Server IRC

Laboratorio di ingegneria informatica a.a. 2016/2017

Edoardo Vanin mat. 1094068

Università degli studi di Padova

Contents

1 Introduzione

Il progetto consiste nella realizzazione di un server basato sul protocollo IRC. Il server deve sempre rimanere in ascolto ed accettare le connessioni da parte dei nuovi utenti. Per poter gestire al meglio le connessione di nuovi utenti, viene utilizzata la programmazione multithreading. Il programma è scritto in C, essendo un linguaggio di programmazione a basso livello permette una migliore elaborazione delle richieste grazie alla sua velocità di esecuzione

1.1 Obbiettivi

L'obbiettivo del progetto è implementare un server in grado di far comunicare gli untenti all'interno delle stanza IRC per fare ciò c'è bisogno di implementare un numero minimo di comandi tra tutti i possibili definiti dal protocollo per la corretta comunicazione server-client

I seguenti comandi sono stati implementati:

- NICK comando per modificare il nomeutente
- USER comando per la registrazione dell'utente
- JOIN comando per entrare in una stanza
- MODE comando per visualizzare o modificare le caratteristiche di un canale
- WHO comando per avere informazioni su utenti/canali
- WHOIS comando per avere informazioni su utenti/canali
- PING comando per verificare la connessione
- PRIVMSG comando per inviare messaggi
- PART comando per abbandonare una stanza
- QUIT comando per disconnettersi dal server
- LIST comando per visualizzare tutte le stanza presenti nel server

1.2 Motivazioni

Da sempre sono stato affascinato dalla comunicazione client-server così ho colto l'occasione al balzo per approfondire le mie conoscenze in questo ambito. Già dalle superiori sapevo che la comunicazione di rete tra programmi avviene mediante socket ma non ho mai avuto modo di poterla provare con mano. Un altro aspetto interessante è la programmazioe concorrente, anche di questa ne sapevo dell'esistenza ma non ho mai avuto modo di provarla con mano, qui la programmazione multithreading è assolutamtnente necessaria per gestire in modo indipendente i comandi provenienti dai vari utenti.

2 Architettura e progettazione del software

Il progetto consiste in un programma che gestisce i comandi che gli arrivano dal più client.

Per farlo nel modo più veloce e corretto possibile si è seguito l'approccio *divide et impera*, il problema è stato diviso in due sottoproblemi, l'accettazione della connessione del client e la gestione dei comandi utente

L'accettazione è gestita all'interno del main mentre per la gestione dei comandi utente si è ricorso alla programmazione multithreading per poter gestire contemporaneamente comandi provenienti da utenti diversi

2.1 Accettazione della connessione

Una volta avviato il programma il software rimane in ascolto di una connessione alla porta di default del protocollo IRC ovvero la porta 6667.

Una volta stabilita la connessione con il client, mediante protocollo TCP, il programma associa al nuovo utente un thread che gestirà i comandi ricevuti dal client.

2.2 Gestione comandi utenti

La funzione che gestisce i comandi utente entra in un loop infinito per legge la stringa in arrivo nel socket associato all'utente mediante la funzione read, che data la sua natura bloccante non manda il programma in un loop infinito ma resta in attesa sino a quando l'utente non invia un *ritorno a capo.* Con il ritorno capo la funzione ritorna e il ciclo while ricomincia per fare il parsing del comando appena inviato

3 Sviluppo ed implementazione

3.1 Stumenti per lo sviluppo

Per lo sviluppo di questo progetto sono stati utilizzati vari tool esterni, ed esempio:

CMake CMake è un software libero multipiattaforma per l'automazione della compilazione il cui nome è un'abbreviazione di cross platform make.

Git Git è un software di controllo versione distribuito utilizzabile da inter- faccia a riga di comando.

Doxygen Doxygen è una applicazione per la generazione automatica della documentazione a partire dal codice sorgente di un generico software.

Atom "A hackable text editor" Un semplice editor di testo creato dalla comunità di github appositamente per gli sviluppatori

Lyx Lyx è un sistema per la preparazione di documenti tex-like

3.2 Implementazione del codice

Di seguito verranno presentate e commentate le parti di codice più importanti essendo impossibile commentare tutti il codice date le sue notevoli dimensioni

Il server per funzionare dipende dalle librerie standard del C e da due librerie svilippate appositamente per gestire gli utenti e i canali.

3.2.1 Libreria user

Questa libreria fornisce una struttura per descrivere gli utenti ed una lista doppiamente concatenata per collezionare gli utenti

Particolarmente interessanti risultano le funzioni di creazione utente ed inserimento dell'utente nella lista.

```
1  User* create_user(string name, string hostname,int id, int socket){
2    User* u =  malloc(sizeof(User));
3    u -> name = malloc(strlen(name));
4    u -> hostname = malloc(strlen(hostname) + 1);
5    strcpy(u -> name, name);
6    strcpy(u -> hostname, hostname);
7    u -> id = id;
8    u -> socket = socket;
9    for (size_t i = 0; i < 15; i++) {
10     u -> channels[i] = NULL;
11   }
12   return u;
13 }
```

La funzione alloca una zona di memoria per il nuovo utente e per le stringhe che conterranno nome utente e hostname, poi copia i parametri della funzione nei rispettivi campi dell'utente, infine inizializza a NULL il vettore che conterrà i nomi dei canali a cui l'utente è collegato.

```
1  int add_user(User_list** list, User* user){
2    // Utente nullo o già presente
3    if(user == NULL || find_by_id(*list,user->id) != NULL){
4      return -1;
5    }
6    User_list* temp = malloc(sizeof(User_list)); // allocazione memoria
7    temp -> payload = user; // inserisco l'utene nel nodo di lista
8    //gestisco il caso critico: il primo inserimento
9    if(*list == NULL){
10     temp -> prev = NULL;
11     temp -> next = NULL;
12   } else {     //inserimento in testa
13     temp -> prev = NULL;
14     temp -> next = *list;
15     (*list) -> prev = temp;
16   }
17   *list = temp; // riassegnazione della testa della lista
18   return 1; // ritorno di successo
19 }
```

La funzione add_user gestisce l'inserimento nella lista di utenti, come parametri accetta l'indirizzo del puntatore alla struttura lista in cui inserire il nuovo utente

Come prima cosa controlla che l'utente sia un utente valido oppure non sia già presente nella lista, in questo caso non modifica la lista (non è permesso l'inserimento multiplo di utenti) e ritorna *"-1"* come valore, ad indicare che l'operazione di inserimento non è andata a buon fine.

Poi si ditiunguono due casi, il caso critico ovvero il primo inserimento e il caso in cui almeno un utente è presente nella lista.

Il primo inserimento è un caso critico in quanto l'operazione

```
(*list) -> prev = temp
```

è vietata perché si cerca di accedere ad un campo di un puntatore nullo.

Infine riposiziona il puntatore passato come parametro alla testa della lista.

3.2.2 Libreria channel

Questa libria fornisce una struttura per descrivere i canali ed una lista doppiamente concatenata collezionare i canali.

La struttura le funzioni che operano sulla struttura Channel_list sono una copia opportunamente modificata delle funzioni che operano sulla lista User_list, infatti entrambe le liste hanno una struttura di lista doppiamente concatenata ma accettano come oggetto da collezionare due tipi di dato diversi.

Per completezza verranno trattati la funzione di rimozione di un canale dalla lista, che a parte modifiche sul tipo di dato di ritorno è uguale alla funzione remove_user della libreria user_lib

```
1  Channel* remove_channel(Channel_list** list, string name){
2    if(*list == NULL) return NULL;
3    Channel_list* temp = *list;
4    //scorro la lista per portarmi al canale scelto
5    while (strcmp(temp -> payload -> name, name) != 0) {
6      if(temp -> next == NULL) {
7        return NULL;
8      } else {
9        temp = temp -> next;
10     }
11   }
12   if(temp -> next == NULL && temp -> prev == NULL){ //c'è un solo nodo
13     (*list) = NULL;
14   } else if(temp -> prev == NULL){ //caso in cui il nodo da rimuovere è il primo
15     temp -> next -> prev = NULL;
16     *list = (*list) -> next;
17   } else if(temp -> next == NULL) { //devo rimuovere l'ultimo
18     temp -> prev -> next = NULL;
19   } else {
20     temp -> prev -> next = temp -> next;
21     temp -> next -> prev = temp -> prev;
22   }
23   Channel* chan = temp -> payload; //salvo il canale
24   free(temp); // libero la cella di memoria puntata
25   return chan; //ritoro il canale
26 }
```

Il primo controllo che fa la funzione riguarda la correttezza del puntatore alla lista, se esso è NULL non ha senso cercare un nodo da rimuovere che non

c'è e anzi se si cercassero di fare operazioni su tale puntatore si solleverebbe un errore di *"core dump"* che farebbe terminare il programma.

Se il primo controllo va a buon fine, con un puntatore temporaneo si scorre la lista fino a puntare il nodo da eliminare.

Per rimuoverlo si possono identificare quattro casi, tre critici con gestioni leggermente diverse dal caso generale, essi sono la rimozione del nodo centrale, la rimozione del nodo in coda e la presenza di un sono nodo. Sono casi critici perché nel caso rimozione del nodo di testa e rimozione del nodo in coda vengono svolte operazioni che causano un errore di *"core dump"* ed esse sono rispettivamente

```
1 temp -> prev -> next = temp -> next;
2 temp -> next -> prev = temp -> prev;
```

Le operazioni tentano di eseguire un accesso ad un parametro di una struttura che punta a NULL.

Infine nel caso di un solo nodo sono violate contemporaneamente perche un sono nodo ha i puntatori precedente e successivo a NULL.

Una volta rimosso il nodo di lista che contiene il canle cercato, colleziono con un nuovo puntatore il canale cercato, libero la zona di memoria occupata dal nodo della lista ed infine ritorno il canale cercato, sarà il chiamante a decidere cosa farne.

Una differenza tra struttua Channel e la struttura User è il fatto che all'interno della struttura Channel esiste una lista che colleziona gli utenti presenti nel canale mentre nella struttura User è presente un vettore di stringhe contente i nomi di canali a cui l'utente è accreditato. Ho adoperato questa scelata perché è conveniente ai fini della velocità di esecuzione del programma in quanto l'invio di messaggi nel canale è un'operazione svolta con maggior frequenza, mentre l'abbandono del server, che è l'unico motivo per cui esiste questo vettore, è molto più raro e può avere tempi di latenza maggiori rispetto all'invio di messaggi, operazione che deve essere svolta in modo più veloce.

3.2.3 Libreria utils

Come suggerisce il nome contiente funzioni di utilità, utilità a gestire i casi del comando WHO infatti le tre funzioni implementate corrispondono ai vari casi in cui viene chiamato il comando WHO prendiamo in esame il caso in cui viene chiamato il comando WHO e come parametro viene inserito il nome di un canale.

```
1  void send_channel_info(User* u, Channel* c) {
2    User_list* list = c -> users;
3    User* target;
4    char* send_line = malloc(MAXLINE + 1);
5    while (list != NULL) {
6      target = list -> payload;
7      send_user_info(u, target, c -> name);
8      list = list -> next;
9    }
10   strcpy(send_line, ":");
11   strcat(send_line, SERVER_NAME);
```

```
12    strcat(send_line, " ");
13    strcat(send_line, RPL_ENDOFWHO);
14    strcat(send_line, " ");
15    strcat(send_line, u -> name);
16    strcat(send_line, ENDOFWHO);
17    write(u -> socket, send_line, strlen(send_line));
18    free(send_line);
19  }
```

Le operazioni iniziali creano un puntatore alla lista utenti del canale per raccogliere ed inviare all'utente u che ha chiamato il comando WHO le informazioni di tutti gli utenti presenti nel canale. Il ciclo while serve appunto per iterare la lista e inviare le informazioni di tutti gli utenti presenti nel canale.

Infine viene costruita la striga per indicare la fine della lista in risposta al comando WHO, viene inviata e liberata la zona di memoria occupata.

3.2.4 Libreria user_thread

La libreria principe che contiene la funzione da associare al thread dell'utente, essa gestisce i comandi inviati dall'utente.

```
1   void user_thread(User* u){
2     char recvline[MAXLINE + 1];
3     char* line = malloc((MAXLINE + 1) * sizeof(char));
4     char* send_line = malloc((MAXLINE + 1) * sizeof(char));
5     char* command;
6     ssize_t n;
7     pthread_mutex_lock(&(u -> socket_mutex));
8     n = read(u -> socket, recvline, MAXLINE);
9     pthread_mutex_unlock(&(u -> socket_mutex));
10    while(n > 0){
11      recvline[n] = 0;
12      line = strcpy(line, recvline);
13      command = strtok(line, " \t\r\n/");
14      while (command != NULL) {
15        printf("L'utente: %i ha inviato il comando: %s\n",u -> id, command);
16        /*Qui parte lo switch di comparazaione con i comandi conosciuti*/
17            if(strcmp(command, NICK) == 0){
18                  ...
19            }
20            ...
21        command = strtok(NULL, " \t\r\n/");
22      }
23      pthread_mutex_lock(&u->socket_mutex);
24      n = read(u->socket, recvline, MAXLINE);
25      pthread_mutex_unlock(&u->socket_mutex);
26    }
27  }
```

Nella parte iniziale creo tutte le variabili per poter fare un parsing della stringa line, ovvero quella che conterrà il comando con i suoi parametri.

Per poter leggere la stringa di dati uso la funzione read sul socket dell'utente ma, essendo il socket una risorsa condivisa, ed accessibile conteporanemente da più thread, ho usato il sistema di semafori messo a disposizione dalla libreria pthread per gestire la lettura e scritura concorrente da parte di più utenti, in

particolare le funzioni pthread_mutex_lock e pthread_mutex_unlock bloccano o liberano la risorsa condivisa o meglio il semaforo associato alla risorsa.

Segue il ciclo while principale per gestire la successione di stringhe inviate dall'utente, ed annidato il secondo ciclo while per parsare la stringa con la funzione strtok e selezionare il comando ricchiesto dall'utente mediante una serie di decisioni if - else if.

3.2.5 File main.c

Nel main file c'è l'unica funzione main, la principale, il cui unico compito è gestire le connessioni in ingresso dai vari utenti

```
1  while(1){
2      //accept non è all'interno dell'if di decisione perché è una funzione bloccante
3      if ((user_socket = accept(server_socket,
4                                (struct sockaddr *) &client_address,
5                                &client_address_size)) == -1 ) {
6        fprintf(stderr, "[Errore chiamando accept: %s]\n", strerror(errno));
7        exit(5);
8      }
9      // aggiunge l'utente alla lista se permesso
10     if(count < MAXUSER){
11       strcpy(username, "user"); //pulisco la stringa dal nome precendente
12       sprintf(temp,"%d", u_id);
13       strcat(username, temp);
14       new_user = create_user( username,
15                               inet_ntoa(client_address.sin_addr),
16                               u_id,
17                               user_socket);
18       u_id = u_id + 1; //incrementa il contatore degli id
19       pthread_mutex_lock(&main_user_list_mutex);
20       add_user(&main_user_list,new_user);
21       count = count + 1; //va a braccetto con la lista, deve andare qui
22       pthread_mutex_unlock(&main_user_list_mutex);
23       pthread_create(&(new_user -> thread), NULL, (void* (*)(void*))user_thread, new_user);
24     } else { //troppi utenti glielo dico
25       close(user_socket); // così rifiuto la connessione, chiudendo il socket
26     }
27 }
```

Prima di questo loop infinito, ovviamente ci sono tutte le chiamate a funzione per inizializzare la porta e le strutture dati necessarie per gestire il cliente.

Interessante è il ruolo giocato dalla funzione accept, una funzione bloccante che ritorna solo se c'è un elemento nella coda di attesa delle connessioni, se la code fosse vuota, la funzione accept aspetta finchè non c'è una connessione da gestire.

La funzione estrae la prima connessione dalla coda del listen socket, ovvero il socket associato al servizio creato con le funzioni precedenti, crea un nuovo socket per l'utente e ritorna un file descriptor per tale socket, il nuovo socket creato non è nello stato di listen.

Se l'operazione di accettazione è andata a buon fine e non ci sono troppi utenti nel sistema, inserisco il nuovo utente nella lista principale degli utenti,

aumento il contatore associato alla lista e associo all'utente il thread che dovrà gestire i comandi.

Se invece l'operazione di accettazione è andata a buonfine ma ci sono troppi utenti, semplicemente non creo l'utente e chiudo il socket appena creato.

Infine se l'accettazione non è andata a buon fine termino il programma, evento abbastanza critico ma necessario perchè se la funzione accept fallisce è sintomo di problemi di rete.

4 Valutazione e collaudo

Il programma è stato collaudato man mano con esempi inseriti manualmente ed ogni libreria presente nel progetto è stata provata in diversi scenari durante lo sviluppo cercando di eliminare i bug che si pressentavano via via che il progetto cresceva, importanti sono state le modifiche alla struttra del progetto per la gestione dei comandi WHO e PART. Per risolvere in modo efficente il problema di raggiungere tutti i canali a cui un utente è collegato ho dovuto aggiungere il campo *channels* all'interno della struttura *User* con conseguente modifica dei metodi per la creazione e rimozione dell'utente.

Per testare tutti i casi critici delle liste utente e canale mi sono avvalso del framework Check creando due test case ed è grazie a questi test che sono riuscito a identificare e risolvere i casi critici che non avevo considerato.

Un altro bug che mi ha fatto penare è stata la dimenticanza del carattere "$\backslash n$" come terminazione delle stringhe di invio all'utente solo a posteriori ho capito dell'importanza di tale terminazione e, dopo una sera intera alla ricerca dell'errore, sono riuscito a far funzionare il comando PRIVMSG, comando che per primo presentò tale bug.

Altri possibili bug sono dovuti all'utilizzo improprio dei parametri dei vari comandi, ad esempio se si cerca di chiamare il comando PING e passando come paramtro il nome di un utente, c'è una possibilità significativamente alta che il programma vada in crash. Nonostante tale configurazione sia lecita da specifica di protocollo.

5 Conclusioni e lavoro futuro

Nonostante le molteplici difficoltà trovate, specialmente nella gestione dei comandi utenti, mi sento soddisfatto del lavoro svolto.

Ho acquisito molte più competenze specialmente nella gestione di un progetto mediante git. Sicuramente continuerò ad approfondire la programmazione di rete anche in vista della tesi e della laurea magistrale.

Possibili sviluppi del software potrebbero includere:

- Ampliare lo spettro dei comandi gestibili dal server
- Possibilià dello scambio di file
- Estensione del comando PRIVMSG al caso utente-utente
- Una migliore formattazione delle risposte al client
- Implementazione di tutti i casi