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

RANDOM CHAT

Docente:
Prof. Giovanni SCALA

Autori:
Giovanni Falcone *N*86002329
giova.falcone@studenti.unina.it
Nicola Esposito *N*86002206
nicola.esposito15@studenti.unina.it

IdGruppo:
*LSO*_2122_21

Anno Accademico 2020/2021

Indice

Capitolo 1

Tutorial compilazione

1.1 Server

La struttura del server, ospitato sulla piattaforma Microsoft Azure è la seguente:

```
Server
├── Server.c
├── Room.c
├── Room.h
├── Mutex.c
├── Mutex.h
├── Global.h
├── AVL.c
├── AVL.h
├── CircularBuffer.c
├── CicularBuffer.h
├── serverConfig.txt
└── server.out
```

dove

- `Server.c` è il sorgente in cui viene inizializzato il server tramite le classiche funzioni dei socket, la funzione di avvio dei thread e la gestione della chat tra client
- Il file `Room.c` e il rispettivo header `Room.h`, contiene tutte le funzioni di gestione dei client relativi alla rispettiva room e delle funzioni di utility
- Il file `Mutex.c` e il rispettivo header, contengono le funzioni relative ai mutex e condition variable
- Il file `Globah.h`, come intuibile dal nome, contiene alcune variabili globali necessarie per diverse funzioni dei vari file
- Il file `AVL.c` e il rispettivo header `AVL.h`, contiene tutte le funzioni utili per la gestione dell'AVL
- Il file `CircularBuffer.c` e il rispettivo header `CircularBuffer.h`, contiene tutte le funzioni utili per la gestione della coda circolare

- `serverConfig.txt`: per evitare di dover compilare ogni volta il progetto si è fatto uso di un file di configurazione in cui è possibile, qualora si voglia, cambiare la capacità di una stanza e la porta su cui il server è in ascolto.
- Infine, `server.out` rappresenta l'eseguibile del server

Per la compilazione è necessario aprire la Powershell:

```
ssh azureuser@20.126.36.139
azureuser@20.126.36.139 password: Azureuser@123
# ci spostiamo alla cartella contentente il server
cd Server
# compiliamo i sorgenti
gcc -o server.out Server.c Room.c Room.h Mutex.c Mutex.h Global.h AVL↩
   .c AVL.h CircularBuffer.c CircularBuffer.h -pthread
```

1.2 Client

La struttura del client è semplicemente:

```
client
├── Client.c
├── Menu.c
├── Menu.h
├── clientConfig.txt
└── client.out
```

dove:

- `Client.c` è, ovviamente, il sorgente del client
- `Menu.c` e relativo header, rappresentano, appunto, il menù attraverso il quale il client potrà scegliere a quale stanza collegarsi oppure conoscere il numero di persone attualmente collegate in una certa stanza
- `clientConfig.txt`, ossia il file di configurazione in cui è possibile modificare l'indirizzo IP a cui il client si collegherà
- Infine `client.out` che rappresenta l'eseguibile

La compilazione, su terminale Unix, è banale:

```
cd client
# compiliamo i sorgenti
gcc -o client.out Client.c Menu.c Menu.h
```

Capitolo 2

Protocollo di comunicazione

Nel momento in cui i client si trovano nella stanza, il server deve accoppiare in modo randomico questi ultimi affinchè possa avvenire la chat tra i due client. La condizione è che ci debbano essere almeno due client nella stanza, altrimenti la comunicazione, ovviamente, non potrà avvenire. Gli utenti di una stanza che sono in attesa di essere accoppiati dal sistema si trovano nello stato di **W**(waiting), **T**(talking) altrimenti.

2.1 Comandi client

Il client ha a disposizione tre comandi:

- `@stop`, per stoppare la comunicazione con un altro client. Una volta digitato entrambi i client, eventualmente, verranno accoppiati con altri client all'interno della stanza.

```
#client "enzo" dopo l'avvio di una chat con un altro client

Now you can chat!
You are connected with nicola

nicola: ciao da nicola
ciao da enzo
nicola: @stop

******
Wait for chat
Digit @exit if you want to exit

```

```
#client "nicola" dopo l'avvio di una chat con un altro client

Now you can chat!
You are contacted by enzo

ciao da nicola
enzo: ciao da enzo
@stop

******
Wait for chat
Digit @exit if you want to exit

```

- `@exit`, per uscire dal server (sia durante la chat che durante l'attesa di un accoppiamento per la chat).

```
#client
******
Wait for chat
Digit @exit if you want to exit
@exit

```

Listing 2.1: client che digita @exit durante l'attesa (anologo se è in chat con un altro client)

```
#server
[4] nicola left the chat #4 e' l'sd di nicola

[4] exit from server

```

Listing 2.2: server dopo aver ricevuto @user da un client

- `@user`, per sapere il numero di utenti che si trovano in una stanza o in attesa della stessa.

```
************************************************************
                          COMMANDS
************************************************************
- @stop to suspend chat
- @exit to exit from the chat
- @user to know how many users are connected in each room
************************************************************




************************************************************
                           MENU
************************************************************
1. Topical room
2. Sport room
3. Music room
4. University room
5. Exit
Enter a choice: @user
Welcome to the multichat server!


La stanza tematica 1 contiene 3 utenti di cui 0 in coda;
La stanza tematica 2 contiene 0 utenti di cui 0 in coda;
La stanza tematica 3 contiene 0 utenti di cui 0 in coda;
La stanza tematica 4 contiene 0 utenti di cui 0 in coda;
```

Listing 2.3: client che digita @user durante la scelta nel menu

2.2 Comandi server

Il server dispone di un unico comando, ovvero `@menu` mediante il quale si potranno visualizzare a schermo una serie di informazioni quali utenti in chat e utenti in attesa, entrambi a seconda della stanza. Si tratta di un thread separato che legge da tastiera il suddetto comando mostrando, appunto, un menu.

2.3 Server side

```
MULTICHAT SERVER START PROCESSING :Thu Jun  9 20:47:52 2022

Room capacity = 5

New connection from : 79.43.145.99
139629082236672: manage sd 4
User gio for room number 1

Create chat in room 1
[gio] First client data: sd: 4 room: 1, state: W
[gio] cond_wait

New connection from : 79.43.145.99
139629065451264: manage sd 5
User nico for room number 1

[gio] Possibile chat?
[gio] Chat with partner nico
Chat started between gio and nico

Create chat in room 1
[nico] First client data: sd: 5 room: 1, state: T
[nico] cond_wait
From nico to gio
nico s message: ciao da nico
From gio to nico
gio s message: ciao
From gio to nico
gio s message: da
From gio to nico
gio s message: gio
```

Listing 2.4: output del server ad ogni nuova connessione/chat

2.3.1 La funzione *manage_Client*

Tale funzione rappresenta la funzione di avvio del thread che si occupa di un client ogni volta che si connette. Il server, per ogni connessione, inizialmente può ricevere due tipologie di messaggi:

- "@user", il cui funzionamento è stato mostrato precedentemente
- un messaggio nella forma "*room* + uno spazio + nickname".

Nel secondo caso il server spezzerà il messaggio (tramite la funzione di libreria `strtok`), verificherà l'univocità del nick scelto dall'utente e in caso positivo salverà nella struttura dell'utente le suddette informazioni, in caso contrario invierà un messaggio per richiedere un nuovo nickname finchè questo non rispetterà la condizione di univocità. Successivamente verrà stabilita la struttura in cui inserire l'utente (illustrate nel capitolo 3) a seconda della capacità della stanza: in un caso dovrà aspettare che questa si liberi nell'altro potrà entrare e avviare una chat qualora possibile.

```
static void *manageClient(void *client_sd) {
  struct T_user user;
  int sd, err;
```

```
4   ssize_t n_bytes;
5
6   sd = *((int *) client_sd);
7   printf("%ld: manage sd %d\n", pthread_self(), sd);
8
9   user = getUserClt(sd);
10  user.user_sd = sd;
11
12  printf("User %s for room number %c\n", user.nickname, user.room);
13  if(user.room == '5'){
14    printf("The client close the connection...\n");
15    close(sd);
16    free((int*)client_sd);
17    pthread_exit(0);
18  }
19
20  mutex_lock('M', user.room);
21  if(getSizeTreeByRoom(user.room) < ROOM_CAPACITY){
22    insert_Tree(user, pthread_self());
23  } else {
24    if(!manageEnqueue(user)){
25      mutex_unlock('M', user.room);
26      printf("[%s] Queue full, close client...\n", user.nickname);
27      sendMsg("Queue full, try later...", sd);
28      close(sd);
29      free((int*) client_sd);
30      pthread_exit(0);
31    }
32    while(getSizeTreeByRoom(user.room) == ROOM_CAPACITY){
33      printf("[%s] Room full\n", user.nickname);
34      sendMsg("Room full, you have to wait...\n", sd);
35      cond_wait('Q', user.room);
36    }
37    user = manageDequeue(user.room);
38    printf("[%s] user dequeued\n" ,user.nickname);
39    insert_Tree(user, pthread_self());
40  }
41  cond_broadcast('C', user.room);
42  cond_broadcast('S', user.room);
43  mutex_unlock('M', user.room);
44
45  struct T_user tmp_user;
46
47  while(1){
48    tmp_user = create_Chat(user.room, user.nickname);
49    sleep(1);
50    if(tmp_user.exit == 1){
51      printf("[%d] %s left the chat\n", sd, tmp_user.nickname);
52      fflush(stdout);
53
54      // delete the user from the tree and wake up who were waiting in↩
            queue
55      mutex_lock('M', tmp_user.room);
56      deletefromTree(tmp_user.room, tmp_user.nickname);
57      cond_broadcast('Q', tmp_user.room);
58      mutex_unlock('M', tmp_user.room);
59
60      close(tmp_user.user_sd);
```

```
61
62        // if the sd is equal to the sd of the user who has write "@exit↩
              " then break and exit from current thread
63        if(tmp_user.user_sd == sd) {
64          break;
65        }
66      }
67    }
68
69    printf("[%d] exit from server \n",sd);
70    free((int*) client_sd);
71    pthread_exit(0);
72  }
```

Listing 2.5: thread che gestisce un client, in Server.c

Alla riga 16 viene richiamata la funzione di supporto che si occuperà di inoltrare messaggi al client a seconda del messaggio ricevuto(comando o nickname)

```
1   struct T_user getUserClt(int sd) {
2     int SIZE_INIT_MESSAGE = NICKNAME_SIZE + 3;  //room + space + ↩
          nickname + \0
3
4     char buffer[60];
5     char msg_client[SIZE_INIT_MESSAGE];
6     ssize_t n_bytes = 0;
7     struct T_user user;
8     bool userFound;                 // to check if nickname is already↩
          present in a queue
9     Tree *userSearch = NULL;        // to check if nickname is already↩
          present in a tree
10
11    sendMsg("Welcome to the multichat server!", sd);
12
13    userFound = true;
14
15    while(userFound) {
16      userFound = false;
17
18      if((n_bytes = recv(sd, msg_client, sizeof(msg_client), 0)) < 0){
19        fprintf(stderr, "receive error: %s\n", strerror(errno));
20        exit(EXIT_FAILURE);
21      }
22      msg_client[n_bytes] = '\0';
23
24      while(strcmp(msg_client, "@user") == 0){
25        numberUsers(sd);
26
27        if((n_bytes = recv(sd, msg_client, sizeof(msg_client), 0)) < ↩
              0){
28          fprintf(stderr, "recv error: %s\n", strerror(errno));
29          exit(EXIT_FAILURE);
30        }
31        msg_client[n_bytes] = '\0';
32      }
33
34      // the user has digit '5' (exit on client menu) after @user
35      if(msg_client[0] == '5'){
36        user.room = '5';
```

```
37          return user;
38        }
39
40        user = getUser(msg_client);
41        userSearch = search_Tree(user.room, user.nickname);
42        if(userSearch != NULL) {
43          printf ("Nickname already present...\n");
44          userFound = true;
45        } else {
46          userFound = search_Queue(user);
47          if (userFound)
48          printf ("Nickname already present...\n");
49        }
50
51        if(userFound){
52          sendMsg("KO", sd);
53        } else {
54          sendMsg("OK", sd);
55        }
56      }
57      return(user);
58    }
```

Listing 2.6: funzione di supporto in Room.c

Il messaggio ricevuto dal client ha dimensione 23 dal momento che si contano i 20 del nickname sommati allo spazio e alla stanza(un solo carattere).

Da menù il client, come già detto, potrà digitare @user e nel caso la funzione richiamata alla riga 23 risponderà alla richiesta. A questo punto, lato client verrà rimostrato il menù. Il menù, però, fornisce la possibilità di uscire dal programma, pertanto se l'utente scegliesse tale possiblità prima ancora di inviare il messaggio contenente stanza e nickname, lato client verrà inviato un messaggio (la scelta del menù) in modo tale che anche il server possa chiudere la connessione.

Una volta ricevuto il messaggio contenente nickname e room alla riga 38 viene richiamata la funzione di supporto che si occuperà di spezzare il messaggio dell'utente per permettere il salvataggio dei dati all'interno della struttura del client. Essa ritornerà tale struttura e verrà verificato se il nickname è già presente o meno. In entrambi i casi verrà inviato un messaggio al client mediante il quale si potrà capire se richiedere o meno l'inserimento di un nuovo nickname. Se è univoco imposta un flag per l'uscita e ritorna la struttura del client.

```
1   struct T_user getUser(char *msg){
2     struct T_user user;
3     char nickname[NICKNAME_SIZE];
4
5     // get room chosen by client
6     char *token = strtok(msg, " ");
7     user.room = token[0];
8
9     while(token != NULL){
10      strcpy(nickname, token);
11      token = strtok(NULL, " ");
12    }
13
14    strcpy(user.nickname, nickname);
15    return user;
```

```
16  }
```

Listing 2.7: funzione di supporto in Room.c per spezzare il messaggio del client per permettere il salvataggio delle informazioni separatamente nella struttura

Per completezza, mostriamo la struttura del client:

```
1   struct T_user{
2     pthread_t client_tid;                    /* client's tid */
3     char nickname[NICKNAME_SIZE];            /* nickname inserted by ↩
          user client */
4     char room;                               /* chosen room by user ↩
          client*/
5     char state;                              /* 'W' for user who ↩
          waiting to chat and 'T' for user who talking */
6     char nickname_partner[NICKNAME_SIZE];    /* nickname of the last ↩
          user he chatted with or is chatting with */
7     int user_sd;                             /* socket descriptor of ↩
          current user */
8     int stop;                                /* flag used to know if ↩
          user has write "@stop". 1 if pressed, 0 otherwise */
9     int exit;                                /* flag used to know if ↩
          user has write "@exit". 1 if pressed, 0 otherwise */
10  };
```

Listing 2.8: client struct in AVL.h

2.3.2 La funzione *init_Chat*

```
1   /**
2   * This function initialize chat between two client using select ↩
        function.
3   * @param first_User_Node first user node selected randomly
4   * @param second_User_Node second user node selected randomly
5   * @return nickname of user who write "exit" into the chat, NULL ↩
        otherwise
6   */
7   static struct T_user init_chat(Tree *first_User_Node, Tree *↩
        second_User_Node);
```

Una volta che i due utenti sono stati selezionati randomicamente(lavoro svolto dalla funzione *create_Chat*) è possibile metterli in comunicazione. Per farlo è stata utilizzata una `select` per evitare che il server si blocchi su un socket descriptor su cui non c'è nulla da leggere con il fine di realizzare una comunicazione asincrona che rispetta l'idea concettuale della chat visto che entrambi i client possono scrivere un numero indefinito di messaggi (anche in sequenza) e/o restare in "attesa" leggendo semplicemente i messaggi che arrivano dall'altro client, con il fine di non creare dei thread appositi evitando di avere ulteriori thread in esecuzione oltre a quelli già necessari evitando degli overhead non necessari appunto. Al termine del timout impostato per la select semplicemente la funzione ritornerà uno dei due utenti connessi (nel nostro caso il primo ma è indifferente ritornare il primo o il secondo) i quali verranno poi messi in comunicazione con altri client, qualora possibile.

La seguente funzione, quindi, si occupa di realizzare la select e chiamare un'altra funzione (`manage_chat`, in cui è definita la comunicazione vera e propria) a seconda del socket descriptor pronto in quel momento.

Tale funzione ritornerà l'utente che ha digitato `@stop` o `@exit`.

Infine, se la comunicazione è stata interrotta a causa del comando **@stop**, rimettiamo gli utenti in stato di attesa, altrimenti se **@exit** è stato inviato da un utente allora ritorniamo proprio il nome dell'utente in modo tale da poter chiudere il suo socket descriptor.

```
1  static struct T_user init_chat(Tree *first_User_Node, Tree *second_User_Node){
2    fd_set readfds;
3    struct T_user user;              // user who has write "@exit" or @stop
4
5    timeout.tv_sec = 9600;
6    timeout.tv_usec = 0;
7
8    printf("Chat started between %s and %s\n", first_User_Node -> key.nickname, second_User_Node -> key.nickname);
9    FD_ZERO(&readfds);               // initialize set (empty)
10
11   int max_sd;
12   max_sd = (first_User_Node -> key.user_sd > second_User_Node -> key.user_sd) ?
13   first_User_Node -> key.user_sd : second_User_Node -> key.user_sd;
14
15   while(1){
16     FD_SET(first_User_Node -> key.user_sd,  &readfds);  //Insert file descriptor of first user into the set
17     FD_SET(second_User_Node -> key.user_sd, &readfds);  //Insert file descriptor of second user into the set
18
19     if(select(max_sd + 1, &readfds, NULL, NULL, &timeout) < 0){
20       perror("select");
21       exit(EXIT_FAILURE);
22     }
23
24     if (FD_ISSET(first_User_Node -> key.user_sd, &readfds)){
25       user = manage_chat(first_User_Node, second_User_Node);
26     } else if(FD_ISSET(second_User_Node -> key.user_sd, &readfds)){
27       user = manage_chat(second_User_Node, first_User_Node);
28     } else {
29       printf("TIMEOUT\n");
30       return first_User_Node -> key;
31     }
32
33     FD_ZERO(&readfds);
34
35     // return the user who has write @stop or @exit
36     if(user.exit == 1 || user.stop == 1){
37       return user;
38     }
39   }
40 }
```

2.3.3 La funzione *manage_chat*

```
1  /**
```

```
2   * This function manage the chat between two client using read and write functions.
3   * @param from_Client the user who has write
4   * @param to_Client the user who will have to read what the first one wrote
5   * @return
6   */
7   static struct T_user manage_chat(Tree *from_Client, Tree *to_Client);
```

Tale funzione si occupa della comunicazione tra i due client selezionati.

Semplicemente, quando questa funzione viene invocata allora uno dei due socket descriptor è pronto e quindi il client(cioè `from_Client`) ha scritto qualcosa e pertanto il server può leggere il messaggio e inviarlo al secondo client(`to_Client`). La dimensione massima dei messaggi che i client possono scambiare è pari a 256.

Tale funzione ritorna l'utente che ha digitato i comando di stop o di uscita.

Nel primo caso, l'altro client non avendo richiesto esplicitamente di uscire viene nuovamente messo in stato di attesa in modo che continui a restare nella stanza precedentemente selezionata.

```
1   static struct T_user manage_chat(Tree *from_Client, Tree *to_Client){
2     char read_buffer[MESSAGE_SIZE];
3     char write_buffer[MESSAGE_SIZE + NICKNAME_SIZE + 2];
4     ssize_t n_read;
5
6     printf ("From %s to %s\n", from_Client -> key.nickname, to_Client -> key.nickname);
7
8     // read from first client
9     if ((n_read = read(from_Client -> key.user_sd, read_buffer, MESSAGE_SIZE)) < 0){
10      perror("recv first user");
11      exit(EXIT_FAILURE);
12    }
13    read_buffer[n_read] = '\0';
14    printf("%s 's message: %s\n", from_Client -> key.nickname, read_buffer);
15
16    if(strcmp(read_buffer, "@user") == 0) {
17      numberUsers(from_Client -> key.user_sd);
18    } else {
19      /* write on second client */
20      sprintf(write_buffer, "%s: %s", from_Client -> key.nickname, read_buffer);
21      if((write(to_Client -> key.user_sd, write_buffer, strlen(write_buffer))) != strlen(write_buffer)){
22        perror("send second user");
23        exit(EXIT_FAILURE);
24      }
25    }
26
27    if(strcmp(read_buffer, "@exit") == 0){
28      from_Client -> key.state = 'W';
29      from_Client -> key.exit = 1;
30      from_Client -> key.stop = 0;
31      to_Client -> key.state = 'W';
```

```
32        to_Client -> key.exit = 0;
33        to_Client -> key.stop = 0;
34      } else  if(strcmp(read_buffer, "@stop") == 0) {
35        from_Client -> key.state = 'W';
36        from_Client -> key.stop = 1;
37        from_Client -> key.exit = 0;
38        to_Client -> key.state = 'W';
39        to_Client -> key.exit = 0;
40        to_Client -> key.stop = 0;
41      }
42
43      return from_Client -> key;
44    }
```

2.4 Client side

Il protocollo di comunicazione è analogo al server: si basa sulla chat asincrona "non bloccante" realizzata dalla funzione `select` che sonda i descrittori pronti a inviare/ricevere i bytes.

Essenzialmente il client effettua una scelta da menu(stanza o comando `@user`):

- nel primo caso, una volta selezionata la stanza dovrà inviare il proprio nickname (su cui il server farà un controllo per verificarne l'univocità, in caso contrario riceverà un messaggio e dovrà riprovare con la scelta del nickname)
- nel secondo caso potrà o digitare nuovamente @user o proseguire inserendo il nickname (ritornando al primo caso)

Fatto ciò il server invierà un messaggio per far capire al client se si trova in stato di attesa o se può già iniziare una chat(specificando il secondo client).

```
1   // @user command control during menu choice
2   strcpy(choice, user_Count(choice, socket_fd));
3
4   // if user has digit '5' in order to exit
5   if(choice[0] == '5'){
6     // send the char in order to close the connection from server
7     if(write(socket_fd, "5", 1) != 1)
8     perror("write"), exit(EXIT_FAILURE);
9     close(socket_fd);
10    exit(EXIT_SUCCESS);
11  }
12
13  buff[0] = choice[0];
14  buff[1] = '\0';
15
16  printf("Enter a nickname: ");
17  scanf("%19[^\n]s%*c", nickname);
18
19  // Now we concatanate the nickname to the room choosed
20  strcat(buff, " ");
21  strcat(buff, nickname);
22
23  // send message in form ROOM + SPACE + NICKNAME
24  if(write(socket_fd, buff, strlen(buff)) != strlen(buff)){
```

```
25      perror("write");
26      exit(EXIT_FAILURE);
27    }
28
29    // check if nickname is valid:
30    //  - strcmp(buffer, "OK") == 0 if nickname is valid
31    //  - "KO" otherwise
32    if((n_bytes = read(socket_fd, buffer, MESSAGE_SIZE)) < 0){
33      perror("read"), exit(EXIT_FAILURE);
34    }
35    buffer[n_bytes] = '\0';
36    //debug printf("Server: %s\n", buffer);
37
38    if(strcmp(buffer, "KO") == 0)
39    user_Check(socket_fd, choice[0]);
40
41    // wait message after everything is OK
42    if((n_bytes = read(socket_fd, buffer, MESSAGE_SIZE)) < 0){
43      perror("read"), exit(EXIT_FAILURE);
44    }
45    buffer[n_bytes] = '\0';
46    printf("%s\n", buffer);
47
48    int exit_chat = 0;
49    while(exit_chat != 1){
50      printf("**************************************************\n");
51      do {
52        FD_SET(STDIN_FILENO, &readfds);
53        FD_SET(socket_fd, &readfds);
54
55        if(select(socket_fd + 1, &readfds, NULL, NULL, &tv) < 0){
56          perror("select");
57          exit(EXIT_FAILURE);
58        }
59
60        // read from keyboard and write to server
61        if (FD_ISSET(STDIN, &readfds)){
62          fgets(buffer, MESSAGE_SIZE, stdin);
63          if(buffer[strlen(buffer) - 1] == '\n')
64          buffer[strlen(buffer) - 1] = '\0';
65
66          if(write(socket_fd, buffer, strlen(buffer)) != strlen(buffer))↩
               {
67            perror("write"), exit(EXIT_FAILURE);
68          }
69
70          if(strcmp(buffer,"@stop") == 0 || strcmp(buffer,"@exit") == 0 ↩
               || strcmp(buffer,"@user") == 0){
71            if (strcmp(buffer, "@exit") == 0){
72              exit_chat = 1;
73              break;
74            } else if(strcmp(buffer, "@user") == 0)
75            printf("\n");
76          }
77          // read from server and print the message
78        } else if (FD_ISSET(socket_fd, &readfds)) {
79          if((n_bytes = read(socket_fd, buffer, MESSAGE_SIZE)) < 0){
80            perror("write"), exit(EXIT_FAILURE);
```

```
81          } else if(n_bytes == 0) {
82            printf("Server has close connection...\n");
83            close(socket_fd);
84            exit(EXIT_SUCCESS);
85          }
86          buffer[n_bytes] = '\0';
87          printf("%s\n", buffer);
88
89          //if current client read @stop then clean screen
90          clear_screen(buffer);
91        } else {
92          printf("timeout\n");
93          strcpy(buffer, "@exit");
94          if(write(socket_fd, buffer, strlen(buffer)) != strlen(buffer))↩
               {
95            perror("write"), exit(EXIT_FAILURE);
96          }
97          exit_chat = 1;
98          break;
99        }
100
101       FD_ZERO(&readfds);
102     } while (strcmp(buffer, "@stop") != 0);
103   }
104
105   printf("Goodbye!\n");
106
107   close(socket_fd);
```

Capitolo 3

Dettagli implementativi

Come intuibile dalla struttura del progetto precedentemente descritta, lato server si è fatto uso di due strutture dati: AVL e Buffer circolare. Prima di motiviare questa scelta procediamo nel definire cosa succede nel momento in cui un client si collega. Dopo aver digitato in quale stanza essere inserito per avviare una o più chat vi sono due possibili situazioni in cui può trovarsi il client:

- Il client trova libera la stanza e pertanto subito viene inserito
- Il client, trovando la stanza piena, è costretto ad aspettare che questa si liberi (e che quindi almeno un client si disconnetta)

Pertanto, le strutture rappresentano la situazione in cui il client si trova:

- l'AVL rappresenta l'albero di utenti connessi ad una certa stanza. Tali utenti possono trovarsi o in stato di **W**(waiting) o **T**(talking).
- il buffer circolare invece rappresenta la coda di utenti in attesa che la stanza da loro selezionata si liberi.

3.1 Circular buffer

Come già detto, nel momento in cui un client è costretto ad aspettare che la stanza digitata precedentemente (tramite il menù) si liberi, questo viene inserito in una coda circolare. La coda circolare(o anche buffer circolare) non è altro che un tipo di queue, quindi basata sulla strategia FIFO(**F**irs **I**n **F**irst **O**ut), in cui l'ultima posizione è "collegata" alla prima per formare un cerchio.

Una coda classica ha il seguente svantaggio: ogni volta che un elemento viene consumato, se la coda è piena non è più possibile inserire nuovi elementi e quindi diventa necessario spostare tutti gli elementi quando uno viene consumato, il che è costoso. Con il buffer circolare questo non è necessario dal momento che l'esatta posizione iniziale non è importante in questo tipo di struttura e quindi sarà possibile inserire nuovi elementi fintantoché ci sarà spazio disponibile. Inoltre il buffer circolare rappresenta una buona strategia di implementazione per una coda con dimensioni massime fisse, che è il nostro caso dal momento che abbiamo considerato che una gestione con una coda che si espande in modo arbitrario fosse superfluo visto il contesto.

Nello specifico, per realizzare tale struttura, si utilizzano due indici: `head` cioè la testa della coda e `tail` che rappresenta il punto in cui è possibile inserire ancora elementi. Tale posizione viene calcolata mediante la formula:

$$TAIL = (TAIL + 1)\%MAX$$

dove MAX rappresenta la dimensione massima della coda.

Nel momento in cui `tail` e `head` coincidono allora vorrà dire che la coda è piena.

Pertanto, in questo modo, sarà possibile inserire ed estrarre utenti nella/dalla coda in tempo costante, infatti sia `enqueue` che `dequeue` hanno complessità pari a $O(1)$

3.2 AVL Tree

Nel momento in cui la stanza è libera (sia che l'utente abbia aspettato o meno) questo viene inserito nell'albero AVL relativo alla stanza desiderata.

Gli utenti una volta nella stanza vengono accoppiati in modo randomico, quindi occorre scegliere oppurtamente una struttura dinamica al fine di realizzare la ricerca randomica degli utenti, l'inserimento degli stessi e la cancellazione nel momento in cui questi si scolleghino. Una struttura lineare non è conveniente poichè nella maggior parte dei casi avremo $O(n)$ (per esempio una ricerca e/o cancellazione in una linked list). Di conseguenza si è scelto un albero binario di ricerca bilanciato, un AVL appunto, per realizzare le operazioni di inserimento, cancellazione e ricerca in $O(\log n)$

Al fine di selezionare randomicamente i nodi dell'albero, un nodo possiede un campo *children*. Tale campo è utilizzato come massimo valore che la funzione rand potrà generare. Una volta generato il numero semplicemente facciamo una visita dell'albero andando al nodo con quell'indice.

3.3 Realizzazione chat: la funzione *create_Chat*

Come già detto, una volta spezzato il messaggio ricevuto dal client e memorizzato le informazioni del client a seconda della capacità della stanza viene stabilito in che struttura inserire l'utente, cioè se nell'AVL o se nella coda. Più precisamente se la stanza è libera

allora l'utente viene inserito nell'albero relativo alla stanza scelta, altrimenti finisce in coda (relativa alla stanza scelta) e vi rimane finchè qualcuno non digita "@exit", cioè finchè non viene eseguita una cancellazione dell'albero (come mostrato in 2.5).

Nel momento in cui entra nell'albero allora avviene la creazione della chat. A questo punto viene selezionato il primo client cercando il nodo a esso relativo nell'albero. Fatto ciò avviene l'accoppiamento randomico. Questo è reso possibile da una funzione chiamata `insertChildren` (definita in AVL) che aggiorna il numero di figli per ogni nodo e dalla funzione `randomNode` (sempre definita in AVL) che seleziona un nodo in base ad un numero random compreso tra 0 e il numero di figli della radice(cioè il valore massimo).

Successivamente vengono incrementate il numero di persone attualmente in stato di talking e modificati i valori dei nodi (ossia vengono aggiornati i stati dei due utenti che passano dallo stato di waiting allo stato di talking, il nome del corrente/ultimo partner per evitare chat consecutive con lo stesso utente e i flag di chiusura). A quel punto avviene la chat.

Ovviamente ci saranno casi in cui un utente dovrà restare in attesa:

- l'utente è l'unico client presente nella stanza
- vi sono due utenti(nella stessa stanza), che hanno terminato la chat (tra di loro) e non essendoci altri client entrambi restano in attess
- vi è un numero n dispari di persone di cui n - *1* in stato di talking, pertanto l'unico utente rimasto va in attesa
- vi è un numero pari n di persone di cui n - *2* in stato di talking e le restanti due avendo già parlato tra di loro (nell'ultima chat ad essi relativa) entrano in stato di attesa

```
1  struct T_user create_Chat(char room, char nickname_current_user[]){
2    pthread_t tid;
3    Tree *firstUser = NULL, *secondUser = NULL;
4    struct T_user user;
5    int err, people_condition, t_kill = 0;
6
7    printf("\nCreate chat in room %c\n", room);
8
9    mutex_lock('C', room);
10   firstUser = search_Tree(room, nickname_current_user);
11   printf("[%s] First client data: sd: %d room: %c, state: %c\n", ↩
       firstUser -> key.nickname, firstUser -> key.user_sd,
12   firstUser -> key.room, firstUser -> key.state);
13
14   mutex_lock('M', room);
15   people_condition = getSizeTreeByRoom(room) - people_ChattingFunc(↩
       room, 0);
16   mutex_unlock('M', room);
17
18   while(people_condition < 2 || firstUser -> key.state == 'T'){
19     // create a thread in order to read @exit while user is waiting
20     if(firstUser -> key.state == 'W'){
21       t_kill = 1;
22       if(((err = pthread_create(&tid, NULL, exitUser, (void*) ↩
           firstUser)) != 0)){
23         fprintf(stderr, "thread create: %s\n", strerror(err));
```

```
24            exit(EXIT_FAILURE);
25          }
26
27          if((err = pthread_detach(tid)) != 0){
28            fprintf(stderr, "thread detach: %s\n", strerror(err));
29            exit(EXIT_FAILURE);
30          }
31
32          sendMsg("Wait for chat...\n", firstUser -> key.user_sd);
33        }
34
35        printf("[%s] cond_wait \n", firstUser -> key.nickname);
36        cond_wait('C', room);
37
38        // if a thread was created then send a signal to kill it
39        if(t_kill != 0) pthread_kill(tid, SIGUSR1);
40
41        // if user has write @exit while they were waiting then return ↩
              them in order to delete their node from tree
42        if(firstUser -> key.exit == 1){
43          mutex_unlock('C', room);
44          return firstUser -> key;
45        }
46
47        printf("\n[%s] Possibile chat?\n", firstUser -> key.nickname);
48
49        mutex_lock('M', room);
50        people_condition = getSizeTreeByRoom(room) - people_ChattingFunc↩
              (room, 0);
51        mutex_unlock('M', room);
52      }
53      mutex_unlock('C', room);
54
55      mutex_lock('M', room);
56      insertChildrenByRoom(room);
57      secondUser = randomNodeRoom(room);
58      mutex_unlock('M', room);
59      printf("[%s] second user chosen randomly: %s\n", firstUser -> key.↩
            nickname, secondUser -> key.nickname);
60
61      mutex_lock('S', room);
62      t_kill = 0;
63      while(secondUser -> key.state != 'W' ||  firstUser -> key.state !=↩
            'W' || secondUser -> key.exit != 0
64      || (strcmp(secondUser -> key.nickname, firstUser -> key.↩
            nickname_partner) == 0)
65      || (strcmp(secondUser -> key.nickname, firstUser -> key.nickname) ↩
            == 0)) {
66        while(clientRecalculation(firstUser -> key.room, firstUser, ↩
              secondUser)) {
67          // create a thread in order to read @exit while user is ↩
                waiting
68          if(firstUser -> key.state == 'W'){
69            t_kill = 1;
70            if(((err = pthread_create(&tid, NULL, exitUser, (void*) ↩
                  firstUser)) != 0)){
71              fprintf(stderr, "thread create: %s\n", strerror(err));
72              exit(EXIT_FAILURE);
```

```
73            }
74
75            if((err = pthread_detach(tid)) != 0){
76              fprintf(stderr, "thread detach: %s\n", strerror(err));
77              exit(EXIT_FAILURE);
78            }
79
80            sendMsg("Wait for chat...\n", firstUser -> key.user_sd);
81          }
82
83          printf("[%s] wait for second user \n", firstUser -> key.↩
                nickname);
84          cond_wait('S',room);
85
86          // send signal to thread who handle the @exit in order to ↩
                close it
87          if(t_kill != 0) pthread_kill(tid, SIGUSR1);
88
89          // check if the user, during the waiting, has write @exit
90          if(firstUser -> key.exit == 1){
91              mutex_unlock('S', room);
92              return firstUser -> key;
93          }
94
95          printf("[%s] possible chat \n", firstUser -> key.nickname);
96        }
97
98        printf("[%s] user %s not valid, then recalculate\n", firstUser ↩
             -> key.nickname, secondUser -> key.nickname);
99
100       mutex_lock('M', room);
101       insertChildrenByRoom(room);
102       secondUser = randomNodeRoom(room);
103       mutex_unlock('M', room);
104     }
105     mutex_unlock('S',room);
106
107     printf("[%s] Chat with Partner clt %s\n", firstUser -> key.↩
          nickname, secondUser -> key.nickname);
108
109     /* update info of nodes chosen */
110     firstUser -> key.state = 'T';
111     secondUser -> key.state = 'T';
112
113     strcpy(firstUser -> key.nickname_partner, secondUser -> key.↩
          nickname);
114     strcpy(secondUser -> key.nickname_partner, firstUser -> key.↩
          nickname);
115
116     firstUser -> key.stop = 0;
117     secondUser -> key.stop = 0;
118     firstUser -> key.exit = 0;
119     secondUser -> key.exit = 0;
120     /* end update */
121
122     mutex_lock('C', user.room);
123     mutex_lock('S', user.room);
124     people_ChattingFunc(room, 2);
```

```
125     mutex_unlock('S', user.room);
126     mutex_unlock('C', user.room);
127
128     welcomeToTheChat(firstUser, secondUser);
129
130     user = init_chat(firstUser, secondUser);
131
132     mutex_lock('C', user.room);
133     mutex_lock('S', user.room);
134     people_ChattingFunc(room, -2);
135     cond_broadcast('C', user.room);
136     cond_broadcast('S', user.room);
137     mutex_unlock('S', user.room);
138     mutex_unlock('C', user.room);
139
140     return user;
141   }
```

3.4 System call

Per lo scambio di messaggi si è fatto uso delle note `read` e `write`, mentre le restanti sono relative ai thread in particolare si è fatto uso dei `trylock` necessari per evitare situazioni di deadlock, in particolare si è fatto uso di tre mutex: uno per le strutture(M), uno per il primo utente della chat(C) ed uno per il secondario(S) essendo considerabili due entità diverse. Inoltre, sarà il thread relativo al primo utente ad occuparsi della chat, quello del secondario andrà in stato di attesa.

- Per ogni accesso alle strutture si è fatto uso di un mutex 'M'
  - se la taglia dell'albero non è uguale alla capacità massima della stanza allora l'utente può entrare nella stanza (l'albero quindi) e a questo punto si manda un signal a chi era in attesa di un secondo utente per la creazione di una chat
  - altrimenti entra in coda e resta in attesa su una condition variable 'Q' finchè non viene fatta una cancellazione dall'albero
- Per il mutex 'C' sostanzialmente l'utente va in attesa se è già in chat con qualcuno (quindi il thread relativo all'altro utente con cui è in chat si sta occupando di quella chat) oppure tutti gli altri utenti sono già impegnati o ancora se è l'unico utente in stanza e quindi va in attesa
- Per il mutex 'S', invece, bisogna verificare se l'accoppiamento è valido, cioè se l'utente selezionato randomicamente è già in stato di talking, se già vi ha parlato nell'ultima chat o se vi sono client con cui è effettivamente possibile avviare una chat
- Entrambe le sezioni critiche di 'C' ed 'S' fanno uso di una variabile globale che conta il numero di persone attualmente impegnate in una chat. Inoltre, entrambe le sezioni critiche fanno uso dell'AVL (taglia dell'albero e modifica dei figli per ogni nodo dell'albero per ottenere randomicamente il nodo), pertanto l'accesso è racchiuso tra i mutex 'M'.

Inoltre sono stati gestiti i seguenti segnali:

- lato server
  - Un segnale SIGUSR1: quando un client va in attesa viene lanciato un ulteriore thread affinchè si possa leggere il comando "@exit" nonostante il client sia in attesa: quindi una volta digitato il comando viene mandato un segnale in broadcast affinchè si svegli anche il thread relativo all'utente che lo ha digitato in modo tale da poter ritornare il nome dello stesso e rimuoverlo dall'albero e chiudere il suo socket. L'handler si occuperà di effettuare, semplicemente, la exit relativa al seguente thread:

```
1       static void *exitUser(void *arg){
2         Tree *firstUser = (Tree*) arg;
3         char buff[81];
4
5         memset(buff, '\0', 81);
6         strcpy(buff, "\nDigit @exit if you want to exit...\n"↩
              );
7         if((write(firstUser -> key.user_sd, buff, strlen(buff↩
              )) != strlen(buff))){
8           perror("send first user");
9           exit(EXIT_FAILURE);
10        }
11        memset(buff, '\0', 81);
12
13        if((read(firstUser -> key.user_sd, buff, 5)) < 0) {
14          perror("read first user");
15          exit(EXIT_FAILURE);
16        }
17
18        if (strcmp(buff, "@exit") == 0){
19          /* Wake up the thread who manage the user who has ↩
                write @exit
20          *  in order to return their name and delete them ↩
                from tree */
21          firstUser -> key.exit = 1;
22          cond_broadcast('C', firstUser -> key.room);
23          cond_broadcast('S', firstUser -> key.room);
24        }
25
26        pthread_exit(EXIT_SUCCESS);
27      }
```

  - lato client:
    - è stato utilizzato semplicemente un segnale per ignorare CTRL+C, cioè `signal(SIGINT, SIG_IGN)` in modo da poter chiudere il client solo mediante l'apposito comando @exit per far sì che il server possa sapere chi rimuovere dall'albero