Immagine che contiene computer

Descrizione generata automaticamente

Trasferimento file su UDP

Ingegneria di Internet e del Web | Settembre 2020

Caliandro Pierciro

Falcone Gian Marco

Minut Robert Adrian

Sommario

[Traccia del progetto 3](#_Toc51154197)

[Funzionalità del server 3](#_Toc51154198)

[Funzionalità del client 3](#_Toc51154199)

[Trasmissione affidabile 4](#_Toc51154200)

[Architettura e scelte progettuali 5](#_Toc51154201)

[Server 5](#_Toc51154202)

[Trasmissione 5](#_Toc51154203)

[Implementazione 7](#_Toc51154204)

[Comunicazione affidabile 7](#_Toc51154205)

[Timeout 11](#_Toc51154206)

[Controllo di flusso 13](#_Toc51154207)

[File di log 16](#_Toc51154208)

[Limitazioni riscontrate 16](#_Toc51154209)

[Testing 17](#_Toc51154210)

[Esempi di funzionamento 18](#_Toc51154211)

[Valutazione delle prestazioni 19](#_Toc51154212)

[Al variare della dimensione della finestra di spedizione 19](#_Toc51154213)

[Al variare della probabilità di perdita dei messaggi 19](#_Toc51154214)

[Al variare della durata del timeout 19](#_Toc51154215)

[Manuale 20](#_Toc51154216)

[Installazione 20](#_Toc51154217)

[Configurazione 20](#_Toc51154218)

[Esecuzione 20](#_Toc51154219)

# Traccia del progetto

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.

La comunicazione tra client e server deve avvenire tramite un opportuno protocollo. Il protocollo di comunicazione deve prevedere lo scambio di due tipi di messaggi:

* messaggi di comando: vengono inviati dal client al server per richiedere l’esecuzione delle diverse operazioni;
* messaggi di risposta: vengono inviati dal server al client in risposta ad un comando con l’esito dell’operazione.

## Funzionalità del server

Il server, di tipo concorrente, deve fornire le seguenti funzionalità:

* L’invio del messaggio di risposta al comando list al client richiedente; il messaggio di risposta contiene la filelist, ovvero la lista dei nomi dei file disponibili per la condivisione;
* L’invio del messaggio di risposta al comando get contenente il file richiesto, se presente, od un opportuno messaggio di errore;
* La ricezione di un messaggio put contenente il file da caricare sul server e l’invio di un messaggio di risposta con l’esito dell’operazione.

## Funzionalità del client

I client, di tipo concorrente, devono fornire le seguenti funzionalità:

* L’invio del messaggio list per richiedere la lista dei nomi dei file disponibili;
* L’invio del messaggio get per ottenere un file
* La ricezione di un file richiesta tramite il messaggio di get o la gestione dell’eventuale errore
* L’invio del messaggio put per effettuare l’upload di un file sul server e la ricezione del messaggio di risposta con l’esito dell’operazione.

## Trasmissione 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 (cfr. Kurose & Ross “Reti di Calcolatori e Internet”, 7° Edizione).

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. I 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).

Opzionale: Implementare anche il controllo di flusso e il controllo di congestione.

# Architettura e scelte progettuali

## Server

Riguardo l'implementazione del server, naturalmente di tipo concorrenziale, abbiamo deciso di attuare una soluzione ibrida basata sia sui processi che sui thread, così da sfruttare i vantaggi di entrambe le tecnologie.

L’utilizzo di processi ha reso il server più robusto, in quanto un eventuale crash in un determinato processo non comporterebbe danni per gli altri che potrebbero continuare a funzionare senza problemi.

Per alleggerire il carico di lavoro che il server avrebbe dovuto eseguire nel gestire una nuova richiesta di connessione abbiamo implementato il pre-forking, istanziando un numero fisso di processi all'avvio. In questo modo siamo riusciti a limitare l’overhead dovuto alla chiamata della funzione *fork*, cosa che avrebbe altrimenti rallentato il three-way handshake iniziale.

Per far sì che ogni richiesta venga gestita da un diverso processo è stata creata nel processo padre una socket di ascolto principale. Grazie all’utilizzo della funzione *select*, il processo padre può accorgersi dell’arrivo di nuove connessioni e avvisare, con un segnale, un ben determinato processo figlio che sarà incaricato di gestirla.

A questo punto il processo figlio crea un nuovo thread incaricato di mantenere attiva la connessione con il client e di rispondere ai comandi che quest'ultimo invia. Abbiamo scelto di far gestire le connessioni a thread piuttosto che a nuovi processi in quanto creare un nuovo thread è un’operazione meno dispendiosa per la CPU del computer sul quale è in esecuzione il server ed anche molto più veloce.

## Trasmissione

Lavorando a livello applicativo, abbiamo dovuto effettuare delle scelte diverse rispetto a quelle che hanno dovuto effettuare i progettisti del protocollo TCP.

Una delle grosse differenze è che l’unico flusso di dati da trasmettere che è necessario gestire è quello dell’applicazione, nel nostro caso un client ed un server. Inoltre, utilizzando il protocollo UDP al livello inferiore abbiamo potuto trascurare alcuni dettagli per la consegna dei dati e la verifica della correttezza dei dati ricevuti.

Abbiamo scelto di realizzare una fase iniziale di instaurazione della connessione con un handshake a tre vie, proprio come accade nel protocollo TCP: il client invia un segmento di SYN al server, che risponde con un SYN-ACK ed attende l’ACK dal client.

La trasmissione di dati comincia nel momento in cui il client invia un comando al server: se il comando inserito è di list, il server invierà al client tutti i file che sono presenti nella sua cartella. Nel caso di una get il client agirà come sender ed il server come receiver, mentre nel caso di un comando di put i ruoli saranno scambiati.

Abbiamo deciso di implementare lo scambio dei file andando a dividere quest’ultimo in blocchi ed inviando un blocco in ogni sessione, in modo che il sender andasse a frammentare questo blocco di dati ed inviasse i segmenti risultanti da questa frammentazione al recevier.

Il receiver risponde con un acknoledgment, elemento "base" per la comunicazione affidabile, che può andare a riscontare più di un segmento, e nel momento in cui ha ricevuto tutti i dati che compongono un blocco, li scrive nel suo file.

Questo pattern si ripete finché il sender non ha inviato tutti i dati ed il receiver non li ha riscontrati tutti e quindi, di conseguenza, scritti tutti sul file.

Il flusso di byte non è quindi continuo, ma prevede alcune interruzioni, questo perché essendo ad un livello superiore rispetto a quello in cui è normalmente in esecuzione il protocollo TCP, è necessario andare a scrivere i dati ottenuti dalla comunicazione su file, e fare una scrittura per ogni segmento ricevuto risulta sicuramente più oneroso rispetto a fare scritture di blocchi di byte.

# Implementazione

## Comunicazione affidabile

La struttura dati principale che abbiamo utilizzato per la gestione dei segmenti sia lato sender che lato receiver è la seguente:

typedef struct tcp\_segment

{

  unsigned int sequence\_number;

  unsigned int ack\_number;

  unsigned int data\_length;

  unsigned int receiver\_window;

  //int checksum;

  char data[MSS];

  //char cwr;

  bool syn;

  bool fin;

  bool ack;

  unsigned short int checksum;

  //this field is usefull to keep the segments in a linked list

  struct tcp\_segment \*next;

} tcp;

La struttura non contiene tutti i campi del classico segmento tcp, in quanto per la nostra applicazione non erano richiesti.

Questo segmento viene poi compresso all’interno di un buffer, che verrà poi spedito sulla socket dal sender al receiver, e quest’ultimo estrarrà dal buffer i vari campi utilizzandoli per popolare a sua colta una struct dello stesso tipo.

int make\_seg(tcp segment, char \*send\_segm) {

    int bytes\_written = 0;

    // copy the sequence number

    unsigned int\* send\_segm\_ptr = (unsigned int\*) send\_segm;

    \*send\_segm\_ptr = htonl(segment.sequence\_number);

    send\_segm\_ptr++;

    bytes\_written += sizeof(unsigned int);

    // copy ack

    \*send\_segm\_ptr = htonl(segment.ack\_number);

    send\_segm\_ptr++;

    bytes\_written += sizeof(unsigned int);

    // copy flags

    bool\* send\_segm\_ptr\_flag = (bool\*) send\_segm\_ptr;

    \*send\_segm\_ptr\_flag = segment.ack;

    send\_segm\_ptr\_flag++;

    bytes\_written += sizeof(bool);

    \*send\_segm\_ptr\_flag = segment.syn;

    send\_segm\_ptr\_flag++;

    bytes\_written += sizeof(bool);

    \*send\_segm\_ptr\_flag = segment.fin;

    send\_segm\_ptr\_flag++;

    bytes\_written += sizeof(bool);

    // verify for the receiver window

    send\_segm\_ptr = (unsigned int\*) send\_segm\_ptr\_flag;

    \*send\_segm\_ptr = htonl(segment.receiver\_window);

    send\_segm\_ptr++;

    bytes\_written += sizeof(unsigned int);

    // send data field length

    \*send\_segm\_ptr = htonl(segment.data\_length);

    send\_segm\_ptr++;

    bytes\_written += sizeof(unsigned int);

    memcpy(send\_segm\_ptr, segment.data, segment.data\_length);

    char \*send\_segm\_char = (char \*)send\_segm\_ptr;

    send\_segm\_char += segment.data\_length;

    bytes\_written += segment.data\_length;

    #ifdef ACTIVE\_LOG

        snprintf(msg, LOG\_MSG\_SIZE, "make\_segment \nseq num: %d\nack num: %d\nASF: %d%d%d\nData length: %d\nbytes written on send\_buf: %d\n",

            segment.sequence\_number, segment.ack\_number, segment.ack, segment.syn, segment.fin, segment.data\_length, bytes\_written);

        print\_on\_log(fd, msg);

        memset(msg, 0, LOG\_MSG\_SIZE);

    #endif

    return bytes\_written;

}

int extract\_segment(tcp \*segment, char \*recv\_segm) {

    int bytes\_recv = 0; // usefull to calculate the chsum

    memset(segment, 0, sizeof(\*segment));

    memset(&(segment->data), 0, MSS);

    //deserialize sequence number

    unsigned int\* recv\_buf\_ptr = (unsigned int\*) recv\_segm;

    segment->sequence\_number = ntohl(\*recv\_buf\_ptr);

    recv\_buf\_ptr++;

    bytes\_recv += sizeof(unsigned int);

    // deserialize ack number

    segment->ack\_number = ntohl(\*recv\_buf\_ptr);

    recv\_buf\_ptr++;

    bytes\_recv += sizeof(unsigned int);

    // deserialize flags

    bool\* recv\_buf\_ptr\_flag = (bool\*) recv\_buf\_ptr;

    segment->ack = \*recv\_buf\_ptr\_flag;

    recv\_buf\_ptr\_flag++;

    bytes\_recv += sizeof(bool);

    segment->syn = \*recv\_buf\_ptr\_flag;

    recv\_buf\_ptr\_flag++;

    bytes\_recv += sizeof(bool);

    segment->fin = \*recv\_buf\_ptr\_flag;

    recv\_buf\_ptr\_flag++;

    bytes\_recv += sizeof(bool);

    // deserialize rcvwnd

    recv\_buf\_ptr = (unsigned int\*) recv\_buf\_ptr\_flag;

    segment->receiver\_window = ntohl(\*recv\_buf\_ptr);

    recv\_buf\_ptr++;

    bytes\_recv += sizeof(unsigned int);

    // deserialize data field length

    segment->data\_length = ntohl(\*recv\_buf\_ptr);

    recv\_buf\_ptr++;

    bytes\_recv += sizeof(unsigned int);

    memcpy(segment->data, (char\*)recv\_buf\_ptr, segment->data\_length);

    bytes\_recv += segment->data\_length;

    char \*recv\_buf\_char = (char \*)recv\_buf\_ptr;

    recv\_buf\_char += segment->data\_length;

    #ifdef ACTIVE\_LOG

        snprintf(msg, LOG\_MSG\_SIZE, "extract\_segment \nseq num: %d\nack num: %d\nASF: %d%d%d\nData length: %d\n", segment->sequence\_number, segment->ack\_number, segment->ack, segment->syn, segment->fin, segment->data\_length);

        print\_on\_log(fd, msg);

        memset(msg, 0, LOG\_MSG\_SIZE);

    #endif

    return segment->data\_length;

}

Abbiamo poi utilizzato un'altra struttura dati per tenere traccia della finestra di spedizione e ricezione, in quanto vogliamo sempre sapere quale sarà il numero di segmento del prossimo segmento atteso, quanti byte sono stati inviati, quanti sono stati riscontrati e quanti possono essere inviati:

typedef struct sliding\_window {

  int on\_the\_fly; // the number of bytes actually on the fly

  int n\_seg; // keeps the number of segments that can be sent

  int next\_to\_ack; //the left limit of the win

  int next\_seq\_num; //the next byte we are going to send as soon as possible

  int max\_size; // the maximum number of bytes that can be on the fly at the same time

  int last\_to\_ack; // the right limit of the win

  int tot\_acked; // the total byte that have been acked

  int last\_correctly\_acked; // the last segment correctly acked, usefull for retx in case of loss / 3 dupl. ack

  int dupl\_ack; // this field will keep the number of dupicate acks received for a segment

  //int congWin;

  int rcvwnd;

  int last\_byte\_buffered;

  int bytes\_acked\_current\_transmission;

} slid\_win;

Come nel protocollo TCP, anche nella nostra implementazione sono presenti i meccanismi di fast retransmission, che prevede la ritrasmissione immediata del più vecchio segmento non ancora riscontrato nel momento in cui si ricevono 3 ack duplicati e di delayed ack, ovvero il receiver attende per 500ms l'arrivo di un nuovo segmento dopo averne ricevuto uno, prima di mandare un riscontro.

In particolare, l'utilizzo del delayed ack permette al receiver, in condizioni di traffico normali, di inviare un riscontro ogni due segmenti, diminuendo così il numero di ack necessari per completare lo scaricamento del file.

Essendo in esecuzione su localhost, rilevare perdita di segmenti sarebbe stato molto difficile, così abbiamo simulato questa situazione mediante un’apposita funzione di send: questa genera un numero casuale compreso fra 0 e 100 (in virgola mobile) e se questo numero è maggiore di una data percentuale di soglia, che è possibile configurare all’avvio dell’applicativo, l’invio avviene con successo; altrimenti il segmento viene “perso”.

int send\_unreliable(int sockd, char \*segm\_to\_go, int n\_bytes) {

    float p = ((float)rand()/(float)(RAND\_MAX)) \*100;

    // we check if we will "lose" the segment

    if(p >= loss\_prob) {

        int n\_send = send(sockd, segm\_to\_go, n\_bytes, 0);

        #ifdef ACTIVE\_LOG

            snprintf(msg, LOG\_MSG\_SIZE, "send\_unreliable: Send success(%d bytes), asked to send %d bytes...\n", n\_send, n\_bytes);

            print\_on\_log(fd, msg);

            memset(msg, 0, LOG\_MSG\_SIZE);

        #endif

    }

    else {

        #ifdef ACTIVE\_LOG

            snprintf(msg, LOG\_MSG\_SIZE, "send\_unreliable: lost %d bytes \n", n\_bytes);

            print\_on\_log(fd, msg);

            memset(msg, 0, LOG\_MSG\_SIZE);

        #endif

    }

    return 0;

}

## Timeout

Nel protocollo TCP, il timeout associato ai segmenti viene stimato mediante un’apposta funzione, che utilizza i valori stimati della media e della varianza dell’RTT (Round Trip Time) per definire il timeout.

Tale stima prevede di andare a misurare l’RTT calcolando il tempo che passa tra l’invio di un segmento e la ricezione del suo ACK, usando questo valore in una media esponenziale pesata per andare a stimare sia media che varianza dell’RTT come segue:

Il timeout viene poi impostato come:

La stessa cosa avviene nel nostro applicativo, per mantenere i risultati abbiamo utilizzato la struttura dati timeval, che permette di registrare i secondi ed i microsecondi di tempo.

void estimate\_timeout(time\_out \*timeo, struct timeval first\_time, struct timeval last\_time) {

    struct timeval result; // temp struct to save the result;

    result.tv\_sec = last\_time.tv\_sec - first\_time.tv\_sec;

    result.tv\_usec = last\_time.tv\_usec - first\_time.tv\_usec;

    // it may be that the microsec is a negative value, so we scale it until it is positive

    while(result.tv\_usec < 0) {

        result.tv\_usec += 1000000; // we add 1 sec

        result.tv\_sec --; // we subtract the sec we added to the microsec

    }

    // calculate the value of the Estimated\_RTT

    timeo->est\_rtt.tv\_sec = 0.125\*result.tv\_sec + (1-0.125)\*timeo->est\_rtt.tv\_sec;

    timeo->est\_rtt.tv\_usec = 0.125\*result.tv\_usec + (1-0.125)\*timeo->est\_rtt.tv\_usec;

    // calculate the value of the Dev\_RTT

    timeo->dev\_rtt.tv\_sec = (1-0.25)\*timeo->dev\_rtt.tv\_sec + 0.25\*abs(timeo->est\_rtt.tv\_sec - result.tv\_sec);

    timeo->dev\_rtt.tv\_usec = (1-0.25)\*timeo->dev\_rtt.tv\_usec + 0.25\*abs(timeo->est\_rtt.tv\_usec - result.tv\_usec);

    // set the new value for the timeout

    timeo->time.tv\_sec = timeo->est\_rtt.tv\_sec + 4\*timeo->dev\_rtt.tv\_sec;

    timeo->time.tv\_usec = timeo->est\_rtt.tv\_usec + 4\*timeo->dev\_rtt.tv\_usec;

    while(timeo->time.tv\_usec > 1000000) {

        timeo->time.tv\_sec += 1;

        timeo->time.tv\_usec -= 1000000;

    }

    #ifdef ACTIVE\_LOG

        snprintf(msg, LOG\_MSG\_SIZE, "estimate\_timeout: TO: %ld s, %ld us\n", timeo->time.tv\_sec, timeo->time.tv\_usec);

        print\_on\_log(fd, msg);

        memset(msg, 0, LOG\_MSG\_SIZE);

    #endif

}

Il timeout viene impostato ogni volta che il sender riceve un nuovo ACK, o nel caso in cui scade il precedente timeout (che comporta quindi di ricalcolarlo con le formule sopra), per essere poi resettato nel momento in cui termina la sessione TCP.

## Controllo di flusso

Per l’implementazione del controllo di flusso abbiamo adottato una strategia leggermente diversa rispetto al protocollo TCP.

Durante lo scambio di messaggi si fa uso principalmente di due funzioni, che abbiamo denominato *send\_tcp* e *recv\_tcp*. Una lato sender e l’altra lato receiver.

Poiché l’obiettivo era effettuare una trasmissione affidabile a livello applicativo, la strategia utilizzata è stata utilizzare il buffer passato nei parametri della funzione *recv\_tcp* come unico buffer per la memorizzazione dei dati ricevuti e di cui si effettua la consegna in ordine, un buffer con dimensione pari ad un intero segmento, compreso di header, per la ricezione dei dati dal livello inferiore su cui si fa uso del protocollo UDP, ed una lista collegata in cui si mantengono tutti i segmenti fuori ordine.

Nella funzione *recv\_tcp* si tiene conto dello spazio rimanente nel buffer passatogli nei parametri utilizzando una struttura dati denominata *recv\_win* che memorizza dati relativi finestra di ricezione, in particolare il campo *rcvwnd*, che tiene conto di quanti dati può ancora memorizzare in ordine nel buffer.

Per effettuare la consegna dei dati in ordine, i segmenti fuori ordine vengono memorizzati in una lista collegata, che viene ispezionata ad ogni arrivo di un nuovo segmento per memorizzarlo in una posizione che tenga conto del suo numero di sequenza e mantenerli così sempre in ordine all’interno della lista.

La lista viene ispezionata periodicamente in base all’arrivo di nuovi segmenti per effettuare la consegna dei dati, a cui segue la liberazione dell’area di memoria e poi l’invio di un ACK al mittente, comunicando qual è l’ultimo byte di cui ha effettuato la consegna in ordine.

Nel momento in cui viene effettuata una consegna in ordine, e quindi vengono copiati i dati all’interno del segmento nel buffer di memorizzazione dei dati, il campo *rcvwnd* della struttura *recv\_win* viene decrementato in base alla dimensione del campo dati del segmento. Il valore all’interno di questo campo viene letto ogni volta che si prepara il header di un nuovo segmento da inviare, comunicando in questo modo al mittente quanti dati può ancora ricevere.

Il mittente estrae questo dato dal header del segmento di riscontro e verifica se può inviare ulteriori dati effettuando la differenza tra l’ultimo byte che ha inviato e che al momento è in volo e l’ultimo byte riscontrato, ottenendo così un valore confrontabile con la *rcvwnd*; se la differenza tra *rcvwnd* e questo valore è maggiore di 0 si procede a inviare altri dati, altrimenti, se minore o uguale a 0 si interrompe la trasmissione dei dati e si ritorna al chiamante il numero di byte che si è riusciti a trasmettere e di cui si è ricevuto il riscontro.

Il chiamante di *send\_tcp* potrà iniziare una nuova trasmissione nel caso debba inviare il resto dei dati, tuttavia la trasmissione verrà sbloccata solamente nel momento in cui il destinatario avrà effettuato una nuova chiamata a *recv\_tcp*, passando un buffer con nuova memoria disponibile.

Server e Client FTP

Le operazioni che il client mette a disposizione dell’utente sono:

* LIST: per visualizzazione i file disponibili sul server;
* GET: per effettuare il download di un file dal server;
* PUT: per effettuare l’upload di un file sul server;
* HELP: per visualizzare il menu e, di conseguenza, i comandi che l’utente può inserire.

Tutti i comandi, oltre al trasferimento del file vero e proprio e ad eventuali messaggi di errore, viaggiano sulla rete grazie ai comandi *send\_tcp* e *recv\_tcp* già descritti in precedenza, naturalmente previa connessione tra client e server grazie all’utilizzo delle socket.

Inoltre, per permettere il corretto trasferimento dei files, si utilizzano due funzioni rispettivamente denominate *RetrieveFile* e *SendFile,* la prima che appunto permette di inviare un file (usata lato server nel caso di *get* e lato client nel caso di *put*), e la seconda per permetterne la ricezione (lato server in caso di *put* e lato client in caso di *get*).

La *SendFile,* dopo aver aperto il file richiesto, entra in un ciclo *while* nel quale legge i bytes del file, li salva in un buffer ausiliario di lunghezza pari al massimo numero di byte che il receiver può accettare e li invia al destinatario sfruttando per l’appunto la funzione *send\_tcp*. Il *while* termina nel momento in cui tutti i bytes sono stati inviati o se ci dovesse essere un errore nella *send\_tcp*.

Analogamente la *RetrieveFile*, dopo aver creato il file, entra in un ciclo *while* nel quale legge i bytes dalla socket grazie alla funzione *recv\_tcp*, li alloca in un buffer ausiliario e li scrive sul file precedentemente creato. Anche questo *while* termina solo nel momento in cui il receiver non trova più byte da leggere dalla socket, cosa che comporta la corretta ricezione del file, o nel momento in cui dovesse avvenire degli errori nella *recv\_tcp* o nella scrittura del file.

## File di log

Per facilitare la ricerca degli errori abbiamo utilizzato due file di log, in cui venivano scritti i passaggi che veniva seguiti durante la fase di invio di segmenti / ricezione di ack.

Ogni messaggio è caratterizzato da un timestamp, così da riuscire a capire meglio l’ordine cronologico con cui si svolgono le azioni durante tutta la comunicazione.

Il file è unico per ogni thread del server e per ogni client che si connette, e viene creato subito dopo la terminazione del 3-way handshake.

Siccome le scritture sul file possono incidere sulle prestazioni del protocollo, abbiamo deciso di utilizzare un ifdef per attivare o meno la creazione del log, così da non generarlo nella fase di testing per avere dei risultati più accurati sulle effettive capacità del protocollo da noi realzzato.

# Limitazioni riscontrate

# Testing

Indicazione della piattaforma software usata per lo sviluppo ed il testing del sistema.

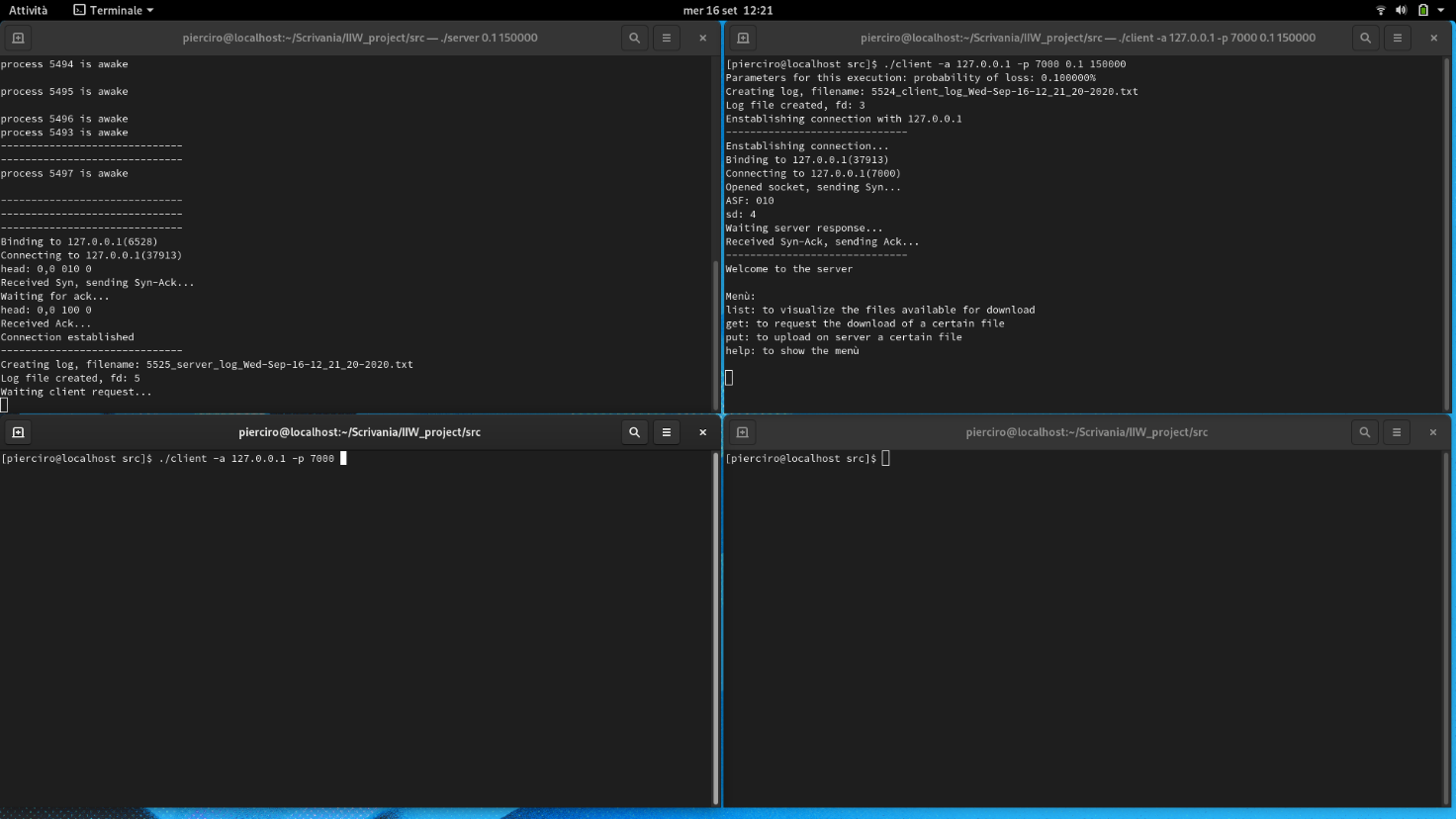
Le piattaforme software utilizzate per lo sviluppo ed il testing del sistema sono le seguenti:

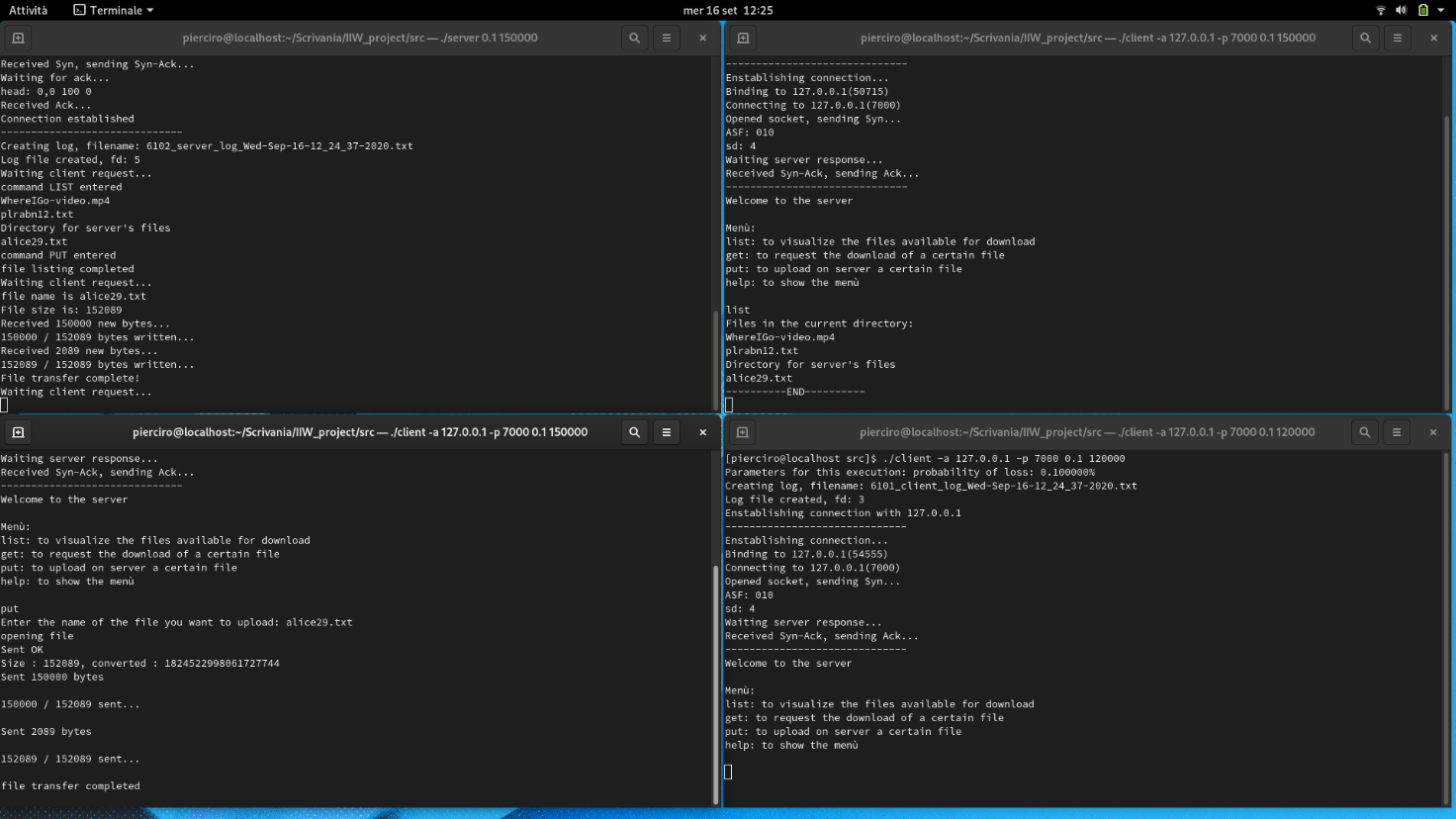
Ide: Visual Studio code v 1.49

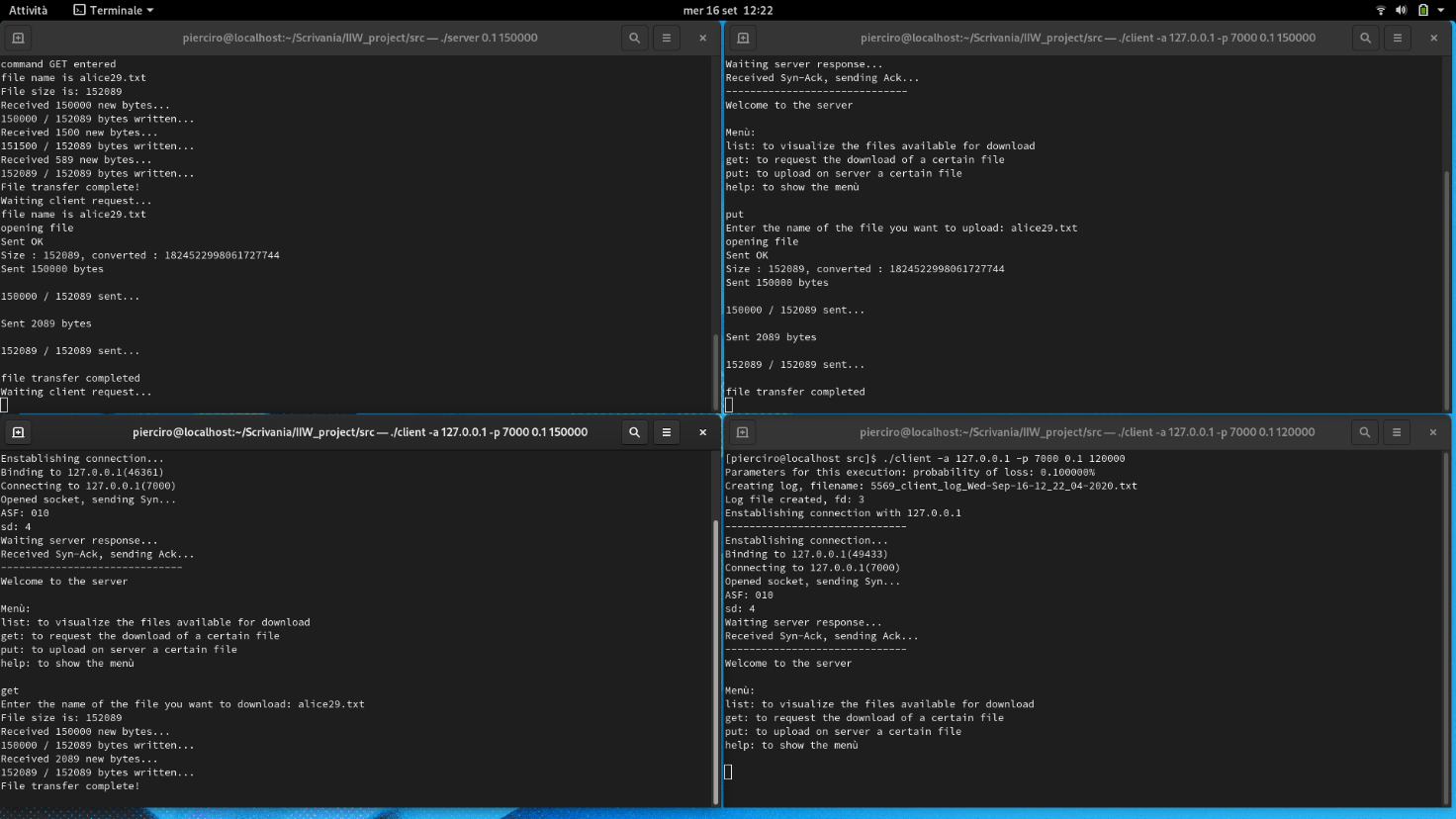
Sistemi Operativi: Fedora v 32, aggiungete i vostri

# Esempi di funzionamento

Una volta che un client si connette al server, può inviare i comandi per scaricare la lista dei server attivi, fare l’upload o il download dei file.







# Valutazione delle prestazioni

## Al variare della dimensione della finestra di spedizione

## Al variare della probabilità di perdita dei messaggi

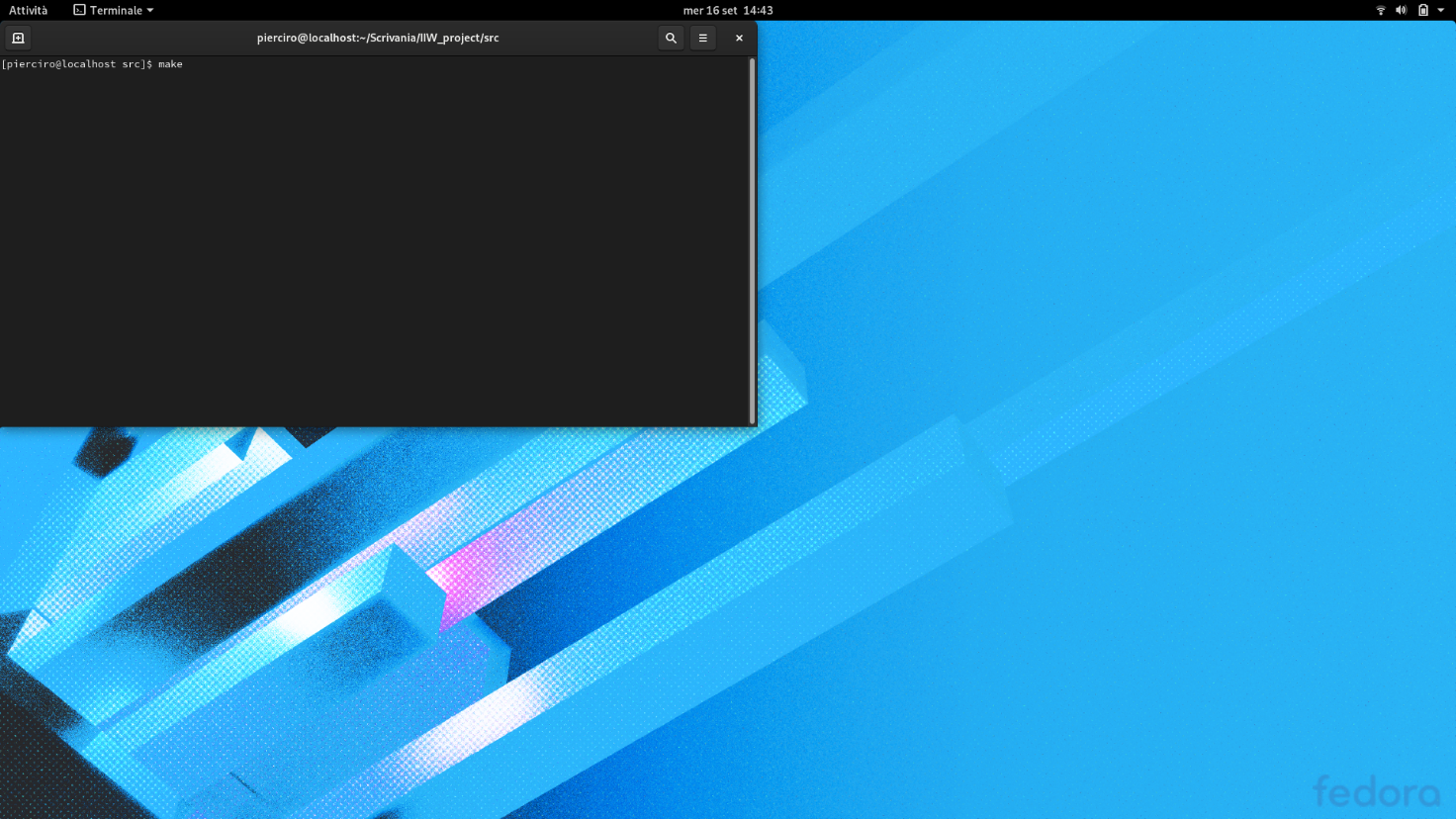
## Al variare della durata del timeout

# Manuale

## Installazione

Per l’installazione dell’applicativo viene fornito un apposito Makefile, che può essere eseguito nel terminale tramite il comando make.

A questo punto sarà possibile configurare ed eseguire client e server.

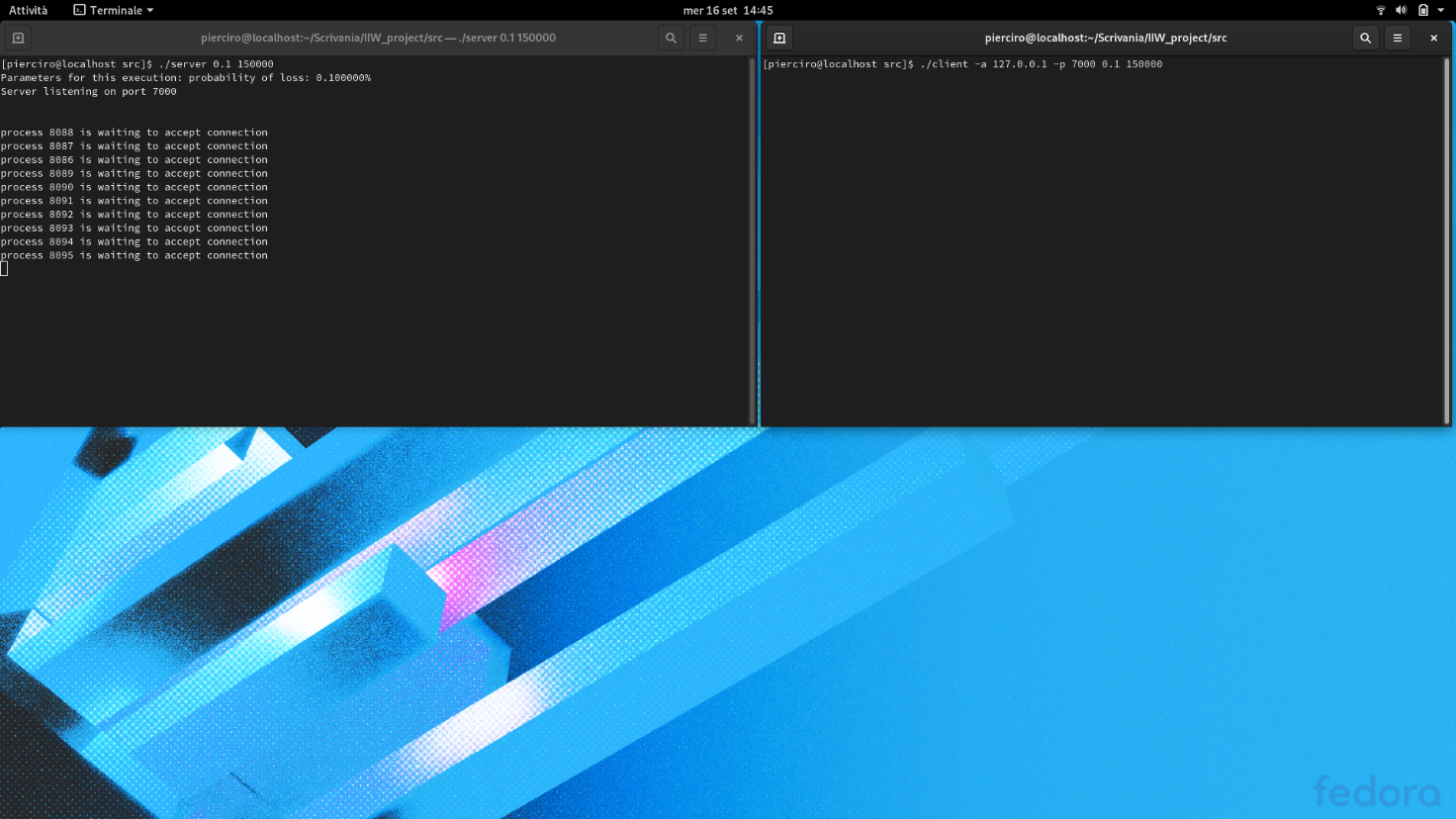


## Configurazione

La configurazione e la successiva esecuzione di client e server sono molto semplici:

per la configurazione del client bisogna passare come argomenti l’indirizzo ip del server a cui connettersi, la porta del server, un valore float per simulare la perdita ed un valore per la finestra di spedizione.

Per il server è sufficiente specificare il valore per la probabilità di perdita e quello per la finestra di spedizione.



## Esecuzione

Per eseguire gli applicativi è sufficiente digitare da terminale i comandi ./server per eseguire il server e ./client per il client (è necessario che venga eseguito prima il server e poi il client).

Il client avvierà la prima fase dell’handshake a 3 vie e nel momento in cui questo è completato sarà possibile inviare i comandi dal client al server



