Ingegneria di Internet e Web

Progetto A.A. 2019/2020

Trasferimento file su UDP - Go-Back N

0244242

Matteo Conti

Indice

1. Architettura e scelte progettuali

L'architettura è di tipo Client-Server dove il server è di tipo concorrenziale a processi ed offre tre servizi:

- Get, il client scarica un file dal server.
- Put, il client carica un file sul server.
- List, il server invia al client la lista dei file scaricabili.

Il server ed il client fanno utilizzo di UDP come protocollo di livello 4 e la fruizione del servizio avviene in tre fasi:

- Instaurazione della connessione tramite 3-Way-Handshake.
- Erogazione del servizio da parte del server (Get, Put, List) in modo affidabile tramite protocollo Go-Back N.
- Chiusura della connessione tramite 2-Way-Handshake al termine dell'erogazione del servizio.

Tutte e tre le fasi risultano robuste ad eventuali disconnessioni del client evitando di lasciare risorse del server allocate ma inutilizzate, i messaggi scambiati sono di tre tipi dati, comandi e risposta.

Si è scelto di realizzare un server concorrenziale che fa utilizzo di processi per semplicità di sviluppo e manutenibilità in quanto per gli scopi di questa applicazione 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.

L'instaurazione come visto avviene tramite 3-Way-Handhsake, cioè vengono scambiati tre messaggi:

- SYN, inviato dal client al server per richiedere di connettersi.
- SYNACK, inviato dal server al client, riscontra il SYN e manda al client un nuovo numero di porta che sarà associato al socket del figlio che si occuperà di servire il client.
- ACKSYNACK, inviato dal client al server, riscontra il SYNACK ed è necessario al server per avere conferma che il client è ancora presente ed ha ricevuto le informazioni necessarie per l'instaurazione della connessione, in questo modo 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.

Il protocollo Go-Back N è stato leggermente modificato al fine di permettere la rilevazione di connessioni morte, questo è stato realizzato introducendo un numero massimo di 10 ritrasmissioni consecutive.

La chiusura della connessione è stata realizzata tramite 2-Way-Handshake dato che non è necessario deallocare particolari risorse, in questa fase vengono scambiati due messaggi:

- FIN che indica la fine dell'erogazione del servizio e conseguente chiusura della connessione, nei casi di Get e List esso viene mandato dal server al client, nel caso della Put viene inviato dal client al server.
- FINACK che riscontra il FIN, inviato dal client al server in caso di Get e List e dal server al client in caso di Put.

2. Implementazione

I messaggi scambiati tra server e client come visto nella sezione precedente sono di tre tipi (dati, comandi e risposta) e sono stati implementati tramite due struct, segment_packet e ack_packet, la prima viene utilizzata per i dati ed i comandi mentre la seconda per il riscontro dei segment_packet. Le struct sono definite come segue:

- **segment_packet**
  Essa viene utilizzata per lo scambio di dati, messaggi di comando e messaggi di sincronizzazione, per lo svolgimento di tale compito possiede quattro attributi:
  - type (int), il quale indica il tipo di messaggio che può essere un comando (PUT, LIST, GET), un dato (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 [MAXLINE], dove MAXLINE vale 497), il quale contiene il dato che viene scambiato, nel caso NORMAL contiene un dato relativo al tipo di servizio richiesto dal client (chunk di file oppure un nome di file su server che può essere scaricato dal client) nel caso di PUT e LIST contiene il nome del file che si vuole scaricare/caricare dal/sul server.
- **struct ack_packet**
  Essa viene utilizzata per il riscontro dei segment_packet, per lo svolgimento di tale compito possiede due attributi:
  - type (int), indica il tipo di ack che può essere un riscontro ad un comando (PUT, LIST, GET), ad un dato (NORMAL) oppure ad un messaggio di sincronizzazione (SYN, FIN).
  - seq_no (long), indica il numero di sequenza dell'ack cioè il numero di sequenza del messaggio che l'ack riscontra.

Per simulare la perdita di messaggi è stata utilizzata la funzione "**simulate_loss**" il cui funzionamento è il seguente:

Alla funzione viene passato un float "loss_rate" che rappresenta la probabilità di perdita inserita dall'utente, quello che viene fatto è utilizzare la funzione "drand48" la quale sceglie un numero di tipo double casuale nell'intervallo [0,1] se il numero pescato è minore di loss_rate viene ritornato true (cioè il messaggio viene perso) altrimenti false (il messaggio non viene perso).

Di seguito verranno viste nel dettaglio le implementazioni del server e del client i quali fanno uso della API di Berkley per la comunicazione.

Server

Il codice del server si divide essenzialmente in quattro parti, una di inizializzazione, una di attesa ed instaurazione di connessioni con i client, una di attesa del comando ed una di scambio dati.

Inizializzazione

Nella fase di inizializzazione il server prende i parametri inseriti dall'utente al lancio (#porta del server, dimensione finestra, probabilità di perdita, timer) e li salva in delle opportune variabili, successivamente imposta il seed per la funzione random che sarà usata per la perdita simulata, dopo di questo crea un socket (UDP) di ascolto utilizzando il numero di porta passato dall'utente e ne fa la "bind".

Infine, installa i gestori della SIGALRM e della SIGCHLD i quali essenzialmente stampano una stringa a schermo quando arriva il segnale.

Attesa ed instaurazione della connessione

Nella fase di attesa ed instaurazione della connessione il server si mette in un loop infinito la cui prima istruzione è una "recvfrom" bloccante con la quale si attendono richieste di connessione da parte dei client (segment_packet). Alla ricezione di una richiesta di connessione da parte di un client il server estrae l'id della richiesta dal campo seq_no del segment_packet ricevuto e lo salva, successivamente crea un nuovo socket figlio (UDP, con numero di porta scelto dal sistema operativo tra quelli disponibili) che sarà associato al processo e ne fa la bind, dopo di questo il server entra in un altro loop infinito nel quale vengono eseguiti essenzialmente tre compiti:

- Come prima cosa viene controllato il "trial_counter", in particolare se esso è maggiore del valore MAX_TRIALS_NO pari a 10 allora il client viene considerato come morto ed il socket creato viene chiuso per rendere il numero di porta nuovamente disponibile.
- Successivamente se il flag "syn_ack_sended" è basso viene inviato al client il SYNACK il quale è un segment_packet di tipo SYN, con 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, questo viene fatto prendendo un campione di tempo "timer_sample" tramite la funzione "clock" e alzando un flag "timer_enable" che indica che il timer è attivo, infine viene alzato un flag "syn_ack_sended" che indica che il SYNACK è stato inviato.
- Viene poi controllato se c'è stato un timeout, questo viene fatto controllando se quest'espressione è vera "(double)(clock()-timer_sample)*1000/CLOCKS_PER_SEC > synack_timer) && (timer_enable)" cioè si guarda che il flag del timer sia alzato e che l'intervallo di tempo trascorso dal lancio del timer (convertito in ms) supera il valore del timer (inizialmente pari a DEFAULT_TIMER di 50 ms), in caso l'espressione sia falsa si va avanti, altrimenti si incrementa il trial_counter che indica quanti tentativi di ritrasmissione consecutivi sono avvenuti, si abbassa il flag syn_ack_sended, infine in caso di timer dinamico esso viene raddoppiato in quanto il timeout indica che il timer è probabilmente troppo breve e l'esecuzione del while continua.

- Infine, vi è il blocco di ricezione dell'ACKSYNACK, il quale è composto da una "recvfrom" non bloccante che attende un ack_packet quando ne viene ricevuto uno, se il tipo dell'ack è SYN ed il seq_no è pari all'id della richiesta, viene abbassato il flag syn_ack_sended e si esce dal while, altrimenti il while continua.

Quando si esce dal loop l'instaurazione della connessione con il client è avvenuta con successo perciò viene fatta la fork del processo figlio dedicato al client, se dopo la fork ci si trova nel padre allora ciò che viene fatto è chiudere il riferimento al socket associato al client con il quale si è instaurata la connessione, in modo che alla terminazione del processo figlio il socket venga effettivamente chiuso, mentre se ci si trova nel figlio ci si mette in attesa del comando del client.

Attesa del comando

Nella fase di attesa del comando si entra in un loop infinito nel quale viene lanciato un timer del valore MAX_CHOICE_TIME di 130 secondi tramite la funzione "alarm", successivamente si entra in attesa del comando (segment_packet) tramite una "recvfrom" bloccante, alla ricezione del comando si entra in uno switch nel quale a seconda del tipo del segment_packet (PUT 1, GET 2, LIST 3) si eseguirà la Put, la Get o la List utilizzando le impostazioni inserite dall'utente al lancio del server, nel caso di Put e Get il campo data del segment_packet contiene il nome del file da caricare/scaricare sul/dal server, ogni caso dello switch inoltre ferma il timer e invia un ack della corretta ricezione del comando indicando nel campo type il tipo di comando ricevuto.

Se in fase di attesa del comando il timer scade, viene chiuso il socket del figlio e quest'ultimo viene terminato.

Scambio dati

La fase di scambio dati varia a seconda dell'operazione scelta dal client, di seguito verranno visti i dettagli di tutti i casi:

- **Put**

  In questo caso il server va ad implementare il ricevitore del Go-Back N, come prima cosa vengono allocate ed inizializzate le risorse necessarie, dopodiché vengono create due stringhe "path" ed "rm_string" le quali sono utilizzate rispettivamente per l'apertura del file in scrittura e lettura con la funzione "open" (che in questo caso crea il file o lo tronca se già esistente), e per la rimozione del file corrotto in caso di errore tramite la funzione "system" (e.g system(rm filepath)), la stringa path è necessaria in quanto i file caricati/scaricati sul server si trovano nella cartella "./files" tuttavia dal client si riceve solo il nome del file di interesse, la stringa rm_string è necessaria in quanto la funzione system accetta come parametro solo una stringa che rappresenta un comando Bash e quindi quest'ultima va costruita prima.

  Dopo questa fase iniziale si entra in quella di ricezione dei messaggi, per fare questo si entra in un loop infinito, come prima cosa viene controllato che il trial_counter sia minore di MAX_TRIALS_NO, in caso affermativo il figlio termina, dopo di questo viene una "recvfrom" bloccante che attende dei segment_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 due opzioni, se il segment_packet ricevuto sono dati (NORMAL) quello che viene fatto è scrivere 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 ack_segment di tipo NORMAL e con seq_no pari al seq_no del segment_packet

ricevuto, infine si incrementa il contatore "expected_sequence_number" il quale come intuibile dal nome indica il prossimo pacchetto che il server si aspetta. 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 ack_segment di tipo FIN con seq_no pari al seq_no del segment_packet ricevuto e si esce dal while. Dopo la fase di ricezione, viene inviato l'ack_segment generato in precedenza tramite la "sendto" e si continua il while. Se si è usciti dal while significa necessariamente che è stato ricevuto un FIN, non avendolo fatto prima viene inviato con la "sendto" l'ack_segment di FIN che era stato solamente generato, viene chiuso il file, il socket del figlio e quest'ultimo viene terminato.

- **Get**

  In questo caso il server va ad implementare il trasmettitore del Go-Back N, come prima cosa vengono allocate ed inizializzate le risorse necessarie, dopodiché se al lancio è stato selezionato il timer dinamico esso viene inizializzato con il valore DEFAULT_TIMER di 50ms e successivamente viene creata la stringa "path" con il quale subito dopo verrà aperto il file in lettura con la "open", dopo l'apertura del file ne viene calcolata la dimensione "file_size" tramite la "lseek" posizionandosi alla fine del file (SEEK_END, la lseek il numero di byte a cui ci si è posizionati), dopodiché ci si riposiziona all'inizio dato che la lettura dei file avviene in modo sequenziale, il calcolo della dimensione del file è necessario per capire quando il file è stato inviato completamente. In questa fase iniziale nel caso in cui l'allocazione di risorse o la open falliscano, viene generato un segment_packet di tipo FIN il cui campo data contiene il messaggio di errore da inviare al client e l'esecuzione salta tramite la "goto" alla fase di terminazione nella quale viene inviato il FIN e terminata la Get come sarà specificato a breve.

  Terminata questa fase iniziale si entra in un while in cui avviene la trasmissione, la condizione di uscita è la seguente "((ntohl(ack.seq_no)+1)*497 < file_size)" ntohl è necessario in quanto ack.seq_no è un long che viene dal client il quale prima di inviarlo lo converte in network byte order tramite htonl, il +1 ed il *497 sono necessari per far combaciare il numero di sequenza dell'ACK con il numero di byte 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, una volta usciti vengono pulite le strutture segment_packet ed ack_packet e si entra in un loop infinito dove si gestisce l'invio del FIN in modo analogo a quanto visto per l'invio del SYNACK nella fase di attesa ed instaurazione della connessione, con la differenza che ora il flag di invio del messaggio si chiama "FIN_sended" e che il seq_no del FINACK non viene controllato in quanto superfluo.

  Verrà ora analizzato il corpo del while che gestisce lo scambio del file anch'esso si divide essenzialmente in quattro blocchi:

  - Come prima cosa viene controllato il "trial_counter" nello stesso modo già descritto in precedenza.
  - Successivamente 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 e ed è possibile inviare messaggi (la finestra è implementata tramite un array di segment_packet di dimensione window_size chiamato "packet_buffer"), in tal caso, vengono letti MAXLINE (497) byte dal file tramite la funzione "read" 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 tramite la “sendto”. Subito dopo l’invio se è attivo il timer dinamico viene preso un campione di tempo “timer_sample” con la “clock” e alzato un flag “RTT_sample_enable” per il calcolo dell’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 prendendo un campione di tempo “timer_sample” con la “clock” ed alzando un flag “timer_enable” come già visto in precedenza, dopo questi due controlli, viene incrementato “next_seq_no” di 1.

- Dopo la trasmissione vi è il blocco di ritrasmissione che avviene se c’è stato un timeout, come già visto ciò avviene quando l’espressione “(double)(clock()-timer_sample)*1000/CLOCKS_PER_SEC > timer) && (timer_enable)” è vera, in tal caso, viene incrementato il “trial_counter”, raddoppiato il timer, campionato per il timer in “timer_sample” e successivamente si entra in un for nel quale viene ritrasmesso tutto quello che è presente in “packet_buffer”, dopo di questo, viene preso un campione con la “clock” per l’RTT in “start_sample_RTT” e alzato il flag “RTT_sample_enable”.
- Dopo la ritrasmissione vi è il blocco di ricezione, in questa parte viene utilizzata una “recvfrom” non bloccante che attende degli ack_packet, alla ricezione di uno di questi vi è la “simulate_loss” se ritorna vero allora si stampa a schermo che è stata simulata una perdita e si va avanti, altrimenti si pone la base pari al seq_no dell’ack_packet ricevuto +1, si azzera “trial_counter” in quanto ricevere un ACK implica che il client non è morto, successivamente viene calcolato il “sample_RTT” come “(double)(clock()-start_sample_RTT)*1000/CLOCKS_PER_SEC” cioè l’intervallo passato tra il primo ed il secondo campione portato in ms (double). Calcolato il sample_RTT si hanno gli strumenti necessari per calcolare il nuovo timer come in TCP con le seguenti formule:
  - $Estimated\ RTT = (1-\alpha)OldEstimatedRTT + \alpha SampleRTT\ con\ \alpha = 0.125$
  - $DevRTT = (1-\beta)DevRTT + \beta|SampleRTT - EstimatedRTT|\ con\ \beta = 0.25$
  - $Timeout = EstimatedRTT + 4DevRTT$

  Infine, se la base è pari a next_seq_no viene fermato il timer.

- **List**

In questo caso il server implementa il trasmettitore Go-Back N, la struttura della List è identica a quella della Get, tuttavia cambiano alcune cose in conseguenza al fatto che ora non viene letto un file ma una directory, in particolare, viene aperta la directory tramite la funzione “opendir” 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”. Dopo questa fase iniziale la List continua in modo identico alla Get solamente invece di leggere dei chunk di file si scorre nuovamente la lista della directory come visto poc’anzi con la “readdir”mettendo nel campo “data” dei segment_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 le close sono sostituite con "closedir".

Client

Il codice del client si divide essenzialmente in tre parti, una di inizializzazione, una di instaurazione della connessione con il server ed una di invio del comando e scambio dati.

Inizializzazione

Nella fase di inizializzazione il client prende i parametri inseriti dall'utente al lancio (indirizzo IP del server, #porta del server, dimensione finestra, probabilità di perdita, timer) e li salva in delle opportune variabili, successivamente imposta il seed per la funzione random che sarà usata per la perdita simulata, dopo di questo crea un socket (UDP) che utilizzerà per comunicare con il server e farà la "connect" per fissare le componenti remote del server, infine viene installato il gestore della SIGALRM, il quale avrà il solo scopo di segnalare all'utente tramite una print a schermo quando avviene il timeout nella fase di scelta del servizio.

Richiesta di connessione con il server

Dopo la fase di inizializzazione viene un loop infinito nel quale viene inviato il SYN e atteso il SYNACK, questo avviene in modo analogo a quanto visto per il SYNACK nella fase di attesa ed instaurazione della connessione nel server, con una piccola aggiunta, quando viene inviato il SYN viene anche generato l'id casuale della richiesta tramite la funzione "lrand48", l'id generato viene utilizzato come numero di sequenza del SYN.

Ricevuto il SYNACK ne viene estrapolato il contenuto, cioè il nuovo numero di porta che sarà associato al processo server che servirà il client. Successivamente viene inviato l'ACKSYNACK, un ack_packet con tipo SYN e seq_no pari all'id della richiesta. Inviato l'ACKSYNACK vengono aggiornate le informazioni remote del server, cioè viene settata una nuova "struct sockaddr" che è stata chiamata "child_addr" con il numero di porta ricevuto, dopodiché viene fatta di nuovo la connect per fissare queste informazioni.

Invio del comando e scambio dati

In seguito a quello visto nel paragrafo precedente vi è un loop infinito al cui inizio viene lanciato un timer con la "alarm" pari a MAX_CHOICE_TIME (120 secondi, nel server è di 10 secondi in più alto per dare un margine consistente in caso di latenze gravi), successivamente vi è una print dei comandi selezionabili dall'utente seguita da una "scanf" di un int con la quale l'utente può selezionare il comando desiderato. Infine, vi è uno switch nel quale in caso venga inserito un comando non valido viene stampato un errore a schermo e si ritorna alla "scanf" con una "goto" altrimenti, a seconda del comando selezionato dall'utente, viene fermato il timer ed eseguita la Put, la Get, oppure la List,.

Prima di continuare occorre sottolineare che per come è stato implementato il server è stato necessario mettere lato client l'invio del comando all'interno delle funzioni che implementano Get, Put e List in modo da non perdere la sincronia client-server, un' altra possibile soluzione è quella di lasciare l'invio del comando nel main del client e mettere all'interno di Get, Put e List l'invio di un pacchetto "start" il quale una volta ricevuto dal server indica che il client è

pronto a ricevere o ad inviare dati di un certo file, infatti questo compito per come è stata fatta l'implementazione è implicito nell'invio del comando di Get, Put o List da client a server.

Procediamo ora con l'analisi di Get, Put e List:

- **Get**

  Come prima cosa vengono allocate ed inizializzate le risorse necessarie, successivamente se l'utente ha selezionato timer dinamico al lancio, viene inizializzato il timer con il valore DEFAULT_TIMER (50 ms) e viene richiesto all'utente di inserire il nome del file che desidera scaricare dal server (con estensione) quest'ultimo viene messo direttamente nel campo "data" del segment_packet di comando, si genera poi la "rm_string" per rimuovere il file sporco in caso di errori (come visto nel server) e si apre il file in lettura e scrittura, creandolo se non esiste o troncandolo se esisteva già. Successivamente si entra in un loop infinito con un funzionamento analogo a quanto visto per il SYNACK nella fase di attesa ed instaurazione della connessione, solamente che in questo caso viene inviato un segment_packet di comando con tipo GET, contenente nel campo data il nome del file da scaricare e si attende un ack_packet di tipo GET.

  Dopo la ricezione dell'ack del comando si entra nella fase di ascolto che consiste in un loop infinito identico a quanto visto per la Put lato server.

  Usciti dal loop viene inviato il FINACK (generato nel loop), viene chiuso il file e viene terminata l'esecuzione.

- **Put**

  Come prima cosa vengono allocate ed inizializzate le risorse necessarie, successivamente se l'utente ha selezionato timer dinamico al lancio, viene inizializzato il timer con il valore DEFAULT_TIMER (50 ms) e viene richiesto all'utente di inserire il nome del file che desidera caricare sul server (con estensione) quest'ultimo viene messo direttamente nel campo "data" del segment_packet di comando, dopo di questo viene aperto il file selezionato in lettura e se ne calcola la dimensione tramite il valore di ritorno della funzione "lseek" posizionandosi alla fine del file e poi riposizionandosi di nuovo all'inizio.

  Dopo di questo vi è il loop per l'invio del comando e l'attesa del riscontro come visto per il SYNACK nella fase di attesa ed instaurazione della connessione, il comando è di tipo PUT e contiene nel campo "data" il nome del file da caricare su server.

  Ricevuto il riscontro del comando si entra nella fase di invio del file che consiste in un loop infinito il cui funzionamento è analogo a quanto visto per la Get lato server.

  Terminata la fase di invio del file vi è un loop nel quale viene inviato il FIN e atteso il FINACK, in modo analogo a quanto visto nella fase di attesa ed instaurazione della connessione per il SYNACK, dopodiché viene chiuso il file e l'esecuzione termina.

- **List**

  Come prima cosa vengono allocate ed inizializzate le risorse necessarie, successivamente se l'utente ha selezionato timer dinamico al lancio, viene inizializzato il timer con il valore DEFAULT_TIMER (50 ms), successivamente viene si entra in un loop infinito con un funzionamento analogo a quanto visto per il SYNACK nella fase di attesa ed instaurazione della connessione, solamente che in questo caso viene inviato un segment_packet di comando con tipo LIST e si attende un ack_packet di tipo LIST.

Dopo la ricezione dell'ack del comando si entra nella fase di ascolto che consiste in un loop infinito identico a quanto visto per la Put lato server con la differenza che 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.
Usciti dal loop viene inviato il FINACK (generato nel loop) e viene terminata l'esecuzione.

3. Limitazioni riscontrate

In fase di sviluppo sono state incontrate principalmente tre limitazioni, una riguardante il timer e due riguardanti la prevenzione di connessioni morte.

Timer

Inizialmente il timer (sia statico che dinamico) è stato implementato tramite l'utilizzo del SIGALRM utilizzando funzione "setitimer(int which, const struct itimerval *new_value, const struct itimerval *old_value)" della libreria "sys/time.h", questa funzione è stata scelta in quanto permetteva di poter impostare un timer in µs cosa non permessa dalla funzione "alarm(unsigned int seconds)" della libreria "unistd.h", il meccanismo di funzionamento è il seguente:

É stato dichiarato un flag globale "timeout_event" inizializzato a 0, questo flag viene messo ad 1 solamente all'interno del gestore del SIGALRM. I cicli che implementano il Go-Back N contengono un blocco di codice che controlla il flag globale e capisce se si è verificato un timeout o meno ed in caso affermativo abbassava il flag ed eseguiva la ritrasmissione di quello presente nella finestra del Go-Back N.

Questa soluzione rispetto a quella illustrata nella sezione implementazione risultava molto più efficiente, infatti impiegava a parità di probabilità di perdita, timer, file e dimensione della finestra anche la metà del tempo, tuttavia per un motivo di cui non è stata trovata una soluzione creava spesso degli stalli, in particolare si è notato che gli stalli avvenivano con maggiore frequenza in casi di timer molto piccoli (dell'ordine delle decine di µs). Da notare che la soluzione utilizzata nel progetto finale è utilizzabile solo in presenza di recv/recvfrom non bloccanti.

Prevenzione connessioni morte

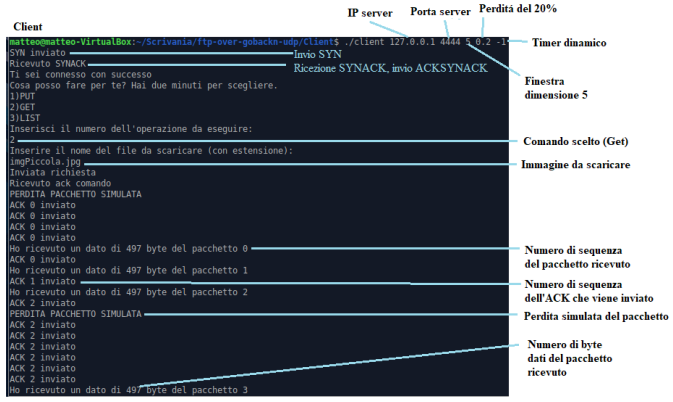
- Per rendere il protocollo più "intelligente" come visto nella sezione implementazione è stato inserito un numero di ritrasmissioni consecutive dopo la quale la connessione viene dichiarata morta, tuttavia questo non viene fatto nei blocchi di codice "che ascoltano" cioè nella Get e List del client e nella Put del server, inizialmente era stata inserita questa feature anche qui, in particolare veniva lanciato un timer all'invio di un ack e veniva fermato alla ricezione del messaggio successivo, alla scadenza del timer veniva incrementato un contatore il cui raggiungimento del valore massimo 10 decretava la connessione come morta causando la terminazione del client o del processo figlio del server. Questa feature è stata rimossa in quanto occasionalmente causava malfunzionamenti nel protocollo Go-Back N e per questioni di tempo non è stato possibile identificarne la causa al fine di trovare una soluzione.
- 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.

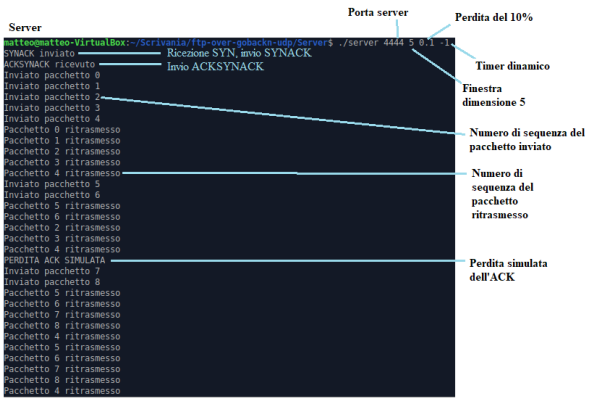
4. Piattaforma utilizzata per sviluppo e testing

Lo sviluppo è avvenuto in ambiente Unix in particolare si è utilizzata una macchina virtuale su cui è installata la distribuzione “Xubuntu 18.04”, le risorse assegnate alla macchina virtuale sono 2GB di RAM, 2 CPU cores ed un disco virtuale di 30 GB. Per la scrittura del codice è stato utilizzato il text editor “Sublime Text”, il codice è stato eseguito sulla shell Bash offerta dal sistema operativo.

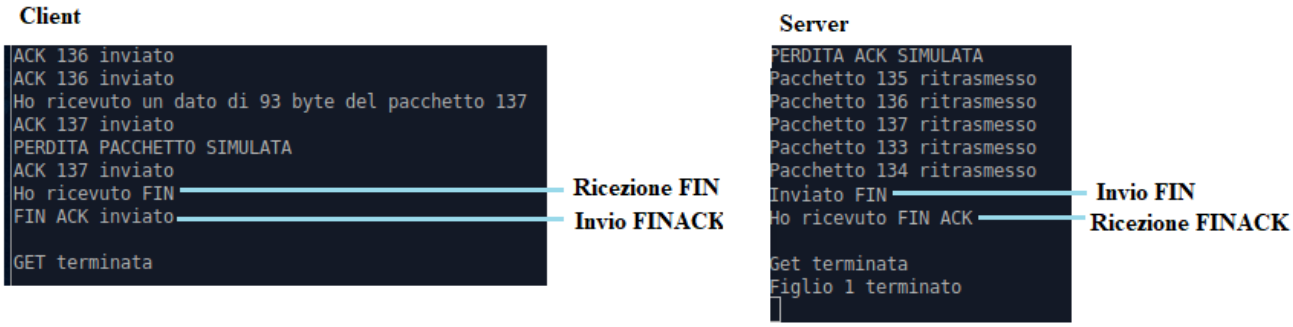
5. Esempi di utilizzo

- **Esempio di Get senza errori**
  - Inizio

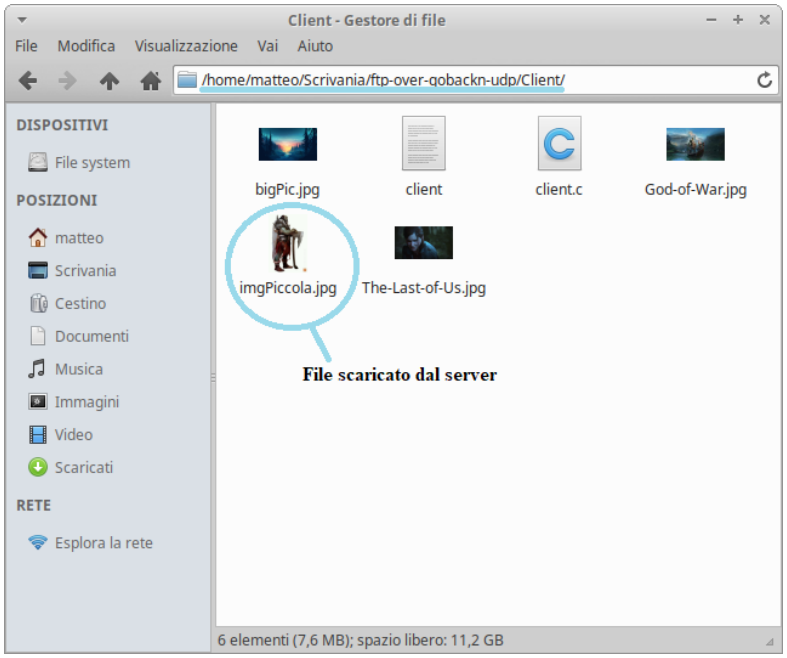




  - Fine

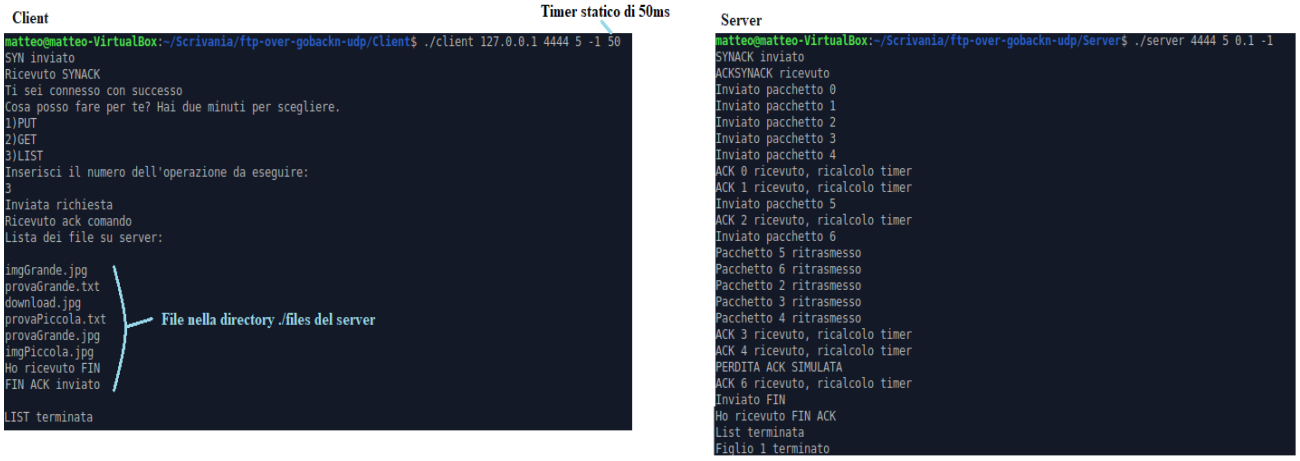


  - Risultato nel client



- **Esempio di Get di un file non presente su server**

- **Esempio di List**



- **Esempio di Put**

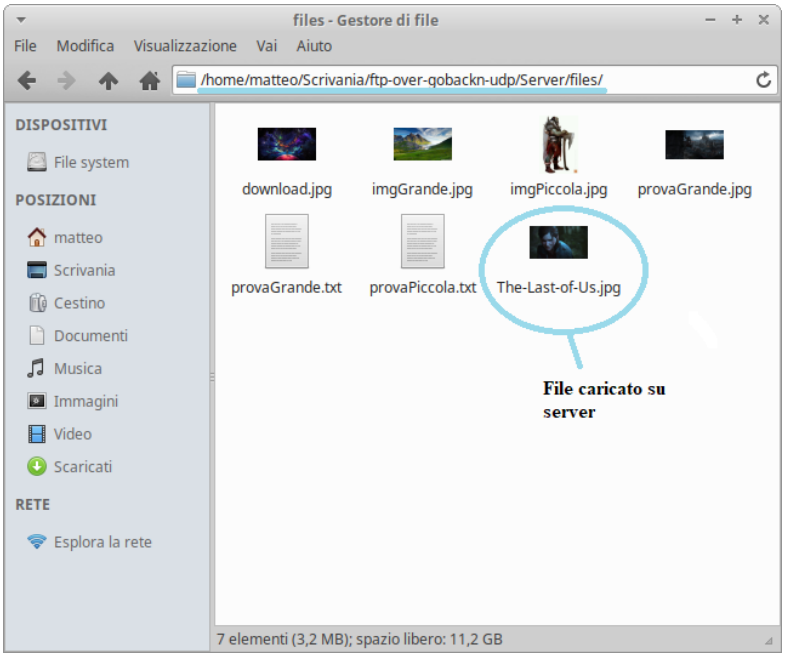
Client

```
matteo@matteo-VirtualBox:~/Scrivania/ftp-over-gobackn-udp/Client$ ./client 127.0.0.1 4444 5 -1 -1
SYN inviato
Ricevuto SYNACK
Ti sei connesso con successo
Cosa posso fare per te? Hai due minuti per scegliere.
1)PUT
2)GET
3)LIST
Inserisci il numero dell'operazione da eseguire:
1
Inserire il nome del file da scaricare (con estensione):
The-Last-of-Us.jpg
Inviata richiesta
Ricevuto ack comando
Inviato pacchetto 0
Inviato pacchetto 1
Inviato pacchetto 2
Inviato pacchetto 3
Inviato pacchetto 4
ACK 0 ricevuto, ricalcolo timer
ACK 1 ricevuto, ricalcolo timer
Inviato pacchetto 5
Pacchetto 5 ritrasmesso
Pacchetto 1 ritrasmesso
Pacchetto 2 ritrasmesso
Pacchetto 3 ritrasmesso
Pacchetto 4 ritrasmesso
ACK 2 ricevuto, ricalcolo timer
Inviato pacchetto 6
ACK 3 ricevuto, ricalcolo timer
Inviato pacchetto 7
ACK 4 ricevuto, ricalcolo timer
```

Server

```
SYNACK inviato
ACKSYNACK ricevuto
Ho ricevuto un dato di 497 byte del pacchetto 0
ACK 0 inviato
Ho ricevuto un dato di 497 byte del pacchetto 1
ACK 1 inviato
Ho ricevuto un dato di 497 byte del pacchetto 2
ACK 2 inviato
Ho ricevuto un dato di 497 byte del pacchetto 3
ACK 3 inviato
Ho ricevuto un dato di 497 byte del pacchetto 4
ACK 4 inviato
ACK 5 inviato
ACK 5 inviato
ACK 5 inviato
ACK 5 inviato
ACK 5 inviato
PERDITA PACCHETTO SIMULATA
ACK 5 inviato
ACK 5 inviato
ACK 5 inviato
Ho ricevuto un dato di 497 byte del pacchetto 5
ACK 5 inviato
Ho ricevuto un dato di 497 byte del pacchetto 6
ACK 6 inviato
Ho ricevuto un dato di 497 byte del pacchetto 7
ACK 7 inviato
Ho ricevuto un dato di 497 byte del pacchetto 8
ACK 8 inviato
ACK 9 inviato
Ho ricevuto un dato di 497 byte del pacchetto 9
ACK 9 inviato
```

  - Risultato Put



- **Esempio scadenza tempo per la scelta del comando**

Client

```
matteo@matteo-VirtualBox:~/Scrivania/ftp-over-gobackn-udp/Client$ ./client 127.0.0.1 4444 5 -1 -1
SYN inviato
Ricevuto SYNACK
Ti sei connesso con successo
Cosa posso fare per te? Hai due minuti per scegliere.
1)PUT
2)GET
3)LIST
Inserisci il numero dell'operazione da eseguire:
Tempo per la scelta terminato
```

Alla scadenza del tempo a disposizione dell'utente per la scelta del comando viene scritto a schermo che il tempo è terminato e l'esecuzione termina

Server

```
matteo@matteo-VirtualBox:~/Scrivania/ftp-over-gobackn-udp/Server$ ./server 4444 5 0.1 -1
SYNACK inviato
ACKSYNACK ricevuto
SIGALRM
errore in recvfrom comando: Interrupted system call
Figlio 1 terminato
```

Alla scadenza del tempo a disposizione dell'utente per la scelta del comando viene generato un SIGALRM, si va nell'handler di quest'ultimo che stampa a schermo che è avvenuto un timeout, l'esecuzione continua nel controllo di errore della recvfrom, viene scritto il tipo di errore e l'esecuzione del processo figlio termina

6. Analisi delle prestazioni

I tempi dei test sono stati calcolati tramite una media aritmetica, in particolare ogni caso di test è stato ripetuto dieci volte eccetto per i casi che hanno superato l'ora di durata in quanto un paio di esecuzioni sono risultate necessarie a definire la qualità delle prestazioni in quel caso.

Per i test è stata utilizzata l'immagine "orco.jpg" di 68KB presente nella directory /server/files.

- **Get**

Probabilità di perdita	Dimensione finestra	Timer	Tempo impiegato
0	1	Dinamico	11.39 ms
0	5	Dinamico	7.45 ms
0	20	Dinamico	13.25 ms
0	100	Dinamico	28.6 ms
0	1000	Dinamico	244.73 ms
0	1	10 ms	7.65 ms
0	5	10 ms	2.17 ms
0	20	10 ms	2.13 ms
0	100	10 ms	2.45 ms
0	1000	10 ms	3.31 ms
0	1	100 ms	8.59 ms
0	5	100 ms	2.45 ms
0	20	100 ms	3.55 ms
0	100	100 ms	2.17 ms
0	1000	100 ms	2.65 ms
0	1	1000 ms	11.39 ms
0	5	1000 ms	2.03 ms
0	20	1000 ms	2.53 ms
0	100	1000 ms	2.83 ms
0	1000	1000 ms	2.34 ms
0.1	1	Dinamico	11.78 ms
0.1	5	Dinamico	7.46 ms
0.1	20	Dinamico	11.27 ms
0.1	100	Dinamico	23.12 ms

0.1	1000	Dinamico	127.32 ms
0.1	1	10 ms	226.32 ms
0.1	5	10 ms	1.89 ms
0.1	20	10 ms	1.97 ms
0.1	100	10 ms	2.23 ms
0.1	1000	10 ms	3.89 ms
0.1	1	100 ms	1318.25 ms
0.1	5	100 ms	2.68 ms
0.1	20	100 ms	2.23 ms
0.1	100	100 ms	2.74 ms
0.1	1000	100 ms	3.44 ms
0.1	1	1000 ms	14016.4 ms
0.1	5	1000 ms	2.12 ms
0.1	20	1000 ms	2.75 ms
0.1	100	1000 ms	3.96 ms
0.1	1000	1000 ms	3.95 ms
0.5	1	Dinamico	224800000 ms
0.5	5	Dinamico	13.5 ms
0.5	20	Dinamico	19.2 ms
0.5	100	Dinamico	17.09 ms
0.5	1000	Dinamico	17.67 ms
0.5	1	10 ms	1446.4 ms
0.5	5	10 ms	18.7 ms
0.5	20	10 ms	9.8 ms
0.5	100	10 ms	10.1 ms
0.5	1000	10 ms	61.3 ms
0.5	1	100 ms	14348.16 ms
0.5	5	100 ms	411 ms
0.5	20	100 ms	17.3 ms
0.5	100	100 ms	14.09 ms
0.5	1000	100 ms	6.85 ms
0.5	1	1000 ms	121064.2 ms

0.5	5	1000 ms	2007.15 ms
0.5	20	1000 ms	8.45 ms
0.5	100	1000 ms	8.19 ms
0.5	1000	1000 ms	8.7 ms
0.8	1	Dinamico	316800000 ms
0.8	5	Dinamico	24662.69 ms
0.8	20	Dinamico	28.3 ms
0.8	100	Dinamico	22.4 ms
0.8	1000	Dinamico	22.3 ms
0.8	1	10 ms	5350 ms
0.8	5	10 ms	260 ms
0.8	20	10 ms	13.11ms
0.8	100	10 ms	23.8 ms
0.8	1000	10 ms	83.82 ms
0.8	1	100 ms	59551.7 ms
0.8	5	100 ms	2707.3 ms
0.8	20	100 ms	23.54 ms
0.8	100	100 ms	22.62 ms
0.8	1000	100 ms	14.7 ms
0.8	1	1000 ms	563159.4 ms
0.8	5	1000 ms	20005 ms
0.8	20	1000 ms	1002.2 ms
0.8	100	1000 ms	23.54 ms
0.8	1000	1000 ms	24.53 ms

Come è visibile dai test in generale le esecuzioni facenti utilizzo del timer dinamico sono sempre più lente di quelle che fanno utilizzo del timer statico in modo abbastanza consistenze ad eccezione dei casi in cui la finestra è di dimensione 1 (stop&wait) in cui questo fatto non è sempre vero, in generale questi numeri non rispecchiano completamente quella che è uno scenario reale in quanto essendo i test eseguiti in locale l'RTT è sicuramente più basso che in uno scenario reale.

Per quanto riguarda la variazione della probabilità di perdita come era lecito aspettarsi all'aumentare di quest'ultima il tempo impiegato aumenta tuttavia non sempre in modo disastroso, molto spesso si rimane nello stesso ordine di grandezza.

Per quanto riguarda la variazione della dimensione della finestra come ci si poteva aspettare se la dimensione è 1 le prestazioni degradano in modo abbastanza evidente (arrivando anche ad ore) tanto che già aumentando la dimensione a 5 si vede un grande miglioramento prestazione il quale intorno ad una finestra di dimensione 20 smette di variare considerevolmente anzi in generale tende a degradare leggermente, tuttavia in modo abbastanza curioso questo discorso non vale se la probabilità di perdita è abbastanza alta (0.5, 0.8) infatti in questi casi le prestazioni vanno a migliorare in modo abbastanza consistente anche con finestre più grandi di 20.

La Put (con lo stesso file) presenta delle prestazioni del tutto analoghe alla Get mentre la List risulta più veloce in quanto dovrà inviare solamente poche stringhe e pertanto paragonarla alla Put e alla Get non ha senso.

7. Manuale di installazione e configurazione

Per poter utilizzare il sistema è necessario dapprima compilare il codice del client “client.c” e del server “server.c” per generare gli eseguibili, questo viene fatto tramite il comando gcc su shell come segue:

- gcc client.c -o client
- gcc server.c -o server

Dopo aver generato gli eseguibili per il corretto funzionamento del sistema è necessario che nella directory dove risiede il file eseguibile del server sia presente una directory di nome “files” la conterrà i file scaricabili dai cliente ed i file caricati dai client.

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 del server> <dimensione finestra> <probabilità di perdita (0.x), -1 per non avere perdita> <timer in ms, -1 per il timer dinamico>
- ./client <indirizzo IP del server> <porta del server> <dimensione finestra> <probabilità di perdita (0.x), -1 per non avere perdita> <timer in ms, -1 per il timer dinamico>

Nota

Al fine di facilitare le dimostrazioni di utilizzo ed il testing è stato commentato il codice che implementa la probabilitá di perdita sulla ricezione di SYN, SYNACK, ACKSYNACK, comandi e ack dei comandi, tuttavia in uno scenario reale anch’essi possono essere persi, per abilitare la probabilità di perdita anche in questi casi è opportuno rimuovere i commenti dalle suddette casistiche, di seguito un esempio nel quale sono evidenziati i commenti da rimuovere (che sono gli stessi per tutte le casistiche):

```
  //Attendo ACK richiesta
  if(recv(sockfd,&ack, sizeof(ack), MSG_DONTWAIT)>0){
    //if(!simulate_loss(loss_rate)){
      if(ntohs(ack.type)==PUT){
        printf("Ricevuto ack comando\n");
        timer_enable=false;
        break;
      }
    //}
    //else
      //printf("PERDITA ACK COMANDO SIMULATA\n");
  }
}
```