|  |  |
| --- | --- |
|  |  |

Ingegneria di Internet e Web

Progetto A.A. 2019/2020

Trasferimento file su UDP reso affidabile con Go-Back N

0244242

Matteo Conti

**Indice**

[1. Architettura e scelte progettuali 2](#_Toc54110076)

[2. Implementazione 3](#_Toc54110077)

[3. Limitazioni riscontrate 6](#_Toc54110078)

[4. Piattaforma utilizzata per sviluppo e testing 7](#_Toc54110079)

[5. Esempi d’uso 8](#_Toc54110080)

[6. Analisi delle prestazioni 9](#_Toc54110081)

[7. Manuale di installazione e configurazione 10](#_Toc54110082)

# Architettura e scelte progettuali

L’architettura é 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, lo scambio dei messaggi avviene tramite due tipi messaggio, uno di comando ed uno di risposta.

Si è scelto di realizzare un server concorrenziale che fa utilizzo di processi per semplicità di sviluppo e manutenibilità ed in quanto per gli scopi del progetto difficilmente si incorrerà nella saturazione delle risorse di sistema, tuttavia, come si vedrà più avanti nel paragrafo riguardante [l’analisi delle prestazioni](#_Analisi_delle_prestazioni), a causa del cambio di contesto abbastanza oneroso rispetto ad altre soluzioni (e.g. thread), questo andrà ad incidere in modo abbastanza evidente sulle prestazioni della Get (lato server) se messa in paragone con la Put (lato client).

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 e serve al server per avere conferma che il client è ancora presente ed ha ricevuto le informazioni necessarie per la fruizione del servizio, 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.

# Implementazione

I messaggi scambiati tra server e client come visto nella sezione precedente sono di due tipi (comando e risposta) e sono stati implementati tramite due struct, segment\_packet e ack\_packet, mentre la seconda per il riscontro dei segment\_packet. Le quali sono definite nel seguente modo:

* **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.

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.

Nella fase di inizializzazione il server prende i parametri (necessari) inseriti dall’utente al lancio e li salva, successivamente 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.

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 attendono richieste di connessione da parte dei client (segment\_packet). Alla ricezione di una richiesta 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 di nuovo il numero di porta disponibile.
* Successivamente 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 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 µs) supera il valore del timer (inizialmente pari a DEFAULT\_TIMER di 50 µs), 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 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.

Nella fase di attesa del comando la connessione è stata instaurata con successo perciò viene fatta la fork del processo figlio, se ci si trova nel padre viene chiuso il riferimento al socket del figlio in modo che al termine del figlio il socket sia chiuso realmente, se ci si trova nel figlio si entra in un loop infinito nel quale viene lanciato un timer del valore MAX\_CHOICE\_TIME di 140 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 case 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.

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 strutture, dopodiché vengono create due stringhe “path” ed “rm\_string” le quali sono utilizzate rispettivamente per l’apertura del file 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”, 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, questa volta il controllo non viene fatto per le ritrasmissioni bensì per la ricezione di pacchetti corrotti oppure per malfunzionamenti persistenti della “recvfrom” nel caso in cui è molto probabile che la scelta più conveniente sia lasciar stare, 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 questa fase di ricezione viene inviato l’ack\_segment con 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 solo generato, viene chiuso il file, il socket del figlio e quest’ultimo viene terminato.

* **Get**
* **List**

## Client

# Limitazioni riscontrate

In fase di sviluppo sono state incontrate principalmente tre limitazioni, una riguardante il timer, una riguardante l’architettura a processi ed una riguardante 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 é 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 é 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 é 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](#_Implementazione) era risultata molto più efficiente, impiegando 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).

## Architettura a processi

Come accennato nella sezione [architettura e scelte progettuali](#_Architettura_e_scelte) è stata scelta per semplicità di sviluppo e manutenzione un architettura a processi, tuttavia questo è stato pagato con le prestazioni, in particolare come accennato il cambio di contesto si manifesta in modo abbastanza evidente (anche ad occhio) sulle prestazioni della Get (server) e Put (client) quando le si mettono a confronto.

In fase di testing non sono stati riscontrati i limiti tipici dell’architettura a processi riguardanti la saturazione delle risorse di sistema e l’overhead dell’operazione di fork, tuttavia in contesti reali dove la richiesta potrebbe essere più alta o con concentrazioni maggiori in determinati istanti di tempo è possibile che questi si manifestino.

## Prevenzione connessioni morte

Per rendere il protocollo più “intelligente” come visto nella sezione [implementazione](#_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.

# Piattaforma utilizzata per sviluppo e testing

Lo sviluppo è avvenuto in ambiente Unix in particolare nella distribuzione “Xubuntu 18.04”, 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.

# Esempi d’uso

# Analisi delle prestazioni

# 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”, questo viene fatto tramite il comando gcc come segue:

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

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>