Progetto Sistemi Operativi 2017/2018.

Realizzazione di un servizio ”talk” via internet gestito tramite server

Marco Lanciotti
Linda Ludovisi ?

Rome, January 21, 2019.

Contents

1 Introduzione

L'obiettivo del progetto è stato quello di realizzare un servizio di chat gestito tramite un server. Nel far ciò si è preferito utilizzare come protocollo di trasporto il TCP affinchè esso potesse garantire affidabilità e sicurezza nell'invio del flusso di dati. A livello imprementativo,inoltre, si è deciso di realizzare un server multithread in cui ogni client, richiedente la connessione, fosse affidato ad un thread privato che ne gestisse le richieste. Nel realizzare il progetto è stato molto utile ricorrere all'API select , per non parlare poi del principio di lista collegata utilizzata assiduamente dal server per la gestione di client connessi. Inoltre ci si è concentrati anche sulla gestione di alcuni segnali in cui sia il server sia un client generico sarebbe potuto incorrere. Terminata questa piccola premessa entriamo nell'implementazione vera e propria del progetto.

2 ARCHITETTURA DEL SISTEMA

2.1 Server

Innanzitutto come possiamo notare abbiamo un file "server.c", il quale risponde in modo concorrente a tutte le richieste provenienti dai vari client. Nella cartella inerente al Server sono presenti numerosi header-file utili per suddividere il codice e renderlo il piu leggibile possibile. Elenchiamo qui gli header presenti , contenenti operazioni utilizzate per i seguenti scopi:

- **serverSocketOperation.h**: gestire tutte le funzioni utilizzate per la connessione;
- **ClienList.h**: gestire una lista collegata che tiene traccia di tutti gli utenti connessi alla chat;
- **socketConnectList.h**: gestire tutti gli utenti impegnati in una conversazione privata;
- **Client.h**: gestire una struttura che contiene il messaggio da inviare e i relativi client interessati;
- **string.h**: gestire le funzioni inerenti alle stringhe;

2.2 Client

Riguardo l'architettura client, è presente un file principale, chiamato Client.c. Anche qui abbiamo diversi header file:

- **clientsocketoperation.h**: gestire tutte le funzioni utilizzate per la connessione;
- **Client.h**: gestire una struttura che contiene il messaggio da inviare e i relativi client interessati;
- **string.h**: gestire le funzioni inerenti alle stringhe;

3 SPECIFICHE IMPLEMENTATIVE

3.1 Instaurazione della connessione

Una connessione tra un client ed un server si instaurerà nel momento in cui, il server dopo essersi messo in ascolto su una determinata porta , riceva una richiesta da parte di un client, caratterizzato anch'esso da una porta e un'indirizzo ip predefinito. Ovviamente affinché si possa stabilire data connessione si deve rispettare il seguente vincolo: la porta di ascolto lato server e di connessione lato client deve essere la stessa. Dunque ogni qualvolta un client si voglia collegare con il nostro server, si accoppierà ad esso tramite una porta predefinita, o di default. Il server prenderà in carico le richieste dei vari client intenzionati dunque a connettersi e instaurerà sequenzialmente per ciascuno di essi un thread con annessa struttura privata che ne gestisca le richieste.

3.1.1 Specifiche connessione

In fin dei conti, dal punto di vista implementativo, sia il server così come il client avranno bisogno di istanziare un canale di comunicazione tramite la funzione socket dove verranno impostati i parametri:

- Dominio del socket (famiglia di protocolli) $\rightarrow$ AF _ INET: ossia InterNET Protocol Family (comunicazione tra processi in Internet);
- Tipo di comunicazione $\rightarrow$ SOCK_ STREAM: orientato alla connessione (flusso continuo);
- Protocollo specifico: solitamente, si pone a 0 per selezionare ilprotocollo di default indotto dalla coppia domain e type $\rightarrow$ AF_INET + SOCK_STREAM determinano una trasmissione TCP(protocol==IPPROTO_TCP).

Successivamente entrambi (server e client)dovranno settare i campi di una struttura struct sockaddr_in in cui verranno memorizzati(nel formato corretto $\rightarrow$ in formato della rete) indirizzo ip e porta per instaurare la connessione. Ora il Server ed il Client prenderanno due strade differenti.
1)Il Server effettuerà in ordine:

- **int bind(int sockfd, const struct sockaddr *addr,socklen_t len)**: necessaria per associare la socket all'indirizzo ip prestabilito;
- **int listen(int sockfd, int backlog)**: Utile per mettere in ascolto il socket di eventuali connessioni; fornisce inoltre il max numero di connessioni accettabili(backlog);
- **int accept(int sockfd, struct sockaddr *addr,socklen_tt *addrlen)**: invoca l'accettazione di una connesione su un socket.

Ora Il Server si bloccherà in attesa sull'accept fino a quando non riceverà una richiesta da parte di un client.

2) Il client effettuata la socket() si presterà ad eseguire la:

- **int connect(int dsn_socks, struct sockaddr *addr, int addrlen)**: Permette al client TCP di aprire la connessione con un server TCP (invio segmento SYN)

Efftuata la connect in client instaurerà la connessione con il server e avrà la possibilità di scambiare/ricevere messaggi con quest'ultimo tramite sendto/rcvfrom.

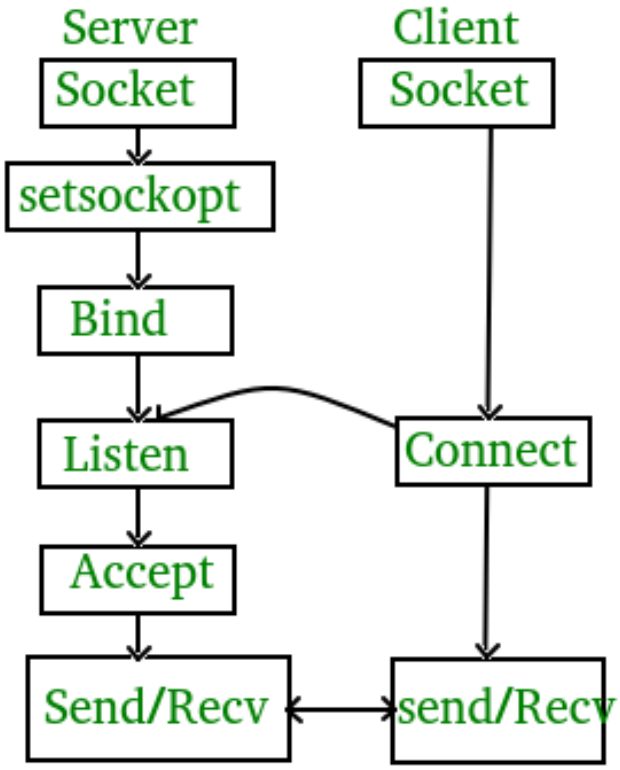

Figure 3.1: TCP Connection

3.2 Gestione Concorrenza

Come detto in precedenza, Il server ha la competenza di offrire un servizio a più client simultaneamente. Per raggiungere tale scopo si è deciso di strutturare il server tramite il principio del multithreading. La nostra scelta è virata sul multithreading piuttosto che sul multiprocesso poiché, come sappiamo, nei server di rete, il cui carico lavorativo è molto elevato ,esiste un limite di multiprogrammazione. Il server superato tale limite rifiuta richieste da parte dei client per evitare uno scenario in cui il server non riesce a servire nemmeno le richieste attive. L'utilizzo dei thread,inoltre, è molto meno oneroso a livello di utilizzo di risorse rispetto ad un approccio multiprocesso. Infatti un applicativo

multithread presuppone la creazione di un unico processo con la conseguente partizione dello stack tra i vari thread. In fin dei conti ogni thread, come un processo, avrà il suo stack privato allocato tuttavia sullo stack fisico del main_process. Da questo punto di vista, l'approccio multiprocesso presuppone l'allocazione di maggior memoria e un maggior dispendio di risorse sulla macchina lavorante. Dal punto di vista di performance effettive è noto che utlizzare dei thread ci permette:

- un minor tempo di context switch;
- maggior efficienza del caching;
- minor tempo nella creazione e nella terminazione di un thread;
- miglior efficienza nella comunicazione e nella sincronizzazione dato che la comunicazione interprocesso richiede l'intervento del kernel, per motivi di protezione, con il conseguente impatto sulle prestazioni.

N.B: In genere risulta più conveniente implementare server concorrenti tramite multithreading, con creazione dinamica di un nuovo thread che si fa carico della gestione di una nuova richiesta di servizio, rispetto alla soluzione in cui ogni nuova richiesta determini l'attivazione di un nuovo processo.

OSS-CLIENT: Per instaurare la conversazione si è deciso di creare anche lato Client un thread che si occuppasse esclusivamente della conversazione. Al termine della conversazione il thread viene killato e il flusso di controllo ritorna al thread_main. Dunque ogni qualvolta si voglia instaurare una conversazione si creerà un thread apposito che si occuperà di gestire la conversazione.

3.2.1 Specifiche Concorrenza

Dal punto di vista implementativo abbiamo deciso di creare un thread-server per ogni client effettuante connessione. Oltretutto abbiamo preferito associare ad ogni thread una struttura privata, al fine di poter gestire le risorse in maniera ottimale e sicura. Infatti in questo modo ogni client avrà una sua struttura dedicata sul server con socket privata e porta privata. Tuttavia è stato necessario utilizzare un mutex che controlasse le richieste di connessione. Infatti qualora due client tentassero contemporaneamente di instaurare una connessione con il server verranno gestiti dal server un per volta sequenzialmente secondo una strategia FIFO. Il server servirà la prima richiesta ricevuta e mettendo in coda le altre. Servito il primo client e instaurata la connessione, il server potrà occuparsi della seconda richiesta arrivata e cosi via.
Per quando riguarda l'implementazione è stato fondamentale l'ultilizzo di un mutex anche per la gestione delle conversazioni tra i vari client.

3.3 Scelte Implementative

3.3.1 Uso della funzione Select

Nella realizzazione del progetto è stato fondamentale l'utilizzo della funzione **int select (int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)**; questa permette di esaminare più canali di I/O contemporaneamente e realizzarare quindi il multiplexing dell'I/O. Si blocca finché:

1)Non avviene un'attività (lettura o scrittura) su un descrittore appartenente ad un dato insieme di descrittori;
2)Non viene generata un'eccezione;
3)Non scade un timeout.
La select è stata utile nella gestione del client il quale tramite la select ha avuto la possibilità di gestire più input simultaneamente. Nel nostro caso, ad esempio, il client aveva la possibilità di gestire due flussi di input:

- Lo Standard input;
- Il canale della Socket;

Utilizzando la select() il primo dei due canali su cui avviene un attività("pronto") viene sbloccato e servito, bloccando di conseguenza l'altro che si metterà in attesa.

```
while(1)
{
    FD_ZERO(&file_descriptors);
    FD_SET(STDIN_FILENO, &file_descriptors);
    FD_SET(connection.socket, &file_descriptors);
    fflush(stdin);
    errno = 0;
    alarm(5000);
    if(select(connection.socket+1, &file_descriptors, NULL, NULL, NULL) < 0 && errno != EINTR)
    {
        perror("Select failed.");
        exit(1);
    }

    if(FD_ISSET(STDIN_FILENO, &file_descriptors))
    {
        handle_user_input(&connection);
        alarm(0);
    }

    if(FD_ISSET(connection.socket, &file_descriptors))
    {
        handle_server_message(&connection);
        alarm(0);
    }
```

Figure 3.2: Funcion Select

3.3.2 USO LISTE COLLEGATE -Lista collegata utenti connessi-

Nella realizzazione del server si è deciso di creare una lista collegeta che tenesse traccia di tutti gli utenti connessi alla chat. Ogni qualvolta un client si connette al server,inserendo un nome univoco, quest'ultimo istanzierà per esso un nodo e lo inserirà in coda in questa lista collegata. Ovviamente ogni utente, nel momento in cui verrà inserito, avrà il flag online = 1 che ne testimonierà la disponibilità. Cosi facendo il server potrà tenere traccia di tutti gli utenti online, di tutti gli utenti effettivamente inseriti, potrà cercare la socket di uno specifico utente qualora ne abbia bisogno, potrà impostare offline un determinato utente. Infine il Server qualora un utente decida di disconnettersi tramite segnale (SIGINT —— SIGALRM) oppure comando /quit avrà la possiblità di rimuovere esattamente quel nodo dalla lista e killare il thread i-esimo adibito alla connessione con esso.

3.3.3 USO LISTE COLLEGATE-Lista collegata Utenti in conversazione-

In questa sezione si vuole porre l'attenzione sull'utilizzo di una lista collegata che permettesse al Server di tener traccia,invece, di tutti gli utenti non disponibili. Un utente diventa non disponibile nel momento in cui instaura una connessione con un altro utente. Il server nel momento in cui due client instaurano una connessione inserirà i loro nodi all'interno di questa lista. Il server ogni volta che riceverà un messaggio verficherà se l'utente è in una conversazione e se lo è, tramite una funzione apposita, risalirà all'utente con cui è collegato nella lista e gli consegnerà il messaggio. Nel momento in cui termina la conversazione il server eliminer à i nodi dalla lista, reimpostandone il flag online = 1, cosiì da indicare la nuova disponibilità e dunque la possibilità per questi di instaurare una nuova conversazione.

3.3.4 Disconnessione & Gestione Segnale

Un Client per disconnettersi dal server associato, ossia dal thread i-esimo chi si occupa della sua gestione, può optare per due strade differrenti: 1) La prima casistica consiste nell'immettere da tastiera il comando /quit presente nella schermata dei comandi disponibili per l'applicativo 2) La seconda scelta si riversa sull'invio e di conseguenza gestione di uno dei seguenti segnali: SIGINT,SIGALRM,SIGQUIT. Tuttavia la seconda strada percorribile può essere testata lanciando l'applicativo da terminale.

OSSERVAZIONE:

- SIGINT(ctrl+C)
- SIGQUIT(ctrl+\)

Nel momento in cui un Client decida di disconnettersi avrà la possibilità di farlo inviando un segnale SIGINT o SIGQUIT al Server. Il Client catturerà uno di questi segnali e si occuperà della loro gestione richiamando un handler. Tale handler noificherà la disconnessione del client a schermo e in seguito disconnetterà il relativo Client. Fatto ciò il Server prenderà carico della disconnessione rimuovendo dapprima il nono dalla lista collegata tenente traccia degli utenti connessi e in seguito killando il thread-iesimo relativo a tale thread. Il SIGALRM, allo stesso modo, qualora il client risultasse inattivo per un tempo prestabilito hard-coded nel codice, disconnetterà il client allo stesso modo degli altri due segnali sopra descritti. Inoltre si è aggiunto anche il controllo e la gestione di un segnale SIGPIPE.

4 Funzionamento Applicazione

4.1 Elenco Comandi

Un Client una volta instaurata una connessione conm il Server tramite l'utilizzo di un nome univoco di riconoscimento avrà la possibilità tramite il comando **/help** di vedere tutti i comandi disponibili.

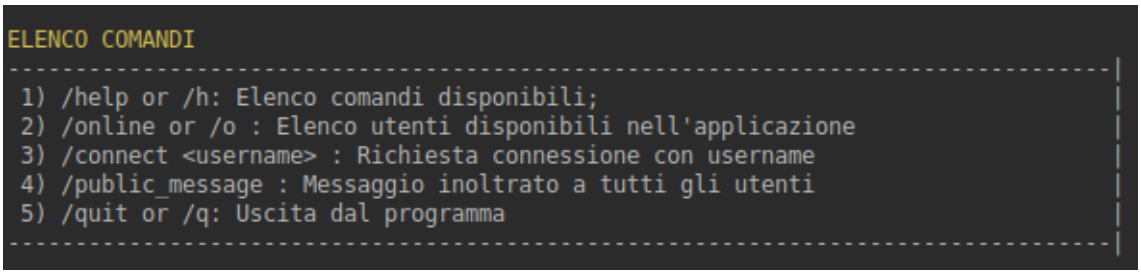

Figure 4.1: Comandi Disponibili

Tramite il comando **/online** un utente potrà visionare tutti gli utenti online & disponibili

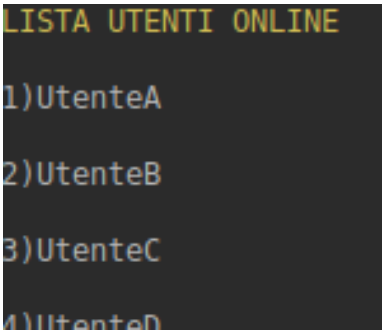

Figure 4.2: Utenti online

Tramite commando **/connect** <**username**> l'utente avrà la possibilità di collegarsi con un dato utente.

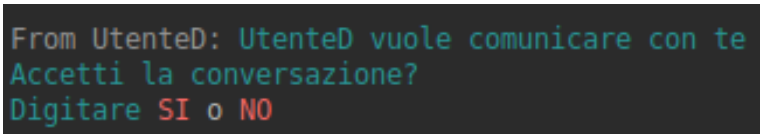

Figure 4.3: RIchiesta /connect <username>

Quest'ultimo avrà la facolta di accettare la connessione(SI) e iniziare la conversazione oppure rifiutare (NO)

Figure 4.4: Inizio Conversazione

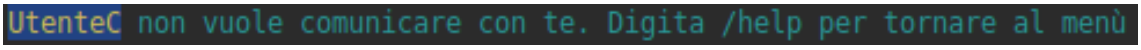

Figure 4.5: No Conversazione

Se utente vuole interrompere la conversazione con un altro utente per instaurare una nuova conversazione immetterà il messaggio exit

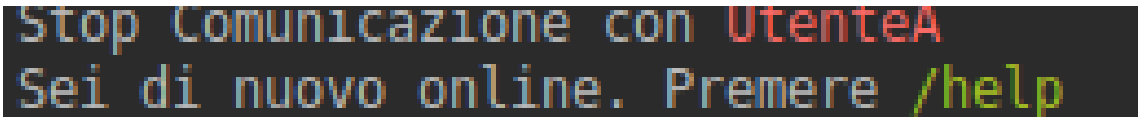

Figure 4.6: Termine Conversazione

Se un utente vuole entrare in chat ma la chat room è piena il server lo disconnetterà immediatamente

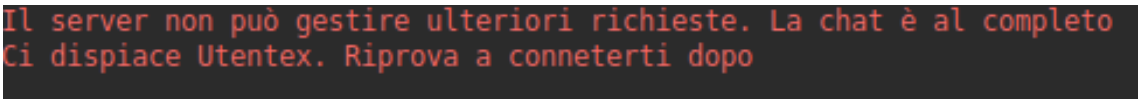

Figure 4.7: Chat al completo

Se un utente si disconnette tramite segnale, comando /quit o perchè risulti inattivo

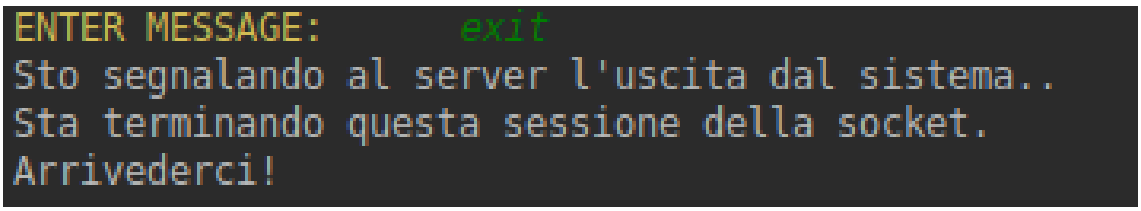

Figure 4.8: Disconnessione Client

5 Manuale D'uso

Il capitolo finale di questa relazione lo dedichiamo ad una semplice guida all'istallazione del nostro sistema. Seguendo i passi sottoelencati è possibile eseguire il codice sul proprio dispositivo:

- prelevare il file compresso "SO" e spostarlo all'interno del proprio file system;
- decomprimere la cartella;
- aprire la cartella "progetto_so_Final";
- aprire due terminali separati e posizionarsi nella giusta cartella
- Con il primo terminale compilare il file "Server.c" all'interno della cartella "Server";
- Con il secondo terminale compilare il file "Client.c" all'interno della cartella "Client"
- eseguire infine entrambi