

UNIVERSITÀ 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 DI LABORATORIO DI SISTEMI OPERATIVI

ANNO ACCADEMICO 2022/2023

Progettazione e sviluppo di un'applicazione Client – Server, multithread, con l'ausilio delle system call del sistema UNIX.

Autore:

Fabiola Salomone
Matricola N86/2870
fab.salomone@studenti.unina.it

Docente:

Prof. Alessandra Rossi

Indice

Guida all'uso del Server e alla compilazione	3
Guida all'uso del Client	5
Codice sorgente : Server	9
Dettagli Implementativi del Server	21
Dettagli Implementativi del Client	27
○ Android Studio e pattern	28
○ Neo4J	28

Guida all'uso del Server e alla compilazione

La compilazione dell'applicativo avviene mediante una **compilazione manuale**.

Si parte dalla cartella principale **Progetto**, che nel nostro caso risiede in **Desktop**. Eseguiamo da terminale, il comando:

```
cd Desktop/Progetto/Progetto_Server
```

In tale cartella troviamo tre file:

- client.h
- client.c
- LSO_server.c

Che compiliamo con il comando:

```
gcc -c <nomeFile.c>
```

Da cui otteniamo i file con estensione “.o” che, a sua volta, eseguiamo con:

```
gcc -o "LSO_server" ./LSO_server.o ./client.o -pthread
```

Si genera un file eseguibile chiamato “LSO_server” che compiliamo con:

```
./LSO_server
```

Da qui parte l'applicativo che mostra un messaggio attraverso il quale l'utente viene invitato ad inserire la porta su cui il server si metterà in ascolto.

Prima di mettersi in ascolto, il server, verificherà se è possibile utilizzare quella porta indicata e se riesce a creare correttamente la socket.

Dopodiché, i/il client può/possono connettersi al Server ed effettuare tutte le varie funzioni offerte da quest'ultimo.

Tra le funzioni offerte abbiamo:

- Registrazione alla piattaforma
- Accesso alla piattaforma

Alla **registrazione** dell'utente alla piattaforma, gli viene richiesto di inserire un username e una password, se entrambe queste credenziali oltrepassano i controlli implementati nel server verrà generato il relativo account.

I controlli implementati nel server, per verificare se le credenziali immesse dall'utente nella fase di registrazione risultano valide, sono:

- Controllo della lunghezza delle stringhe;
- Controllo della presenza di spazi all'interno delle stringhe immesse dall'utente;
- Controllo duplicato utente nel “database” dell'applicativo.

Mentre all'**accesso** l'applicativo richiede all'utente di inserire le proprie credenziali ossia username e password ed anche in questo caso vengono effettuati, da parte del Server, dei controlli per stabilire l'autenticità di tali dati. I controlli effettuati in tal caso sono:

- Controllo della lunghezza delle stringhe;

- Controllo della presenza di spazi all'interno delle stringhe immesse dall'utente;
- Controllo dell'esistenza effettiva dell'account corrispondente alle credenziali immesse
- Controllo che account inserito ha già effettuato l'accesso al sistema.

Una volta che l'utente ha inserito le credenziali (username e password) e che queste superano i controlli implementati nel server, l'utente potrà usufruire di ulteriori funzioni messe a disposizione dall'applicativo come:

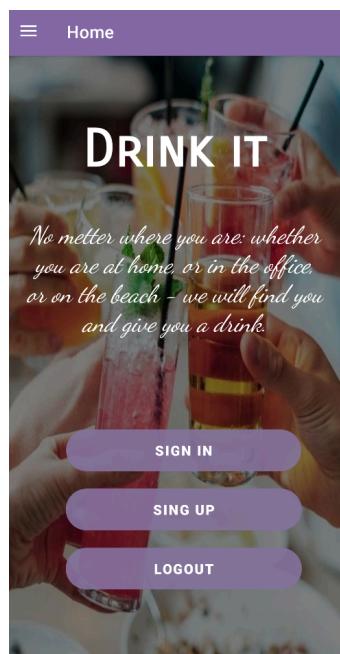
- Sceglierà i drink d'acquistare;
- Selezionare la quantità di prodotti;
- Effettuare uno o più ordini.

Guida all'uso del Client

Segue una descrizione dell'applicazione per consentire la comprensione del suo funzionamento e del suo utilizzo.

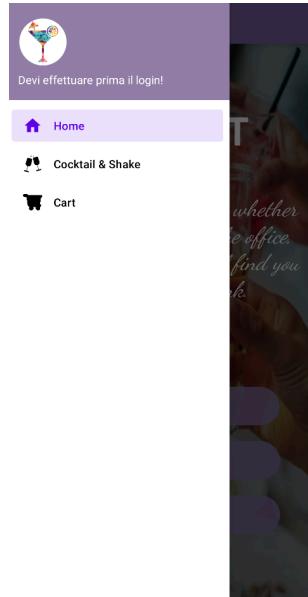
L'applicazione si presenta con una schermata di collegamento al Server, in cui si dovrà inserire l'indirizzo IP del server a cui ci vogliamo collegarci e la porta su cui il server si metterà in ascolto.

Quindi è importante che l'utente, per poter utilizzare a pieno l'applicazione, conosca queste informazioni.



Una volta che il collegamento al server è stato effettuato correttamente, da parte dell'utente, il client mostrerà una schermata denominata “Home” attraverso cui l'utente avrà accesso ad un menu e sarà possibile, come vedremo in seguito, far eseguire al server i meccanismi di Accesso e di Registrazione all'applicazione.

Le schermate di “menu” è la seguente:



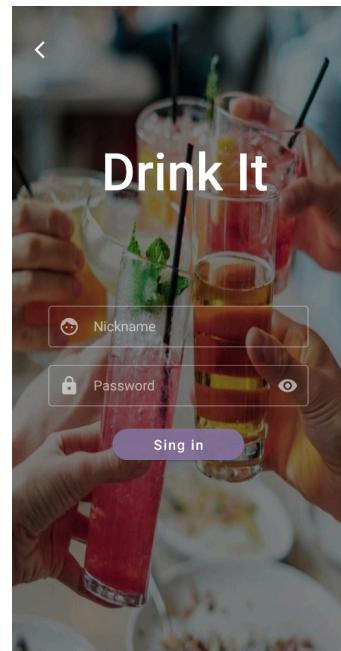
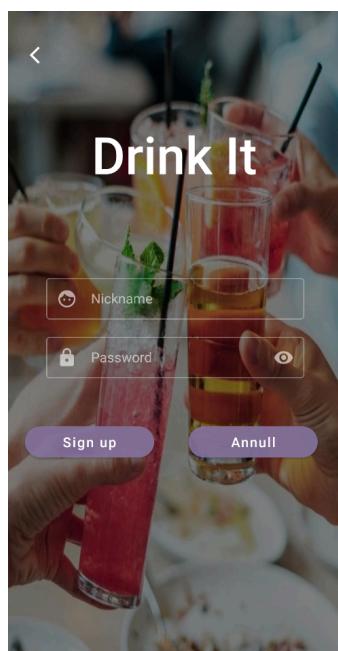
L’utente da tale menù avrà accesso solo alla voce “Home”, se non ha effettuato l’accesso, le altre schermate presentano un messaggio che notifica l’utente che deve effettuare l’accesso per poterne usufruire .

La voce “Home” permette all’utente di effettuare tre operazioni:

- Registrazione
- Accesso
- LogOut

Le schermate di accesso e registrazione sono le seguenti:

REGISTRAZIONE

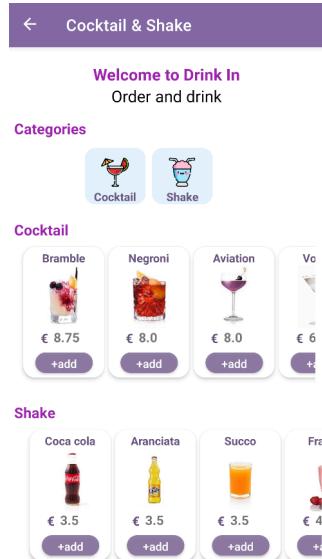


ACCESSO

Se l'operazione di registrazione e accesso va buon fine gli utenti possono usufruire di ulteriori funzioni presenti nel menu dell'applicativo, invece, in caso d'errore, sia per la registrazione che l'accesso, il server notifica al client che è presente un'errore e il client mostrerà un messaggio che fa presente all'utente l'errore commesso.

Ipotizziamo che i dati inseriti siano corretti, effettuato l'accesso alla piattaforma, l'utente ha la possibilità di usufruire delle restanti funzionalità attraverso apposite sezioni dell'applicativo client.

Le schermate di “Cocktail & Shake” è la seguenti:



La voce “Cocktail & Shake” permette all’utente di selezionare il/i drink desiderato/i e visualizzare le relative informazioni

Cliccando su “+add” l’utente, ha accesso ad una nuova schermata dell’applicazione che gli permette di visualizzare il prodotto nel dettaglio osservandone nome, prezzo, immagine e descrizione e ha, inoltre, la possibilità di aggiungerlo ad un ipotetico “carrello”.

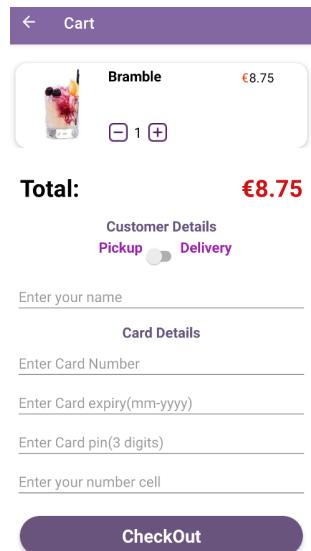
La schermata che dettaglierà il drink è la seguente:



Cliccando sul pulsante “Add to Cart”, dalla schermata precedente, il prodotto verrà aggiunto al carrello.

Invece la voce del menù “Cart” permette all’utente di scegliere la quantità di prodotto che desidera acquistare e di inserire una serie di dati per effettuare l’ordine.

La schermata “Cart” è la seguente:



Codice sorgente : Server

Il server è stato implementato facendo uso di linguaggio C ed i file che possiamo trovare all'interno della cartella **Progetto** sono:

- **client.h**

Un file attraverso il quale viene definita la struttura con cui il server gestisce le informazioni legate ad un client connesso.

```
1.  /***** INIT CLTINF HEADER *****/
2.
3.  #ifndef CLIENT_H_
4.  #define CLIENT_H_
5.
6.  #endif /* CLIENT_H_ */
7.  #include <netinet/in.h>
8.  #include <time.h>
9.  #include <stdbool.h>
10. #include <pthread.h>
11.
12. #define CLTINF_USERNAME_STRLEN    10
13. #define CLTINF_PASSWORD_STRLEN    10
14.
15. #define CLTINF_GUEST              1
16. #define CLTINF_LOGGED             3
17.
18.
19. /**
20.  * @attribute tidHandler: TID del thread principale che gestisce un determinato client.
21.  * @attribute clientSocket: Socket descriptor della socket associata ad un determinato client.
22.  * @attribute clientAddressIPv4: IPv4 del client.
23.  * @attribute username: Username del client.
24.  * @attribute timestamp: Ora della connessione
25.  * @attribute status: LOGGED, GUEST.
26.  * @attribute prevClientInfo: Puntatore al nodo clientInfo precedente.
27.  * @attribute nextClientInfo: Puntatore al nodo clientInfo successivo.
28.  * @attribute mutexSocket: Mutex associato alla socket.
29. */
30.
31. struct ClientInfo
32. {
33.
34.     pthread_t tidHandler;
35.     int clientSocket;
36.     char clientAddressIPv4[INET_ADDRSTRLEN];
37.     char username[CLTINF_USERNAME_STRLEN];
38.     time_t timestamp;
39.     int status;
40.     pthread_mutex_t mutexSocket;
41.
42.     /* lista doppiamente puntata */
43.     struct ClientInfo* prevClientInfo;
44.     struct ClientInfo* nextClientInfo;
45. };
46.
47. typedef struct ClientInfo ClientInfo;
48. typedef ClientInfo* LpClientInfo;
49.
50. /**
51.  * La funzione newClientInfo alloca un nuovo nodo clientInfo con gli attributi passati
52.  * in ingresso.
53.  */
54.
55. LpClientInfo newClientInfo(int, char[], char[]);
56.
57. /**
58.  * La funzione insertClientInfo inserisce un nuovo nodo clientInfo in testa alla lista
59.  * listClientInfo.
60.  */
61.
62. void insertClientInfo(LpClientInfo*, LpClientInfo);
63.
64. /**
65.  * La funzione deleteClientInfo elimina il nodo specificato da targetedClientInfo dalla
66.  * lista in cui risiede lo stesso.
67.  */
68.
69. void deleteClientInfo(LpClientInfo*, LpClientInfo*);
70.
71. /***** END CLTINF HEADER *****/
```

• client.c

Un file in cui sono definite le funzioni di gestione della lista delle informazioni legati ai client connessi (struttura definita in client.h).

```
1.  **** INIT CLTINF ****
2.
3.  #include <stdio.h>
4.  #include <stdlib.h>
5.  #include <string.h>
6.  #include <sys/types.h>
7.  #include "client.h"
8.
9.
10. /**
11.  * Nella funzione newClientInfo viene creato un nodo ClientInfo con le informazioni fornite dal client
12. */
13.
14. LpClientInfo newClientInfo(int clientSocket, char clientAddressIPv4[], char username[]){
15.     LpClientInfo newClientInfo = (LpClientInfo)malloc(sizeof(ClientInfo));
16.     if(newClientInfo != NULL){
17.         newClientInfo->clientSocket = clientSocket;
18.         strcpy(newClientInfo->clientAddressIPv4, clientAddressIPv4);
19.         strcpy(newClientInfo->username, username);
20.         newClientInfo->timestamp = time(NULL);
21.         newClientInfo->status = CLTINF_GUEST;
22.         newClientInfo->prevClientInfo = NULL;
23.         newClientInfo->nextClientInfo = NULL;
24.         pthread_mutex_init(&(newClientInfo->mutexSocket), NULL);
25.     }
26.     return newClientInfo;
27. }
28.
29.
30.
31. /**
32.  * La funzione insertClientInfo permette di inserire un nuovo nodo ClientInfo dalla testa della lista
33. */
34.
35. void insertClientInfo(LpClientInfo* listClientInfo, LpClientInfo newClientInfo){
36.     if(newClientInfo != NULL){
37.         newClientInfo->nextClientInfo = *listClientInfo;
38.         if(*listClientInfo != NULL){
39.             newClientInfo->prevClientInfo = (*listClientInfo)->prevClientInfo;
40.             (*listClientInfo)->prevClientInfo = newClientInfo;
41.             if(newClientInfo->prevClientInfo != NULL){
42.                 newClientInfo->prevClientInfo->nextClientInfo = newClientInfo;
43.             }
44.         }
45.         *listClientInfo = newClientInfo;
46.     }
47. }
48.
49.
50. /**
51.  * La funzione deleteClientInfo permette di eliminare un nodo ClientInfo specificato in ingresso dalla
52.  * lista
53. */
54. void deleteClientInfo(LpClientInfo* listClientInfo, LpClientInfo* targetedClientInfo){
55.     if(targetedClientInfo != NULL){
56.         LpClientInfo tmp = *targetedClientInfo;
57.         *targetedClientInfo = (*targetedClientInfo)->nextClientInfo;
58.         if(tmp->prevClientInfo != NULL){
59.             tmp->prevClientInfo->nextClientInfo = *targetedClientInfo;
60.         }
61.         if(*targetedClientInfo != NULL){
62.             (*targetedClientInfo)->prevClientInfo = tmp->prevClientInfo;
63.         }
64.         if(*listClientInfo == tmp){
65.             *listClientInfo = *targetedClientInfo;
66.         }
67.         free(tmp);
68.     }
69. }
70.
71. **** END CLTINF ****
```

• LSO_server.c

Un file che contiene le funzioni di gestione del server, tra le quali possiamo trovare: Inizializzazione del server, gestione dei thread e gestione della socket.

```
1.  ****INIT SERVER*****
2.  //____LIBRARIES_____
3.
4.
5.  #define BUFFER_STRLEN 150
6.  #define LISTENER_QUEUE_STRLEN 50
7.  #define INCOMING_MSG_STRLEN 70
8.  #define OUTCOMING_MSG_STRLEN 82
9.  #define GRAPHICS_CHAT_WIDTH 82
10.
11. //____COSTANTS_____
12.
13. #include <math.h>
14. #include <stdio.h>
15. #include <stdlib.h>
16. #include <sys/socket.h>
17. #include <sys/types.h>
18. #include <netinet/in.h>
19. #include <errno.h>
20. #include <arpa/inet.h>
21. #include <pthread.h>
22. #include <stdbool.h>
23. #include <string.h>
24. #include <signal.h>
25. #include <unistd.h>
26. #include <time.h>
27. #include <fcntl.h>
28. #include <ctype.h>
29. #include <stdbool.h>
30. #include "client.h"
31.
32. //____GLOBAL VARIABLES_____
33.
34. pthread_mutex_t mutexClientInfo = PTHREAD_MUTEX_INITIALIZER;
35. pthread_mutex_t mutexDatabase = PTHREAD_MUTEX_INITIALIZER;
36. pthread_mutex_t mutexLogs = PTHREAD_MUTEX_INITIALIZER;
37. pthread_mutex_t mutexCursor = PTHREAD_MUTEX_INITIALIZER;
38.
39. LpClientInfo listClientInfo = NULL;
40. int totalConnections = 0;
41. int connectedClients = 0;
42. int totalInfections = 0;
43. int listenerSocket;
44. int listenerPort;
45. int logs;
46. int database;
47.
48. //____FUNCTIONS DECLARATION_____
49.
50. int createSocket();
51. void *listenerClient(void *arg);
52. void *connectionRequestsManagement(void *arg);
53. bool signIn(LpClientInfo clientInfo);
54. bool login(LpClientInfo clientInfo);
55. int initFile(void);
56. void sendMsg(LpClientInfo clientInfo, char *outcomingMsg);
57. void sendMsgToAll(LpClientInfo, char *);
58. void signalHandler(int);
59.
60. int main()
61. {
62.
63.     pthread_t tidConnectionRequestsManagement;
64.
65.     if (initFile())
66.     {
67.
68.         printf("\n!> Errore durante l'inizializzazione del Server 1.\n\n");
69.
70.         return 1;
71.     }
72.
73.     if (createSocket())
74.     {
75.         return 1;
76.     }
77.
78.     signal(SIGSTOP, signalHandler);
79.     signal(SIGINT, signalHandler);
80.
81.     //handler sigstop e sigint
82.     pthread_create(&tidConnectionRequestsManagement, NULL, connectionRequestsManagement, NULL);
83.     printf("\nIn ascolto ... \n\n");
```

```

84,
85.         while (true) {}
86.
87.         return 0;
88.     }
89.
90. //_____FUNCTION FOR MANAGEMENT CLIENT_____
91. int createSocket()
92. {
93.
94.
95.     int bytesReaded;
96.     char buffer[BUFFER_STRLEN];
97.     int port;
98.     bool error;
99.
100.    struct sockaddr_in serverAddress;
101.    memset(&serverAddress, '0', sizeof(serverAddress));
102.    serverAddress.sin_family = PF_INET;
103.    serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
104.
105. /* inserire porta */
106. printf("\n » Inserire la porta del server: ");
107. fflush(stdout);
108.
109. do {
110.
111.     error = false;
112.     if ((bytesReaded = read(STDIN_FILENO, buffer, BUFFER_STRLEN)) == -1)
113.     {
114.         perror("\n<!> Errore read");
115.
116.         return 1;
117.     }
118.     buffer[bytesReaded] = '\0';
119.
120. /* Verifico se la porta fornita è valida */
121. for (int i = 0; i < strlen(buffer) - 1; i++)
122. {
123.     if (buffer[i] < '0' || buffer[i] > '9')
124.     {
125.         error = true;
126.         break;
127.     }
128.
129. }
130. if (error)
131. {
132.
133.     printf("\n<!> Porta non valida, caratteri non ammessi - riprovare: ");
134.
135.     fflush(stdout);
136. }
137. else
138. {
139.     if ((port = atoi(buffer)) == 0 || !(port >= 0 && port <= 65535))
140.
141.     {
142.         printf("\n<!> Porta non valida, out of range - riprovare: ");
143.
144.         fflush(stdout);
145.         error = true;
146.     }
147.     else
148.     {
149.         listenerPort = port;
150.
151.         printf("\n    » Porta %d inserita con successo.\n", port);
152.
153.         fflush(stdout);
154.         serverAddress.sin_port = htons(atoi(buffer));
155.     }
156.
157.     if (!error)
158.     {
159.         listenerSocket = socket(PF_INET, SOCK_STREAM, 0);
160.         /* Assegno un'indirizzo alla socket del server */
161.         if (bind(listenerSocket, (struct sockaddr *) &serverAddress,
162.                  sizeof(serverAddress)) == -1)
163.
164.             perror("\n<!> Errore bind");
165.             puts("");
166.
167.             sleep(1);
168.             error = true;
169.     }
170.     printf("bind riuscito");
171.
172. /*Rimango in ascolto di richieste di connessione da parte di client */
173.     if (listen(listenerSocket, LISTENER_QUEUE_STRLEN) == -1)

```

```

174.         {
175.             perror("\n<!> Errore listen");
176.             puts("");
177.             sleep(1);
178.             return 1;
179.         }
180.     }
181. }
182. }
183. } while (error != false);
184.
185. return 0;
186.
187. }
188. }
189.
190. /**
191. * @param clientInfo: puntatore della struttura che contiene informazioni del client.
192. *
193. * @return: puntatore a void.
194. *
195. * La funzione disconnectionManagement gestisce le operazioni di disconnessione del client.
196. *
197. */
198.
199. void disconnectionManagement(LpClientInfo clientInfo)
200. {
201.     time_t timestamp = time(NULL);
202.     char logsBuffer[BUFFER_STRLEN];
203.     pthread_t tidHandler;
204.
205.
206.     sprintf(logsBuffer, " > Il client %s si è disconnesso dal Server - %s", clientInfo-
>clientAddressIPv4, ctime(&timestamp));
207.     printf("\n > Il client %s si è disconnesso dal Server - %s", clientInfo->clientAddressIPv4,
ctime(&timestamp));
208.
209.     pthread_mutex_lock(&mutexLogs);
210.     if (write(logs, logsBuffer, strlen(logsBuffer)) == -1)
211.     {
212.         printf("Errore durante la scrittura nel file di logs.");
213.     }
214.     pthread_mutex_unlock(&mutexLogs);
215.
216.     pthread_mutex_lock(&mutexCursor);
217.     connectedClients -= 1;
218.     pthread_mutex_unlock(&mutexCursor);
219.
220.     pthread_mutex_lock(&mutexClientInfo);
221.     /*Chiudo la socket di comunicazione */
222.     tidHandler = clientInfo->tidHandler;
223.     close(clientInfo->clientSocket);
224.     /*Elimino dalla lista dei client, il client disconnesso */
225.     deleteClientInfo(&listClientInfo, &clientInfo);
226.     pthread_mutex_unlock(&mutexClientInfo);
227.     /*elimino il thread*/
228.     pthread_exit(tidHandler);
229.
230. }
231.
232. /**
233. * @param clientinfo : arg -> puntatore della struttura che contiene informazioni
234. * del client connesso alla socket del server.
235. *
236. * @return: puntatore a void.
237. *
238. * La funzione listenerClient si occupa di gestire le azioni di ascolto
239. * dei messaggi da parte del client, e l'invio dei comandi utilizzabili
240. * al client.
241. *
242. */
243.
244. void *listenerClient(void *arg)
245. {
246.
247.     LpClientInfo clientInfo = (LpClientInfo) arg;
248.     char incomingMsg[INCOMING_MSG_STRLEN]; /* Buffer messaggio in entrata */
249.     int bytesReaded; /* Numero di bytes letti dalla read */
250.
251.     bool exited = false;
252.
253.     /* Invio il messaggio di benvenuto */
254.     sendMsg(clientInfo, "$Server: Benvenuto/a! Inserisci i comandi.\n");
255.     memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
256.
257.     while (exited != true && (bytesReaded = read(clientInfo->clientSocket, incomingMsg,
INCOMING_MSG_STRLEN)) > 0)
258.     {
259.         incomingMsg[bytesReaded] = '\0';
260.
261.         switch (clientInfo->status)

```

```

262.
263.        {
264.            case CLTINF_GUEST:
265.                if (strncmp(incomingMsg, "signin", 6) == 0)
266.                {
267.                    sendMsg(clientInfo, "$Server: sei entrato in registrazione.\n");
268.
269.
270.
271.
272.
273.
274.
275.
276.
277.
278.
279.
280.
281.
282.
283.
284.
285.
286.
287.
288.
289.                    riprova!");
290.
291.                }
292.                break;
293.
294.            case CLTINF_LOGGED:
295.
296.                if (strncmp(incomingMsg, "drinks", 6) == 0)
297.                {
298.                    sendMsg(clientInfo, "$Server: sei entrato in drink.\n");
299.
300.
301.
302.
303.
304.
305.
306.
307.
308.
309.
310.
311.                    riprova!");
312.
313.
314.
315.                }
316.                memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
317.
318.                if (exited)
319.                {
320.                    sendMsg(clientInfo, "$Server: Torna presto! Disconnessione in corso... \n");
321.                }
322.                disconnectionManagement(clientInfo);
323.            }
324.
325. /**
326. * @param arg : NULL.
327. *
328. * @return: puntatore a void.
329. *
330. * La funzione connectionRequestsManagement gestisce le operazioni di
331. * connessione alla socket del server da parte dei client.
332. *
333. */
334.
335. void *connectionRequestsManagement(void *arg)
336. {
337.
338.     int connectSocket;
339.     pthread_t tidListenerClient;
340.     struct sockaddr_in clientAddress;
341.     char logsBuffer[BUFFER_STRLEN];
342.     LpClientInfo clientInfo;
343.     char clientAddressIPv4[INET_ADDRSTRLEN];
344.     unsigned int clientAddressSize = sizeof(clientAddress);
345.
346.     while (true)
347.     {
348.
349.         /* Accetto le richieste di connessione da parte dei client */

```

```

350.         if ((connectSocket = accept(listenerSocket, (struct sockaddr *) &clientAddress,
351.                                     &clientAddressSize)) == -1)
352.         {
353.             printf("<!> Impossibile accettare richiesta di connessione");
354.         }
355.     }
356.     inet_ntop(AF_INET, &clientAddress, clientAddressIPv4, INET_ADDRSTRLEN);
357.     clientInfo = NULL;
358.   }
359. /* Creo una struttura clientInfo che conterrà tutte le informazioni riguardo al client */
360.   if ((clientInfo = newClientInfo(connectSocket, clientAddressIPv4, "")) == NULL)
361.   {
362.       printf("<!> Impossibile allocare memoria, %s disconnesso.", clientAddressIPv4);
363.   }
364.   else
365.   {
366.
367.       /*Aggiorno le strutture che contengono informazioni sullo status attuale */
368.       pthread_mutex_lock(&mutexClientInfo);
369.       insertClientInfo(&listClientInfo, clientInfo);
370.       pthread_mutex_unlock(&mutexClientInfo);
371.
372.       /* Crea un thread che gestirà l'accesso del client al Server */
373.       pthread_create(&tidListenerClient, NULL, listenerClient, clientInfo);
374.       clientInfo->tidHandler = tidListenerClient;
375.
376.       printf("Nuova connessione accettata:[Client: %s] - %s", clientAddressIPv4,
377.              ctime(&(clientInfo->timestamp)));
378.       pthread_mutex_lock(&mutexCursor);
379.       totalConnections += 1;
380.       printf(" \u25cf Total CN: %7.2d", totalConnections);
381.       connectedClients += 1;
382.       printf(" \u25cf Connected: %6.2d\n", connectedClients);
383.       pthread_mutex_unlock(&mutexCursor);
384.
385.       /* scrittura file di log */
386.
387.       pthread_mutex_lock(&mutexLogs);
388.       sprintf(logsBuffer, " > Nuova connessione accettata:[Client: %s] - %s",
389.               clientAddressIPv4, ctime(&(clientInfo->timestamp)));
390.       if (write(logs, logsBuffer, strlen(logsBuffer)) == -1)
391.       {
392.           printf("Errore durante la scrittura nel file di logs.");
393.       }
394.       pthread_mutex_unlock(&mutexLogs);
395.   }
396.
397. /**
398. * @param clientinfo: Nodo ClientInfo con informazioni sul client.
399. *
400. * @return: 1 in caso di errore, 0 altrimenti.
401. *
402. * La funzione signin gestisce il signin da parte
403. * del client al Server. Verifica inoltre che l'username del client
404. * sia univoco.
405. *
406. */
407.
408. bool signIn(LpClientInfo clientInfo)
409. {
410.
411.     char username[CLTINF_USERNAME_STRLEN]; /* Buffer per username del client */
412.     char password[CLTINF_PASSWORD_STRLEN]; /* Buffer per password del client */
413.     char incomingMsg[INCOMING_MSG_STRLEN]; /* Buffer per messaggi in entrata */
414.     char record[GRAPHICS_CHAT_WIDTH]; /* Buffer per la lettura di un record del DB */
415.     char msg[GRAPHICS_CHAT_WIDTH];
416.     char character[1]; /* Array per la lettura dei dati dal DB */
417.     char *usernameDB; /* Username estratto dal DB */
418.     bool passed = false; /* Flag che indica il superamento di una fase */
419.     int recordIndex; /* Indice di posizione del buffer record */
420.     int bytesReaded; /* Numero di bytes letti dalla read */
421.     time_t timestamp;
422.     char logsBuffer[BUFFER_STRLEN];
423.
424.     memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
425.     //sendMsg(clientInfo, "$Server: Inserisci l'username - massimo 10 caratteri.");
426.
427.     do {
428.         passed = true;
429.         /* Leggo l'username inserito dal client */
430.         /* Verifico se l'utente ha inserito il comando di uscita */
431.         /* Controllo se lo username supera la lunghezza di 10 caratteri stabiliti */
432.         /* Controllo se nell'username siano presenti caratteri di spazio */
433.         if ((bytesReaded = read(clientInfo->clientSocket, incomingMsg, INCOMING_MSG_STRLEN)) <=
434.             0)
435.         {
436.             return false;
437.         }

```

```

437.             incomingMsg[bytesReaded] = '\0';
438.
439.             if (!strcmp(incomingMsg, "exit"))
440.             {
441.                 return true;
442.             }
443.             else
444.             {
445.
446.                 if (strlen(incomingMsg) > CLTINF_USERNAME_STRLEN)
447.                 {
448.                     sendMsg(clientInfo, "se1\n"); //username troppo lungo -[massimo 10
449.                     caratteri]
450.                 }
451.                 else
452.                 {
453.
454.                     for (int i = 0; i < strlen(incomingMsg); i++)
455.                     {
456.                         if (incomingMsg[i] == ' ')
457.                         {
458.                             sendMsg(clientInfo, "se2\n"); //gli spazi non sono
459.                             consentiti
460.                         }
461.                     }
462.                 }
463.             /* Nel caso in cui l'username fosse ancora valido proseguo con le
464.             verifiche */
465.
466.
467.
468.
469.
470.
471.
472.
473.             esista già */
474.             viene trovato un newline oppure \0 */
475.
476.
477.
478.
479.
480.             un fine stringa? */
481.
482.
483.
484.
485.
486.
487.
488.
489.
490.
491.             nel database, se sì, stampo un errore e riclico il do */
492.
493.
494.             "se3\n"); //username non disponibile
495.
496.
497.
498.
499.             appoggio */
500.             GRAPHICS_CHAT_WIDTH);
501.
502.
503.
504.
505.
506.
507.
508.
509.
510.
511.             Database.");
512.
513.
514.             dell'impossibilità di catturare */

```

```

515.     /* l'exit status del thread chiamante signin (numero
516.        di utenti non deterministico). */
517.     /* lista di tid, tuttavia */
518.     /* costosa, considerando che in certi */
519.     /*
520.      */
521.      /*
522.       */
523.       /*
524.        */
525.        /*
526.         */
527.         /*
528.          */
529.          /*
530.           */
531.           /*
532.            */
533.            /*
534.             */
535.             /*
536.              */
537.              /*
538.               */
539.               /*
540.                 */
541.                 /*
542.                  */
543.                  /*
544.                   */
545.                   /*
546.                     */
547.                     /*
548.                      */
549.                      /*
550.                        */
551.                        /*
552.                          */
553.                          /*
554.                            */
555.                            /*
556.                              */
557.                              /*
558.                                */
559.                                /*
560.                                  */
561.                                  /*
562.                                    */
563.                                    /*
564.                                      */
565.                                      /*
566.                                        */
567.                                        /*
568.                                          */
569.                                          /*
570.                                            */
571.                                            /*
572.                                              */
573.                                              /*
574.                                                */
575.                                                /*
576.                                                 */
577.                                                 /*
578.                                                   */
579.                                                   /*
580.                                                     */
581.                                                     /*
582.                                                       */
583.                                                       /*
584.                                                        */
585.                                                        /*
586.                                                          */
587.                                                          /*
588.                                                            */
589.                                                            /*
590.                                                              */
591.                                                              /*
592.                                                                */
593.                                                                /*
594.                                                                  */
595.                                                                  /*
596.                                                                    */
597.                                                                    /*
598.                                                                      */
599.                                                                      */

/* l'exit status del thread chiamante signin (numero
   Chiaramente una soluzione sarebbe potuta essere una
   /* sarebbe stata un'implementazione "inutilmente" costosa, considerando che in certi
   /* casi la terminazione è obbligatoria.
   pthread_kill(pthread_self(), SIGTERM);
}
else if (passed != false)
{
    strcpy(username, incomingMsg);
}
pthread_mutex_unlock(&mutexDatabase);

}

memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);

} while (passed != true);

sendMsg(clientInfo, "$Server: In attesa della password \n");
memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);

do
{
    passed = true;
    /* Leggo la password inserita dal client */
    if ((bytesReaded = read(clientInfo->clientSocket, incomingMsg, INCOMING_MSG_STRLEN)) <=
0)
    {
        return false;
    }
    incomingMsg[bytesReaded] = '\0';

    /* Verifico se l'utente ha inserito il comando di uscita */
    if (!strcmp(incomingMsg, "exit"))
    {
        return true;
    }
    else
    { /* Controllo se la password supera la lunghezza di 10 caratteri stabiliti */
        if (strlen(incomingMsg) > CLTINF_PASSWORD_STRLEN)
        {
            sendMsg(clientInfo, "se4\n"); //pass troppo lunga
            passed = false;
        }
        else
        { /* Controllo che nella password non siano presenti caratteri di spazio */
            for (int i = 0; i < strlen(incomingMsg); i++)
            {
                if (incomingMsg[i] == ' ')
                {
                    sendMsg(clientInfo, "se5\n"); //pass con spazi
                    passed = false;
                    break;
                }
            }
            if (passed != false)
            {
                strcpy(password, incomingMsg);
            }
        }
        memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
    }
} while (passed != true);

memset(record, '\0', GRAPHICS_CHAT_WIDTH);
/*Concateno username e password in un solo buffer */
sprintf(record, "%s %s\n", username, password);

pthread_mutex_lock(&mutexDatabase);

/* Scrivo i dati di registrazione del client nel database */
if (write(database, record, strlen(record)) == -1)
{
    printf("<!> Impossibile scrivere dati nel Database.");
    sleep(2);
    /* La terminazione è necessariamente bruta a causa dell'impossibilità di catturare */
    /* l'exit status del thread chiamante signin (numero di utenti non deterministico) */
    /* Chiaramente una soluzione sarebbe potuta essere una lista di tid, tuttavia */
    /* sarebbe stata un'implementazione "inutilmente" costosa, considerando che in certi */
    /* casi la terminazione è obbligatoria. */
    pthread_kill(pthread_self(), SIGTERM);
}
pthread_mutex_unlock(&mutexDatabase);
/* Registrazione effettuata con successo */
timestamp = time(NULL);
pthread_mutex_lock(&mutexLogs);
sprintf(logsBuffer, " > Registrazione del client %s avvenuta con successo - %s", clientInfo->clientAddressIPv4, ctime(&(clientInfo->timestamp)));

```

```

600.         if (write(logs, logsBuffer, strlen(logsBuffer)) == -1)
601.     {
602.         printf("Errore durante la scrittura nel file di logs.");
603.     }
604.     pthread_mutex_unlock(&mutexLogs);
605.     printf("Registrazione del client %s avvenuta con successo.", clientInfo->clientAddressIPv4);
606.
607.     sendMsg(clientInfo, "seok\n"); //notifico la registrazione avvenuta con successo
608.
609.
610.     return false;
611. }
612.
613.
614. /**
615. * @param clientinfo: Nodo ClientInfo con informazioni sul client.
616. *
617. * @return: true in caso di errore, false altrimenti.
618. *
619. * La funzione login gestisce il login da parte
620. * del client al Server.
621. *
622. */
623.
624. bool login(LpClientInfo clientInfo)
625. {
626.
627.     bool passed = false;
628.     char incomingMsg[INCOMING_MSG_STRLEN];
629.     char record[GRAPHICS_CHAT_WIDTH];
630.     char username[CLTINF_USERNAME_STRLEN];
631.     char password[CLTINF_PASSWORD_STRLEN];
632.     char incomingRecord[GRAPHICS_CHAT_WIDTH];
633.     int recordIndex;
634.     char character[1];
635.     int bytesReaded;
636.
637.     bool alreadyLogged;
638.
639.     char logsBuffer[BUFFER_STRLEN];
640.
641.     do {
642.         //sendMsg(clientInfo, "$Server: In attesa dell'username. \n");
643.         memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
644.
645.         do {    sendMsg(clientInfo, "$Server: In attesa dell'username. \n");
646.             passed = true;
647.             /*Leggo l'username inserito dal client */
648.             if ((bytesReaded = read(clientInfo->clientSocket, incomingMsg,
649.             INCOMING_MSG_STRLEN)) <= 0)
650.             {
651.                 return false;
652.             }
653.             incomingMsg[bytesReaded] = '\0';
654.             if (!strcmp(incomingMsg, "exit"))
655.             {
656.                 return true;
657.             }
658.             else
659.             {
660.                 if (strlen(incomingMsg) > CLTINF_USERNAME_STRLEN)
661.                 {
662.                     sendMsg(clientInfo, "se1\n"); //errore username troppo
663.                     passed = false;
664.                 }
665.                 else
666.                 {
667.                     for (int i = 0; i < strlen(incomingMsg); i++)
668.                     {
669.                         if (incomingMsg[i] == ' ')
670.                         {
671.                             sendMsg(clientInfo, "se2\n"); //errore
672.                             passed = false;
673.                             break;
674.                         }
675.                     }
676.                     if (passed != false)
677.                     {
678.                         strcpy(username, incomingMsg);
679.                     }
680.                 }
681.                 memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
682.             }
683.         } while (passed != true);
684.
685.         //sendMsg(clientInfo, "$Server: In attesa della password. \n");
686.         memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
687.

```

```

688.         do {    sendMsg(clientInfo, "$Server: In attesa della password. \n");
689.             //puts("devo leggere la password");
690.             passed = true;
691.             /* Leggo la password inserita dal client */
692.             if ((bytesReaded = read(clientInfo->clientSocket, incomingMsg,
693.             INCOMING_MSG_STRLEN)) <= 0)
694.             {
695.                 return false;
696.             }
697.             incomingMsg[bytesReaded] = '\0';
698.             /* Verifico se l'utente ha inserito il comando di uscita */
699.             if (!strcmp(incomingMsg, "exit"))
700.             {
701.                 return true;
702.             }
703.             else
704.             { /* Controllo se la password supera la lunghezza di 10 caratteri stabiliti */
705.                 if (strlen(incomingMsg) > CLTINF_PASSWORD_STRLEN)
706.                 {
707.                     sendMsg(clientInfo, "se3\n");           //errore lunghezza password
708.                     passed = false;
709.                 }
710.                 else
711.                 { /* Controllo che nella password non siano presenti caratteri di
712.                     spazio */
713.                     password contenente spazi
714.                     for (int i = 0; i < strlen(incomingMsg); i++)
715.                     {
716.                         if (incomingMsg[i] == ' ')
717.                         {
718.                             sendMsg(clientInfo, "se4\n");           //errore
719.                             passed = false;
720.                             break;
721.                         }
722.                         if (passed != false)
723.                         {
724.                             strcpy(password, incomingMsg);
725.                         }
726.                         memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
727.                     }
728.                 } while (passed != true);
729.
730.                 memset(incomingRecord, '\0', CLTINF_USERNAME_STRLEN *2);
731.                 sprintf(incomingRecord, "%s %s", username, password);
732.                 pthread_mutex_lock(&mutexDatabase);
733.                 lseek(database, 0, SEEK_SET);
734.                 memset(record, '\0', GRAPHICS_CHAT_WIDTH);
735.                 recordIndex = 0;
736.
737.                 /* Ciclo sul file database per verificare che lo username non esista già */
738.                 do { /* La lettura avviene un carattere alla volta fintanto non viene trovato un newline
739.                     */
740.                     alreadyLogged = false;
741.                     if ((bytesReaded = read(database, character, 1)) > 0)
742.                     { /* Il carattere è diverso da un newline? */
743.                         if (character[0] != '\n')
744.                         {
745.                             record[recordIndex++] = character[0];
746.                         }
747.                         else
748.                         {
749.                             if (!strcmp(record, incomingRecord))
750.                             {
751.                                 pthread_mutex_lock(&mutexClientInfo);
752.                                 LpClientInfo tmp = listClientInfo;
753.                                 while (tmp != NULL && alreadyLogged == false)
754.                                 {
755.                                     if ((tmp->status == CLTINF_LOGGED) && !
756.                                         strcmp(tmp->username, username) && tmp != clientInfo)
757.                                     {
758.                                         errore di connessione al server
759.                                         sleep(1);
760.                                         alreadyLogged = true;
761.                                     }
762.                                     tmp = tmp->nextClientInfo;
763.                                 }
764.                                 pthread_mutex_unlock(&mutexClientInfo);
765.                                 if (alreadyLogged == false)
766.                                 {
767.                                     clientInfo->status = CLTINF_LOGGED;
768.                                     strcpy(clientInfo->username, username);
769.                                     pthread_mutex_lock(&mutexLogs);
770.                                     sprintf(logsBuffer, "> Login effettuato con
successo:[Client: %s - %s] - %s", clientInfo->clientAddressIPv4, clientInfo->username,
ctime(&(clientInfo->timestamp)));
771.                                 }
772.                             }
773.                         }
774.                     }
775.                 }
776.             }
777.         }
778.     }
779. }
```

```

771.     strlen(logsBuffer)) == -1)
772.     nel file di logs.");
773.
774.     effettuato con successo
775.
776.     }
777.
778.     }
779.
780.     }
781.     }
782.     }
783.     }
784.     }
785.     }
786.     }
787.     }
788.     }
789.     }
790.     }
791.     }
792.     }
793.     }
794.     }
795.     }
796.     }
797.     }
798.     return false;
799.
800. }
801.
802. /**
803. * @param void.
804. *
805. * @return: return 1 in caso di errore e 0 altrimenti.
806. *
807. * La funzione initFile apre i file di logs, database.
808. *
809. */
810.
811. int initFile(void)
812. {
813.
814.     /* Apro il file di log */
815.     if ((logs = open("logs.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRWXU)) == -1)
816.     {
817.         printf("%d", logs);
818.         return 1;
819.     }
820.
821.     /* Apro il file di database */
822.     if ((database = open("database.txt", O_RDWR | O_CREAT | O_APPEND, S_IRWXU)) == -1)
823.     {
824.
825.         return 1;
826.     }
827.
828.     return 0;
829.
830. }
831.
832. /**
833. * @param clientInfo: puntatore della struttura che contiene informazioni
834. * del client.
835. * @param outgoingMsg: Messaggio da scrivere da scrivere al client.
836. *
837. * @return: void.
838. *
839. * La funzione sendMsg inoltra il messaggio catturato
840. * dal listenerTextField al client.
841. *
842. */
843.
844. void sendMsg(LpClientInfo clientInfo, char *outcomingMsg)
845. {
846.
847.     char buffer[OUTCOMING_MSG_STRLEN];
848.
849.     memset(buffer, '\0', OUTCOMING_MSG_STRLEN);
850.     strcpy(buffer, outcomingMsg);
851.     pthread_mutex_lock(&(clientInfo->mutexSocket));
852.     write(clientInfo->clientSocket, buffer, OUTCOMING_MSG_STRLEN);
853.     pthread_mutex_unlock(&(clientInfo->mutexSocket));
854. }
855.
856. /**
857. * @param signal: segnale catturato
858. */

```

```

859.     * @return void.
860.     *
861.     * La funzione signalHandler cattura i segnali in questione
862.     * e ne detta il comportamento al loro verificarsi.
863.     *
864.     */
865.
866. void signalHandler(int signal)
867. {
868.     LpClientInfo tmp;
869.     switch (signal)
870.     {
871.         case SIGINT:
872.             printf("Disconnessione del server in corso!");
873.             close(listenerSocket);
874.             close(logs);
875.             close(database);
876.             tmp = listClientInfo;
877.             while (tmp != NULL)
878.             {
879.                 close(tmp->clientSocket);
880.                 tmp = tmp->nextClientInfo;
881.             }
882.             sleep(2);
883.             raise(SIGTERM);
884.             break;
885.         case SIGSTOP:
886.             printf("Disconnessione del server in corso!");
887.             close(listenerSocket);
888.             close(logs);
889.             close(database);
890.             tmp = listClientInfo;
891.             while (tmp != NULL)
892.             {
893.                 close(tmp->clientSocket);
894.                 tmp = tmp->nextClientInfo;
895.             }
896.             sleep(2);
897.             raise(SIGTERM);
898.             break;
899.     }
900. }
901. **** END SERVER ****

```

Dettagli Implementativi del Server

- La funzione ConnectionRequestManagement [Server]

La funzione ConnectionRequestManagement, eseguita da un thread, gestisce le connessioni in entrata sulla

porta nella quale il Server è in ascolto. Il compito principale di questa funzione è quello di allocare un nuovo nodo ClientInfo (un nodo ClientInfo è una struttura contenente diverse informazioni riguardanti il client) e creare un thread per l'ascolto dei messaggi in arrivo il quale esegue la funzione listenerClient.

Le informazioni del nodo ClientInfo sono:

```
1. struct ClientInfo
2. {
3.
4.     pthread_t tidHandler;
5.     int clientSocket;
6.     char clientAddressIPv4[INET_ADDRSTRLEN];
7.     char username[CLTINF_USERNAME_STRLEN];
8.     time_t timestamp;
9.     int status;
10.    pthread_mutex_t mutexSocket;
11.
12.    //lista doppiamente puntata
13.    struct ClientInfo* prevClientInfo;
14.    struct ClientInfo* nextClientInfo;
15. };
16.
17. typedef struct ClientInfo ClientInfo;
18. typedef ClientInfo* LpClientInfo;
19.
```

Essenzialmente il corpo della funzione è un ciclo infinito, il quale rimane in attesa di richieste di connessione al Server. Quando un Client si connette al Server, la funzione prova ad allocare, come detto in precedenza, un nodo ClientInfo, lo inserisce nella lista dei Clients connessi al Server e inizializza il thread di gestione associato al Client.

Oltre a fare ciò, ogni qualvolta un Client effettua con successo la connessione al Server, la funzione ConnectionRequestManagement aggiorna i vari status del Server indicando connessione/i totale/i e i client attualmente connessi.

```
1. pthread_mutex_lock(&mutexClientInfo);
2. insertClientInfo(&listClientInfo, clientInfo);
3. pthread_mutex_unlock(&mutexClientInfo);
4.
5. /* Creo un thread che gestirà l'accesso del client al Server */
6. pthread_create(&tidListenerClient, NULL, listenerClient, clientInfo);
7. clientInfo->tidHandler = tidListenerClient;
8.
9. printf("Nuova connessione accettata:[Client: %s] - %s", clientAddressIPv4, ctime(&(clientInfo->timestamp)));
10.
```

○ La funzione disconnectionManagement [Server]

La funzione disconnectionManagement gestisce la disconnessione di un Client dal Server. In particolare si occupa della deallocazione del nodo ClientInfo e della chiusura della socket.

```
1. void disconnectionManagement(LpClientInfo clientInfo)
2. {
3.     time_t timestamp = time(NULL);
4.     char logsBuffer[BUFFER_STRLEN];
5.     pthread_t tidHandler;
6.
7.
8.     sprintf(logsBuffer, " > Il client %s si è disconnesso dal Server - %s", clientInfo-
>clientAddressIPv4, ctime(&timestamp));
9.     printf("\n > Il client %s si è disconnesso dal Server - %s", clientInfo->clientAddressIPv4,
ctime(&timestamp));
10.
11.    pthread_mutex_lock(&mutexLogs);
12.    if (write(logs, logsBuffer, strlen(logsBuffer)) == -1)
13.    {
14.        printf("Errore durante la scrittura nel file di logs.");
15.    }
16.    pthread_mutex_unlock(&mutexLogs);
17.
```

```

18.     pthread_mutex_lock(&mutexCursor);
19.     connectedClients -= 1;
20.     pthread_mutex_unlock(&mutexCursor);
21.
22.     pthread_mutex_lock(&mutexClientInfo);
23.     /* Chiudo la socket di comunicazione */
24.     tidHandler = clientInfo->tidHandler;
25.     close(clientInfo->clientSocket);
26.     /* Elimino dalla lista dei client, il client disconnesso */
27.     deleteClientInfo(&listClientInfo, &clientInfo);
28.     pthread_mutex_unlock(&mutexClientInfo);
29.     /* Elimino il thread */
30.     pthread_exit(tidHandler);
31.
32. }
33.

```

○ La funzione creaSocket() [Server]

Questa funzione definisce una socket da utilizzare per il protocollo di TCP/IP. Dopo di che, viene richiesta la porta sulla quale si vuole avviare la socket all'utente. Si prosegue con la verifica sulla disponibilità della porta e che la porta inserita sia corretta, e successivamente si usa la funzione socket() che permette di creare un unbound socket del dominio PF_INET(TCP/IP) e restituisce un file descriptor, con il quale possiamo proseguire con la chiamata al metodo bind, che associa al socket locale definito dal file descriptor, un socket descriptor definito da una struttura come sockaddr_in; infine, con il metodo listen(), il server si mette in attesa di richieste da parte del client.

Dopo che la creazione della socket è andata a buon fine, il server avvia un thread che esegue il metodo connectionRequestsManagement(), quest'ultimo accetta la richiesta con il metodo accept() e crea un processo figlio a cui affidare la gestione delle richieste ricevute dal client.

```

1. int createSocket()
2. {
3.
4.     int bytesReaded;
5.     char buffer[BUFFER_STRLEN];
6.     int port;
7.     bool error;
8.     struct sockaddr_in serverAddress;
9.
10.    memset(&serverAddress, '0', sizeof(serverAddress));
11.    serverAddress.sin_family = PF_INET;
12.    serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
13.
14.    //inserire porta
15.    printf("\n » Inserire la porta del server: ");
16.    fflush(stdout);
17.
18.    do {
19.
20.        error = false;
21.        if ((bytesReaded = read(STDIN_FILENO, buffer, BUFFER_STRLEN)) == -1)
22.        {
23.            perror("\n <!> Errore read");
24.
25.            return 1;
26.        }
27.        buffer[bytesReaded] = '\0';
28.
29.        /* Verifico se la porta fornita è valida */
30.        for (int i = 0; i < strlen(buffer) - 1; i++){
31.            if (buffer[i] < '0' || buffer[i] > '9')
32.            {
33.                error = true;
34.                break;
35.            }
36.        }
37.
38.        if (error){
39.
40.            printf("\n<!> Porta non valida, caratteri non ammessi - riprovare: ");
41.
42.            fflush(stdout);
43.
44.        }else{
45.            if ((port = atoi(buffer)) == 0 || !(port >= 0 && port <= 65535)){
46.                printf("\n<!> Porta non valida, out of range - riprovare: ");
47.
48.                fflush(stdout);
49.                error = true;
50.            }
51.
52.        }
53.
54.    }
55. }
56.

```

```

50.
51.        }else{
52.            listenerPort = port;
53.
54.            printf("\n      > Porta %d inserita con successo.\n", port);
55.
56.            fflush(stdout);
57.            serverAddress.sin_port = htons(atoi(buffer));
58.
59.        }
60.
61.        if (!error){
62.
63.            listenerSocket = socket(PF_INET, SOCK_STREAM, 0);
64.
65.            /* Assegno un'indirizzo alla socket del server */
66.            if (bind(listenerSocket, (struct sockaddr *) &serverAddress, sizeof(serverAddress)) == -1){
67.
68.                perror("\n<!> Errore bind");
69.                puts("");
70.
71.                sleep(1);
72.                error = true;
73.            }
74.            printf("bind riuscito");
75.
76.            /*Rimango in ascolto di richieste di connessione da parte di client */
77.
78.            if (listen(listenerSocket, LISTENER_QUEUE_STRLEN) == -1){
79.
80.                perror("\n<!> Errore listen");
81.                puts("");
82.
83.                sleep(1);
84.                return 1;
85.
86.            }
87.        }
88.
89.    } while (error != false);
90.
91.    return 0;
92.
93. }
94.

```

○ La funzione signalHandler() [Server]

E' una funzione che gestisce i segnali **SIGINT** e **SIGSTOP**, dove quando viene ricevuto uno dei due segnali, si procede con la chiusura della socket, dei file Database.txt e Logs.txt, e infine la funzione termina inviando un segnale di **SIGTERM** che procede con la terminazione del programma.

```

1. void signalHandler(int signal)
2. {
3.
4.     LpClientInfo tmp;
5.
6.     switch (signal)
7.     {
8.         case SIGINT:
9.             printf("Disconnessione del server in corso!");
10.
11.            close(listenerSocket);
12.            close(logs);
13.            close(database);
14.
15.            tmp = listClientInfo;
16.            while (tmp != NULL) {
17.
18.                close(tmp->clientSocket);
19.                tmp = tmp->nextClientInfo;
20.
21.            }
22.            sleep(2);
23.
24.            raise(SIGTERM);
25.            break;
26.
27.         case SIGSTOP:
28.             printf("Disconnessione del server in corso!");
29.
30.             close(listenerSocket);
31.             close(logs);
32.             close(database);
33.
34.             tmp = listClientInfo;

```

```

35.         while (tmp != NULL){
36.
37.             close(tmp->clientSocket);
38.             tmp = tmp->nextClientInfo;
39.
40.         }
41.         sleep(2);
42.         raise(SIGTERM);
43.         break;
44.     }

```

○ La funzione signIn() [Server]

Questo metodo viene richiamato dalla funzione listenerClient(), che non è altro che un thread che gestisce le richieste pervenute dal client (ogni client ne ha uno assegnato).

Tale funzione signIn(), permette di ricevere informazioni inserite dall'utente attraverso il client, e eventualmente notifica quest'ultimo attraverso messaggi di errore che verranno poi interpretati dal/dai client.

```

1.   bool signIn(LpClientInfo clientInfo)
2.   {
3.
4.       char username[CLTINF_USERNAME_STRLEN];
5.       char password[CLTINF_PASSWORD_STRLEN];
6.       char incomingMsg[INCOMING_MSG_STRLEN];
7.       char record[GRAPHICS_CHAT_WIDTH];
8.       char msg[GRAPHICS_CHAT_WIDTH];
9.       char character[1];
10.      char *usernameDB;
11.      bool passed = false;
12.      int recordIndex;
13.      int bytesReaded;
14.      time_t timestamp;
15.      char logsBuffer[BUFFER_STRLEN];
16.
17.      memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
18.
19.      /* Ciclo finchè il client non inserisce un username che rispetti le condizioni di lunghezza, e che non sia già esistente */
20.      do {
21.          passed = true;
22.
23.          /*Leggo l'username inserito dal client */
24.          /*Verifico se l'utente ha inserito il comando di uscita */
25.          /*Controllo se lo username supera la lunghezza di 10 caratteri stabiliti */
26.          /*Controllo se nell'username siano presenti caratteri di spazio */
27.
28.          if ((bytesReaded = read(clientInfo->clientSocket, incomingMsg, INCOMING_MSG_STRLEN)) <= 0)
29.          {
30.              return false;
31.          }
32.          incomingMsg[bytesReaded] = '\0';
33.
34.          /* Verifico se l'utente ha inserito il comando di uscita */
35.
36.          if (!strcmp(incomingMsg, "exit"))
37.          {
38.              return true;
39.          }
40.          /* Controllo se lo username supera la lunghezza di 10 caratteri stabiliti */
41.
42.          if (strlen(incomingMsg) > CLTINF_USERNAME_STRLEN) {
43.              sendMsg(clientInfo, "se1\n");           //username troppo lungo -[massimo 10 caratteri]
44.              passed = false;
45.          }
46.          /* Controllo se nell'username siano presenti caratteri di spazio */
47.
48.          for (int i = 0; i < strlen(incomingMsg); i++){
49.
50.              if (incomingMsg[i] == ' '){
51.
52.                  sendMsg(clientInfo, "se2\n"); //gli spazi non sono consentiti
53.
54.                  passed = false;
55.                  break;
56.
57.              }
58.          }
59.
60.          /* Nel caso in cui l'username fosse ancora valido proseguo con le verifiche */
61.          if (passed != false){
62.
63.              pthread_mutex_lock(&mutexDatabase);
64.              lseek(database, 0, SEEK_SET);
65.

```

```

66.             memset(record, '\0', GRAPHICS_CHAT_WIDTH);
67.             recordIndex = 0;
68.
69.             /* Ciclo sul file database per verificare che lo username non esista già */
70.
71.             do {
72.
73.                 /* La lettura avviene un carattere alla volta fintanto non viene trovato un newline oppure \0 */
74.
75.                 if ((bytesReaded = read(database, character, 1)) > 0){
76.
77.                     /*Il carattere è diverso da un newline oppure un fine stringa? */
78.
79.                     if (character[0] != '\n'){
80.
81.                         record[recordIndex++] = character[0];
82.
83.                     }else{
84.
85.                         record[recordIndex] = '\0';
86.                         usernameDB = strtok(record, " ");
87.
88.                         /* Controllo se l'username esista già nel database, se sì, stampo un errore e riclico il do */
89.
90.                         if (!strcmp(usernameDB, incomingMsg)){
91.
92.                             sendMsg(clientInfo, "se3\n");           //username non disponibile
93.
94.                             passed = false;
95.                             break;
96.
97.                         } else { /* Ripulisco le strutture di appoggio */
98.
99.                             memset(record, '\0', GRAPHICS_CHAT_WIDTH);
100.                            recordIndex = 0;
101.                        }
102.                    }
103.                }
104.            } while (bytesReaded != 0);
105.
106.            /* Verifico l'eventuale presenza di errori */
107.
108.            if (bytesReaded == -1){
109.
110.                sprintf(msg, "<!> Impossibile leggere dati dal Database.");
111.
112.                sleep(2);
113.
114.                /* La terminazione è necessariamente bruta a causa dell'impossibilità di catturare      */
115.                /* l'exit status del thread chiamante signin (numero di utenti non deterministico).      */
116.                /* Chiaramente una soluzione sarebbe potuta essere una lista di tid, tuttavia          */
117.                /* sarebbe stata un'implementazione "inutilmente" costosa, considerando che in certi   */
118.                /* casi la terminazione è obbligatoria.                                              */
119.
120.                pthread_kill(pthread_self(), SIGTERM);
121.
122.            }else if (passed != false){
123.
124.                strcpy(username, incomingMsg);
125.
126.            }
127.
128.        }
129.    }
130.
131.    memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
132. }
133.
134.
135. } while (passed != true);
136.
137. sendMsg(clientInfo, "$Server: In attesa della password \n");
138. memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
139.
140. do {
141.     passed = true;
142.     /* Leggo la password inserita dal client */
143.
144.     if ((bytesReaded = read(clientInfo->clientSocket, incomingMsg, INCOMING_MSG_STRLEN)) <= 0)
145.     {
146.         return false;
147.     }
148.
149.     incomingMsg[bytesReaded] = '\0';
150.     /* Verifico se l'utente ha inserito il comando di uscita */
151.
152.     if (!strcmp(incomingMsg, "exit")){
153.
154.         return true;
155.     }else{
156.     }

```

```

157.     /* Controllo se la password supera la lunghezza di 10 caratteri stabiliti */
158.     if (strlen(incomingMsg) > CLTINF_PASSWORD_STRLEN){
159.         sendMsg(clientInfo, "se4\n"); //pass troppo lunga
160.         passed = false;
161.     } else {
162.         /* Controllo che nella password non siano presenti caratteri di spazio */
163.         for (int i = 0; i < strlen(incomingMsg); i++)
164.         {
165.             if (incomingMsg[i] == ' ')
166.                 sendMsg(clientInfo, "se5\n"); //pass con spazi
167.                 passed = false;
168.                 break;
169.             }
170.         }
171.         if (passed != false){
172.             strcpy(password, incomingMsg);
173.         }
174.     }
175.     if (passed != false){
176.         strcpy(password, incomingMsg);
177.     }
178.     memset(incomingMsg, '\0', INCOMING_MSG_STRLEN);
179. }
180.
181. }
182. }
183. }
184.
185. } while (passed != true);
186.
187. memset(record, '\0', GRAPHICS_CHAT_WIDTH);
188.
189. /* Concateno username e password in un solo buffer */
190.
191. sprintf(record, "%s %s\n", username, password);
192.
193. pthread_mutex_lock(&mutexDatabase);
194.
195. /* Scrivo i dati di registrazione del client nel database */
196.
197. if (write(database, record, strlen(record)) == -1)
198. {
199.     printf("<!> Impossibile scrivere dati nel Database.");
200.
201.     sleep(2);
202.
203.
204.     /* La terminazione è necessariamente bruta a causa dell'impossibilità di catturare */
205.     /* l'exit status del thread chiamante signin (numero di utenti non deterministico) */
206.     /* Chiaramente una soluzione sarebbe potuta essere una lista di tid, tuttavia */
207.     /* sarebbe stata un'implementazione "inutilmente" costosa, considerando che in certi */
208.     /* casi la terminazione è obbligatoria. */
209.
210.     pthread_kill(pthread_self(), SIGTERM);
211.
212.
213.     pthread_mutex_unlock(&mutexDatabase);
214.
215.     /* Registrazione effettuata con successo */
216.
217.     timestamp = time(NULL);
218.     pthread_mutex_lock(&mutexLogs);
219.
220.     sprintf(logsBuffer, " > Registrazione del client %s avvenuta con successo - %s", clientInfo->clientAddressIPv4, ctime(&(clientInfo->timestamp)));
221.
222.     if (write(logs, logsBuffer, strlen(logsBuffer)) == -1)
223.     {
224.         printf("Errore durante la scrittura nel file di logs.");
225.     }
226.
227.     pthread_mutex_unlock(&mutexLogs);
228.     printf("Registrazione del client %s avvenuta con successo.", clientInfo->clientAddressIPv4);
229.
230.     sendMsg(clientInfo, "seok\n"); //notifico la registrazione avvenuta con successo
231.
232.
233.     return false;
234. }
235.

```

Dettagli Implementativi del Client

Il client dell'applicativo è stato realizzato su piattaforma **Android Studio** facendo uso:

- di un linguaggio object-oriented: **Java**;

- di un database orientato ai grafi: **Neo4J**.
-

○ [Android Studio e pattern](#)

Android Studio è un ambiente di sviluppo integrato (IDE) basato sul software di JetBrains IntelliJ IDEA per lo sviluppo di applicazioni Android. Le versioni di Android Studio sono compatibili con Linux, Windows e Macintosh. Android Studio rende lo sviluppo dell'applicazione più facile e veloce.

Altre caratteristiche messe a disposizione da Android Studio sono:

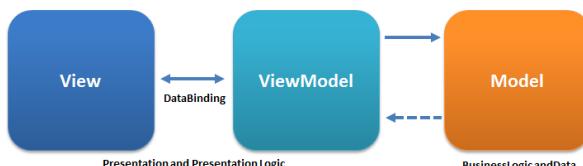
- editor per layout visuale utilizzabile in modalità drag and drop;
- accesso a SDK Manager e AVD Manager;
- inline debugging;
- monitoraggio delle risorse di memoria e della CPU utilizzate dall'applicazione.

Si è scelto di utilizzare Android Studio per la vastità di materiale e documentazione disponibile su internet. Inoltre dalle ricerche effettuate per la valutazione delle tecnologie da utilizzare, Android Studio è risultato quello più adatto per semplicità e chiarezza di utilizzo dell'interfaccia, nonché lo strumento più utilizzato a livello mondiale per lo sviluppo di applicazioni Android.

Per l'implementazione si è scelto di utilizzare un linguaggio Object Oriented, in particolare Java, con l'ausilio del design pattern “Model View ViewModel” (MVVM).

Il MVVM (Model View ViewModel) è un design pattern architettonale attraverso cui è possibile trarre vantaggio in termini di prestazioni, leggibilità e modularità del codice, si è preferito in quanto si adatta meglio all'ambiente Android.

Composto da tre moduli indipendenti:



- **Model** : è responsabile dell'astrazione delle origini dei dati. Model e ViewModel lavorano insieme per ottenere e salvare i dati.
- **ViewModel** : espone quei flussi di dati che sono rilevanti per la View. Inoltre, funge da collegamento tra il Model e la View.
- **View** : lo scopo di questo livello è informare il ViewModel sull'azione dell'utente. Questo livello osserva il ViewModel e non contiene alcun tipo di logica dell'applicazione.

○ [Neo4J](#)

Neo4j è un sistema di gestione per database a grafo nativo open-source, NoSql, che fornisce un back-end transazionale conforme ad **ACID**.

- **Transazionale** : con transazione si indica una qualunque sequenza di operazioni lecite che, se eseguita in modo corretto, produce una variazione nello stato di una base di dati.
- **ACID** : deriva dall'acronimo inglese **Atomicity, Consistency, Isolation e Durability**
 - **Atomicity** : il processo deve essere suddivisibile in un numero finito di unità invisibili, chiamate transazioni. L'esecuzione di una transazione deve essere per definizione o totale o nulla, e non sono ammesse esecuzioni parziali.
 - **Consistency** : la transazione deve rispettare i vincoli di integrità del database. Non devono verificarsi contraddizioni (incoerenza dei dati) tra i dati archiviati nel DB.
 - **Isolation** : ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni, l'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione
 - **Durability** : detta anche **Persistence**, si riferisce al fatto che una volta che una transazione abbia richiesto un commit work, i cambiamenti apportati non dovranno essere più persi.

Si è scelto l'utilizzo di un database a grafo a differenza di un database relazionale per tre circostanze di cui godono i database a grafo:

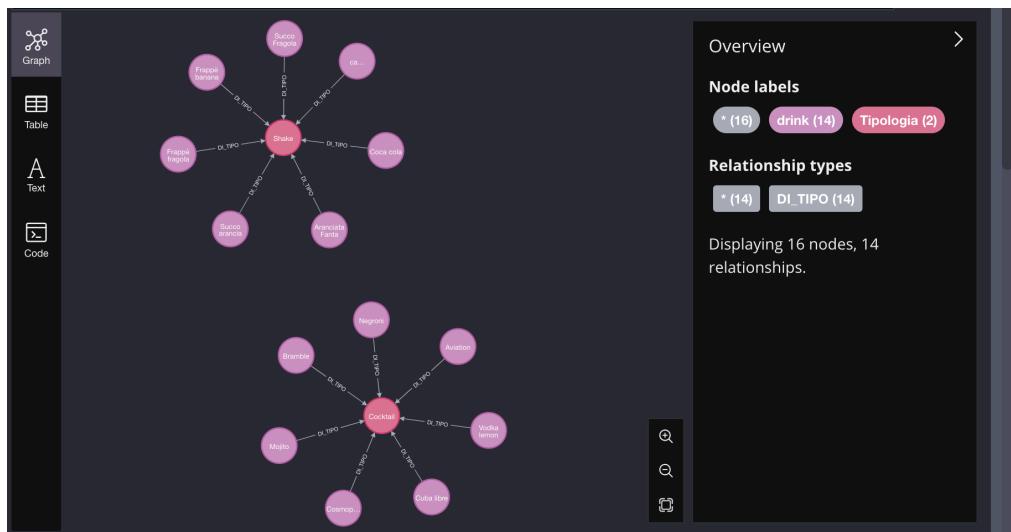
1. **Gestiscono meglio le relazioni** : Quando si tratta di gestire le relazioni, i database a grafo risultano essere ottimi per il modo in cui sono costruiti. Possono facilmente rappresentare relazioni complesse tra punti dati.
2. **Sono più scalabili** : Ciò significa che i database a grafo possono gestire grandi quantità di dati senza incorrere in problemi. Inoltre, poiché i volumi di dati continuano a crescere, questo è un fattore sempre più importante da considerare.
3. **Offrono una modellazione dei dati più flessibile** : Con un database a grafo si possono modellare i dati come meglio si preferisce, il che significa che non si è limitati alle rigide strutture di un database relazionale. Rappresentando i dati come una serie di nodi interconnessi, i database a grafo possono catturare in modo più accurato l'intricata rete di relazioni.

Per interagire con un database a grafo è possibile usufruire di due linguaggi, **Cypher** e **Gremlin**, progettati con obiettivi diversi e presentano alcune differenze in termini di sintassi, funzionalità e casi d'uso. Per la realizzazione del database dell'applicazione il linguaggio utilizzato in NEO4J è **Cypher**:

un linguaggio dichiarativo sviluppato da NEO4J, progettato per essere espressivo, flessibile e facile da usare, con una sintassi semplice e leggibile all'uomo che consente agli utenti di esprimere query complesse in modo conciso.

Si è preferito utilizzare tale linguaggio in quanto è ispirato a SQL, e di conseguenza consente di concentrarci su quali dati si desidera dal grafo (non su come ottenerli).

Per l'applicativo poiché risulta necessario che utenti possono selezionare i prodotto in base alla tipologia , sono realizzati due strutture una che prende il nome di “Cocktail” e l'altra che prende il nome di “Shake” contenenti i relativi drink con le loro informazioni.



Struttura del database a grafo dell'applicazione