UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Corso di Laurea Triennale in Informatica

PROGETTO DI LABORATORIO DI SISTEMI OPERATIVI

RANDOM CHAT by RMC Inc. Gruppo LSO_2122_34

Docenti:

Prof. Giovanni Scala

Prof. Francesco Cutugno

Studenti:

Matteo Richard Gaudino N86003226

Emanuele La Daga N86002987

Claudio Velotti N86002882

ANNO ACCADEMICO 2021/2022

Indice

Capitolo 1

Introduzione

Sommario

1.1 Presentazione del problema

RandomChat è un app android per chattare con sconosciuti in stanze tematiche.
Di seguito la lista dei requisiti funzionali dell'applicazione:

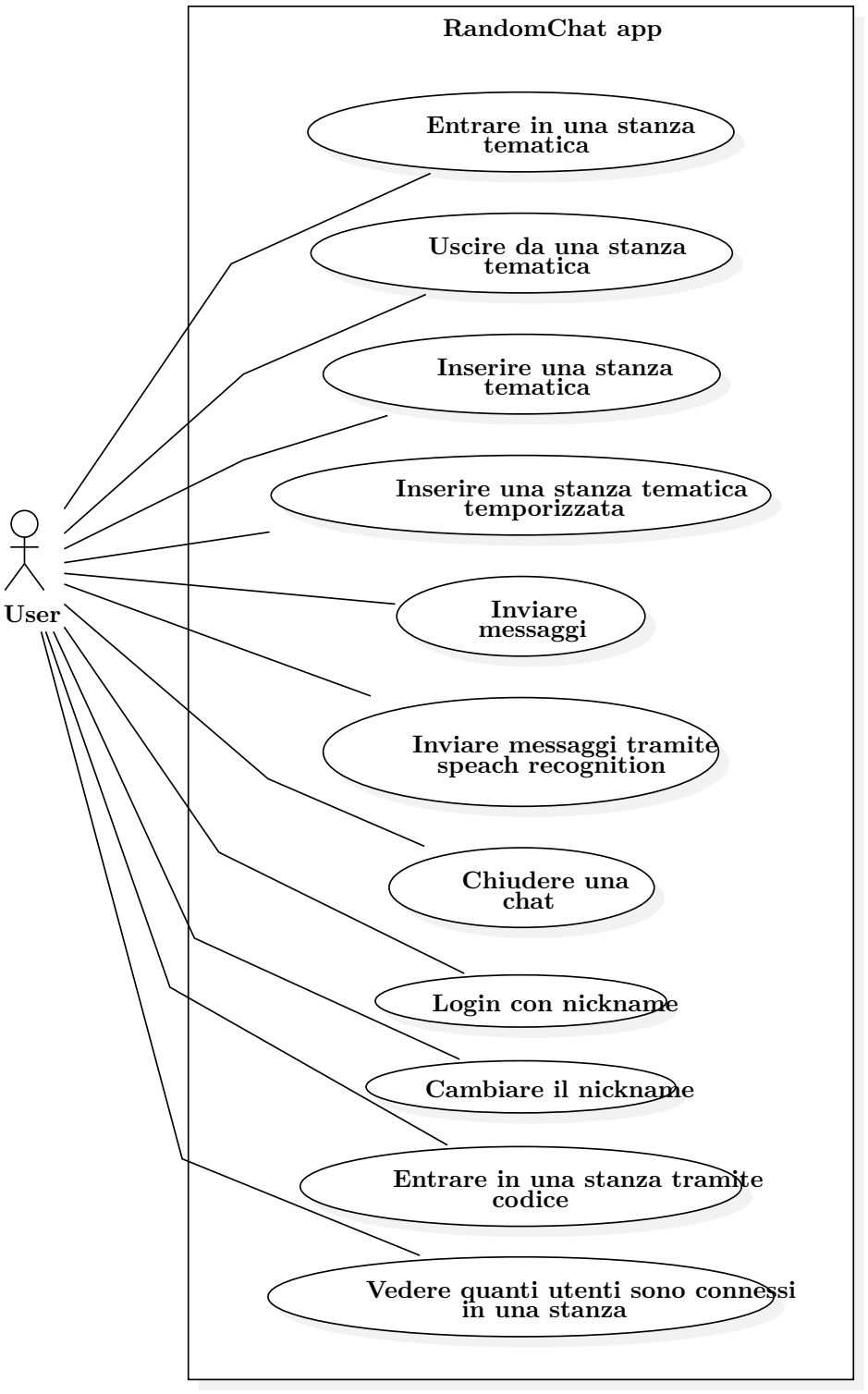
1. Il sistema deve gestire una random chat in cui i client si collegano ad una stanza tematica e vengono messi in contatto con altri client in maniera random. I client una volta accoppiati, possono scambiarsi messaggi testuali fino alla chiusura da parte di una delle parti della chat.
2. Ciascun client è identificato pubblicamente da un nickname, scelto dall'utente.
3. Permettere all'utente di sapere quanti client sono connessi in ogni stanza.
4. Permettere all'utente di mettersi in attesa di una chat in determinata stanza.
5. Una volta stabilito il match permettere all'utente di scambiare messaggi con l'utente assegnato e di chiudere la conversazione in qualsiasi momento.
6. Non essere assegnato ad una medesima controparte nella stessa stanza in due assegnazioni consecutive.
7. Permettere l'invio di messaggi tramite i servizi di speech recognition del client android.
8. Prevedere un tempo massimo di durata di una conversazione dopo il quale si viene assegnati ad un'altra chat.

Di seguito la lista dei requisiti non funzionali dell'applicazione:

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

1.2 Casi d'uso

La lista dei requisiti funzionali si può riassumere nel seguente diagramma dei casi d'uso:



Capitolo 2

Server

Sommario

2.1 Compilazione

Il server utilizza CMake per l'automazione dello sviluppo. Per compilare e installare il programma RandomChatServer lanciare in ordine i seguenti comandi:

$ *cmake make .*

$ *make*

$ *make install*

Con la prima riga cmake produce i makefile per la compilazione del programma. Con la seconda riga make compila il programma producendo il file eseguibile *RandomChatServer* nella cartella *src*. La terza riga è opzionale, nel caso venga eseguita make copierà l'eseguibile nella cartella */usr/local/bin*. Per cambiare la cartella di installazione va cambiato il percorso nel file *CMakeList*:

$$install(TARGETS\ RandomChatServer\ DESTINATION\ < newPath >)$$

Una volta eseguito il server si metterà in ascolto sulla porta 8125. Per cambiare porta va cambiata la costante *PORT* in *src/server.h* e vanno rieseguite le istruzioni di compilazione:

$$\#define\ PORT\ 8125 \Rightarrow \#define\ PORT\ < newPort >$$

2.2 Esecuzione

Per eseguire il programma lanciare il comando:

$$\$\ RandomChatServer$$

Una volta avviato, il server cercherà nella cartella di esecuzione il file *rooms.rc* contenenti le stanze tematiche da caricare. *rooms.rc* è un file di testo, ogni riga corrisponde ad una stanza formattata come segue:

$$< id > < roomColor > < time > \ [< RoomName >]$$

Nella cartella principale del progetto è presente un file *rooms.rc* contenente delle stanze di esempio. Il server sovrascriverà il file ogni 5 minuti aggiungendo le nuove stanze. Il file delle stanze è opzionale e nel caso non venga trovato verrà creato dal server quando salverà le stanze aggiunte.
Se si avvia il programma nella cartella con il file di esempio si otterrà il seguente output:

```
Adding room: 0 15743803 0 [Amanti della pizza]
Adding room: 1 5533306 0 [Star Wars]
Adding room: 2 9976816 60 [Appassionati di musica]
Adding room: 3 15769147 5 [Gattini]
Creating server socket
Binding server socket
Set server to listening mode.
Await for client ...
```

Arrivati a questo punto il server è in attesa di una connessione. Nell'esempio seguente un client si connette al server, imposta il proprio nickname a "NomeDiProva" e richiede la lista delle stanze:

```
New connection (n 1) starting thread ...
Thread started.

Await for client ...
[1] Thread started
[1] Nickname is set to [NomeDiProva].
[1] Data received from client: l 0 0 []

[1] Sending rooms to client: {
[1] l 0 0 15743803 0 [Amanti della pizza]
[1] l 1 0 5533306 0 [Star Wars]
[1] l 2 0 9976816 60 [Appassionati di musica]
[1] l 3 0 15769147 5 [Gattini]
[1] }
[1] Data received from client: e

[1] Closing connection
```

Quando viene aperta una connessione il server crea un nuovo thread per gestirla. Ogni thread stamperà i suoi log con il formato $[connectionCount]\,info$ dove $connectionCount$ è un numero univoco assegnato ad ogni nuova connessione.

2.3 Dockerizzazione

Nel caso si voglia utilizzare docker per creare un container, nella root del progetto è presente il seguente Dockerfile:

```
FROM ubuntu:latest
RUN apt update
RUN apt install -y cmake
EXPOSE 8125:8125

COPY . /home
WORKDIR /home
RUN cmake make .
RUN make
RUN make install
RUN rm -r *
COPY rooms.rc .
ENTRYPOINT ["RandomChatServer"]
```

L'immagine creata è una copia di ubuntu nella quale viene installata cmake e RandomChatServer. Per creare l'immagine docker eseguire il comando:

$$\$\ docker\ build\ .\ -t\ randomchat$$

Per avviare il container eseguire il seguente comando:

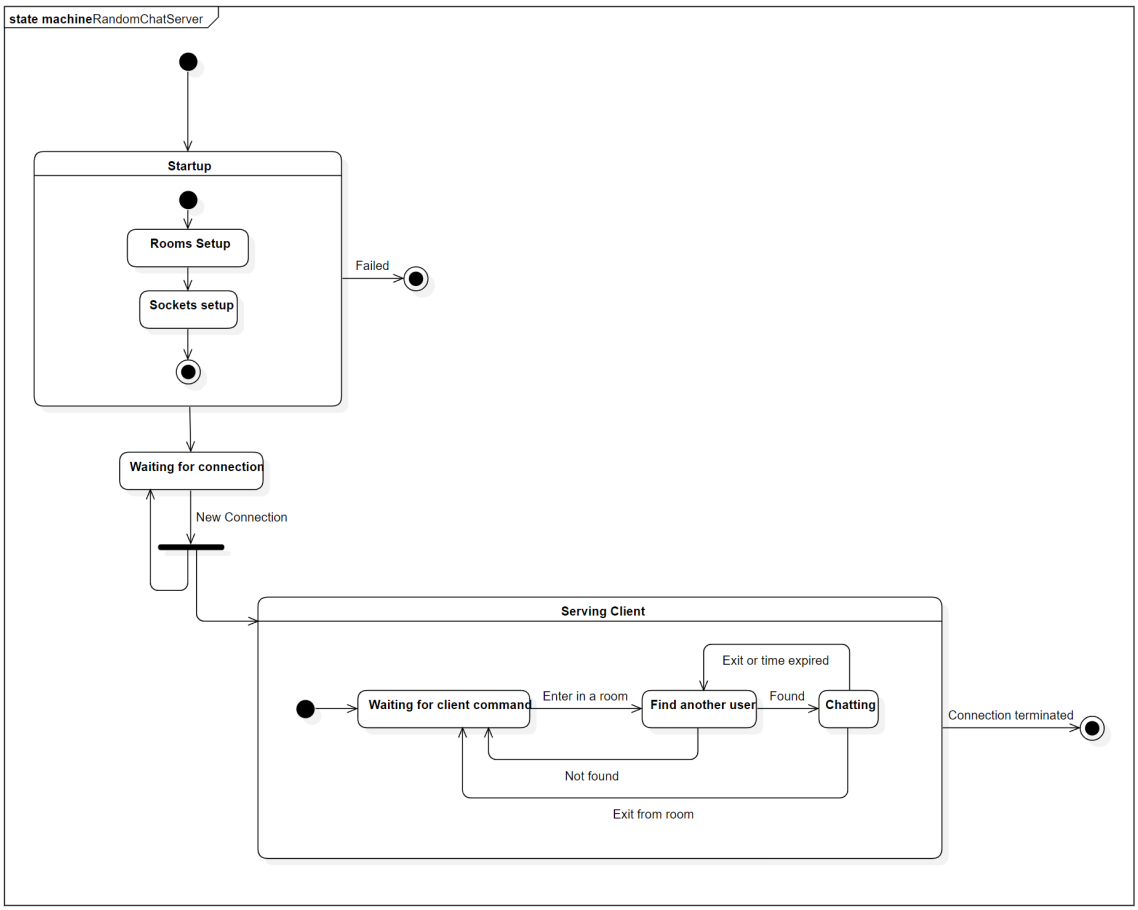
$$\$\ docker\ run\ -p\ 8125:8125\ randomchat$$

Capitolo 3

Progettazione del Server

Sommario

3.1 Diagramma di stato

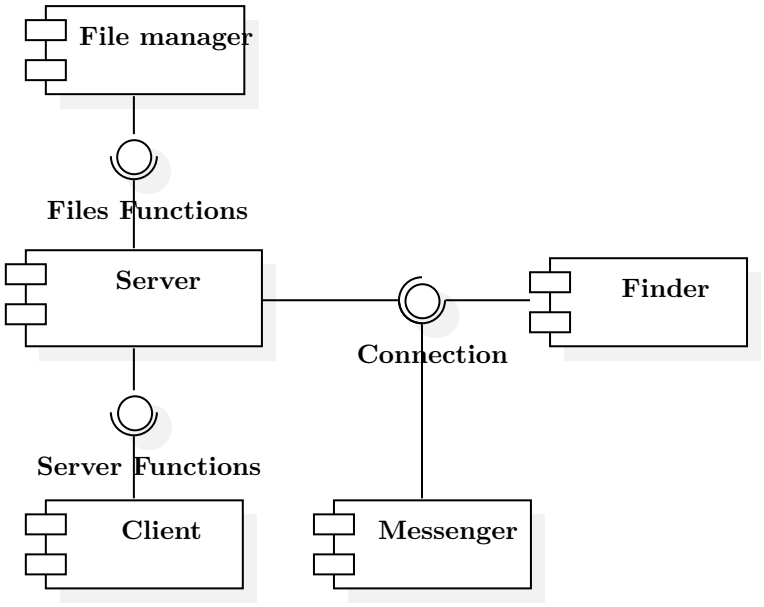


Come si evince dal diagramma a stati il server è stato progettato con protocollo stateful. All'avvio il server si trova in uno stato di configurazione in cui carica le stanze da un file (se presente) e imposta il socket per accettare comunicazioni con i client. Finita la configurazione il server è pronto per accettare una nuova connessione. Quando un client tenta di crearne una, il programma crea un nuovo thread per gestirla e si rimette in ascolto. Appena connesso il client dovrà fornire un nickname con il quale sarà identificato pubblicamente dagli altri utenti. Per questioni di privacy il nickname non è univoco e viene associato al client solo durante la connessione. Ciò significa che se un client termina la comunicazione o si disconnette, al seguente rientro dovrà fornire nuovamente il nickname.

Durante la connessione il client può entrare in una stanza e se la ricerca va a buon fine inizia la chat con un altro utente. In questo stato il client può scambiare messaggi con l'altro utente, terminare la conversazione o uscire dalla stanza.

Lo stato del server non viene preservato alla chiusura del collegamento.

3.2 Diagramma dei componenti



L'analisi Top-Down del problema ha portato allo sviluppo dei moduli rappresentati nel diagramma dei componenti.

Il modulo File manager si occupa del caricamento e salvataggio delle stanze su file. L'interfaccia Files Function consiste nelle funzioni dichiarate nel file *fileManager.h*

```
int stringInside(const char* in, char left, char right, char* out, int maxLen);
void loadFromFile(RoomVector* vec, const char* filename);
pthread_t startAutoSave(RoomVector* vec, const char* filename, int period);
```

L'interfaccia è implementata tramite il file fileManager.c. La funzione *stringInside* è una utility che copia la sotto stringa di *in*, presente nei delimitatori *left* e *right*, nella stringa *out*. La funzione *loadFromFile* carica le stanze dal file *filename* nel vettore *vec*. La funzione *startAutoSave* fa partire un thread che ogni *period* secondi salva le stanze di *vec* nel file *filename*.

Il modulo Finder si occupa di accoppiare utenti nella stessa stanza. L'interfaccia è realizzata tramite i file *finder.h* e *connection.h*, ed è implementata tramite i file *finder.c* e *conncection.c*.

```
Connection* find(User* user, Room*, char* buff, int buff_size);
```

La funzione find contiene l'algoritmo per l'accoppiamento degli utenti. I dettagli dell'implementazione si trovano nella sezione dedicata 3.4.

Il modulo Messenger si occupa dello scambio di messaggi tra utenti accoppiati. È realizzato tramite la funzione *startChatting* definita in *server.h* e implementata in *server.c*.

```
int startChatting(User* userRecv, User* userSend, Connection* conn, char* buff, Room* room)
```

Il modulo Server è il più importante e contiene tutte le funzioni per gestire le richieste del client. Le funzioni del server sono descritte nella sezione apposita 3.3.

Il modulo client contiene la user interface e le funzioni per comunicare con il server. Come da requisito il client è implementato in android. I dettagli possono essere trovati nel capitolo dedicato 4.

3.3 Protocollo di Comunicazione

Client e Server comunicano tra di loro scambiandosi messaggi testuali di massimo 500*bytes* tramite un unico socket tcp. Il primo carattere inviato dal client rappresenta la funzione che dovrà essere eseguita dal server e i caratteri successivi rappresentano gli argomenti (se necessari). I messaggi inviati dal server terminano tutti con una newline.
Una sessione si può trovare in 3 stati principali: Menu, Ricerca e Chatting. Ogni stato ha il suo set di comandi. Quando il client stabilisce la connessione si trova nello stato Menu. Di seguito i comandi con la relativa rappresentazione in caratteri:

- CHANGE_NICKNAME ('c')

  Pattern: CHANGE_NICKNAME [<newNick>]

  Descrizione: Cambia il precedente nickname con newNick. È il primo comando da inviare quando si stabilisce la connessione. La lunghezza massima per il nickname è di 20*bytes*, può contenere qualunque carattere tranne che ']' e non può essere composto da soli spazi vuoti.

  Risposta: Il Server non invia nessuna risposta, sia in caso di successo che di insuccesso. Nel caso si tratti del primo comando e il nickname non fosse valido il server chiuderà la connessione. Nel caso in cui il client invii un comando diverso da questo appena stabilito il collegamento il server chiuderà la connessione.

- ENTER_IN_ROOM ('r')

  Pattern: ENTER_IN_ROOM <roomId>

  Descrizione: Entra nella stanza con id $roomId$.

  Risposta: Quando viene trovato un utente il server risponde con "ENTER_IN_ROOM <otherNick>" dove otherNick è il nickname dell'utente con cui si è stati accoppiati. Nel caso la stanza non esista il server risponderà con "EXIT" ('e').

- NEW_ROOM ('a')

  Pattern: NEW_ROOM <color> <time> [<RoomName>]

Descrizione: Crea una nuova stanza con colore <color> e tempo massimo per chat <time>. Il colore è in formato rgb decimale. Il tempo è epresso in secondi e indica la durata massima delle chat nella stanza. Durante la conversazione allo scadere del tempo il server invia al client TIME_EXPIRED ('t') e chiuderà la connessione tra i due utenti. Per stanze non temporizzate <time> va settato a 0. <RoomName> rappresenta il nome della stanza di massimo 30*bytes*, sono accettati tutti i caratteri tranne ']'.

Risposta: "NEW_ROOM <newId>" dove <newId> è l'id della stanza appena creata.

- ROOM_LIST ('l')

  Pattern: ROOM_LIST <from> <to> [<search>]

  Descrizione: Restituisce una lista di stanze ordinate in ordine decrescente per numero di utenti online. <from> e <to> rappresentano gli indici della lista da inviare. <search> serve per effettuare una ricerca per nome, se lasciato vuoto non effettuerà nessuna ricerca. "ROOM_LIST 0 0 []" invierà la lista di tutte le stanze.

  Risposta: Il primo messaggio di risposta dal server sarà "ROOM_LIST <n>" dove <n> è il numero di stanze che verranno inviate. In seguito verranno inviati altri <n> messaggi ognuno rappresentante una stanza con il formato "ROOM_LIST <id> <onlineUsers> <color> <time> [<roomName>]".

- EXIT ('e'),

  Pattern: EXIT

  Descrizione: Chiude la connessione

  Risposta: EOF

Quando viene utilizzato il comando ENTER_IN_ROOM si entra nello stato di ricerca. Di seguito i comandi disponibili:

- USERS_IN_ROOM ('u')

  Pattern: USERS_IN_ROOM

  Descrizione: Restituisce il numero di utenti online nella stanza

  Risposta: "USERS_IN_ROOM <n>" dove <n> è il numero di utenti online nella stanza.

- EXIT ('e'),

  Pattern: EXIT

  Descrizione: Esce dalla stanza

Risposta: ”EXIT_FROM_ROOM” (’x’)

Quando si viene accoppiati con un altro utente si entra nello stato di chatting. Di seguito i comandi disponibili:

- NEXT_USER (’n’)

  Pattern: NEXT_USER

  Descrizione: Termina la comunicazione con l’utente corrente e ricerca un nuovo utente.

  Risposta: Quando viene trovato un utente il server risponde con ”ENTER_IN_ROOM <otherNick>” dove otherNick è il nickname dell’utente con cui si è stati accoppiati. All’altro utente viene inviato il messaggio ”NEXT_USER”

- SEND_MSG (’m’)

  Pattern: SEND_MSG <msg>

  Descrizione: Invia la stringa <msg> all’utente con cui si è stati accoppiati.

  Risposta: Nessuna. All’altro utente viene inviato il messaggio ”SEND_MSG <msg>”

- USERS_IN_ROOM (’u’)

  Pattern: USERS_IN_ROOM

  Descrizione: Restituisce il numero di utenti online nella stanza

  Risposta: ”USERS_IN_ROOM <n>” dove <n> è il numero di utenti online nella stanza.

- EXIT (’e’),

  Pattern: EXIT

  Descrizione: Esce dalla stanza

  Risposta: ”EXIT_FROM_ROOM” (’x’). All’altro utente viene inviato il messaggio ”EXIT”.

Quando si invia un EXIT dallo stato di ricerca o di chatting si torna allo stato Menu. Se si invia un ”NEXT_USER” dallo stato di chatting si passa allo stato di ricerca.

3.4 Accoppiamento di utenti

Quando un utente entra in una stanza il server deve ricercare un altro utente per poter iniziare la chat. Come già anticipato dalle sezioni precedenti l’algoritmo di accoppiamento è situato nella funzione $find(4)$ di $finder.h/c$. In questa particolare implementazione viene utilizzata una

struttura FIFO, il primo utente che entrerà nella stanza scelta sarà il primo a poter chattare. La randomicità è lasciata alla casualità di arrivo delle connessioni. Questo metodo oltre ad essersi dimostrato valido tramite test euristici, garantisce il progresso e minimizza i tempi di attesa.

```
Connection* find(User* user, Room* room, char* buff, int buff_size);
```

user è l'utente che deve entrare nella stanza *room*. $find$ blocca finchè non viene trovato un altro utente o si esce dalla stanza, per questo sono necessari $buff$ e $buff\_size$ per gestire temporaneamente i messaggi dell'utente. $find$ restituisce un puntatore a Connection in caso di successo, null in caso l'utente sia uscito dalla stanza o abbia chiuso la connessione.

Connection è una struttura che rappresenta la connessione logica di 2 utenti all'interno del server:

```
typedef struct {
    User* user1;
    User* user2;

    int status;
    pthread_mutex_t mutex;
    int pipefd[2];
    Timer* timer;
} Connection;
```

Contiene una coppia di utenti ($user1, user2$), una variabile *status* che stabilisce se la connessione è aperta o no e un timer che chiude la connessione in automatico allo scadere del tempo. La pipe è necessaria per l'implementazione di $find$. Di seguito le funzioni di Connection:

```
Connection* createConnection(User*);
void setConnectionTimeout(Connection*, unsigned int timeout);
void connectUser(Connection*, User*);
void closeConnection(Connection*);
int isOpen(const Connection*);
```

createConnection crea la connessione e imposta user1 al valore dell'utente in input. *setConnectionTimeout* imposta il timeout della connessione in secondi, se la funzione non viene usata allora la connessione non ha timeout. *connectUser* imposta user2 al valore dell'utente in input e modifica il valore di status. *isOpen* ritorna 0 se la connessione è chiusa 1 se aperta. *closeConnection*() chiude la connessione. In caso di successo $find$ restituisce lo stesso puntatore a Connection a entrambi i thread degli utenti perciò non viene mai chiamato esplicitamente $free(connection)$ ma lo spazio viene invece deallocato in automatico, quando entrambi gli utenti hanno chiuso la

connessione. La connessione viene aperta quando i valori di user1 e user2 sono diversi da null e nessuno ha chiamato closeConnection.

Room è una struttura presente in *src/datastructures/entity/room.h*:

```
typedef struct {
    char name[ROOM_NAME_LEN];
    unsigned int id;
    unsigned long long roomColor;
    unsigned int time;
    pthread_mutex_t mutex;
    Queue waitlist;
    long usersCount;
} Room;
```

Ogni stanza tematica ha la propria waitlist cioè una lista di utenti (in realtà una lista di Connection) in attesa di chattare. Queue è una coda di *void∗* implementata come linked list.

Quando un client chiama find ci sono 3 possibilità: nel primo caso waitlist è vuota, ciò significa la stanza è vuota o che gli utenti nella stanza sono tutti accoppiati. nel secondo caso la waitlist ha un solo utente, ma questo utente è proprio l'ultimo con il quale l'utente chiamante ha chattato (o viceversa). Nel terzo caso la waitlist ha almeno un utente diverso dal precedente con cui si ha chattato.

Di seguito il contenuto di find:

```
    pthread_mutex_lock(&room->mutex);

    Connection* firstValidConnection = NULL;
    struct QueueNode* tmp = room->waitlist.front;

    while (tmp != NULL && firstValidConnection == NULL){
        Connection* curr = ((Connection*) tmp->data);
        if(user->prev != curr->user1->connectionCount
        && curr->user1->prev != user->connectionCount){
            firstValidConnection = curr;
        }
        tmp = tmp->next;
    }
```

Dato che room è un dato condiviso da più thread la prima cosa da fare prima di effettuare modifiche è acquisire il lock sulla struttura. In seguito viene ricercata la prima connessione

valida all'interno di waitlist. Una connessione è valida se user1 è diverso dal prev (l'ultimo utente con cui si ha chattato) dell'utente chiamante e viceversa. Gli utenti sono identificati univocamente da connectionCount (> 0). Se un utente non ha mai chattato con nessuno, prev è 0.

I primi 2 casi dei 3 specificati sopra vengono trattati allo stesso modo:

```
Connection *conn = createConnection(user);
enqueue(&room->waitlist, conn);
pthread_mutex_unlock(&room->mutex);
```

Viene creata la connessione con $user1 = user$ e viene accodata in waitlist. Le modifiche a room sono finite e quindi viene rilasciato il lock.

```
while (conn->user2 == NULL) {
    // ... //
}
```

A questo punto il client deve aspettare chè qualche utente si connetta a *conn*.

```
fd_set rfdSet, errfdSet;
FD_ZERO(&rfdSet);
FD_ZERO(&errfdSet);
FD_SET(user->socketfd, &rfdSet);
FD_SET(user->socketfd, &errfdSet);
FD_SET(conn->pipefd[0], &rfdSet);
int retVal = select(((user->socketfd > conn->pipefd[0])
? user->socketfd
: conn->pipefd[0]) + 1, &rfdSet,NULL, &errfdSet, NULL);
```

Durante l'attesa, find deve gestire i messaggi in arrivo dal client e allo stesso tempo terminare le istruzioni di lettura bloccanti quando un altro utente si connette a *conn*. Per questo scopo si è deciso di utilizzare la funzione select per effettuare I/O multiplexing. $connectUser(2)$ quando viene chiamata, scrive un carattere sulla pipe $conn->pipefd$ per far terminare la select.

```
if (FD_ISSET(conn->pipefd[0], &rfdSet)) {
    char c;
    read(conn->pipefd[0], &c, 1);
    continue;
}
```

Quando select termina se $conn->pipefd[0]$ è pronto per la lettura allora viene letto un caratterre dalla pipe e passa alla prossima iterazione.

```
if (FD_ISSET(user->socketfd, &rfdSet)) {
    unsigned int len = recv(user->socketfd, buff,
    buff_size, MSG_DONTWAIT | MSG_NOSIGNAL);
    if (len > 0) {
        buff[len] = '\0';
        printf("[%llu][FINDER] Recived message: %s\n", user->connectionCount, buff);
        if (buff[0] == 'e') terminate = 1;
        else if(buff[0] == 'u'){
            len = sprintf(buff, "%c %lu\n", 'u', room->usersCount);
            send(user->socketfd, buff, len, MSG_NOSIGNAL);
        }
    } else if (errno != EINTR) terminate = 1;
}
```

Se invece $user- > socketfd$ è pronto alla lettura il client ha scritto un comando al server. Come documentato nella sezione 3.3, gli unici comandi accettati da finder sono 'u' ed 'e'. In caso il client abbia scritto 'u' viene inviato il numero di utenti nella stanza. Se viene inviata 'e' oppure c'è stato un errore nella lettura viene settata a 1 la variabile *terminate*.

```
if (FD_ISSET(user->socketfd, &errfdSet) || terminate){
    printf("[%llu][FINDER] Try extract connection from waitlist\n",
    user->connectionCount);
    pthread_mutex_lock(&room->mutex);
    void *removeRet = extract(&room->waitlist, conn);
    pthread_mutex_unlock(&room->mutex);

    if (removeRet == NULL) { // Connection extract by other user
        printf("[%llu][FINDER] Connection not found in waitlist closing connection\n",
        user->connectionCount);
        if (isOpen(conn)) {
            closeConnection(conn);
            buff[0] = 'e';
            buff[1] = '\n';
            send(conn->user2->socketfd, buff, 2, MSG_NOSIGNAL);
        } else closeConnection(conn);
    } else {
        printf("[%llu][FINDER] Connection removed\n",
        user->connectionCount);
        closeConnection(conn); closeConnection(conn);
```

```
    }
    return NULL;
}
```

Nel caso ci sia stato un errore o l'utente abbia inviato una 'e' per uscire dalla stanza va estratta la connessione da waitlist. Se extract ritorna null significa che il thread è arrivato troppo tardi e un altro utente si è già collegato a *conn*. In questo caso va notificato l'altro utente che la connessione è stata chiusa inviandogli una 'e' come specificato dal protocollo. Se invece extract ha successo viene chiamata 2 volte la funzione closeConnection per simulare la chiusura da entrambi i lati e liberare lo spazio allocato. al termine find ritorna null.

```
printf("[%llu][FINDER] Extract user from waitlist\n",
user->connectionCount);
Connection *conn = (Connection *) extract(&room->waitlist, (void*) firstValidConnection);
connectUser(conn, user);
pthread_mutex_unlock(&room->mutex);
return conn;
```

Nell'ultimo caso dei 3 elencati sopra waitlist non è vuota e firstValidConnection è diverso da null. Viene quindi estratta firstValidConnection da waitlist e salvata in conn. In fine l'utente si connette a conn e ritorna la connessione. Conn è null safe dato che viene aquisito il lock su room prima di effettuare la ricerca.

```
char c;
while (read(conn->pipefd[0], &c, 1) != -1); // Clear pipe


if(room->time > 0){
    setConnectionTimeout(conn, room->time);
    startTimer(conn->timer);
}
return conn;
```

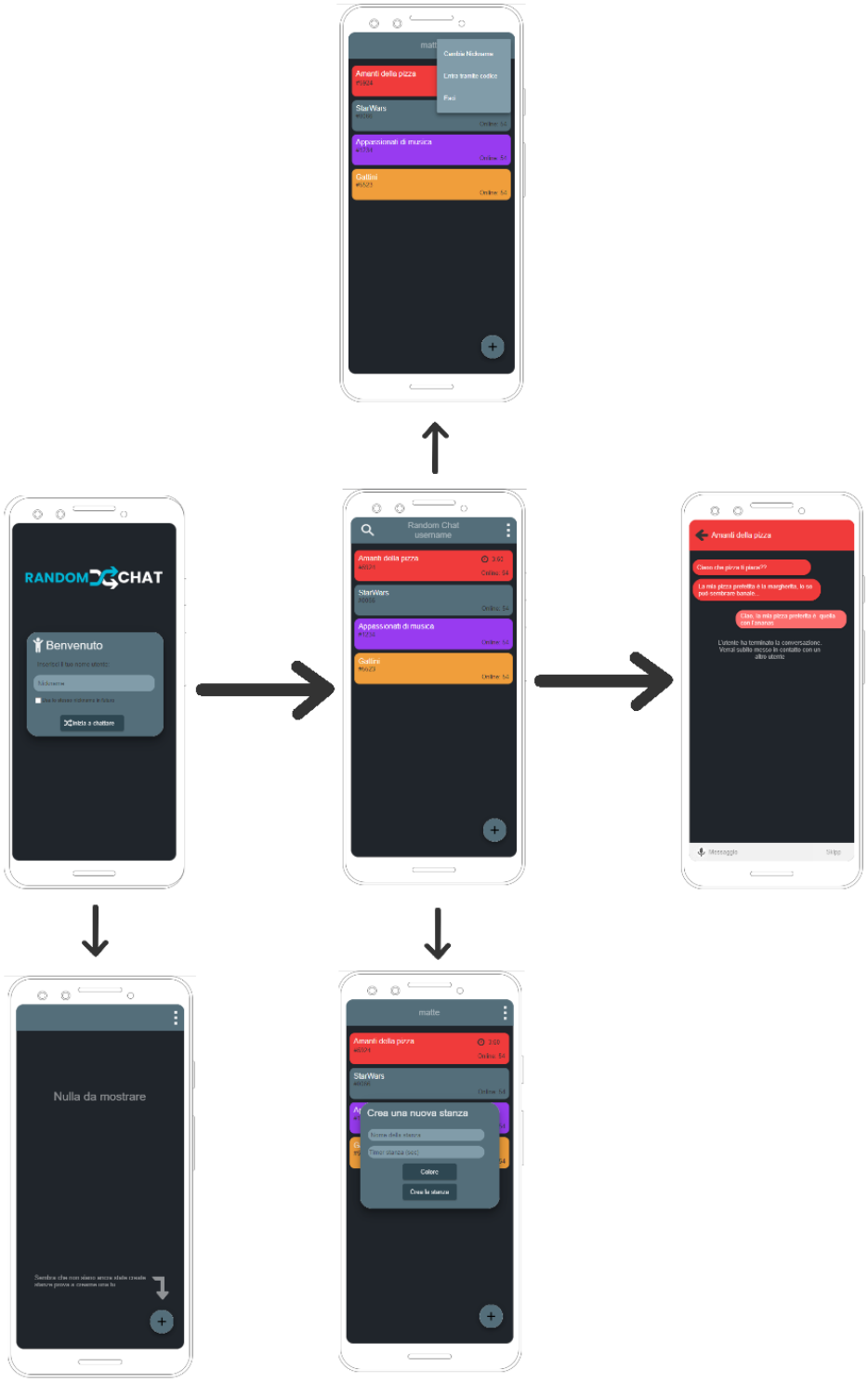
Quando un utente in attesa viene notificato dell'estrazione della connessione esce dal while, pulisce la pipe da eventuali caratteri non letti (la pipe è settata come O_NONBLOCK tramite fcnt), imposta e avvia il timer della connessione se la stanza è temporizzata ed in fine ritorna *conn*.

Capitolo 4

Client Android

Sommario

4.1 Mockup


RANDOM CHAT
Benvenuto
Random Chat
username
Amanti della pizza
StarWars
Appassionati di musica
Gattini
Nulla da mostrare
matte
Crea una nuova stanza

4.2 Implementazione dei Mockup

Per l'implementazione dei mockup e la costruzione dell'app è stato utilizzato il linguaggio Java e l'IDE Android Studio.

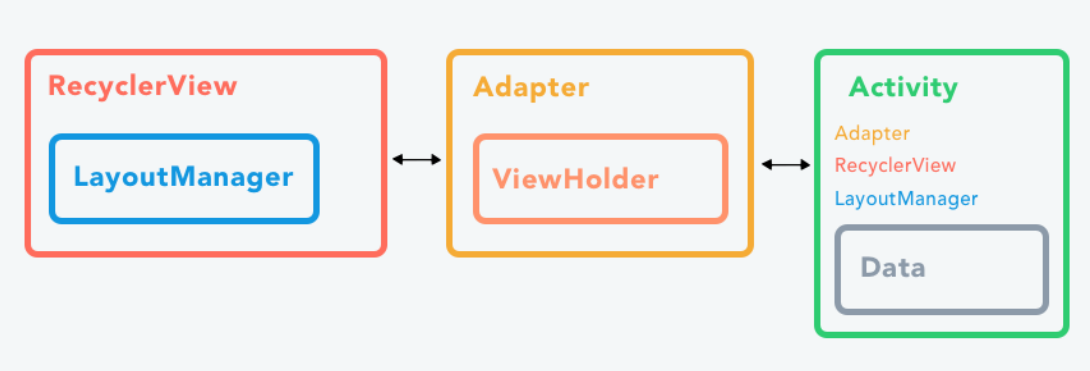
L'intera UI design è stata realizzata sulla base delle Golden Rules di Ben Shneiderman. L'utente ha la facoltà di scegliere tra il **tema chiaro** e **tema scuro**. Al primo avvio sarà guidato, da una serie di piccoli tutorial, implementati tramite Material Tap Target Prompt, che gli permetteranno di comprendere in maniera veloce ed intuitiva il funzionamento dell'applicazione.

Le principali classi estendono AppCompactActivity, in questo modo è possibile navigare tra le varie fasi del ciclo di vita delle Activity attraverso i metodi di callback.

Ad ogni Activity principale è associato il relativo layout in formato **xml**. Per layout complessi (**MainActivity,ActivityRoom,ActivityChat**) sono stati usati due **ViewGroup** specifici **ConstraintLayout** e **RelativeLayout** che consentono di posizionare e dimensionare gli elementi dell'interfaccia utente nel modo più responsive possibile evitando così l'annidamento di layout lineari ed anche eventuali problemi di ridimensionamento degli elementi tra differenti device. All'interno dei vari layout è stato fatto uso delle principali componenti grafiche built-in (TextView, EditText, Button, CheckBox, FloatingActionButton, ImageView, SearchView).

Alcune classi (**PopupEnterInRoom,PopupChangeNickname,PopupNewRoom**) sono rese visibili tramite **PopupView** ed implementate tramite **LayoutInflater** per creare una nuova View sovrapposta all'activity sottostante.

Le classi ActivityChat e ActivityRoom, vista la necessita di mostrare una lista di elementi in sequenza, sono state realizzate tramite **RecyclerView**.



In particolare le classi adapter (**MessagesRecyclerAdapter, RoomAdapter**), ereditate da

RecyclerView.Adapter, ci consentono di associare le viste RecyclerView alle nostre classi principali (ActivityChat e ActivityRoom). L'adapter è costruito con riferimento a una lista di oggetti (Rooms,Messages) e tre fondamentali metodi: **onCreateViewHolder(), onBindViewHolder(), getItemCount()**

- **onCreateViewHolder()**
  Ha il compito di inizializzare nuova view quando non è presente una view già esistente che la RecyclerView possa riutilizzare.

- **onBindViewHolder()**
  Ha il compito di aggiornare il contenuto del singolo item in modo che rifletta l'elemento nella posizione specificata

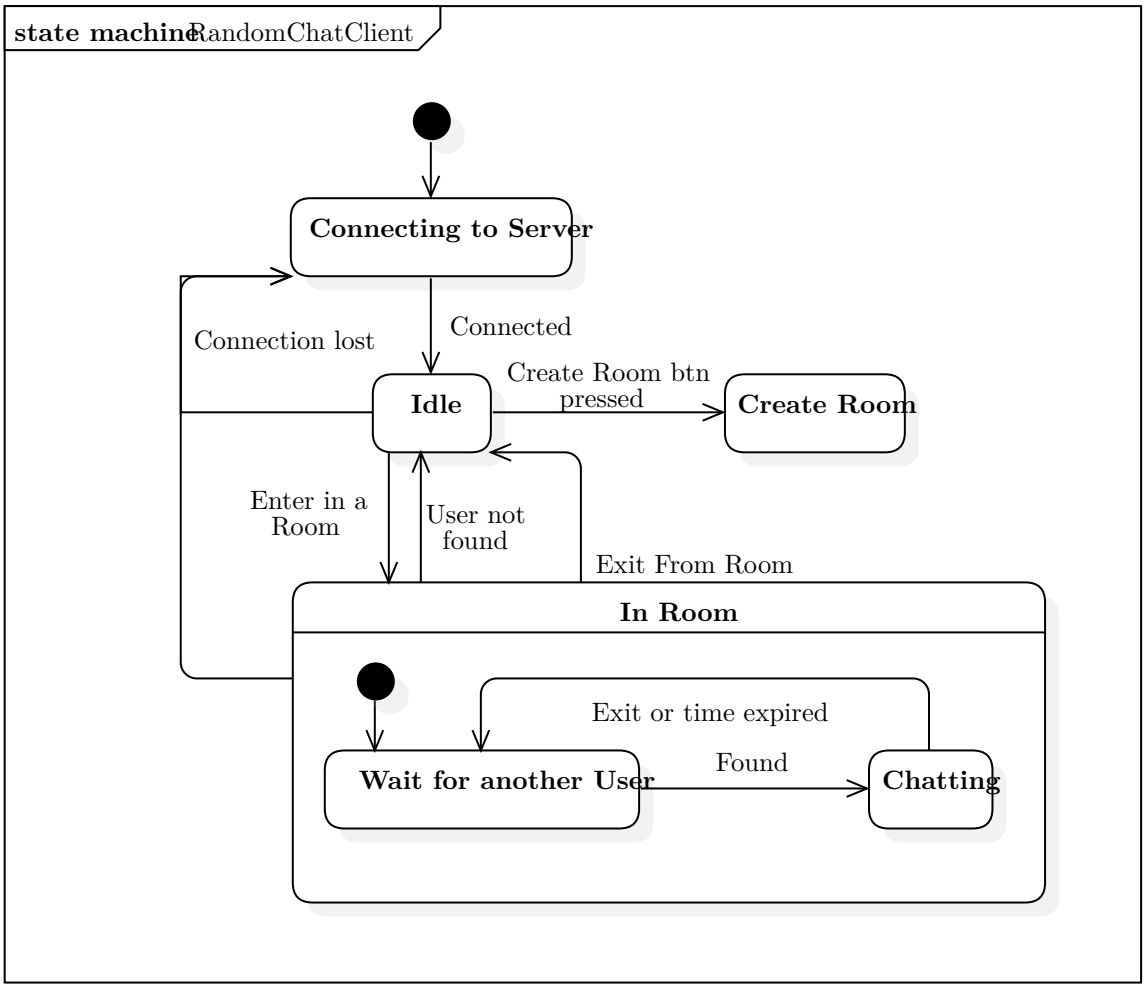
- **getItemCount()**
  Ha il compito di restituire la dimensione del nostro elenco di oggetti, e quindi indicare all'Adapter il numero di righe che la RecyclerView può contenere.

All'interno delle classi MessagesRecyclerAdapter e RoomAdapter sono presenti due classi nidificate chiamate **ViewHolder** il loro obbietivo è quello di modellare, in un oggetto java, la precedente vista XML statica che rappresenta il singolo elemento della RecyclerView.

L'uso della RecyclerView permette di ridurre l'occupazione della memoria e velocizzare il caricamento in quanto, tramite il riciclo delle viste attraverso l'utilizzo del ViewHolder, ci permette di non creare un layout per ogni singolo elemento da mostrare nella lista.

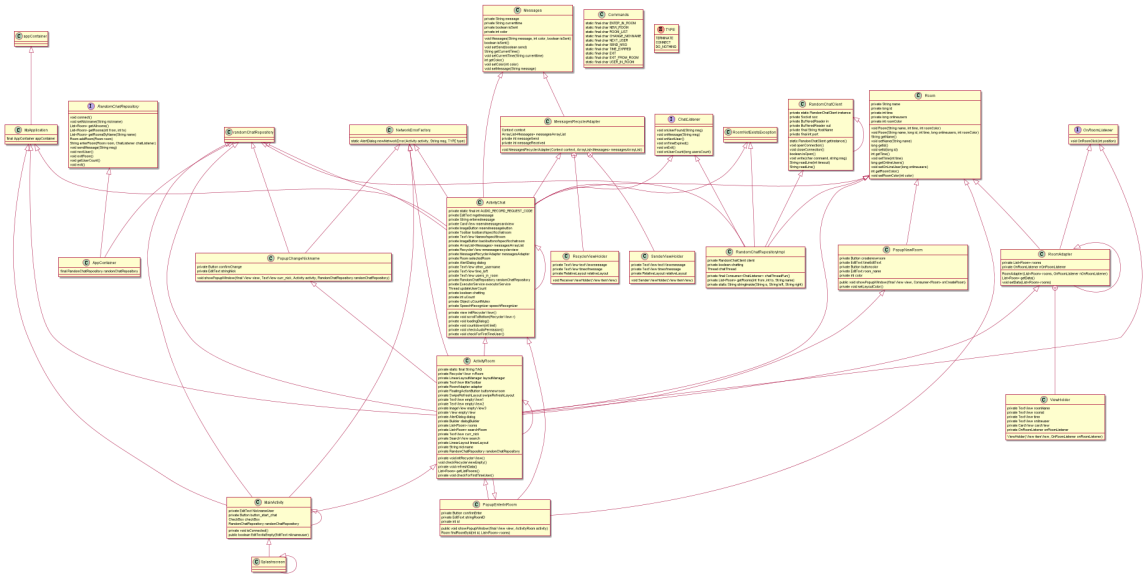
All'interno della classe ActivityRoom è stato implementato anche uno **Swiperefreshlayout**, che permette l'aggiornamento della lista delle stanze senza dover uscire e rientrare nell'activity.

4.3 Diagramma di stato



Il diagramma di stato del client è molto simile a quello del server. All'avvio dell'applicazione il client si connette al server, a connessione avvenuta si mette in attesa di un qualche input dall'utente. A questo punto l'utente può entrare in una stanza o crearne una. Entrati nella stanza il server cercherà un utente con cui chattare. In qualunque momento è possibile uscire dalla stanza o chiudere la conversazione.

4.4 Diagramma delle Classi



4.5 Comunicazione con il Server

Per progetti costituiti da una moltitudine di classi che si interfacciano per effettuare operazioni in locale e in remoto, è sempre conveniente ricercare il massimo livello di separazione possibile tra lo strato di logica di accesso ad host remoti e logica funzionale/grafica locale.
Sono molte le tecniche con cui è possibile ottenere un'astrazione del genere, ma l'**Inversione del controllo (IoC)** si è rivelata la più efficace.

Generalmente, in un tipico flusso di esecuzione di un software procedurale, le funzioni custom definite dal programmatore richiedono la partecipazione diretta di funzioni generiche di libreria, per adempiere a singoli task.

In software che sfruttano la IoC, il flusso di esecuzione è invertito:
Un framework generico richiede l'esecuzione di porzioni di codice definite dallo sviluppatore, in modo tale da incentivare la customizzazione dei moduli e la loro riusabilità.
Ciò allo scopo di astrarre il più possibile l'esecuzione di un task dalla sua implementazione.

Mentre in un codice procedurale senza inversione del controllo il risultato dell'esecuzione dipende principalmente da oggetti legati staticamente gli uni agli altri, un software che sfrutta l'IoC può costruire un grafo delle relazioni tra oggetti diverso per ogni esecuzione, dipendendo dal suo comportamento a Run-Time.
Un flusso dinamico e mutevole è possibile grazie ad una definizione astratta delle relazioni tra oggetti, il cui binding a run-time è ottenuto tramite alcune tecniche come la **Dependency Injection** oppure un **Service Locator**. Quest'ultimo, tuttavia, è piuttosto difficile da implementare correttamente. Si basa infatti sull'uso di un registro contenente le informazioni sulle

classi, ma che omette le loro dipendenze. Se usato impropriamente può dunque causare errori di binding a run-time o memory leaks.

Per ottimizzare le operazioni di accesso e ricerca sul server è stato scelto di utilizzare il Repository Pattern, attraverso cui è possibile garantire la separazione tra le classi di dominio e quelle inerenti alla raccolta e alla trasmissione dei dati. Ciò è stato realizzato utilizzando la tecnica della Dependency Injection, che verrà approfondita in seguito.

Il suddetto pattern architetturale prevede l'utilizzo di un'interfaccia nella quale sono definite le signature dei principali metodi adempienti alle funzioni CRUD ed eventualmente altri metodi specifici al caso d'uso; essa viene poi implementata da una o più classi che definiscono i corpi dei prima citati metodi.

Nel caso del presente lavoro, la classe RandomChatRepository.java svolge il ruolo di **Repository Interface**, mentre RandomChatRepositoryImpl.java rappresenta l'**Entity Manager**, ossia la classe implementante, che effettivamente definisce le operazioni da effettuare sul server remoto.

Nel contesto del progetto in questione, tutte le classi che necessitano di effettuare operazioni di comunicazione con il server condividono un'unica istanza della repository.
Per assicurare l'unicità, è stato fatto uso di un container, **AppContainer.java**, il cui scopo è quello di fornire dipendenze in caso di richiesta.
Per garantire la condivisione, è stata creata la classe **MyApplication.java** che estende Application, e consente di raccogliere le dipendenze in un unico luogo, attraverso la loro gestione tramite un oggetto appContainer:AppContainer.

Il recupero della dipendenza può essere effettuato nel seguente modo, come da documentazione ufficiale Android:

```
AppContainer appContainer = ((MyApplication) getApplication()).appContainer;
```

In conclusione, la tecnica della Dependency Injection si è rivelata utile per centralizzare la logica di accesso ai dati, oltre ad essere stata utile per migliorare la leggibilità del codice. Tuttavia c'è uno svantaggio, ossia la creazione di molte classi diverse; per questo motivo le relative classi sono state incluse nel package "depinjection".

Per approfondimenti: Manual Dependency Injection