



UNIVERSITÀ DEGLI STUDI DI PADOVA

LABORATORIO DI INGEGNERIA INFORMATICA

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

A.A. 2015/16

c-chat

Chat multiutente con architettura client/server

Enrico Vianello [mat. 1068326]

30 giugno 2016

1 INTRODUZIONE

c-chat è un semplice programma di chat eseguibile da terminale per sistemi operativi GNU/Linux e iOS.

Offre la possibilità ad un grande numero di utenti di scambiare messaggi di testo attraverso un'architettura di tipo client/server.

Il progetto vuole porsi come un'alternativa veloce e minimale a servizi di chat utilizzati per lavoro o interessi comuni.

L'implementazione è realizzata in linguaggio C per mantenere il programma efficiente e di ridotte dimensioni. Infatti, data la natura non troppo complessa di questo progetto, è possibile sfruttare appieno le potenzialità del linguaggio senza particolari impedimenti.

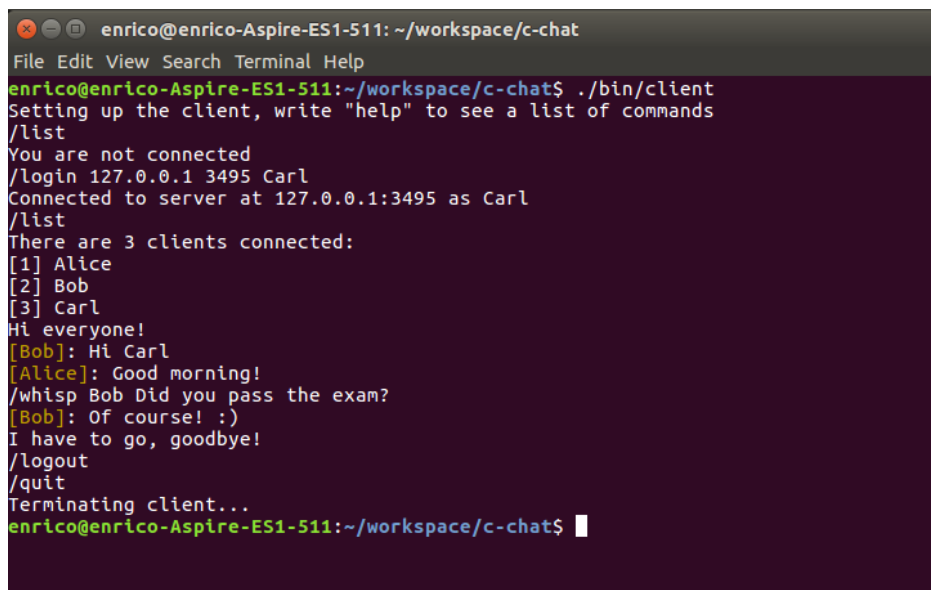
2 DESIGN DEL PROGETTO

Il progetto è sostanzialmente suddiviso in due parti distinte, ognuna delle quali dotata di un eseguibile: client e server.

2.1 CLIENT

È la parte del programma destinata all'utente finale che permette di inviare messaggi e visualizza quelli ricevuti a terminale.

Richiede la connessione ad un server e consente di impostare un alias che avrà funzione di identificatore dell'utente.

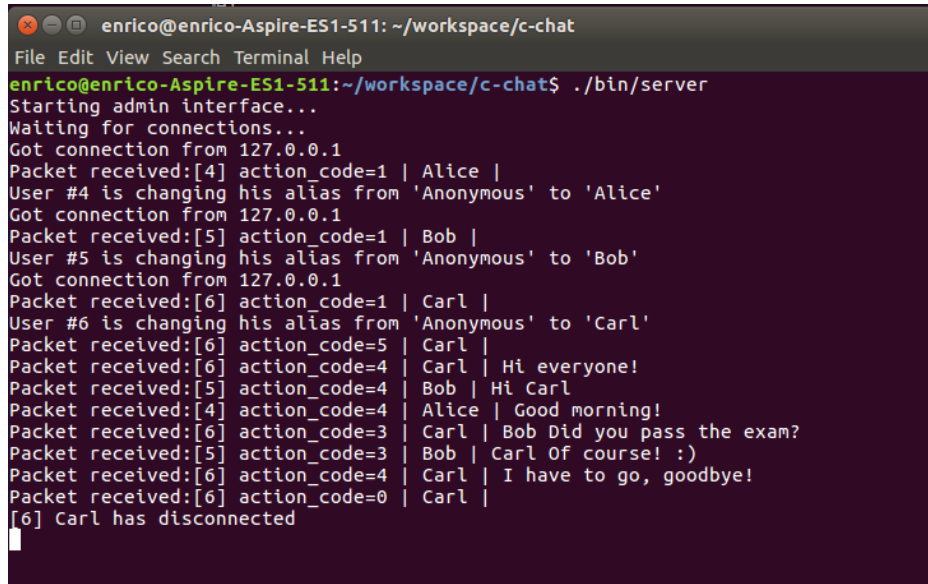


```
enrico@enrico-Aspire-ES1-511: ~/workspace/c-chat
File Edit View Search Terminal Help
enrico@enrico-Aspire-ES1-511:~/workspace/c-chat$ ./bin/client
Setting up the client, write "help" to see a list of commands
/list
You are not connected
/login 127.0.0.1 3495 Carl
Connected to server at 127.0.0.1:3495 as Carl
/list
There are 3 clients connected:
[1] Alice
[2] Bob
[3] Carl
Hi everyone!
[Bob]: Hi Carl
[Alice]: Good morning!
/whisp Bob Did you pass the exam?
[Bob]: Of course! :)
I have to go, goodbye!
/logout
/quit
Terminating client...
enrico@enrico-Aspire-ES1-511:~/workspace/c-chat$
```

Figura 2.1: L'applicazione client

2.2 SERVER

È l'applicazione che dovrà essere avviata nel terminale al quale si conatteranno i client. Ha il compito di ricevere messaggi e comandi inviati dai vari client e agire di conseguenza (smistando il messaggio o ottemperando all'azione associata al comando ricevuto).



```
enrico@enrico-Aspire-ES1-511: ~/workspace/c-chat
File Edit View Search Terminal Help
enrico@enrico-Aspire-ES1-511:~/workspace/c-chat$ ./bin/server
Starting admin interface...
Waiting for connections...
Got connection from 127.0.0.1
Packet received:[4] action_code=1 | Alice |
User #4 is changing his alias from 'Anonymous' to 'Alice'
Got connection from 127.0.0.1
Packet received:[5] action_code=1 | Bob |
User #5 is changing his alias from 'Anonymous' to 'Bob'
Got connection from 127.0.0.1
Packet received:[6] action_code=1 | Carl |
User #6 is changing his alias from 'Anonymous' to 'Carl'
Packet received:[6] action_code=5 | Carl |
Packet received:[6] action_code=4 | Carl | Hi everyone!
Packet received:[5] action_code=4 | Bob | Hi Carl
Packet received:[4] action_code=4 | Alice | Good morning!
Packet received:[6] action_code=3 | Carl | Bob Did you pass the exam?
Packet received:[5] action_code=3 | Bob | Carl Of course! :)
Packet received:[6] action_code=4 | Carl | I have to go, goodbye!
Packet received:[6] action_code=0 | Carl |
[6] Carl has disconnected
```

Figura 2.2: L'applicazione server

Va notato che, affinché la comunicazione tra due o più client abbia successo, è assolutamente necessario che entrambi siano collegati al medesimo processo server di cui è noto l'indirizzo. Inoltre il progetto è stato realizzato in modo tale che non vi sia alcuno scambio di informazione diretto tra due client, e che nessuno di questi conservi informazioni relative agli altri. Ciò permette di garantire che qualsiasi dato proveniente dalla chat possa essere monitorato o filtrato dal server per ragioni di sicurezza.

2.3 INTERFACCIA UTENTE

Il programma è pensato per essere utilizzato da terminale e sia la parte client che la parte server accettano una serie di comandi predefiniti aventi forma `~/comando`. Tali comandi sono facilmente consultabili in una pagina di aiuto.

Qualsiasi stringa in input che non inizia con il carattere `~/` viene considerata un messaggio di chat e in quanto tale inviata agli altri utenti.

I messaggi ricevuti sono visualizzati nella stessa finestra del terminale preceduti dall'alias del mittente.

3 PROTOCOLLO DI COMUNICAZIONE

Per la comunicazione tra client e server viene utilizzato il protocollo TCP/IP mediante socket. Tale protocollo è stato preferito ad UDP/IP perchè più robusto, alla luce del fatto che la mole di dati da trasmettere è ridotta (in quanto si tratta il più delle volte di messaggi testuali).

Sono supportati sia indirizzi in formato IPv4 che IPv6.

Ogni informazione trasmessa è codificata all'interno di pacchetti; questi sono di uguale dimensione e struttura per qualsiasi tipo di azione intrapresa. Tale scelta è stata effettuata al fine di limitare la complessità del codice e velocizzare il processo di decodifica, pur aumentando potenzialmente il traffico di dati e l'utilizzo di memoria (elementi comunque non critici data la modesta dimensione dei pacchetti).

```
struct Packet {  
    unsigned char action;  
    char alias[ALIASLEN]; // ALIASLEN = 32  
    int len;  
    char payload[PAYLEN]; // PAYLEN = 2048  
};
```

Struct relativa ad un singolo pacchetto

Esistono varie tipologie di pacchetti contraddistinte dal campo "action", ognuna delle quali sfrutta gli altri field come specificato di seguito.

- EXIT
Pacchetto inviato da un client per chiedere la disconnessione dal server.
- ALIAS
Pacchetto inviato da un client per chiedere la modifica del proprio alias. Il campo "alias" contiene il nuovo alias.
- MSG
Pacchetto inviato dal server ad un client contenente un messaggio. Il campo "alias" contiene il mittente. Il campo "payload" contiene il messaggio.
- WHISPER
Pacchetto inviato da un client per inviare un messaggio privato ad un altro utente. Il campo "alias" contiene il mittente. Nel campo "payload": il primo token (delimitato da blank spaces) è il destinatario, la stringa seguente è il corpo del messaggio.
- SHOUT
Pacchetto inviato da un client per inviare un messaggio pubblico alla chat. Il campo "alias" contiene il mittente. Il campo "payload" contiene il messaggio.
- LIST_Q
Pacchetto inviato da un client per richiedere la lista degli utenti connessi.

- LIST_A
Pacchetto inviato dal server per comunicare la lista degli utenti connessi. Il campo "len" contiene il numero di elementi della lista. Il campo "payload" contiene la lista sottoforma di stringa, ad ogni elemento è assegnato un numero di caratteri uguale alla massima lunghezza di un alias.
- UNF
Pacchetto "User Not Found" inviato dal server ad un client in risposta ad un pacchetto WHISPER quando il destinatario non è connesso. Il campo "alias" contiene il mittente non trovato.

4 ARCHITETTURA DEL PROGETTO

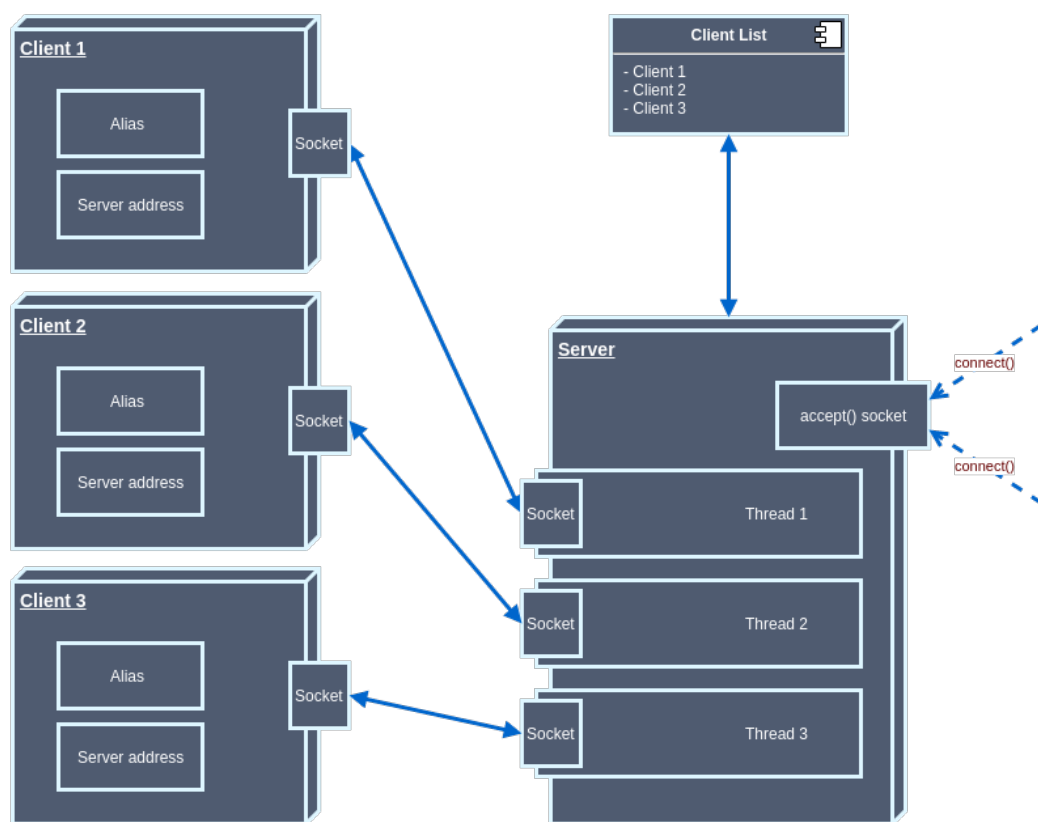


Figura 4.1: Diagramma della struttura del progetto

Il progetto è costituito dai seguenti files:

- client.c (client.h)
- server.c (server.h)

- clientlist.c (clientlist.h)
- networkutil.c (networkutil.h)
- networkdef.h

In gran parte del codice si utilizzano librerie per il networking e la programmazione di socket (presenti nella libreria standard), inoltre per la programmazione concorrente si è utilizzata pthread.h (POSIX thread library).

4.1 CLIENT.C (CLIENT.H)

È il file contenente il metodo main per l'eseguibile del client. Questo, dopo l'inizializzazione, entra in un loop in attesa di un comando dell'utente.

I comandi disponibili sono presenti in un apposito file contenuto nella directory ``helpfiles/`` il cui contenuto viene visualizzato all'inserimento del comando ``/help``.

Brevemente le azioni che attualmente è possibile intraprendere sono le seguenti:

- Connettersi/disconnettersi da un server
- Cambiare nickname
- Inviare un messaggio pubblico/privato
- Ottenere la lista degli utenti connessi
- Uscire dal programma

Ogni comando provoca l'invocazione di un apposito metodo di client.c che effettua le operazioni necessarie come la creazione o l'invio di un pacchetto.

In particolare lo stabilimento della connessione con il server è il tratto più interessante: si può infatti vedere dal codice la sequenza di operazioni necessarie per configurare il socket e avviare la connessione.

```
/**
 * @brief Establish a connection with the server application.
 *
 * @param ip String containing the server's IP (IPv4 or IPv6).
 * @param port String containing the server's listening port.
 *
 * @return Always a \c NULL pointer.
 */
static int connect_server(char *ip, char *port) {
    /* Prepare the server's address */
    struct addrinfo hints;
    struct addrinfo *servinfo; // will point to the results
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
```

```

hints.ai_socktype = SOCK_STREAM;
int status;
status = getaddrinfo(ip, port, &hints, &servinfo)
/* Open a socket */
int newfd;
newfd = socket(servinfo->ai_family, servinfo->ai_socktype,
servinfo->ai_protocol)
/* Try to connect with server */
connect(newfd, servinfo->ai_addr, servinfo->ai_addrlen
/* Free the linked-list used to set up the socket */
freeaddrinfo(servinfo);
freeaddrinfo(&hints);
return 0;
}

```

Metodo connect_server() (privo di error handling)

Inoltre dopo tale operazione, viene eseguito in un thread separato un metodo che rimane costantemente in ascolto sul socket, in attesa di eventuali pacchetti provenienti dal server.

```

/**
 * @brief Routine that constantly listens for incoming packets.
 *
 * @return Always a \c NULL pointer.
 */
static void *receiver() {
    struct Packet packet; // this packet will be used to contain
        the received data
    while(1) {
        if(!(recv(serversfd, (void *)&packet, sizeof(struct
            Packet), 0))) {
            /* When recv returns 0, it means that the connection was
                interrupted */
            fprintf(stderr, "client: connection lost from server\n");
            connected = 0;
            close(serversfd);
            break;
        }
        switch (packet.action) {
            /* Message to display received */
            case MSG : [...]
            /* List of clients received */
            case LIST_A : [...]
            /* There are no clients with the alias specified in the
                whisper command */
            case UNF : [...]

```

```

    }
    /* Clean the packet */
    memset(&packet, 0, sizeof(struct Packet));
}
return NULL;
}

```

Metodo receiver()

4.2 SERVER.C (SERVER.H)

È il file contenente l'implementazione del server.

Dopo essere stato avviato inizializza immediatamente un socket in ascolto per eventuali richieste di connessione e crea quindi un thread separato per gestire i comandi inseribili da terminale (come per il client essi sono disponibili in un file di aiuto, accessibile mediante il comando "/help").

Quindi il programma entra in un ciclo infinito che attende richieste di connessione e, al loro arrivo, crea un nuovo thread per gestire parallelamente in modo indipendente ognuna di queste.

```

while(1) { // main accept() loop
    /* Block the server till a pending connection request is
       present, then accept it */
    sin_size = sizeof client_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&client_addr,
                    &sin_size);
    /* Convert the client address to a printable format, then
       print a message */
    char s[INET6_ADDRSTRLEN];
    inet_ntop(client_addr.ss_family, get_in_addr((struct
        sockaddr *)&client_addr), s, sizeof s);
    printf("server: got connection from %s\n", s);
    /* Set the client data */
    struct ClientInfo client_info;
    client_info.sockfd = new_fd;
    strcpy(client_info.alias, DEFAULTALIAS);
    /* Add the new client to the client list */
    pthread_mutex_lock(&clientlist_mutex);
    list_insert(&client_list, &client_info);
    pthread_mutex_unlock(&clientlist_mutex);
    /* Create a thread to handle the new client */
    pthread_create(&client_info.thread_ID, NULL, client_handler,
                  (void *)&client_info);
}

```



```
}
```

Ciclo di accettazione del server (privo di error handling)

Il server al suo avvio crea anche una lista vuota destinata a contenere i client connessi. Questa andrà aggiornata ad ogni login/logout.

Il problema dell'accesso multiplo a tale lista è stato risolto ricorrendo ad una variabile di lock definita nel file `pthread.h`. Questa può essere utilizzata da un thread per assicurarsi l'esclusivo accesso alla risorsa (è fondamentale tuttavia rilasciarne il possesso al termine dell'utilizzo).

Un esempio del suo utilizzo può essere visto nel codice qui riportato.

Il thread assegnato ad un singolo utente invece si occupa di riceverne i pacchetti e processarli come opportuno; un esempio è la gestione di un pacchetto SHOUT in arrivo da un client.

```
/* Send a message to every client connected */
case SHOUT :
    pthread_mutex_lock(&clientlist_mutex);
    for(curr = client_list.head; curr != NULL; curr =
        curr->next) {
        /* If the found client is the sender, keep searching */
        if(!compare(&curr->client_info, &client_info)) {
            continue;
        }
        /* Build a new packet containing the message and send it */
        struct Packet msgpacket;
        memset(&msgpacket, 0, sizeof(struct Packet));
        msgpacket.action = MSG;
        strcpy(msgpacket.alias, packet.alias);
        strcpy(msgpacket.payload, packet.payload);
        send(curr->client_info.sockfd, (void *)&msgpacket,
            sizeof(struct Packet), 0)
    }
    pthread_mutex_unlock(&clientlist_mutex);
    break;
```

Gestione di un pacchetto SHOUT (priva di error handling)

Il file di header contiene la definizione di alcune proprietà del programma server tra le quali l'indirizzo.

A differenza del client, dove l'indirizzo è inseribile come parametro al momento del login, è stato scelto di inserire hard-coded l'indirizzo del server in quanto è molto probabile che esso rimarrà invariato una volta compilato il programma.

4.3 CLIENTLIST.C (CLIENTLIST.H)

È l'implementazione di una linked list destinata a contenere nei nodi i dettagli delle connessioni con i vari client. Tale lista viene utilizzata esclusivamente nel programma server.

Ogni nodo contiene (oltre al riferimento al successivo), una struct di tipo ClientInfo definita nel file networkdef.h. Quest'ultima porta informazioni su thread ID e socket file descriptor associati alla connessione, e sull'alias del client in questione.

```
struct LLNode {
    struct ClientInfo client_info;
    struct LLNode *next;
};

struct LinkedList {
    struct LLNode *head, *tail;
    int size;
};

struct ClientInfo {
    pthread_t thread_ID;
    int sockfd;
    char alias[ALIASLEN];
};
```

Definizione di struct utilizzate nella implementazione della linked list

4.4 NETWORKUTIL.C (NETWORKUTIL.H)

È una libreria di metodi utili per l'elaborazione di indirizzi. Attualmente contiene solamente il metodo get_in_addr, realizzato per disporre di indirizzi con la formattazione appropriata.

```
void *get_in_addr(struct sockaddr *sa) {
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }
    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}
```

Metodo get_in_addr()

4.5 NETWORKDEF.H

File header che contiene parametri relativi al comportamento delle applicazioni e al protocollo utilizzato.

Tra le altre nozioni definite vi è quella di pacchetto e delle varie tipologie di pacchetto disponibili.

5 CONCLUSIONI E POSSIBILI SVILUPPI

Il programma rappresenta nella sua versione attuale un elementare ma completamente funzionante sistema di chat.

Sono qui proposte alcuni possibili sviluppi che il programma potrà percorrere in futuro. Le migliorie più importanti che possono essere apportate sono:

- L'introduzione di un ID univoco associato ad ogni client (nella versione attuale viene usato l'alias, ma se questo viene cambiato è difficile tener traccia di ogni utente).
- La modifica del sistema di assegnazione degli alias: attualmente sono concessi alias multipli, in futuro dovrà essere il server ad assegnare un alias al momento della connessione e ad evitare così doppioni.
- L'implementazione della possibilità per l'invio di messaggi più lunghi della dimensione massima di un pacchetto.
- L'introduzione di comandi lato server per il controllo diretto e la moderazione dei client.

Altre possibilità di sviluppo secondarie possono essere:

- L'introduzione di una semplice interfaccia grafica.
- Il porting del progetto su Windows sfruttando per la gestione dei socket la libreria windows.h.
- L'introduzione di molteplici chat room separate su uno stesso server.
- L'implementazione di privilegi speciali e limitazioni assegnabili ai client.
- L'introduzione della possibilità di registrare permanentemente un dato alias mediante password