UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

FACOLTA' DI INFORMATICA

Corso di Laurea Triennale

Dipartimento di ingegneria elettrica
e delle tecnologie di informazione

Documentazione Progetto Laboratorio di Sistemi Operativi

RandomChat LSO21/22_N_39

Relatori:
Marco Grazioso
Francesco Cutugno
Giovanni Scala

Progettisti:
Francesco De Stasio - N86003294

ANNO ACCADEMICO 2021/2022

Indice

Capitolo 1

Introduzione

La documentazione presentata nei capitoli successivi, tratta di specifiche tecniche e guide all'utilizzo del prodotto software *RandomChat*, di cui committenti: Francesco Cutugno, Marco Grazioso, Giovanni Scala, docenti della Federico II di Napoli, i quali assegnano la progettazione e l'implementazione del prodotto, al gruppo LSO_2122_39.

Inizialmente verranno presentate le idee progettuali e i vari diagrammi effettuati durante la fase di progettazione, successivamente verrà illustrata una guida alla compilazione sia del client che del server, procedendo quindi anche a mostrare l'utilizzo pratico di questi ultimi, come si collegano e il funzionamento effettivo dell'app, e infine alcuni dettagli implementativi, quelli più interessanti e caratteristici.

La cura e revisione è stata fatta da **Francesco De Stasio**.

1.0.1 Requisiti

Requisiti funzionali

I requisiti funzionali richiesti sono elencati di seguito:

- Permettere all'utente di sapere quanti clients sono connessi in ogni stanza.
- Permettere all'utente di mettersi in attesa di una chat in determinata stanza.
- Una volta stabilito il match permettere all'utente di scambiare messaggi con l'utente assegnato e di chiudere la conversazione in qualsiasi momento.
- Non essere assegnato ad una medesima controparte nella stessa stanza in due assegnazioni consecutive.
- Permettere l'invio di messaggi tramite i servizi di speech recognition del client android.

Requisiti non funzionali

Il server va realizzato in linguaggio C su piattaforma UNIX/Linux e deve essere ospitato online su Microsoft Azure. Il client va realizzato in linguaggio Java su piattaforma Android e fa utilizzo dei servizi di speech recognition. Client e server devono comunicare tramite socket TCP o UDP. Oltre alle system call UNIX, il server può utilizzare solo la libreria standard del C. Il server deve essere di tipo concorrente, ed in grado di gestire un numero arbitrario di client contemporaneamente. Il server effettua il log delle principali operazioni (nuove connessioni, sconnessioni, richieste da parte dei client) su standard output.

Capitolo 2

Progettazione

Di seguito verranno presentate le varie fasi di progettazione in ordine cronologico, così da avere un'idea più chiara dell'andamento degli eventi.

2.1 Analisi dei Requisiti

In una fase iniziale vengono studiati a fondo i requisiti richiesti dai committenti, si svolge una prima fase di setUp, quale l'istanziazione del public cloud **Azure**, il quale farà da host del server, e successivamente si procede a estrapolare informazioni dai requisiti dati, per poter capire a pieno il lavoro da svolgere.

2.2 BrainStorming

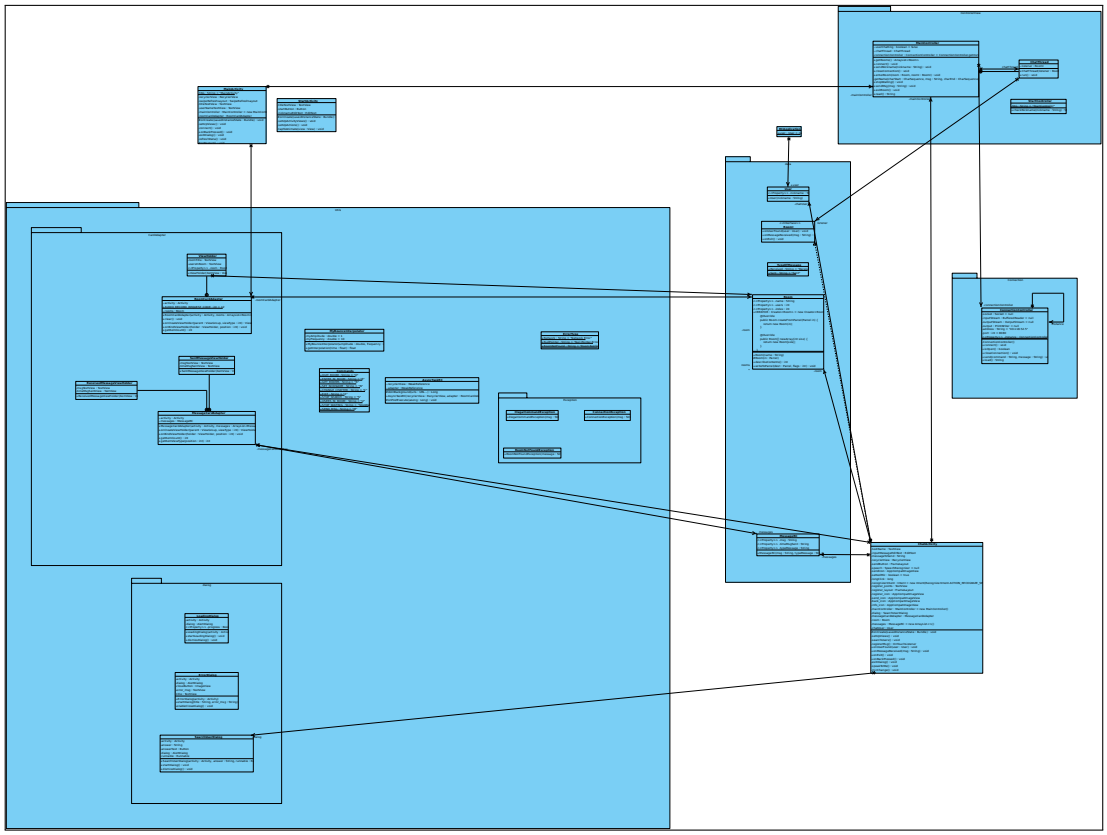
Vengono esposte varie idee sul come procedere per la realizzazione del progetto in modo efficiente e corretto. Di seguito **alcune** delle idee proposte durante questa fase:

- Creare strutture dati per potere salvare in modo efficiente i dati
- Utilizzo di id per poter avere una chiave per ogni utente
- Creazione di un thread per ogni utente connesso
- Ottimizzare lo scambio dei messaggi tramite un protocollo efficiente
- Creazione di file con nomi delle stanze, file per ogni stanza contente gli utenti connessi (*abolita*

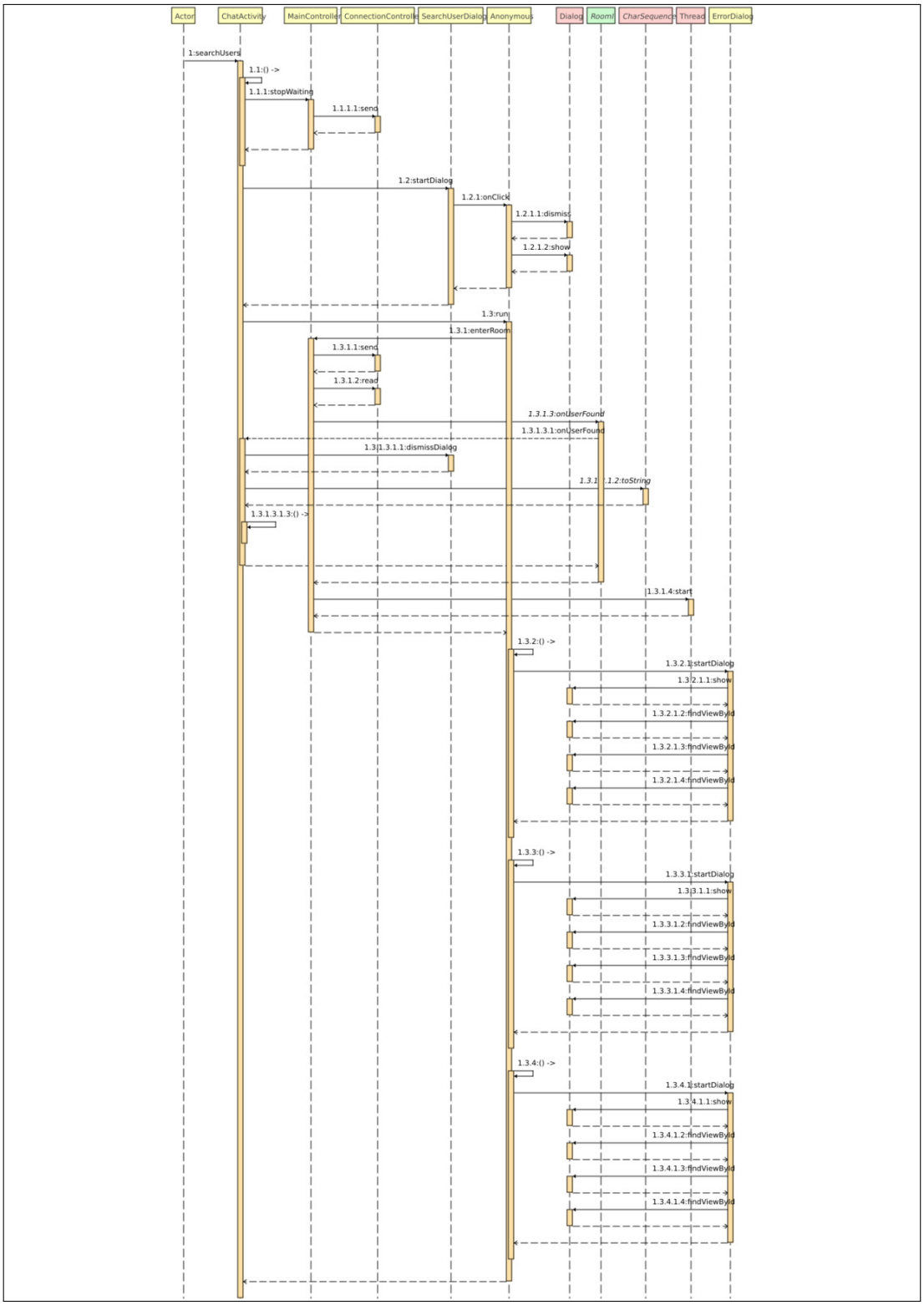
2.3 Grafici

Verranno mostrati di seguito dei diagrammi progettuali del progetto, o di alcune sue *funzionalità*.

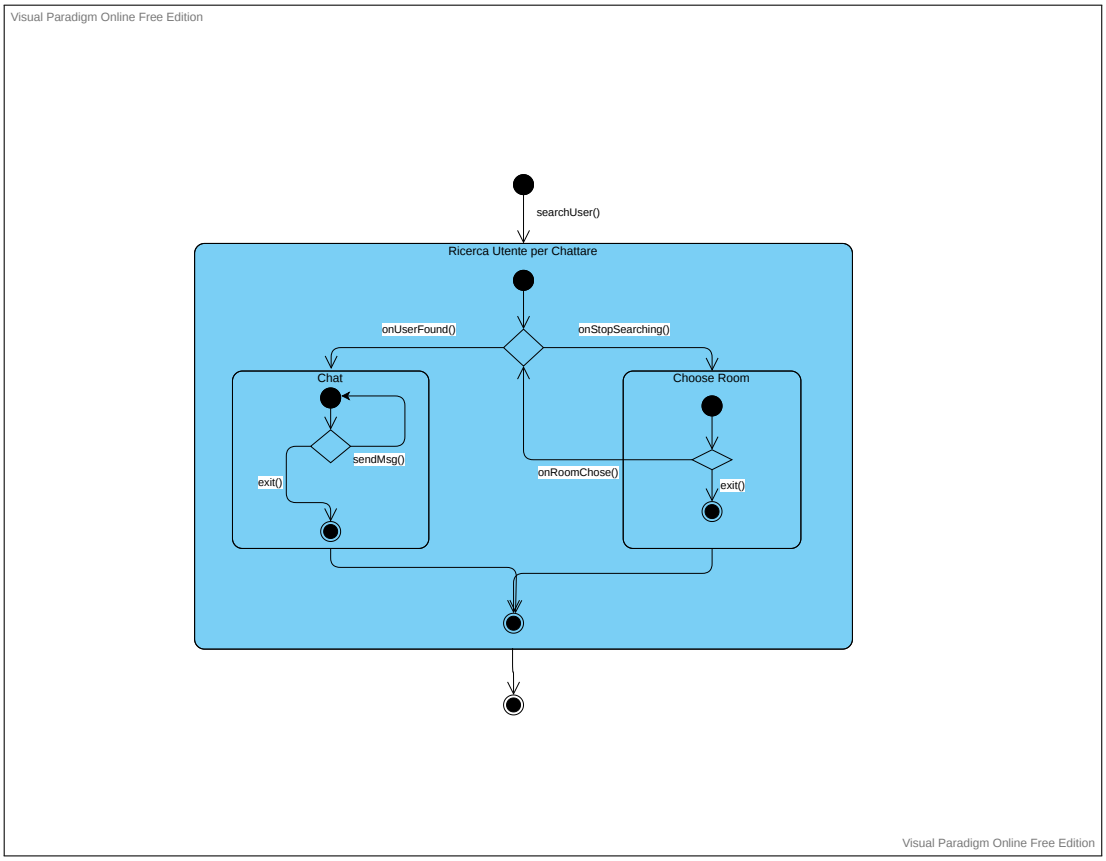
2.3.1 Class Diagram Client



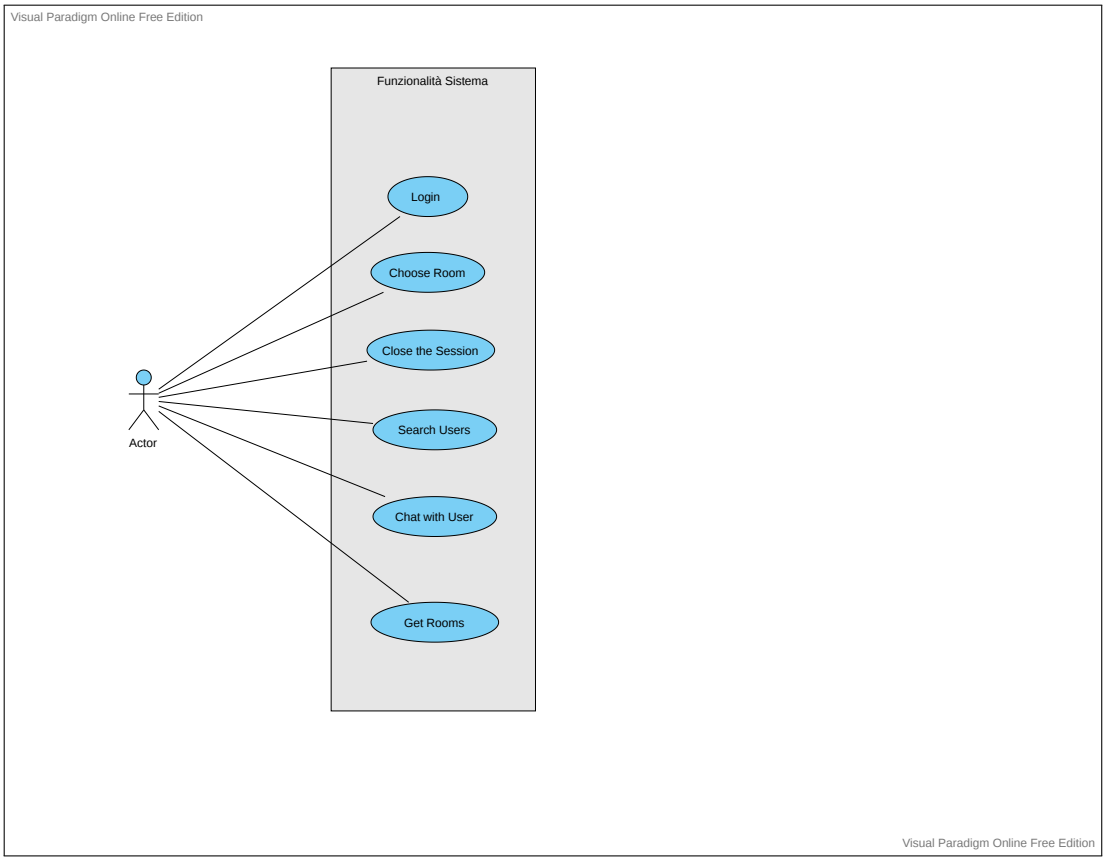
2.3.2 Sequence Diagram Search User



2.3.3 State Chart Search User



2.3.4 UseCase


Visual Paradigm Online Free Edition
Funzionalità Sistema
Login
Choose Room
Close the Session
Search Users
Chat with User
Get Rooms
Actor
Visual Paradigm Online Free Edition

2.4 Idee Progettuali e applicazione

Si decide quindi di iniziare il progetto dalla parte ***server***, tralasciando inizialmente il client Android. Per testare lo scambio di messaggi, si decide invece di utilizzare il programma **telnet** e di utilizzare come area di implementazione la WSL di Windows con connessione locale. Il protocollo per lo scambio dei messaggi definisce il primo carattere ricevuto come **Command** (Comando) e i successivi come **Message** (Messaggio), in caso di comando sconosciuto o non valido, il server chiude la connessione con il client. I comandi utilizzati sono tutti char, tranne per alcune eccezioni, e sono mostrati di seguito:

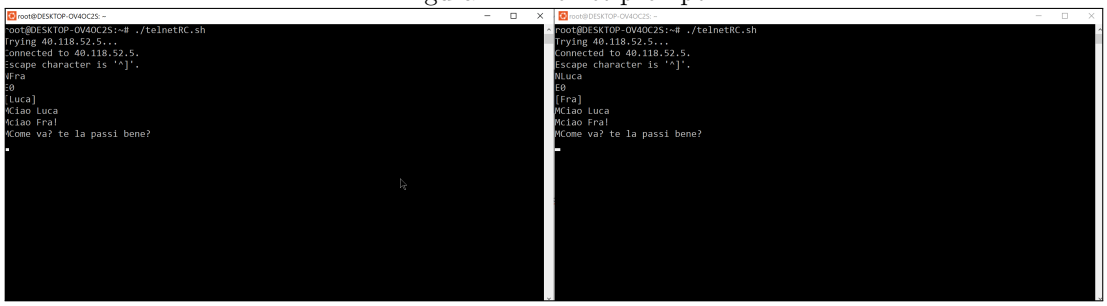
COMMANDS

1. ENTER_ROOM = 69 // 'E'
2. NICKNAME = 78 // 'N'
3. EXIT = 88 // 'X'
4. MSG = 77 // 'M'
5. PUSH_PUSH_LIST // 'R'
6. STOP_SEARCH = 83 // 'S'
7. QUIT_ROOM = 81 // 'Q'

Il messaggio inviato può avere grandezza massima ***400 byte***. Le stanze verranno invece create tramite uno scanner che leggerà da un file di testo i titoli delle stanze da creare, separate da un separatore di righe (in ASCII TABLE = 10) e passate a un vettore creato appositamente. Verranno utilizzate più strutture dati o struct, per la corretta implementazione del software da parte del server, quali *Queue*, *RoomVector*, *User*, *Link*, ognuna delle quali svolge un diverso compito. Si decide di utilizzare una connessione **socket TCP**, utilizzando le varie funzioni della libreria standard del C, e l'utilizzo anche di segnali per la comunicazione fra diversi thread; la decisione di utilizzare **segnali** è stata presa in merito alla loro facilità di utilizzo e la buona utilità che possono dare se utilizzati correttamente.

Ritornando invece alla parte ***client*** si decide invece di suddividere l'applicazione in due Activity principali, l'Activity denominata Main, in cui si effettua la scelta delle stanza, e nella quale è possibile visionare il numero di utenti in ogni stanza; e l'Activity denominata Chat in cui avviene il vero e proprio scambio di messaggi. Si pensa di utilizzare anche un'Activity precedenti a quella principale (Main) in cui sarà possibile "*catturare*" (catch) il Nickname dell'utente, così da poter effettuare vari controlli sul testo e confermare la sua correttezza, per poi passarlo alle Activity successive, dando inizio alla RandomChat!

Figura 2.1: Telnet prompt



Capitolo 3

Guida alla compilazione

3.1 Compile Server

Dopo aver scaricato il pacchetto server, esso dovrà compilare più file *.c, è possibile farlo attraverso il compilatore ***gcc*** tramite il comando:

```
1    gcc -pthread server.c structures/searcher.c structures/link.c structures/
     roomvector.c structures/room.c structures/queue.c setUpRandomChat.c
     auxServer.c -o server
```

Listing 3.1: Command for compile server

Il comando visto precedentemente creerà un file denominato **server**, la quale esecuzione porterà all'avvio del server. E' possibile farlo tramite la normale esecuzione di programmi in bash.

```
1    ./server
```

Il pacchetto server contiene anche uno script **run** che eseguirà i comandi sopra citati.

```
1    ./run
```

Il server è stato testato in un ambiente Linux, distribuzione Ubuntu, con una macchina virtuale hostata dal public cloud Azure.

3.2 Compile Client

Dopo aver scaricato l'applicazione su un dispositivo **Android** sarà possibile poter far partire l'esecuzione con un click sull'app, dopodichè sarà richiesto il permesso di accesso al microfono, che servirà per poter utilizzare l'app a pieno.

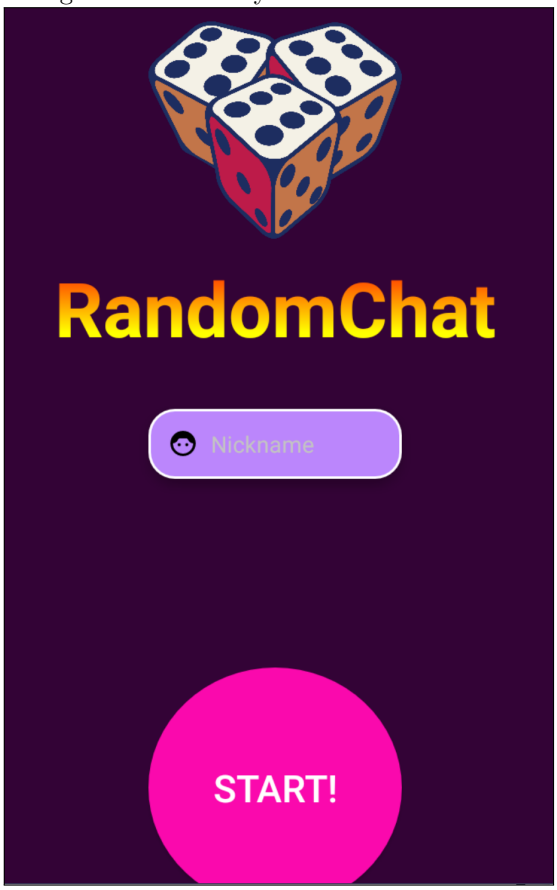
Capitolo 4

Guida all'uso

4.1 Client Android

L'applicativo Android ha una prima schermata iniziale nella quale è possibile inserire un nickname, un nome identificativo per l'intera durata della sessione all'interno dell'app.

Figura 4.1: Activity iniziale RandomChat

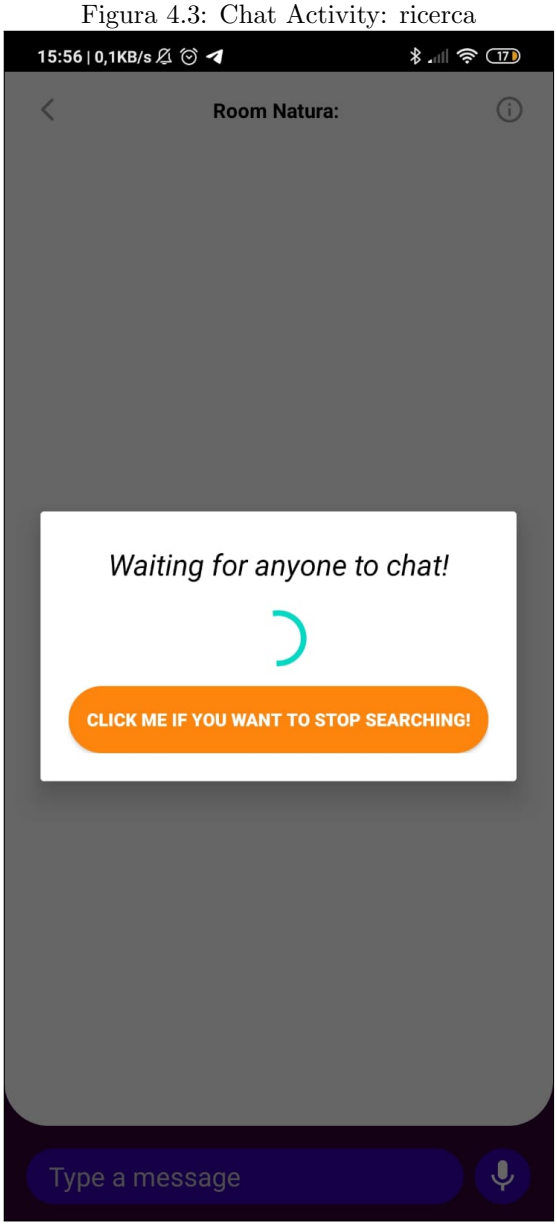


Successivamente, dopo aver inserito un nickname valido, si accede direttamente alla Room Activity, dove sarà possibile scegliere fra una delle stanze tematiche e poter vedere quanti utenti sono connesse a tali stanze.

Figura 4.2: Room Activity

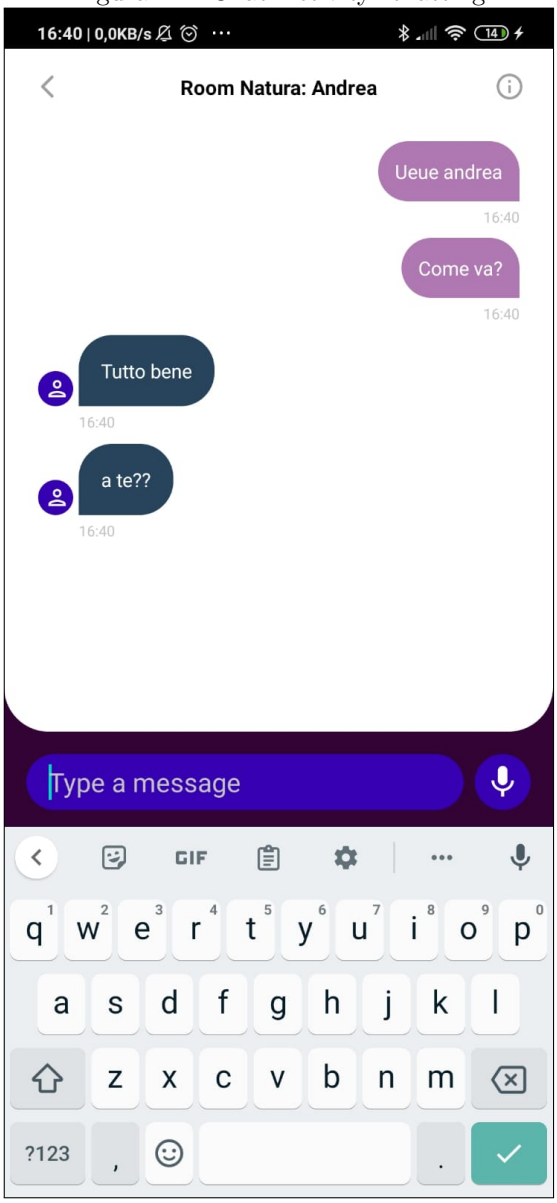
Una volta scelta una stanza, si ricerca un altro utente con cui chattare.

Figura 4.3: Chat Activity: ricerca



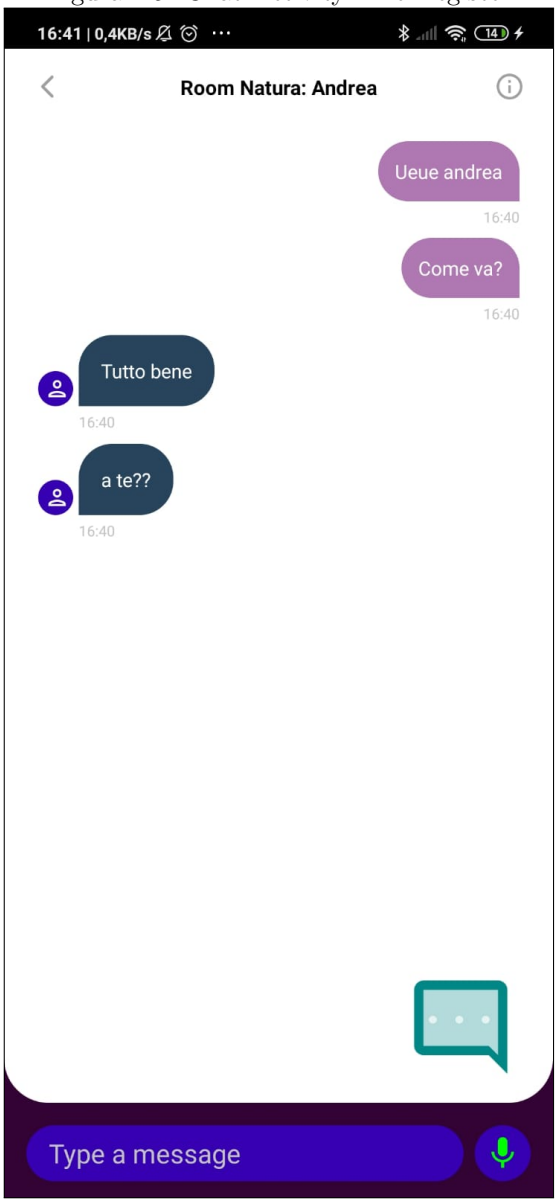
Quando viene matchato un altro utente inizia la chat vera e propria, in cui sarà possibile inviare messaggi di testo vicendevolmente.

Figura 4.4: Chat Activity: chatting



Sarà possibile inviare messaggi vocali tramite un *long click* sull'icona del microfono, al rilascio inizierà a registrare, fin quando non verrà stoppato da un altro click dell'utente.

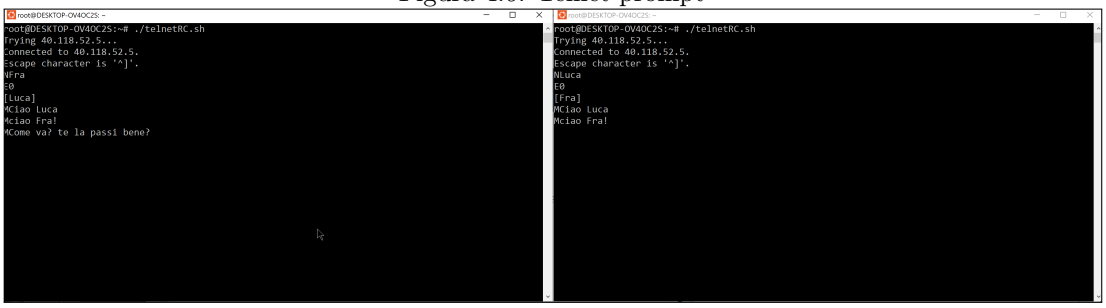
Figura 4.5: Chat Activity: mic. register



4.2 Client prompt

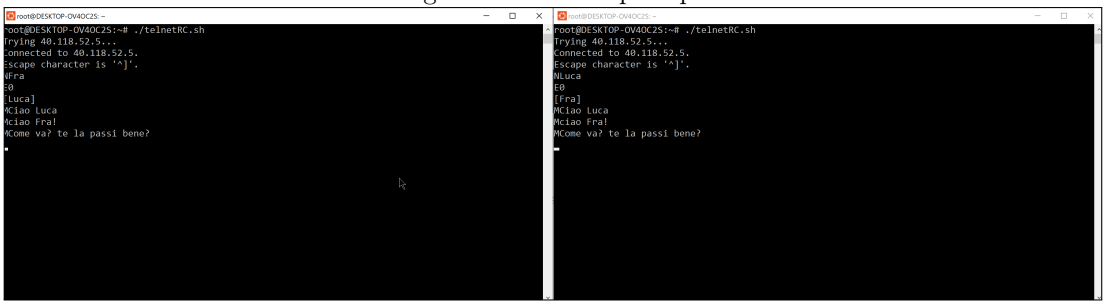
E' possibile anche utilizzare telnet o netcat come client, collegandosi al corretto Ip con la medesima porta, è possibile chattare attraverso l'uso dei comandi: 2.4, inseriti come primo carattere, e successivamente il messaggio da inviare. Un esempio d'uso:

Figura 4.6: Telnet prompt



Cliccando ***ENTER*** sulla tastiera per inviare la stringa.

Figura 4.7: Telnet prompt



Capitolo 5

Dettagli implementativi

Maggiore attenzione può essere data alla decisione di utilizzare alcune fra i principali strumenti informatici , quali ad esempio l'utilizzo dei **Thread** 5, il quale principale uso è stato adattato per servire ogni utente in base alle sue richieste, l'utilizzo dei **semafori** (mutex) 5.2 per poter gestire le parti critiche del codice in cui c'era bisogno di un ordine di scrittura, i **Segnali** 5.3 della libreria standard *signal.h* che hanno permesso di capire quando un thread voleva mettersi in comunicazione con l'altro, e infine le funzioni principali delle **Socket** 5.4, read/recv e write/send, le quali hanno permesso il vero e proprio scambio di messaggi e lettura di questi ultimi.

Soffermandoci infine sulla struttura di base di cattura dei messaggi 5.5, che in combinazione con gli strumenti sopra elencati, ha reso una facile e corretta implementazione del server, con un sistema di *Log* semplice ed efficace.

5.1 Thread

```
1       printf("Creating the thread for the new client..\n");
2         User* user = malloc(sizeof(User));
3         user->id = ++idClient;
4         user->socketfd = client;
5         if(pthread_create(&user->tid, NULL, clientManagerFun, user) == -1){
6             perror("!Error creating the thread:");
7             exit(EXIT_FAILURE);
8         };
9         if(pthread_detach(user->tid) == -1){
10            perror("!Error detaching the thread:");
11            exit(EXIT_FAILURE);
12        };
13        printf("Done!\nStarting the thread..");
```

5.2 Mutex

```
    ...
    room->mutex = (pthread_mutex_t) PTHREAD_MUTEX_INITIALIZER;
    ...
    pthread_mutex_lock(&room->mutex);
    enqueue(&room->users, user); // critic code
    pthread_mutex_unlock(&room->mutex);
```

5.3 Signal

```
void signal_handler(int signal){
    printf("[Signal] Signal handled by thread: %d\n", gettid());
    return;
}
...
 __sighandler_t  old = signal(SIGUSR1, signal_handler);
    fdcount = select(user->socketfd + 1, &rd, NULL, NULL, NULL);
```

5.4 Socket

```
        memset(buff,'\0',BUFF_LEN);
        msglen = read(user->socketfd, buff, BUFF_LEN);
        if(msglen <= 0){
            printf("[Client: %d] Connection closed by client\n", user->id);
            break;
        }
        buff[msglen] = '\0';
        printf("[Client: %d] Ricevuto: %s\n", user->id, buff);
        ...
        memset(buff,'\0',BUFF_LEN);
        sprintf(buff, "[%s]\n",user1->nickname);
        if(send(user2->socketfd, buff, strlen(buff), MSG_NOSIGNAL) < 0){
        perror("[!] Error starting the chat..\n");
        };
```

5.5 Command catch

```
int cmdCatch(RoomVector* roomVector, User* user,int cmd,  char* msg){
    switch(cmd){
        case ENTER_ROOM:
            enterInRoom(user, roomVector, atoi(msg), msg);
            break;
        case NICKNAME:
            setNickname(user, msg);
            break;
        case EXIT:
            return 0;
        case PUSH_ROOM_LIST:
            sendRoomList(user, roomVector, msg);
            break;
        default:
            printf("Command not found: %d\n",cmd);
            return 0;
    }
    return 1;
}
   ...
   do{
    memset(buff,'\0',BUFF_LEN);
    msglen = read(user->socketfd, buff, BUFF_LEN);
    if(msglen <= 0){
        printf("[Client: %d] Connection closed by client\n", user->id);
        break;
    }
    buff[msglen] = '\0';
    printf("[Client: %d] Ricevuto: %s\n", user->id, buff);

}while(cmdCatch(roomVector, user, buff[0], buff+1));
```

Capitolo 6

Client Classes

In ***Android*** è possibile sfruttare le funzioni della libreria java.net.Socket, java.io.InputStreamReader, java.io.OutputStream per creare una connessione Socket TCP, le quali sono state strumento della classe **ConnectionController**. Quest'ultima sfrutta il Pattern ***Singleton*** per creare un'unica connessione con il server, permettendo di inviare e leggere messaggi alfanumerici attraverso le sue funzioni 6.1.

6.1 Connection Controller

```
1  public class ConnectionController {
2
3      private static ConnectionController instance = null;
4      private Socket socket = null;
5      private BufferedReader inputStream = null;
6      private OutputStream outputStream = null;
7      private PrintWriter output = null;
8      private final String address = ""; // lasciato vuoto per sicurezza
9      private int port = ; // lasciato vuoto per sicurezza
10     private ConnectionController(){
11
12     }
13
14     public static ConnectionController getInstance(){
15         if(instance == null){
16             synchronized (ConnectionController.class){
17                 if(instance == null){
18                     instance = new ConnectionController();
19                 }
20             }
21         }
22         return instance;
23     }
24
```

```
25      public void connect() throws IOException{
26          socket = new Socket(address, port);
27          inputStream = new BufferedReader(new InputStreamReader(socket.
        getInputStream()));
28          outputStream = socket.getOutputStream();
29          output = new PrintWriter(outputStream);
30      }
31
32
33      public boolean isOpen(){
34          if(socket != null){
35              if(!socket.isClosed()){
36                  return true;
37              }
38          }
39          return false;
40      }
41
42      public void closeConnection() throws IOException{
43          if(isOpen()) {
44              output.close();
45              outputStream.close();
46              socket.close();
47              instance = null;
48          }
49      }
50
51      public void send(String command, String message){
52          synchronized (output){
53              output.write((command + message));
54              output.flush();
55          }
56      }
57
58      public String read() throws IOException{
59          synchronized (inputStream){
60              String msg = inputStream.readLine();
61              if(msg != null){
62                  return msg;
63              }
64          }
65          return null;
66      }
67  }
```

Queste a loro volta vengono richiamate dal **MainController** 6.2 che usufruisce di queste funzioni per il corretto invio e la corretta lettura dei messaggi, che devono essere coerenti con i comandi conosciuti dal server. Il MainController gestisce quindi l'invio del Nickname, la creazione di un Thread per leggere dalla socket i messaggi scambiati con un altro client,

la richiesta delle Stanze con i relativi utenti, l'apertura della socket con il Server,l'ingresso e l'uscita da una specifica stanza e infine la Chat con un altro client.

6.2 Main Controller

```
1  public class MainController {
2
3      ChatThread chatThread;
4      private ConnectionController connectionController = ConnectionController.
       getInstance();
5      boolean userChatting = false;
6
7      public ArrayList<Room> getRooms() throws IOException{
8          ArrayList<Room> rooms = null;
9          if(connectionController != null && connectionController.isOpen()){
10             connectionController.send(Commands.GET_ROOMS, "");
11             int i = Integer.parseInt(connectionController.read());
12             rooms = new ArrayList<>();
13             for(int j = 0; j < i; j++){
14                 String roomStr = connectionController.read();
15                 Scanner scanner = new Scanner(roomStr);
16                 String roomName = scanner.next();
17                 int userCount = scanner.nextInt();
18                 Room room = new Room(roomName);
19                 room.setUsers(userCount);
20                 room.setIndex(j);
21                 rooms.add(room);
22             }
23         }
24         return rooms;
25     }
26
27     public void connect() throws IOException{
28         connectionController.connect();
29     }
30
31     public void sendNickname(String nickname) throws ConnectionException {
32         if(connectionController != null && connectionController.isOpen()){
33             if(!nickname.isEmpty()){
34                 nickname = nickname.trim();
35                 nickname = nickname + '\0';
36                 connectionController.send(Commands.SET_NICKNAME, nickname);
37             }
38         }else{
39             throw new ConnectionException("Connection not open");
40         }
41     }
42
```

```
43      public void closeConnection() throws IOException{
44          connectionController.closeConnection();
45      }
46
47      public void enterRoom(Room room, RoomI roomI) throws IOException,
    RoomNotFoundException, ConnectionException {
48          if(connectionController != null && connectionController.isOpen()){
49              connectionController.send(Commands.ENTER_IN_ROOM, String.valueOf(
    room.getIndex()));
50          }else{
51              throw new ConnectionException("Connection not open");
52          }
53
54          do{
55              String msg = connectionController.read();
56              if(msg != null){
57                  msg = msg.replace('\n', '\0');
58                  switch(String.valueOf(msg.charAt(0))){
59                      case Commands.STOP_SEARCH:
60                          return;
61                      case Commands.EXIT:
62                          throw new RoomNotFoundException("Room not found, index
     error!");
63                      default:
64                          roomI.onUserFound(new User(getName("[",msg, "]")));
65                          chatThread = new ChatThread(roomI);
66                          chatThread.start();
67                          userChatting = true;
68                          return;
69                  }
70              }
71          }while(true);
72      }
73
74      private String getName(CharSequence charStart, String msg, CharSequence
    charEnd){
75          String ret = null;
76          if(msg.contains(charStart) && msg.contains(charEnd)){
77              if(msg.indexOf(String.valueOf(charStart)) < msg.indexOf(String.
    valueOf(charEnd))){
78                  ret = msg.substring(msg.indexOf(String.valueOf(charStart))+1,
    msg.indexOf(String.valueOf(charEnd)));
79              }
80          }
81          return ret;
82      }
83
84      public void stopWaiting(){
85          if(connectionController != null && connectionController.isOpen()){
```

```
86              connectionController.send(Commands.STOP_WAITING,"");
87          }
88      }
89
90      public void sendMsg(String msg){
91          if(connectionController != null && connectionController.isOpen()){
92              connectionController.send(Commands.SEND_MSG, msg+'\n');
93          }
94      }
95
96      public void exitRoom(){
97          if(!userChatting){
98              if(connectionController != null && connectionController.isOpen()){
99                  connectionController.send(Commands.EXIT, "\n");
100             }
101         }else {
102             userChatting = false;
103             connectionController.send(Commands.EXIT, "\n");
104             try {
105                 chatThread.join();
106             }catch (InterruptedException e){
107                 e.printStackTrace();
108             }
109
110         }
111
112     }
113
114     public String read() throws IOException {
115         String ret = null;
116         if(connectionController != null && connectionController.isOpen()){
117             ret = connectionController.read();
118         }
119         return ret;
120     }
121
122
123
124     private class  ChatThread extends Thread{
125         RoomI listener;
126
127         public ChatThread(RoomI listener){
128             this.listener = listener;
129         }
130
131         @Override
132         public void run() {
133             String msgReceived = " ";
134             while(!String.valueOf(msgReceived.charAt(0)).equals(Commands.EXIT)
```

```
     || userChatting){
135                 try {
136                     msgReceived = read();
137                     System.out.println(msgReceived);
138                     String command = String.valueOf(msgReceived.charAt(0));
139                     if(command.equals(Commands.EXIT)){
140                         if(!userChatting){
141                             return;
142                         }else{
143                             listener.onExit();
144                         }
145                     }else if(command.equals(Commands.SEND_MSG)){
146                         listener.onMessageReceived(msgReceived.substring(1));
    //per togliere M
147                     }else{
148                         throw new IllegalCommandException("Unkown command: " +
     msgReceived.charAt(0));
149                     }
150                 } catch (IOException | IllegalCommandException exception) {
151                     exception.printStackTrace();
152                 }
153             }
154         }
155     }
156
157 }
```

Capitolo 7

Conclusione

In conclusione si vuole chiarire lo scopo del progetto RandomChat, cosicchè da poter definire aspetti non ancora chiari. Il progetto **RandomChat** vuole dare la possibilità di poter comunicare in modo rapido e veloce riguardo una specifica tematica, con persone con gli stessi interessi, avere la possibilità di poter chiacchierare con chiunque nel mondo e poter scambiare opinioni e idee, direttamente da casa.

Per dare uno sguardo al codice e ai vari tempi impiegati per il progetto, il link di **Github** è il seguente: RandomChat, per accedervi, inviare una mail all'indirizzo "france.destasio@studenti.unina.it" con oggetto: "Richiesta di Accesso RC Github".

Riportiamo di seguito alcuni ***concetti chiave*** del progetto e della medesima documentazione.[1]

- Java
- Android
- Socket
- Pthread
- Pthread Mutex
- Signal
- Singleton
- TCP Connection
- Github

[1] E' possibile chiedere informazioni all'indirizzo france.destasio@studenti.unina.it per eventuali dubbi e chiarimenti riguardo il progetto RandomChat, le mail devono avere come Oggetto: LSO_2021/2022.