Relazione progetto IIW 17/18
Reliable UDP

Calicchia Cristiano 0231938
Scaccia Flavio 0230163
Swid Marco 0227134

Indice

1.Introduzione

Il progetto da noi sviluppato riguarda la realizzazione di un'architettura client-server che utilizza il protocollo UDP per l'invio di pacchetti.

L' UDP è un protocollo del livello di trasporto non orientato alla connessione e il cui trasferimento non è affidabile. La nostra applicazione implementa l'orientamento alla connessione e l'affidabilità nel trasferimento a livello di applicazione.

2. Scelte progettuali e architettura

2.1 Orientamento alla connessione

Per gestire al meglio la nostra applicazione, abbiamo deciso di creare una connessione persistente per ogni client connesso al server. Il server è unico, mentre i client sono molteplici, quindi il server è in grado di gestire più connessioni contemporaneamente (server di tipo concorrente) e più trasferimenti di pacchetti in contemporanea con ogni singolo client (client di tipo concorrente). La connessione con ogni client cessa nel momento in cui esso dichiara di aver terminato i propri compiti grazie al comando QUIT o quando risulta inattivo per almeno 300 secondi, parametro da noi scelto arbitrariamente.

2.2 Trasferimento dati affidabile

Il protocollo UDP non garantisce né che i pacchetti inviati arrivino al destinatario né che, una volta arrivati, siano salvati nel giusto ordine. Come da traccia, abbiamo ottenuto il trasferimento dati affidabile implementando una nostra versione dell' algoritmo Selective Repeat. Questo è un algoritmo ARQ che utilizza numeri di sequenza per ordinare i pacchetti, i quali vengono inviati in un blocco di dimensione fissa chiamato finestra. Inoltre per ciascun pacchetto si attende una conferma di ricezione chiamata ACK. Ogni qualvolta il pacchetto con numero di sequenza minore ancora in attesa di conferma riceve l'ACK, la

finestra scorre permettendo l'invio dei pacchetti successivi. Sia mittente che destinatario possiedono la propria finestra di uguale dimensione, ma capace di scorrere in maniera diversa. Si ha quindi la necessità di coordinare le trasmissioni e ritrasmissioni dovute alle perdite di pacchetti e gestite da meccanismi di timeout. La nostra implementazione dell'algoritmo verrà descritta successivamente. Utilizzandolo per l'invio di un file, possiamo essere sicuri che questo venga ricevuto correttamente.

2.3 Architettura client-server

Ciascun client, una volta avviato, stabilisce una connessione con il server, già avviato precedentemente. A questo punto, sono disponibili 4 comandi:

1. List: per ottenere una lista dei file presenti nel server
2. Get: per scaricare un file dal server
3. Put: per caricare un file nel server
4. Quit: per interrompere la connessione con il server

Inoltre, è presente il comando Help, che fornisce all'utente una lista dei comandi disponibili.

2.4 Configurabilità dei parametri

Avviando gli eseguibili da linea di comando con il flag '-c' o '-config', è possibile entrare nella modalità di configurazione dei parametri prima di avviare la sessione del server o del client. È quindi possibile configurare alcuni parametri, che rimarranno tali fino alla chiusura del processo e che influenzeranno il comportamento del trasferimento di dati. Il client può modificare l' indirizzo IP del server con cui connettersi e la porta su cui contattarlo. Il server può modificare la porta su cui rimanere in ascolto di nuove connessioni e alcuni parametri relativi alla selective repeat, quali finestra di spedizione, probabilità di perdita e timeout.

2.5 Piattaforme software utilizzate

L'applicazione è stata testata sulle seguenti piattaforme software:

1. Terminale di Linux Mint 19, con compilatore gcc, IDE Atom;
2. Terminale di macOS Mojave 10.14.2, con compilatore gcc, IDE Xcode.

3. Implementazione

3.1 Strutture e funzioni utilizzate

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

- la struttura pkt : utilizzata per rappresentare i diversi tipi di pacchetti, tra cui i diversi comandi, gli ACK e i dati, assegnando a ciascuno un numero di sequenza durante i trasferimenti di file, rendendo possibile l' ordinamento durante la ricezione;

```
typedef struct pkt {
  short int type;
  int n_seq;
  char data[PKT_SIZE - (sizeof(short int) + sizeof(int))];
} t_pkt;
```

- la struttura circular_buffer : un buffer circolare utilizzato per rappresentare la finestra d'invio durante l'algoritmo della selective repeat; in esso vengono salvati i pacchetti inviati di cui si attende ancora un ACK; Il circular buffer è un buffer che utilizza due indici: il primo indica sempre il pacchetto che è stato trasmesso per primo tra quelli all'interno della finestra, quindi in attesa di ACK, mentre il secondo indica la prima posizione libera all' interno del buffer. Il buffer viene, appunto, riempito in modo circolare, cioè una volta arrivati alla fine, si torna all' inizio;

```
typedef struct circular_buffer {
  int expct;
  int snd;
  t_pkt *buff_pkts;
} cb_window;
```

- funzioni per il timeout : tramite le funzioni *set_timeout* e *get_timeout* impostiamo il timeout in microsecondi alle socket, cioè il massimo tempo di attesa di un pacchetto durante una recvfrom. Entrambe utilizzano le funzioni di sistema *setsockopt* e *getsockopt*, le quali modificano le opzioni delle socket. Inoltre, la funzione *set_timeout_sec* lo imposta in secondi utilizzando lo stesso procedimento.
- funzione rand_send: dato che in locale non avvengono mai perdite di pacchetti, per simularla in modo randomico così da testare l'effettivo funzionamento del nostro algoritmo di trasferimento file, abbiamo implementato questa funzione. Generando un numero pseudo-casuale, opportunamente fornito dalla funzione *rand* precedentemente avviata con il proprio seed in ogni processo, e confrontandolo con la probabilità di perdita scelta, viene deciso se inviare o no il pacchetto;

```
void rand_send(int sockfd, t_pkt *pkt, struct sockaddr_in *addr, float loss_prob){
    float random_prob = (float)rand() / (float)RAND_MAX;
    if(random_prob > loss_prob) {
        if(sendto(sockfd, pkt, sizeof(*pkt), 0, (struct sockaddr *)addr, sizeof(*addr)) < 0) {
            perror("sendto pkt in rand_send");
            exit(EXIT_FAILURE);
        }
    }
}
```

- la funzione create_socket : utilizzata per creare di volta in volta una nuova socket, associandole una porta libera casuale scelta dal sistema;
- funzioni di gestione file: utilizzate durante la lettura e scrittura di file, il loro indirizzamento nelle cartelle, il controllo della loro esistenza e per conoscere la loro dimensione;
- funzioni di configurazione: sia nel server che nel client, come precedentemente descritto, ci sono dei parametri configurabili avviando l'eseguibile con '-c' o '-config'. Quando ciò accade, viene mostrata una semplice interfaccia grafica che mostra l'attuale valore del parametro considerato e permette di modificarlo per la sessione corrente;
- sig_handler : funzioni di gestione dei segnali inviati dai figli, per evitare che rimangano processi zombie;

3.2 Parametri di default e configurabili

Nel file *comm.h* sono definiti tutti i parametri di default del nostro programma. Tra i più importanti ci sono:

- PKT_SIZE : la dimensione dei pacchetti, da noi scelta uguale a 1500 byte, che comprendono il tipo, il numero di sequenza e i dati;
- SERV_IP e SERV_PORT : l' IP e la porta di default del server, configurabili in run-time come precedentemente spiegato;
- LOSS_PROB : la probabilità di perdita di default, impostata al 20% e configurabili in run-time come precedentemente spiegato;
- WINDOW : la dimensione di default della finestra di trasmissione nell' algoritmo Selective Repeat, inizialmente impostato a 20 e configurabile in run-time, come precedentemente spiegato;
- TIMEOUT_PKT : il valore di default del timeout delle socket, cioè il tempo di attesa massimo in una recvfrom, inizialmente impostato a 5000 microsecondi = 5 millisecondi e configurabile in run-time; è possibile scegliere un altro valore statico o adattativo, il quale verrà spiegato successivamente in dettaglio;
- WRITABLE_BUFF : la dimensione in KByte del buffer cache di scrittura su file, utilizzato per limitare gli accessi in scrittura, velocizzando il processo; cambiando questo parametro nel codice, è possibile migliorare o peggiorare la velocità di trasferimento dei file;

3.3 Connessione

Avviando il server, il processo iniziale crea una socket, associandola ad un indirizzo, definito in C da una struttura contenente il numero di porta e il protocollo da utilizzare, nel nostro caso l' UDP. A questo punto, si mette in ricezione di richieste di connessione tramite la funzione *reliable_accept*, la quale entra in una *recvfrom* e aspetta pacchetti di tipo SYN.

Avviando un client questo crea una socket, tramite la quale nella funzione *reliable_connection* invia al server un pacchetto di tipo SYN

all'indirizzo definito dai parametri da lui configurabili, precedentemente descritti : IP e porta. Quando il server riceve il SYN, effettua una fork(), generando un processo figlio che da questo momento sarà l'unico responsabile delle comunicazioni con il client appena arrivato. Per farlo, il figlio crea una nuova socket, comunicando attraverso una nuova porta precedentemente libera. Il processo padre torna nuovamente in ascolto di nuove richieste di connessione, mentre il figlio continua l'instaurazione della stessa. Quest'ultimo invia un pacchetto di tipo SYNACK, dentro al quale vengono inseriti i parametri caratterizzanti delle future trasmissioni di file, quali finestra di invio, probabilità di perdita e timeout. Successivamente si rimette in attesa di un ACK, per decretare l'avvenuta connessione. Se scadesse il timeout prima della ricezione dell'ACK, verrebbe ritrasmesso il SYNACK un numero limitato di volte, superato il quale il client sarebbe dichiarato inattivo e il processo figlio terminato. Viceversa, se il client non ricevesse il SYNACK, dopo un numero limitato di volte il server sarebbe dichiarato inattivo e il processo del client verrebbe chiuso.

```
for(int tries = 5; tries > 0; --tries){
    //funzione che invia il pacchetto in maniera probabilistica
    rand_send(new_sockfd, synack_pkt, cliaddr, loss_prob);
    errno = 0;
    if(recvfrom(new_sockfd, ack_pkt, sizeof(*ack_pkt), 0,
                (struct sockaddr *)cliaddr, len) < 0 && errno != EAGAIN) {
        perror("recvfrom - listening ack conn error");
        _exit(EXIT_FAILURE);
    }
    if(errno == EAGAIN){
        continue;
    }
    if(ack_pkt->type == ACK){
        free(ack_pkt);
        free(synack_pkt);

        return new_sockfd;
    }
}
```

Invece, una volta ricevuto il SYNACK, salva i parametri inviati e manda un ACK, rimanendo in ascolto di eventuali SYNACK ripetuti per un periodo di tempo limitato. Se venisse ricevuto un altro SYNACK, il client invierebbe nuovamente l'ACK, altrimenti assumerebbe che la connessione sia andata a buon fine e inizierebbe la routine di lettura di comandi e invio al server, tramite l'interfaccia fornita da *menu_cli*. Una volta ricevuto l'ACK, il processo figlio del server si mette in attesa di comandi da parte del client a lui associato.

3.4 Comandi

Le azioni disponibili durante l'esecuzione di *menu_cli* per l'utente sono quelle precedentemente descritte: list, get, put, quit e help. Le prime quattro corrispondono ai comandi che il client può inviare al server.

```
-> Socket created, connecting to [ip = 127.0.0.1]

Connection established
*****************************
CONFIGURATION PARAMETERS
Window = 10
Loss probability = 0.200000
Timeout data = 5000
*****************************

Choose what you want to do
   [L]IST ; [G]ET ; [P]UT ; [Q]UIT ; [H]ELP
```

Di seguito viene descritto in particolare il flusso di esecuzione di ciascuno di essi. I primi tre possono anche essere eseguiti più volte in contemporanea sullo stesso client e, dato che per lo svolgimento è necessario lo scambio di dati tramite socket, abbiamo ritenuto

necessaria la creazione di un processo figlio con una nuova propria socket associata per ogni nuovo trasferimento dati. In questo modo ciascun processo legge solamente dati dalla propria socket, evitando che i dati si mischino e ottenendo la natura concorrente del client da noi voluta. Il processo padre tornerà in ascolto sullo standard input per nuovi compiti da eseguire, mentre ciascun figlio chiuderà la propria socket e terminerà una volta finito il trasferimento ad esso assegnato.

3.4.1 List

Il comando list viene utilizzato per ricevere la lista dei file presenti nel server. In quest'ultimo, viene ricavata una lista dei file presenti tramite la funzione *files_from_folder*. La lista dei file, sotto forma di stringa, viene inserita in un file temporaneo nel server. Successivamente l'algoritmo della selective repeat, a cui passiamo il file da inviare, lo trasmette al client, che lo salva in un altro file temporaneo. A questo punto il client effettua la cat del file tramite la funzione di sistema *system*, mostrando su standard output all'utente i dati ricevuti. I file temporanei vengono opportunamente eliminati una volta compiuto il proprio compito.

3.4.2 Get

Il comando get viene utilizzato per scaricare sul client un file presente nel server. Nel caso in cui venisse richiesto un file non esistente, il server invierebbe un pacchetto di errore e il client mostrerebbe all'utente un messaggio che comunica la non esistenza del file. Quando l'utente inserisce il nome del file da scaricare, il processo genera un figlio, il quale prepara un pacchetto di tipo GET, con il nome desiderato nel campo riservato ai dati. Prima di mandare il pacchetto, controlla il nome con cui verrà salvato il file, per evitare

che ci siano file con lo stesso nome. Se venisse trovato un file che ha già quel nome, verrebbe aggiunto un indice tra parentesi (e.g. prova(1).txt), gestendo la presenza di eventuali estensioni. A questo punto, viene inviato il comando e si entra in fase di ricezione della selective repeat senza attendere il relativo ACK. Ciò avviene poiché, se si ricevessero dei dati, vorrebbe dire che il comando è stato ricevuto, altrimenti vorrebbe dire che è andato perso e verrebbe reso necessario il reinserimento del comando da parte dell'utente. In questo modo si utilizza una sorta di *piggy-back* ACK, limitando il numero di messaggi da inviare. Quando il server riceve il messaggio di get, controlla se il file richiesto esista: qualora non esistesse, verrebbe mandato un messaggio di errore, che nel client verrebbe gestito all'interno della funzione di ricezione della selective repeat, altrimenti inizierebbe il trasferimento tramite la funzione di invio.

3.4.3 Put

Il comando put viene utilizzato per caricare sul server un file presente nel client. Se il file da caricare non fosse presente sul client, l'utente riceverebbe un messaggio d'errore e dovrebbe inserire di nuovo il comando put con un altro nome. Quando l' utente inserisce il nome di un file esistente, il processo figlio crea il pacchetto di tipo PUT con il nome scelto nel campo riservato ai dati. Il pacchetto viene mandato, ma, diversamente dalla get, il processo si mette in attesa di ACK, poiché, senza riceverne uno, dovrebbe iniziare a mandare i dati senza sapere se il server è in ascolto, perdendo tempo e risorse. Se l'ACK non arrivasse, rimanderebbe per un numero limitato di volte il comando. Se lo scambio comando-ACK non andasse a buon fine, il server sarebbe decretato inattivo, altrimenti il trasferimento del file inizierebbe tramite l' algoritmo d' invio della selective repeat. Quando il server riceve un pacchetto di tipo PUT, controlla il nome del file che

deve essere caricato. Se esistesse già un file con questo nome, verrebbe salvato con un indice, come precedentemente spiegato nella get. Per limitare il numero di copie dello stesso file, l'indice massimo è stato da noi scelto pari a 9. Una volta superato, il client riceverebbe un messaggio di errore, che verrebbe opportunamente gestito. Una volta inviato l'ACK, il server entra nella funzione di ricezione della selective repeat. Se l'ACK venisse perso, il client rimanderebbe il comando PUT. Questa situazione è gestita all'interno della funzione appena nominata, la quale, una volta ricevuto un ulteriore pacchetto PUT, invierebbe nuovamente l'ACK. Quest'ultimo avrebbe numero di sequenza uguale a -1, per evitare che venga scambiato con l'ACK di un successivo pacchetto di dati tornando in ascolto.

3.4.4 Quit

Il comando quit viene utilizzato per chiudere la connessione con il server. Quando l'utente intende concludere la propria sessione di utilizzo, inserisce questo comando, che viene completamente gestito dal processo padre nel client. Questo prepara un pacchetto di tipo QUIT e lo manda al server tre volte, per aumentare la probabilità che il server lo riceva. Una volta inviato, la propria socket viene chiusa e il processo padre termina. Quando il processo nel server associato ad un determinato client riceve un comando di tipo quit, capisce che il client ha terminato la propria esecuzione, quindi chiude la socket su cui è in ascolto di comandi e termina la propria esecuzione. Questo approccio non garantisce che il server riceva i comandi di tipo quit, che vengono inviati una sola volta, senza attesa di ACK. Questa scelta è stata fatta per rendere più fluido il funzionamento della connessione dal punto di vista dell' utente, il quale, una volta finito il lavoro, può chiudere la connessione senza aspettare eventuali ACK. Se si dovesse aspettare l'ACK, la chiusura verrebbe rallentata in caso

di rete satura, dovendo tentare successivi invii di QUIT e aspettando ulteriore tempo. Invece, in questo modo, il client viene chiuso istantaneamente, mentre, nel caso peggiore, il processo figlio nel server verrà chiuso allo scadere del timeout di 300 secondi dall'ultimo comando, che decreterebbe l'inattività del client.

3.5 Selective Repeat

L'algoritmo della selective repeat è diviso in due parti: una parte gestisce l'invio del file, l'altra gestisce la ricezione. In base all'operazione da svolgere, il client e il server chiamano l'una o l'altra. Per ottenere il trasferimento affidabile, i pacchetti vengono numerati con un numero di sequenza, che poi viene usato per riordinarli in fase di ricezione. Per evitare l'utilizzo di numeri sempre più grandi e per utilizzare le strutture di appoggio da noi scelte, abbiamo deciso di riutilizzare i numeri di sequenza, secondo una formula ben nota, secondo cui i numeri di sequenza possono essere riutilizzati con una frequenza pari a due volte la dimensione della finestra d'invio più uno. Questa dimensione garantisce che i pacchetti passati non vengano scambiati per pacchetti ancora da inviare o da ackare. All'assegnazione dei numeri di sequenza, questi vengono incrementati di uno e poi viene calcolato il modulo rispetto al valore massimo. Allo stesso modo, gli indici dei buffer utilizzati vengono riportati tra zero e il valore massimo. Durante il trasferimento, quando si verificano un gran numero di scadenze del timeout successive, viene decretata l' inattività dell' altro processo.

3.5.1 Sender Side

La funzione *sr_snd_side* è la responsabile dell'invio di pacchetti tramite l'algoritmo di selective repeat da noi implementato. Come argomento, oltre ai parametri d'invio, riceve il file da inviare, che,

come prima cosa viene aperto. Per tenere traccia della finestra d'invio, cioè il numero di pacchetti massimo che possono essere trasmessi in rete e possono rimanere in attesa di ACK in contemporanea, viene utilizzato il circular buffer precedentemente descritto. Il file viene aperto in lettura e viene calcolata la sua dimensione. Il primo pacchetto inviato è di un tipo diverso e contiene la dimensione del file, in modo tale da far conoscere anche al ricevente la sua dimensione e per gestire la durata della routine dell'algoritmo. Questo pacchetto viene inserito nel buffer e trattato come tutti gli altri, quindi viene ritrasmesso fino a quando non viene ricevuto il suo ACK. Iniziata la routine, fino a quando ci sono spazi liberi nel nostro buffer, leggiamo dal file i dati pari alla dimensione del campo dati del pacchetto, riempiamo un nuovo pacchetto, lo mettiamo nel buffer e lo inviamo. Questo processo va avanti fino a quando ci sono spazi liberi nel buffer o fino a quando si arriva alla fine del file e si legge l'ultimo pacchetto da inviare. In entrambi i casi, si entra nella fase di ricezione dell'ACK. Per tenere conto dei pacchetti trasmessi di cui non si è ancora ricevuto l'ACK, viene utilizzato un array di dimensione pari al numero di sequenza massimo, cioè due volte la finestra più uno. Gli elementi di questo array possono valere o -1, che significa 'in attesa di ACK', o 1, che significa 'ACK ricevuto'. Quando un pacchetto viene mandato la prima volta, il suo rispettivo valore nell'array viene posto a -1 e quando il suo ACK viene ricevuto, viene posto a 1. In questo modo, possiamo tenere traccia dei pacchetti non ackati in ordine. Se durante la fase di ricezione degli ACK scadesse il timeout della recvfrom, vorrebbe dire che qualche pacchetto è andato perso, quindi vengono rimandati tutti i pacchetti presenti nella finestra che ancora non sono stati ackati. Quando viene ricevuto l'ACK del pacchetto più vecchio ancora in attesa di ACK, vuol dire che la finestra può avanzare. Per simulare l'avanzamento della finestra, il primo indice del circular buffer, che indica proprio il

pacchetto considerato, viene incrementato di tante posizioni quanti sono gli ACK ricevuti in ordine. Quindi l'incremento termina quando nell' array viene incontrato il primo -1.

```
do {
  acked++;
  wb->expct = (wb->expct + 1) % (window + 1);
  expct_n_seq = (expct_n_seq + 1) % (capacity);

  //aggiorno valori nel buff per evitare errori
  if(expct_n_seq - 1 < 0){
    buff_ack[capacity - 1] = -1;
  } else{
    buff_ack[expct_n_seq - 1] = -1;
  }
  buff_ack[(expct_n_seq + window - 1)%(capacity)] = -1;
  buff_ack[(expct_n_seq + window)%(capacity)] = -1;
} while(buff_ack[expct_n_seq] != -1 && acked != total_pkt);
```

Spostando la finestra, vengono aggiornati i valori nell'array per evitare errori nei successivi scorrimenti. La routine inizia nuovamente, effettuando le nuove trasmissioni se possibile o attendendo ulteriori ACK. Quando tutti i pacchetti da trasmettere sono stati ackati, vuol dire che il ricevente ha ricevuto il file completo. A questo punto, viene iniziata la fase di chiusura della trasmissione. Viene inviato un pacchetto di tipo FIN, gestendo le eventuali ritrasmissioni, fino a quando non viene ricevuto un FINACK che rappresenta l'effettiva terminazione con successo della trasmissione.

3.5.2 Receiver Side

La funzione *sr_rcv_side* è la responsabile della ricezione di pacchetti tramite l'algoritmo di selective repeat da noi implementato. Come argomento, oltre ai parametri d'invio, riceve il nome del file da ricevere che, come prima cosa, viene creato e aperto in scrittura; questo file verrebbe cancellato se la trasmissione non andasse a buon

fine. Per tenere traccia della finestra di ricezione, uguale a quella d'invio, utilizziamo due interi, i quali assumono il valore degli estremi attuali della finestra. Inoltre, siccome i pacchetti non arrivano sempre in ordine, essi vengono salvati in un buffer di pacchetti di grandezza pari a due volte la finestra più uno, cioè il massimo numero di sequenza. Quando un pacchetto viene ricevuto, viene messo nell'apposita posizione, determinata proprio dal suo numero di sequenza: in questo modo, i pacchetti vengono ordinati. Inoltre, per evitare di effettuare un gran numero di accessi in scrittura sul file, utilizziamo un ulteriore buffer di dati, che viene utilizzato come memoria cache. Quando questo buffer si riempie o non ci sono più pacchetti da ricevere, il contenuto viene scritto su file ed esso viene svuotato. Terminata l'inizializzazione di queste strutture, la routine dell'algoritmo ha inizio. Ogni iterazione, come prima cosa, si attende la ricezione di un nuovo pacchetto, che può essere di diversi tipi. In base al tipo di pacchetto ricevuto, l'algoritmo effettua alcune operazioni. Gli scenari possibili sono i seguenti:

- DATA_SIZE : contiene la dimensione del file che si sta ricevendo. Questa viene salvata per ricavare il numero totale di pacchetti da ricevere, viene inviato l'ACK e la finestra di ricezione viene spostata se è il primo pacchetto di questo tipo ricevuto, altrimenti si invia solamente l'ACK;
- ERROR : il pacchetto contiene il messaggio d'errore da mostrare, che viene stampato. Il file viene rimosso e la trasmissione termina;
- PUT : come spiegato precedentemente nel paragrafo 3.4.3, la ricezione di un pacchetto di questo tipo indica che l'ACK inviato dal server per la put è andato perso. E' quindi necessario rinviarlo, per poi iniziare l'iterazione successiva;
- DATA : contiene i dati effettivi da scrivere sul file. Come prima cosa viene inviato l'ACK corrispondente, successivamente viene

controllato se il pacchetto ricevuto è nella nostra finestra di ricezione, altrimenti si tratterebbe di una ritrasmissione causata da una perdita dell'ACK, che abbiamo già rinviato, quindi non sarebbero necessarie ulteriori operazioni. Nel caso in cui il numero di sequenza del pacchetto risultasse essere all'interno della finestra, questo verrebbe salvato nel nostro buffer di pacchetti nella posizione determinata dal numero e successivamente verrebbero distinti due casi:

1. il numero di sequenza non è il più piccolo all'interno della finestra, dunque questa non può essere spostata;
2. il numero di sequenza è quello più piccolo all'interno della finestra, quindi possiamo farla avanzare di tanti passi quanti sono i pacchetti con numeri consecutivi già ricevuti.

```
while(buff_pkts[left_range % capacity].n_seq != -1) {
  size_to_write = (file_size - total_written < pkt_size_data) ?
                        file_size - total_written : pkt_size_data;
  memcpy(buff + written, buff_pkts[left_range % capacity].data,
                                                    size_to_write);
  written =(written+size_to_write)%(WRITABLE_BUFF*pkt_size_data);
  if(written == 0){
    if(write_block(fd, buff, WRITABLE_BUFF * pkt_size_data) < 0){
      perror("write_block in sr_rcv_side");
      _exit(EXIT_FAILURE);
    }
    memset(buff, 0, WRITABLE_BUFF * pkt_size_data);
  } else if(size_to_write != pkt_size_data){
    if(write_block(fd, buff, written) < 0){
      perror("write_block in sr_rcv_side");
      _exit(EXIT_FAILURE);
    }
  }
  total_written += size_to_write;
  buff_pkts[left_range % capacity].n_seq = -1; //già inserito
  left_range = (left_range + 1) % capacity;
  right_range = (right_range + 1) % capacity;
}
```

Quando ciò accade, i dati dei pacchetti uscenti dalla finestra vengono concatenati al buffer di scrittura, il quale, se venisse

riempito o se i pacchetti fossero finiti, verrebbe svuotato, scrivendo tutto il contenuto sul file. Inoltre, quando i dati di un pacchetto vengono inseriti nel buffer di scrittura, il loro numero di sequenza nel buffer di pacchetti viene posto a -1, per evitare che il pacchetto venga considerato nuovamente;

- FIN : indica che il mittente ha ricevuto l'ACK di tutti i pacchetti inviati. Si deve quindi inviare il FINACK per confermare la chiusura della trasmissione. A questo punto, ci rimettiamo in attesa per un determinato periodo di eventuali FIN ripetuti, che indicherebbero la perdita del FINACK, che verrebbe quindi mandato nuovamente e il procedimento verrebbe ripetuto fino a quando non si riceveranno altri FIN. A questo punto la trasmissione viene conclusa, la routine termina e il file viene chiuso.

3.5.3 Timeout adattativo

```
void decrease_timeout(int sockfd){
    int timeout;
    if(static_time == 0){
        timeout = get_timeout(sockfd);
        if(timeout >= MIN_TIMEOUT + TIME_UNIT){
            set_timeout(sockfd, timeout - TIME_UNIT);
        }
    }
}
void increase_timeout(int sockfd){
    int timeout;
    if(static_time == 0){
        timeout = get_timeout(sockfd);
        if(timeout >= MAX_TIMEOUT - TIME_UNIT){
            set_timeout(sockfd, timeout + TIME_UNIT);
        }
    }
}
```

Il timeout è configurabile all'avvio del server, come precedentemente spiegato. Oltre alla possibilità di impostare un nuovo valore statico diverso da quello di default, è possibile renderlo di tipo adattativo. Questo tipo di timeout serve per agevolare la trasmissione di file sia in caso di rete satura, sia in caso di rete libera. Nel primo caso, si verificherebbe la perdita di tanti pacchetti, da noi simulabile impostando una probabilità di perdita alta, la quale farebbe scattare il timeout delle recvfrom molto spesso. Secondo la nostra implementazione, ogni volta che una recvfrom fa scattare il timeout, il suo valore viene aumentato di un'unità temporale, da noi posta pari a 0.5 millisecondi. Quando invece un pacchetto viene ricevuto, il suo valore viene decrementato della stessa quantità, rendendo più veloce il trasferimento nel caso in cui la rete diventasse libera. Per evitare valori troppo alti o troppo bassi, abbiamo individuato un valore minimo e un massimo per limitare i cambiamenti. Il valore minimo è stato scelto pari a 1 millisecondo, dato che nei test abbiamo constatato che sotto questo valore il trasferimento diventa problematico. Il valore massimo è pari a 999999 microsecondi, in quanto dimensione massima inseribile nelle funzioni da noi utilizzate.

4. Limitazioni

Dal punto di vista grafico, essendo il client concorrente e potendo inviare più comandi insieme, è possibile che durante l'inserimento di un comando da standard input venga effettuata una stampa su standard output di una o più richieste precedenti riguardo trasferimenti o errori. Ma, non essendo la grafica il punto principale del nostro progetto, abbiamo deciso di non impiegare ulteriori risorse.

Per quanto riguarda il salvataggio di file con lo stesso nome, in cui inseriamo un indice tra parentesi per differenziarli, abbiamo deciso che il numero massimo di copie salvabili sia 10 (e.g. da file.txt a

file(9).txt). Questo numero sarebbe facilmente sostituibile con un numero più grande, ma lo abbiamo ritenuto sufficiente per l'utilizzo che noi facciamo del programma.

In generale, tutte le variabili utilizzate per scambio di file, vengono allocate considerando file di dimensioni ragionevoli, senza superare i GB. Inoltre, il numero di client contemporaneamente attivi sul server è stato da noi lasciato libero, dato che le eventuali limitazioni dipenderebbero dalle caratteristiche della piattaforma su cui le applicazioni vengono eseguite. Provando a stressare in maniera notevole il sistema in locale, potrebbero verificarsi problemi, ma non abbiamo ritenuto opportuno imporre dei limiti statici sul numero di richieste o connessioni contemporanee.

5. Testing

Abbiamo effettuato tre tipi di test sulla nostra applicazione:

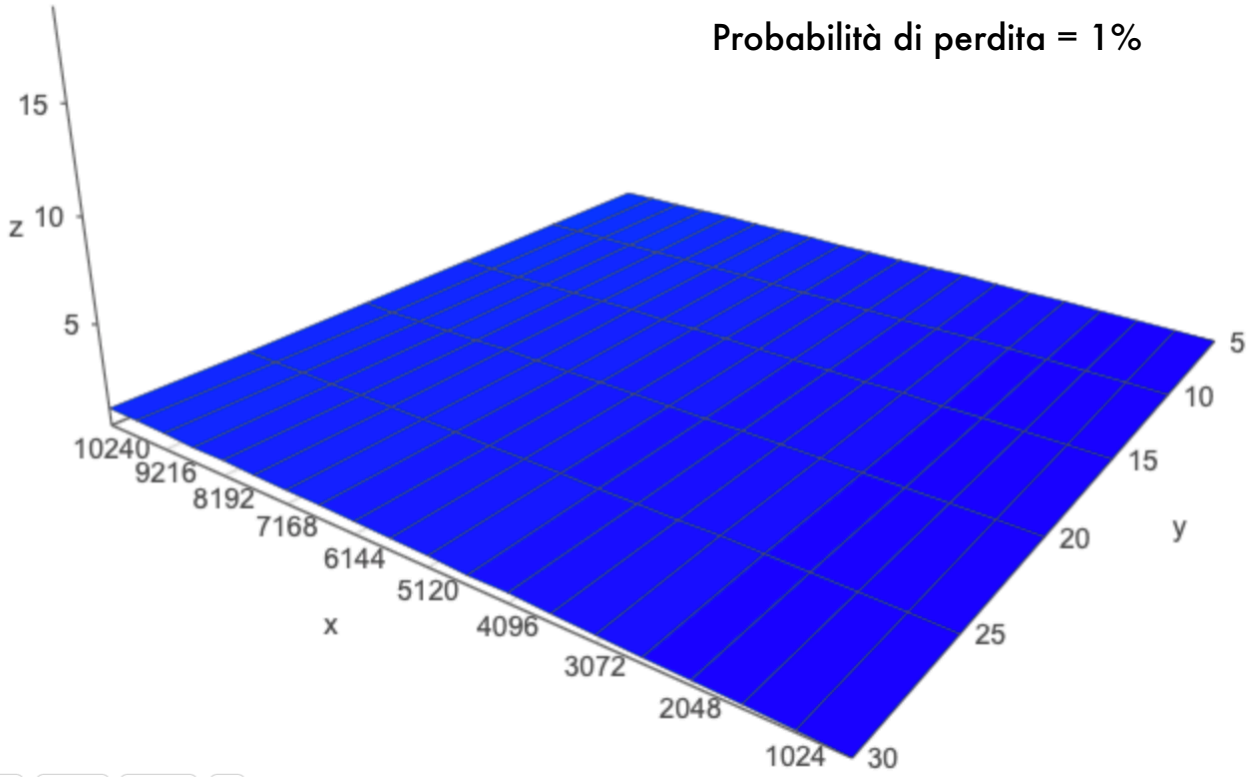
1. Sulle prestazioni al variare dei parametri di funzionamento;
2. Sulle capacità di gestione di diverse richieste client di trasmissione file, effettuate in concorrenza ad un singolo server;
3. Sulle prestazioni alla variazione del timeout delle recvfrom;

5.1 Test sui parametri

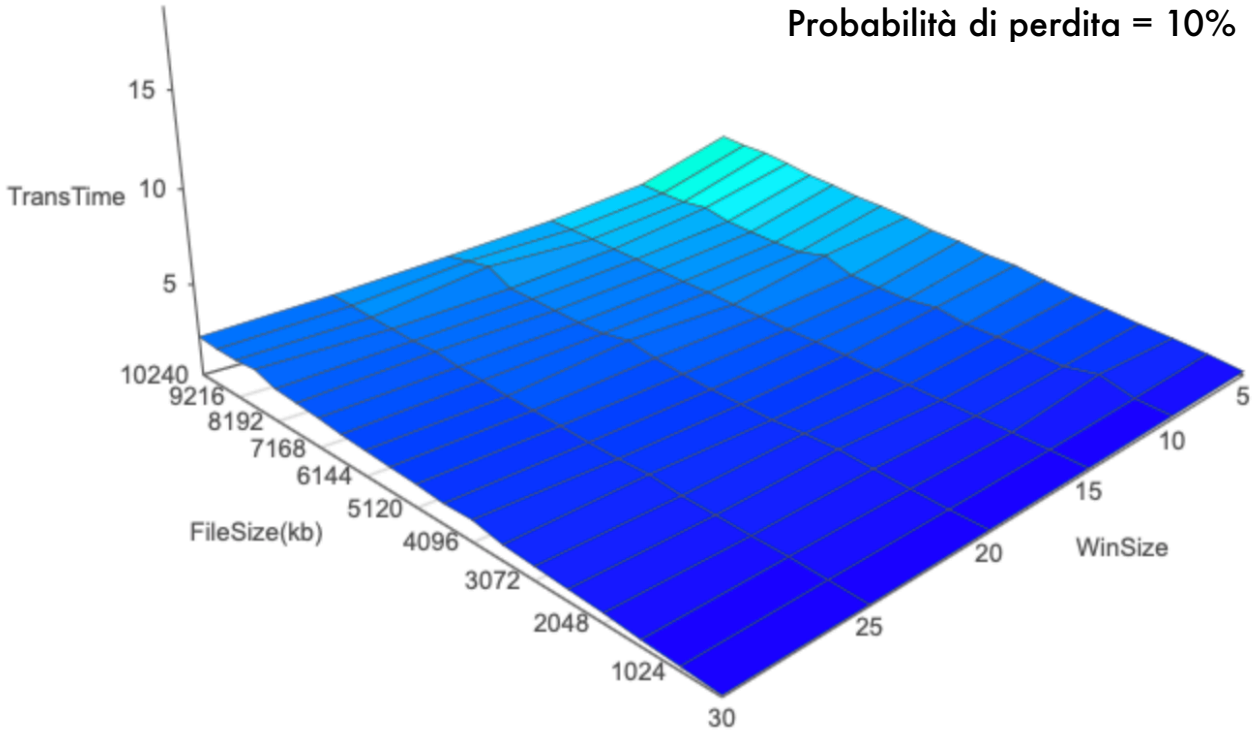
Questo test consiste nel variare i parametri in una tupla (FileSize, WinSize, TransTime, LossProb), che rappresentano in ordine: la dimensione del file, la dimensione della finestra, il tempo di trasferimento del file e la probabilità di perdita. Inoltre il timeout è stato lasciato fisso a 5 millisecondi. Come valori abbiamo preso FileSize da 512KB a 10240KB con passi da 512, WinSize da 5 a 30 con passi da 5 e probabilità di perdita 1%, 10%, 20% e 30%; il tempo di trasmissione viene ottenuto dai risultati del test. Per ottenere questi dati abbiamo anzitutto creato file di test con nome "file_X" dove

X sta per la dimensione in KB, ottenuti attraverso la chiamata da terminale "mkfile -n size[b|k|m|g] filename" (su Mac) oppure "fallocate -l size[b|k|m|g] filename" (su Linux). Abbiamo variato in ordine tutti i parametri della tupla in maniera annidata, registrando i tempi di trasmissione in un file CSV presente nella cartella Testing. Da questo file abbiamo elaborato un grafico 3D attraverso il servizio web "http://almende.github.io/chap-links library/js/graph3d/playground/", che ha generato quattro grafici.

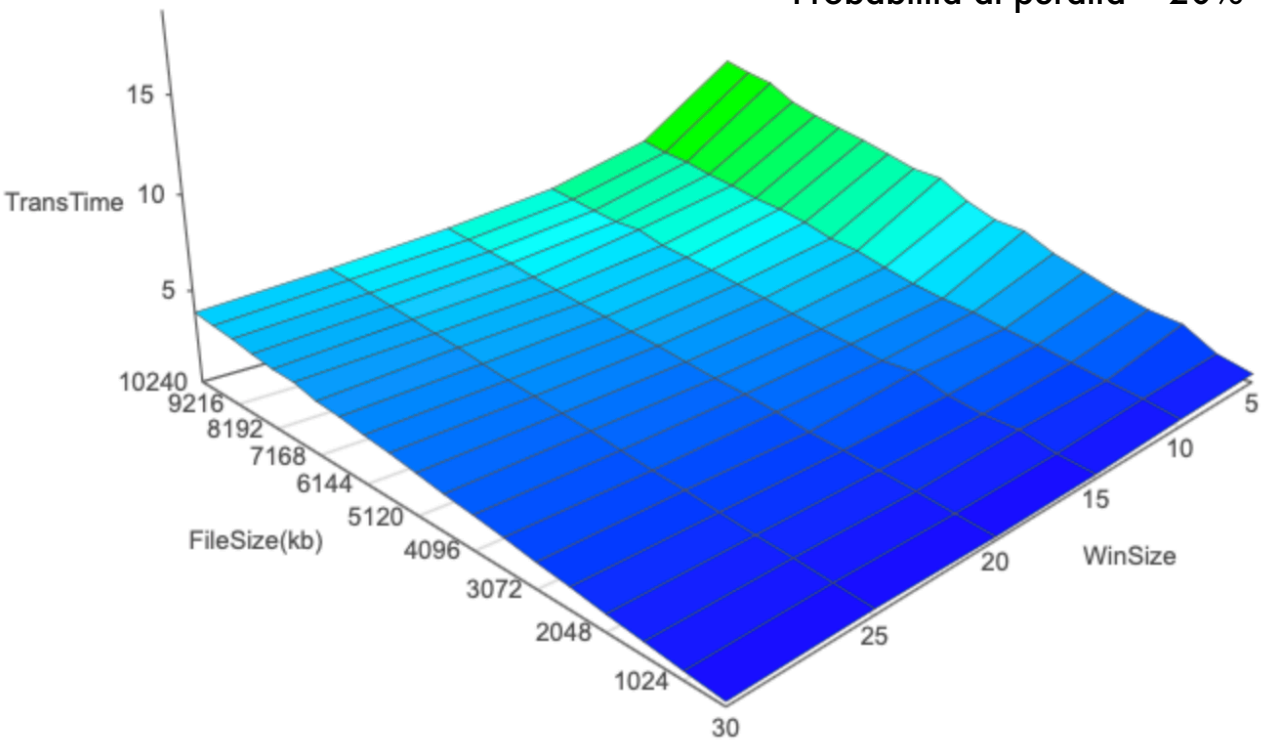
Probabilità di perdita = 1%



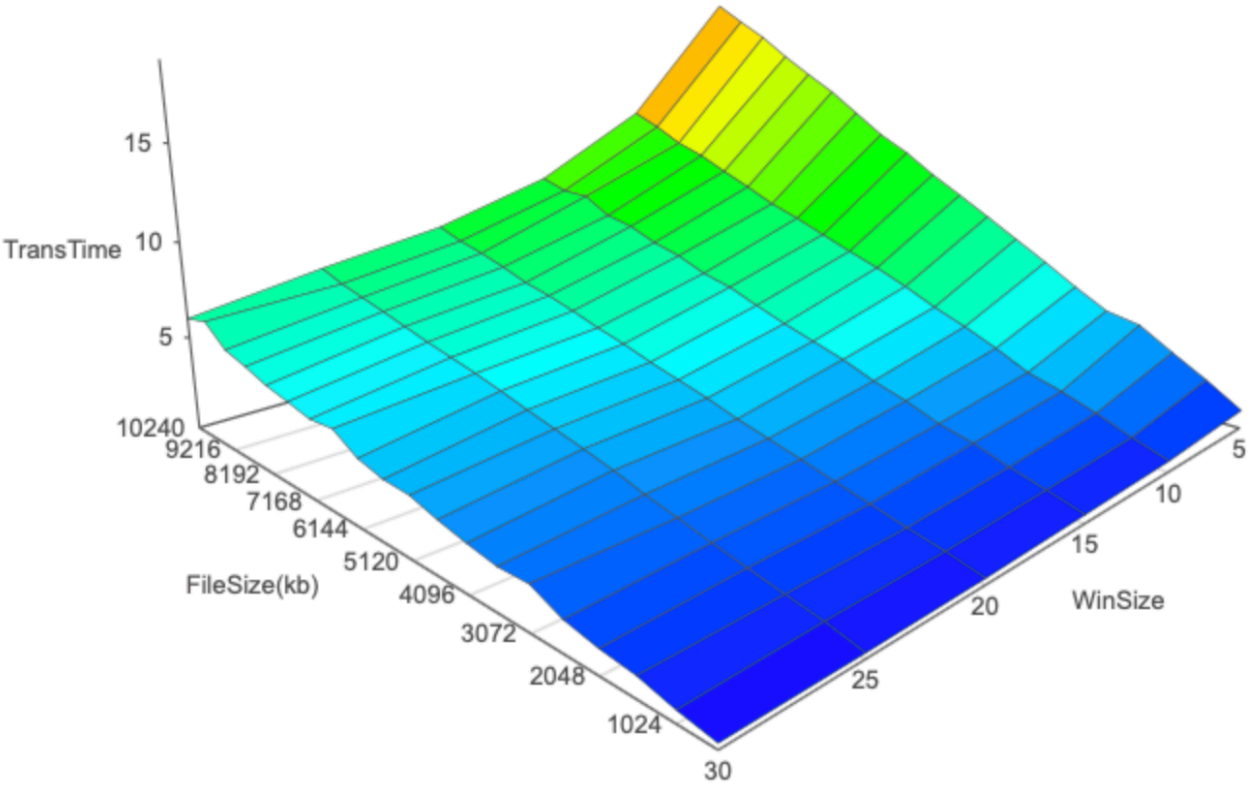
Probabilità di perdita = 10%



Probabilità di perdita = 20%



Probabilità di perdita = 30%

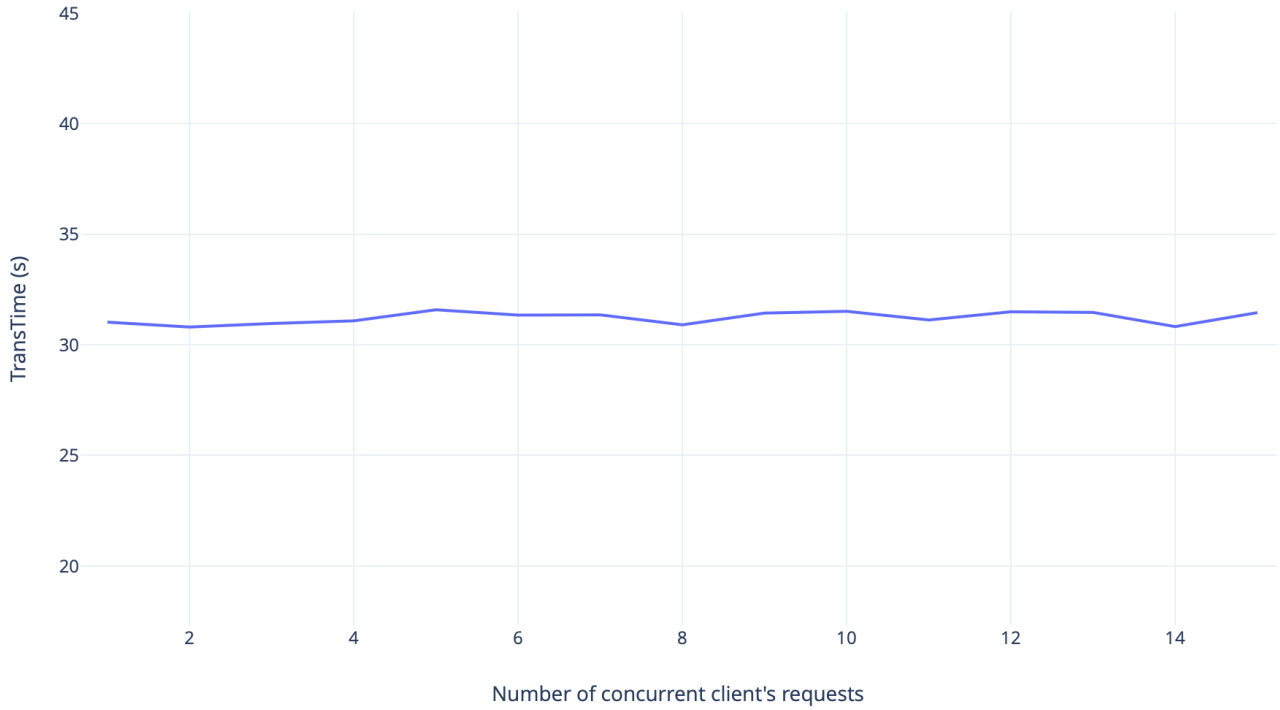


I grafici sono risultati coerenti al funzionamento previsto dall'applicazione. Nello specifico questi evidenziano che:

1. Dimensioni della finestra maggiori generano tempi di trasmissione minori;
2. Dimensioni del file maggiori generano tempi di trasmissione maggiori;
3. Probabilità di perdita maggiori generano tempi di trasmissione maggiori;

5.2 Test sulla concorrenza

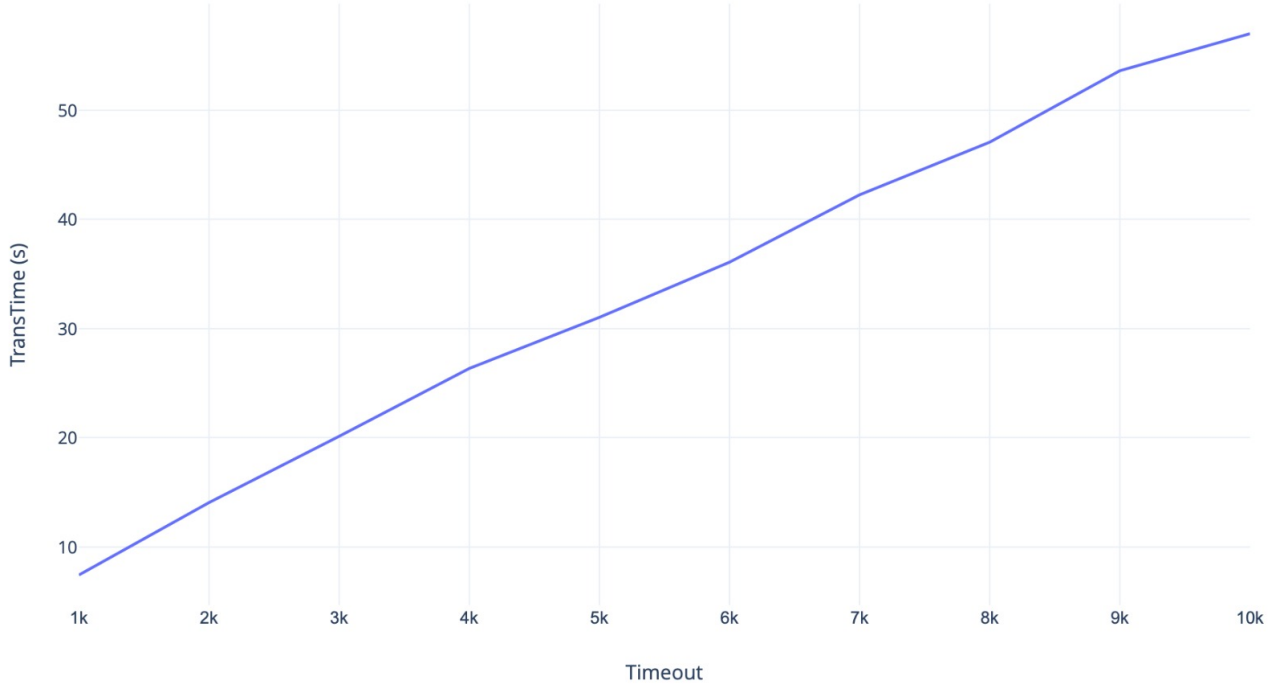
Per effettuare il test sull'invio di richieste client concorrenti abbiamo scelto parametri fissi: dimensione di file da 50MB, dimensione della finestra a 15, probabilità di perdita 20% e timeout a 5 millisecondi. Abbiamo eseguito quindi in parallelo le richiesta da 1 a 15 file della stessa dimensione, registrando i tempi di trasmissione. Abbiamo ripetuto il test più volte anche per lo stesso numero di client prendendo successivamente la media dei valori ottenuti.



Come si evince dal grafico, la nostra applicazione risulta pressoché stabile al variare del numero di richieste concorrenti.

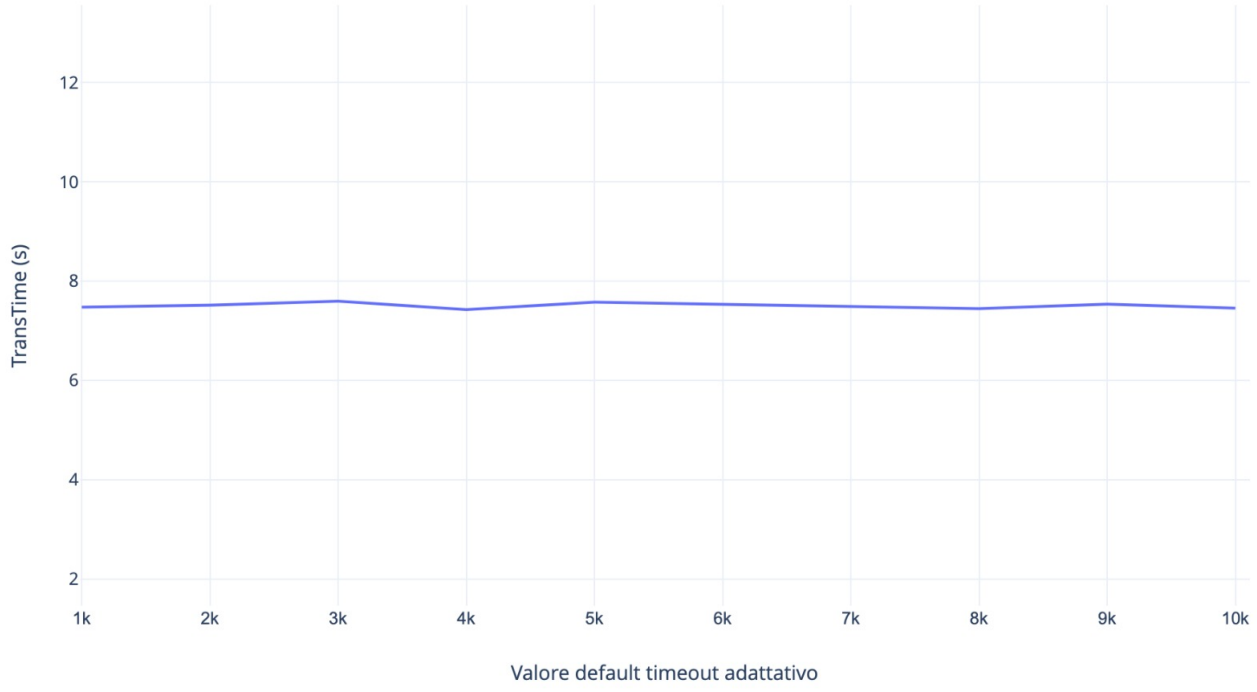
5.3 Test sul timeout

Per effettuare questi test, abbiamo tenuto fissi i seguenti parametri: dimensione del file a 50MB, dimensione della finestra 15, probabilità di perdita a 20%. Questa volta, la variazione è stata effettuata sul valore del timeout, registrando i tempi di trasmissione per generare il seguente grafico.



Come si evince dal grafico, aumentando il valore del timeout, il tempo di trasmissione aumenta notevolmente. Ciò avviene poiché si passa molto più tempo in attesa di pacchetti all' interno delle recvfrom prima di effettuare le ritrasmissioni dei pacchetti persi.

Per quanto riguarda l'utilizzo del timeout adattativo, abbiamo ripetuto gli stessi test precedenti, scorrendo comunque le probabilità di default, da cui il timeout adattativo parte inizialmente. Il grafico ottenuto è il seguente.



Come si evince dal grafico, l'utilizzo di questo strumento permette al trasferimento di avvenire in maniera molto più rapida rispetto al caso precedente, rispettando il suo scopo.

6. Manuale d'uso

Per utilizzare il programma, è necessario aprire due terminali all'interno della cartella in cui è presente il Makefile. Se nella cartella non sono presenti gli eseguibili, digitare il comando 'make'. Una volta effettuata la compilazione, scrivere './server [-c|-config]' in uno dei due terminali per avviare il server nella modalità base o per avviarlo nella modalità di configurazione. Nell'altro terminale digitare il comando './client [-c|-config]' per avviare il client nella modalità base o per avviarlo nella modalità di configurazione. Una volta seguite le eventuali istruzioni di configurazione in entrambi, è possibile utilizzare il client seguendo le indicazioni rappresentate sullo schermo. Per la lista completa dei comandi disponibili digitare 'h' o 'help'. Per terminare la connessione, utilizzare il comando quit digitando 'q' o 'quit'. Per far terminare il server, uccidere il processo premendo 'ctrl+c'.