

Reliable UDP

Trasferimento file affidabile su protocollo UDP

Luca Santopadre 0257118

Università di Roma Tor Vergata

Ingegneria Internet e Web 2019/2020

Sommario

[Premessa 2](#_Toc59982013)

[Architettura 2](#_Toc59982014)

[Scelte Progettuali 3](#_Toc59982015)

[Tool utilizzati 3](#_Toc59982016)

[Orientamento alla connessione 3](#_Toc59982017)

[Trasferimento dati affidabile 4](#_Toc59982018)

[Implementazione 4](#_Toc59982019)

[Go-Back-N 4](#_Toc59982020)

[Timeout 4](#_Toc59982021)

[Comandi 5](#_Toc59982022)

[List 5](#_Toc59982023)

[Get 5](#_Toc59982024)

[Put 5](#_Toc59982025)

[Close 6](#_Toc59982026)

[Strutture dati e funzioni principali 6](#_Toc59982027)

[Utilizzo e configurazione 6](#_Toc59982028)

[Manuale 6](#_Toc59982029)

[Parametri di configurazione 7](#_Toc59982030)

[Analisi delle prestazioni 8](#_Toc59982031)

[Timeout 8ms, % errori e finestra variabili 8](#_Toc59982032)

[Finestra 8, % errori e timeout variabili 9](#_Toc59982033)

[Finestra 8, timeout statico, adattivo, %errori variabil 10](#_Toc59982034)

[Conclusioni 11](#_Toc59982035)

# Premessa

Il presente documento intende descrivere ed analizzare il progetto finale del corso di Ingegneria di Internet e web anno 2019/2020.

Lo scopo de progetto è quello di progettare ed 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);

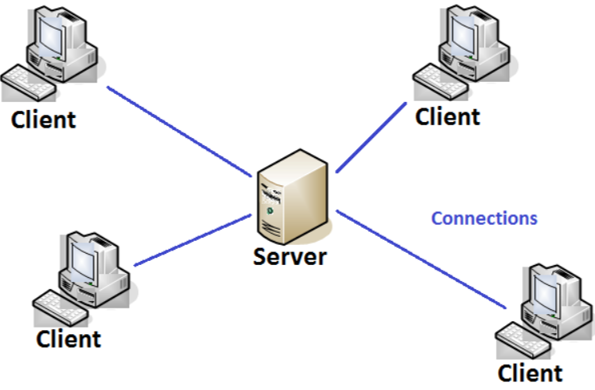
• Il trasferimento file in modo affidabile.

# Architettura

Per l’esercizio di questo software si è utilizzata un’architettura client-server orientata alla connessione.

Il client è un software che utilizzerà l’utente finale allo scopo di scambiare file con il server “connesso”, ed in maniera affidabile.

Il server si occpuerà di gestire le connessioni e dell’invio dei file anche da più client contemporaneamente.



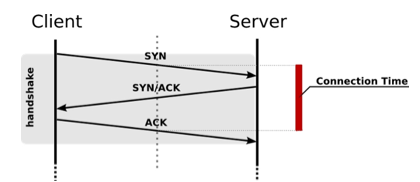
# Scelte Progettuali

## Orientamento alla connessione

Il protocollo UDP utilizzato per questa soluzione sofware, identifica una socket con la sola coppia IP/PORT, pertanto se a sul server utilizzassimo una sola socket per più client, non sarà possibile distinguere a quale processo server indirizzare i pacchetti. Quindi si è scelto di assegnare una nuova socket server ad ogni client connesso.

Per permettere a più client di essere connessi contemporaneamente e quindi avere un server di tipo concorrente, il server dispatcher, all’avvio, crea una socket sempre in ascolto di richieste di connessioni dai client. Una volta ricevuta una richiesta di connessione il server dispatcher eseguirà un “handshake” con il client, e, a connessione stabilita, eseguirà una fork() e delegherà il processo figlio alla creazione di una nuova socket dedicata alla connessione con un unico client.

Viene sfruttata la logica di connessione “handshake a 3 vie”, in dettaglio:



Il server dispatcher al suo avvio inizializza la sua socket e si metterà in ascolto di richieste di connessione da parte dei client tramite la funzione “handshake”, il server è nello stato “SYN wait”. Alla ricezione di un messaggio di “SYN” da parte del client, il server invierà il messaggio di risposta “SYNACK”, il client quindi, una volta ricevuto questo messaggio invia al server il messaggio “ACK”.

Al termine di questo scambio SYN-SYNACK-ACK la connessione può considerarsi stabilita ed il server dispatcher effettuerà una fork() di un nuovo processo, dedicato alla gestione della connessione e dello scambio di messaggi con il client. A questo punto il processo “figlio” invia al client un messaggio “READY” per notificarlo che è pronto a ricevere richieste dal client stesso attraverso la socket dedicata.

## Trasferimento dati affidabile

Il trasferimento di dati affidabile per questa soluzione sfrutta il protocollo Go-Back-N.

Il protocollo GBN è un protocollo pipeline che prevede 2 attori coinvolti: un sender ed un receiver.

Il sender, dopo aver assegnato un numero di sequenza ai pacchetti, può inviarne contemporaneamente fino alla dimensione della sua finestra N, avere quindi fino a N pacchetti “in volo”, cioè senza che essi siano riscontrati con opportuno “ack”. Il sender imposta un timer per l’ultimo pacchetto che ha inviato e non ha ancora ricevuto “ack”, che una volta scaduto, scaturisce un reinvio di tutti i pacchetti inviati nella finestra che non hanno ancora ricevuto l’ack.

Il receiver a sua volta invierà ack cumulativi al sender, se il pacchetto è quello corretto nella sequenza da ricevere, altrimenti invierà di nuovo l’ultimo ack inviato.

## Tool utilizzati

Per sviluppo e test dell’applicativo è stata utilizzata una *Virtual Machine Ubuntu 19.10* in *virtualbox*, *gcc* per la compilazione del codice sorgente, *terminale* linux per il run delle applicazioni, e *IDE Visual Studio Code* per lo sviluppo.

# Implementazione

## Go-Back-N

Per l’implementazione del protocollo GBN sono state utilizzate 2 funzioni principali

* *sendToGBN*, la quale una volta inizializzati inumeri di sequenza, e la finestra di trasmissione, eseguirà il lato sender del protocollo GBN. La ricezione degli ack invece è delegata ad un nuovo thread il quale è in ascolto di ricevere gli ack.
* *recvFromGBN*, che invece eseguirà il lato receiver della trasmissione, inviando ack al sender.

## Timeout

La perdita di pacchetti viene simulata attraverso una funzione che, prendendo come paramtro “p” la probabilità di perdita, decida se inviare o meno il pacchetto. Il congestionamento della rete viene quindi simulato e non ricopre il caso in cui ci siano veri e propri ritardi della rete, difficili da riprodurre se entrambi client e server sono eseguiti sullo stesso terminale.

Il timeout per l’ultimo pacchetto non acked viene prima assegnato attraverso la funzione “startTimer” e quindi gestito tramite segnale ALARM che invocherà la funzione “timeoutRoutine” allo scadere dello stesso. Il timeout, come la probabilità di perdita, il valore massimo e minimo, può essere scelto come valore configurabile e può essere di 2 tipi, statico oppure adattivo.

L’impostazione di *timeout adattivo*, si basa sull’evento di perdita che quindi si può interpretare come un congestionamento della rete. Viene diminuito il timeout di un’unità se la trasmissione è andata a buon fine, mentre viene aumentato se si verifica un evento di perdita.

## Comandi

Il client, dopo essersi connesso al server, mostrerà in output una lista di comandi disponibili da inviare al server (list, get, put, close, di seguito in dettaglio).

L’utente, una volta scelto il comando premerà “invio”, il comando selezionato verrà inviato al server tramite un pacchetto avente payload il comando stesso.

Terminata di eseguire l’operazione, il client mostrerà di nuovo all’utente la lista di comandi disponibili, ed il server tornerà in stato di attesa di ricevere una nuova operazione da eseguire.

### List

Il comando LIST permette al client di ricevere la lista dei file disponibili sul server.

Il server recupererà la sua lista di file disponibili, la quale verrà inserita in un file temporaneo, quest’ultimo quindi inviato al client tramite la funzione “sendToGBN” prima di essere eliminato.

Il client una volta ricevuta la lista, la stamperà a video all’utente.

### Get

Il comando GET permette al client di scaricare un file disponibile sul server.

Se il client richiede un file non esistente su server, quest’ultimo invierà un messaggio “NFOUND” al client informandolo che il file richiesto non esiste.

Se il client richiede un file che ha lo stesso nome di un suo file in locale, verrà mostrato all’utente un messaggio di conferma alla sovrascrittura del file. In caso l’utente sceglie di non sovrascrivere il file, il client invia al server un messaggio “NOVERWITE” terminando la sua operazione e mostrando all’utente la lista di comandi iniziale.

Se il file è presente sul server, e l’utente sceglie di sovrascrivere il file, il server lo invierà al client tramite la funzione “sentToGBN”. Al termine dell’operazione il server si metterà di nuovo in ascolto di una nuova operazione da parte del client.

### Put

Il comando PUT permette al client di inviare un file al server sfruttando la funzionalità di trasferimento affidabile.

Se il file che l’utente sceglie non è presente in locale, verrà mostrato un messaggio di errore all’utente ed il comando non verrà inviato al server.

Se il client vuole caricare sul server un file che ha lo stesso nome di un altro già presente su quest’ultimo, verrà mostrato un messaggio di errore all’utente mostrando di nuovo la lista dei comandi. Si è fatta questa scelta per assicurare sul server la non sovrascrittura di file da parte di client.

Se il client, dopo aver inviato il comando PUT, e dopo la verifica di non esistenza sul server, inizierà la trasmissione del file tramite la funzione “sendToGBN”.

Al termine il client mostrerà all’utente la lista dei comandi iniziale ed il server si metterà in attesa di ricevere una nuova operazione.

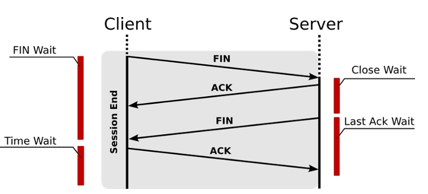
### Close

Il comando CLOSE permette al client di chiudere la connessione con il server in maniera affidabile. Il client utilizzerà la funzione “closeConnectionClient” ed il server “closeConnectionServer”. Per questa operazione viene sfruttata la logica di chiusura di connessione utilizzata in TCP.

Il client invia un messaggio “FIN” al server, il quale risponderà con un messaggio “FINACK” al client.

Il server, a sua volta dopo aver inviato il “FINACK”, invia il suo messaggio di “FIN” al quale il client risponderà con il suo messaggio “FINACK”.

Al termine dello scambio di questi pacchetti, la connessione client-server viene chiusa sia lato server, sia lato client. Il client terminerà la sua esecuzione, il server resterà in attesa di una nuova connessione da un altro client.



## Strutture dati e funzioni principali

- Packet{…}:

#define PAYLOAD ( PKT\_SIZE-sizeof(int)-sizeof(short int) )

typedef struct packet{

    int seq\_num;

    short int pkt\_dim;

    char data[PAYLOAD];

} packet;

Con questa struttura dati viene definito il “pacchetto” con numero di sequenza, la dimensione effettiva del pacchetto, ed il payload.

- sendToGBN():

int sendtoGBN(int socket, struct sockaddr\_in \*receiver\_addr, int N, int lost\_prob, int fd) {

    int i;

    struct timeval endTime, startTime;

    // INIT SOCKET

    sock = socket;

    rcv\_addr = receiver\_addr;

    addr\_len = sizeof(struct sockaddr\_in);

    // INIT ACK'S THREAD

    struct thread\_args t\_args;

    t\_args.sock = sock;

    t\_args.rcv\_addr = rcv\_addr;

    t\_args.addr\_len = addr\_len;

    int ret = pthread\_create(&thread,NULL,recvACK,(void\*)&t\_args);

    // CALCOLO TOTALE PACCHETTI

    file\_dim = lseek(fd, 0, SEEK\_END);

    if(file\_dim%PAYLOAD==0){

        tot\_pkts = file\_dim/PAYLOAD;

    }

    else{

        tot\_pkts = file\_dim/PAYLOAD+1;

    }

    lseek(fd, 0, SEEK\_SET);

    // INIT params and buffers

    srand(time(NULL));

    initParams(N);

    char \*buff = calloc(PKT\_SIZE, sizeof(char) + 8);

    pkts=calloc(tot\_pkts + 1, sizeof(packet) );

    memset(pkts, 0, (tot\_pkts + 1)\*sizeof(packet) );

    // INIT SEQ\_NUM PKTS AND WINDOW

    win\_end = MIN(tot\_pkts, sendBase + WINDOW - 1);

    for(i=sendBase; i<tot\_pkts+1; i++){

        pkts[i].seq\_num = i;

        pkts[i].pkt\_dim=read(fd, pkts[i].data, PAYLOAD);

        if(pkts[i].pkt\_dim==-1){

            pkts[i].pkt\_dim=0;

        }

    }

    //INT TX

    gettimeofday(&startTime, NULL);

    while(tot\_acked<tot\_pkts && err\_count<MAX\_ERR){

        sendWindow();

    }

    if(err\_count==MAX\_ERR){

        printf("File transfer failed (inactive receiver)\n");

        return -1;

    }

    //END TX -> SEND "-1" PKT

    memset(buff, 0, PKT\_SIZE);

    ((packet\*)buff)->seq\_num=-1;

    if(sendto(sock, buff, sizeof(int), 0, (struct sockaddr \*)rcv\_addr, addr\_len) > 0) {

        startTimer(0);

        alarm(0);

        logger("INFO", \_\_func\_\_,\_\_LINE\_\_, "File transfered, time: ");

        gettimeofday(&endTime, NULL);

        double tm=endTime.tv\_sec-startTime.tv\_sec+(double)(endTime.tv\_usec-startTime.tv\_usec)/1000000;

        double tp=file\_dim/tm;

        double kbps= tp/1024;

        char str[20];

        sprintf(str, "%lf,\n", kbps);

        int fd\_res = open("results.csv", O\_CREAT | O\_APPEND | O\_RDWR, 0666);

        write(fd\_res, str, strlen(str));

        close(fd\_res);

        if(kbps < 1000.0){

            printf("%f sec [%f KB/s]\n", tm, kbps);

        }else {

            tp = tp/1024;

            printf("%f sec [%f MB/s]\n", tm, tp/1024);

        }

        return 0;

    }

    logger("ERROR", \_\_func\_\_,\_\_LINE\_\_, "File transfert failed\n");

    return -1;

}

Questa è la funzione principale utilizzata dal sender, dopo aver inizializzato la socket, il thread per la ricezione degli ack, il numero di sequenza dei pacchetti che compongono il file, la finestra di trasmissione, viene invocata la funzione sendWindow() la quale si occuperà di inviare i pacchetti in una finestra di trasmissione, al termine dopo un controllo sugli errori, verrà inviato il pacchetto finale con numero di sequenza “-1” e payload vuoto. Infine stamperà a video il tempo di trasmissione ed il throughput registrato.

- recvACK()

void \*recvACK(void \*arg){

    struct thread\_args \*args = arg;

    int socket\_ack = args->sock;

    struct sockaddr\_in \*rcv\_addr = args->rcv\_addr;

    socklen\_t addr\_len = args->addr\_len;

    int old\_acked;

    int ack\_num = 0;

    int expected\_ack = 1;

    signal(SIGALRM, timeoutRoutine);

    while(tot\_acked<=tot\_pkts){

        if (recvfrom(socket\_ack, &ack\_num, sizeof(int), 0, (struct sockaddr \*)rcv\_addr, &addr\_len) < 0){

            //logger("ERRO",\_\_func\_\_,\_\_LINE\_\_, "Error receiving ack\n");

            err\_count++;

            if(ADAPTIVE) {

                increase\_timeout();

                isTimerStarted = false;

            }

        }else {

            // ACK cumulative

            if(ack\_num >= sendBase && ack\_num >= expected\_ack && ack\_num <= win\_end){

                if(ADAPTIVE) {

                    decrease\_timeout();

                }

                sendBase = ack\_num+1;

                win\_end = MIN(tot\_pkts, sendBase + WINDOW -1);

                tot\_acked = ack\_num;

                expected\_ack = sendBase;

                startTimer(timeoutInterval);

                isTimerStarted = true;

                if (tot\_acked == tot\_pkts){

                    startTimer(0);

                    //logger("INFO", \_\_func\_\_,\_\_LINE\_\_, "Thread for ack end\n");

                    break;

                }

            }

            // ACK duplicated

            else if(ack\_num < expected\_ack){

                if(ADAPTIVE) {

                    increase\_timeout();

                }

            }

        }

    }

}

La funzione recvAck() è quella dedicata alla ricezione degli ack inviati dal receiver ed è eseguita su un thread apposito. Con questa funzione si ricoprono i casi del Go-Back-N quindi ack cumulativo oppure ack duplicato.

- recvFromGBN():

int recvfromGBN(int socket, struct sockaddr\_in \*sender\_addr, int loss\_prob, int fd){

    sock = socket;

    addr\_len = sizeof(struct sockaddr\_in);

    client\_addr = sender\_addr;

    error\_count = 0;

    expected\_seq\_num = 1;

    srand(time(NULL));

    logger("INFO", \_\_func\_\_, \_\_LINE\_\_, "File transfer started...\n");

    while(pkt\_aux.seq\_num!=-1 && error\_count<MAX\_ERR){

        recvPkt(fd);

    }

    memset(&pkt\_aux, 0, sizeof(packet));

    if(error\_count==MAX\_ERR){

        logger("ERROR", \_\_func\_\_, \_\_LINE\_\_, "File transfer error (inactive sender)...\n");

        return -1;

    }

    else{

        logger("INFO", \_\_func\_\_, \_\_LINE\_\_, "File transfer END\n");

        return 0;

    }

}

La funzione si occupa di ricevere i pacchetti dal sender, il controllo nel ciclo while sul numero di sequenza ricevuto deciderà che la trasmissione è terminata, in particolare se il numero di sequenza ricevuto è “-1”.

- handshake():

int handshake (int server\_sock, struct sockaddr\_in\* client\_addr) {

    int control;

    char \*buff = calloc(PKT\_SIZE, sizeof(char));

    socklen\_t addr\_len = sizeof(\*client\_addr);

    // SYN

    logger("INFO", \_\_func\_\_, \_\_LINE\_\_, "SYN wait...\n");

    control = recvfrom(server\_sock, buff, PKT\_SIZE, 0, (struct sockaddr \*)client\_addr, &addr\_len);

    if (control < 0 || strncmp(buff, SYN, strlen(SYN)) != 0) {

        logger("ERROR", \_\_func\_\_, \_\_LINE\_\_, "connection failed (receiving SYN)\n");

        return 1;

    }

    printf("------------------ HANDSHAKE ------------------\n");

    // SYNACK

    logger("INFO", \_\_func\_\_,\_\_LINE\_\_, "SYN received      | <--- |\n");

    logger("INFO", \_\_func\_\_, \_\_LINE\_\_, "SYNACK sent       | ---> |\n");

    control = sendto(server\_sock, SYNACK, strlen(SYNACK), 0, (struct sockaddr \*)client\_addr, addr\_len);

    if (control < 0) {

        logger("ERROR", \_\_func\_\_, \_\_LINE\_\_, "connection failed (sending SYNACK)\n");

        return 1;

    }

    // ACK

    control = recvfrom(server\_sock, buff, PKT\_SIZE, 0, (struct sockaddr \*)client\_addr, &addr\_len);

    if (control < 0 || strncmp(buff, ACK\_SYNACK, strlen(SYN)) != 0) {

        logger("ERROR", \_\_func\_\_, \_\_LINE\_\_, "connection failed (receiving ACK)\n");

        return 1;

    }

    //  ESTAB

    logger("INFO", \_\_func\_\_,\_\_LINE\_\_, "ACK received      | <--- |\n");

    logger("INFO", \_\_func\_\_,\_\_LINE\_\_, "ESTAB connection. |v----v|\n");

    printf("-----------------------------------------------\n");

    return 0;

}

Handshake è la funzione che utilizza il server dispatcher per effettuare la connessione con il client. Viene quindi eseguita una recvfrom per la ricezione del SYN, poi una sendto per inviare il SYNACK ed una recvfrom per la ricezione dell’ACK. Al termine viene considerata la connessione stabilita, stato ESTAB.

- handshakeClient():

void handshakeClient (int client\_sock, struct sockaddr\_in \*server\_addr) {

    int res;

    char \*buff = calloc(PKT\_SIZE, sizeof(char));

    socklen\_t addr\_len = sizeof(\*server\_addr);

    // SYN

    printf("------------------ HANDSHAKE ------------------\n");

    logger("INFO", \_\_func\_\_, \_\_LINE\_\_, "SYN   sent        | <--- |\n");

    res = sendto(client\_sock, SYN, strlen(SYN), 0, (struct sockaddr \*)server\_addr, addr\_len);

    if (res < 0) {

        logger("ERROR", \_\_func\_\_, \_\_LINE\_\_, "SYN failed       |  X-- |\n");

        exit(-1);

    }

    // SYNACK

    logger("INFO", \_\_func\_\_, \_\_LINE\_\_, "SYNACK wait       | ---> |\n");

    memset(buff, 0, sizeof(buff));

    res = recvfrom(client\_sock, buff, strlen(SYNACK), 0, (struct sockaddr \*)server\_addr, &addr\_len);

    if (res < 0 || strncmp(buff, SYNACK, strlen(SYNACK)) != 0) {

        logger("ERROR", \_\_func\_\_, \_\_LINE\_\_, "SYNACK failed    | --X |\n");

        exit(-1);

    }

    // ACK

    logger("INFO", \_\_func\_\_, \_\_LINE\_\_, "ACK   sent        | <--- |\n");

    res = sendto(client\_sock, ACK\_SYNACK, strlen(ACK\_SYNACK), 0, (struct sockaddr \*)server\_addr, addr\_len);

    if (res < 0) {

        logger("ERROR", \_\_func\_\_, \_\_LINE\_\_, "ACK failed       |  X-- |\n");

        exit(-1);

    }

    // ESTAB

    logger("INFO", \_\_func\_\_,\_\_LINE\_\_, "ESTAB connection. |v----v|\n");

    printf("-----------------------------------------------\n");

}

Allo stesso modo del server, il client eseguirà l’handshake, iniziando con l’invio del SYN, la ricezione del SYNACK, e l’invio dell’ ACK. Al termine viene considerata la connessione stabilita, stato ESTAB.

Nella libreria “utils” sono invece state inserite delle funzioni comuni quali:

- logger():

void logger(const char\* tag, const char\* function, const int linenum, const char\* message) {

   time\_t now;

   time(&now);

   printf("[%s | %s | %s:%d]: %s", timeNow(), tag, function, linenum, message);

}

Un semplice log in output a video che stampa in maniera formattata gli eventi accaduti, con il timestamp, un tag, e utile per debug la funzione e la linea di codice dell’esecuzione.

Si può pensare in futuro di creare un file “.log” e di redirezionare l’output sul file invece che a video, per avere traccia anche in futuro degli eventi. Inoltre si potrebbe anche impostare il livello di log desiderato in base al “tag”.

# Utilizzo e configurazione

## Manuale

Per l’utilizzo del software, eseguire i programmi da 2 terminali linux, portarsi nella cartella principale del progetto “reliableUDP”, quindi:

* COMPILARE I SORGENTI

Eseguire il comando ***make,*** il quale genererà gli eseguibili del client e del server, rispettivamente in “out/client.out” e “out/server.out”.

* ESECUZIONE DEL SERVER

Per avviare il server, aprire un terminale ed eseguire il comando ***./out/server.out***

Il server rimmarrà quindi in ascolto di richieste di connessione da parte dei client. Una volta avvenuta una richiesta viene effettuato l’handshake con il client, ed il server sarà in attesa di ricevere sia una richiesta dal client connesso, sia una nuova richiesta di connessione.

Per terminare il server eseguire il comando “Ctrl + c”.

* ESECUZIONE DEL CLIENT

Per avviare il client, aprire un secondo terminale ed eseguire il comando ***./out/client.out***

Il client si connetterà al server attraverso l’handshake e viene visulizzata una lista dei file già disponibili al client, e la lista dei comandi. Si hanno 20 secondi per la scelta del comando da inviare al server, altrimenti viene chiusa la connessione.

Per terminare il client eseguire il comando “4) Close Connection”.

------------------ HANDSHAKE ------------------

[18:29:38.170217 | INFO | handshakeClient:297]: SYN sent | <--- |

[18:29:38.170397 | INFO | handshakeClient:305]: SYNACK wait | ---> |

[18:29:38.170468 | INFO | handshakeClient:314]: ESTAB connection. |v----v|

-----------------------------------------------

WELCOME, you are connected to the server!

--------------------- LOCAL FILE LIST ---------------------

file\_list.txt

sendme.txt

f1mec.jpg

-----------------------------------------------------------

============== COMMAND LIST ================

1) List available files on the server

2) Download a file from the server

3) Upload a file to the server

4) Close connection

============================================

> Choose an operation (20 seconds):

## Parametri di configurazione

Nella libreria common.h sono stati inseriti i parametri configurabili prima dell’avvio del software:

* SERVER\_PORT porta del server dispatcher [default 25440]
* SERVER\_IP indirizzo ip del server dispatcher [default “127.0.0.1”]
* LOST\_PROB percentuale di perdita di un pacchetto [range 0 – 100]
* WINDOW dimensione della finestra del sender [default 8]
* PKT\_SIZE dimensione del pacchetto [default 1500]
* MAX\_ERR numero massimo di errori in ricezione [default 25]
* TIMEOUT\_PKT timeout per la socket dedicata in us
* TIME\_UNIT unità minima di timeout da incrementare o decrementare in us
* MAX\_TIMEOUT valore massimo timeout in us [default 400000]
* MIN\_TIMEOUT valore minimo timeout in us [default 4000]
* ADAPTIVE tipo di timeout, 0=static, 1=adaptive
* REQUEST\_SEC secondi di attesa per digitare il comando dal client
* SELECT\_FILE\_SEC secondi di attesa per digitare nome file in “GET”
* CLIENT\_DIR directory per file del client [default “./files/client”]
* SERVER\_DIR directory per file del server [default “./files/server”]

# Analisi delle prestazioni

Le prestazioni sono state valutate effettuando una media su trasmissioni multiple (file 2MB) calcolando il througput al variare di finestra di trasmissione, percentuale di errore e timeout.

## Timeout 8ms, % errori e finestra variabili

Dai grafici sottostanti si può notare il calo rapido del throughput alla presenza anche di pochi errori, questo dovuto all’inefficienza in termini di prestazione del protocollo GBN. Si è arrivati a registrare un valore massimo di 88Mbps con una finestra di trasmissione di 32.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | % errors | | | | | | | | |
| window | **.** | **0** | **2** | **5** | **10** | **25** | **40** | **50** | **75** |
| **4** | 24263,97 | 7415,01 | 3710,29 | 1574,02 | 516,5 | 256,49 | 166,05 | 57,22 |
| **8** | 52485,71 | 13643,92 | 3806,04 | 1611,43 | 509,78 | 255,94 | 167,37 | 58,33 |
| **16** | 62646,42 | 12665,95 | 3835,31 | 1591,98 | 506,58 | 257,85 | 172,9 | 57,94 |
| **32** | 88151,65 | 13045,65 | 3740,02 | 1635,43 | 514,59 | 256,67 | 169,58 | 58,48 |
| **64** | 86211,36 | 12763,6 | 3850,71 | 1637,18 | 519,76 | 257,99 | 172,27 | 57,26 |

## Finestra 8, % errori e timeout variabili

Mantenendo una finestra fissa di 8 pacchetti in volo, si è fatto variare il timeout e la percentuale di errori, anche qui si evince un forte calo di prestazioni alla presenza di pochi errori.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | % errors | | | | | | | | |
| timeout | **V1** | **0** | **2** | **5** | **10** | **25** | **40** | **50** | **75** |
| **4000** | 53567,28 | 19148,61 | 6460,52 | 2932,52 | 891,42 | 460,2 | 312,18 | 114,68 |
| **16000** | 48918,6 | 8323,45 | 2031,64 | 863,95 | 268,46 | 135,67 | 87,57 | 30,2 |
| **32000** | 44813,07 | 4573,72 | 1071,69 | 452,21 | 141,52 | 67,28 | 48,35 | 15,85 |
| **80000** | 45292,61 | 1933,07 | 424,7 | 183,02 | 58,62 | 28,33 | 19,76 | 6,39 |
| **160000** | 42628,11 | 915,08 | 225,65 | 96,66 | 28,31 | 14,88 | 9,66 | 3 |

## Finestra 8, timeout statico, adattivo, %errori variabil

Con questo caso si è messo a confronto il tipo di timeout, statico oppure adattivo. Analizzando i risultati ottenuti si nota un beneficio nell’usare il timeout adattivo alla presenza di pochi errori, aumentando il throughput, nel caso di 2% di errori, da 4Mbps a 16Mbps, questo dovuto al fatto che in presenza di pochi errori nella rete si predilige un decremento del timeout. Andamento che si inverte già ad arrivare ad un circa 10% di errori, i quali comportano un incremento del timeout elevato.

|  |  |  |  |
| --- | --- | --- | --- |
|  | window | | |
|  | **V1** | **static 32.000** | **adaptive** |
| % errors | **0** | 44813,07 | 46252,53 |
| **2** | 4573,72 | 16297,35 |
| **5** | 1071,69 | 3023,93 |
| **10** | 452,21 | 421,76 |
| **25** | 141,52 | 11,24 |
| **40** | 67,28 | 5,47 |
| **50** | 48,35 | 3,84 |
| **75** | 15,85 | 0,85 |

# Conclusioni