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

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](#_Toc606296459)

[2. Implementazione](#_Analisi_dei_Requisiti) 3

[3. Limitazioni riscontrate 4](#_Toc2081466291)

[4. Piattaforma utilizzata per sviluppo e testing](#_Toc2147004904) 5

[5. Analisi delle prestazioni](#_Toc518560220) 6

[6. Manuale di installazione e configurazione](#_Toc403811585) 7

# 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.

Si è scelto di realizzare un server concorrenziale che fa utilizzo di processi per semplicitá di sviluppo e manutenibilitá ed in quanto per gli scopi del progetto difficilmente si incorrerá nella saturazione delle risorse di sistema, tuttavia, come si vedrá piú 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 andrá 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, cioé 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 sará associato al socket del figlio che si occuperá 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 ció non è vero il server puó 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 sono stati implementati tramite due struct, segment\_packet e ack\_packet, la prima viene usata per lo scambio di dati, messaggi di comando e messaggi di sincronizzazione, mentre la seconda per il riscontro dei segment\_packet. Sono definite nel seguente modo:

* **struct segment\_packet** {

int type;

/\*Indica il tipo di messaggio che puó essere un comando (PUT, LIST, GET), un dato (NORMAL) oppure un messaggio di sincronizzazione (SYN, FIN)\*/

long seq\_no;

/\*Indica il numero di sequenza del pacchetto, in caso di SYN indica un codice identificativo della richiesta\*/

int length;

/\*Indica la lunghezza del dato nel campo data\*/

char data[MAXLINE];

/\*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 puó 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** {

int type;

/\* Indica il tipo di ack che puó essere un riscontro ad un comando (PUT, LIST, GET), ad un dato (NORMAL) oppure ad un messaggio di sincronizzazione (SYN, FIN)\*/

long seq\_no;

/\*Indica il numero di sequenza dell’ack cioè il numero di sequenza del messaggio che l’ack riscontra\*/

};

## Server

## 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) é 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 piú efficiente, impiegando a paritá di probabilitá di perdita, timer, file e dimensione della finestra anche la metá 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 semplicitá 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 piú alta o con concentrazioni maggiori in determinati istanti di tempo è possibile che questi si manifestino.

## Prevenzione connessioni morte

Per rendere il protocollo piú “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.

# 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 piú 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> <probabilitá 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> <probabilitá di perdita (0.x), -1 per non avere perdita> <timer in ms, -1 per il timer dinamico>