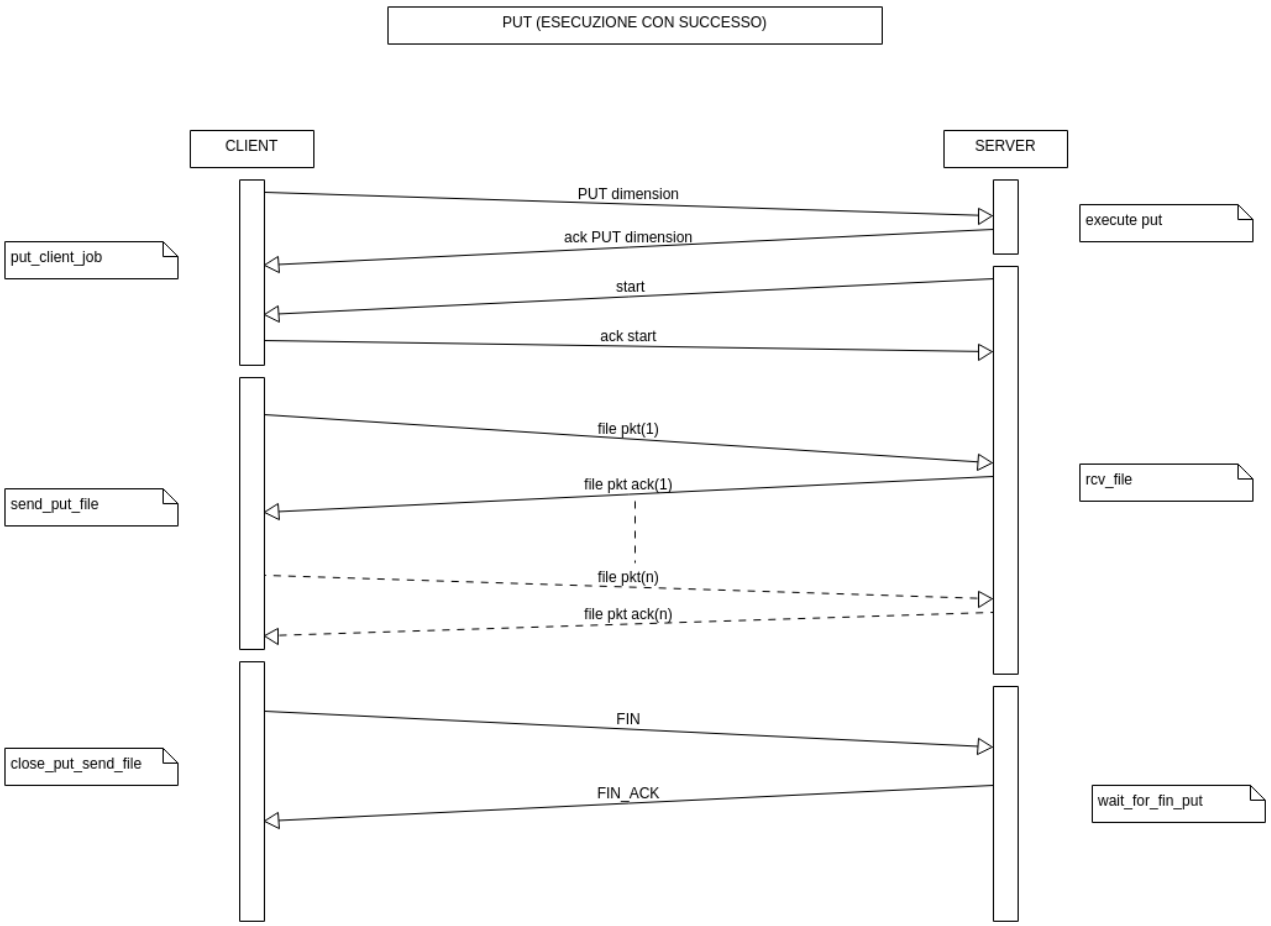
Nelle immagini seguenti è possibile osservare alcuni esempi di esecuzioni delle funzioni principali del programma, ovvero le operazioni LIST, PUT e GET. Per ognuna delle tre viene presentato un caso di esecuzione andato a buon fine, ed un caso in cui invece si incorre in un errore (gestito comunque nel codice in modo da evitare comportamenti inaspettati).
Un esempio di esecuzione con "errore" potrebbe essere una GET effettuata su un file inesistente nella cartella del Server.

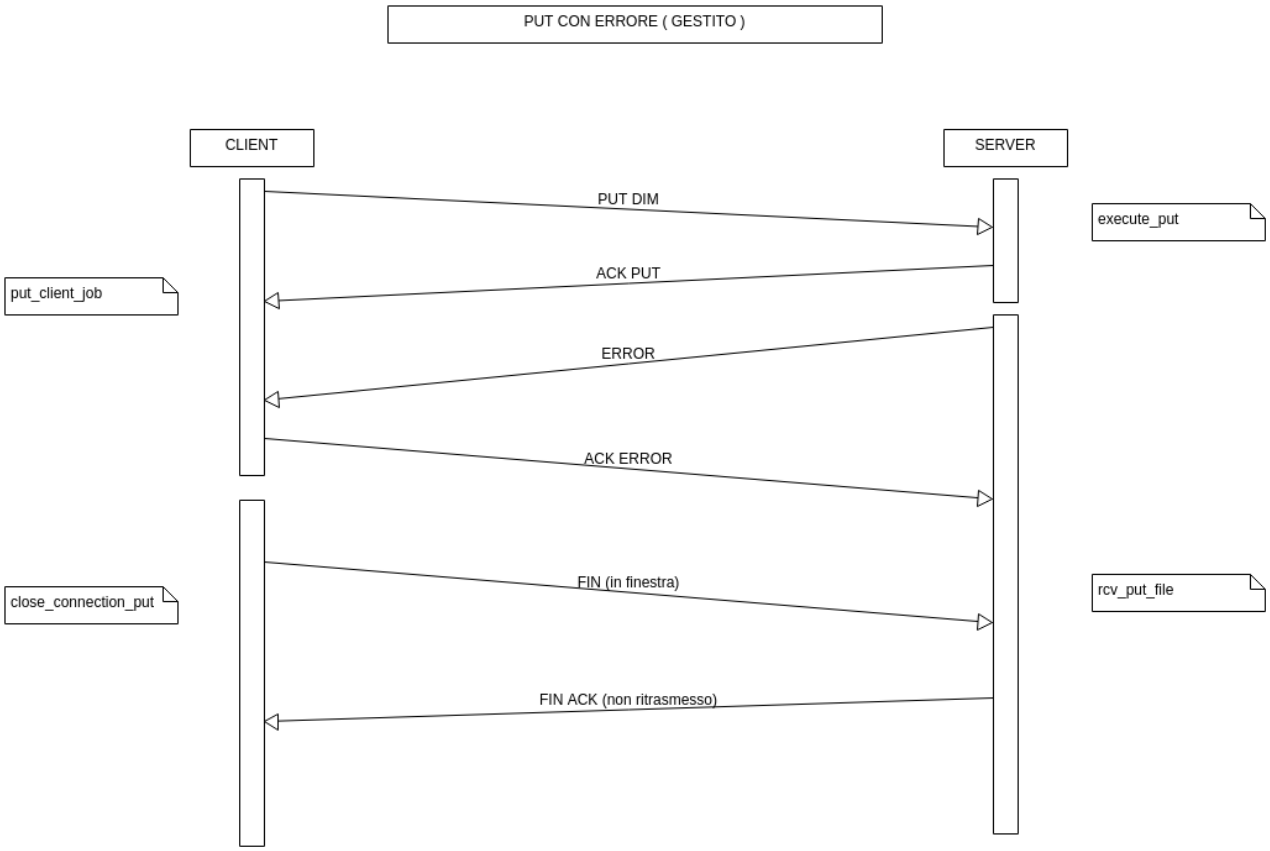
- Comando list:



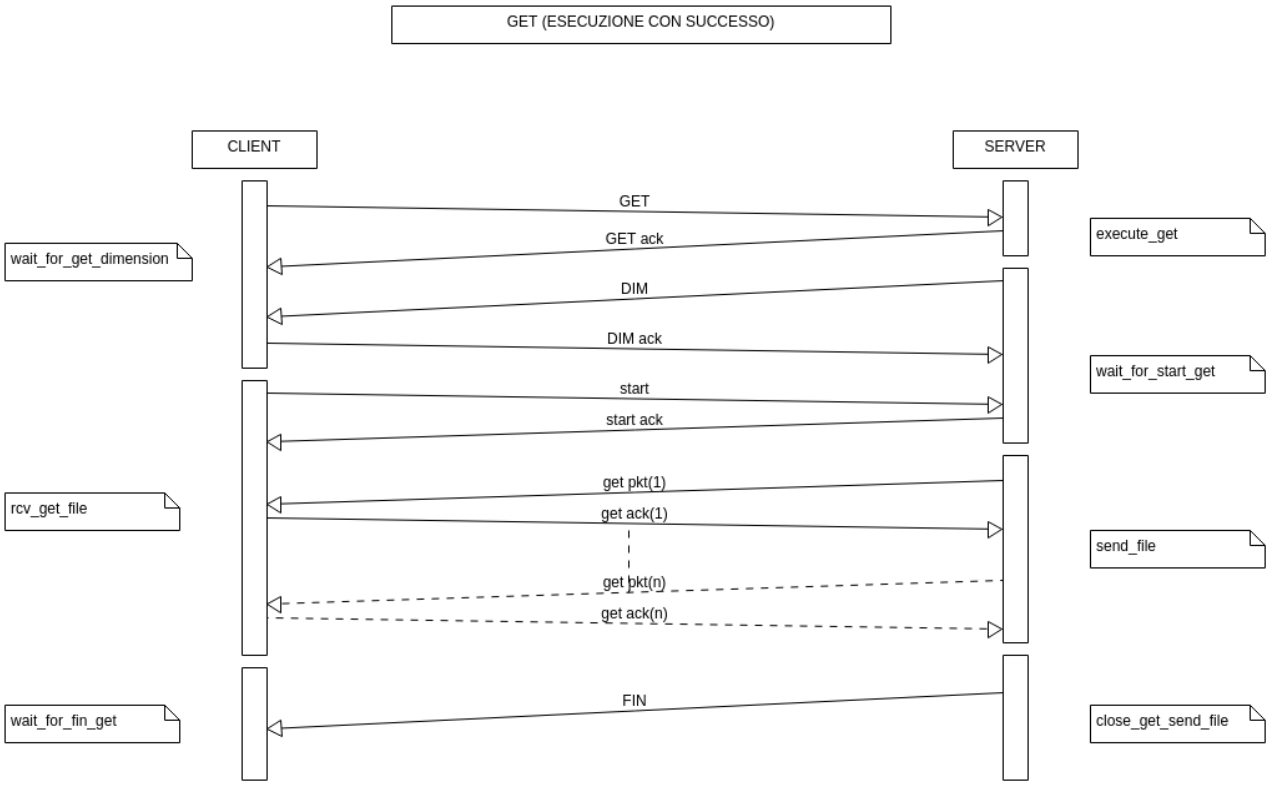

LIST CON ERRORE (GESTITO)
CLIENT
SERVER
LIST
ack LIST
execute_list
wait_for_list_dimension
ERROR
ack ERROR
FIN (in finestra)
close_connecion_list
wait_for_start_list
FIN_ACK

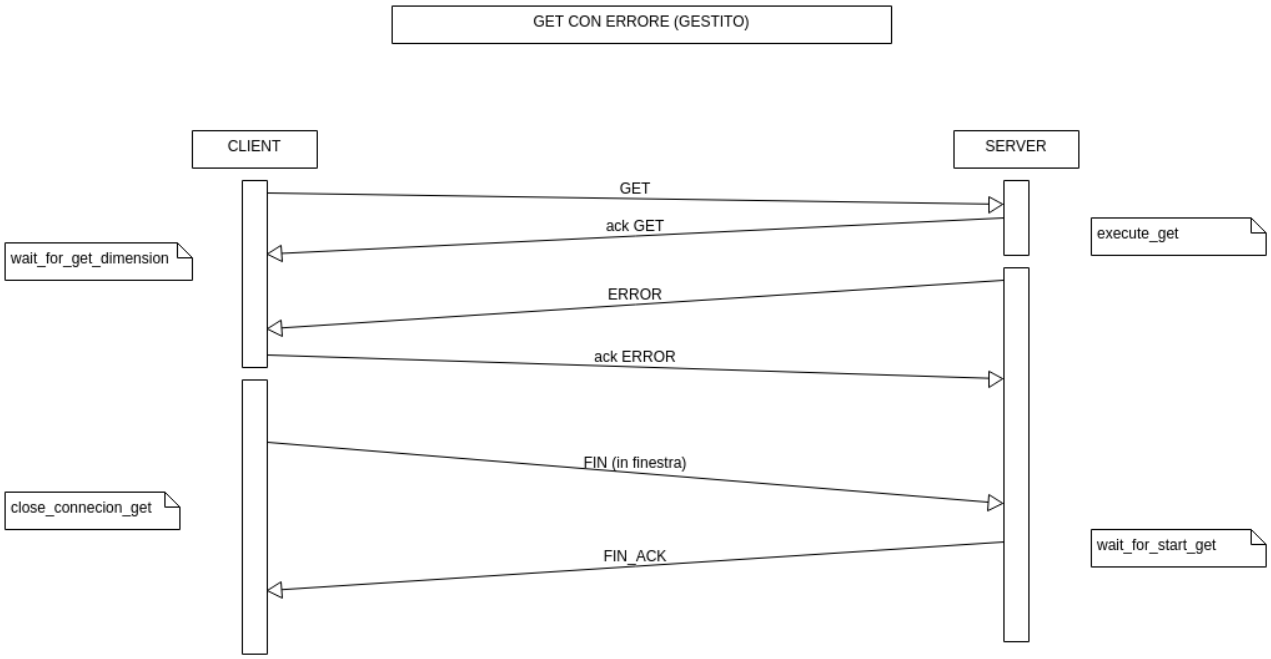
- Comando put:


PUT (ESECUZIONE CON SUCCESSO)
CLIENT
SERVER
PUT dimension
execute put
ack PUT dimension
put_client_job
start
ack start
file pkt(1)
file pkt ack(1)
rcv_file
send_put_file
file pkt(n)
file pkt ack(n)
FIN
close_put_send_file
FIN_ACK
wait_for_fin_put


PUT CON ERRORE (GESTITO)
CLIENT
SERVER
PUT DIM
execute_put
ACK PUT
put_client_job
ERROR
ACK ERROR
FIN (in finestra)
close_connection_put
rcv_put_file
FIN ACK (non ritrasmesso)

- Comando get:


GET (ESECUZIONE CON SUCCESSO)
CLIENT
SERVER
GET
execute_get
GET ack
wait_for_get_dimension
DIM
DIM ack
wait_for_start_get
start
start ack
get pkt(1)
rcv_get_file
get ack(1)
send_file
get pkt(n)
get ack(n)
FIN
wait_for_fin_get
close_get_send_file



Lock su file

Per evitare file corrotti,durante lo scaricamento di un file,sia lato client(comando get) sia lato server(comando put) ,non deve essere possibile inviarlo (comando put lato client e comando get lato server) finquando il trasferimento non è completato. Per far ciò sono stati usati dei lock su file .Essi bloccano il file in scaricamento e non rendono disponibile l'invio dello stesso fin quando la richiesta non termina.
Per quanto riguarda la generazione di files con lo stesso nome, il programma genera atomicamente un filename non ancora in uso che contiene nel nome sia il filename originario sia un certo numero di copia. [file file_lock.c]

Limitazioni

Durante lo sviluppo del progetto abbiamo cercato di evitare il più possibile limitazioni imposte da noi.
Oltre ad un numero di copie dello stesso file (filename_n generato in modo automatico in caso di file già presente) che non può superare i 4 miliardi (unsigned int), ed una dimensione massima supportata praticamente infinita (long int, il che implica 8192 petabyte), la vera limitazione risulta nei filename.
Filename contenenti spazi, parentesi, ed altri caratteri speciali generano problemi sia alle funzioni di sistema quali open(), ma soprattuto al momento del calcolo del MD5.
Ulteriori limitazioni sono quelle dovute al carico del sistema, più strettamente legate all'hardware del calcolatore in uso che all'implementazione.
In particolare, in esecuzione sui nostri calcolatori (sia client che server sulla stessa macchina), il numero di richieste in parallelo (con finestra 100) si assesta attorno ai 200, momento in cui saturiamo la CPU, mandando in timeout le richieste successive.
Per quanto riguarda l'occupazione di memoria (sempre entrambi in esecuzione su macchina locale), questa è stimata essere

4*2100*2W*R byte

dove 4 è dovuto al fatto che per ogni richiesta io abbia 2 finestre di ricezione e 2 di trasmissione allocate, W è la dimensione della finestra, ed R è il numero di richieste attualmente in esecuzione.
L'occupazione dei singoli processi è circa la metà della forma precedente.
Ne consegue che, con finestra 100, e 4GB di memoria libera a disposizione, potremmo supportare un massimo di circa 2500 richieste, ben oltre i limiti riscontrati dovuti alla CPU.

Scelte progettuali

Nel corso della progettazione e dello sviluppo del progetto sono state affrontate diverse "problematiche" che hanno portato a scelte che di seguito motiviamo:

-Formato dei messaggi

I messaggi inviati in rete sono stati gestiti tramite la seguente struttura:

```
struct temp_buffer{
    int seq;
    int ack;
    int lap;
    char command;
    char payload[MAXPKTSIZE-OVERHEAD];
```

I campi che troviamo in temp_buffer rappresentano:
-seq: il numero di sequenza del pacchetto
-ack: intero che ci dice se il pacchetto è stato riscontrato o meno
-lap: numero del "lap" a cui appartiene il pacchetto. Una volta arrivati al massimo numero di sequenza disponibile, si passa al lap successivo a quello attuale.
-command: identifica il "tipo" del pacchetto. Dettagli a seguire.
-payload: il vero e proprio contenuto informativo utile del pacchetto.

Richiamiamo l'attenzione in particolare sul campo command. La scelta dell'inserimento di tale campo è dovuta al fatto che grazie ad esso è possibile stabilire rapidamente la tipologia del pacchetto considerato senza accedere al payload. Il campo command può corrispondere ad uno dei seguenti char definiti tramite delle macro che ne facilitano la comprensione:

```
#define DATA 0
#define GET 1
#define FIN 2
#define FIN_ACK 3
#define START 4
#define ERROR 5
#define DIMENSION 6
#define PUT 7
#define LIST 8
#define SYN 9
#define SYN_ACK 10
```

In questo modo, ricevendo ad esempio un pacchetto che nel rispettivo campo command arreca il numero “9” sappiamo che tale pacchetto è un SYN, e senza dover analizzare altro possiamo procedere all’azione corrispondente.

-Threading/Prethreading

Nella progettazione del sistema abbiamo scelto di utilizzare modelli multi-processo invece che multi-thread allo scopo di rafforzare la separazione tra le varie richieste parallele (eg. gestione dei segnali non deterministica da parte dei thread, separazione della memoria).
-Client di modello “fork”

Per progettare il Client abbiamo preferito un approccio di tipo “fork”. Tale decisione è scaturita dal semplice fatto che è inutile creare a prescindere un pool di processi disponibili, poiché un Client generico potrebbe voler effettuare operazioni non sufficienti per poter sfruttare tutti i processi creati, ed in questo modo avremmo dei processi esistenti che non svolgono alcun lavoro. Creare dei processi al momento del bisogno invece garantisce che nessuno di questi rimanga “disoccupato”.

-Server modello “prefork”

Differentemente dal Client, il server è basato sul modello “prefork”. Esso crea inizialmente un numero prestabilito di processi, in modo da poter soddisfare le richieste di più Client connessi in parallelo. Tali processi si uccideranno dopo aver svolto un determinato numero di compiti(impostato a 10), ed il sistema farà in modo di sostituire il vecchio processo appena morto con uno nuovo.

-Gestione delle ritrasmissioni dei messaggi di connessione e fine connessione

In alcuni casi è stato deciso di evitare la ritrasmissione di pacchetti speciali aventi lo scopo di accordare Client e Server sulla connessione o disconnessione. Questa decisione è stata presa perché (soprattutto in presenza di una elevata probabilità di perdita) avrebbe poco senso continuare la ritrasmissione, ad esempio, di un pacchetto SYN_ACK, soprattutto se il pacchetto con command START è stato già riscontrato (ovviamente se i messaggi vengono mandati in finestra). Pacchetti soggetti a tale eccezione possono essere: SYN_ACK, FIN, FIN_ACK.
In particolare in mancanza di ritrasmissione del pacchetto FIN, la disconnessione avviene allo scadere di un timeout preimpostato.

-Bufferizzazione delle trasmissioni

Per velocizzare la ritrasmissione dei pacchetti persi, si è deciso di memorizzare l’intero pacchetto anziché solamente l’offset dello stesso. In questo modo purtroppo la memoria utilizzata dall’applicazione è maggiore, ma le prestazioni migliorano. Memorizzando solo l’offset del pacchetto sarebbe stata necessaria un’ulteriore lettura dal file.

Piattaforma di sviluppo

Per lo sviluppo del progetto si è fatto uso del terminale e del programma “Clion”.
Il programma Clion è un IDE per linguaggi C e C++ ed è stato usato come editor di testo,quindi impiegato esclusivamente per scrivere il codice del programma.Il terminale,invece,è stato usato per la compilazione di tutti i file e per l’esecuzione del client e del server.
La compilazione dei file è stata semplificata grazie all’uso di un makefile.
Per i dubbi sono stati consultati il sito “Stack Overflow” e la man page di Linux.

Esempi di funzionamento

Il programma permette:
- La visualizzazione di tutti i file che è possibile scaricare dal server (comando list);
- Il download di un file dal server (comando get);
- L’upload di un file sul server (comando put);
- La visualizzazione dei file locali presenti nella directory del client (comando local list);
- L’uscita dal programma con terminazione del processo principale e di tutti i processi figli(comando exit);

Il formato varia da comando a comando.
Per quanto riguarda il comando list,local list e exit,basta semplicemente digitare il nome del comando (e.g. “list”).Per i comandi put e get bisogna digitare nome_comando+filename(e.g. put lavitaèbella.mp4).

Per cambiare i parametri di esecuzione (finestra,probabilità di perdita e timer)è necessario creare un file parameter.txt,collocarlo nella directory dei “file.c” e impostare per ogni parametro un valore.
Per cambiare ip del server e porta del server basta andare nel file basic.h,cambiare i valori e ricompilare il client e il server.

Analisi di prestazioni

Timer:

Il valore del timer ha un effetto decisivo sulle prestazioni del sistema, in quanto, in caso di pacchetto (o relativo ack) perso, prima questo verrà ritrasmesso, tanto prima avremo la possibilità di ricevere l'ack, fondamentale per far avanzare la finestra di trasmissione.

In caso di timer statico, ad esempio, più il valore è vicino a RTT, tanto più il sistema sarà efficiente e veloce nel completare il trasferimento.

Con timer adattivo, questo tenderà al valore di RTT, evitando quindi casi di timer sovradimensionati, e limitando il più possibile ritrasmissioni premature.

Dimensione finestra:

La dimensione della finestra permette al sistema di evitare di incappare in tempi morti, in cui non può trasmettere nulla in quanto ha già una finestra satura, mal al contempo è in attesa di ack, che

potrebbero impiegare molto tempo ad arrivare. Tutto ciò è inficiato anche dalla probabilità di perdita, che impedendo la ricezione di ack, blocca la finestra di spedizione, rallentando così il trasferimento.
In caso di probabilità di perdita elevate (oltre il 50%), una dimensione della finestra grande fa sì che il numero di pacchetti in trasmissione sia sufficientemente grande da sopperire alla perdita, evitando quindi il timeout, e avanzando con il trasferimento, che potrebbe comunque fallire all'avvicinarsi del completamento, dato che il numero di pacchetti in trasmissione diminuirà sempre più.
Da notare come una finestra sovradimensionata rispetto al RTT, faccia sì che il buffer di ricezione di UDP si riempia, incappando in una perdita di pacchetti non dovuta al parametro di perdita.
Rientra in questa particolarità il caso di test con probabilità di perdita 0%, finestra maggiore di 10, ed un timer inferiore ai 10 ms, provato su macchina locale.
In questo caso passiamo da un throughput massimo raggiunto di circa 65MB/s (raggiunto con una sola richiesta in esecuzione, finestra 10, perdita 0%, e timer 5 ms), ad uno di circa 10MB/s, questo dovuto ad un numero di trasmissioni e ritrasmissioni talmente elevato da causare un continuo overflow del buffer UDP.
Questa è l'unica anomalia di prestazioni rivelata durante i test da noi effettuati.

Perdita pacchetti:

La probabilità di perdita è determinante per quanto riguarda l'andamento del sistema, infatti è da notare come questa abbia un effetto esponenziale sul numero di ritrasmissioni, in quanto affligge sia la trasmissione del pacchetto, che del relativo ack, moltiplicando quindi le probabilità.
Ad esempio, con una probabilità di perdita del 50%, si immagina che un pacchetto su 2 venga riscontrato, ragionamento errato in quanto ci si assesterà su una media di un pacchetto su 4 che riceve l'ack senza essere ritrasmesso.
Questo inficia molto sulle prestazioni del sistema, in quanto (soprattutto con finestre di trasmissioni relativamente piccole) potrebbe impedire l'avanzamento della finestra, quindi impedendo la trasmissione di nuovi pacchetti e di fatto bloccare il trasferimento. Se accoppiata con tempi di ritrasmissione molto alti, questa può portare il processo incaricato di inviare il file (o lista), a credere che l'altro non sia più disponibile, portando il trasferimento in timeout.

Dimensione del pacchetto:

A discapito dei 3 parametri, ci siamo trovati di fronte anche la scelta della dimensione del singolo pacchetto.
Inizialmente la scelta era ricaduta su 1468 byte (valore per cui, eccezion fatta per reti intercontinentali, i protocolli di livello 2 non frammentano il pacchetto), ma successivamente sono stati eseguiti test al variare anche di questo valore (se si volesse cambiarlo, basta recarsi nel file basic.h, è presente tra i define all'inizio), e, mentre per valori inferiori a circa 2048, ottenevamo miglioramenti sul throughput per singola richiesta dovuto alla diminuzione dell'overhead computazionale.
Successivamente, a causa della saturazione troppo rapida del buffer UDP, il numero di ritrasmissioni aumenta, rallentando la trasmissione nel suo totale.
La scelta è quindi ricaduta su di un pacchetto di 2061 byte, 2048 di payload, e i restanti 13 di intestazione.

MANUALE D'USO

Per compilare il programma basta aprire un terminale nella directory contenente i file, ed eseguire make install, questa compilerà automaticamente, in due file diversi, sia file che server.
I parametri di probabilità di perdita, timer, e finestra sono contenuti nel file parameter.txt (sempre presente nella stessa directory).
Il primo è la finestra, il secondo è la probabilità di perdita , il terzo il valore del timer, che se 0, attiva il timer adattivo.
A questo punto si può eseguire scrivendo ./client (o server) [directory].
N.B. In caso si voglia ricompilare il codice, assicurarsi che entrambi i programmi (sia client che server) siano stati terminati, altrimenti porterà ad errori di compilazione; per risolvere basterà interrompere i programmi, cancellare i vecchi file compilati, e solo allora rieseguire la make install.