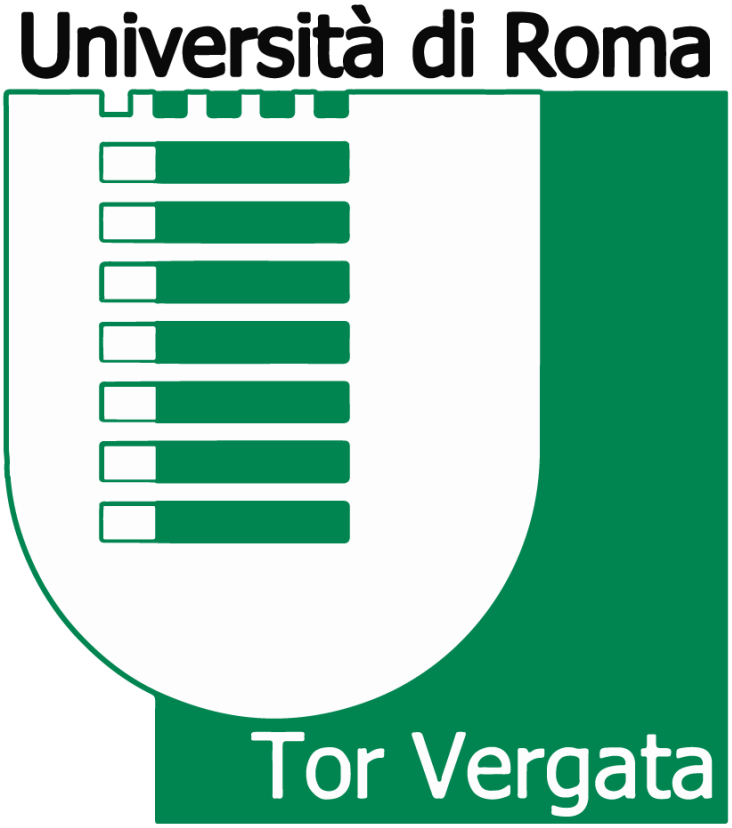
Relazione progetto IIW

Trasferimento file affidabile su UDP

Danilo Sabellico 0268692



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

- 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(nominata nel file settings.h "WINDOW_SIZE").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(nominata nel file settings.h "LOST_PROB").
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

Per questo progetto si e deciso di implementare un server multi processo , che sfrutta prettamente le socket UDP atte a comunicare e a scambiare messaggi fra client e server. Ne verrà stanziata una principale che instaurerà la connessione per poi generarne una altra, il cui scoto è continuare la comunicazione e ad offrire i servizi offerti.

NOTA : questo piccolo preambolo di instaurazione e allocazione di una nuova socket atta alla comunicazione può essere svolta egregiamente stanziando una prima socket TCP per poi spawnare un nuovo processo , chiudere la stessa e creare una socket UDP .

Il principio di asservimento dei servizi implementati è molto semplice . Inoltre come da testo del corso si e deciso di instaurare la connessione tramite il metodo della THREE-WAY-HANDSHAKE, tramite stesso procedimento verrà eseguita anche la fase di close della connessione .
Esplichiamo come avviene la connessione iniziale al connettersi del client :

Passo 1 : Una socket UDP main instanziata avente PORTA PORT-1(vedasi il paragrafo 4.2) in fase di avvio del server entra in stato di LISTEN (ascolto di un possibile pacchetto inviato da un utente intento a connettersi) , questo stato viene indotto dalla API recvfrom (si ricorda che affinché la

chiamata a tale API sia bloccante la socket non dovrà avere nessun timer impostato precedentemente).

PASSO 2 :
Al ricevimento di un primo pacchetto di SYN (sincronizzazione) il server procederà all'invio di un SYNACK ed e ritornerà in fase di ascolto dietro recvfrom questa volta per questa API dovrà contenere un timing di socket in quanto si potrebbe contemplare il caso in cui il pacchetto possa perdersi e quindi fa restare il server in fase di ascolto indefinito con la sua socket principale .

PASSO 3 :
instaurata la connessione il server si procederà inizialmente ad assegnare uno slot del server al client (tale funzionalità che a mio avviso deve essere sempre implementata eviterà il sovraccarico del server da parte di un overflow di utenti e soprattutto ad evitare eventuali attacchi di tipo DDOS atti a creare un DENIAL OF SERVICE) il funzionamento della bitmask verrà spiegata nel paragrafo dove verranno descritte tutte le funzioni implementate nel server .

PASSO 4:
una volta acquisito uno slot si procederà ad incrementare la porta e a creare un processo figlio il quale esso chiuderà la socket main duplicata un fase di spawn e a crearne una nuova , esclusiva per la comunicazione con il client.

A questo punto saremo giunti nel vivo del nostro software dove verranno espletati i servizi offerti dal server.

3 Implementazione dei servizi principali del server

Siamo quindi giunti al nel vivo della comunicazione . Il nostro server entra in un infinite loop e andrà in attesa che il client comunichi un messaggio . Il client d'altro canto si accinge a mandare i primi comandi suggeriti da una piccola legenda iniziale che spiega sinteticamente i servizi offerti. Vediamo i servizi.

3.1 LIST

All'arrivo da parte del client del comando di list il server chiamerà la corrispondente funzione "list_funct" il quale accetta la socket , un campo opt (spiegato di seguito) e una stringa dummy (una stringa inutile questo particolare parametro verrà utilizzato in fase di GET E PUT).Entrati quindi nella funzione ci avvaliamo della libreria di sistema "dirent.h" le cui API fornite ci aiutano a interpretare e ad estrapolare le entry della folder che stiamo puntando tramite il puntatore "DIR *dir" il cui indirizzo viene fornito dall API "opendir". Una volta acquisito l'indirizzo tramite un ulteriore comando e un while verranno lette tutte le entry della cartella target che stiamo puntando e tramite un rapido costrutto inseriremo i nomi delle entry all'interno di un array di puntatori a carattere il quale una volta saturo o terminato il while verranno mandate tutte le stringhe al client

che le graficherà all'utente intento a conoscere il contenuto della cartella del server contenente i file da scaricare .

3.2 GET

Terminata la list e spiegato quindi il suo funzionamento possiamo procedere a spiegare il comando di GET. Questa viene invocata dall'utente quando esso desidera scaricare qualche file disponibile sul server . Il dialogo iniziale prevede che subito dopo l'invocazione del servizio verrà chiesto quale file si desidera ottenere (sapientemente l'utente si spera abbia prima invocato la list per comprendere cosa può ottenere), l'utente inserisce il nome completo del file e il server chiamando la list_funct avente socket il campo opt impostato a 1 e il nome del file (adesso torna utile sia il campo opt che il parametro stringa che precedentemente era "dummy"), la list questa volta avrà il solo scopo di confrontare con i file presenti la stringa in input e una volta ricevuto un match questa darà esito positivo indicando che il file esiste .Avuto esito positivo il server chiamerà una receive atta a sincronizzare l'esecuzione delle librerie SENDER e RECEIVER tra client e server.
A fine trasmissione la funzione di libreria "main_sender" restituirà l'esito dell' operazione e ciò verrà comunicato all'utente tramite un messaggio.

3.3 PUT

essenzialmente la put fa le stesse identiche cose della GET ma evitando il match delle stringhe e passando alla funzione di libreria "main_receiver" la stringa ricevuta dal client . La funzione provvederà a creare il file con il nome passato e a riempirlo con i dati nella sequenza giusta (lo vedremo in seguito quando cominceremo a parlare del vero CORE del nostro software).

3.4 CLOSE

Infine spenderemo 2 paroline per il comando di close questo servizio e utile per generare un corretto logout dell'utente e quindi di conseguenza invocare il three way handshake per la chiusura della socket e la chiusura definitiva del processo . Il 3 way handshake in chiusura si compone di una prima ricezione di un pacchetto di FIN , L'invio della akc packet al client e il conseguente invio del pacchetto di fin da parte del server , infine riscontrato la fin mandata dal server si procederà a fare eventuali free di memorie globali allocate (onde evitare i più classici leak) e infine con l'API "exit" verrà terminato il processo figlio .

4 Organizzazione del software e sue dipendenze

4.1 librerie (custom)

Per pervenire all'esigenza indotta dai comandi di GET e PUT si e deciso di creare un set di librerie esterne che potessero essere usate indistintamente sia dal server che dal client . Il set di librerie sono:
SENDER → Questa libreria ha il compito di inviare il file target
RECEIVER → La controparte del SENDER atta a ricevere il file
UTILITY → Questa libreria contiene tutte le funzioni di servizio utili al funzionamento del server e del client .Un ulteriore incentivo alla modularità del software è stato il riuscire a gestire in modo più efficiente ed ordinato la stesura del progetto in quanto per esperienza si e notato che il debug nonché la modifica di un software "mono-blocco" risulta alquanto pesante per lo sviluppatore . Inoltre, con delle librerie ben fatte e possibile riutilizzare le stesse anche per altri scopi e progetti futuri.

4.2 file header

Nel comparto file del software sono presenti anche dei file di header , alcuni introdotti per dichiarare le firme delle stesse librerie su citate nel server e nel client. Altri file invece hanno delle funzioni ben specifiche e mirate :
1) BASIC.H → questo file contiene solamente le dichiarazioni alle librerie standard di sistema , e la dichiarazione delle librerie descritte nel paragrafo 4.1
2) MESSAGE.H → un piccolo file che contiene tutti i messaggi che l'utente osserva durante tutto l'uso del server , l'esigenza di questo file sovviene dal fatto che ad oggi non ci basta creare del software solamente in una data lingua (l'inglese) ma per aumentare l'usabilità da parte di più community si dichiara un file atto al solo scopo di contenere i testi per poter essere agevolmente tradotto senza dover stare ogni qual volta e rileggere l'intero sorgente e modificare i testi nella lingua giusta . Ci preserviamo però di mantenere in lingua inglese in tutti i testi che riguardano il comparto tecnico del software , tipo: segnalazioni di errori dovuti a controlli inseriti nelle API , etc . Insomma tutto ciò che e tecnico del software rimarrà in inglese in quanto si suppone che chiunque voglia risolvere eventuali problemi sia competente e lo sappia leggere.
3) SETTINGS.H → veniamo adesso al file principale che contiene tutte le direttive principali al funzionamento dell'intero software e della comunicazione fra client e server. Vediamo le voci più importanti contenute in esso:

- PORT= Porta standard del Server
- MAX_PORT= porta massima stanziabile
- MAXCLIENT = numero massimo di porte che si possono assegnare al client
- TIMEOUT = valore del time out statico
- LOST_PROB = Probabilità di perdita dei pacchetti (min. 0 – max. 100)
- WINDOW_SIZE= Dimensione della finestra di trasmissione

- ADAPTIVE = Variabile booleana se 1 la trasmissione avviene con un timeout adativo , altrimenti verrà eseguito il timeout statico
- TIME_UNIT = valore di incremento del time out in caso fosse adattivo
- MAX_TIMEOUT= Valore massimo per il timeout adattivo
- MIN_TIMEOUT = Valore minimo per il timeout adattivo
- MAX_ERR = numero massimo di errori dopo il quale verrà terminata la trasmissione del pacchetto .
- TIMEOUT_CON= timer di un sigalarm atta a terminare il client in caso di inattività per un lungo periodo
- PKT_SIZE = grandezza del payload di ogni pacchetto

In fine nel file sono state dichiarate 2 importanti strutture

1) **struct packet** → una struttura che contiene la dichiarazione di tutto il pacchetto dati , questo contiene le seguenti voci :
   seqNum → numero di sequenza del pacchetto
   win → la finestra a cui appartiene il pacchetto (di solito può essere WINDOW_SIZE o minore in caso fossimo arrivati alla fine del file)
   byte_send → la posizione del pacchetto all'interno dei byte totali del file target
   size → grandezza del payload del pacchetto
   data[PKT_SIZE]→ area di memoria atta a contenere i dati letti del file
   sent → variabile booleana atta a indicare se il pacchetto e stato mandato a no
   received→ variabile booleana atta a indicare se il pacchetto e stato ricevuto dal ricevente
   ack → variabile booleana atta a indicare se il pacchetto e stato anche riscontrato

2) **struct ackPacket** → struttura atta creare un pacchetto di riscontro che il ricevente rimanda al mittente , contiene i seguenti campi
   seqNum → Numero di sequenza associato al pacchetto ricevuto
   size → grandezza del pacchetto che e stato ricevuto
   write_byte -> → la posizione del pacchetto all'interno dei byte totali del file target

5 Selective Repeate

Arriviamo al core del nostro progetto e cominciamo a parlare del protocollo atto a recuperare gli eventuali errori e perdite sui pacchetti inviati e ricevuti . Il Selective Repeat implementato, a differenza di quello studiato, ha un time-out sulla socket, e non sul pacchetto, motivo per il quale non è possibile né stimare il time-out sul RTT, né ritrasmettere esclusivamente i pacchetti scaduti. In parole povere, se scade il time-out vengono ritrasmessi tutti i pacchetti della finestra non ancora riscontrati (con un ACK), diventando quindi più efficiente dal punto di vista energetico rispetto ad un GO BACK N, ma anche meno efficiente dal punto di vista del Throughput rispetto ad un vero Selective Repeat. Adesso cominciamo con lo spiegare sia l'algoritmo del SENDER e poi del RECEIVER.

5.1 Il Sender

La prima funzione che verrà chiamata sarà la main_sender che accetta i seguenti argomenti:
1) la socket (per comunicare con il client) ;
2) una variabile struct che mi definisce la socket e le sue proprietà (usata nella sendto e nella recvfrom);
3) il percorso relativo atto a identificare il file target da leggere e mandare al richiedente.

Il codice inizia con un formatting di tutte le variabili di appoggio questa operazione risulta fondamentale per le successive invocazioni in quanto eventuali dati garbage generati dai precedenti invii potrebbero creare degli undefine behavior(comportamenti indefiniti) da parte del server.

In seguito si aprirà in lettura il file identificato da "pathname" e si procederà a calcolare la grandezza del file tramite una api di libreria standard di sistema chiamata "fseek" in modo da definire la taglia in byte del file e in seguito anche il numero di pacchetti da inviare .Con un piccolo controllo si definirà una finestra di pacchetti (utile nel caso remoto in cui il numero totale dei pacchetti costituenti il file siano inferiori alla finestra di invio preimpostata nel file header "settings.h"), esso poi provvederà a dichiarare un array di puntatori della grandezza di "Window_size " che conterranno l'indirizzo al singolo pacchetto contenente tutte le info già citate precedentemente . Una volta faccio ciò si provvede a inviare la finestra di pacchetti al ricevente e si entrerà in uno stato di attesa dei riscontri (tale attesa avrà una durata limite definita dal timeout impostato nella socket) . Da qui il codice prevede 3 casi fondamentali possibili.

1) <u>il pacchetto inviato viene riscontrato entro lo scadere del timing della socket:</u>
il pacchetto una volta riscontrato verrà sovrascritto da un nuovo slot di dati letti da una api "fread" e verrà messo in stato di attesa pronto per essere inviato (per delineare al meglio le finestre di invio dei pacchetti adotteremo un count_s che traccia la finestra di invio dei pacchetti);

2) il timing della socket scade :
non avendo la possibilità di creare un timing inerente al solo pacchetto inviato il server si limiterà a cercare i pacchetti non ancora riscontrati all'interno della finestra di scorrimento e ritrasmetterli al client.

<u>3) arriva un ack inerente ad un pacchetto già riscontrato :</u>
Si provvederà ad ignorare l'ack doppio e a continuare l'esecuzione ritornando in fase di receive. Il loop principale della "wait_ack" si basa principalmente sul tracciamento e riscontro dei pacchetti . Il loop terminerà solo dietro 2 condizioni fondamentali

1) Il caso in cui i pacchetti siano stati tutti riscontrati dal client e quindi si assume che il trasferimento del file abbia avuto successo.

2) In caso di errori ripetuti o troppi time-out scaduti da parte della secket sender.
   a. La mole di errori ripetuti ha come fattore scatenante l'alta probalilità di errore
   b. La mole di time out scaduti avviene per finestre di time-out troppo corte oppure di finestra di pacchetti troppo alta per ricevere un ACK.

5.2 Il Receiver

Parliamo della controparte del nostro software il cui scopo è ricevere i pacchetti e scriverli in ordine e quindi ripristinare il file oggetto di trasmissione . Anche qui la prima parte del codice si limita a fare il formatting delle variabili globali , aprire un nuovo file vuoto che sarà popolato dai dati trasmessi in seguito verrà chiamata la funzione "initWindowSeq" che crea la finestra della grandezza di "WINDOW_SIZE" . Una volta inizializzata la finestra di trasmissione procederemo a chiamare la "receive_data_start" che avrà il compito di ricevere e scrivere pacchetti. Giungiamo quindi al cuore della receiver che : riceve pacchetti , scrive nel file e manda riscontri. Essa è composta da 3 principali casi possibili (di cui il caso 2 e 3 sono stati uniti)

1) viene ricevuto un nuovo pacchetto :
   esso verrà immediatamente contrassegnato ad 1 la parte del pacchetto di receive e in seguito si entrerà in un while atto a scrivere il pachetto nel file aperto in principio e verrà riscritto lo stesso con un "nullPkt"(pacchetto speciale atto ad indicare che e idle e pronto a ricevere nuovi dati da una finestra di pacchetti successiva) . Infine verrà trasmesso l'ack di riscontro al mandatario indicandogli che il pacchetto e stato ricevuto correttamente .
   Qualche riga merita di essere spesa per parlare di come i pacchetti possano essere scritti in sequenza , mi sono avvalso di un piccolo while e un contatore indipendente che tiene traccia dell'ultimo pacchetto scritto in memoria. Questo permetterà di scrivere la corretta sequenza di pacchetti e quindi evitare la corruzione del file(dovuta ad una scrittura fuori sequenza) .
   Facciamo un piccolo esempio :
   Viene trasmessa una finestra di 8 pacchetti , e supponiamo che venga perso il pacchetto 5 . il receiver scriverà ipacchetti 1,2,3,4 ricevuti correttamente e di conseguenza verranno riscritti con il nullpkt indicando che sono idle , inoltre verranno mandati gli ack dei pachetti 6 , 7 , 8 però a differenza dei primi 4 essi non verranno sovrascritti in quanto il recever saprà che gli manca il pacchetto 5 e deve chiudere la finestra scrivendo la sequenza esatta 5,6,7,8. Il sender quindi al non ricevimento del pacchetto 5 lo ritrasmetterà insieme alla trasmissione dei successivi 4 pacchetti(appartenenti alla nuova finestra) , il receiver riceverà esattamente i pacchetti 5,1 ,2,3, 4 che procederà a riscontrarli con un ack. Il receiver a sua volta avrà la diponibilità dei pacchetti 5,6,7,8,1,2,3,4 (qui si vedrà l'utilità del while e del contatore indipendente che farà scorrere la finestra fino a quando non troverà il primo slot della finestra dei pacchetti che conterrà il nullpkt) che seguiranno i destini dei primi 4 pacchetti correttamente scritti ad inizio esempio. Si e volutamente affrontare l'esempio più semplice in quanto a parole risulta complicato spiegarlo .

2) viene ricevuto un pacchetto già riscontrato :
   il receiver provvederà a mandare un ACK e tornare nella receive .
3) viene ricevuto un pacchetto gia scritto in memoria (appartenente a vecchie finestre):
   anche qui si provvede a mandare un ack e tornare nella receive.
   Una volta quindi terminata anche la ricezione dei pacchetti torneremo nel main receiver e verrà mandato un pacchetto speciale atto a indicare che siamo giunti alla fine.

5.3 Timeout adattivo

Non avendo la possibilità di perdita di pacchetti, al fine di simulare una situazione di questo tipo, sono state implementate funzioni che gestiscono la perdita in modo randomico .

```
int correct_send() { // funzione che imposto in modo che ogni pacchetto abbbia la stessa probabilita di perdersi
// il seme randomico deve essere calcolato su la base dei microsecondi e non
// con time(NULL) in quanto quest ultima quantizza fino al secondo e non al microsecondo.
    struct timeval t1;
    gettimeofday(&t1 , NULL);
    srand(t1.tv_usec*t1.tv_usec);
    int randint = rand()%100+1;
    if(LOST_PROB <randint) {
        return 1;
    }
    return 0;
}
```

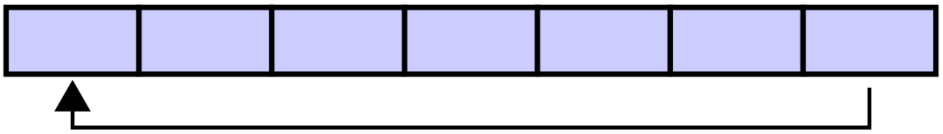
La gestione del timeout adattativo passa per le due funzioni decrease_timeout e increase_timeout. L'idea di base è quella di diminuire il timeout della recvfrom nel caso in cui la trasmissione vada a buon fine, oppure aumentarlo nel caso in cui sia avvenuta una perdita. Questo perché, se esso scade , si assume che la rete sia congestionata, di conseguenza per evitare ritrasmissioni si aumenta il time-out. Nel caso in cui la trasmissione del pacchetto avvenga correttamente, si assume che la rete non sia intasata, e di conseguenza si aumenta per evitare ritrasmissioni. Il time-out viene aumentato o diminuito di un valore fisso a TIME_UNIT. Inoltre si ha la possibilità di aumentare il time-out fino a MAX_TIMEOUT e diminuirlo fino a MIN_TIMEOUT, che sono i valori massimi e minimi possibili (configurabili) a cui si può arrivare.

```
int get_timeout(int sockfd){
    // Restituisce l'attuale valore del timeout della socket
    struct timeval time;
    int dimension = sizeof(time);
    time.tv_sec = 0;
    time.tv_usec = 0;
    if(getsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO,(char*)&time, &dimension) < 0) {
        perror("getsockopt error");
        exit(-1);
    }
    return time.tv_usec;
}
void set_timeout(int sockfd, int timeout) {
    // Imposta il timeout della socket in microsecondi
    struct timeval time;
    time.tv_sec = 0;
    time.tv_usec = timeout;
    if(setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, (char*)&time, sizeof(time)) < 0) {
        perror("setsockopt set_timeout");
        exit(-1);
    }
}
void decrease_timeout(int sockfd){
    // Decrementa il valore attuale del timeout in caso di timeout adattativo
    int timeout;
    timeout = get_timeout(sockfd);
    if(timeout >= MIN_TIMEOUT + TIME_UNIT){
        set_timeout(sockfd, timeout - TIME_UNIT);
    }
}

void increase_timeout(int sockfd){
    //Incrementa l'attuale valore del timeout in caso di timeout adattativo
    int timeout;
    timeout = get_timeout(sockfd);
    if(timeout <= MAX_TIMEOUT - TIME_UNIT){
        set_timeout(sockfd, timeout + TIME_UNIT);
    }
}
```

6 L’idea dietro la finestra dei pacchetti

L ‘ idea che si cela dietro l’algoritmo di invio e ricezione di un file è molto semplice.
Si è adoperato sempre il medesimo array inizializzato all’inizio del esecuzione per entrambi gli attori in gioco, alla ricezione di un pacchetto lo slot dell’array verrà sovrascritto indicando che è disponibile a ricevere un altro pacchetto, il sender riscriverà invece lo slot del suo array solamente dopo che ha ricevuto il riscontro dal ricevente tramite ack che identifica il pacchetto mandato . Il maggiore vantaggio subito riscontrabile da questo approccio e che verrà riutilizzata sempre le medesima area di memoria ciò permettendo di mantenere la performance del codice anche ad un alto bacino di utenza concorrente (di default il bacino di utenza ospitabile nel server e dichiarato dalla variabile MAXCLIENT).



7 Limitazioni Riscontrate – Ed eventuali

Le limitazioni riscontrate sono state :

1) Il non poter impostare un timer specifico per ogni pacchetto (come in realtà viene fatto);
2) La perdita dei pacchetti viene impostata su ogni singolo sendto del receiver e del sender. Inoltre come e giusto che sia fatto il codice e stato scritto in modo da non poter avere nessun feedback nella trasmissione del pacchetto.
3) Impossibilità di fare un Thread per ogni invio, perché bisognerebbe generare un numero di Thread grandissimo perdendo molto tempo, ma soprattutto perché la variabile errno che viene modificata dalla recvfrom()(l’ errore di cui parliamo e EINVAL cioè nessun pacchetto trovato in socket) e che permette di controllare il time-out, sarebbe in comune tra tutti i thread, e la si dovrebbe quindi proteggere con un mutex, trasformando di nuovo il Selective Repeat in uno Stop & Wait
4) Impossibilità di fare fork(), perché sebbene risolverebbe il problema della variabile errno, richiederebbe uno sforzo e dei tempi eccessivi per l’istanziazione, il che avrebbe ridotto le prestazioni all’aumentare della finestra invece di migliorarle
5) Una limitazione riscontrata è quella della generazione di alcuni errori all’aumentare della finestra di ricezione . Per comprendere questo problema dobbiamo prima spiegare che la socket udp non e altro che un canale bidirezionale la cui lettura segue il modello di una coda FIFO . I pacchetti permangono generalmente su questa socket finchè non verranno letti da qualsiasi client che sia in grado di ascoltarla (tale concetto e deducibile dalla definizione

stessa di SOCKET UDP). Dai test effettuati nel software impostando una finestra di pacchetti pari a 128 e probabilità di errore del 0%, viene generato 1 errore ogni 333000 Byte inviati. L'errore che riscontriamo deriva da una scadenza del timeout della socket in fase di ricezione dell'ack. Rimane cmq oscura tale sintomatologia visto che l errore ha dei gap troppo lunghi perché si verifichi il che genera una scorrelazione dovuta al invio e trasmissione e riscontro del pacchetto . Non si esclude l'ipotesi che con finestre troppo grandi venga introdotta una sorta di latenza dovuta alla scrittura del file o di un eventuale saturazione della coda dei pacchetti trasmessi che limita la risposta del receiver .

8 Installazione e Manuale d'uso (con documentazione fotografica)

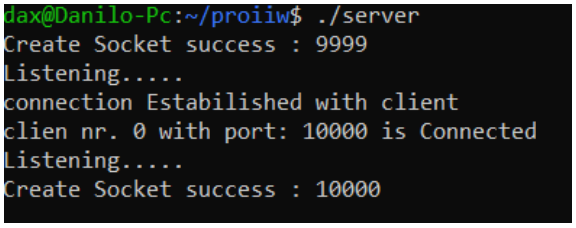
L'installazione e molto semplice in quanto occorre

1) andare su la cartella dove e situato il sorgente compilato
2) aprire un terminale in questa cartella (generalmente con il tasto destro del mouse e scegliendo "apri nel terminale ") .
3) digitare il comando "make" nel terminale .

L' uso del software viene eseguito allo stesso modo come descritto nel punto 1 e 2 su citati però con la sola differenza che anziché aprire un solo terminale se ne dovranno aprire 2 nel caso noi fossimo interessati ad eseguire lo il software su la stessa macchina. Una volta aperti i terminali digitare i seguenti comandi :

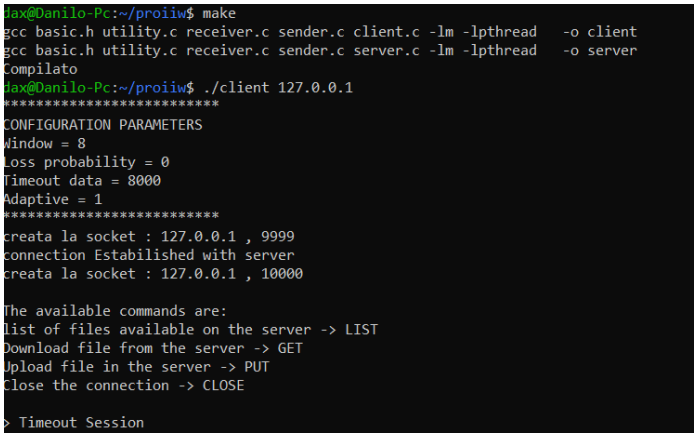
./server

nel caso volessimo avviare il server; (Terminale 1)



I. ./client <indirizzo IP del server >

Nel caso volessimo connetterci ad un server già in esecuzione; (Terminale 2)



Piccola nota per il client:

Se ci si trova su la stessa macchina dove è in esecuzione il server, dovremmo specificare l'indirizzo IP "127.0.0.1". Nel caso fossimo su 2 macchine distinte ma sulla medesima sottorete (NAT) digiteremo l'IPv4 associato alla macchina che sta eseguendo l'applicativo server ad esempio "192.168.x.x".

9 Piattaforme utilizzate per sviluppo e testing .

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

- Ubuntu 22.04 LTS / 32GB RAM / i7 4th gen, compilatore gcc;
- Ubuntu 20.04 LTS / 32GB RAM / i7 4th gen, compilatore gcc, emulato dal Windows Subsystem for Linux;

Per lo sviluppo software si e usato un più comune blocco note in questo caso “kate” della suite appartenente alla distro derivata kubuntu e il più classico terminale (Konsole) .

Il testing e stato eseguito mandanndo il loop sia il client che il software e facendo inviare lo stesso file di taglia circa 2MB 10 volte per ogni elemento costituente la tabella di sotto raffigurate . In seguito tramite excel del pacchetto office sono stati creati grafici e calcolate le medie sul throughput generato dagli invii.

10 Risultati ottenuti

Come gia anticipato Il testing delle performance del software sono stati calcolati su la base di un file mp3 della grandezza di circa 2 MegaByte (1.954.212 Byte) ritrasmesso 10 volte per ogni probabili tà di errore descritta nella tabella , tale pratica di testing è stata ripetuta variando la finestra di pacchetti “Window_Size” .
I vari test eseguiti che troveremo nelle seguenti pagine sono :

1) Test del TIMEOUT ADATTIVO con con:
   Timeout Base : 8000 microsecondi
   Timeout Massimo : 80000 microsecondi
   Unità incrementale : 4000 microsecondi

2) Test del TIMEOUT STATICO con con:
   Timeout minimo : 4000 microsecondi
   Timeout Massimo : 80000 microsecondi

3) Test di cumulazione dell’errore all’aumento della finestra di trasferimento e con un timeout adattivo (tale idea su questo test e nata dall’esigenza di comprendere il perché con una finestra di circa 128 pacchetti essa generi un throughput dimezzato seppur avendo un probabilità di errore nulla). (questo grafico e perfettamente coerente con l’abbattimento del throughput visto nella tabella alla voce 1 .

10.1 Risultati ottenuti tramite TIMEOUT ADATTIVO

TIMEOUT ADATTIVO: 8000 – 80000 usec		FINESTRA				
		8	16	32	64	128
Prob Perdita	0%	34625.34 Kb/s	35602.67 Kb/s	34971.78 Kb/s	32822.13 Kb/s	16168.72 Kb/s
	5%	1670.61 Kb/s	2394.67 Kb/s	3653.44 Kb/s	6061.48 Kb/s	7704.30 Kb/s
	10%	965.88 Kb/s	1546.64 Kb/s	2318.92 Kb/s	3808.67 Kb/s	5816.49 Kb/s
	15%	695.77 Kb/s	1078.24 Kb/s	1831.08 Kb/s	2836.82 Kb/s	4606.03 Kb/s
	20%	521.22 Kb/s	827.01 Kb/s	1404.64 Kb/s	2348.13 Kb/s	3766.66 Kb/s
	25%	373.66 Kb/s	622.92 Kb/s	993.31 Kb/s	1827.67 Kb/s	2970.95 Kb/s
	30%	312.59 Kb/s	500.43 Kb/s	842.40 Kb/s	1495.68 Kb/s	2421.00 Kb/s
	40%	187.29 Kb/s	339.71 Kb/s	518.07 Kb/s	1037.10 Kb/s	1778.96 Kb/s
	60%	12.99 Kb/s	41.44 Kb/s	135.48 Kb/s	312.31 Kb/s	589.83 Kb/s
	80%	2.88 Kb/s	4.59 Kb/s	7.49 Kb/s	17.72 Kb/s	54.92 Kb/s

Grafico delle prestazioni raffigurate in tabella

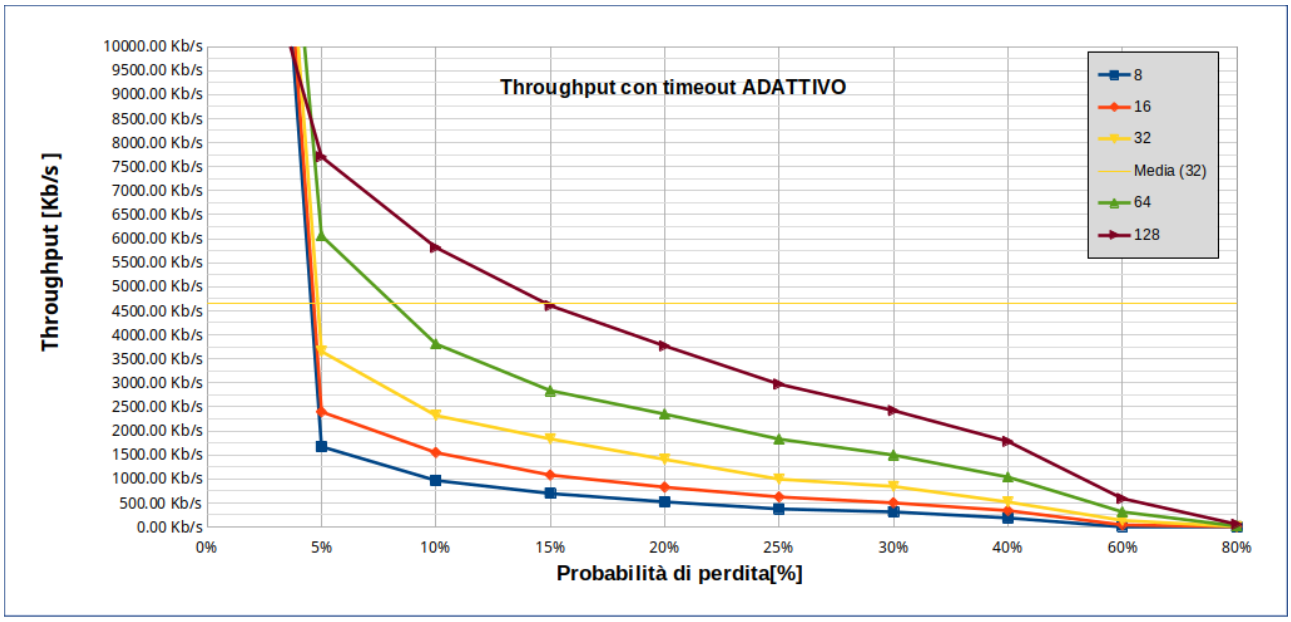
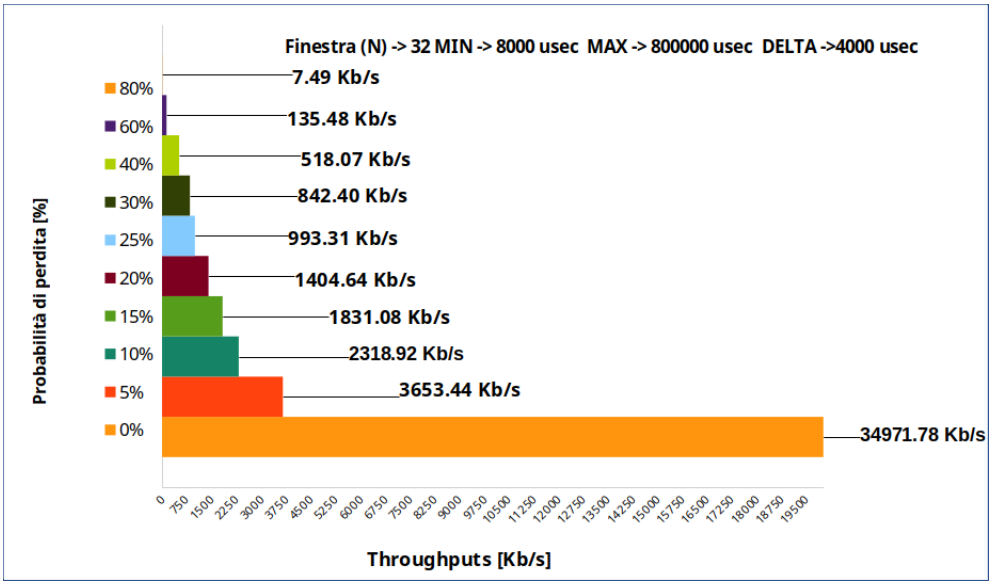


Grafico del throughput per la finestra di 32 pacchetti



10.2 Risultati ottenuti tramite TIMEOUT STATICO

TIMEOUT STATICO & FINESTRA N = 32		TIMEOUT in microsecondi					
		4000 (min value)	8000	16000	32000	64000	80000
Prob Perdita	0%	39653.63 Kb/s	32443.26 Kb/s	28928.45 Kb/s	34412.50 Kb/s	42966.73 Kb/s	36106.52 Kb/s
	5%	4727.65 Kb/s	3576.49 Kb/s	2490.50 Kb/s	1277.62 Kb/s	835.57 Kb/s	685.13 Kb/s
	10%	3500.83 Kb/s	2502.25 Kb/s	1768.88 Kb/s	909.38 Kb/s	525.79 Kb/s	425.64 Kb/s
	15%	2714.11 Kb/s	1914.10 Kb/s	1191.13 Kb/s	655.45 Kb/s	348.69 Kb/s	283.20 Kb/s
	20%	2206.86 Kb/s	1512.28 Kb/s	892.03 Kb/s	585.37 Kb/s	267.25 Kb/s	254.48 Kb/s
	25%	1694.11 Kb/s	1068.69 Kb/s	690.16 Kb/s	382.73 Kb/s	218.91 Kb/s	164.52 Kb/s
	30%	1304.02 Kb/s	926.47 Kb/s	562.95 Kb/s	309.86 Kb/s	170.61 Kb/s	132.60 Kb/s
	40%	830.81 Kb/s	578.49 Kb/s	355.41 Kb/s	204.92 Kb/s	108.62 Kb/s	88.31 Kb/s
	60%	324.29 Kb/s	221.31 Kb/s	133.52 Kb/s	75.07 Kb/s	39.99 Kb/s	32.43 Kb/s
	80%	75.57 Kb/s	50.82 Kb/s	30.73 Kb/s	17.16 Kb/s	9.09 Kb/s	7.36 Kb/s
	90%	18.94 Kb/s	12.70 Kb/s	7.62 Kb/s	4.25 Kb/s	2.25 Kb/s	1.35 Kb/s

Grafico delle prestazioni raffigurate in tabella

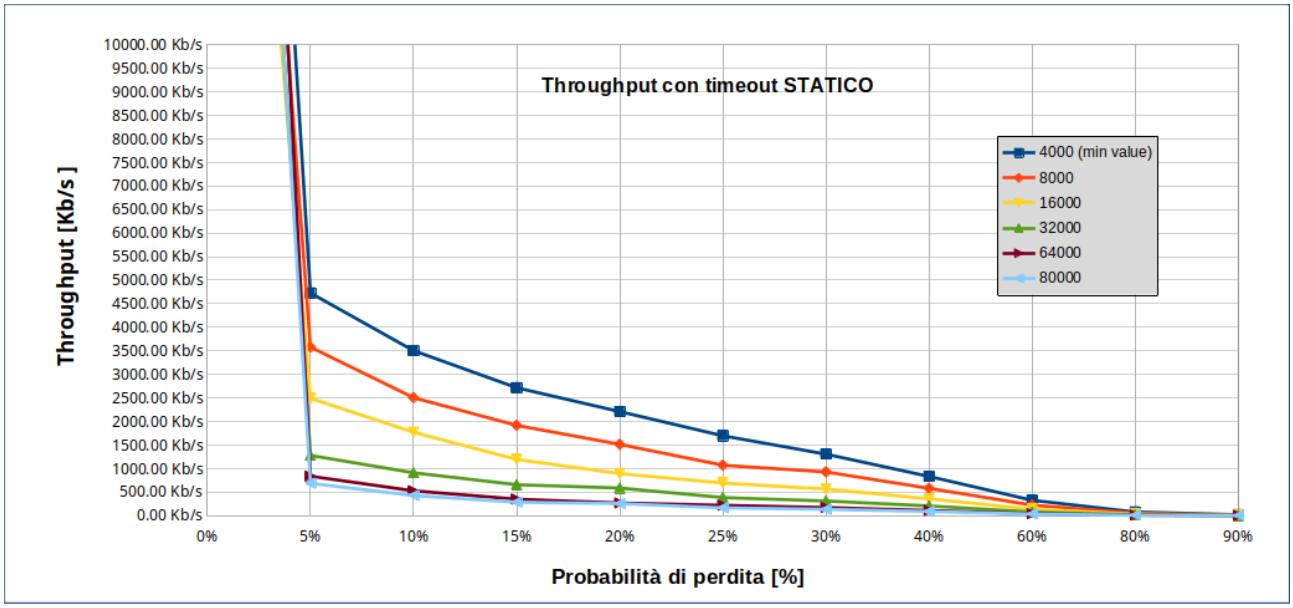
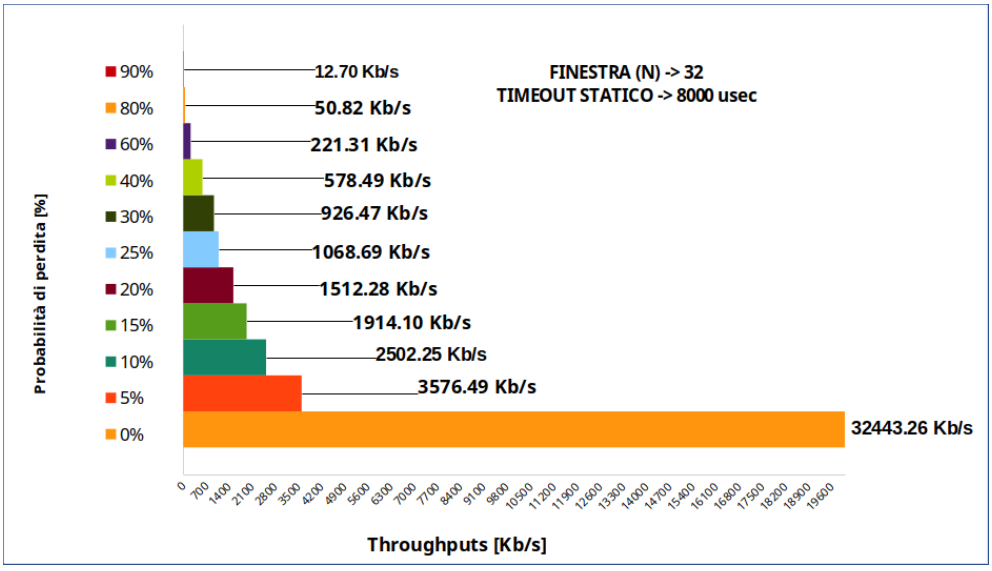


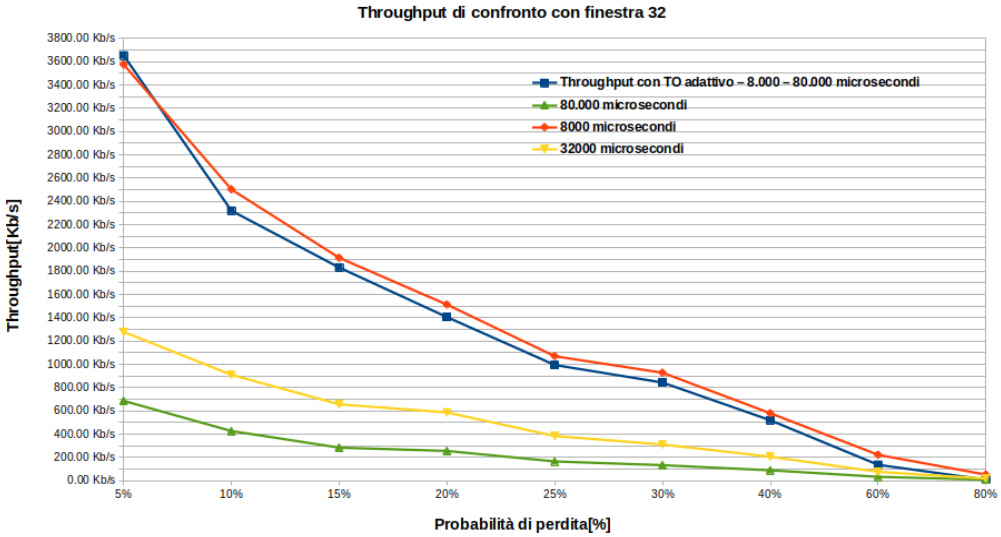
Grafico del throughput per la finestra di 32 pacchetti

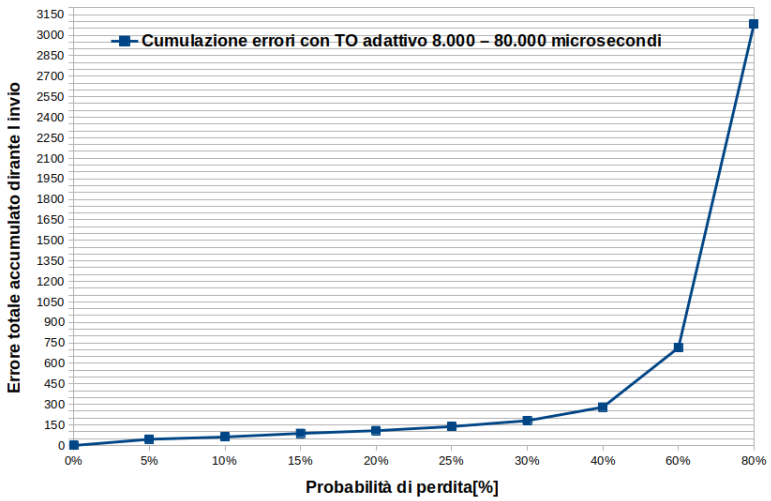


10.3 Risultati sul confronto fra time-out statico e adattivo .

Di seguito vediamo il confronto frà time-out statico e time-out adattivo a parità della stessa finestra di invio .

Tabella di confronto fra timeout statico e adattivo e comulazione d errore						
Finestera : 32		Cumulazione errori con TO adattivo 8.000 – 80.000 microsecondi	Throughput con TO adattivo – 8.000 – 80.000 microsecondi	Prestazionei con vari TO statici		
				8000 microsecondi	32000 microsecondi	80.000 microsecondi
Prob Perdita	0%	0	34971.78 Kb/s	32443.26 Kb/s	34412.50 Kb/s	36106.52 Kb/s
	5%	34 – 45	3653.44 Kb/s	3576.49 Kb/s	1277.62 Kb/s	685.13 Kb/s
	10%	53 – 62	2318.92 Kb/s	2502.25 Kb/s	909.38 Kb/s	425.64 Kb/s
	15%	73 – 88	1831.08 Kb/s	1914.10 Kb/s	655.45 Kb/s	283.20 Kb/s
	20%	89 - 108	1404.64 Kb/s	1512.28 Kb/s	585.37 Kb/s	254.48 Kb/s
	25%	128 – 138	993.31 Kb/s	1068.69 Kb/s	382.73 Kb/s	164.52 Kb/s
	30%	164 - 181	842.40 Kb/s	926.47 Kb/s	309.86 Kb/s	132.60 Kb/s
	40%	257 – 280	518.07 Kb/s	578.49 Kb/s	204.92 Kb/s	88.31 Kb/s
	60%	693 - 715	135.48 Kb/s	221.31 Kb/s	75.07 Kb/s	32.43 Kb/s
	80%	2980 - 3082	7.49 Kb/s	50.82 Kb/s	17.16 Kb/s	7.36 Kb/s

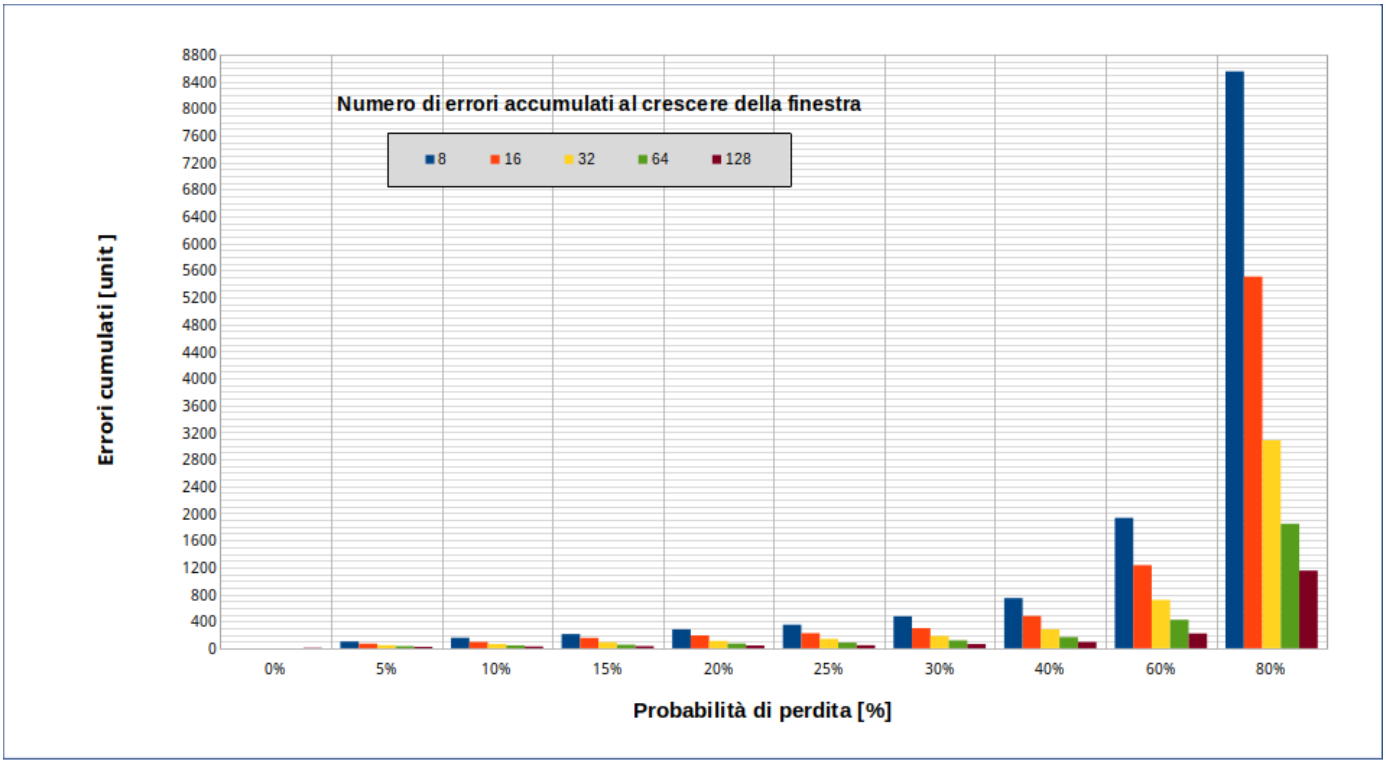




10.4 Risultati su la cumulazione degli errori

Tabella degli errori cumulati durante l invio del file alle variae probabilità di errore		FINESTRA				
		8	16	32	64	128
Prob Perdita	0%	0	0	0	0	6
	5%	85 – 99	56 – 67	34 – 45	24 - 30	13 – 20
	10%	127 – 155	86 – 92	53 – 62	32-42	23 – 25
	15%	184 – 209	122 – 153	73 – 88	43 – 51	26 - 30
	20%	240 - 279	169 - 189	89 - 108	62 – 69	32 – 39
	25%	331 – 348	207 – 222	128 – 138	74 - 82	42 – 48
	30%	444 - 472	285 – 297	164 - 181	98 - 116	55 – 61
	40%	681 – 742	453 – 477	257 – 280	151 - 163	87 - 92
	60%	1889 - 1932	1202 - 1230	693 - 715	374 – 421	140 – 218
	80%	7963 – 8548	5028 - 5507	2980 - 3082	1682 - 1844	899 – 1150

Grafico della cumulazione degli errori al crescere della probabilità di errore



11 Analisi delle prestazioni

I grafici mostrano come i risultati previsti sono coerenti con quelli raccolti, in particolare si evince:

1) Dalla tabella degli errori cumulativi vediamo come effettivamente all'aumentare della finestra, la quantità di errori diminuisca significativamente anche per alte probabilità di errore rendendo così il throughput ancora su soglie accettabili anche in fase di canali molto degradati.
2) Dalla tabella del throughput adattivo e dai grafici vediamo invece come il throughput aumenti all'aumentare della dimensione della finestra e diminuisca all'aumentare della probabilità di perdita.

Non è possibile invece mostrare come, con il time-out adattativo, diminuiscano sia il throughput sia la % di fallimenti. Questo è sicuramente vero, dato che in caso di congestione il time-out aumenta e quindi il throughput diminuisce di conseguenza, ma, avendo più tempo prima del time-out, eventuali pacchetti giunti in ritardo vengono bufferizzati e riscontrati, diminuendo il numero di errori consecutivi. In questo progetto però purtroppo si simula la congestione semplicemente non inviando i pacchetti, quindi non vi è alcuna possibilità di misurare gli effetti di un loro ritardo.

NOTA finale :
Il software e stato sviluppato adottando l'interfaccia di LOOPBACK del device di sviluppo pertanto risulta non testato su interfacce hardware esterne con basso indice prestazionale come raspberry e affini .

12 Esempio di funzionamento

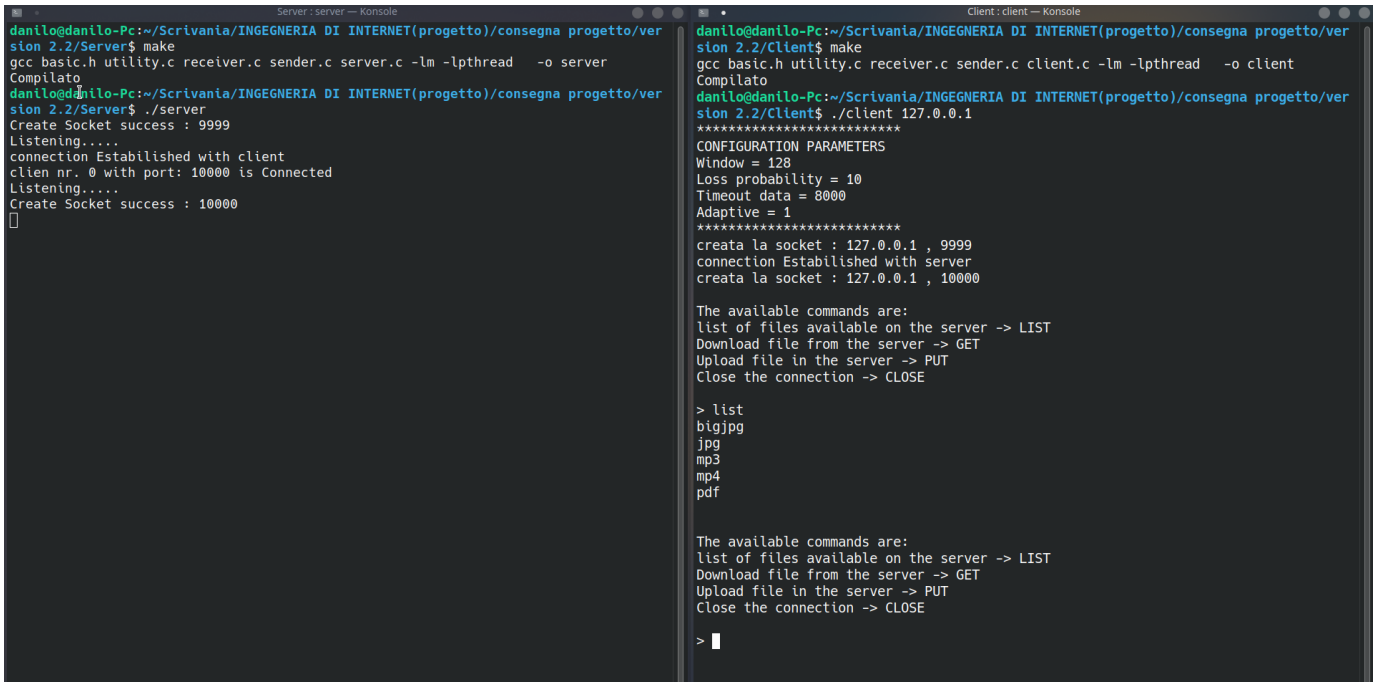
Si vuole adesso far vedere un esempio applicativo

Sono qui caricat i video dimostrativi presso i seguenti link :
https://www.youtube.com/watch?v=RC4n2twdtr0 → video dimostrativo 1
https://www.youtube.com/watch?v=Q1Kz8pqXSDU → video dimostrativo 2

1) Compilazione : aprendo 2 terminali. Il primo dentro la cartella client e il secondo nella cartella server (ovviamente dove risiedono i sorgenti) e poi si e provveduto a lanciare il comando di **make** .

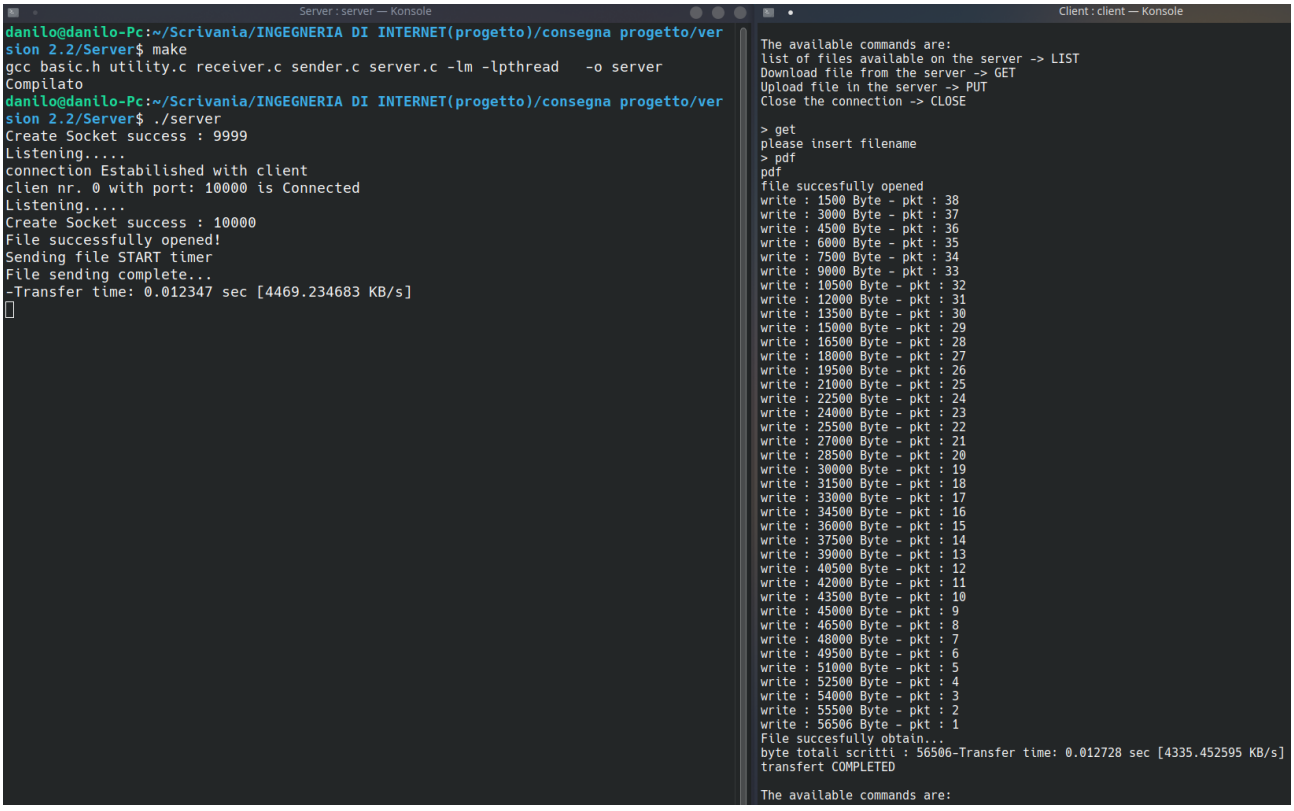
2)adesso verranno avviati server e client
lancio del comando **./server** nel terminale a SX
lancio del comando **./client 127.0.0.1** nel terminale DX

3) nella schermata sottostante il client ha provveduto a lanciare un primo comando di **list** perché interessato a capire che file può scaricare dal server .
(osservare a pagina successiva un immagine esplicativa).

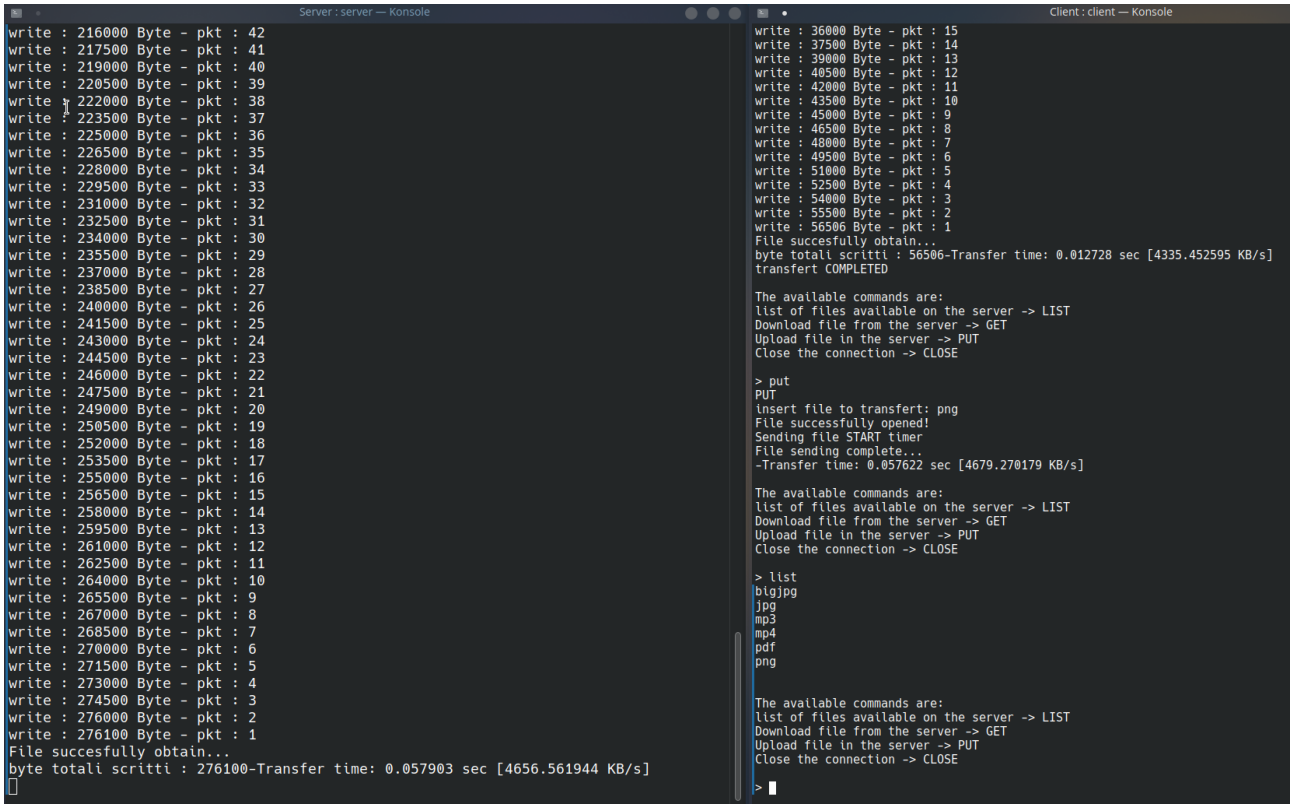


4) il client adesso lancia il comand **get** viene chiesto dal server il nome del file che desidera e inserisce il nome **pdf** (per una migliore comprensione sono stati rinominati tutti i file con le loro estensioni).

Vediamo che a fine trasmissione si ritorna nel menù di scelta dei comandi e viene stampato il tempo di trasmissione con relativo throughput in kb/s



5) il cient adesso desidera caricare nel server un in immagine png (all'occorrenza rinominata solamente png) tramite il comandi di **put.**
6) in seguito il client (paranoico) vuole essere sicuro che l immagine sia effettivamente caricata (nonostante il messaggio del corretto trasferimento), rimanda quindi un comando di **list** .



7) il nostro amico client ormai ha eseguito le operazioni che desiderava e quindi chiude la trasmissione lanciando il comando di **CLOSE**
8) Il server segnala al client il corretto log-out e stampa nel suo terminale la corretta disconnessione con il client . (per una questione didattica si sono limitate le stampe del server ma in un contesto prettamente reale potranno tranquillamente essere tolte).