Ingegneria di Internet e Web

Progetto A.A. 2019/2020

UDP-GBN

Massimo Mazzetti

0253467

INDICE

1. Traccia del Progetto

Reti di Calcolatori ed Ingegneria del Web - A.A. 2019/20

Progetto B1: Trasferimento file su UDP

Lo scopo de progetto è quello di progettare e implementare in linguaggio C usando l'API del socket di Berkeley un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione (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, deve 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 Go-Back 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, e la durata del timeout T, sono tre costanti configurabili ed uguali per tutti i processi. Oltre all'uso di un timeout fisso, deve essere possibile scegliere l'uso di un valore per il timeout adattativo calcolato dinamicamente in base alla evoluzione dei ritardi di rete osservati. 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).

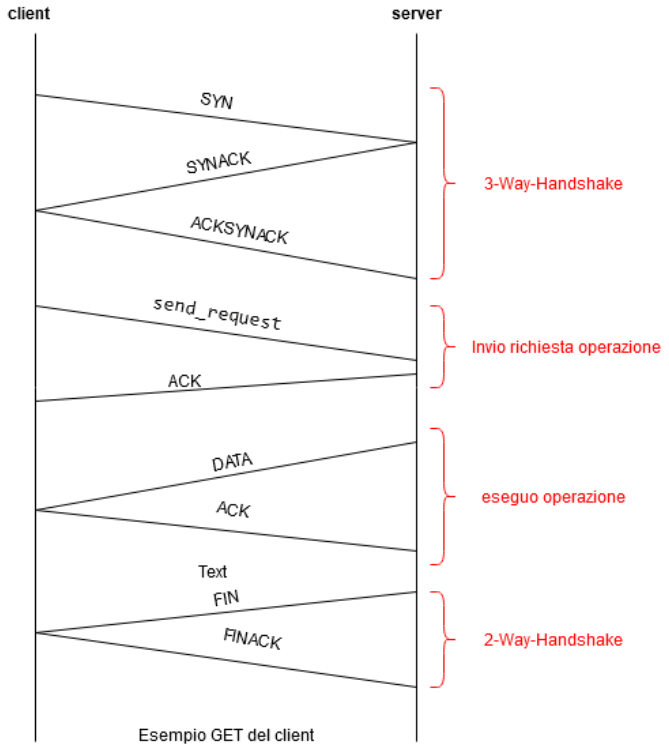
2. Architettura e scelte progettuali

L'applicazione presenta un'architettura di tipo Client-Server.

Il client e il server utilizzano il protocollo di livello 4 UDP.

La fruizione del servizio avviene in quattro fasi:

- Instaurazione della connessione tramite 3-Way-Handshake.
- Il client richiede l'operazione da eseguire (GET, PUT, LIST).
- Il server esegue la richiesta dal client attraverso un servizio di comunicazione reso affidabile attraverso l'implementazione del protocollo GBN.
- Chiusura della connessione tramite il 2-Way-Handshake al termine dell'operazione.



I messaggi scambiati sono di 4 tipi:

- Comandi;
- Dati;
- Risposta;
- Sincronizzazione;

Si è scelto un server concorrenziale a processi per semplicità sviluppo e manutenibilità in quanto difficilmente si incorrerà nella saturazione delle risorse di sistema; tuttavia, rimane il fatto che essa risulterà più lenta rispetto ad altre soluzioni a causa del cambio di contesto.

Al fine di permettere la rilevazione di connessioni morte è stato realizzato un sistema che consente un numero massimo di ritrasmissioni e un tempo massimo per inviare una richiesta da parte del client.

L'Instaurazione avviene, come già detto, tramite 3-Way-Handshake, cioè vengono scambiati 3 messaggi:

- SYN: Inviato dal client al server per la richiesta di connessione;
- ACKSYN: Inviato dal server al client come riscontro del SYN. Contiene l'informazione riguardante il numero di porta che sarà associata alla socket del figlio che servirà il client;
- ACKSYNACK, inviato dal client al server, riscontra il SYNACK e serve al server per avere conferma che il client è ancora presente ed ha ricevuto le informazioni necessarie per l'instaurazione della connessione, così se ciò non è vero il server può deallocare le risorse dedicate a quello specifico client e renderle di nuovo disponibili per altre eventuali richieste di connessione.

La richiesta dell'operazione avviene con 2 messaggi:

- Il client invia un dato contenente quale operazione svolgere.
- Il server invia un ACK per riscontrare quell'operazione e inizia ad eseguirla.

La chiusura della connessione, come già detto, tramite 2-Way-Handshake, cioè vengono scambiati 2 messaggi:

- FIN: inviato dal Client/Server che ha trasmesso la risorsa al termine dell'operazione con conseguente chiusura della connessione;
- ACKFIN: riscontro del FIN.

3. Implementazione

Struttura pacchi

I messaggi scambiati tra server e client come visto nella sezione precedente sono di quattro tipi (comandi, dati, risposta e sincronizzazione) e sono stati implementati tramite una struct data_packet.

```
struct data_packet {
    int type;
    long seq_no;
    int length;
    char data[MTU];
};
```

Essa viene utilizzata per lo scambio di dati, messaggi di comando, messaggi di sincronizzazione e messaggi di risposta. Per lo svolgimento di tale compito possiede quattro attributi:

- type (int), indica il tipo di messaggio; che può essere un comando (PUT, LIST, GET), un dato o una risposta (NORMAL) oppure un messaggio di sincronizzazione (SYN, FIN).
- seq_no (long), il quale indica il numero di sequenza del pacchetto, in caso di SYN indica un codice identificativo della richiesta.
- length (int), il quale indica la lunghezza in byte del dato nel campo “data”.
- data (char [MTU], dove MTU vale 1024), il quale contiene il dato che viene scambiato (chunk di file oppure un nome di file su server che può essere scaricato/caricato dal client).

Perdita simulata

Per simulare la perdita di messaggi è stata creata la funzione:

```
double value;
struct timespec tms;
if (clock_gettime(CLOCK_REALTIME, &tms))
{
  srand(time(NULL));
}
else
{
  int64_t micros = tms.tv_sec * 1000000;
  micros += tms.tv_nsec / 1000;
  if (tms.tv_nsec % 1000 >= 500)
  {
    ++micros;
  }
  srand(micros);
}


int r1 = rand() % 100;
int r2 = rand() % 100;
int r3 = rand() % 100;
unsigned short vec[3] = {r1, r2, r3};
seed48(vec);
value = drand48();

if (value < loss_rate) {
  return true;
}
return false;
```

La funzione prende in ingresso un parametro float relativo alla probabilità di perdita inserita dall'utente, ed in base ad esso, attraverso un procedimento di generazione di valori randomici per l'impostazione del seed della funzione drand48() della libreria stdlib.h, produce un valore float pseudorandomico nell'intervallo [0, 1). In base al valore ottenuto la funzione ritornerà un booleano per indicare se debba essere applicato o meno l'evento di perdita.

Server

L'implementazione del server si divide in quattro parti:

1. **Inizializzazione:**
   - **1.1.** Si prendono i parametri inseriti dall'utente e tramite questi si crea la socket (UDP) di ascolto utilizzando il numero di porta passato dall'utente e se ne fa la bind.
2. **Instaurazione della connessione:**
   - **2.1.** Si entra in un loop infinito dove la prima operazione è una "**recvfrom**" bloccante dove si attendono richieste di connessione da parte dei client. Quando arriva una richiesta di connessione si prende il dato nel campo seq_no ricevuto e lo si salva, infine si crea una socket per il figlio associata al processo e se ne fa la bind.
   - **2.2.** Poi, si entra in un nuovo loop infinito dove per prima cosa si controlla il numero di tentativi (meccanismo per evitare connessioni morte), in caso il numero di tentativi sia maggiore di 20 la socket ad esso assegnata viene chiusa e il numero di porta diventa nuovamente disponibile.
   - **2.3.** Si invia un data_packet al client di tipo SYN, che nel flusso di instaurazione della connessione rappresenta il messaggio di conferma (SYNACK), con il seq_no uguale all'id della richiesta ricevuta e contenente nel campo data il numero di porta associato al nuovo socket. Successivamente viene lanciato il timer (prendendo un campione di tempo).
   - **2.4.** Viene poi controllato il timeout verificando che intervallo di tempo trascorso dal lancio del timer superi il valore del timer inserito dall'utente $(((clock() - timer\_sample) * 1000) / CLOCKS\_PER\_SEC > timer)$. In caso ritorni vero (quindi si verifichi il timeout) si incrementa il trial_counter e si rinvia il SYNACK.
   - **2.5.** Infine, vi è il blocco di ricezione dell'ACKSYNACK con una "recvfrom" non bloccante che attende un data_packet dal client e se è di tipo SYN e il seq_no è uguale all'id della richiesta si esce dal while.
   - **2.6.** Quando si esce dal while l'instaurazione della connessione con il client è avvenuta con successo, viene fatta la fork del processo figlio dedicato al client.
3. **Attesa del comando:**
   - **3.1.** Si entra in un loop infinito nel quale viene lanciato un timer tramite la funzione "alarm" che chiude la connessione se il client non invia un pacchetto di comando in un determinato lasso di tempo.
   - **3.2.** Ci si mette in attesa tramite una "recvfrom" bloccante. Una volta ricevuto il comando invia un ACK della corretta ricezione del comando indicato nel campo type.

3.3. Tramite uno switch a seconda del tipo del data_packet (GET, PUT, LIST) si eseguirà l'operazione scelta utilizzando le impostazioni inserite dall'utente al lancio del server.

4. **Esecuzione dell'operazione:**
   - **4.1. GET** si dichiara il data_packet, nel campo type si dichiara che è di tipo GET e nel campo data si inserisce il nome del file da inviare. Infine, viene chiamata la funzione **Upload*.**
   - **4.2. PUT** si dichiara il data_packet, nel campo type si dichiara che è di tipo PUT e nel campo data si inserisce il nome del file da ricevere. Infine, viene chiamata la funzione **Download*.**
   - **4.3. LIST** si dichiara il data_packet e nel campo type si dichiara che è di tipo LIST e viene chiamata la funzione **Upload***.

Client

L'implementazione del client si divide in quattro parti:

1. **Inizializzazione:**
   - **1.1.** si prendono i parametri inseriti dall'utente e tramite questi si crea la socket (UDP) che utilizzerà per comunicare con il server e fa la connect per fissare i componenti del server, infine viene installato il gestore della SIGALARM che segnalerà all'utente quando avverrà il timeout per la scelta dell'operazione.
2. **Richiesta connessione:**
   - **2.1.** Si entra in un loop infinito dove per prima cosa si verifica il numero di tentativi
   - **2.2.** Poi viene inviato il SYN che è un data_packet di tipo SYN in cui il campo seq_no viene generato randomicamente
   - **2.3.** Viene fatto partire il timer
   - **2.4.** Viene poi controllato il timeout verificando che l'intervallo di tempo trascorso dal lancio del timer supera il valore del timer inserito dall'utente $(((clock() - timer\_sample) * 1000) / CLOCKS\_PER\_SEC > timer)$. In caso ritorni vero (quindi si verifichi il timeout) si incrementa il trial_counter e si rinvia il SYN.
   - **2.5.** Poi vi è il blocco di ricezione dell'SYNACK con una "recv" che attende un data_packet dal client e se è di tipo SYN e il seq_no è uguale all'id della richiesta si esce dal while.
   - **2.6.** Ricevuto il SYNACK ne viene estrapolato il contenuto, cioè il nuovo numero di porta che sarà associato al processo server che servirà il client
   - **2.7.** Successivamente viene inviato l'ACKSYNACK di tipo SYN e seq_no pari all'id della richiesta
   - **2.8.** Vengono aggiornate le informazioni del server remoto, cioè viene settata una nuova "struct sockaddr" che è stata chiamata "child_addr" con il numero di porta ricevuto
   - **2.9.** Viene fatta di nuovo la connect per fissare queste informazioni.
   - **2.10.** Si entra in un loop dove viene lanciato il timer (tempo massimo per selezionare un'operazione)
   - **2.11.** Si prende il valore inserito dall'utente e tramite uno switch si eseguirà l'operazione scelta utilizzando le impostazioni inserite dall'utente al lancio del client.

3. **Invio del comando:**
   - **3.1.** Si entra in un loop infinito
   - **3.2.** Si verifica il numero dei tentativi
   - **3.3.** Invia il comando che è un data_packet in cui nel campo type viene inserito il tipo di operazione che si vuole svolgere e si fa partire un timer
   - **3.4.** Viene poi controllato il timeout verificando che l'intervallo di tempo trascorso dal lancio del timer supera il valore del timer inserito dall'utente $(((clock() - timer\_sample) * 1000)/ CLOCKS\_PER\_SEC > timer)$. In caso ritorni vero (quindi si verifichi il timeout) si incrementa il trial_counter e si rinvia il comando.
   - **3.5.** Si mette in attesa di ricevere l'ACK e se è dello stesso tipo del comando inviato si esce dal ciclo
4. **Scambi di dati:**
   - **4.1. GET** si dichiara il data_packet, nel campo type si indica che è di tipo GET e nel campo data si inserisce il nome del file da ricevere. Infine, viene chiamata la funzione **Download*.**
   - **4.2. PUT** si dichiara il data_packet, nel campo type si indica che è di tipo PUT e nel campo data si inserisce il nome del file da inviare. Infine, viene chiamata la funzione **Upload*.**
   - **4.3. LIST** si dichiara il data_packet e nel campo type si indica che è di tipo LIST e viene chiamata la funzione **Download***.

Timer

Vengono implementati due tipi di timer:

- Timer Statico;
- Timer Dinamico.

Timer Statico

In caso di timer statico il suo valore espresso in millisecondi, inserito dall'utente al momento in fase di configurazione, rimarrà invariato per tutta la durata dell'esecuzione.

Timer Dinamico

In caso di un timer dinamico, il cui valore, anch'esso espresso in millisecondi, tende a adattarsi in base allo stato del canale.

Viene inizializzato al valore di default di dieci millisecondi.

Il timer viene ricalcolato in due diversi momenti:

- **Quando avviene un Timeout:** Il Timer viene settato al doppio dell'ultimo valore utilizzato, se il valore è superiore a 10 secondi il timer viene reimpostato al suo valore di default (10 millisecondi), questo viene fatto per "sbloccare" il canale nei casi in cui il timer cresca sensibilmente e limitare l'attesa.
- **Quando, in fase di invio dei file, ricevo un ACK**:

```
sample_RTT = (clock() - start_sample_RTT) * 1000 / CLOCKS_PER_SEC;
estimated_RTT = (double)(0.875 * estimated_RTT) + (0.125 * sample_RTT);
dev_RTT = (double)(0.75 * dev_RTT) + (0.25 * fabs(sample_RTT - estimated_RTT));
timer = (double)estimated_RTT + (4 * dev_RTT);
```

Upload

La funzione Upload viene eseguita quando si vuole inviare un file o la lista dei file presenti sul server.

1. La finestra è implementata tramite un array di data_packet di dimensione window_size chiamato "packet_buffer".
2. Se il tipo di operazione è diverso da LIST si apre in lettura il file che si vuole inviare e si calcola la sua dimensione.
3. Se invece il tipo di operazione è LIST si apre la cartella "/files"(che contiene i file presenti sul server) ed il puntatore ritornato viene salvato in "d", la directory viene manipolata come una lista collegata di strutture "dir", dopo l'apertura viene creata una copia di "d", cioè la testa della lista, in "head" utilizzando la funzione "telldir", successivamente per calcolare il numero di messaggi da inviare si scorrono tutti gli elementi della directory con la funzione "readdir" incrementando un contatore, gli elementi speciali "." e ".." vengono lasciati fuori dal conteggio, finito lo scorrimento ci si riposiziona alla posizione "head" con la funzione "seekdir".
4. Si entra nel loop dove avviene la trasmissione, la cui condizione di uscita è *(((ntohl(ack.seq_no) + 1) * MTU < file_size) || ((ntohl(ack.seq_no) + 1) < num_of_files))*.
   Questa condizione verifica che il numero di sequenza dell'ACK ricevuto moltiplicato per la grandezza dei chunck inviati sia minore della grandezza del file, quando questa quantità supera o eguaglia file_size significa che tutti i messaggi contenenti i chunk di file sono stati riscontrati e che quindi l'invio del file è terminato e si può uscire dal while **(caso GET e PUT)** oppure che il numero di sequenza dell'ACK ricevuto sia minore al numero dei file presenti nella directory, quando questa quantità supera o eguaglia num_of_files significa che tutti i messaggi contenenti le stringhe (nomi dei files) sono stati riscontrati e che quindi l'invio del file è terminato e si può uscire dal while **(caso LIST).**
   - **4.1.** Si controlla il numero dei tentativi.
   - **4.2.** Si verifica la seguente condizione "next_seq_no < base+window_size" se è vera significa che il numero di sequenza del prossimo messaggio da inviare cade nella finestra del Go-Back N e che quindi quest'ultima non è ancora stata saturata ed è possibile inviare messaggi.
   - **4.3.** Se l'operazione da effettuare **non** è LIST si leggono MTU (1024) byte dal file e vengono messi nel campo data di "packet_buffer[next_seq_no%window_size]", l'indice preso in questo modo corrisponde alla corretta posizione del messaggio nella finestra, il valore di ritorno della "read" viene messo nel campo "lenght" del suddetto elemento dell'array, il messaggio viene successivamente indicato come tipo NORMAL, etichettato con numero di sequenza

next_seq_no ed inviato. Se il timer dinamico è attivo viene preso un campione di tempo (per il calcolo del sample RTT). Inoltre, se il messaggio appena inviato corrisponde alla base della finestra del Go-Back N cioè “next_seq_no==base” allora viene lanciato il timer per prendere il TIMESTMP di riferimento iniziale (per il timeout).

4.4. Se l’operazione è LIST si scorre nuovamente la lista della directory con la “readdir” mettendo nel campo “data” dei data_packet la stringa “dir->d_name” cioè il nome del file puntato in quel momento da “dir”, e nel campo “length” la lunghezza della stringa “dir->d_name”. Inoltre, come nell’altro caso si prendono campioni di tempo con le stesse condizioni.

4.5. Dopo la trasmissione vi è il blocco di ritrasmissione che avviene se c’è stato un timeout.

4.5.1. Si incrementa il trial_counter.

4.5.2. In caso si aggiorna il Timer Dinamico.

4.5.3. Si ritrasmette tutto quello che è presente nel packet_buffer.

4.5.4. Infine, viene ripreso un campione per il calcolo del sample RTT.

4.6. Dopo la ritrasmissione vi è il blocco di ricezione, in questa parte viene utilizzata una “recvfrom” non bloccante che attende dei data_packet (ACK), alla ricezione di uno di questi vi è la “simulate_loss” se ritorna falso si pone la base pari al seq_no del data_packet ricevuto +1, si azzera “trial_counter” in quanto ricevere un ACK implica che il client non è morto, successivamente si ricalcola il timer dinamico (se attivo).

5. Quando finisce la trasmissione si entra in un loop infinito che invia il pacchetto di FIN (in modo analogo a come il client effettua la richiesta di connessione) e si esce in caso di ricezione di FIN ACK oppure in caso si superi il numero di tentativi (in caso di perdita del FIN ACK).

Download

La funzione Download viene eseguita quando si vuole ricevere un file o la lista dei file presenti sul server.

1. Si alloca e si assegna un valore ad rm_string (conterrà il comando “rm nomedelfile” che eliminerà il file corrotto se si verifica un errore)
2. Se il tipo dell’operazione (passato dal data_packet) è GET o PUT si apre/crea/tronca il file che si intende scaricare (nome del file passato nel campo data del data_packet)
3. Si entra in un loop infinito
   - **3.1.** Si controlla il numero dei tentativi
   - **3.2.** Si mette in attesa di una “recvfrom” bloccante che attende dei data_packet i quali conterranno i chunk di file provenienti dal client (NORMAL) oppure il FIN, il pacchetto può essere scartato per simulare la perdita tramite la funzione “simulate_loss”, in caso non venga scartato, ci possono essere tre opzioni:
   - **3.3.** Se l’operazione non è LIST e se il data_packet ricevuto sono dati (NORMAL) si scrive su file con la funzione “write” lenght byte del campo data del messaggio ricevuto (se per qualche motivo non vengono scritti tutti, ci si riposiziona indietro nel file di un numero di byte pari a quelli effettivamente scritti e non si riscontra il pacchetto) si genera poi un data_packet di tipo NORMAL e con seq_no pari al seq_no del data_packet ricevuto, infine si incrementa il contatore “expected_sequence_no” il quale come intuibile dal nome indica il prossimo pacchetto che si aspetta.
   - **3.4.** Se l’operazione è LIST e se il data_packet ricevuto sono dati (NORMAL) i dati ricevuti non sono chunk di file ma stringhe contenenti il nome dei file scaricabili da server, quest’ultime vengono solamente stampate a schermo.
   - **3.5.** In caso si riceva un FIN, se il FIN è di errore (si riconosce perché il campo lenght è diverso da 0) si rimuove il file che si era creato utilizzando la rm_string, si estrapola la stringa di errore contenuta nel FIN, lo si stampa a schermo ed infine si genera un data_packet di tipo FIN con seq_no pari al seq_no del data_packet ricevuto e si esce dal while.
   - **3.6.** Dopo la fase di ricezione, viene inviato il data_packet (ACK) generato in precedenza tramite la “sendto” e si continua il while.
4. Se si è usciti dal while significa necessariamente che è stato ricevuto un FIN, non avendolo fatto prima viene inviato con la “sendto” il data_packet di FINACK che era stato solamente generato, viene chiuso il file (se l’operazione non era LIST) e la socket del figlio.

4. Limitazioni

- La prevenzione delle connessioni morte è utile al fine di rendere robusto il server ad eventuali disconnessioni dei client; tuttavia, per come è implementato nel caso di probabilità di perdita alte è possibile che si verifichi un aborto dell'operazione che poteva invece terminare con successo, questo risulta evidente nel momento in cui si scelgono dei timer piccoli.
- Il Timer Dinamico viene aggiornato ogni volta che si riceve un ACK sulla base di un TIMESTMP campionato ogni volta si trasmette/ritrasmette un pacchetto, ciò comporta ad avere un timer molto basso e quindi di sovraccaricare il canale, questo viene "risolto" tramite il contatore del numero dei tentativi, in quanto, se si ha un tasso di perdita molto elevato è meglio chiudere la connessione e nel caso provare in un secondo momento piuttosto che occupare uno slot del server per cercare di inviare/ricevere il file in un tempo molto elevato.

5. Piattaforma utilizzata per sviluppo e testing

Lo sviluppo è avvenuto in ambiente Unix in particolare si è utilizzata una **WSL2** (Windows Subsystem for Linux) su cui è installata la distribuzione "**Ubuntu 20.04.4 LTS**".

La versione del Kernel è "**5.10.60.1-microsoft-standard-WSL2**".

Per la compilazione è stato utilizzato "**gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0**".

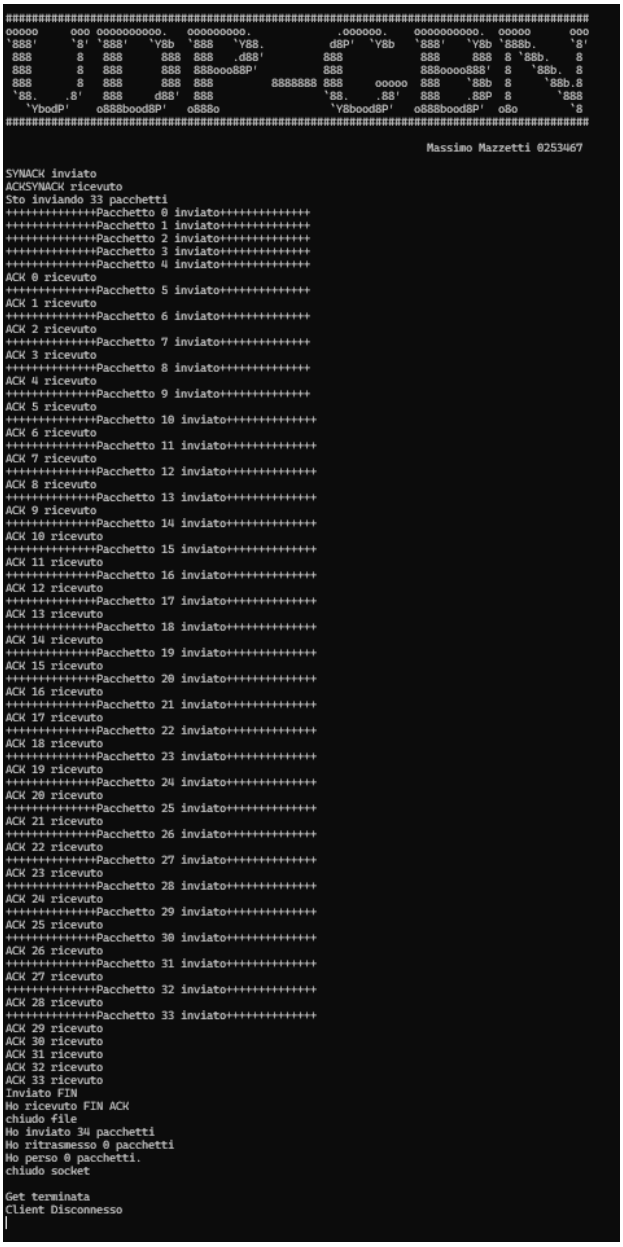
Per la scrittura del codice è stato utilizzato il text editor "**Visual Studio Code**".

Per la generazione del Makefile è stato utilizzato "**CMake version 3.16.3".**

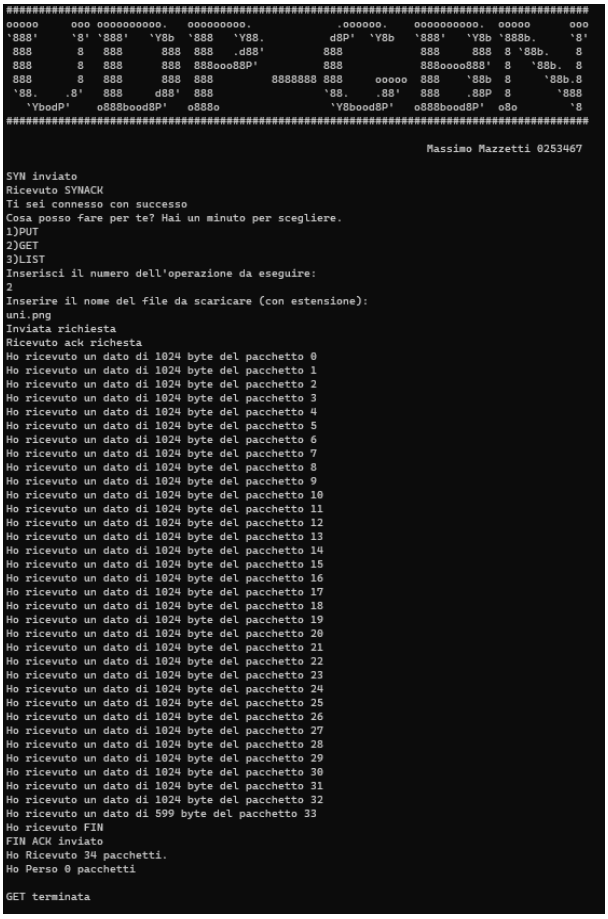
6. Esempi

- **GET**

Server

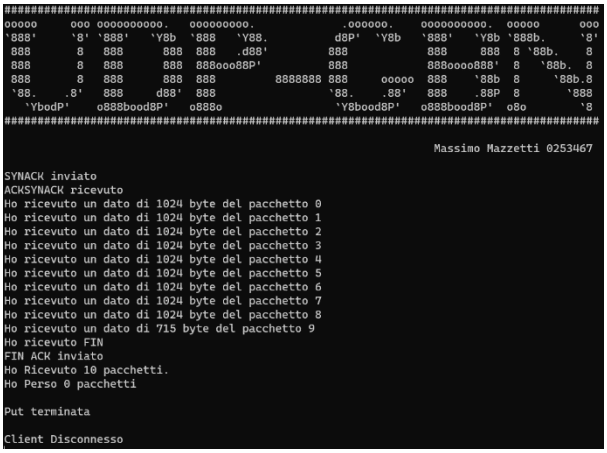


Client



- **PUT**

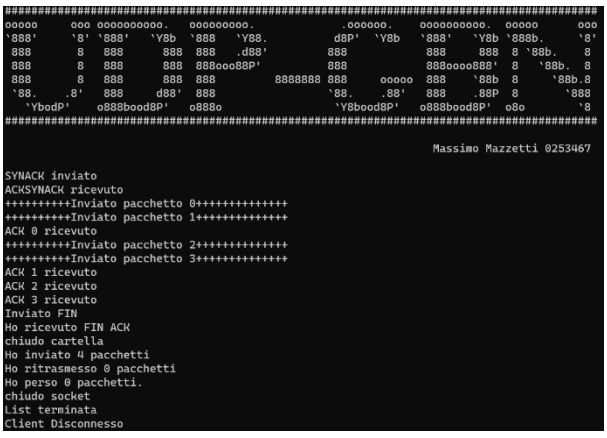
Server



Client

- **LIST**

Server



Client

7. Valutazione Prestazioni

I tempi dei test sono stati calcolati tramite una media aritmetica, in particolare ogni caso di test è stato ripetuto dieci volte.

Il controllo del numero dei tentativi in fase di test è stato disabilitato.

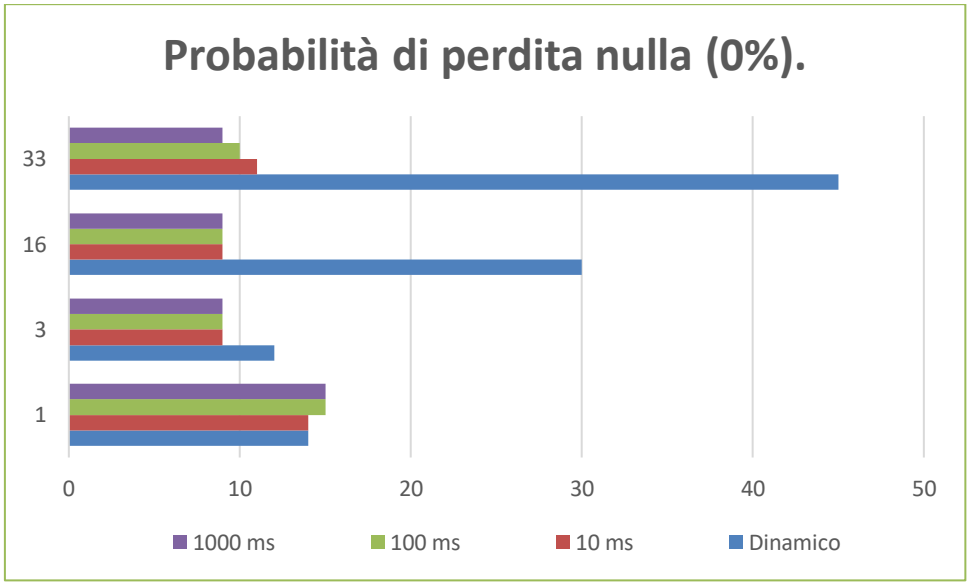
Per i test è stata utilizzata l'immagine "uni.jpg" di 34KB presente nella directory /server/files.

Per i test è stato fissato il tasso di perdita, la dimensione della finestra Go Back N e il valore del timer.

1) Dimensione della finestra:
   a) **1 :** in cui il protocollo lavora nella modalità stop and wait;
   b) **3 :** caso in cui la finestra può contenere dati per circa 1/10 della dimensione del file;
   c) **16 :** caso in cui la finestra può contenere dati per circa 1/2 della dimensione del file;
   d) **33 : c**aso in cui la finestra può contenere dati per l'intera dimensione del file;
2) Valore del Timer:
   a) **Dinamico**
   b) **10 ms**
   c) **100 ms**
   d) **1000 ms**

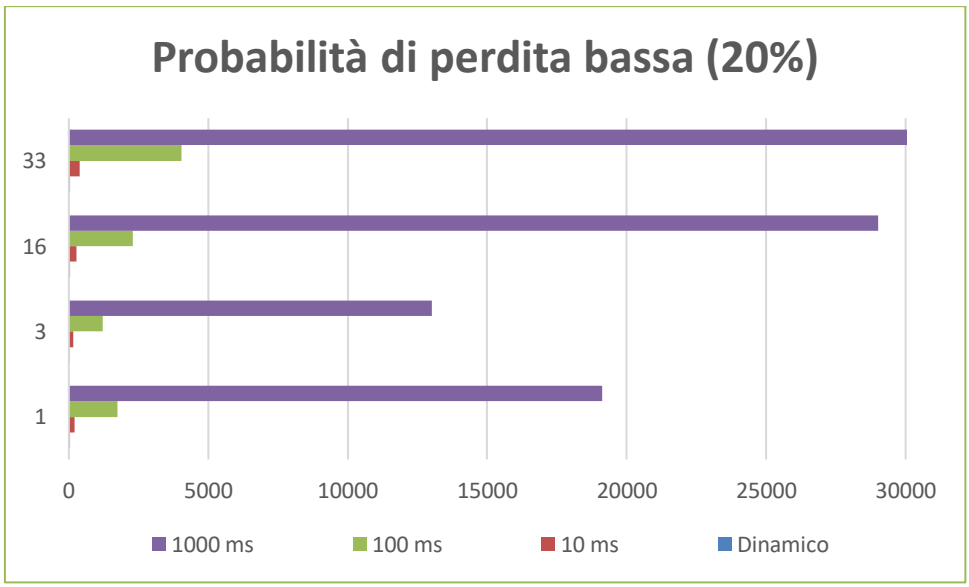
- **Probabilità di perdita nulla (0%).**

Perdita	Finestra	Timer [ms]	Tempo esecuzione [ms]
0	1	Dinamico	14 ms
0	3	Dinamico	12 ms
0	16	Dinamico	30 ms
0	33	Dinamico	45 ms
0	1	10 ms	14 ms
0	3	10 ms	9 ms
0	16	10 ms	9 ms
0	33	10 ms	11 ms
0	1	100 ms	15 ms
0	3	100 ms	9 ms
0	16	100 ms	9 ms
0	33	100 ms	10 ms
0	1	1000 ms	15 ms
0	3	1000 ms	9 ms
0	16	1000 ms	9 ms
0	33	1000 ms	9 ms



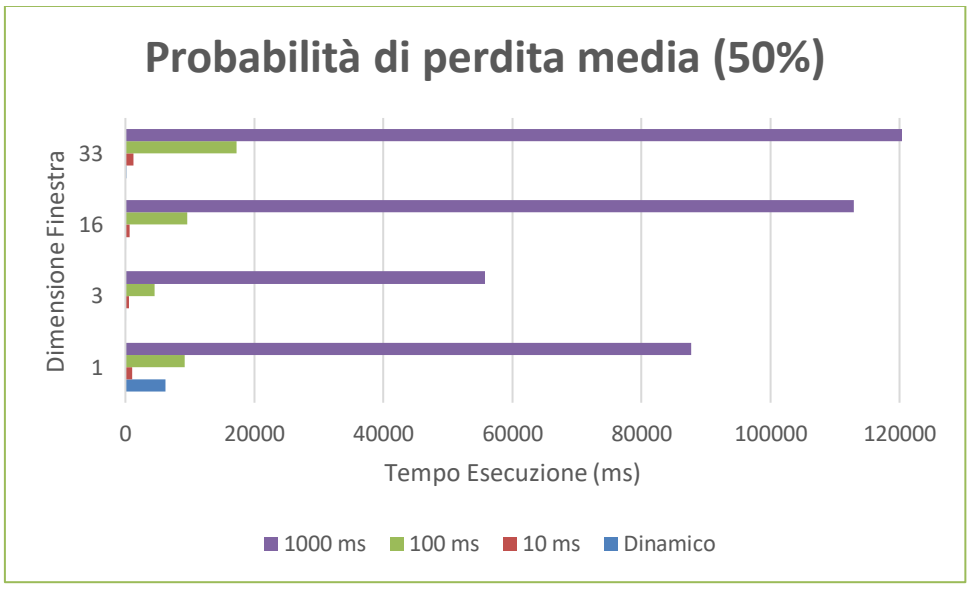
- **Probabilità di perdita bassa (20%)**

Perdita	Finestra	Timer [ms]	Tempo esecuzione [ms]
0.2	1	Dinamico	33 ms
0.2	3	Dinamico	13 ms
0.2	16	Dinamico	35 ms
0.2	33	Dinamico	50 ms
0.2	1	10 ms	200 ms
0.2	3	10 ms	150 ms
0.2	16	10 ms	265 ms
0.2	33	10 ms	390 ms
0.2	1	100 ms	1737 ms
0.2	3	100 ms	1213 ms
0.2	16	100 ms	2302 ms
0.2	33	100 ms	4049 ms
0.2	1	1000 ms	19118 ms
0.2	3	1000 ms	13021 ms
0.2	16	1000 ms	29015 ms
0.2	33	1000 ms	43199 ms



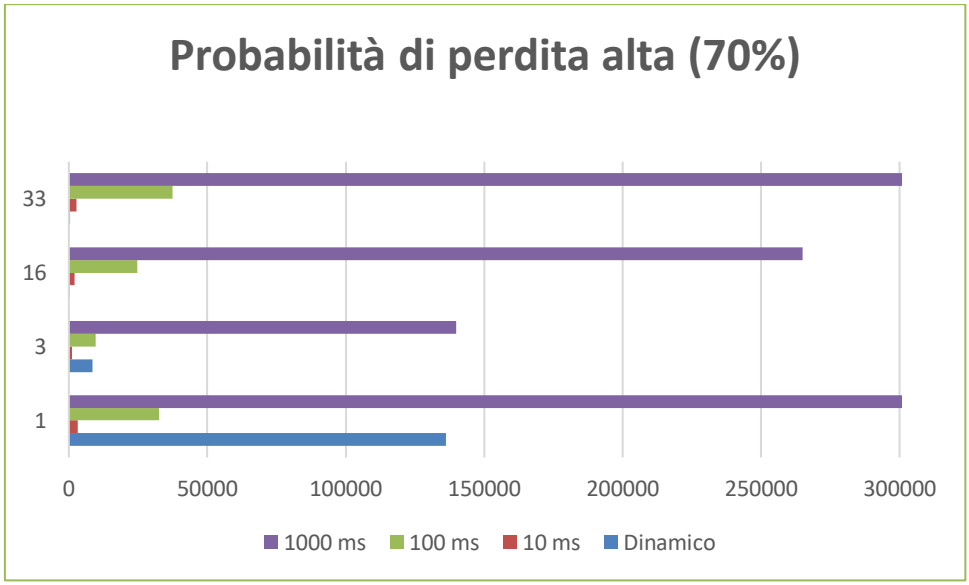
- **Probabilità di perdita media (50%)**

Perdita	Finestra	Timer [ms]	Tempo esecuzione [ms]
0.5	1	Dinamico	6200 ms
0.5	3	Dinamico	62 ms
0.5	16	Dinamico	81 ms
0.5	33	Dinamico	138 ms
0.5	1	10 ms	1034 ms
0.5	3	10 ms	534 ms
0.5	16	10 ms	663 ms
0.5	33	10 ms	1243 ms
0.5	1	100 ms	9220 ms
0.5	3	100 ms	4491 ms
0.5	16	100 ms	9628 ms
0.5	33	100 ms	17223 ms
0.5	1	1000 ms	87686 ms
0.5	3	1000 ms	55699 ms
0.5	16	1000 ms	112902 ms
0.5	33	1000 ms	189791 ms



- **Probabilità di perdita alta (70%)**

Perdita	Finestra	Timer [ms]	Tempo esecuzione [ms]
0.7	1	Dinamico	136123 ms
0.7	3	Dinamico	8445 ms
0.7	16	Dinamico	194 ms
0.7	33	Dinamico	536 ms
0.7	1	10 ms	3332 ms
0.7	3	10 ms	1156 ms
0.7	16	10 ms	1964 ms
0.7	33	10 ms	2846 ms
0.7	1	100 ms	32493 ms
0.7	3	100 ms	9663 ms
0.7	16	100 ms	24683 ms
0.7	33	100 ms	37359 ms
0.7	1	1000 ms	354029 ms
0.7	3	1000 ms	139866 ms
0.7	16	1000 ms	265031 ms
0.7	33	1000 ms	352260 ms



- **Conclusioni**

In condizioni di canale ottimali la soluzione con timer statici risulta essere lievemente più efficiente in quanto avvengono meno ritrasmissioni.

In condizioni di bassa e media probabilità di perdita il timer dinamico risulta essere notevolmente migliore rispetto a quelli statici.

In condizioni di alta probabilità di perdita il timer dinamico risulta essere migliore rispetto a quelli statici se la dimensione della finestra è grande (1/2 o intero file) in quanto, avvenendo più ritrasmissioni, il ricevente avrà più probabilità di ricevere e segnalare la ricezione di un pacchetto, anche se in questo caso molto probabilmente tutti i tentativi verrebbero bloccati dal contatore dei tentativi.

Per quanto riguarda i valori della finestra di Go Back N sono stati riscontrati valori migliori di efficienza nei casi di dimensione pari a 3.

Nel caso di dimensione della finestra 1 (stop and wait) nella maggior parte dei casi il timer statico impostato a 10 ms risulta essere il migliore.

8.Configurazione, Installazione ed Esecuzione

Per effettuare l'istallazione della soluzione è necessario installare un compilatore ISO C compatibile ed il programma CMake versione maggiore o uguale alla 3.16.

Per installare il sistema basta lanciare il file "./build.sh" che creerà nella cartella "/out/build/x64-linux-release/" 2 cartelle:

- Client: contenente l'eseguibile del client e una sottocartella "files" contenente i files presenti nel client.
- Server: contenente l'eseguibile del server e una sottocartella "files" contenente i files presenti nel server.

```
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN$ ls
CMakeLists.txt  CMakePresets.json  Client  Lib  README.md  Relazione.docx  Server  build.sh  documenti
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN$ ./build.sh
-- The C compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN/out/build/x64-linux-release
Scanning dependencies of target client
[ 25%] Building C object Client/CMakeFiles/client.dir/client.c.o
[ 50%] Linking C executable client
[ 50%] Built target client
Scanning dependencies of target server
[ 75%] Building C object Server/CMakeFiles/server.dir/server.c.o
[100%] Linking C executable server
[100%] Built target server
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN$ ls
CMakeLists.txt  CMakePresets.json  Client  Lib  README.md  Relazione.docx  Server  build.sh  documenti  out
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN$
```

Successivamente vanno utilizzate due o più shell (a seconda di quanti sono i client) sulle quali verranno lanciati gli eseguibili generati in fase di compilazione, è importante sottolineare che il server deve essere lanciato prima di qualsiasi client, questo viene fatto nel seguente modo:

- ./server <porta server> <dimensione finestra> <probabilità perdita> <timeout (in ms double, 0 per timer dinamico)>

```
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN$ ls
 CMakeLists.txt      Client   README.md        Server     documenti  '~$lazione.docx'
 CMakePresets.json   Lib      Relazione.docx   build.sh   out
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN$ cd out/build/x64-linux-release/Server/
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN/out/build/x64-linux-release/Server$ ./server 1026 5 0 0
```

- ./client <indirizzo IPv4 server> <porta server> <dimensione finestra> <probabilità perdita> <timeout (in ms double, 0 per timer dinamico)

```
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN$ ls
 CMakeLists.txt   CMakePresets.json   client   git   README.md   Relazione.docx   Server   build.sh   documenti   out   '~$lazione.docx'
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN$ cd out/build/x64-linux-release/Client/
massimo@Mac:/mnt/c/Users/massi/Desktop/Progetto_IIW/UDP-GBN/UDP-GBN/out/build/x64-linux-release/Client$ ./client 127.0.0.1 1026 5 0 0
```