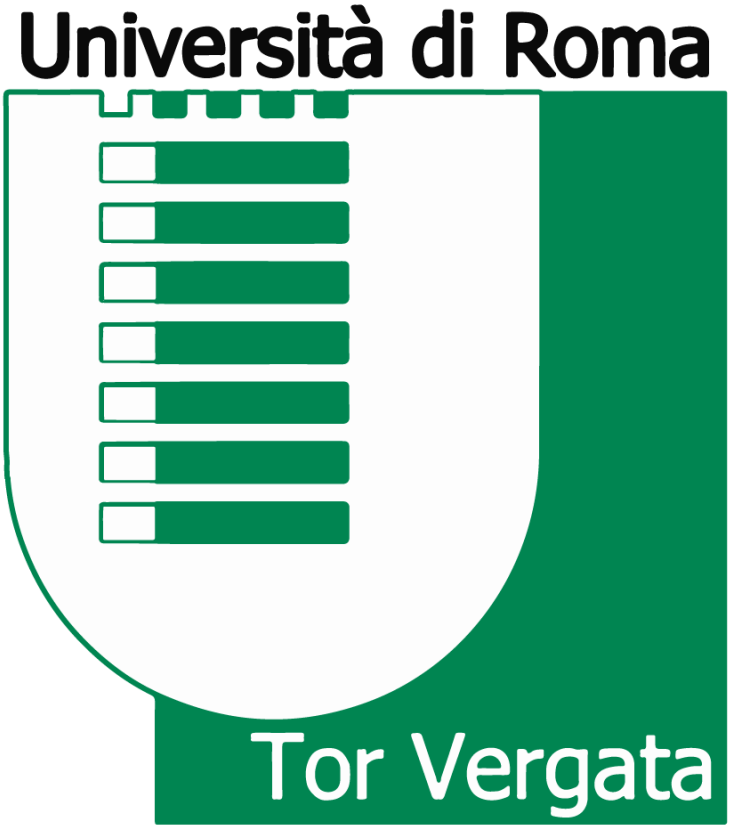
ù

Relazione progetto IW

Trasferimento file affidabile su UDP

Danilo Dell'Orco 0245513
Michele Salvatori 0253519



Sommario

1 Specifica

Lo scopo del progetto è quello di progettare e implementare in linguaggio C usando l'API del socket di Berkeley un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione (socket tipo SOCK_DGRAM, ovvero UDP come protocollo di strato di trasporto). Il software deve permettere:

- Connessione Client-Server senza autenticazione;
- La visualizzazione sul client dei file disponibili sul server (comando **list**);
- Il download di un file dal server (comando **get**);
- L'upload di un file sul server (comando **put**);
- Il trasferimento file in modo affidabile.

Lo scambio di messaggi avviene usando un servizio di comunicazione non affidabile. Al fine di garantire la corretta spedizione/ricezione dei messaggi e dei file sia i client che il server implementano a livello applicativo il protocollo di comunicazione affidabile di TCP con dimensione della finestra di spedizione fissa N.
Per simulare la perdita dei messaggi in rete (evento alquanto improbabile in una rete locale per non parlare di quando client e server sono eseguiti sullo stesso host), si assume che ogni messaggio sia scartato dal mittente con probabilità p.
La dimensione della finestra di spedizione N, la probabilità di perdita dei messaggi p sono configurabili ed uguali per tutti i processi. Il client ed il server devono essere eseguiti nello spazio utente senza richiedere privilegi di root. Il server deve essere in ascolto su una porta di default (configurabile).

2 Scelte progettuali e architettura

2.1 Orientamento alla connessione

Poiché ad ogni coppia IP/PORTA è associata una socket UDP, non è possibile distinguere su una porta quali pacchetti sono destinati ad un processo e quali ad un altro. Per questo si è scelto di assegnare una socket UDP ad ogni nuovo client connesso, limitando quindi il numero di connessioni massime in parallelo al numero massimo di porte disponibili all'utente (16383).

Per avere un server di tipo concorrente, ogni client è configurato per connettersi al Server Dispatcher, comunicando su una socket fissa (IP SERVER/PORTA FISSA). Avviene quindi un setup di connessione che permette ai client di connettersi affidabilmente al Server Dispatcher tramite un 3-way-handshaking. Il Dispatcher è sempre in ricezione senza alcun time-out sulla socket fissa, in quanto non è possibile sapere in alcun modo se e quando un client decida di connettersi al servizio.

Per ogni nuovo client ben connesso, il Server Dispatcher esegue una fork() e torna in ascolto di nuove connessioni. Il processo figlio crea una nuova socket tra quelle disponibili, la comunica al client tramite un messaggio di READY, e si mette in attesa di ricevere comandi da quest'ultimo. Se il client riceve correttamente il messaggio di READY è ben connesso con il server e può quindi servirsi di tutte le operazioni supportate.

2.2 Architettura client-server

Nel momento in cui il client stabilisce la connessione con il server, ha la possibilità di eseguire quattro operazioni:

- LIST – Scaricare in modo affidabile la lista dei file disponibili sul server, e mostrare a schermo la suddetta lista.
- GET – Scaricare in modo affidabile uno dei file disponibili sul server, inserendo il nome del file.
- PUT – Caricare in modo affidabile sul server un file locale del client, inserendo il nome del file.
- CLOSE – Permette di chiudere la connessione tra client e server in modo affidabile, de-allocando tutte le risorse utilizzate.

Se un utente digita un comando non previsto, viene mostrato un messaggio di errore, e viene riportato alla scelta dei comandi.

2.3 Trasferimento dati affidabile TCP

UDP è un protocollo non affidabile, ossia non garantisce né che i pacchetti inviati arrivino al destinatario né che questi arrivino nel giusto ordine. Per far sì che il trasferimento sia affidabile, come da traccia, si è implementato il protocollo TCP.

Il protocollo TCP utilizza dei numeri di sequenza per ordinare i pacchetti, che vengono inviati in pipeline a seconda della dimensione della finestra di trasmissione. La **finestra di trasmissione** stabilisce il numero massimo di pacchetti in volo, ovvero i pacchetti che sono stati inviati, ma che non hanno ancora ricevuto un riscontro. Il ricevente invece mantiene una **finestra di ricezione**, nella quale vengono bufferizzati i pacchetti ricevuti fuori ordine. Questo permette di minimizzare il numero di pacchetti che vengono scartati in ricezione.

Per il riscontro dei pacchetti TCP utilizza un meccanismo di **Cumulative Acknowledgment**. Un host A che invia un pacchetto con numero di ACK pari a N, comunica ad un host B che tutti i pacchetti fino N-1 sono stati ricevuti correttamente, e che ora è in attesa del pacchetto numero N.

Per offrire trasmissione affidabile, i pacchetti persi vengono gestiti da TCP tramite due meccanismi: **Ritrasmissione Rapida** e **Timeout di ritrasmissione.**

2.3.1. Timeout di Ritrasmissione

In TCP viene associato un timer al più vecchio pacchetto in volo. Allo scadere del timer viene ritrasmesso quest'ultimo, permettendo al ricevente di ricevere i pacchetti precedentemente persi all'interno della rete.

Il valore del timer non viene impostato in maniera statica, ma viene stimato in base al valore del **Round Trip Time (RTT)**. Ogni volta che si riceve un ACK non duplicato si misura un *SampleRTT*, ovvero il tempo che intercorre tra l'invio del pacchetto (no ritrasmissione) e la ricezione del suo ACK.

Tramite i valori di *SampleRTT* viene calcolata una media ponderata definita come *EstimatedRTT*. Ogni volta che si calcola un nuovo SampleRTT viene aggiornato il valore dell' *EstimatedRTT* tramite la formula:

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$

$\alpha = 0{,}125$

Per tenere conto di quanto velocemente variano i valori del RTT, viene inoltre calcolata la variabilità DevRTT, che stima quanto il valore SampleRTT si discosta da EstimatedRTT

$$DevRTT = (1 - \beta) * DevRTT + \beta * |SampleRTT - EstimatedRTT|$$

$\beta = 0{,}250$

DevRTT viene utilizzato come safety margin per il calcolo effettivo del timeout, definito come:

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

2.3.2. Ritrasmissione Rapida

Quando il ricevente riceve un pacchetto fuori ordine, invia un ACK cumulativo relativo all'ultimo pacchetto ricevuto correttamente in ordine. Pertanto il mittente, in seguito ad un evento di perdita, può ricevere numerosi ACK relativi allo stesso numero di sequenza. Questi ACK vengono definiti come *ACK duplicati*.

Se il mittente riceve tre ACK duplicati, considera il pacchetto come perso, ed utilizza il meccanismo di **Fast Retransmission**: ricevuti tre pacchetti con numero di acknowledgment pari a K, viene ritrasmesso immediatamente il pacchetto con numero di sequenza K, senza attendere lo scadere del relativo timeout. Questo meccanismo permette di ridurre il tempo che un mittente attende prima di ritrasmettere un pacchetto perso, minimizzando i tempi di idle.

3 Implementazione

Come da specifica, l'affidabilità della trasmissione è garantita tramite il Transmission Control Protocol. Per gestire le operazioni del mittente abbiamo creato la libreria *tcp_sender,* mentre per quelle del ricevente la libreria *tcp_receiver.* Il **mittente** mantiene la finestra di trasmissione tramite le variabili $SendBase$ e $WindowEnd$. $SendBase$ tiene traccia della base della finestra di trasmissione e viene incrementata ogni volta che si riceve un ACK non duplicato. $WindowEnd$ si sposta di conseguenza come $SendBase + TRAN\_WIN$ e indica la fine della finestra. La variabile $NextSeqNum$ indica il numero di sequenza del prossimo pacchetto da inviare, e viene incrementato ad ogni invio di un nuovo pacchetto della finestra.

Lato **ricevente** invece viene mantenuta la finestra di ricezione tramite le variabili $ReceiveBase$ e $WindowEnd$. $ReceiveBase$ punta al pacchetto atteso con più piccolo numero di sequenza e viene incrementata ad ogni arrivo ordinato di un pacchetto fino al primo pacchetto non ancora ricevuto. $WindowEnd$ si sposta di conseguenza come $SendBase + TRAN\_WIN$ e indica la fine della finestra di ricezione. Se arrivano pacchetti fuori dalla finestra di ricezione vengono scartati senza essere bufferizzati.

3.1 Configurazione

Per rendere il codice facilmente configurabile abbiamo inserito i seguenti parametri all'interno della libreria *config.h.* Questi parametri possono essere impostati liberamente prima della compilazione del codice.

- SERVER_PORT = Porta standard del Server
- SERVER_IP = IP standard del Server
- LOST_PROB = Probabilità di perdita dei pacchetti (min. 0 – max. 100)
- TRAN_WIN = Dimensione della finestra di trasmissione
- RECV_WIN = Dimensione della finestra di ricezione
- MAX_RTO = Valore massimo per il timeout adattivo
- MIN_RTO = Valore minimo per il timeout adattivo
- ALPHA = Costante moltiplicativa per il calcolo di EstimatedRTT
- BETA = Costante moltiplicativa per il calcolo di DevRTT
- PKT_SIZE = Dimensione del pacchetto (dati + header)
- STATIC_RTO = 0 timeout adattivo, 1 timeout statico
- RTO _VALUE = Valore fisso del timeout se si utilizza quello statico

3.2 Strutture e funzioni

Per l'implementazione del nostro progetto, abbiamo scritto le seguenti funzioni e strutture:

- **Struttura packet**: utilizzata per gestire il trasferimento di pacchetti di un file, composta da un campo *Data* e da un *Header*. Il campo data corrisponde al payload, ovvero ai dati effettivi da trasmettere. L'header contiene tutti i campi utilizzati per la nostra implementazione del protocollo TCP, che sono:

    - *seq_num*: Numero di sequenza del pacchetto.

    - *pkt_dim*: Dimensione del pacchetto. Fa sì che in ricezione venga scritto sul file un numero di byte pari a quelli effettivamente trasmessi, evitando situazioni in cui il file scritto sia corrotto.

    - *num_pkts*: Intero che indica il numero totale di pacchetti relativi al file in trasmissione. Permette che il ricevente sappia esattamente quanti pacchetti dovrà ricevere relativamente ad una trasmissione.

    - *sent_time*: Misura in microsecondi del tempo in cui è stato trasmesso il pacchetto. Permette di calcolare il *SampleRTT* per impostare correttamente il timeout di ritrasmissione.

```c
typedef struct packet{
    // Header
    int seq_num;
    short int pkt_dim;
    int num_pkts;
    uint64_t sent_time;

    // Payload
    char data[PKT_SIZE-2*sizeof(int)-sizeof(short int)];
} packet;
```

- **Funzione tcp_sender:** Gestisce l'invio dei pacchetti tramite il protocollo TCP. Suddivide il file in pacchetti, assegna ad ognuno di essi il suo numero di sequenza, e li invia in pipeline tramite *sendto*. La ricezione degli ACK avviene in parallelo all'invio dei pacchetti, tramite il thread *receive_ack.*

  La funzione *tcp_sender* resta in un ciclo indefinito finchè ci sono pacchetti da inviare; ad ogni iterazione viene invocata la funzione *send_window,* che si occupa di inviare i pacchetti non in volo relativi all'attuale finestra di trasmissione.

  Parallelamente il thread trasla la finestra ad ogni ACK non duplicato ricevuto correttamente. Ricevuto l'ACK cumulativo relativo all'ultimo pacchetto, termina l'esecuzione del ciclo while e di conseguenza l'invio dei pacchetti.

  Per mantenere lo stato di invio dei pacchetti utilizziamo l'array di interi *check_pkt*, dove *check_pkt[i]* indica lo stato dell'i-esimo pacchetto (0 da inviare, 1 in volo, 2 acked).

```c
void *receive_ack(void *arg){
    ...
    int duplicate_ack_count = 0;
    while(true){
        if (recvfrom(socket, &ack_num, sizeof(int), 0, ... ) < 0){
            perror ("Errore ricezione ack");
            exit(-1);
        }

        // Ricevuto ACK non duplicato
        if (ack_num>SendBase){
            SendBase = ack_num;
            WindowEnd = MIN(tot_pkts,SendBase+TRAN_WIN-1);
            duplicate_ack_count = 0;
            cumulative_ack(ack_num);
            update_timeout(pkt[ack_num-2]);
            if (tot_acked == tot_pkts){
                fileTransfer = false;           //Stoppa il ciclo while del sender
                set_retransmission_timer(0);    //Stoppa il timer
                break;                          //Termina la ricezione degli ACK
            }
        }

        //Ricevuto ACK duplicato
        else {
            duplicate_ack_count++;
            if (duplicate_ack_count == 3){
                fast_retransmission(SendBase-1);
                duplicate_ack_count = 0;
            }
        }
    }
}
```

```c
void tcp_sender(int socket, struct sockaddr_in *receiver_addr,...) {
    pthread_create(&thread,NULL,receive_ack,(void*)&t_args);
    ...
    // Suddivisione del file in pacchetti e assegnazione dei numeri di sequenza
    for(i=0; i<tot_pkts; i++){
        pkt[i].seq_num = i+1;
        pkt[i].num_pkts = tot_pkts;
        pkt[i].pkt_dim=read(fd, pkt[i].data, pkt_data_size);
        if(pkt[i].pkt_dim==-1){
            pkt[i].pkt_dim=0;
        }
    }
    // Trasmissione dei pacchetti
    while(fileTransfer){ //while ho pachetti da inviare
        send_window(socket, receiver_addr, pkt);
    }
    end_transmission();
}

//Invia tutti i pacchetti nella finestra
void send_window(int socket, struct sockaddr_in *client_addr, packet *pkt){
    WindowEnd = MIN(tot_pkts,SendBase+TRAN_WIN-1);

    for(i=NextSeqNum-1; i<WindowEnd; i++){
        WindowEnd = MIN(tot_pkts,SendBase+TRAN_WIN-1);
        send_packet(i);
        NextSeqNum++;

        if (!isTimerStarted){
            isTimerStarted = true;
            set_retransmission_timer(timeoutInterval);
        }
    }
}
```

- **Funzione tcp_receiver:** Gestisce la ricezione dei pacchetti tramite il protocollo TCP. Questa funzione resta in ricezione indefinita di pacchetti tramite *recvfrom*, e termina solamente quando sono stati ricevuti tutti correttamente.

  Ricevuto il primo pacchetto, tramite l'apposito campo nell'header, il ricevente ricava il numero totale di pacchetti da ricevere. Da qui, fino al termine della trasmissione, viene effettuato il controllo tra numero di sequenza atteso (*expected_seq_num*) ed il numero di sequenza del pacchetto appena ricevuto.

  Nel caso in cui arrivi un pacchetto con numero di sequenza atteso viene traslata in avanti la finestra di ricezione e viene bufferizzato il payload del pacchetto.

  Nel caso arrivi un pacchetto non in ordine contenuto nella finestra, viene bufferizzato senza traslare la finestra.

  Se arriva un pacchetto duplicato o fuori dalla finestra esso verrà scartato senza essere bufferizzato. In ogni caso viene sempre inviato al mittente un ACK cumulativo relativo all'ultimo pacchetto ricevuto correttamente in ordine.

  La finestra viene traslata in avanti tramite la funzione *move_window* che incrementa la variabile *ReceiveBase* fino all'ultimo pacchetto ricevuto in ordine. Terminata la ricezione dei pacchetti, questi vengono scritti in ordine sul file.

  Per mantenere lo stato di ricezione dei pacchetti utilizziamo l'array di interi *check_pkt_received*, dove *check_pkt_received[i]* indica lo stato dell'i-esimo pacchetto (0 da ricevere, 1 ricevuto).

```c
// Gestisce la ricezioni di pacchetti secondo il protocollo TCP
void tcp_receiver(int socket, struct sockaddr_in *sender_addr, ... ){
    while(tot_received != tot_pkts){
        memset(&new_pkt, 0, sizeof(packet));

        if((recvfrom(socket, &new_pkt, PKT_SIZE, 0, (struct sockaddr *), ... )<0)) {
            perror("error receive pkt ");
            continue;
        }

        // Alloca le risorse necessarie
        if (!allocated){
            tot_pkts = new_pkt.num_pkts;
            pkt=calloc(tot_pkts, sizeof(packet));
            check_pkt_received=calloc(tot_pkts, sizeof(int));
            allocated = true;
        }

        // Arrivo di un nuovo pacchetto non in ordine
        if (ReceiveBase < new_pkt.seq_num && new_pkt.seq_num <= WindowEnd ... ){
            mark_recvd(new_pkt.seq_num);
            memset(pkt+new_pkt.seq_num-1, 0, sizeof(packet));
            pkt[new_pkt.seq_num-1] = new_pkt;
        }

        // Arrivo ordinato di segmento con numero di sequenza atteso
        else if (new_pkt.seq_num == ReceiveBase){
            mark_recvd(new_pkt.seq_num);
            move_window();
            memset(pkt+new_pkt.seq_num-1, 0, sizeof(packet));
            pkt[new_pkt.seq_num-1] = new_pkt;
        }
        send_cumulative_ack(ReceiveBase, socket);
    }

    // Scrivo un pacchetto per volta in ordine sul file
    for(i=0; i<tot_pkts; i++){
        write(fd, pkt[i].data, pkt[i].pkt_dim);
    }
}
```

```c
void mark_recvd(int seq){
    check_pkt_received[seq-1] = 1;
    tot_received++;
}

int is_received(int seq){
    if (check_pkt_received[seq-1] == 1){
        return 1;
    }
    return 0;
 }

// Trasla in avanti la finestra fino al primo pacchetto non ancora ricevuto.
void move_window(){
    int j = ReceiveBase;
    for (j = ReceiveBase;j<=WindowEnd;j++){
        if (is_received(j)){
            ReceiveBase++;
            WindowEnd = MIN(ReceiveBase + RECV_WIN,tot_pkts);
        }
        else{
            break;
        }
    }
}
```

- **Funzione update_timeout**: Utilizzata per calcolare il timeout di ritrasmissione. In input la funzione accetta un pacchetto, dal quale si ricava il valore in millisecondi relativo al suo invio tramite l'apposito campo nell'header (*sentTime*). Il valore *recvTime* indica il valore in millisecondi di quando è stato ricevuto il pacchetto ed il sampleRTT viene calcolato come la differenza tre *recvTime* e *sentTime*.
  Il timeout di ritrasmissione viene calcolato tramite la formula mostrata nel paragrafo 2.3.1 ed il suo valore viene salvato nella variabile globale *timeoutInterval.*
  Questa funzione viene invocata ogni volta che si riceve un acknowledgment non duplicato, passando come parametro il pacchetto riscontrato dall'ACK ricevuto.

```
void update_timeout(packet to_pkt) {
    uint64_t recvTime = time_now();
    uint64_t sentTime = to_pkt.sent_time;
    uint64_t sampleRTT = recvTime - sentTime;
    estimatedRTT = (1-ALPHA) * estimatedRTT + ALPHA * sampleRTT;
    devRTT = (1-BETA)*devRTT + BETA * abs(sampleRTT - estimatedRTT);
    timeoutInterval = (estimatedRTT + 4 * devRTT);
}
```

- **Funzione set_retransmission_timer**: Imposta il timer di ritrasmissione per il più vecchio pacchetto in volo in base al valore passato come input. Utilizza la funzione *setitimer* della libreria *<sys/time.h>* che inizializza un timer, allo scadere del quale invia un segnale al processo.
  Ricevuto il segnale viene invocata la funzione *timeout_routine* che effettua la ritrasmissione del pacchetto. Tramite i parametri `MAX_RTO` e `MIN_RTO` è possibile configurare i valori massimi e minimi del timer. Se invece è abilitata l'opzione di testing `STATIC_RTO`, il timer verrà impostato sempre pari a `RTO_VALUE`.

```
// Imposta il timer di ritrasmissione
void set_retransmission_timer(int micro){
    struct itimerval it_val;
    if (STATIC_RTO == 1){
        micro = STATIC_RTO;
    }
    else if (micro >= MAX_RTO){
        micro = MAX_RTO;
    }
    else if (micro <= MIN_RTO){
        micro = MIN_RTO;
    }
    it_val.it_value.tv_sec = 0;
    it_val.it_value.tv_usec = micro;
    it_val.it_interval.tv_sec = 0;
    it_val.it_interval.tv_usec = 0;
    if (setitimer(ITIMER_REAL, &it_val, NULL) == -1) {
        perror("Set Timer Error:");
        exit(1);
    }
}
```

```
void timeout_routine(){
    int rtx_seq = SendBase;
    isTimerStarted = false;
    timeout_retransmission(rtx_seq-1);
    return;
}
```

- **Funzione is_packet_lost**: Simula la perdita di un pacchetto, in quanto in una rete locale è un evento molto raro. In input accetta un intero che corrisponde alla probabilità di perdita configurata prima di lanciare l'applicazione. All'interno della funzione viene calcolato un numero aleatorio; se questo risulta minore o uguale della probabilità di perdita scelta, ritorna *true* (pacchetto perso) altrimenti *false* (pacchetto non perso). Questa funzione viene invocata prima di inviare ogni pacchetto, e solo nel caso in cui ritorni false verrà effettivamente chiamata *sendto.*

```
bool is_packet_lost(int prob){
  int random = rand() %100;
  if (random<prob){
      return true;
  }
  return false;
}
```

```
int send_packet(int index){
    if (is_packet_lost(LOST_PROB)){
        set_packet_sent(index);
        num_packet_lost++;
        return -1;
    }
    else{
        sendto(sock,pkt+index,PKT_SIZE,...
        return 1;
    }
}
```

- **Funzioni folder_files**: Utilizzate per la gestione della lista dei file su client/server. Queste funzioni restituiscono il numero di file all'interno della cartella client/server, e popolano un array con tutti i nomi dei file contenuti nella relativa cartella.

  Sul server viene utilizzata *server_folder_files* per implementare il comando LIST: viene creato un file con tutti i filename contenuti sulla cartella del server, e viene inviato al client per comunicargli i file disponibili.

  Sul client viene utilizzata *client_folder_files* per implementare il comando PUT, in modo da mostrare a schermo i file presenti nella cartella locale che può caricare sul server.

  In entrambe le funzioni il file inviato viene cancellato non appena viene mostrata a schermo la lista di file disponibili.

3.3 Connessione

Quando il server viene avviato, inizializza la sua socket e si mette in attesa di richieste da parte dei client. Il setup di connessione è gestito rispettivamente tramite le funzioni *client_reliable_conn* e *server_reliable_conn*.

Il server tramite *recvfrom* si mette in attesa di un messaggio di SYN dal client. Il client invia un messaggio di SYN al server tramite *sendto*, comunicando la volontà di instaurare una connessione.

Nel momento in cui il server riceve il SYN, concorda la richiesta di connessione ed invia tramite *sendto* un messaggio di SYNACK al client, confermando la ricezione del messaggio di SYN. Quando il client riceve il messaggio di SYNACK ha instaurato la connessione ed invia un messaggio di ACK al server. Quando il server riceve l'ACK considera la connessione come stabilita.

```c
// Stabilisce la connessione con il client tramite 3-way handshake
int server_reliable_conn (int server_sock, struct sockaddr_in* client_addr) {
    int control;
    char *buff = calloc(PKT_SIZE, sizeof(char));
    socklen_t addr_len = sizeof(*client_addr);

    // In attesa di ricevere SYN
    control = recvfrom(server_sock, buff, PKT_SIZE, 0, (struct sockaddr *)client_addr, &addr_len);
    if (control < 0 || strncmp(buff, SYN, strlen(SYN)) != 0) {
        printf("SERVER: connection failed (receiving SYN)\n");
        return 1;
    }

    // Invio del SYNACK
    printf("%s SERVER: sending SYNACK\n", time_stamp());
    control = sendto(server_sock, SYNACK, strlen(SYNACK), 0, (struct sockaddr *)client_addr, addr_len);
    if (control < 0) {
        printf("SERVER: connection failed (sending SYNACK)\n");
        return 1;
    }

    // In attesa di ricevere ACK
    control = recvfrom(server_sock, buff, PKT_SIZE, 0, (struct sockaddr *)client_addr, &addr_len);
    if (control < 0 || strncmp(buff, ACK, strlen(ACK)) != 0) {
        printf("SERVER: connection failed (receiving ACK)\n");
        return 1;
    }
    // Connessione stabilita
    printf("%s SERVER: connection established\n", time_stamp());
    return 0;
}
```

A questo punto il Server Dispatcher genera un ID numerico univoco relativo al nuovo client connesso, e crea un processo figlio e una relativa socket associata per gestire la nuova connessione.

Il processo figlio, tramite un messaggio di READY, comunica al client con cui dovrà relazionarsi che è pronto per comunicare. Grazie a questo messaggio il client entra a conoscenza dell'IP/PORTA della socket con cui interfacciarsi, e dell'ID assegnatogli dal server.

Il processo figlio si occupa dunque di gestire tutte le operazioni relative alla connessione per il quale è stato generato, e terminerà solamente alla chiusura del client tramite comando CLOSE.

L'ID numerico assegnato dal Server al Client non ha valenza effettiva all'interno del protocollo, ma permette di distinguere a livello di interfaccia quale client sta interagendo con il server.

Server.c

```
// Eseguo la fork del server per ogni nuova connessione da parte di un client
if (server_reliable_conn(server_sock, &client_address) == 0){
pid = fork();
num_client++;

if (pid < 0){
    printf("> SERVER: fork error\n");
    exit(-1);
}
if (pid == 0){
    pid = getpid();
    child_sock = create_socket();
    send_ready_pkt.clientNum = num_client;
    snprintf(send_ready_pkt.message,6,"%s",READY);

    // Invio del pacchetto di READY che comunica al client il suo ID di connessione
    control = sendto(child_sock, &send_ready_pkt, sizeof(send_ready_pkt),0,...);
    if (control < 0) {
        printf("> SERVER %d: port comunication failed\n", pid);
    }
```

Client.c

```
// Ricezione del pacchetto di READY per ricevere un ID dal server
control = recvfrom( ..., (struct sockaddr *)&server_address, &addr_len);
clientNum = recv_ready_pkt.clientNum;
if (control < 0 || strcmp(READY,recv_ready_pkt.message) != 0) {
    printf("CLIENT: server dispatching failed\n");
    exit(-1);
}
```

3.4 Comandi

Una volta stabilita la connessione tra un client ed il server, viene mostrata una lista delle operazioni disponibili (list, get, put, close). Il client permette l'inserimento di un comando dalla tastiera tramite *scanf*, mentre le diverse operazioni sono state implementate tramite uno switch case. A seconda del comando scelto dal client, verrà inviato al server un pacchetto contenente l'effettiva operazione di cui ci si vuole servire.

Il server riceve il pacchetto contenente il comando, e analogamente al client sceglie l'operazione da eseguire con uno switch case.

Tramite l'istruzione *goto,* terminata l'esecuzione di un'operazione, il client torna all'etichetta *menu* per chiedere l'inserimento di un nuovo comando, mentre il server torna all'etichetta *request* per mettersi nuovamente in ricezione di un pacchetto operazione.

Client.c

```
menu:
    printf("\n\n============= COMMAND LIST ================\n");
    printf("1) List available files on the server\n");
    printf("2) Download a file from the server\n");
    printf("3) Upload a file to the server\n");
    printf("4) Close connection\n");
    printf("==========================================\n\n");
    printf("> Choose an operation: ");
    scanf ("%d",&answer);

    switch (answer) {
```

Server.c

```
request:
    printf("\n\n======================================\n");
    printf("> Server waiting for request....\n");
    memset(buff, 0, sizeof(buff));

    if (recvfrom(server_sock, buff, PKT_SIZE, ...) < 0){
        printf("> request failed\n");
        free(buff);
        free(path);
        return 0;
    }

    switch (*(int*) buff ){
```

3.4.1. List

Il comando LIST viene utilizzato per ricevere la lista dei file presenti sul server. Su quest'ultimo, tramite la funzione *files_from_folder_server*, viene ricavata la lista dei file presenti da poter fornire al client. La lista viene inserita su un file temporaneo, e la trasmissione affidabile da server a client è gestita tramite TCP. Al termine della trasmissione il server elimina il file. Il client stampa il file su standard output mostrando a schermo i file disponibili, dopodiché elimina anch'esso il file.

3.4.2. Get

Il comando GET viene utilizzato per scaricare sul client un file presente nel server. Nel caso in cui venga richiesto un file non esistente, il server invia un pacchetto di errore e il client mostra all'utente un messaggio che comunica la non esistenza del file.

Nel caso in cui il client richieda un file con lo stesso nome di un altro che ha già salvato, viene mostrato un messaggio di conferma, che permette all'utente di scegliere se sovrascrivere o meno il file locale.

Se l'utente sceglie di non sovrascrivere il file il client invia un pacchetto al server contenente il messaggio *NOVERW* e ritorna sulla schermata di scelta del comando. Il server riceve questo pacchetto e ritorna in attesa di un comando da parte del client.

Se l'utente sceglie di sovrascrivere il file, oppure se il file non è presente sul client, viene effettuata la trasmissione del file tramite TCP (funzioni *tcp_sender* e *tcp_receiver*). Al termine della trasmissione, se avvenuta correttamente, il client torna alla scelta delle operazioni e il server si rimette in attesa della ricezione di comandi.

3.4.3. Put

Il comando PUT viene utilizzato per caricare sul server un file presente nel client. Nel caso in cui il client volesse caricare un file non presente nella cartella locale, viene mostrato a schermo un messaggio di errore. Il client torna quindi alla scelta dell'operazione senza inoltrare il comando PUT al server.

Nel caso in cui il client volesse caricare un file già presente sul server, viene mostrato un messaggio di errore e la trasmissione non avviene, in quanto il client non dovrebbe avere i permessi per sovrascrivere un file sul server.

Quando invece l'utente inserisce il nome del file da caricare, presente sul client e non esistente sul server, la trasmissione avviene tramite protocollo TCP (funzioni *tcp_sender* e *tcp_receiver*). Dopo la trasmissione, se avvenuta correttamente, il client torna alla scelta delle operazioni e il server si rimette in attesa di comandi.

3.4.4. Close

Il comando CLOSE viene utilizzato quando l'utente vuole chiudere la connessione con il server in maniera affidabile. Quando il client invia il comando al server, entrambi gestiscono la chiusura rispettivamente con le funzioni *client_reliable_close* e *server_reliable_close*.

Il server è inizialmente in attesa di un messaggio di FIN da parte del client. Quando il client riceve il comando di Close, invia un messaggio di FIN al server, comunicando la volontà di chiudere la connessione. Quando il server riceve il messaggio, invia un FINACK, informando il client di aver ricevuto la richiesta di chiusura. Il client si mette nuovamente in attesa, questa volta di un messaggio di FIN da parte del server, che comunica che sta chiudendo. Quando riceve il messaggio, il client invia un FINACK dando l'ok al server della chiusura e chiude la connessione. Il server chiude alla ricezione del FINACK, terminando l'esecuzione del server figlio che gestiva la connessione. Il Dispatcher non termina e resta in attesa di richieste di connessione.

```c
void server_reliable_close (int server_sock, struct sockaddr_in* client_addr) {
    // In attesa di ricevere FIN
    control = recvfrom(server_sock, buff, PKT_SIZE, 0, ...);
    if (control < 0 || strncmp(buff, FIN, strlen(FIN)) != 0) {
        printf("SERVER: close connection failed (receiving FIN)\n");
        exit(-1);
    }
    // Invio del FINACK
    control = sendto(server_sock, FINACK, strlen(FINACK), 0, ...);
    if (control < 0) {
        printf("SERVER: close connection failed (sending FINACK)\n");
        exit(-1);
    }
    // Invio del FIN
     control = sendto(server_sock, FIN, strlen(FIN), 0, ...);
    if (control < 0) {
        printf("SERVER: close connection failed (sending FIN)\n");
        exit(-1);
    }
    // In attesa del FINACK
    memset(buff, 0, sizeof(buff));
    control = recvfrom(server_sock, buff, PKT_SIZE, 0, ...);
    if (control < 0 || strncmp(buff, FINACK, strlen(FINACK)) != 0) {
        printf("SERVER: close connection failed (receiving FINACK)\n");
        exit(-1);
    }
    // Connessione chiusa
    printf("SERVER: connection closed\n");
    return;
}
```

```c
void client_reliable_close (int client_sock, struct sockaddr_in *server_addr) {
    // Invio del FIN
    control = sendto(client_sock, FIN, strlen(FIN), 0, ...);
    if (control < 0) {
        printf("CLIENT: close failed (sending FIN)\n");
        exit(-1);
    }
    // In attesa del FINACK
    memset(buff, 0, sizeof(buff));
    control = recvfrom(client_sock, buff, strlen(FINACK), 0, ...);
    if (control < 0 || strncmp(buff, FINACK, strlen(FINACK)) != 0) {
        printf("CLIENT: close connection failed (receiving FINACK)\n");
        exit(-1);
    }
    // In attesa del FIN
    memset(buff, 0, sizeof(buff));
    control = recvfrom(client_sock, buff, strlen(FIN), 0, ...);
    if (control < 0 || strncmp(buff, FIN, strlen(FIN)) != 0) {
        printf("CLIENT: close connection failed (receiving FIN)\n");
        exit(-1);
    }
    // Invio del FINACK
    control = sendto(client_sock, FINACK, strlen(FINACK), 0, ...);
    if (control < 0) {
        printf("CLIENT: close connection failed (sending FINACK)\n");
        exit(-1);
    }
    // Connessione chiusa
    printf("CLIENT: connection closed\n");
}
```

4 Installazione e Manuale d'uso

Per utilizzare il programma, aprire due terminali e muoversi nella directory 'tcp' del progetto. Compilare il programma con il comando 'make'. Gli eseguibili generati sono rispettivamente 'server' e 'client'.

Lanciare il comando *./server* sul primo terminale e *./client* sul secondo terminale per stabilire la connessione tra i due processi.

Per eseguire una operazione digitare sul client un comando tra quelli disponibili e premere invio (1 = List, 2 = Get, 3 = Put). Durante l'esecuzione dell'operazione verrà mostrata sul client una barra di avanzamento con la percentuale di download/upload.

Al termine dell'operazione verrà mostrato il risultato sul terminale del client, e sarà possibile digitare un nuovo comando.

Per terminare l'esecuzione, nel client digitare il comando *Close* (numero 4 nel menù), nel server uccidere il processo con il comando "Ctrl+C".

5 Piattaforme utilizzate

Per lo sviluppo ed il testing del programma sono state utilizzate le seguenti piattaforme:

- Linux 18.04 LTS / 8GB RAM / i5 4th gen, compilatore gcc, IDE Visual Studio Code, terminale Linux.
- Linux 18.04 LTS / 8GB RAM / i7 7th gen, compilatore gcc, IDE Atom, terminale xTerminal.

6 Testing

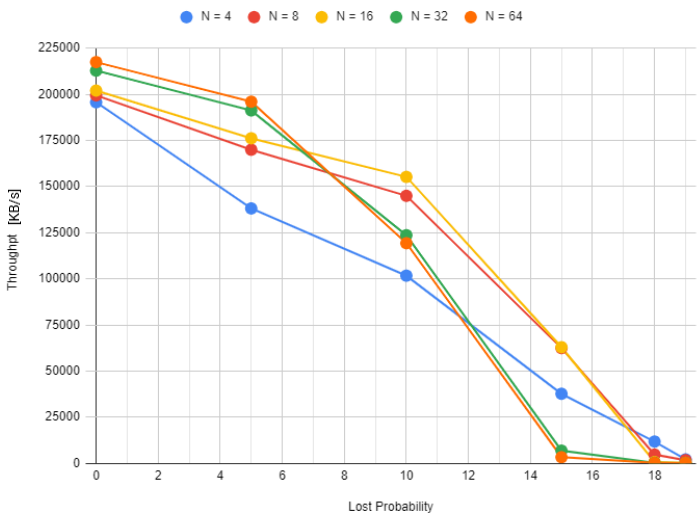
Per valutare le prestazioni della nostra implementazione sono state effettuate una serie di trasmissioni dello stesso file (3.2 MB), calcolando la media del troughput al variare della finestra e della probabilità di errore. Abbiamo testato due casistiche:

- Caso con timeout adattivo (par. 2.3.1) al variare della probabilità di perdita e dimensione della finestra.
- Caso con timeout statico al variare della probabilità di perdita, con dimensione della finestra fissa N=8.

6.1 Testing con timeout adattivo

Nella tabella sono riportati i valori medi dei throughput ottenuti nei test effettuati.

Lost Probability \ Window Size	4	8	16	32	64
0	195696,47	199431,33	202009,76	212837,87	217466,5
5	138139,6	169966,88	176169,18	191263,43	195989,91
10	101694,4	144967,43	155236,53	123631,86	119301,57
15	37666,84	62422,55	62897,16	6829,37	3278,24
18	11757,53	4703,95	652,95	239,91	207,39
19	2101,54	1662,51	205,88	145,21	113,57
20	533,48	431,56	136,11	107,27	89,61
25	85,8	48,98	35,31	28,32	25,09
30	38,8	26,15	20,02	14,25	10,77
40	13,34	9,81	6,72	4,56	3,12
60	4,94	3,66	1,79	1,11	0,85
80	1,8	1,03	0,88	0,65	0,32



Analizzando i risultati dei test possiamo osservare come per basse probabilità di perdita (P<8%) il throughput aumenti con la dimensione della finestra. Questo perché con finestre più grandi verranno inviati e bufferizzati più pacchetti minimizzando i tempi di attesa.

Al contrario con un alto tasso di perdita, il throughput diminuisce all'aumentare della dimensione della finestra. Questo andamento è ragionevole, in quanto con una finestra di ricezione grande e perdendo numerosi pacchetti, verranno bufferizzati molti pacchetti fuori ordine, comportando principalmente due svantaggi:

- Vengono inviati molti ACK duplicati relativi allo stesso pacchetto atteso. Di conseguenza tramite **fast retransmission** si effettuano molte ritrasmissioni inutili.

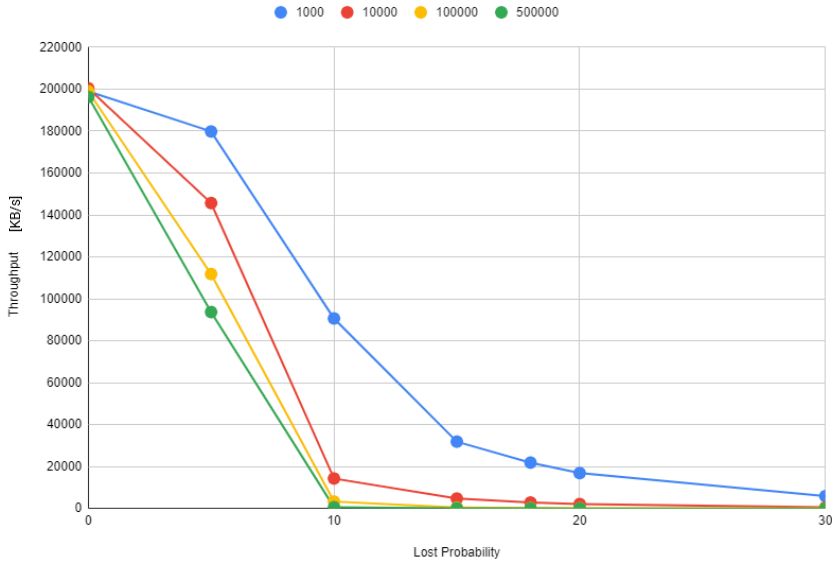
- Aumenta progressivamente il valore del timeout di ritrasmissione, portando a maggiori periodi di IDLE. Infatti con una grande finestra di ricezione, gli ultimi pacchetti bufferizzati verranno riscontrati dopo molto tempo tramite **ACK cumulativo**, portando a valori maggiori del *sampleRTT*.

Per questo in TCP oltre alla gestione del timeout adattivo sono previsti il controllo di flusso e di congestione per il dimensionamento della finestra di ricezione, la cui implementazione non è tuttavia prevista da specifica.

6.2 Testing con timeout statico

Per testare il caso con timeout statico la dimensione della finestra è stata fissata a N=8, mentre si è fatto variare il valore del timer (1000, 10000, 100000, 500000) µs.

Lost Probability \ Timer Value	1000	10000	100000	500000
0	198992,82	200421,39	198992,82	196278,53
5	163800,73	145676,61	111827,34	93701,54
10	90653,64	14342,89	3340,95	637,69
15	31917,18	4771,96	492,47	168,35
18	21919,32	2884,11	243,31	73,7
20	16917,4	2156,35	186,13	51,37
30	5945,95	592,64	64,79	14,25
40	1784,29	171,77	27,89	8,32
80	363,72	36,8	3,59	0,65



Notiamo che all'aumentare della probabilità di perdita, il throughput decresce in maniera più lineare rispetto al caso adattivo, in quanto il tempo di attesa per ritrasmettere un pacchetto è fisso e non dipende dal valore del RTT.

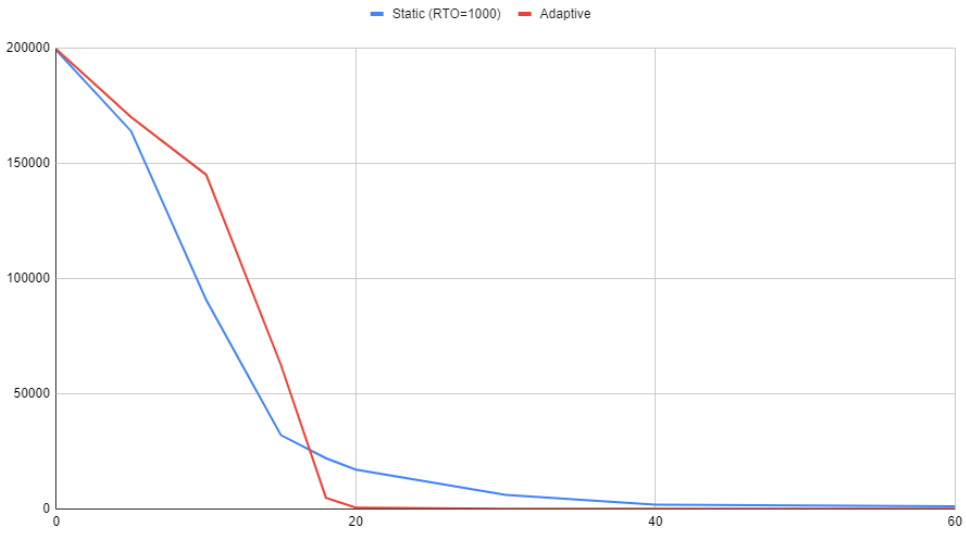
Analizzando i risultati dei test vediamo come con una probabilità di perdita pari a zero il throughput resta lo stesso anche al variare del timer fisso. Infatti non essendoci perdita di pacchetti, non scadrà mai il timer di ritrasmissione (RTT << RTO in locale), e di conseguenza il suo valore risulta ininfluente.

Con valori elevati del timer di ritrasmissione il throughput tende ovviamente ad abbassarsi, in quanto bisognerà attendere più tempo prima di ritrasmettere ogni pacchetto perso.

7 Analisi delle prestazioni e considerazioni finali

Per analizzare l'efficienza della nostra implementazione abbiamo messo a confronto le due casistiche (timer adattivo e statico), con finestra fissata per entrambi a N=8. Per il timer statico abbiamo tenuto conto dei valori ottenuti nel caso migliore di RTO_VALUE = 1000 µs.

N = 8	Static (RTO=1000)	Adaptive
0	198992,82	199431,33
5	163800,73	169966,88
10	90653,64	144967,43
15	31917,18	62422,55
18	21919,32	4703,95
20	16917,4	431,56
30	5945,95	26,15
40	1784,29	9,81
80	363,72	1,03



Possiamo osservare come per tassi di perdita ragionevoli (P<18%) il timer adattivo permetta di raggiungere throughput più elevati rispetto al caso statico. Questo perchè perdendo meno pacchetti, il timer verrà calcolato su valori di sampleRTT più bassi.

All'aumentare della probabilità di perdita si comporta invece meglio il timeout statico, che presenta una decrescita più lineare del throughput, permettendo di mantenere un rate trasmissivo accettabile anche con altissima probabilità di perdita (Thr = 363,72 Kb/s | P=80%).

Tuttavia questo risultato è dovuto al fatto che la perdita dei pacchetti è solo simulata in locale; effettuare numerose ritrasmissioni, anche con alti tassi di perdita, risulta vantaggioso poiché permette di recuperare in minor tempo dai buchi nella finestra di ricezione.

In internet invece un evento di perdita corrisponde generalmente ad uno stato di rete congestionata, nel quale eseguire continue ritrasmissioni risulta controproducente poiché si sovraccarica la rete e/o si manda in overflow il ricevente. Nello scenario reale dunque il protocollo TCP, unito al controllo di flusso e di congestione, funziona meglio con un timeout adattivo che tenga conto dello stato della rete e dello stato del buffer del ricevente.

Un limite della nostra implementazione si trova dunque nella cattiva scalabilità del throughput all'aumentare del tasso di perdita, nel caso di timeout adattivo.

Infatti per probabilità intorno al 17% si ha un peggioramento significativo del rate trasmissivo. Ad esempio, per N=8 e P=15 il throughput è di 62400 KB/s mentre per N=8 e P=18 il throughput cala nettamente a 4700 KB/s.

Tuttavia il protocollo TCP da noi implementato permette di raggiungere throughput elevati anche per tassi di perdita considerevoli. Infatti con una probabilità intorno al 15% tutte le casistiche testate riportano un throughput medio superiore a 20000 KB/s (= 20 MB/s).

Il valore massimo registrato del rate trasmissivo risulta essere 225000 KB/s (=225 MB/s). Questo valore viene raggiunto per probabilità di perdita fino all'8%, e ciò certifica come il nostro software lavori efficacemente anche con tassi di perdita significativi.

Inoltre questo upper bound non viene mai superato a prescindere dai parametri di configurazione scelti, pertanto il collo di bottiglia risulta legato principalmente alle componenti hardware della macchina su cui l'applicazione (client/server) è in esecuzione.