UNIVERSITA' DEGLI STUDI DI NAPOLI
FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA

INSEGNAMENTO LABORATORIO DI SISTEMI OPERATVI

ANNO ACCADEMICO 2021/2022

DOCUMENTAZIONE

"RANDOM CHAT"

Gruppo LSO_2122_38

Autore:

Pio Francesco Falzarano N86002978

Sommario

1. Guida alla compilazione del server 3
2. Protocollo Applicativo 4
3. Server C 6
4. Client Android 10

1. Guida alla compilazione del server

Per ottenere il codice oggetto del server in C è necessario compilare due file presenti nella directory:

```
gcc -pthread -o server server.c connection.c
```

Una volta compilato, potremo lanciare l'esecuzione con il comando:

```
./server <port_number>
```

Specificando il numero di porta sulla quale il server ascolta. Per impostazione del client, si raccomanda di utilizzare il porto 5200.

2. Protocollo Applicativo

Una volta che il server è stato lanciato in esecuzione, può accettare connessioni da differenti client. Dopo una prima fase di inizializzazione, in cui il client invia al server il proprio nickname e il server lo inserisce nella coda dei client attivi, le due entità comunicano scambiandosi messaggi secondo un protocollo stabilito che si appoggia a TCP (livello trasporto).

Il client invia un comando al server secondo la tabella 1. Qualora il comando preveda degli argomenti, devono essere separati da uno spazio in modo tale che il server li processi correttamene.

COMANDO	ARGOMENTI	DESCRIZIONE
SRC	Numero della Room	Permette al client di cercare un endpoint nella stanza specificata
SND	Lunghezza del messaggio Messaggio	Permette al client di mandare un messaggio all'endpoint accoppiato
LVE		Permette al client di abbandonare una chat
QIT		Permette al client di interrompere una ricerca e quindi di abbandonare una stanza

Tabella 1: Richieste Client

Prendendo spunto da altri protocolli applicativi, il server invia al client messaggi specificati da uno status code, come è possibile osservare dalla tabella 2. A seconda del codice del messaggio, il client lo processa.

STATUS CODE	PAYLOAD	DESCRIZIONE
100	Client attivi in ogni Room	Informa il client sugli utenti attivi in ogni stanza
201	Nickname Endpoint	Informa il client che la ricerca di un endpoint in una stanza è terminata con successo
202	Messaggio	Informa il client che l'endpoint a lui associato gli ha inviato un messaggio
204		Informa il client che la ricerca in una stanza è terminata senza aver trovato un endpoint
300		Informa il client che l'endpoint associato ha abbandonato la chat
400		Errore nella richiesta del client

Tabella 2: Risposte Server

3. Server C

Per ogni nuova connessione, il server crea un *pthread* per gestire le richieste del client, che viene eliminato quando il client chiude la connessione. Ogni client viene memorizzato in una struttura dati (*linked-list*), il cui accesso e/o modifica dai vari pthread viene gestito attraverso un semaforo *mutex*, in modo tale che non ci siano race-condition sulle variabili globali, la lista degli utenti già menzionata e un array per tenere traccia dei client attivi in ogni stanza:

```
/******************
* Global Variables
*/
 struct Node *head = NULL;
 int rooms_size[ROOM_NUM] = {0};
 pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;
 pthread_mutex_t room_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Un esempio ne è la funzione di ricerca di endpoint in una stanza; il semaforo si è ritenuto indispensabile per una ricerca accurata, i pthread uno per volta accedono alla struttura dati per non creare accoppiamenti ambigui ed una volta effettuato il match, prima che il semaforo venga sbloccato vengono modificati i campi della struct Client delle due parti accoppiate affinché non risultino più disponibili per altri match fino alla chiusura della chat e anche per informare la controparte accoppiata che un altro client l'ha selezionato per il match e che può terminare la ricerca. La funzionalità di ricerca avviene sempre per mezzo di un altro pthread, in modo tale che il server possa rimanere in ascolto sulla socket del client per altre richieste, ad esempio se il client stesso vuole interrompere la ricerca. La ricerca ha una durata impostata di 60 secondi, dopo la quale si conclude automaticamente.

Di seguito sono riportate alcune righe della funzione **search_friend**:

```
do {


  pthread_mutex_lock(&clients_mutex); /* Blocco il mutex o rimango in attesa se bloccato */
  if(!c->is_searching) { /* Un altro endpoint ha accoppiato il client corrente */
    pthread_mutex_unlock(&clients_mutex);
    is_found = 1;
    goto end;
  }
  struct Node *ptr = head;
  while(ptr != NULL) {  /* scorro la lista e cerco nella stessa room utenti attivi */
   struct Client *curr = ptr->client;
   if (curr->is_searching && curr != c && curr->searching_room == c->searching_room &&
                                          c->last_friend != curr) {
       /* Vengono impostate le flag e i campi di accoppiamento
       del client e del matcher, omesse per brevità */
      break;
    }
    ptr = ptr->next;
  }
  pthread_mutex_unlock(&clients_mutex); /* sblocco il mutex */
  if(is_found) goto end;
  sleep(2);
  time(&end_time);
  search_time = difftime(end_time, start_time); /* calcolo il tempo di ricerca */


} while(search_time < 60 && !c->leave_flag); /* client abbandona la ricerca o tempo scaduto */
```

Dopo il match, i client si scambiano messaggi con il comando **SND**:

```
int n = read(client->sd, buffer_in, BUFF_SIZE); /* leggo dalla socket del client */
…
char command[4] = {buffer_in[0], buffer_in[1], buffer_in[2], '\0'}; /* estraggo il comando */
…
if(strcmp(command, SEND_MESSAGE_COMMAND) == 0) {
    /* EXPECTED: SND <len> <message> */
    if(strlen(buffer_in) < 5) { /* errore nella sintassi del comando */
      send_error_to(client);
      continue;
    }

    if(client->friend != NULL) {
      char *msg = parse_message(&buffer_in[4]);  /* estrae il messaggio a partire dalla length */
      strcpy(buffer_out, FRIEND_MESSAGE_CODE);
      strcat(buffer_out, msg);
      strcat(buffer_out, "\n");

      write((client->friend)->sd, buffer_out, strlen(buffer_out)); /* inoltro il messaggio */
      free(msg);
      continue;
    }

    strcpy(buffer_out, FRIEND_LEAVE_CODE); /* l'utente non ha una chat */
    strcat(buffer_out, "\n");
    write(client->sd, buffer_out, strlen(buffer_out));
  }
```

Questi sono solo alcuni degli esempi delle system call utilizzate; le funzioni di ricerca e di scambio di messaggi prese singolarmente potrebbero essere ostiche da comprendere, per questo si consiglia di leggere il sorgente (*connection.c*) per avere una visione più ampia e strutturata della gestione del protocollo impostato.

4. Client Android

Il client è l'entità che conosce l'indirizzo del server e instaura una connessione persistente. L'approccio utilizzato è stato costruire un'architettura *three-layer*, in cui al livello più basso troviamo i servizi che permettono la comunicazione col server, fatta per mezzo di una classe Singleton che inizializza la socket, ha il compito di scrivere sulla socket e resta in ascolto sulla stessa di risposte dal server (in dettaglio viene creato un thread ad-hoc per il compito con un Runnable custom).

In generale, tutte le operazioni che hanno a che fare con la Socket avvengono per mezzo di thread anonimi, come l'inizializzazione o l'invio di messaggi.

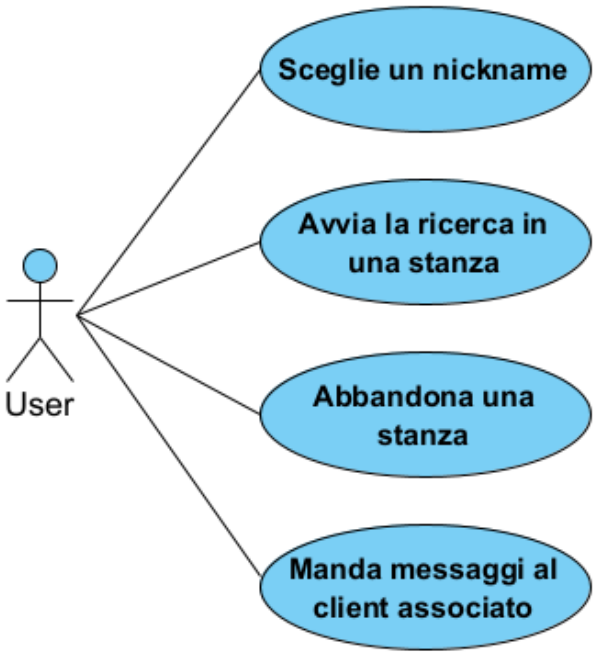

Figura 1: Use Case Diagram

Ogni azione dell'utente sull'interfaccia grafica, come la pressione di un bottone, innesca la chiamata di un metodo del *Controller*, il core della business-logic, che gestisce il caching delle entità e si interfaccia direttamente a *ClientSocket*, la classe che gestisce la comunicazione con il server.

Di seguito si può trovare un esempio di codice che inizializza la socket, del metodo **initSocket**.

```
new Thread(()-> {
    try {
        mSocket = new Socket(SERVER_ADR, SERVER_PORT); /* inizializzo la Socket */
        mSocket.setKeepAlive(true); /* flag per connessioni persistenti */
       /* inizializzo Writer e Reader per le operazioni di write/read sulla socket /*
        out = new PrintWriter(mSocket.getOutputStream(), true);
        in = new BufferedReader(new InputStreamReader(mSocket.getInputStream()));
        new Thread(new SocketListener()).start(); /* creo un thread che resta in ascolto /*
    }catch (IOException e) {
        if(mSocket != null) {
            closeConnection();
        }
    }
 }).start();
```

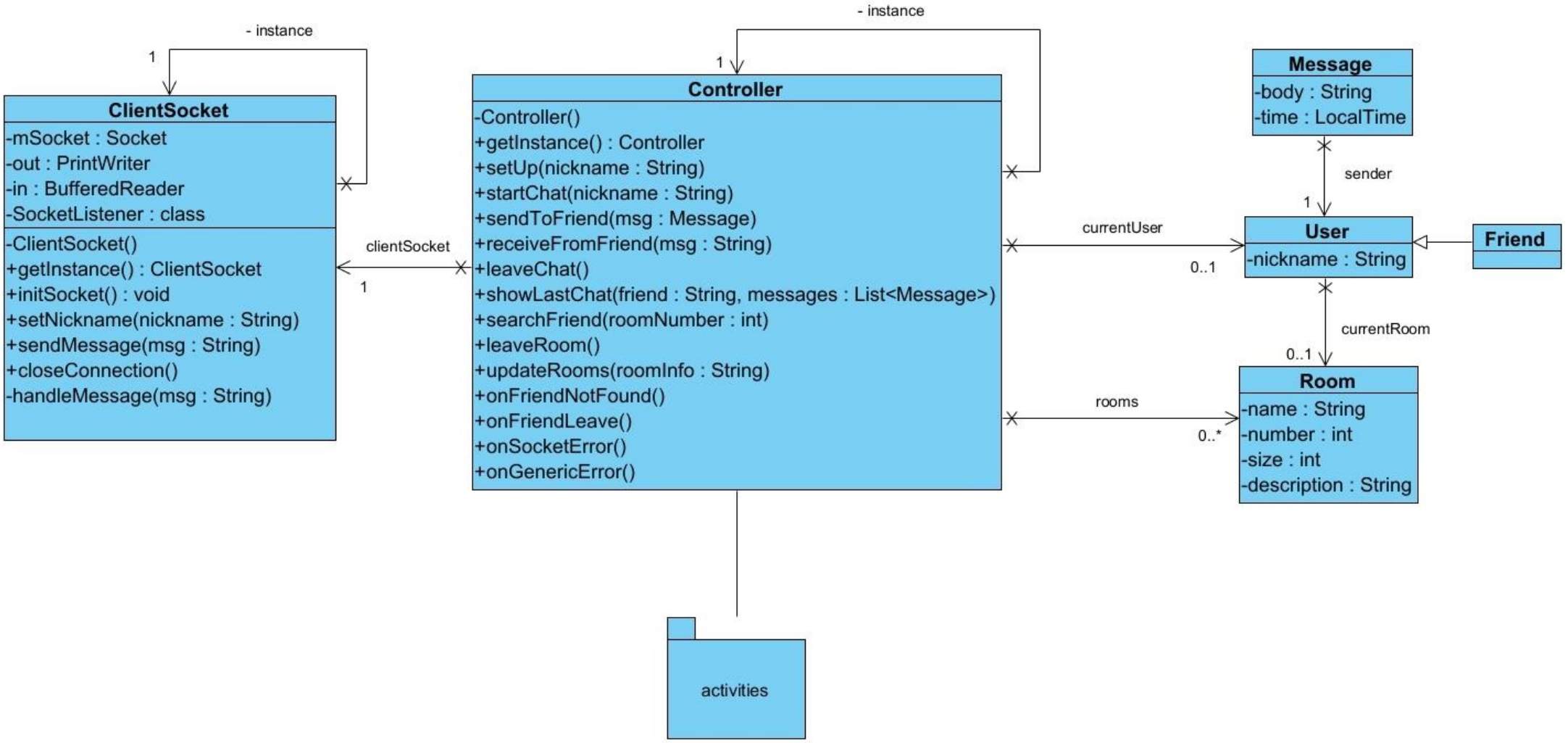

Figura 2: Class Diagramm Client Android