



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
Funzionalità del server.....	3
Funzionalità del client	3
Trasmissione affidabile	4
Architettura e scelte progettuali.....	5
Server	5
Trasmissione.....	5
Implementazione	7
Comunicazione affidabile	7
Time out	9
Controllo di flusso.....	11
Controllo di congestione	13
File di log.....	14
Limitazioni riscontrate	14
Sviluppo e testing	15
Esempi di funzionamento	16
Valutazione delle prestazioni	17
Al variare della dimensione della finestra di spedizione	17
Al variare della probabilità di perdita dei messaggi	20
Al variare della durata del time out	22
Manuale	26
Installazione.....	26
Configurazione	26
Esecuzione	27

Traccia del progetto

Lo scopo del 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 trasporto di dati).

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 richiesto 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 concorrentiale, 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 hand shake 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 receiver.

Il receiver risponde con un acknowledgement, elemento "base" per la comunicazione affidabile, che può andare a riscontrare 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;
    char data[MSS];
    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 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.

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
};
```

Trasferimento file su UDP

```
int last_correctly_acked; // the last segment correctly acked, use-
full for retx in case of loss / 3 dupl. ack
int dupl_ack; // this field will keep the number of duplicate acks re-
ceived for a segment
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);
    }
    return 0;
}
```

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 quella di 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.

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.

Time out

Nel protocollo TCP, il time out 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 time out.

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:

$$Est_{RTT} = \alpha * Sample_{RTT} + (1 - \alpha) * Est_{RTT}$$

$$Dev_{RTT} = \beta * |Sample_{RTT} - Est_{RTT}| + (1 - \beta) * Dev_{RTT}$$

Il time out viene poi impostato come:

$$Est_{RTT} + 4 * Dev_{RTT}$$

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.

Trasferimento file su UDP

```
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;
    }
}
```

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

Controllo di flusso

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.

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.

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.

Per evitare che un utente possa chiedere il download di un file mentre sta avvenendo l'upload di quest'ultimo sul server da parte di un altro client, cosa che comporterebbe non pochi problemi, abbiamo deciso di salvare il file in un primo momento con un nome temporaneo, in modo da renderlo non accessibile agli altri client e, solo a trasferimento completato, rinominarlo.

Si è scelta questa soluzione in confronto all'utilizzo di semafori, in quanto più semplice da implementare e, cosa più importante, non andava a limitare in nessun modo l'efficienza del server nel gestire la concorrenza nel momento in cui sarebbero avvenute più richieste di download per lo stesso file.

Controllo di congestione

Per l'implementazione del controllo di congestione abbiamo cercato di avvicinarci il più possibile a quello che era il comportamento di TCP.

Abbiamo implementato un automa a stati finiti, mantenendo informazioni sullo stato attuale nel quale si trova il sender, oltre al valore delle variabili *congestion window* e *threshold*, in un'apposita struct:

```
typedef struct congestion_struct
{
    int cong_win;
    int threshold;
    int support_variable;
    int state;          // 0 = slow_start
                       // 1 = congestion_avoidance
                       // 2 = fast_recovery
} cong_struct;
```

A seconda poi di ciò che avviene in rete e a seconda dello stato in cui si trova, il sender reagisce in maniera diversa, incrementando la congestion window se riceve un ACK (in maniera esponenziale se in slow start o di 1 MSS se in congestion avoidance), riducendola a metà in caso di ACK duplicato o reimpostandola a 1 MSS in caso di time out.

L'utilizzo della variabile *support_variable* si è reso necessario per permettere l'invio di segmenti di dimensione fissa, aumentando la finestra di spedizione con solo multipli di 1 MSS.

Infatti, rispetto al controllo di congestione classico, abbiamo scelto un approccio più aggressivo per quanto riguarda la *cong_win*. Abbiamo infatti notato che spesso capitava che la finestra di congestione crescesse solo di alcune decine di byte, comportando perciò l'invio di segmenti con lunghezza del campo dati molto piccola.

Poiché non era molto efficiente, abbiamo deciso di scegliere per la *cong_win* solo multipli del valore di MSS, e abbiamo stabilito una soglia per decidere quanti byte fossero necessari per approssimare per eccesso la dimensione della *cong_win*. L'approssimazione per eccesso rende il protocollo leggermente più aggressivo, tuttavia il traffico risultante non è eccessivo, e risulta in un ottimo guadagno di prestazioni.

File di log

Per facilitare la ricerca degli errori abbiamo utilizzato due file di log, in cui venivano scritti i passaggi che venivano 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 hand shake.

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 realizzato.

Limitazioni riscontrate

Una volta che il receiver ha riempito il suo buffer di memorizzazione dei dati in ordine, smette ricevere dati e termina l'esecuzione della `recv_tcp`.

Il chiamante di `send_tcp` potrà iniziare una nuova trasmissione nel caso in cui 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.

Poiché siamo a livello applicativo e siamo noi a gestire il flusso di dati, questa limitazione non comporta perdita di efficienza. Infatti, per garantire che il sender invii tutti i suoi dati, una volta scelta una dimensione per il buffer passato a `recv_tcp`, basta leggere la stessa quantità di dati da passare a `send_tcp`.

Inoltre, per simulare la probabilità di perdita di un pacchetto, abbiamo utilizzato la funzione `rand` di C.

Questa è però una funzione che utilizza un algoritmo pseudo-random per la generazione dei numeri, quindi la sequenza viene generata a partire da un seed iniziale.

Sviluppo e testing

Di seguito elencate le macchine utilizzate per i test e lo sviluppo del sistema:

- Macchina 1:
 - i) Intel core i5-1035G1 @1Ghz
 - ii) 12GB RAM DDR4
 - iii) Fedora v 32
- Macchina 2:
 - i) Intel core i7-3610QM @2.3Ghz
 - ii) 8GB RAM DDR4
 - iii) Windows 10 Home version 2004 con Windows Subsystem for Unix 2 (Ubuntu)
- Macchina 3:
 - i) Intel core i7-6700HQ @2.6Ghz
 - ii) 16GB RAM DDR4
 - iii) Manjaro

Tutte le macchine utilizzano compilatore `gcc` e text editor *Visual Studio code* aggiornato all'ultima versione (attualmente la 1.49).

Esempi di funzionamento

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

The screenshot shows the Visual Studio Code interface with the following content:

- Explorer:**
 - client_files
 - alice29.txt
 - Directory for client's files
 - plrabn12.txt
 - ricerca.pdf
 - WhereIGo-video.mp4
 - LogFiles
 - server_files
 - alice29.txt
 - Directory for server's files
 - plrabn12.txt
 - prova2mb.pdf
 - ricerca.pdf
 - WhereIGo-video.mp4
 - test_files
 - client
 - client.c
 - client.o
 - helper.c
 - helper.h
 - helper.o
 - Makefile
 - manage_client.c
 - manage_client.h
 - manage_client.o
- Output:**

```
head: 0,0 010 0
Received Syn, sending Syn-A
ck...
Waiting for ack...
head: 0,0 100 0
Received Ack...
Connection established
-----
Waiting client request...
Binding to 127.0.0.1(7028)
Connecting to 127.0.0.1(461
07)
head: 0,0 010 0
Received Syn, sending Syn-A
ck...
Waiting for ack...
head: 0,0 100 0
Received Ack...
Connection established
-----
Waiting client request...
command LIST entered
WhereIGo-video.mp4
ricerca.pdf
plrabn12.txt
Directory for server's file
s
command GET entered
prova2mb.pdf
alice29.txt
file listing completed
Waiting client request...
command PUT entered
```
- Terminal:**

```
--
Establishing connection...
Binding to 127.0.0.1(29211)
Connecting to 127.0.0.1(7000
)
Opened socket, sending Syn..
ASF: 010
sd: 3
Waiting server response...
Received Syn-Ack, sending Ac
k...
Welcome to the server

Menù:
list: to visualize the files
available for download
get: to request the download
of a certain file
put: to upload on server a c
ertain file
help: to show the menù

list
Files in the current directo
ry:
WhereIGo-video.mp4
ricerca.pdf
plrabn12.txt
Directory for server's files
prova2mb.pdf
alice29.txt
-----END-----

get
Enter the name of the file y
ou want to download: 
```

The screenshot shows the Visual Studio Code interface with the following content:

- Output:**

```
8700000 / 38444837 bytes wr
itten...
Received 150000 new bytes..
8850000 / 38444837 bytes wr
itten...
Received 150000 new bytes..
9000000 / 38444837 bytes wr
itten...
Received 150000 new bytes..
9150000 / 38444837 bytes wr
itten...
Received 150000 new bytes..
9300000 / 38444837 bytes wr
itten...
Received 150000 new bytes..
9450000 / 38444837 bytes wr
itten...
Received 150000 new bytes..
9600000 / 38444837 bytes wr
itten...
Received 150000 new bytes..
9750000 / 38444837 bytes wr
itten...
Received 150000 new bytes..
9900000 / 38444837 bytes wr
itten...
```
- Terminal:**

```
--
Establishing connection...
Binding to 127.0.0.1(29211)
Connecting to 127.0.0.1(7000
)
Opened socket, sending Syn..
ASF: 010
sd: 3
Waiting server response...
Received Syn-Ack, sending Ac
k...
Welcome to the server

Menù:
list: to visualize the files
available for download
get: to request the download
of a certain file
put: to upload on server a c
ertain file
help: to show the menù

get
Enter the name of the file y
ou want to download: alice29
.txt
File size is: 152089
Received 150000 new bytes...
150000 / 152089 bytes writte
n...
Received 1500 new bytes...
151500 / 152089 bytes writte
n...
Received 589 new bytes...
152089 / 152089 bytes writte
n...
File transfer complete!
```


Valutazione delle prestazioni

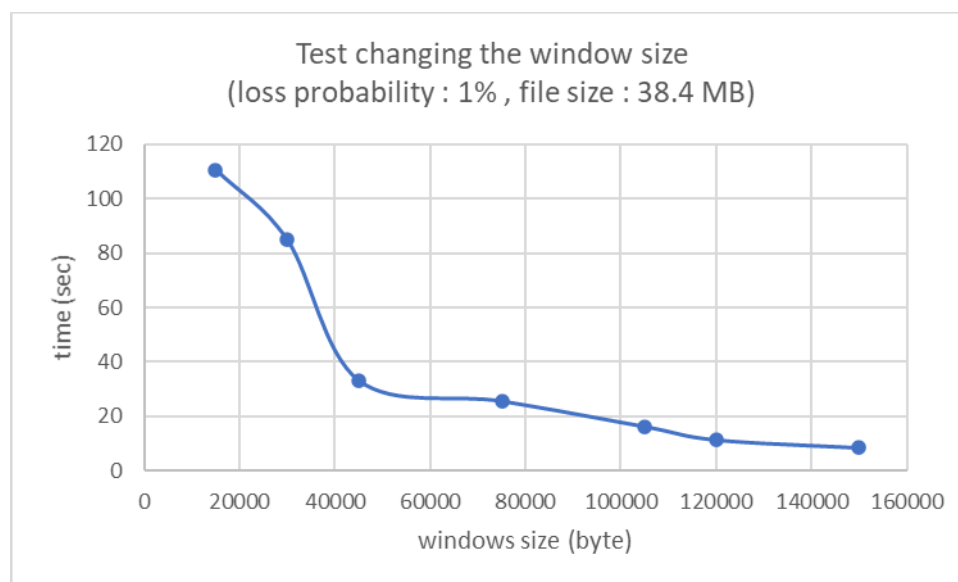
Al variare della dimensione della finestra di spedizione

In questi test, si vuole mostrare come la massima dimensione della finestra di spedizione va ad impattare sul tempo necessario per il trasferimento di un file.

Inoltre, nei test abbiamo utilizzato diversi valori di probabilità di perdita, per mostrare in maniera più significativa le differenze in diversi scenari, che si possono associare a diverse situazioni di traffico nella rete.

Test eseguito su macchina 2

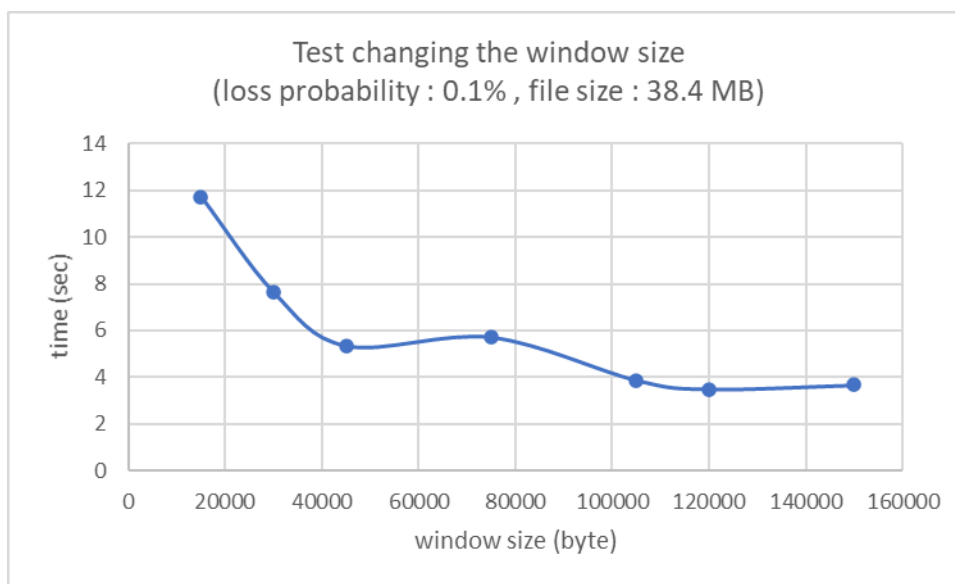
Finestra di spedizione	Tempo
15000	110,604353
30000	84,86932
45000	33,319163
75000	25,740944
105000	16,402079
120000	11,512585
150000	8,570336



Trasferimento file su UDP

Test eseguito su macchina 2

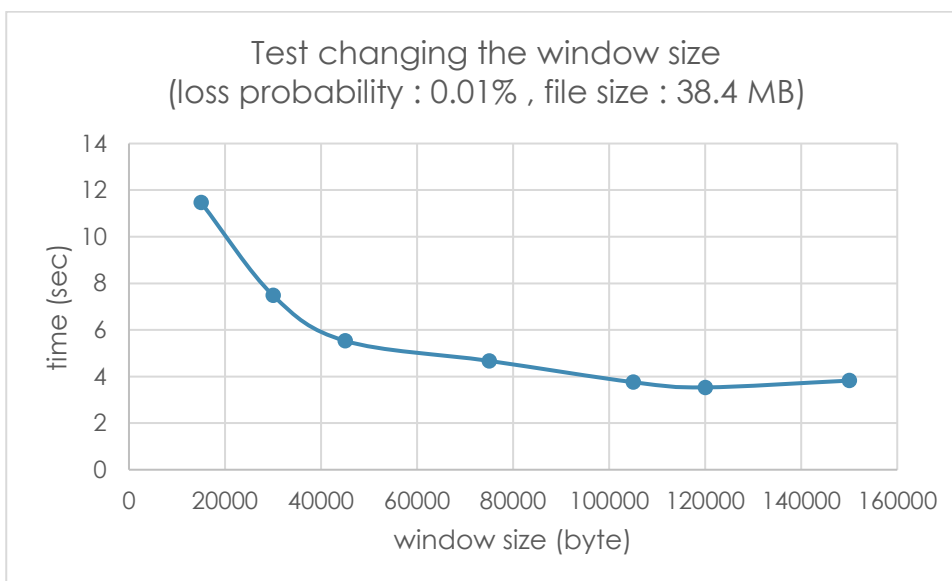
Finestra di spedizione	Tempo
15000	11,728688
30000	7,642265
45000	5,345444
75000	5,713533
105000	3,872734
120000	3,490206
150000	3,661753



Trasferimento file su UDP

Test eseguito su macchina 2

Finestra di spedizione	Tempo
15000	11,47226
30000	7,484207
45000	5,532718
75000	4,671297
105000	3,758404
120000	3,534586
150000	3,824606



Come si vede dai test, con una probabilità di perdita elevata, la dimensione della finestra di spedizione influisce maggiormente sulle presentazioni, poiché i continui pacchetti persi non permettono alla congestion window di crescere in maniera appropriata e di conseguenza al sender di trasmettere in pipeling il numero massimo di pacchetti che potrebbe inviare.

Le differenze si assottigliano invece quando la dimensione della finestra diventa più grande o, naturalmente, quando le probabilità di perdita di pacchetti diventa molto più bassa.

Al variare della probabilità di perdita dei messaggi

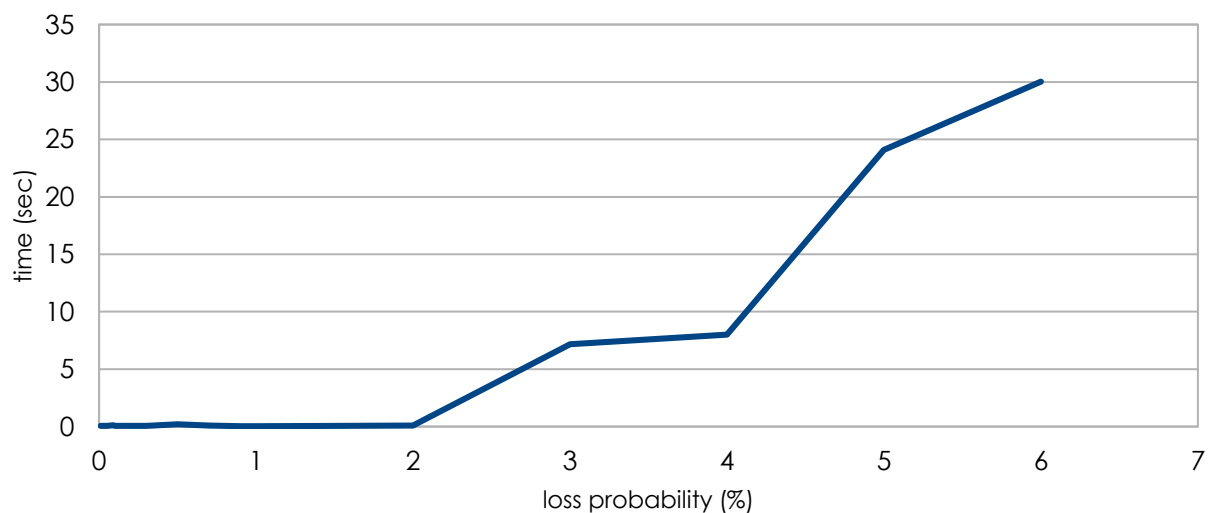
Nei seguenti test si vuole invece mostrare, fissata la dimensione della finestra di spedizione, come vari il tempo di trasmissione per un file al variare della probabilità di perdita.

La scelta è stata quella di partire da valori piccoli per salire fino a valori considerevoli, ciò per simulare scenari che vanno da condizioni buone di traffico (0.01%), a situazioni di traffico pesante ma non eccessivo (1%), a situazioni con probabili guasti sulla rete (>2%).

Test eseguito su macchina 1

Probabilità di perdita (%)	Tempo
0,01	0,058823
0,03	0,059404
0,05	0,059605
0,09	0,114438
0,1	0,062315
0,3	0,076275
0,5	0,207088
0,7	0,083274
0,9	0,040453
1	0,031293
2	0,087833
3	7,173608
4	8,011599
5	24,082674
6	30,029444

Test changing loss
(window size : 150000, file size : 12MB)



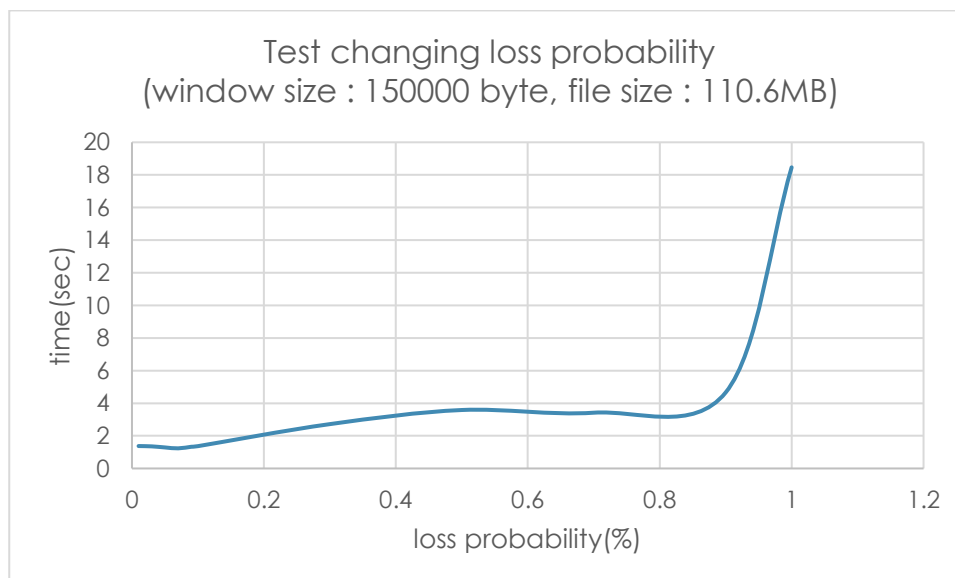
Trasferimento file su UDP

La scelta di usare un file di dimensione non molto grande è dovuta al fatto che i tempi sarebbe stati molto lunghi nel momento in cui ci si avvicinava a valori di probabilità di perdita intorno al 3-4 %, anche se ciò ha comportato la presenza di valori molto vicini e non molto significativi per probabilità di perdita piccole.

Abbiamo quindi deciso di ripetere lo stesso test trasferendo un file di dimensioni più grande, così da poter notare meglio le differenze di tempi per valori di probabilità di perdita piccoli.

Test eseguito su macchina 1

Probabilità (%)	Tempo
0,01	1,374857
0,03	1,354314
0,05	1,288958
0,07	1,234613
0,09	1,330593
0,1	1,368002
0,3	2,719608
0,5	3,584033
0,7	3,418939
0,9	4,659944
1	18,463385



Si può infatti notare che superando lo 0.1% di probabilità di perdita, i tempi aumentano di più di un secondo, mentre per valori precedenti le differenze non risultano significative. Questo test mette anche in evidenza, confermando i valori ottenuti nel precedente, come i tempi tenderanno a crescere in maniera molto più evidente per valori superiori all'1%.

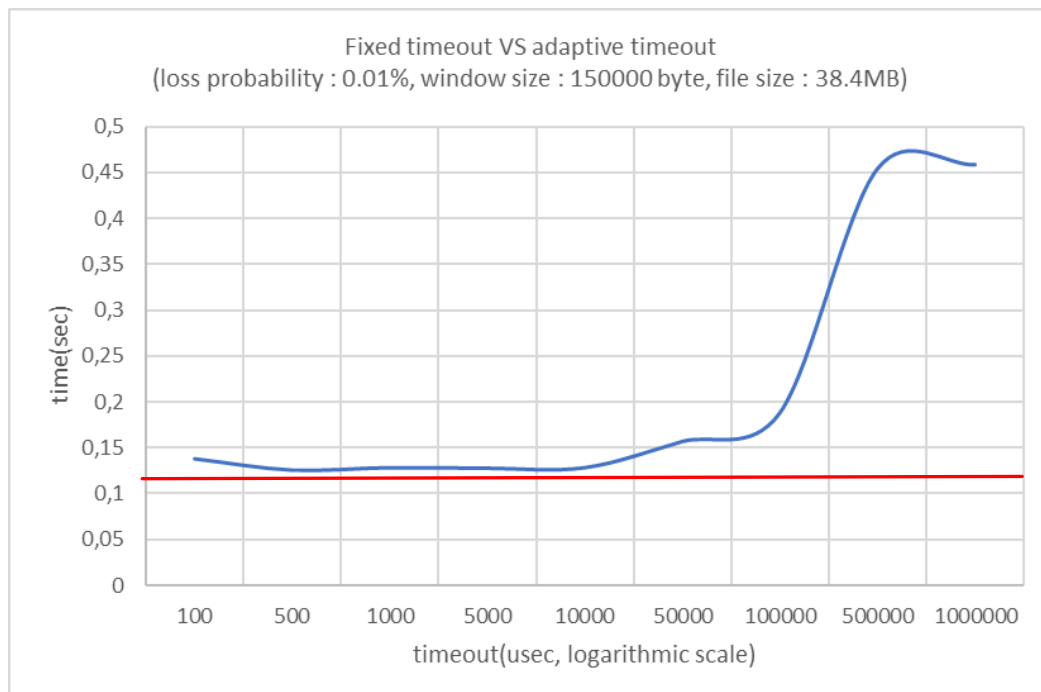
Al variare della durata del time out

In questi ultimi test invece, si vuole confrontare il comportamento del protocollo utilizzando il time out adattativo rispetto all'utilizzare un time out fisso.

Nota bene: la retta orizzontale rossa rappresenta il tempo ottenuto usando il time out adattativo, così da poter essere facilmente confrontabile con tutti gli altri tempi.

Test eseguito su macchina 1

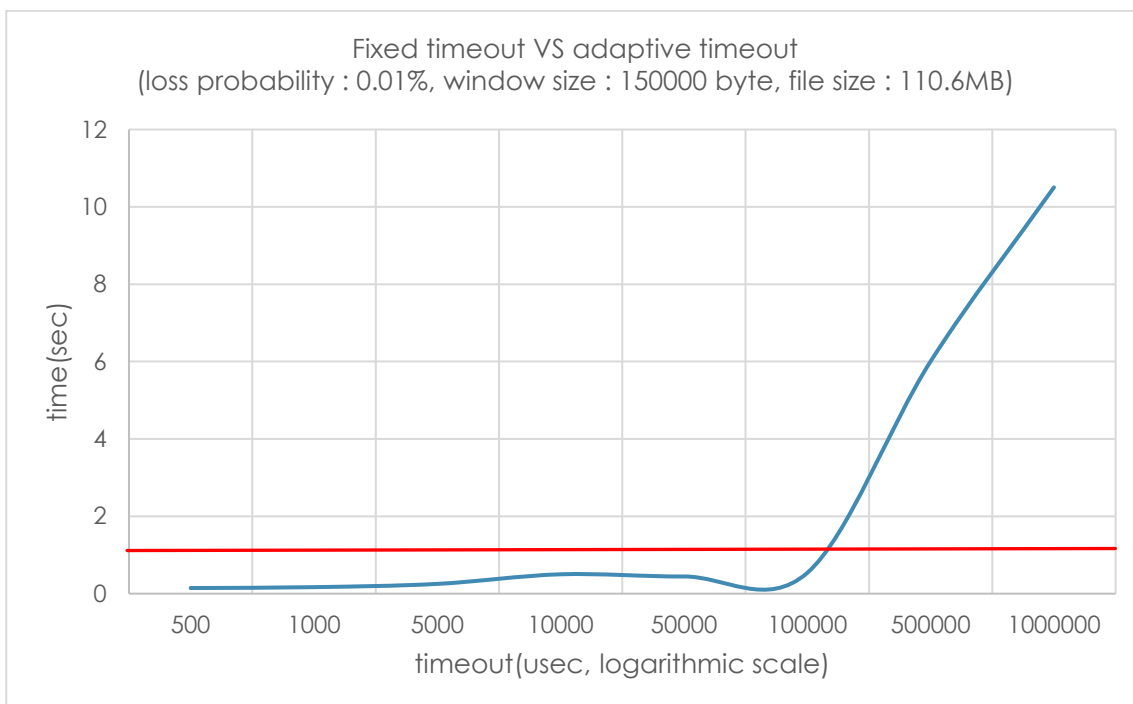
Time out (μsec)	Tempo
100	0,13797
500	0,125673
1000	0,128323
5000	0,12769
10000	0,128297
50000	0,15678
100000	0,18809
500000	0,453613
1000000	0,45799
Time out adattativo	0,114377



Trasferimento file su UDP

Test eseguito su macchina 1

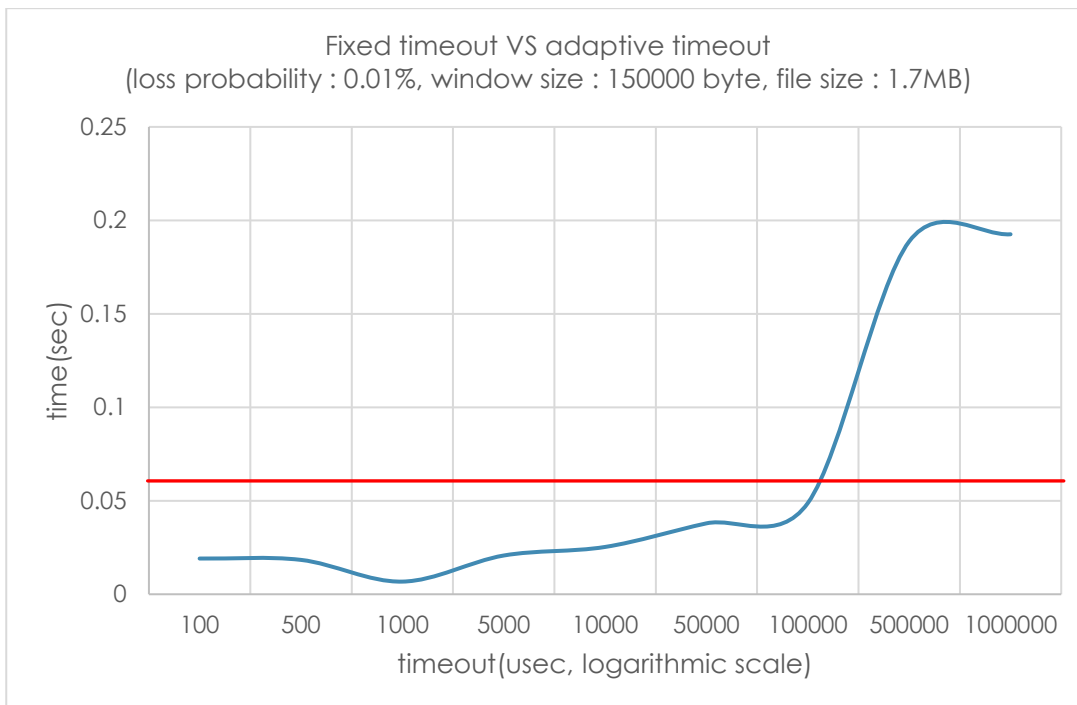
Time out (μsec)	Tempo
500	0,1459133
1000	0,168497
5000	0,252513
10000	0,500643
50000	0,444533
100000	0,538373
500000	6,008253
1000000	10,504623
Time out adattativo	1.1363366



Trasferimento file su UDP

Test eseguito su macchina 1

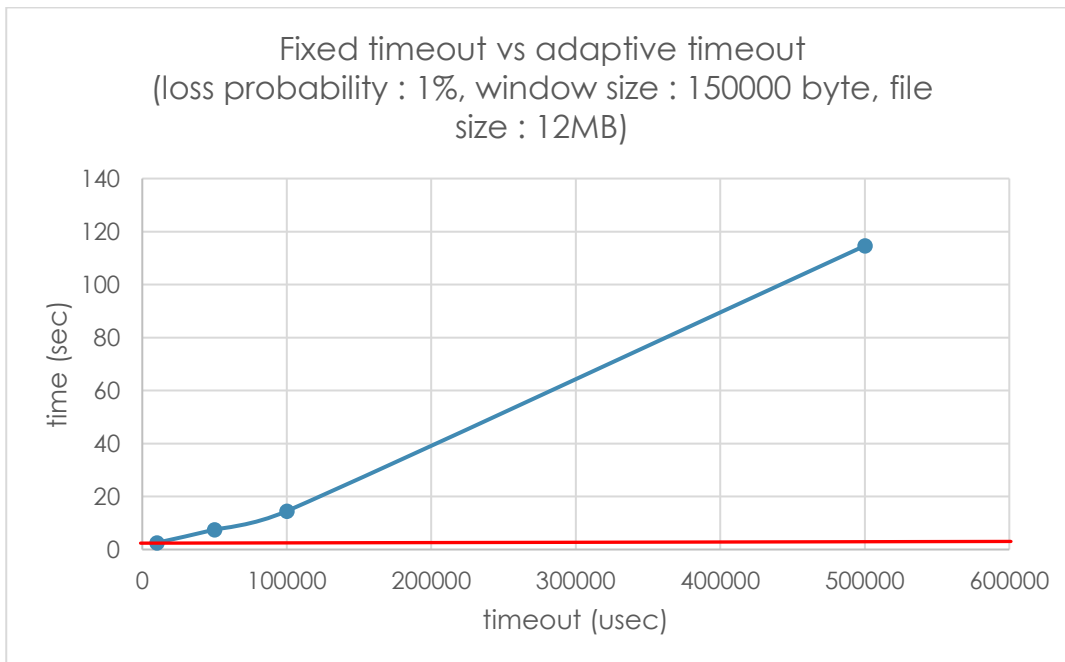
Time out (μsec)	Tempo
100	0,01911
500	0,018383
1000	0,006764
5000	0,020727
10000	0,025303
50000	0,03793
100000	0,048777
500000	0,18873
1000000	0,192567
Timeout adattativo	0,06397



Trasferimento file su UDP

Test eseguito su macchina 2

Timeout (μsec)	Tempo
10000	2,531366
50000	7,513507
100000	14,505645
500000	114,61466
Time out adattativo	2,145838



Dai test si può notare che nel caso medio il time out adattativo è migliore, comportando infatti dei tempi minori per lo scaricamento dei file.

Ci sono alcuni casi in cui sembrerebbe che il time out fisso sia conveniente, ma bisogna osservare che tutti i test sono stati ovviamente eseguiti in localhost, con RTT bassissimi: difatti, in una rete, avere RTT inferiori ai 100/500 microsecondi risulta molto difficile, da cui quindi si deduce che il time out adattativo è sicuramente la scelta migliore in qualsiasi caso.

Inoltre, bisogna anche considerare che il time out adattativo da noi implementato, prendendo come riferimento quello calcolato da TCP, agisce in maniera molto conservativa, ciò comporta che, per trasferimenti di files di piccole dimensioni, non riesca a recuperare lo scarto perso in partenza contro time out fissi ma inizialmente molto più piccoli.

Nell'ultimo test si osserva infatti che, nel caso di un livello di congestione della rete più alto, il time out adattativo si comporta decisamente meglio di quello fisso.

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.

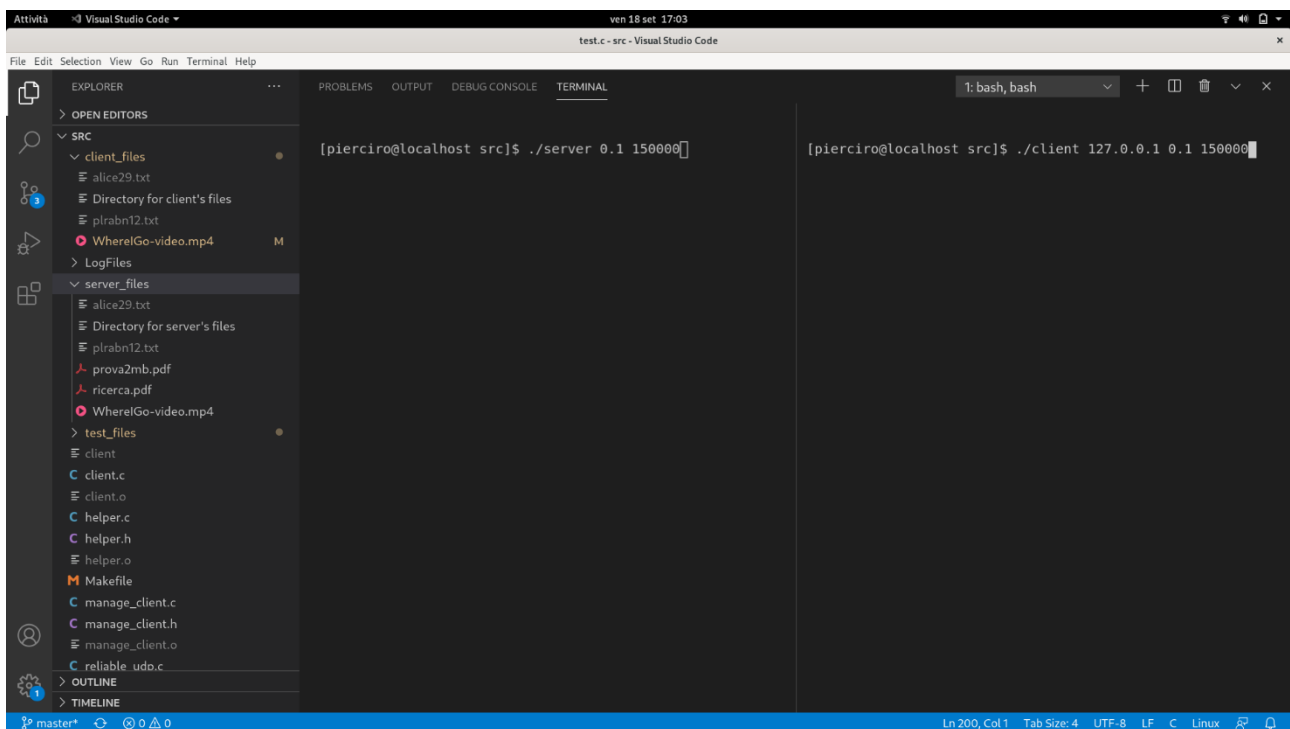
Configurazione

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;
- un valore per la finestra di spedizione.

Per il server è sufficiente specificare:

- il valore per la probabilità di perdita;
- il valore della finestra di spedizione.

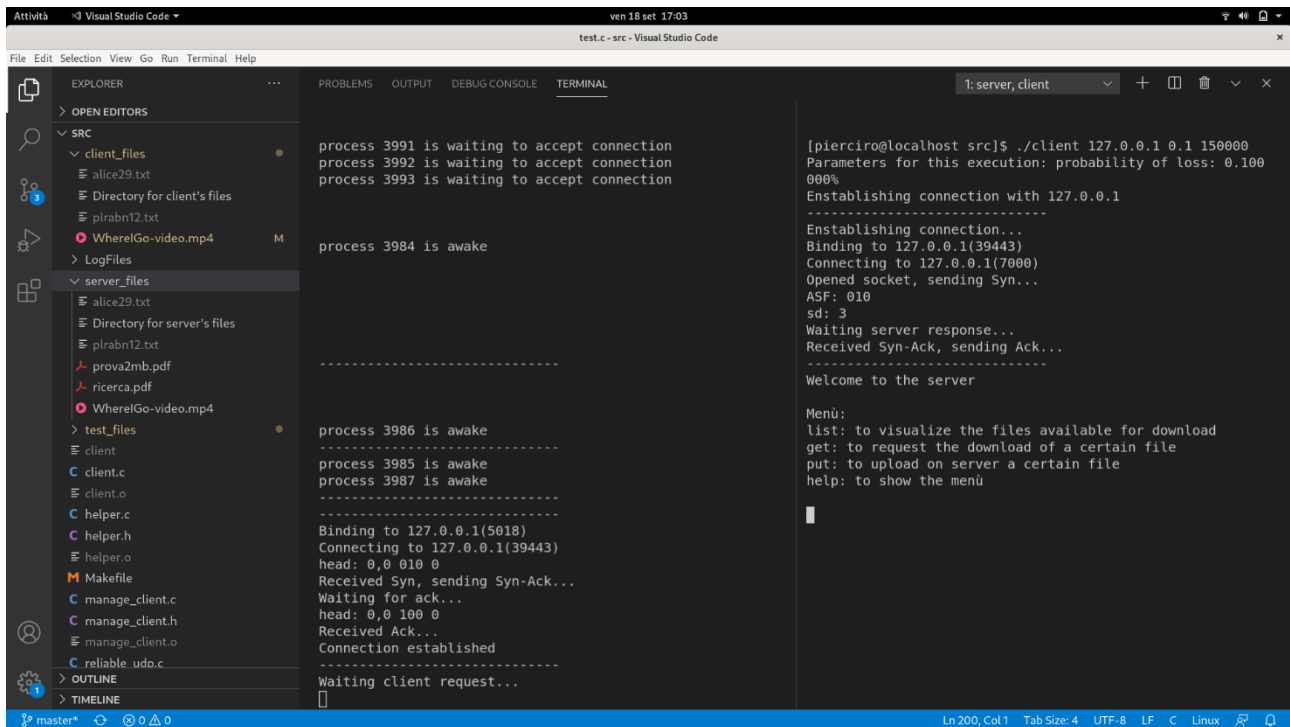


Trasferimento file su UDP

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).

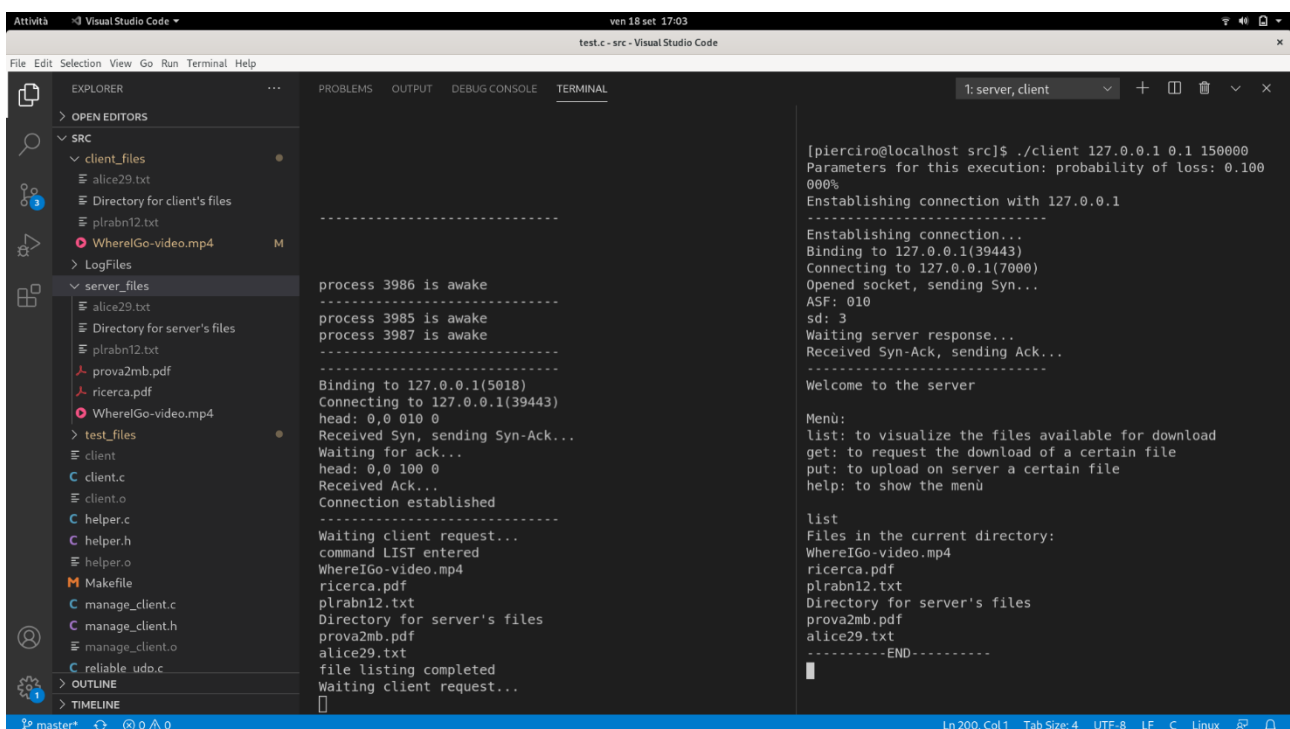
Il client avvierà la prima fase di instaurazione della connessione e, nel momento in cui questo è completato, sarà possibile inviare i comandi dal client al server



```
process 3991 is waiting to accept connection
process 3992 is waiting to accept connection
process 3993 is waiting to accept connection

process 3984 is awake

Binding to 127.0.0.1(5018)
Connecting to 127.0.0.1(39443)
head: 0,0 010 0
Received Syn, sending Syn-Ack...
Waiting for ack...
head: 0,0 100 0
Received Ack...
Connection established
Waiting client request...
```



```
process 3986 is awake
process 3985 is awake
process 3987 is awake

Binding to 127.0.0.1(5018)
Connecting to 127.0.0.1(39443)
head: 0,0 010 0
Received Syn, sending Syn-Ack...
Waiting for ack...
head: 0,0 100 0
Received Ack...
Connection established
Waiting client request...
command LIST entered
WhereIGo-video.mp4
ricerca.pdf
plrabn12.txt
Directory for server's files
prova2mb.pdf
alice29.txt
file listing completed
Waiting client request...
```

Trasferimento file su UDP

```
test.c - src - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER
  OPEN EDITORS
  SRC
    client_files
      alice29.txt
      Directory for client's files
      plrabn12.txt
      WhereGo-video.mp4
    LogFiles
    server_files
      alice29.txt
      Directory for server's files
      plrabn12.txt
      prova2mb.pdf
      ricerca.pdf
      WhereGo-video.mp4
    test_files
      client
      client.c
      client.o
      helper.c
      helper.h
      helper.o
      Makefile
      manage_client.c
      manage_client.h
      manage_client.o
      reliable_udp.c
    OUTLINE
    TIMELINE

TERMINAL
  1: server, client
  -----
  Welcome to the server

  Menù:
  list: to visualize the files available for download
  get: to request the download of a certain file
  put: to upload on server a certain file
  help: to show the menù

  list
  Files in the current directory:
  WhereGo-video.mp4
  ricerca.pdf
  plrabn12.txt
  Directory for server's files
  prova2mb.pdf
  alice29.txt
  -----END-----

  get
  Enter the name of the file you want to download: plrabn12
  .txt
  File size is: 481861
  Received 150000 new bytes...
  150000 / 481861 bytes written...
  Received 150000 new bytes...
  300000 / 481861 bytes written...
  Received 150000 new bytes...
  450000 / 481861 bytes written...
  Received 31500 new bytes...
  481500 / 481861 bytes written...
  Received 361 new bytes...
  481861 / 481861 bytes written...
  File transfer complete!

  150000 / 481861 sent...
  Sent 150000 bytes
  300000 / 481861 sent...
  Sent 150000 bytes
  450000 / 481861 sent...
  Sent 31861 bytes
  481861 / 481861 sent...
  file transfer completed
  Waiting client request...

  WhereGo-video.mp4
  ricerca.pdf
  plrabn12.txt
  Directory for server's files
  prova2mb.pdf
  alice29.txt
  file listing completed
  Waiting client request...
  command GET entered
  file name is plrabn12.txt
  opening file, path: server_files/plrabn12.txt
  Opened fd 7
  Sent OK
  Size : 481861, converted : 4997314433102381056
  Reading from fd 7
  Sent 150000 bytes
```