Di: Davide Grimaldi
Email: davidegrimaldi92@gmail.com
Ingegneria Informatica Catania

PROGETTO CLIENT IRC SISTEMI OPERATIVI

A.A. 2017/2018

Una breve descrizione degli obiettivi raggiunti, di come sono state superate le difficoltà incontrate, e delle conoscenze e abilità ottenute

TABLE OF CONTENTS

Contents

PROGETTO CLIENT IRC

Intro

STILE E SCOPO DEL DOCUMENTO

Si procederà descrivendo il funzionamento generale e poi di ogni componente software concentrando l'attenzione sui punti più importanti: conoscenze utilizzate, difficoltà affrontate, linee di codice chiave. **La lettura è da accompagnare alla visione del codice nella parte relativa alle componenti software**.
In conclusione conoscenze ed abilità ottenute e considerazioni personali.

Lo scopo è quello di descrivere quale è stato l'approccio alle problematiche sorte durante l'elaborazione del progetto, quelle risolte e in che modo.

OBIETTIVI PROGETTO

L'obiettivo è quello di realizzare un Client IRC utilizzando *socket* TCP (SOCK_STREAM) funzionante che permetta l'utilizzo delle normali funzionalità del protocollo IRC e tramite un *executer* fare eseguire da remoto comandi sulla Shell.

Funzionamento generale

FONDAMENTALI PROTOCOLLO IRC

"Internet Relay Chat (IRC) è un protocollo di messaggistica istantanea su Internet. Consente, sia la comunicazione diretta fra due utenti, che il dialogo contemporaneo di gruppi di persone raggruppati in stanze di discussione, chiamate canali." Wikipedia

Il protocollo IRC è stato creato sul finire degli anni '80 con lo scopo di aggirare censure dei mass media, esso infatti venne utilizzato durante eventi storici di rilievo come la prima guerra del golfo e il colpo di stato sovietico del 1991 per diffonde notizie ed informazioni su ciò che stava realmente accadendo, un po' come il Twitter dei giorni nostri.

La struttura generale della rete IRC è di tipo client-server con i server che hanno una struttura ad albero, ma la caratteristica del protocollo più importate al fine del progetto è che la comunicazione avviene tramite l'invio e la ricezione di stringhe e il "pacchetto" massimo ha una lunghezza di 512 caratteri (questo è un dato importante come si vedrà nel seguito). Le operazioni ed istruzioni che si vogliono svolgere (es. mandare messaggi) sono segnalate al server tramite parole chiave che iniziano i *comandi* che il server interpreta, qui elenco quelli fondamentali che ho utilizzato:

`NICK <nickname>`
Permette al client di cambiare il suo nickname, ma è anche uno dei comandi che deve essere inizialmente mandato al server per accedervi.

`USER <user> <mode> <unused> <realname>`
Viene usato all'inizio della connessione per specificare *username, realname* e inizializza la modalità del client che si sta connettendo (Es. USER username 0 * : realname). La *modalità* può dare caratteristiche speciali ad un utente che si connette come per esempio la caratteristica di non poter ricevere messaggi privati e altri ancora, lo zero indica nessuna modalità speciale.

`PING <server1> [<server2>] PONG <server1> [<server2>]`
Importanti, verranno descritti in seguito.

`PRIVMSG <msgtarget> <message>`
Permette di mandare messaggi a canali o utenti. NOTA: i vari campi dopo la parola chiave

che compongono il comando devono essere separati da ":" se si intende utilizzare più parole separate da spazi, il mio primo messaggio in chat derivante dal comando "PRIVMSG #canale hi everyone !!!" è stato "hi" il comando corretto per mandare tutto il messaggio è "PRIVMSG #canale : hi everyone !!!" (Da questo ho dedotto che il server scarta gli argomenti in sovrannumero).

`QUIT [<message>]` Disconnette dal server.

`JOIN <channels> [<keys>]`
Permette di entrare in un canale e specificare la password se richiesta.

IMPOSTAZIONE D'INSIEME

La struttura generale del programma è di un client che provveda al "set-up" del socket e alla sua connessione con il server (di cui il vengono presi nome e porta in ingresso come parametri passati al lancio del programma).

Il client è strutturato in due *thread* uno di scrittura che è sempre in attesa di input dall'utente e lo manda tramite la *send* ed un altro di lettura che è sempre in ascolto sul server tramite la *recv* (send e recv sono i rispettivi più "robusti" di write e read). Lo sviluppo su due *thread* permette una evidente migliore gestione del codice e migliori prestazioni.

L' *executer* è implementato dentro il *thread* di lettura ed esamina il buffer cercando sempre la parola chiave che precede il comando che verrà eseguito su shell, funzionando di fatto come un piccolo malware.

Componenti software

SOCKET_SET_AND_CONNECT ()

La base su cui poggia tutto il programma e la sua parte iniziale. Riceve come parametri quelli passati al programma al suo lancio (*hostname* e porta), ritorna l'identificativo del socket. Qui ho utilizzato tutte le conoscenze acquisite durante il corso di sistemi operativi sui socket, brevemente:

1) Viene creato un *unnamed socket*.

2) Si ottengono le informazioni sull'host tramite *gethostbyname( )*.

3) Si utilizzano queste per impostare il socket.

4) Si procede con la *connect*.

Non ho avuto difficoltà in questa parte, qualche piccolo inceppo nelle seguenti righe di codice, risolto subito con gli opportuni cast e conversioni:

```
server_addr.sin_addr=*(struct in_addr *)*host->h_addr_list;/*il
puntatore al primo elemento della lista h_addr_list viene castato al
tipo struct in_addr e poi deferenziato*/
server_addr.sin_port=htons(atoi(argv[2]));//converte prima la stringa
in intero e poi il numero di porta in formato network
```

CLIENT: *WRITETHREAD_FUNCTION ()*

La funzione che esegue il thread di scrittura, inizia con una serie di istruzioni che inviano automaticamente al server le informazioni di accesso (prese in input dall'utente), comandi NICK e USER, successivamente si mette in ascolto sullo standard input tramite la *fgets( ),* memorizza su buffer e successivamente manda al server. Da notale la MAX_BUF_LEN di 512 che è la massima in entrata/uscita dal server. Necessaria la pulizia del buffer che avviene con *memset( )* accompagnata a quella dello standard input, la funzione *bzero( )* che avevo visto sulle slide ho scoperto essere deprecata. Il funzionamento del thread cessa con il comando QUIT che corrisponde a quello per chiudere la connessione col server.

CLIENT: *READTHREAD_FUNCTION ()*

La funzione che esegue il thread di lettura è sempre in attesa di dati dal server che vengono memorizzati nel *read_buffer* e poi stampati, gestisce il caso in cui la connessione venga chiusa dal server e quando il server manda PING. Il segnale PING viene periodicamente mandato dal server per verificare se un utente è ancora in linea, se non riceve nessuna risposta del tipo PONG entro un determinato periodo di tempo il server chiude la connessione, per questo quando viene ricevuto viene mandato in automatico il PONG di risposta. Dentro il thread di lettura viene inserito l'executer descritto di seguito.

EXECUTER ()

L'executer analizza continuamente il buffer del thread di lettura e tramite la funzione *strstr ( )* cerca la prima occorrenza della parola chiave (KEYWORD_COMANDO:), se viene trovata procede nell'estrazione dal buffer, che viene passato come parametro, del comando da eseguire sulla shell tramite i seguenti passaggi:

```
if((token=strstr(buffer,PAROLACHIAVE))!=NULL) //strstr restituisce
dalla prima occorezzanza della parola chiave in poi

token=strtok(token,PAROLACHIAVE); //separo la parola chiave dal
comando sfruttando la funzione strtok che divide la stringa data in
token il cui separatore è la key word, restituisce il primo token
ottenuto cioè il primo comando dopo la prima parola chiave

strcpy(command,token);

command[strlen(command)-2]='\0'; //tolgo gli ultimi 2 caratteri in
più del server {carriage return(13) line feed(10)}
```

quest' ultima riga di codice è fondamentale per il funzionamento dell'executer, infatti senza di essa all'esecuzione del comando si riceve un "*not found*", il motivo sta nel fatto che il server aggiunge alla fine della stringa che viene mandata due caratteri speciali, non sono visibili tramite *printf( )* per questo nonostante i controlli aggiunti nel codice non riuscivo ad identificare il problema, fino a quando non ho aggiunto una *strlen( )* per contare effettivamente il numero dei caratteri, così ho identificato il problema ed ho "ripulito" la mia stringa comando che successivamente viene eseguito tramite la funzione *system( )* [o alternativamente *popen( )*].

INTERFACCIA (NON IMPLEMENTATA)

Per realizzare l'interfaccia ho utilizzato gtk3+ un toolkit per creare interfacce grafiche (supporta il C). **Nella realizzazione ho commesso l'errore di non verificare la sua fattibilità in tempi ragionevoli**, gtk3+ è difficoltoso per i neofiti della programmazione delle interfacce. Partendo dal presupposto che programmare un'interfaccia in C è già poco raccomandabile, gtk3+ ha il grandissimo problema di non avere sufficiente documentazione per chi vi si approccia. Non ci sono guide, tutorial, libri da cui partire, o meglio, ci sono ma si riferisce tutto alla versione gtk2+ che ha delle sostanziali differenze e incompatibilità con la nuova. Per gtk3 l'unica documentazione completa è quella sul sito ufficiale degli sviluppatori che presenta i primi semplici esempi di utilizzo per poi diventare un "vocabolario" di funzioni prive di spiegazioni.

Utilizzando questo "vocabolario" sono riuscito a costruire le mie interfacce, testate e funzionanti, il problema si è presentato nel collegare queste con il client. Inizialmente, studiando le *pipe* da Beginning, ho pensato di utilizzarle per effettuare questo collegamento: non ne sono venuto a capo. Successivamente ho cercato di capire proprio tramite la documentazione ufficiale i metodi propri di gtk3+ per presentarsi come applicazione di un altro programma: ho fallito ancora. Infine ho cercato di aggirare il problema tramite qualcosa già di mia conoscenza: **le code di messaggi**. L'idea era semplice, utilizzare una coda di messaggi da tramite tra i thread di lettura e scrittura e la parte dell'interfaccia deputata all'output (TextView) e quella all'input (TextEntry).

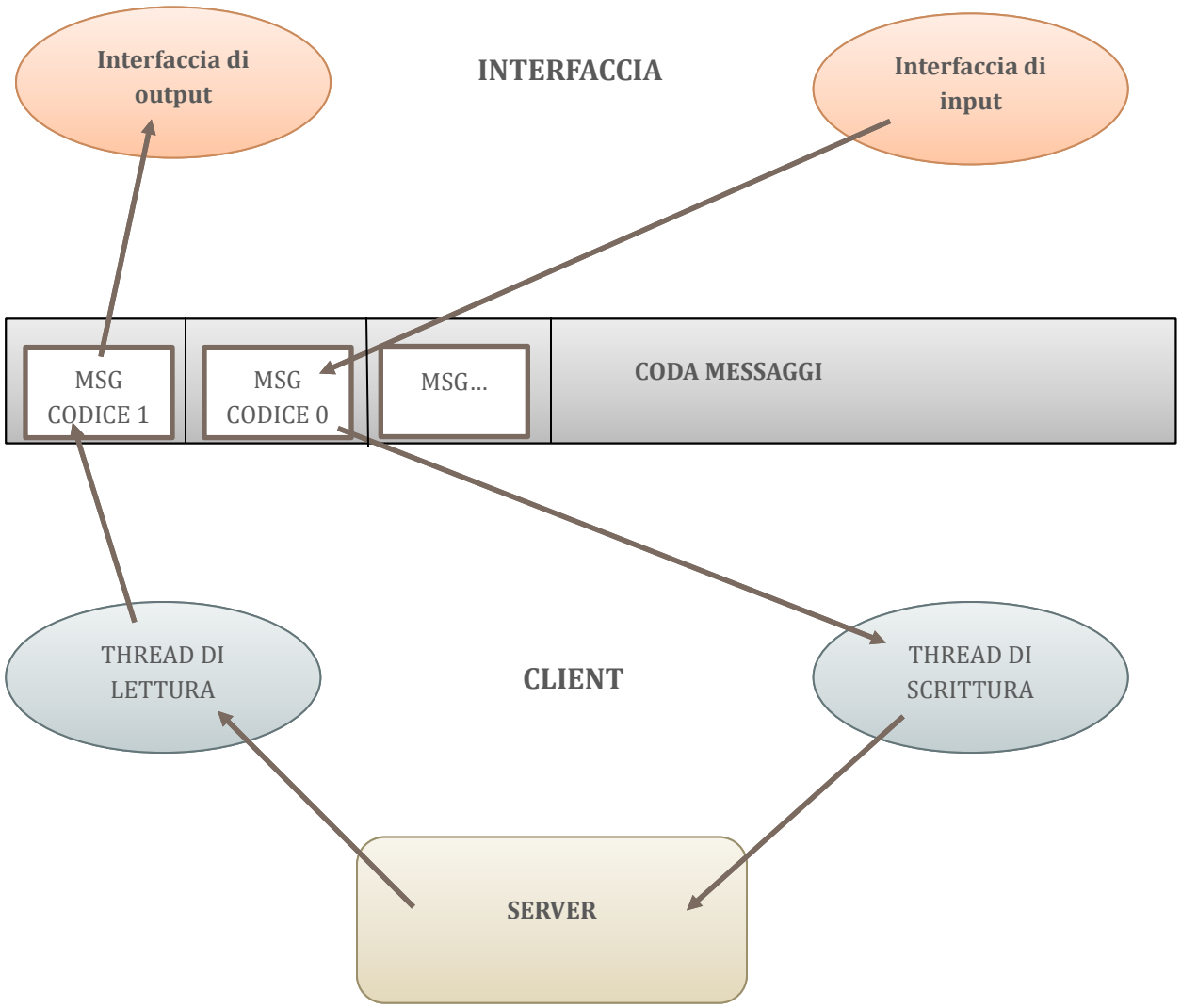
DESCRIZIONE IDEA CON CODA DI MESSAGGI

Per prima cosa dichiaro il tipo messaggio della coda

```
struct s_message //struct msg
	{
	unsigned short int message_type; //from server code 1 to server code 0
	char message[MAX_BUF_LEN];
	};
```

message_type vale come codice di riconoscimento per gestire i flussi dei messaggi fra le componenti software che accedono alla coda. Il codice 1 indica un messaggio che contiene testo proveniente dal server, il thread di lettura invece di stampare in output ciò che riceve dal server lo manderà sulla coda con il codice 1, la funzione che si occuperà dell'output sull'interfaccia sarà sempre in attesa di messaggi sulla coda con codice 1, ed essendo la coda FIFO manterrà l'ordine corretto in output. Questo è possibile perché

nella funzione *msgrcv ( )* della coda di messaggi è possibile specificare il tipo di messaggi da ricevere; lo stesso ragionamento speculare vale fra il thread di scrittura e l'interfaccia di input[VEDI FIGURA]. Vantaggi di questo approccio:
1) modularità componenti (possono essere sostituite o cambiare indipendentemente).
2) nessun intralcio fra le sopra citate, nessuno aspetta nessuno (no bisogno di sincro).
3) con l'aggiunta altri codici di messaggio e di uno "smistatore" nel thread di lettura è possibile implementare interfacce multiple.
L'approccio con la coda di messaggi è stato tentato ma si sono riscontrati problemi, sicuramente imputabili alla mia scarsa conoscenza delle interfacce costruite con gtk3+. Dopo diversi tentativi falliti ho preferito concentrarmi sul client ed executer, è comunque disponibile il codice(non funzionante). Successivamente ho scoperto che le code POSIX sono in disuso da anni ed esisto tante altre tecniche che ne rappresentano l'evoluzione.



Conclusioni

CONOSCENZE ED ABILITÀ OTTENUTE

Durante lo svolgimento del progetto ho acquisito:

-Migliore conoscenza ed abilità lavoro su più file
-Migliore capacità di raccolta ed utilizzo informazioni dalla rete
-Conoscenza di piattaforme per il lavoro in team online e loro utilizzo: *Slack* e *GitHub*
-Migliore gestione del tempo di lavoro
-Conoscenza generale ed utilizzo di Pipe (non svolte durante il corso)
-Conoscenza di base programmazione interfacce con gtk3+

In particolare conoscere GitHub è stato molto utile, essendo una piattaforma su cui è possibile trovare veramente di tutto (parlando dal punto di vista informatico).

CONSIDERAZIONI PERSONALI

La mia valutazione personale sull'iniziativa è sicuramente positiva, dispiace vedere così pochi colleghi interessati, potrebbe essere proposta un'attività del genere come "attività extra" per cui vengano attribuiti 3 crediti. Personalmente oltre ad aver arricchito non di poco il mio bagaglio culturale da futuro ingegnere informatico, mi sono anche divertito, peccato aver sprecato tempo (che era già molto limitato) per l'interfaccia, e non averlo dedicato alla realizzazione di un client migliore. Durante il progetto ho volutamente cercato il meno possibile aiuto volendo mettermi alla prova, esattamente come ho fatto scrivendo questo documento.

Fonti

Link al mio profilo GitHub:
https://github.com/Ishikawa7

Informazioni su IRC:
https://it.wikipedia.org/wiki/Internet_Relay_Chat
https://en.wikipedia.org/wiki/List_of_Internet_Relay_Chat_commands
https://www.unrealircd.org/docs/User_Modes

Dove ho trovato tutte le funzioni del C che mi sono servite:
https://www.tutorialspoint.com/c_standard_library/index.htm

Teoria su thread, socket e code di messaggi:
-Slides prof. Lucia Lo Bello (sistemi operativi)
-Beginning Linux Programming CAPITOLI 12-14-15

Teoria sulle pipe:
-Beginning Linux Programming CAPITOLO 13

La documentazione ufficiale su gtk3+:
https://developer.gnome.org/gtk3/stable/