Università degli Studi di Roma Tor Vergata

Facoltà di Ingegneria

Corso di Ingegneria del Web

Progetto Chat room P2P

Appendice

Professoressa:
Valeria Cardellini

Studenti:
Ibrahim Khalili
Simone Notargiacomo
Lorenzo Tavernese

Anno Accademico 2007-2008

Contents

Chapter 1

Appendice

1.1 common.h

```
#ifndef COMMON_H
#define COMMON_H

typedef unsigned char u_int1;
typedef unsigned short u_int2;
typedef unsigned int u_int4;
typedef unsigned long long u_int8;

#endif
```

1.2 logger.h

```
#ifndef LOGGER_H
#define LOGGER_H

#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <string.h>
#include <stdarg.h>
#include <pthread.h>
#include "common.h"
#include "confmanager.h"

/**
 * Definizione dei livelli del logger.
 */
#define ALARM_INFO              0
#define INFO                    1
#define CTRL_INFO               2
#define SYS_INFO                3
#define PAC_INFO                4
#define HTTP_INFO               5
#define TORTELLA_INFO   6
#define SOCK_INFO               7

static char *pathname;

static FILE *fd_file;

static pthread_mutex_t logger_mutex;

static int verbose_l = 0;

/**
 * Inizializza il logger, scegliendo come path del file su cui scrivere il valore
 * presente nel file di configurazione. Il parametro verbose_level serve per
 * specificare fino a quale livello il logger deve salvare le informazioni. Viene
 * inizializzato anche un mutex per evitare accessi simultanei.
 */
int logger_init(int verbose_level);

/**
 * Chiude il file su cui il logger stava salvando le informazioni.
 */
int logger_close();

/**
 * Ritorna un timestamp.
 * Esempio: Tue Jun 17 16:26:28 2008
 */
char *get_timestamp();

/**
 * Si comporta come una printf, ma oltre alla stampa a video viene eseguita anche
 * una scrittura su file in base al livello di verbosita'.
 */
int logger(int type, const char* text, ...);
```

```
#endif //!LOGGER_H
```

1.3 logger.c

```
#include "logger.h"

/**
 * Inizializza il logger, scegliendo come path del file su cui scrivere il valore
 * presente nel file di configurazione. Il parametro verbose_level serve per
 * specificare fino a quale livello il logger deve salvare le informazioni. Viene
 * inizializzato anche un mutex per evitare accessi simultanei.
 */
int logger_init(int verbose_level) {
 pathname = (char*)malloc(128);
 strcpy(pathname, conf_get_path());
 printf("[logger_init]init log file: %s\n", pathname);

 fd_file = fopen(pathname, "a");
 pthread_mutex_init(&logger_mutex, NULL);
 verbose_l = verbose_level;
 return 1;
}

/**
  * Chiude il file su cui il logger stava salvando le informazioni.
  */
int logger_close() {
 return fclose(fd_file);
}

/**
  * Ritorna un timestamp.
  * Esempio: Tue Jun 17 16:26:28 2008
  */
char *get_timestamp() {
 time_t t = time(NULL);
 char *ret = (char*)malloc(128);
 sprintf(ret, "%s  ", asctime(localtime(&t)));
 ret[strlen(ret)-3]=':';
 ret[strlen(ret)-2]=' ';
 ret[strlen(ret)-1]='\0';
 return ret;
}

/**
  * Si comporta come una printf, ma oltre alla stampa a video viene eseguita anche
  * una scrittura su file in base al livello di verbosita'.
  */
int logger(int type, const char* text, ...) {
 va_list ap;
 va_start(ap, text);
 pthread_mutex_lock(&logger_mutex);
 fprintf(fd_file, "<%u>%s", (int)pthread_self(), get_timestamp());
 vfprintf(fd_file, text, ap);

 if(type<=verbose_l) {
  printf("<%u>", pthread_self());
  vprintf(text, ap);
 }
 va_end(ap);

 pthread_mutex_unlock(&logger_mutex);
 return 0;
}
```

1.4 init.h

```
#ifndef INIT_H
#define INIT_H

#include <glib.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "common.h"
#include "logger.h"


/** dati dei peer inizialmente conosciuti. */
struct init_data {
 char *ip;
 u_int4 port;
};
typedef struct init_data init_data;

/**
  * Legge il file specificato dal parametro filename (che contiene il path) e
  * e aggiunge tutti i peer presenti all'interno del file in una lista contenente
  * strutture di tipo init_data.
  * Il file ha la seguente struttura:
  * 127.0.0.1;2110;
  * 127.0.0.1;2120;
  * ...
  */
GList *init_read_file(const char *filename);

/**
  * istanzia la struttura init_data. Viene invocata da init_read_file per aggiungere
  * gli elementi alla lista. Riceve in ingresso il buffer contenente ip e porta
  * del vicino e tokenizza la stringa riempiendo la struttura dati in modo opportuno.
  */
init_data *init_char_to_initdata(char *buffer);

#endif
```

1.5 init.c

```
#include "init.h"


/**
 * Legge il file specificato dal parametro filename (che contiene il path)
 * e aggiunge tutti i peer presenti all'interno del file in una lista contenente
 * strutture di tipo init_data.
 * Il file ha la seguente struttura:
 * 127.0.0.1;2110;
 * 127.0.0.1;2120;
 * ...
 */
GList *init_read_file(const char *filename) {
 GList *init_list=NULL;
 char buffer;
 int fd=0;
 int nread=0;
 char *tmp=(char *)calloc(22,1);
 int i=0;

 if(filename==NULL || strlen(filename)==0) {
  logger(ALARM_INFO, "[init_read_file] Filename incorrecto or file not present\n");
  return NULL;
 }

 if((fd=open(filename, O_RDONLY|O_EXCL))<0){
  logger(ALARM_INFO, "[init_read_file] Error opening file\n");
  return NULL;
 }

 while((nread=read(fd,&buffer,1))>0){
  tmp[i]=buffer;

  if(tmp[i]=='\n'){
   //!aggiunta dei peer vicini alla lista
   init_list=g_list_append(init_list,(gpointer)init_char_to_initdata(tmp));
   memset(tmp,0,strlen(tmp));
   i=0;
  }
  else
   i++;
 }

 if(close(fd)<0){
  logger(ALARM_INFO, "[init_read_file] Error closing file\n");
  return NULL;
 }
 return init_list;
}

/**
  * istanzia la struttura init_data. Viene invocata da init_read_file per aggiungere
  * gli elementi alla lista. Riceve in ingresso il buffer contenente ip e porta
  * del vicino e tokenizza la stringa riempiendo la struttura dati in modo opportuno.
  */
init_data *init_char_to_initdata(char *buffer){

 char *ip;
 char *port;
 char *saveptr;
 init_data *data=calloc(1,sizeof(init_data));
```

```
 ip=strtok_r(buffer,";",&saveptr);
 data->ip=strdup(ip);
 port=strtok_r(NULL,";",&saveptr);
 data->port=atoi(port);

 return data;
}
```

1.6 confmanager.h

```
//!LOGGER
static int verbose = 3; //!Valore standard di verbosita'

//!SOCKET
static int qlen = 5; //!Coda di servizio per ricezione SYNC
static int buffer_len = 1024; //!Lunghezza buffer ricevzione/trasmissione

//!PACKET
static char *path = "/tmp/";

//!UTILS
static u_int8 gen_start = 100000; //!fake IDs start range

//!SERVENT
static int max_len = 4000;
static int max_thread = 20;
static int max_fd = 100;
static u_int4 timer_interval = 20; //!Intervallo del PING timer
static u_int8 connection_id_limit = 10000; //!Limite inferiore dei fake ID generati

//!SUPERNODE
static char *datadir = "./data";

//!SERVENT
static char *local_ip;
static u_int4 local_port;
static char *nick;

int conf_read(const char *filename);

int conf_save_value(const char *line);

int conf_get_qlen(void);

int conf_get_buffer_len(void);

char *conf_get_path(void);

u_int8 conf_get_gen_start(void);

char *conf_get_datadir(void);

char *conf_get_local_ip(void);

u_int4 conf_get_local_port(void);

char *conf_get_nick(void);

int conf_get_verbose(void);

u_int4 conf_get_timer_interval(void);

u_int8 conf_get_connection_id_limit(void);

#endif /*CONFMANAGER_H*/
```

1.7 confmanager.c

```
 while((nread=read(fd,&buffer,1))>0){
  tmp[i]=buffer;

  if(tmp[i]=='\n'){
   if(strchr(tmp, '#')!=NULL) {
        //!printf("Comment\n");
   }
   else if(strlen(tmp)<=2) {
        //!printf("Empty line\n");
   }
   else {
        conf_save_value(tmp);
   }
   memset(tmp,0,strlen(tmp));
   i=0;
  }
  else
   i++;
 }
 return 0;
}

int conf_save_value(const char *line) {
 if(line==NULL) {
  printf("Unable to read line\n");
  return -1;
 }

 char *line_dup = calloc(strlen(line), 1);
 int i, j;
 for(i=0, j=0; i<strlen(line); i++) {
  if(line[i] != ' ' && line[i] != '\n' && line[i] != '\t') {
   line_dup[j++] = line[i];
  }
 }
 line_dup[j]='\0';

 char *equal = strchr(line_dup, '=');
 int left_len = equal-line_dup;
 char *left = calloc(left_len+1, 1);
 strncpy(left, line_dup, left_len);

 int right_len = (&line_dup[strlen(line_dup)])-equal-2;
 char *right = calloc(right_len+1, 1);
 equal++;
 strncpy(right, equal, right_len);

 if(strcmp(left, "qlen")==0)
  qlen = atoi(right);
 else if(strcmp(left, "buffer_len")==0)
  buffer_len = atoi(right);
 else if(strcmp(left, "path")==0)
  path = right;
 else if(strcmp(left, "gen_start")==0)
  gen_start = atoll(right);
 else if(strcmp(left, "max_len")==0)
  max_len = atoi(right);
 else if(strcmp(left, "max_thread")==0)
  max_thread = atoi(right);
 else if(strcmp(left, "max_fd")==0)
  max_fd = atoi(right);
```

```
 else if(strcmp(left, "datadir")==0)
  datadir = right;
 else if(strcmp(left, "local_ip")==0)
  local_ip = right;
 else if(strcmp(left, "local_port")==0)
  local_port = atoi(right);
 else if(strcmp(left, "nick")==0)
  nick = right;
 else if(strcmp(left, "verbose")==0)
  verbose = atoi(right);
 else if(strcmp(left, "timer_interval")==0)
  timer_interval = atoi(right);
 else if(strcmp(left, "connection_id_limit")==0)
  connection_id_limit = atoll(right);

 return 0;
}

int conf_get_qlen(void) {
 return qlen;
}

int conf_get_buffer_len(void) {
 return buffer_len;
}

char *conf_get_path(void) {
 return path;
}

u_int8 conf_get_gen_start(void) {
 return gen_start;
}

char *conf_get_datadir(void) {
 return datadir;
}

char *conf_get_local_ip(void) {
 return local_ip;
}

u_int4 conf_get_local_port(void) {
 return local_port;
}

char *conf_get_nick(void) {
 return nick;
}

int conf_get_verbose(void) {
 return verbose;
}

u_int4 conf_get_timer_interval(void) {
 return timer_interval;
}

u_int8 conf_get_connection_id_limit(void) {
 return connection_id_limit;
}
```

1.8 utils.h

```
#ifndef UTILS_H
#define UTILS_H

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <ctype.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <linux/if.h>
#include "confmanager.h"
#include "common.h"

/**
 * Genera l'id dei pacchetti e degli utenti, avviene  tramite una combinazione
 * del MAC-ADDRESS della macchina, il valore generato dalla funzione pseudo-randomica
 * srandom(time(NULL)), e dall'ID iniziale presente nel file di configurazione
 *  di ogni peer.
 */
u_int8 generate_id(void);

//!Obsoleto
int generate_id4(void);

/**
 * Conversione in stringa di un unsigned long long.
 */
char *to_string(u_int8 num);

/**
 * Preparazione di un messaggio da inviare agli altri utenti: al messaggio originale
 * da inviare vengono aggiunti data e orario di invio e il nick del mittente.
 * Esempio: Data - ora - nickname : messaggio.
 */
char *prepare_msg(time_t timestamp, const char *nick, char *msg, int msg_len);

/**
 * Effettua il dump in esadecimale di un pacchetto.
 * Esempio di dump:
 *  50 4f 53 54 20 2a 20 48 54 54          POST * HTT
 *  50 2f 31 2e 31 0d 0a 55 73 65          P/1.1..Use
 *  72 2d 41 67 65 6e 74 3a 20 54          r-Agent: T
 *  6f 72 54 65 6c 6c 61 2f 30 2e          orTella/0.
 *  31 0d 0a 43 6f 6e 74 65 6e 74          1..Content
 */
char *hex_dump(const char *packet, int len, int n);

/**
 * Chiama la funzione hex_dump specificando quanti caratteri devono essere stampati
 * su una riga. Il parametro e' impostato di default a 10.
 */
char *dump_data(const char *packet, int len);

/**
 * Ottiene l'indirizzo MAC dell'interfaccia di rete disponibile.
 */
char *get_mac_addr(void);
```

```
#endif //!UTILS_H
```

1.9 utils.c

```
#include "utils.h"

/**
 * Genera l'id dei pacchetti e degli utenti, avviene  tramite una combinazione
 * del MAC-ADDRESS della macchina, il valore generato dalla funzione pseudo-randomica
 * srandom(time(NULL)), e dall'ID iniziale presente nel file di configurazione
 *  di ogni peer.
 */
u_int8 generate_id(void) {
 char *addr;
 addr = get_mac_addr();
 int i=0, j=10;
 unsigned long res = 0;
 for(; i<6; i++, j*=10) {
  res += addr[i]*j;
 }
 res *= random();
 return (conf_get_gen_start()+((random())^res))*(res/2);
}

//!Obsoleto
int generate_id4(void) {
 char *addr;
 addr = get_mac_addr();
 int i=0, j=10;
 unsigned long res = 0;
 for(; i<6; i++, j*=10) {
  res += addr[i]*j;
 }
 return (random())^res;
}

/**
  * Conversione in stringa di un unsigned long long.
  */
char *to_string(u_int8 num) {
 char *ret = (char*)malloc(60);
 sprintf(ret, "%lld", num);
 ret = realloc(ret, strlen(ret)+1);
 return ret;
}

/**
  * Preparazione di un messaggio da inviare agli altri utenti: al messaggio originale
  * da inviare vengono aggiunti data e orario di invio e il nick del mittente.
  * Esempio: Data - ora - nickname : messaggio.
  */
char *prepare_msg(time_t timestamp, const char *nick, char *msg, int msg_len) {
 char *time_str = asctime(localtime(&timestamp));
 time_str[strlen(time_str)-1]='\0';
 char *send_msg = calloc(msg_len+strlen(time_str)+strlen(nick)+60, 1);
 msg[msg_len] = '\0';
 sprintf(send_msg, "%s %s:\n%s\n\n", time_str, nick, msg);
 return send_msg;
}

/**
  * Effettua il dump in esadecimale di un pacchetto.
  * Esempio di dump:
  *  50 4f 53 54 20 2a 20 48 54 54         POST * HTT
  *  50 2f 31 2e 31 0d 0a 55 73 65         P/1.1..Use
```

```
  *  72 2d 41 67 65 6e 74 3a 20 54          r-Agent: T
  *  6f 72 54 65 6c 6c 61 2f 30 2e          orTella/0.
  *  31 0d 0a 43 6f 6e 74 65 6e 74          1..Content
  */
char *hex_dump(const char *packet, int len, int n)
{
 int i=0;
 int count = 0;
 int modulo =0;
 int div = len/n;
 int k = 0;
 int divtemp = div;
 if(len%n!=0) {
  modulo = n-(len%n);
  divtemp++;
 }

 int length =   (len*4+(divtemp)*4+(modulo)*3)*2;
 char *buffer = (char*)calloc(length, 1);
 char *strtemp = (char*)calloc(4, 1);

 strcat(buffer, "\n");

 if(len == 1) {
  sprintf(strtemp,"%02x ",(u_int1)packet[i]);
  strcat(buffer,strtemp);
  int templ = 0;
  templ = templ%n;

  for(k=n;k>templ+1;k--) {
   sprintf(strtemp,"   ");
   strcat(buffer,strtemp);
  }
  sprintf(strtemp,"\t  ");
  strcat(buffer,strtemp);
  sprintf(strtemp,"%c",packet[i]);
  strcat(buffer,strtemp);
  sprintf(strtemp,"\n\n");
  strcat(buffer,strtemp);
  return buffer;
 }

 for(i=0;i<len;i++) {

  if(i==len-1 && ((len)%n)!=0) {
   if((len-1)%n==0) {
         for(k=n;k>0;k--) {
          sprintf(strtemp,"%02x ", (u_int1)packet[i-k]);
          strcat(buffer,strtemp);
         }
         sprintf(strtemp,"\t  ");
         strcat(buffer,strtemp);
         for(k=n;k>0;k--) {
          sprintf(strtemp,"%c", ((isprint(packet[i-k])!=0)?packet[i-k]:'.'));
          strcat(buffer,strtemp);
         }
         sprintf(strtemp,"\n");
         strcat(buffer,strtemp);
   }
   int temp = 0;
   for(k=div*n;k<len;k++) {
         sprintf(strtemp,"%02x ",(u_int1)packet[k]);
         strcat(buffer,strtemp);
         temp = k;
```

```
    }

    temp = temp%n;
    for(k=n;k>temp+1;k--) {
          sprintf(strtemp,"   ");
          strcat(buffer,strtemp);
    }
    sprintf(strtemp,"\t  ");
    strcat(buffer,strtemp);
    for(k=div*n;k<len;k++) {
          sprintf(strtemp,"%c",((isprint(packet[k])!=0)?packet[k]:'.'));
          strcat(buffer,strtemp);
    }
    sprintf(strtemp,"\n");
    strcat(buffer,strtemp);
    break;
   }
   if(i==len-1 && (len)%n==0) {
    for(k=count;k<len;k++) {
          sprintf(strtemp,"%02x ",(u_int1)packet[k]);
          strcat(buffer,strtemp);
    }
    sprintf(strtemp,"\t  ");
    strcat(buffer,strtemp);
    for(k=count;k<len;k++) {
          sprintf(strtemp,"%c",((isprint(packet[k])!=0)?packet[k]:'.'));
          strcat(buffer,strtemp);
    }
    sprintf(strtemp,"\n");
    strcat(buffer,strtemp);
    break;
   }
   if(i!= 0 && i%n==0) {
    count = i;
    for(k=n;k>0;k--) {
          sprintf(strtemp,"%02x ", (u_int1)packet[i-k]);
          strcat(buffer,strtemp);
    }
    sprintf(strtemp,"\t  ");
    strcat(buffer,strtemp);
    for(k=n;k>0;k--) {
          sprintf(strtemp,"%c", ((isprint(packet[i-k])!=0)?packet[i-k]:'.'));
          strcat(buffer,strtemp);
    }
    sprintf(strtemp,"\n");
    strcat(buffer,strtemp);
    continue;
   }

  }

  strcat(buffer, "\n");
  return buffer;
}

/**
   * Chiama la funzione hex_dump specificando quanti caratteri devono essere stampati
   * su una riga. Il parametro e' impostato di default a 10.
   */
char *dump_data(const char *packet, int len) {
  return hex_dump(packet, len, 10);
}
```

```
/**
  * Ottiene l'indirizzo MAC dell'interfaccia di rete disponibile.
  */
char *get_mac_addr(void) {
 char *addr = calloc(7, 1);
 struct ifreq ifr;
 struct ifreq *IFR;
 struct ifconf ifc;
 char buf[1024];
 int s, i;
 int ok = 0;

 s = socket(AF_INET, SOCK_DGRAM, 0);
 if (s==-1) {
  return NULL;
 }

 ifc.ifc_len = sizeof(buf);
 ifc.ifc_buf = buf;
 ioctl(s, SIOCGIFCONF, &ifc);

 IFR = ifc.ifc_req;

 i = (ifc.ifc_len / sizeof(struct ifreq));
 if(i<=1) {
  char *ret = calloc(128, 1);
  time_t t = time(NULL);
  sprintf(ret, "%s", asctime(localtime(&t)));
  memcpy(addr, ret, 6);
  return addr;
 }
 for (; i >= 0;) {

  strcpy(ifr.ifr_name, IFR->ifr_name);
  if (ioctl(s, SIOCGIFFLAGS, &ifr) == 0) {
   if (! (ifr.ifr_flags & IFF_LOOPBACK)) {
         if (ioctl(s, SIOCGIFHWADDR, &ifr) == 0) {
          ok = 1;
          break;
         }
   }
  }
  IFR++;
  i--;
 }

 close(s);
 if (ok) {
  bcopy( ifr.ifr_hwaddr.sa_data, addr, 6);
  addr[6] = '\0';
 }
 else {
  return NULL;
 }
 return addr;
}
```

1.10 tortellaprotocol.h

```
#ifndef TORTELLA_PROTOCOL_H
#define TORTELLA_PROTOCOL_H

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "common.h"
#include "utils.h"

#define GET_PING(packet)                ((ping_desc*)packet->desc)
#define GET_JOIN(packet)                ((join_desc*)packet->desc)
#define GET_LEAVE(packet)               ((leave_desc*)packet->desc)
#define GET_MESSAGE(packet)             ((message_desc*)packet->desc)
#define GET_SEARCH(packet)              ((search_desc*)packet->desc)
#define GET_SEARCHHITS(packet)  ((searchhits_desc*)packet->desc)
#define GET_LIST(packet)                ((list_desc*)packet->desc)
#define GET_LISTHITS(packet)    ((listhits_desc*)packet->desc)
#define GET_BYE(packet)                 ((bye_desc*)packet->desc)

//!Descriptor ID
#define PING_ID                 0x01
#define LIST_ID                 0x03
#define LISTHITS_ID             0x04
#define JOIN_ID                 0x05
#define LEAVE_ID                0x06
#define MESSAGE_ID              0x07
#define SEARCH_ID               0x09
#define SEARCHHITS_ID   0x10
#define BYE_ID                  0x11
#define CLOSE_ID                0x12

//!Status ID
#define ONLINE_ID       0x80
#define BUSY_ID         0x81
#define AWAY_ID         0x82

/*
 *                          TorTella Header
 * +----+---------+-----------+---------+-----------+----------+----------+
 * | ID | desc_ID | sender_ID | recv_ID | timestamp | desc_len | data_len |
 * +----+---------+-----------+---------+-----------+----------+----------+
 */
/** struttura dati dove vengono memorizzati rispettivamente l'id univoco del
 * pacchetto, il descrittore del pacchetto,l'id del servente che invia il pacchetto,
 * l'id del ricevente, il timestamp e la relativa lunghezza sia del descrittore che
 * del campo dati.
 */
struct tortella_header {
 u_int8 id;
 u_int4 desc_id;
 u_int8 sender_id;
 u_int8 recv_id;
 time_t timestamp;
 u_int4 desc_len;
 u_int4 data_len;
};
typedef struct tortella_header tortella_header;

/**
  * struttura dati per il pacchetto di tipo PING, in cui viene memorizzato
```

```
  * lo status dell'utente al momento dell'invio del pacchetto
  */
struct ping_desc {
 u_int4 port;
 u_int1 status;
 //!Campo dati: nickname

};
typedef struct ping_desc ping_desc;

/**
  * struttura dati per il pacchetto LIST, in cui viene memorizzato il valore
  * del TTL e dell' HOPS, e l'ID della chat di cui si richiede la lista degli
  * utenti presenti
  */
struct list_desc {
 u_int8 chat_id;
 u_int1 ttl;
 u_int1 hops;
};
typedef struct list_desc list_desc;

/**
  * struttura dati per il pacchetto di risposta al LIST, in cui sono salvati
  *  gli stessi dati contenuti nella struttura list_desc con aggiunta del numero
  *   di utenti connessi alla chat, di cui era stata richiesta la lista degli users
  */

struct listhits_desc {
 u_int4 user_num;
 u_int1 ttl;
 u_int1 hops;
 u_int8 chat_id;
 //!Campo dati: elenco utenti della chat
};
typedef struct listhits_desc listhits_desc;

/**
  * struttura dati per il pacchetto di join ad una chat.
  * Nella struct sono salvate tutte le info relative all'utente
  * che vuole accedere alla chat: stato(ONLINE-BUSY-AWAY)-ID dell'utente-
  * -Id della chat-numero di porta del socket-indirizzo ip. Inoltre sono
  * salvati i valori dell TTL e dell'HOPS per il flooding del pacchetto.
  * Tutti i dati contenuti da tale descrittore rimarranno identici(salvo per il TTL e
      HOPS)
  * per tutto il percorso fatto dal pacchetto nella rete.
  */

struct join_desc {
 u_int1 status;
 u_int8 user_id;
 u_int8 chat_id;
 u_int4 port;
 char ip[16];
 u_int1 ttl;
 u_int1 hops;
 //!Campo dati: nickname

};
typedef struct join_desc join_desc;

/**
  * struttura dati per l'invio del pacchetto LEAVE ad una chat.
  * I dati salvati sono rispettivamente l'ID dell'utente che vuole uscire
```

```
  * dalla chat e il rispettivo identificativo della chat. Come per il pacchetto
  * JOIN anche in questo caso i dati memorizzati rimarranno identici(a meno del TTL e
      HOPS)
  * per tutto il percorso fatto nella rete.
  */

struct leave_desc {
 u_int8 user_id;
 u_int8 chat_id;
 u_int1 ttl;
 u_int1 hops;
};
typedef struct leave_desc leave_desc;

/**
  * Data
  * +-----+
  * | msg |
  * +-----+
  */
/** struttura dati per l'invio del messaggio ad una specifica chat,
  * tenendo memoria per l'appunto in tlae struttura l'ID della chat a cui
  * e' diretto il messaggio di testo.
  */
struct message_desc {
 u_int8 chat_id;
 //!Campo dati: il msg

};
typedef struct message_desc message_desc;

/*
  * Data
  * +-------+
  * | Query |
  * +-------+
  */
/** struct per il flooding del pacchetto SEARCH. Ogni utente
  * inviera' una richiesta di ricerca, e in tale
  * struttura verra' decrementato il valore del TTL e incrementato quello
  * dell'HOPS per ogni nodo attraversato dal pacchetto durante il suo percorso
  * nella rete.
  */
struct search_desc {
 u_int1 ttl;
 u_int1 hops;
 //!Campo dati: stringa ricerca

};
typedef struct search_desc search_desc;

/**
  * struct per l'invio del pacchetto SEARCHHITS.
  * Il routing di tale pacchetto segue il percorso inverso
  * di quello eseguito dal pacchetto SEARCH.
  *
  */
struct searchhits_desc {
 u_int4 num_res;
 //!Campo dati: risultati ricerca
};
typedef struct searchhits_desc searchhits_desc;

/**
```

```
  * struttura usata per segnalare la disconnessione dalla rete tortella.
  */
struct bye_desc {
};
typedef struct bye_desc bye_desc;

/**
  *         Tortella Packet
  * +--------+------------+------+
  * | Header | Descriptor | Data |
  * +--------+------------+------+
  */
/** struttua del pacchetto Tortella. Nella struttura
  * viene memorizzato il puntatore relativo al suo header,
  * il descrittore di una delle possibili strutture dati(join_desc,search_desc,..)
  */
struct tortella_packet {
 tortella_header *header;
 char *desc;     //!desc_len contenuta nell'header del pacchetto
 char *data;     //!data_len contenuta nell'header del pacchetto
};
typedef struct tortella_packet tortella_packet;

/**
  * Ha il compito di eseguire il parser del pacchetto tortella.
  * In paritcolare prende in input il pacchetto, memorizzato nella sua
  * struttura dati, e restituisce tutto il suo contenuto in un buffer di caratteri.
  * Inoltre il parametro len ritorna la lunghezza di tale buffer.
  */
char *tortella_bin_to_char(tortella_packet *packet, u_int4 *len);

/**
  * Svolge le funzionalita' di parser inverso rispetto alla funzione
      tortella_bin_to_char.
  * La procedura riceve come parametro il buffer, contenente i dati, i quali vengono
      memorizzati nella
  * struttura dati tortella_packet.
  */
tortella_packet *tortella_char_to_bin(char *packet);

/**
  * funzione adibita alla stampa ordinata di tutte le informazioni contenute nella
      struttura dati
  * tortella_packet, ricevuta come parametro d'input.
  */
void print_packet(tortella_packet *packet);

/**
  * funzione che si occupa della creazione del pacchetto tortella.
  * La procedura si occupa dell'allocazione dello spazio di memoria per la struttura
      dati tortella_packet
  * e del relativo setting dei campi con i valori contenuti nei parametri d'input.
  */
tortella_packet *tortella_create_packet(tortella_header *header, char *desc, char *data)
    ;

/**
  * funzione che si occupa della creazione della struttura dati tortella_header e del
      relativo setting
  * dei vari campi dati contenutti nella struttura stessa.
  */
tortella_header *tortella_create_header(u_int8 id, u_int4 desc_id, u_int8 sender_id,
    u_int8 recv_id, u_int4 desc_len, u_int4 data_len);
```

```
/**
  * funzione che sioccupa della creazione contemporanea sia della struttura dati
      relativa all'header del pacchetto,
  * sia della struttura dati relativo al l'intero pacchetto tortella. In realta' tale
      funzione contiene la chiamata
  * alla funzione tortella_create_header e tortella_create_packet.
  */
tortella_packet *tortella_create_packet_header(u_int8 id, u_int4 desc_id, u_int8
    sender_id, u_int8 recv_id, u_int4 desc_len, u_int4 data_len, char *desc, char *data)
    ;

/**
  * funzione che ritorna il puntatore all'header del pacchetto, facendo un casting del
      buffer,
  * il quale contiene come dati iniziali quelli relativi all'header.
  */
tortella_header* tortella_get_header(const char *buffer);

char *tortella_get_desc(const char *buffer);

char *tortella_get_data(const char *buffer);

#endif //!TORTELLA_PROTOCOL_H
```

1.11 tortellaprotocol.c

```
#include "tortellaprotocol.h"

/**
 * Ha il compito di eseguire il parser del pacchetto tortella.
 * In paritcolare prende in input il pacchetto, memorizzato nella sua
 * struttura dati, e restituisce tutto il suo contenuto in un buffer di caratteri.
 * Inoltre il parametro len ritorna la lunghezza di tale buffer.
 */
char *tortella_bin_to_char(tortella_packet *packet, u_int4 *len) {

 char *header = (char *) packet->header;
 char *desc = (char *) packet->desc;
 u_int4 header_len = sizeof(tortella_header);
 u_int4 desc_len = packet->header->desc_len;
 u_int4 data_len = packet->header->data_len;

 *len = header_len + desc_len + data_len;
 char *ret = calloc(header_len + desc_len + data_len, 1);
 memcpy(ret, header, header_len);
 char *iter = ret;
 iter += header_len;

 memcpy(iter, desc, desc_len);
 iter += desc_len;              //!Si posiziona all'inizio del campo data
 memcpy(iter, packet->data, data_len);

 iter += data_len;

 return ret;
}

/**
  * Svolge le funzionalita' di parser inverso rispetto alla funzione
      tortella_bin_to_char.
  * La procedura riceve come parametro il buffer, contenente i dati, i quali vengono
      memorizzati nella
  * struttura dati tortella_packet.
  */
tortella_packet *tortella_char_to_bin(char *packet) {

 tortella_packet *ret = (tortella_packet *) calloc(1, sizeof(tortella_packet));
 tortella_header *header = (tortella_header *) packet;
 char *desc = (packet + sizeof(tortella_header));
 char *data = NULL;

 u_int4 desc_id = header->desc_id;

 data = (packet + sizeof(tortella_header) + header->data_len);

 ret->header = header;
 ret->desc = desc;
 ret->data = data;

 return ret;
}

/**
  * funzione adibita alla stampa ordinata di tutte le informazioni contenute nella
      struttura dati
  * tortella_packet, ricevuta come parametro d'input.
  */
```

```
void print_packet(tortella_packet * packet) {
 if (packet != NULL) {
  printf("--tortella_header--\n");
  printf("id:         %lld\n", packet->header->id);
  printf("desc_id:    %d\n", packet->header->desc_id);
  printf("sender_id: %lld\n", packet->header->sender_id);
  printf("recv_id:    %lld\n", packet->header->recv_id);
  printf("timestamp: %ld\n", packet->header->timestamp);
  printf("desc_len:   %d\n", packet->header->desc_len);
  printf("data_len:   %d\n", packet->header->data_len);

  if(packet->header->desc_id==LIST_ID) {
   printf("--list_desc--\n");
  }
  else if(packet->header->desc_id==LISTHITS_ID) {
   printf("--listhits_desc--\n");
  }
  else if(packet->header->desc_id==JOIN_ID) {
   join_desc *join = (join_desc*)packet->desc;
   printf("--join_desc--\n");
   printf("status:  %d\n", join->status);
   printf("chat_id: %lld\n", join->chat_id);
  }

  printf("data: %s\n", dump_data(packet->data, packet->header->data_len));
 }
}

/**
  * funzione che si occupa della creazione del pacchetto tortella.
  * La procedura si occupa dell'allocazione dello spazio di memoria per la struttura
      dati tortella_packet
  * e del relativo setting dei campi con i valori contenuti nei parametri d'input.
  */
tortella_packet *tortella_create_packet(tortella_header *header, char *desc, char *data)
     {

 tortella_packet *packet = (tortella_packet*)calloc(1, sizeof(tortella_packet));
 packet->header = header;
 packet->desc = desc;
 packet->data = data;

 return packet;
}

/**
  * funzione che si occupa della creazione della struttura dati tortella_header e del
      relativo setting
  * dei vari campi dati contenutti nella struttura stessa.
  */
tortella_header *tortella_create_header(u_int8 id, u_int4 desc_id, u_int8 sender_id,
    u_int8 recv_id, u_int4 desc_len, u_int4 data_len) {

 tortella_header *header = (tortella_header*)calloc(1, sizeof(tortella_header));
 header->id = id;
 header->desc_id = desc_id;
 header->sender_id = sender_id;
 header->recv_id = recv_id;
 header->desc_len = desc_len;
 header->data_len = data_len;

 return header;
}
```

```
/**
  * funzione che sioccupa della creazione contemporanea sia della struttura dati
      relativa all'header del pacchetto,
  * sia della struttura dati relativo al l'intero pacchetto tortella. In realta' tale
      funzione contiene la chiamata
  * alla funzione tortella_create_header e tortella_create_packet.
  */
tortella_packet *tortella_create_packet_header(u_int8 id, u_int4 desc_id, u_int8
    sender_id, u_int8 recv_id, u_int4 desc_len, u_int4 data_len, char *desc, char *data)
     {

 tortella_header *header = tortella_create_header(id, desc_id, sender_id, recv_id,
     desc_len, data_len);
 tortella_packet *packet = tortella_create_packet(header, desc, data);

 return packet;
}

/**
  * funzione che ritorna il puntatore all'header del pacchetto, facendo un casting del
      buffer,
  * il quale contiene come dati iniziali quelli relativi all'header.
  */
tortella_header* tortella_get_header(const char *buffer) {
 if(buffer==NULL)
  return NULL;
 tortella_header *header = (tortella_header*)buffer;

 return header;
}

char *tortella_get_desc(const char *buffer) {
 if(buffer==NULL)
  return NULL;
 tortella_header *header = tortella_get_header(buffer);
 char *desc = (char*)calloc(header->desc_len, 1);
 memcpy(desc, buffer+sizeof(tortella_header), header->desc_len);

 return desc;
}

char *tortella_get_data(const char *buffer) {
 if(buffer==NULL)
  return NULL;
 tortella_header *header = tortella_get_header(buffer);
 char *data = (char*)calloc(header->data_len+1, 1);
 memcpy(data, buffer+sizeof(tortella_header)+header->desc_len, header->data_len);

 return data;
}
```

1.12 httpmanager.h

```
#ifndef HTTP_MANAGER_H
#define HTTP_MANAGER_H

#include "common.h"
#include "tortellaprotocol.h"
#include "socketmanager.h"
#include "utils.h"
#include "logger.h"

#define HTTP_REQ_GET              0x40
#define HTTP_RES_GET              0x41
#define HTTP_REQ_POST             0x42
#define HTTP_RES_POST             0x43
#define HTTP_STATUS_OK            200
#define HTTP_STATUS_CERROR        400
#define HTTP_STATUS_SERROR        500

#define HTTP_REQ_POST_LINE        "POST * HTTP/1.1"
#define HTTP_AGENT                        "User-Agent: "
#define HTTP_REQ_RANGE            "Range: bytes="
#define HTTP_CONNECTION           "Connection: "
#define HTTP_CONTENT_TYPE         "Content-Type: "
#define HTTP_CONTENT_LEN          "Content-Length: "
#define HTTP_SERVER                       "Server: "

#define HTTP_OK                   "HTTP/1.1 200 OK"
#define HTTP_CERROR               "HTTP/1.1 400 Bad Request"
#define HTTP_SERROR               "HTTP/1.1 500 Internal Server Error"

//!header della request http
struct http_header_request {
 char *request;
 char *user_agent;
 u_int4 range_start;
 u_int4 range_end;
 u_int4 content_len;
 char *connection;
};
typedef struct http_header_request http_header_request;

//!header della response http
struct http_header_response {
 char *response;
 char *server;
 char *content_type;
 u_int4 content_len;
};
typedef struct http_header_response http_header_response;

/**
  * pacchetto http, composto da tipo, header (request o response), pacchetto tortella,
  * pacchetto tortella convertito in stringa e relativa lunghezza.
  */
struct http_packet {
 u_int4 type;
 http_header_request *header_request;
 http_header_response *header_response;
 tortella_packet *data;
 char *data_string;
 u_int4 data_len;
};
```

```
typedef struct http_packet http_packet;

/**
  * Creazione del pacchetto http. Converte il pacchetto tortella in stringa e crea
  * il pacchetto a seconda del tipo necessario differenziando il tipo request da
  * quello response in modo da creare i rispettivi header
  */
http_packet *http_create_packet(tortella_packet *packet, u_int4 type, u_int4 status,
     char *filename, u_int4 range_start, u_int4 range_end, char *data, u_int4 data_len);

/**
  * Crea l'header dedicato alla request settando tutti i campi in modo appropriato
  */
http_header_request *http_create_header_request(http_header_request *header, u_int4 type
     , char *filename, u_int4 range_start, u_int4 range_end, u_int4 data_len);

/**
  * Crea l'header dedicato alla response settando tutti i campi in modo opportuno
  */
http_header_response *http_create_header_response(http_header_response *header, u_int4
     type, u_int4 status, u_int4 content_len);

/**
  * Parsing del pacchetto http da binario a puntatore a carattere,
  */
char *http_bin_to_char(http_packet *packet, int *len);

/**
  * Parsing del parametro buffer in un pacchetto http.
  */
http_packet *http_char_to_bin(const char *buffer);

/**
  * Ritorna il valore contenuto nel campo rappresentato da name all'interno del
  * pacchetto (buffer).
  */
char *http_get_value(const char *buffer, const char *name);

/**
  * Ritorna la riga i-esima del pacchetto (buffer) specificata nel parametro num.
  */
char *http_get_line(const char *buffer, u_int4 num);

#endif //!HTTP_MANAGER_H
```

1.13 httpmanager.c

```
/**
 * HTTP_REQ_GET
 * GET filename HTTP/1.1
 * User-Agent: TorTella/0.1
 * Range: bytes=start-end
 * Connection: Keep-Alive
 * ...tortella_packet...
 *
 * HTTP_RES_GET
 * HTTP/1.1 200 OK
 * Server: TorTella/0.1
 * Content-Type: application/binary
 * Content-Length: 4
 * ...data...
 *
 *
 * HTTP_REQ_POST
 * POST * HTTP/1.1
 * User-Agent: TorTella/0.1
 * Connection: Keep-Alive
 * Content-Length: 4 oppure chunked
 * ...data...
 *
 * HTTP_RES_POST
 * HTTP/1.1 200 OK
 * Server: TorTella/0.1
 * Content-Type: application/binary
 * Content-Length: 4
 * non dovrebbero esserci dati
 */

#include "httpmanager.h"

/**
 * Creazione del pacchetto http. Converte il pacchetto tortella in stringa e crea
 * il pacchetto a seconda del tipo necessario differenziando il tipo request da
 * quello response in modo da creare i rispettivi header
 */
http_packet *http_create_packet(tortella_packet *packet, u_int4 type, u_int4 status,
     char *filename, u_int4 range_start, u_int4 range_end, char *data, u_int4 data_len) {
 char *temp;
 u_int4 tortella_len = 0;
 http_packet *ret = NULL;
 if(type==HTTP_REQ_POST) {
  //!conversione del pacchetto http in stringa
  temp = tortella_bin_to_char(packet, &tortella_len);
  if(temp==NULL)
   return NULL;
 }
 if(type==HTTP_REQ_POST || type==HTTP_REQ_GET) {
  logger(HTTP_INFO, "[http_create_packet]Creating packet POST or GET request\n");
  http_header_request *request = (http_header_request*)calloc(1, sizeof(
       http_header_request));
  //!creazione dell'header della request
  request = http_create_header_request(request, type, filename, range_start, range_end,
       tortella_len);
  ret = (http_packet*)malloc(sizeof(http_packet));
  //!settaggio dei parametri dell'header
  ret->header_request = request;
  ret->header_response = NULL;
  if(type==HTTP_REQ_POST) {
```

```
    ret->data = packet;
    ret->data_string = temp;
    ret->data_len = tortella_len;
   }
   else if(type==HTTP_REQ_GET) {
    ret->data = NULL;
    ret->data_string = NULL;
    ret->data_len = 0;
   }

   ret->type = type;

  }
  else if((type==HTTP_RES_POST || type==HTTP_RES_GET) && status>=HTTP_STATUS_OK) {
   http_header_response *response = (http_header_response*)malloc(sizeof(
       http_header_response));
   logger(HTTP_INFO, "[http_create_packet]Creating packet POST or GET response status %d\
       n", status);
   //!creazione dell'header della response
   response = http_create_header_response(response, type, status, data_len);
   if(response==NULL) {
    logger(HTTP_INFO, "[http_create_packet]Response not created\n");
    return NULL;
   }
   ret = (http_packet*)calloc(1, sizeof(http_packet));
   //!settaggio dei parametri dell'header
   ret->header_request = NULL;
   ret->header_response = response;
   ret->data = NULL;
   ret->data_string = data;
   ret->data_len = data_len;
   ret->type = type;
  }
  return ret;
}

/**
  * Crea l'header dedicato alla request settando tutti i campi in modo appropriato
  */
http_header_request *http_create_header_request(http_header_request *header, u_int4 type
     , char *filename, \
                                                                                u_int4
                                                                                    range_s
                                                                                    ,
                                                                                    u_int4
                                                                                    range_e
                                                                                    ,
                                                                                    u_int4
                                                                                    data_le
                                                                                    )
                                                                                    {
                                                                                //
                                                                                     !
                                                                                     settag
                                                                                     dei
```

```
        parametri

        comuni

        dell
        '
        header

header
        ->
        user_agent

        =

        "
        TorTella
        /0.1
        "
        ;

header
        ->
        connection

        =

        "
        Keep
        -
        Alive
        "
        ;

header
        ->
        range_start

        =

        range_start
        ;

header
        ->
        range_end

        =

        range_end
        ;

//
        !
        settaggio

        dei

        parametri

        in

        relazione
```

```
al
tipo
di
pacche
if
(
type
==
HTTP_R
)
{
header
->
reque
=
"
POST
*
HTTP
/1.1
"
;
header
->
conte
=
data_
;
}
else
if
(
type
==
HTTP_R
)
{
header
->
reque
=
callo
(4+
strle
```

```
            (
            filename
            )
            +9,

            1)
            ;

        sprintf
            (
            header
            ->
            request
            ,

            "
            GET

            %
            s

            HTTP
            /1.1
            "
            ,

            filename
            )
            ;

    }

    else

        header
            =
            NULL
            ;

    return

            header
            ;

}
```

```
/**
  * Crea l'header dedicato alla response settando tutti i campi in modo opportuno
  */
http_header_response *http_create_header_response(http_header_response *header, u_int4
     type, u_int4 status, u_int4 content_len) {

 //!settaggio dei parametri dell'header response
 header->server = "TorTella/0.1";
 header->content_type = "application/binary";
 header->content_len = content_len;

 switch(status) {
  case HTTP_STATUS_OK:
        header->response = HTTP_OK;
        break;
   case HTTP_STATUS_CERROR:
        header->response = HTTP_CERROR;
```

```
            break;
        case HTTP_STATUS_SERROR:
            header->response = HTTP_SERROR;
            break;
        default:
            header->response = NULL;
    }

    return header;
}

/**
  * Parsing del pacchetto http da binario a puntatore a carattere,
  */
char *http_bin_to_char(http_packet *packet, int *len) {
    char *buffer = NULL;
    u_int4 type = packet->type;
    logger(HTTP_INFO, "[http_bin_to_char]Converting bin to char\n");
    if(packet->header_request!=NULL) {
        http_header_request *header_request = packet->header_request;
        if(type==HTTP_REQ_POST) {
            //!parsing del pacchetto di invio HTTP REQ POST
            logger(HTTP_INFO, "[http_bin_to_char]HTTP_REQ_POST\n");
            buffer = calloc(strlen(header_request->request)+2\
                                        +strlen(header_request->user_agent)+strlen(HTTP_AGENT
                                            )+2\
                                        +strlen(HTTP_CONTENT_LEN)+strlen(to_string(
                                            header_request->content_len))+2\
                                        +strlen(HTTP_CONNECTION)+strlen(header_request->
                                            connection)+2+2\
                                        +sizeof(tortella_header)+packet->data->header->
                                            desc_len+packet->data->header->data_len+1+1, 1);
                                            //!TODO: memory leak

            sprintf(buffer, "%s\r\nUser-Agent: %s\r\nContent-Length: %d\r\nConnection: %s\r\n\r\n
                ", header_request->request,\
                            header_request->user_agent, header_request->content_len,
                                header_request->connection);

            int buflen = strlen(buffer);
            char *iter = buffer;
            iter += buflen;
            memcpy(iter, packet->data_string, header_request->content_len);
            iter += header_request->content_len;
            memcpy(iter, "\n", 1);

            *len = buflen+header_request->content_len+1;

        }
        else if(type==HTTP_REQ_GET) {
            //!parsing del pacchetto di invio HTTP REQ GET
            buffer = calloc(strlen(header_request->request)+2\
                                        +strlen(header_request->user_agent)+strlen(HTTP_AGENT
                                            )+2\
                                        +strlen(HTTP_REQ_RANGE)+strlen(to_string(
                                            header_request->range_start))+1+strlen(to_string(
                                            header_request->range_end))+2\
                                        +strlen(HTTP_CONNECTION)+strlen(header_request->
                                            connection)+2+2, 1);

            sprintf(buffer, "%s\r\nUser-Agent: %s\r\nRange: bytes=%d-%d\r\nConnection: %s\r\n\r\n
                ", header_request->request, header_request->user_agent, \
                            header_request->range_start, header_request->range_end,
                                header_request->connection);
```

```
   *len = strlen(buffer);
  }
 }
 else if(packet->header_response!=NULL) {
  //!parsing del pacchetto di risposta.
  http_header_response *header_response = packet->header_response;
  if(type==HTTP_RES_GET || type==HTTP_RES_POST) {

   int con_len = strlen(to_string(header_response->content_len));

   if(type==HTTP_RES_POST) {
        header_response->content_type = "text/html";
   }

   if(packet->data_string!=NULL) {
        //!parsing dei dati, qualora presenti
        *len = strlen(header_response->response)+2
        +strlen(HTTP_SERVER)+strlen(header_response->server)+2
         +strlen(HTTP_CONTENT_TYPE)+strlen(header_response->content_type)+2
         +strlen(HTTP_CONTENT_LEN)+con_len+2+2
         +packet->data_len+1;

        buffer = calloc(*len, 1);

        sprintf(buffer, "%s\r\nServer: %s\r\nContent-Type: %s\r\nContent-Length: %d\r\n\
             r\n", header_response->response, header_response->server, header_response->
             content_type, \
                        header_response->content_len);

        char *temp = buffer;
        temp+=strlen(buffer);

        memcpy(temp, packet->data_string, packet->data_len);

        temp+=packet->data_len;
        memcpy(temp, "\n", 1);
   }
   else {

        *len = strlen(header_response->response)+2
         +strlen(HTTP_SERVER)+strlen(header_response->server)+2
         +strlen(HTTP_CONTENT_TYPE)+strlen(header_response->content_type)+2
         +strlen(HTTP_CONTENT_LEN)+con_len+2+2;

        buffer = calloc((*len)+1, 1);

        snprintf(buffer, (*len)+1, "%s\r\nServer: %s\r\nContent-Type: %s\r\nContent-
             Length: %d\r\n\r\n", header_response->response, header_response->server,
             header_response->content_type, \
                         header_response->content_len);
   }
  }

 }

 return buffer;
}

/**
  * Parsing del parametro buffer in un pacchetto http.
  */
http_packet *http_char_to_bin(const char *buffer) {
 char *saveptr;
```

```
http_packet *packet = NULL;

if(buffer!=NULL) {
 packet = (http_packet*)calloc(1, sizeof(http_packet));
 char *result;

 if((result=strstr(buffer, "GET"))!=NULL) {
  //!parsing di un pacchetto di tipo GET
  packet->type = HTTP_REQ_GET;
  http_header_request *header_request = (http_header_request*)calloc(1, sizeof(
      http_header_request));
  //!settaggio dei campi dell'header
  header_request->request = http_get_line(buffer, 0);

  header_request->user_agent = http_get_value(buffer, HTTP_AGENT);

  char *range = http_get_value(buffer, HTTP_REQ_RANGE);
  header_request->range_start = atoi(strtok_r(range, "-",&saveptr));
  header_request->range_end = atoi(strtok_r(NULL, "-",&saveptr));

  header_request->connection = http_get_value(buffer, HTTP_CONNECTION);

  //!settaggio dei campi del pacchetto
  packet->header_request = header_request;
  packet->data = NULL;
  packet->data_string = NULL;
  packet->data_len = 0;

 }
 else if((result=strstr(buffer, "POST"))!=NULL) {
  //!parsing di un pacchetto di tipo POST
  packet->type = HTTP_REQ_POST;
  http_header_request *header_request = NULL;

  header_request = calloc(1, sizeof(http_header_request));
  //!settaggio dei campi dell'header
  header_request->request = http_get_line(buffer, 0);
  header_request->user_agent = http_get_value(buffer, HTTP_AGENT);
  header_request->content_len = atoi(http_get_value(buffer, HTTP_CONTENT_LEN));
  header_request->connection = http_get_value(buffer, HTTP_CONNECTION);
  packet->header_request = header_request;
  packet->data_string = strstr(buffer, "\r\n\r\n")+4;
  tortella_packet *t_packet = (tortella_packet*)malloc(sizeof(tortella_packet));
  tortella_header *t_header = (tortella_header*)packet->data_string;
  char *t_desc = tortella_get_desc(packet->data_string);
  char *t_data = tortella_get_data(packet->data_string);
  //!settaggio dei campi del pacchetto
  t_packet->header = t_header;
  t_packet->desc = t_desc;
  t_packet->data = t_data;
  packet->data = t_packet;
  packet->data_len = header_request->content_len;
 }
 else if((result=strstr(buffer, "application/binary"))!=NULL) {
  //!parsing del pacchetto di ricezione di una GET
  packet->type = HTTP_RES_GET;
  http_header_response *header_response = (http_header_response*)malloc(sizeof(
      http_header_response));
  //!settaggio dell'header di risposta
  header_response->response = http_get_line(buffer, 0);

  header_response->server = http_get_value(buffer, HTTP_SERVER);

  header_response->content_len = atoi(http_get_value(buffer, HTTP_CONTENT_LEN));
```

```
    //!settaggio dei campi del pacchetto
    packet->header_response = header_response;
    packet->data_string = strstr(buffer, "\r\n\r\n")+4;
    packet->data = NULL;
    packet->data_len = header_response->content_len;
   }
   else if((result=strstr(buffer, "text/html"))!=NULL) {
    //!parsing del pacchetto di ricezione di una POST
    packet->type = HTTP_RES_POST;
    http_header_response *header_response = (http_header_response*)malloc(sizeof(
        http_header_response));

    //!settaggio dei campi dell'header di risposta
    header_response->response = http_get_line(buffer, 0);

    header_response->server = http_get_value(buffer, HTTP_SERVER);

    header_response->content_len = atoi(http_get_value(buffer, HTTP_CONTENT_LEN));

    //!settaggio dei campi del pacchetto
    packet->header_response = header_response;
    packet->data_string = strstr(buffer, "\r\n\r\n")+4;
    packet->data = NULL;
    packet->data_len = header_response->content_len;
   }
  }
  return packet;

}

/**
  * Ritorna il valore contenuto nel campo rappresentato da name all'interno del
  * pacchetto (buffer).
  */
char *http_get_value(const char *buffer, const char *name) {
 char *saveptr;
 char *buf = calloc(strlen(buffer), 1);
 memcpy(buf, buffer, strlen(buffer));
 char *token;
 char *ret;

 token=strtok_r(buf, "\r\n",&saveptr);
 if(token==NULL)
  return NULL;
 do {
  if((ret=strstr(token, name))!=NULL)
   return ret+=strlen(name);
 }while((token=strtok_r(NULL, "\r\n",&saveptr))!=NULL);
 return NULL;

}

/**
  * Ritorna la riga i-esima del pacchetto (buffer) specificata nel parametro num.
  */
char *http_get_line(const char *buffer, u_int4 num) {
 char *saveptr;
 char *buf = calloc(strlen(buffer), 1);
 memcpy(buf, buffer, strlen(buffer));
 char *token;
 u_int4 counter = -1;

 token=strtok_r(buf, "\r\n",&saveptr);
 if(token==NULL)
```

```
  return NULL;
 counter++;
 do {
  if(counter==num)
   return token;
  counter++;
 }while((token=strtok_r(NULL, "\r\n",&saveptr))!=NULL);
 return NULL;

}
```

1.14 socketmanager.h

```
#define SOCKETMANAGER_H

#include <sys/types.h>
#include <sys/socket.h>
#include "common.h"
#include "httpmanager.h"

/**
 * Crea un socket di connessione remota ovvero
 * si connette ad un server remoto.
 */
int create_tcp_socket(const char* dst_ip, int dst_port);

/**
 * Crea un socket d'ascolto, ovvero un server TCP in attesa di connessioni.
 */
int create_listen_tcp_socket(const char* src_ip, int src_port);

/**
 * Elimina un socket ovvero chiude una connessione, mettendo il socket nello stato di
      TIME_WAIT.
 */
int delete_socket(int sock_descriptor);

/*In ascolto per una nuova connessione.
 listensocket e' il descrittore socket di ascolto (restituito da
      create_listen_tcp_socket)
  Il parametro mode ha i seguenti significati:
  - LP_WRITE, aperta la connessione invia il buffer al client
   - LP_READ, aperta la connessione attende dati, inserendoli poi nel buffer
   - LP_NONE, crea solo la nuova connessione con il client.
   */
//!int listen_packet(int listen_socket, char* buffer, unsigned int mode);
//!

/**
 * funzione che ritorna il nuovo descrittore di socket e l'indirizzo del client connesso
      ,
 * nel caso la chiamata di sistema accept() abbia avuto un esito positivo
 */
int listen_http_packet(int listen_socket);

/**
 * Legge il pacchetto in ingresso e ritorna il numero di caratteri
 */
int recv_http_packet(int con_socket, char **buffer);

/**
 * Permette di scrivere sul socket connesso specificato dal relativo descrittore
 */
int send_packet(int sock_descriptor, char* buffer, int len);

/**
 * Attende la ricezione di un pacchetto, avente come dimensione massima quella pari al
 * valore assunto dal parametro BUFFER_LEN
 */
int recv_packet(int sock_descriptor,char** buffer);

/**
 * Attende la ricezione di un pacchetto, prefissando il valore massimo(max_len) di byte
      di
```

```
 *  un blocco di dati del pacchetto
 */
int recv_sized_packet(int sock_descriptor,char** buf, int max_len);

//!char *recv_http_packet(int sock_descriptor,char* buffer, int *len);

/**
 * Permette di ottenere l'indirizzo ip e il numero di porta del peer assocciato
 * al socket, passato come parametro
 */
char *get_dest_ip(int socket);

/**
 * Permette di ottenere il numero di porta del peer assocciato
 * al socket, passato come parametro
 */
u_int4 get_dest_port(int socket);

/**
 * Permette la chiusura asimmetrica di una connessione TCP
 */
int shutdown_socket(int sock_descriptor);

#endif //!SOCKETMANAGER_H
```

1.15 socketmanager.c

```
#include "socketmanager.h"

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/**
 * Crea un socket di connessione remota ovvero
 * si connette ad un server remoto.
 */
int create_tcp_socket(const char *dst_ip, int dst_port)
{
 struct sockaddr_in sAddr;
 int sockfd;

 //! creazione socket
 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  logger(SOCK_INFO, "[create_tcp_socket]Errore nella creazione del socket tcp");
  return -1;
 }
 //!inizializzazione a 0 della struttura sockaddr_in
 memset((char *) &sAddr, 0, sizeof(sAddr));
 sAddr.sin_family = AF_INET;
 if (dst_port > 0)
  /**
        * conversione dalla rappresentazione testuale/binaria dell'indirizzo
        * al valore binario da inserire nella struttura sockaddr_in
        */
  sAddr.sin_port = htons(dst_port);
 else {
  logger(SOCK_INFO, "[create_tcp_socket]Errore nella porta: [%d]\n", dst_port);
  return -1;
 }
 //!Per convertire l' indirizzo IP dalla notazione puntata in formato ASCII al network
      byte order in formato binario
 if (inet_pton(AF_INET, dst_ip, &sAddr.sin_addr) <= 0) {
  logger(SOCK_INFO, "[create_tcp_socket]Errore nella conversione dell'indirizzo IP: [%s
       ]\n", dst_ip);
  return -1;
 }
 //!funzione che permette al client TCP di aprire una connessione ad un server TCP
 if (connect(sockfd, (struct sockaddr *) &sAddr, sizeof(sAddr)) < 0) {
  logger(SOCK_INFO, "[create_tcp_socket]Unable to connect to: %s:%d\n", dst_ip, dst_port
       );
  return -1;
 }

 int reuse = 1;

 /**
   * funzione che permette di impostare le caratteristiche del socket.
   * In questa funzione grazie all'utilizzo del parametro SO_KEEPALIVE si
   * ha la possibilita' di gestire le persistenza delle connessioni.
   */
 if(setsockopt(sockfd, SOL_SOCKET, SO_KEEPALIVE, &reuse, sizeof(int)) < 0) {
  logger(SOCK_INFO, "[create_tcp_socket]Error in setsockopt SO_KEEPALIVE\n");
```

```
  return -1;
 }

 return sockfd;
}

/*
  * Crea un socket d'ascolto, ovvero un server TCP in attesa di connessioni.
  */
int create_listen_tcp_socket(const char *src_ip, int src_port)
{
 int listenfd = 0;
 struct sockaddr_in sAddr;
 //! creazione socket
 if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  logger(SOCK_INFO, "[create_listen_tcp_socket]Errore nella creazione del socket tcp");
  return -1;
 }
 //!inizializzazione a 0 della struttura sockaddr_in
 memset((char *) &sAddr, 0, sizeof(sAddr));
 sAddr.sin_family = AF_INET;
 if (src_port > 0)
  /**
        * conversione dalla rappresentazione testuale/binaria dell'indirizzo
        * al valore binario da inserire nella struttura sockaddr_in
        */
  sAddr.sin_port = htons(src_port);
 else {
  logger(SOCK_INFO, "[create_listen_tcp_socket]Errore nella porta: [%d]\n", src_port);
  return -1;
 }
 //!Per convertire l' indirizzo IP dalla notazione puntata in formato ASCII al network
      byte order in formato binario
 if (inet_pton(AF_INET, src_ip, &sAddr.sin_addr) <= 0) {
  logger(SOCK_INFO, "[create_listen_tcp_socket]Errore nella conversione dell'indirizzo
       IP: [%s]\n", src_ip);
  return -1;
 }

 int reuse = 1;
 /**
   * funzione che permette di impostare le caratteristiche del socket.
   * In questa funzione grazie all'utilizzo del parametro SO_REUSEADDR si
   * ha la possibilita' di riutilizzare un indirizzo locale, modificando il comportamento
   * della bind che fallisce in caso l'indirizzo sia gia' in uso in un altro socket.
   */
 if(setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(int)) < 0) {
  logger(SOCK_INFO, "[create_listen_tcp_socket]Error in setsockopt SO_REUSEADDR\n");
  return -1;
 }
 /**
   * funzione che permette di impostare le caratteristiche del socket.
   * In questa funzione grazie all'utilizzo del parametro SO_KEEPALIVE si
   * ha la possibilita' di gestire le persistenza delle connessioni.
   */
 if(setsockopt(listenfd, SOL_SOCKET, SO_KEEPALIVE, &reuse, sizeof(int)) < 0) {
  logger(SOCK_INFO, "[create_listen_tcp_socket]Error in setsockopt SO_KEEPALIVE\n");
  return -1;
 }

 /**
   * funzione utilizzata per la fase di inizializzazione dell'indirizzo ip e del
   * numero di porta utilizzati dal socket, inoltre serve a far sapere al SO a quale
   * processo vanno inviati i dati ricevuti dalla rete.
```

```
   */
 if ((bind(listenfd, (struct sockaddr *) &sAddr, sizeof(sAddr))) < 0) {
  logger(SOCK_INFO, "[create_listen_tcp_socket]Bind error\n");
  return -1;
 }
 /**
   * funzione utilizzata per porre il socket creato dallo stato di CLOSED a quello
   * di LISTEN, specificando il numero di connessioni che possono essere accettate dal
       server
   */
 if (listen(listenfd, conf_get_qlen()) < 0) {
  logger(SOCK_INFO, "[create_listen_tcp_socket]Listen error\n");
  return -1;
 }

 return listenfd;
}

/**
  * Elimina un socket ovvero chiude una connessione, mettendo il socket nello stato di
      TIME_WAIT.
  */
int delete_socket(int sock_descriptor)
{
 if (sock_descriptor < 0) {
  logger(SOCK_INFO, "[delete_socket]SocketDescriptor error: [%d]", sock_descriptor);
  return -1;
 }

 //! funzione utilizzata per permettere la chiusura attiva della connessione
      identificata dal descrittore
 if (close(sock_descriptor) < 0) {
  logger(SOCK_INFO, "[delete_socket]Socket shutdown error");
  return -1;
 }
 return 0;
}
/**
  * funzione che ritorna il nuovo descrittore di socket e l'indirizzo del client
      connesso,
  * nel caso la chiamata di sistema accept() abbia avuto un esito positivo
  */
int listen_http_packet(int listen_socket)
{
 int connFd = 0;
 //!permette  al server di prendere dal backlog la prima connessione completato sul
      socket specificato
 if ((connFd=accept(listen_socket, (struct sockaddr *) NULL, NULL)) < 0) {
  logger(SOCK_INFO, "[listen_http_packet]Errore nell'accept\n");
  return -1;
 }

 return connFd;
}

/**
  * Legge il pacchetto in ingresso e ritorna il numero di caratteri
  */
int recv_http_packet(int connFd, char **buffer) {
 int len = 0;
 if ((len=recv_packet(connFd, buffer)) < 0) {
  logger(SOCK_INFO, "[recv_http_packet]Errore in lettura!");
  return -1;
 }
```

```
 if(*buffer==NULL) {
  logger(SOCK_INFO, "[recv_http_packet]buffer null\n");
  return -2;
 }

 return len;
}
/**
  * Permette di scrivere sul socket connesso specificato dal relativo descrittore
  */
int send_packet(int sock_descriptor, char *buffer, int len)
{
 int char_write = 0;
 if (sock_descriptor < 0) {
  logger(SOCK_INFO, "[send_packet]Socket descriptor not valid, sock_descriptor = %d\n",
       sock_descriptor);
  return -1;
 }
 if(buffer==NULL) {
  logger(SOCK_INFO, "[send_packet]Buffer non valido, sock_descriptor = %d\n",
       sock_descriptor);
  return -3;
 }
 //! Questa blocco si potrebbe ritentare per n volte, dove n e' un
 //! parametro di configurazione.
 if ((char_write=write(sock_descriptor, buffer, len)) != len) {
  logger(SOCK_INFO, "[send_packet]Perdita dati in trasmissione");
  return -2;
 }

 return char_write;
}

/**
  * Attende la ricezione di un pacchetto, avente come dimensione massima quella pari al
  * valore assunto dal parametro BUFFER_LEN
  */
int recv_packet(int sock_descriptor, char **buffer)
{
 return recv_sized_packet(sock_descriptor, buffer, conf_get_buffer_len());
}

/**
  * Attende la ricezione di un pacchetto, prefissando il valore massimo(max_len) di byte
       di
  *  un blocco di dati del pacchetto
  */
int recv_sized_packet(int sock_descriptor, char **buf, int max_len)
{
 u_int4 char_read = 0;
 int flag = 0;
 int len = 0;
 if (max_len < 0)
  return -1;

 char *tmp = calloc(max_len, 1);
 *buf = calloc(max_len, 1);
 char *buffer = *buf;
 char *iter = buffer;
 char *buf2;

 if (sock_descriptor < 0) {
  logger(SOCK_INFO, "[recv_sized_packet]Socket descriptor not valid, sock_descriptor = %
       d", sock_descriptor);
```

```
  return -1;
 }
 logger(SOCK_INFO, "[recv_sized_packet]Receiving from %d\n", sock_descriptor);
 while(!flag) {
  char_read = read(sock_descriptor, tmp, max_len);
  if(char_read==0) {
   flag=1;
   continue;
  }
  len += char_read;

  if(len-char_read<0 || len<0) {
   logger(SOCK_INFO, "[recv_sized_packet]Impossibile allocare memoria\n");
   return -1;
  }
  buf2 = calloc(len-char_read, 1);
  memcpy(buf2, buffer, len-char_read);
  buffer = calloc(len, 1);
  memcpy(buffer, buf2, len-char_read);

  memcpy(iter, tmp, char_read);
  iter += char_read;
  memset(tmp, 0, max_len);

  if(char_read < max_len){
   flag = 1;
  }
 }

 if (len < 0) {
  logger(SOCK_INFO, "[recv_sized_packet]read error");
  return -1;
 }
 return len;
}

/**
  * Permette di ottenere l'indirizzo ip del peer associato
  * al socket, identificato dal descrittore passato come parametro
  */
char *get_dest_ip(int socket) {

 struct sockaddr_in peer;
 memset((char *) &peer, 0, sizeof(peer));
 u_int4 peer_len;
 peer_len = sizeof(peer);
 if (getpeername(socket, (struct sockaddr *)&peer, &peer_len) == -1) {
  logger(SOCK_INFO, "[get_dest_ip]Failed\n");
  return NULL;
 }
 logger(SOCK_INFO, "[get_dest_ip]Peer's IP address is: %s\n", inet_ntoa(peer.sin_addr));
 return inet_ntoa(peer.sin_addr);
}
/**
  * Permette di ottenere il numero di porta del peer associato
  * al socket, identificato dal descrittore passato come parametro
  */
u_int4 get_dest_port(int socket) {

 struct sockaddr_in peer;
 memset((char *) &peer, 0, sizeof(peer));
 u_int4 peer_len;
 peer_len = sizeof(peer);
 if (getpeername(socket, (struct sockaddr *)&peer, &peer_len) == -1) {
```

```
  logger(SOCK_INFO, "[get_dest_port]Failed\n");
  return -1;
 }
 logger(SOCK_INFO, "[get_dest_port]Peer's port is: %d\n", (int) ntohs(peer.sin_port));
 return (u_int4) ntohs(peer.sin_port);
}
/**
  * Permette la chiusura asimmetrica di una connessione TCP
  */
int shutdown_socket(int sock_descriptor)
{
 if (sock_descriptor < 0) {
  logger(SOCK_INFO, "[shutdown_socket]SocketDescriptor error: [%d]", sock_descriptor);
  return -1;
 }


 if (shutdown(sock_descriptor, SHUT_RDWR) < 0) {
  logger(SOCK_INFO, "[shutdown_socket]Socket shutdown error");
  return -1;
 }
 return 0;
}
```

1.16 servent.h

```
/**
 * \file Servent
 *
 * \brief Gestisce il comportamento dei peer
 *
 * Crea le connessioni tra peer, gestisce l'invio e la ricezione dei pacchetti.
 *
 * \author $Author: Simone Notargiaco, Lorenzo Fortunato, Ibrahim Khalili $
 *
 * \version $Revision: 0.1 $
 *
 * \date $Date: 2008/06/18 14:16:20 $
 *
 * Contact: notargiacomo.s@gmail.com
 */

#ifndef SERVENT_H
#define SERVENT_H

#define _GNU_SOURCE

#include "common.h"
#include "packetmanager.h"
#include "utils.h"
#include "datamanager.h"
#include "init.h"
#include "routemanager.h"
#include <pthread.h>
#include <signal.h>
#include <unistd.h>
#include "controller.h"
#include <glib.h>
#include "logger.h"
#include "confmanager.h"
#include <time.h>
#include <sys/stat.h>
#include <string.h>

#define TIMEOUT "timeout error"

/**
 * Tale struttura dati viene utilizzata per le operazioni di comunicazioni
 * con un servente a cui si e' connessi. Ne e' presente una anche per il peer locale
 */
struct servent_data {
 u_int8 id;                              //!ID del peer rappresentato dalla struttura
      dati
 GQueue *queue;                  //!Coda utilizzata per serializzare le richieste di
      invio pacchetti
 GQueue *res_queue;              //!Coda utilizzata per serializzare le risposte dei
      pacchetti inviati
 char *ip;
 u_int4 port;
 u_int1 status;                  //!Stato del peer (ONLINE, BUSY, AWAY)
 char *nick;
 time_t timestamp;               //!Timestamp ricezione pacchetto

 GList *chat_list;               //!Lista delle chat a cui e' connesso il peer

 pthread_rwlock_t rwlock_data;   //!Serve per sincronizzare gli accessi ai dati del peer
```

```
 char *msg;                              //!Messaggio da inviare
 u_int4 msg_len;                         //!Lunghezza messaggio

 GList *chat_res;                //!Risultati della ricerca richiesta dal peer
 u_int1 ttl;                             //!ttl da inviare
 u_int1 hops;                    //!hops da inviare
 u_int8 packet_id;               //!ID del pacchetto da ritrasmettere

 u_int4 post_type;               //!Tipo di pacchetto da inviare
 u_int1 is_online;               //!Flag che indica se il peer e' pronto a ricevere
      pacchetti (viene impostato alla ricezione del primo PING): 1 e 0

 //!FLOODING
 u_int8 user_id_req;             //!Utente che si vuole connettere alla chat con id:
      chat_id_req
 u_int8 chat_id_req;     //!Chat a cui connettersi o creare
 u_int4 port_req;                //!PORT del join da rinviare
 char *nick_req;                         //!NICK del join da rinviare
 char *ip_req;                   //!IP del join da rinviare
 u_int1 status_req;              //!Status per il redirect del join
 char *title;                    //!Titolo chat da creare o ricercare
 u_int4 title_len;               //!Lunghezza del titolo

};
typedef struct servent_data servent_data;

static GHashTable *servent_hashtable;   //!Hashtable di servent_data, una per ogni peer

//!---Routing Hashtables-----

static GHashTable *search_packet_hashtable = NULL;

static GHashTable *join_packet_hashtable = NULL;

static GHashTable *leave_packet_hashtable = NULL;

//!-------------------------

static servent_data *local_servent;             //!Dati del peer locale

static u_int8 new_connection_counter;   //!Limite inferiore generazione ID

static pthread_t *timer_thread;                 //!identificatore timer thread;

static GList *client_fd;                                //!Lista client socket
static GList *server_fd;                                //!Lista server socket in attesa
      di connessioni
static GList *server_connection_fd;             //!Lista server socket connessi

static GList *client_thread;                    //!Lista identificatori client thread
static GList *server_thread;                    //!Lista identificatori server thread in
      attesa di connessioni
static GList *server_connection_thread; //!Lista identificatori server thread connessi

//!Macro di utilita'

#define WLOCK(servent)                  logger(SYS_INFO, "[WLOCK]Try locking %lld\n",
     servent); \
if(servent_get(servent)!=NULL) { \
pthread_rwlock_wrlock( &((servent_get(servent)->rwlock_data))); \
logger(SYS_INFO, "[WLOCK]Lock %lld\n", servent); \
}

#define RLOCK(servent)                  logger(SYS_INFO, "[RLOCK]Try locking %lld\n",
```

```
    servent); \
if(servent_get(servent)!=NULL) { \
pthread_rwlock_rdlock( &((servent_get(servent)->rwlock_data))); \
logger(SYS_INFO, "[RLOCK]Lock %lld\n", servent); \
}

#define UNLOCK(servent)                 logger(SYS_INFO, "[UNLOCK]Try unlocking %lld\n",
    servent); \
if(servent_get(servent)!=NULL) { \
pthread_rwlock_unlock( &((servent_get(servent)->rwlock_data))); \
logger(SYS_INFO, "[UNLOCK]Unlock %lld\n", servent); \
}

#define UNLOCK_F(servent)               pthread_rwlock_unlock( &(((servent)->rwlock_data
    )) ); \
logger(SYS_INFO, "[UNLOCK_F]Unlock %lld\n", servent->id);

//!Tale macro viene utilizzata per copiare un servent_data in un altro
#define COPY_SERVENT(servent, copy)                     copy=calloc(1, sizeof(
    servent_data)); \
memcpy(copy, servent, sizeof(servent_data))

/**
  * Crea un server socket
  */
int servent_create_server(char *src_ip, u_int4 src_port);

/**
  * Crea un client socket
  */
int servent_create_client(char *dst_ip, u_int4 dst_port);

/**
  * Avvia il server thread che viene usato per accettare
  * le nuove connessioni e quindi lanciare nuovi thread
  * per gestirle
  */
int servent_start_server(char *local_ip, u_int4 local_port);

/**
  * Avvia un client thread usato per gestire tutte le richieste di invio
  * pacchetti al peer associato.
  *
  * \param id Se si conosce l'id del peer a cui connettersi lo si specifica, altrimenti
      0.
  */
servent_data *servent_start_client(char *dest_ip, u_int4 dest_port, u_int8 id);

/**
  * Questa funzione viene utilizzata per il boot iniziale, in quanto si connette
  * alla lista dei peer conosciuti, presi dal file di init. Inoltre inizializza
  * tutte le variabili necessarie e avvia il server di ascolto nuove connessioni.
  *
  * \param init_servent Lista dei serventi necessari per il boot iniziale
  */
int servent_start(GList *init_servent);

/**
  * Tale funzione non fa altro che avviare il timer thread utilizzato per supportare
  * il meccanismo di failure detection.
  */
int servent_start_timer(void);

/**
```

```
  * Si connette alla lista dei peer specificati, se qualcuno non e' disponibile lo salta
      .
  */
int servent_init_connection(GList *init_servent);

/**
  * Questa funzione viene chiamata alla chiusura di TorTella, serve per chiudere
  * tutti i socket aperti (non brutalmente).
  */
void servent_close_all(void);

/**
  * Ultima funzione chiamata alla chiusura del programma, termina tutti i thread attivi.
  */
void servent_kill_all_thread(int sig);

/**
  * Inizializza: le variabili del peer locale, il seed, le hashtable, i lock etc...
  */
int servent_init(char *ip, u_int4 port, u_int1 status);

/**
  * Funzione utilizzata per il recupero delle chat conosciute da file,
  * attualmente non piu' utilizzata.
  */
void servent_init_supernode();

/**
  * Funzione utilizzata per il salvataggio delle chat conosciute su file,
  * attualmente non piu' utilizzata.
  */
void servent_close_supernode();

//!-----Gestione servent_data-----

/**
  * Restituisce il servent_data associato all'id richiesto.
  */
servent_data *servent_get(u_int8 id);

/**
  * Restituisce la lista completa delle servent_data.
  */
GList *servent_get_values(void);

/**
  * Restituisce la lista completa delle chiavi associate alle servent_data.
  */
GList *servent_get_keys(void);

/**
  * Restituisce la servent_data del peer locale.
  */
servent_data *servent_get_local(void);

//!-----Gestione Queue-----

/**
  * Aggiunge alla coda di pacchetti da inviare ad uno specifico peer.
  * In particolare si passa una servent_data contenete tutti i dati
  * necessari all'invio del pacchetto.
  */
void servent_send_packet(servent_data *sd);
```

```
/**
  * Rimuove dalla coda di pacchetti da inviare ad uno specifico peer.
  * Se non ci sono pacchetti da rimuovere rimane in attesa.
  */
servent_data *servent_pop_queue(servent_data *sd);

/**
  * Appende alla coda di risposta di uno specifico peer.
  * In particolare aggiunge la risposta ricevuta dopo l'invio di un pacchetto.
  */
void servent_append_response(servent_data *sd, const char *response);

/**
  * Rimuove dalla coda di risposta un elemento se presente,
  * nel frattempo avvia un timer per rilevare eventuali timeout di risposta.
  */
char *servent_pop_response(servent_data *sd);

/**
  * Pulisce la lista dei pacchetti ricevuti che serve per scartare
  * pacchetti con ID uguale a quelli ricevuti recentemente.
  */
void servent_flush_data(void);

//!-----Routing-----

/**
  * Ritorna l'ID del pacchetto richiesto, se presente.
  * E' una sorta di verifica presenta pacchetto.
  * Utilizzato per gestire i duplicati.
  */
char *servent_get_search_packet(u_int8 id);

/**
  * Aggiunge il pacchetto alla lista dei pacchetti ricevuti.
  */
void servent_new_search_packet(u_int8 id);

/**
  * Ritorna l'ID del pacchetto richiesto, se presente.
  * E' una sorta di verifica presenta pacchetto.
  * Utilizzato per gestire i duplicati.
  */
char *servent_get_join_packet(u_int8 id);

/**
  * Aggiunge il pacchetto alla lista dei pacchetti ricevuti.
  */
void servent_new_join_packet(u_int8 id);

/**
  * Ritorna l'ID del pacchetto richiesto, se presente.
  * E' una sorta di verifica presenta pacchetto.
  * Utilizzato per gestire i duplicati.
  */
char *servent_get_leave_packet(u_int8 id);

/**
  * Aggiunge il pacchetto alla lista dei pacchetti ricevuti.
  */
void servent_new_leave_packet(u_int8 id);

//!---------THREAD----------------
```

```
/**
  * Thread che riceve le richieste di connessione e avvia nuovi thread.
  * Ogni nuovo peer (client) che richiede di connettersi al server locale viene
  * assegnato ad un nuovo Thread che si occupera' di rispondere alle richieste del
      client.
  */
void *servent_listen(void *parm);

/**
  * Server thread che riceve i pacchetti e risponde adeguatamente. Ne esiste uno per
      ogni
  * peer a cui si e' connessi. Questa funzione e' il vero cuore di TorTella,
  * infatti gestisce tutti i comportamente dei programma in base ai pacchetti ricevuti.
  *
  * \param parm Socket descriptor della connessione
  */
void *servent_respond(void *parm);

/**
  * Client thread utilizzato per gestire il canale di invio pacchetti ad un peer.
  */
void *servent_connect(void *parm);

/**
  * Thread utilizzato per gestire il meccanismo di failure detection e per pulire
  * la lista dei pacchetti ricevuti. L'intervallo di tempo e' impostato nel file
  * di configurazione.
  */
void *servent_timer(void *parm);

#endif //!SERVENT_H
```

1.17 servent.c

```
#include "servent.h"

/**
 * Crea un server socket
 */
int servent_create_server(char *src_ip, u_int4 src_port) {
 logger(SOCK_INFO, "[servent_create_server]Creating listen tcp socket on: %s:%d\n",
      src_ip, src_port);
 return create_listen_tcp_socket(src_ip, src_port);
}

/**
  * Crea un client socket
  */
int servent_create_client(char *dst_ip, u_int4 dst_port) {
 logger(SOCK_INFO, "[servent_create_client]Creating tcp socket on: %s:%d\n", dst_ip,
      dst_port);
 return create_tcp_socket(dst_ip, dst_port);
}

/**
  * Avvia il server thread che viene usato per accettare
  * le nuove connessioni e quindi lanciare nuovi thread
  * per gestirle
  */
int servent_start_server(char *local_ip, u_int4 local_port) {
 int serfd = servent_create_server(local_ip, local_port); //!Avvia il server di ascolto
 if(serfd<0) {
  logger(ALARM_INFO, "[servent_start]Errore nella creazione del server\n");
  return -1;
 }
 pthread_t *serthread = (pthread_t*)malloc(sizeof(pthread_t));
 pthread_create(serthread, NULL, servent_listen, (void*)serfd);

 //!Aggiunta socket descriptor alle liste
 server_fd = g_list_prepend(server_fd, (gpointer)serfd);
 server_thread = g_list_prepend(server_thread, (gpointer)serthread);

 return 0;
}

/**
  * Avvia un client thread usato per gestire tutte le richieste di invio
  * pacchetti al peer associato.
  *
  * \param id Se si conosce l'id del peer a cui connettersi lo si specifica, altrimenti
      0.
  */
servent_data *servent_start_client(char *dest_ip, u_int4 dest_port, u_int8 id) {
 pthread_t *clithread = (pthread_t*)calloc(1, sizeof(pthread_t));
 u_int8 cliid = new_connection_counter++; //!Incrementa il contatore degli ID falsi
 if(id >= conf_get_gen_start()) //!Verifica che l'id inserito non sia falso (condizione
      di sicurezza)
  cliid = id;

 //!Inizializzazione dati del peer all'interno della servent_data
 servent_data *servent = (servent_data*)calloc(1, sizeof(servent_data));
 servent->id = cliid;
 logger(SYS_INFO, "[servent_start_client]cliid: %lld\n", cliid);
 servent->ip = dest_ip;
 servent->port = dest_port;
```

```
 servent->chat_list = NULL;
 servent->queue = g_queue_new();
 servent->res_queue = g_queue_new();
 servent->is_online = 0;
 pthread_rwlock_init(&servent->rwlock_data, NULL);

 //!Aggiunge il peer all'elenco degli utenti conosciuti (ma non necessariamente connessi
     )
 data_add_user(servent->id, servent->nick, servent->ip, servent->port);

 //!Aggiunge la servent_data generata alla hashtable dei serventi
 g_hash_table_insert(servent_hashtable, (gpointer)to_string(cliid), (gpointer)servent);
 //!Lancia il client thread associato al peer
 pthread_create(clithread, NULL, servent_connect, (void*)&cliid);
 client_thread = g_list_prepend(client_thread, (gpointer)(*clithread));

 //!Attende l'avvenuta ricezione del messaggio di OK (o timeout)
 char *ret = servent_pop_response(servent);
 if(ret!=NULL && strcmp(ret, TIMEOUT)==0) {
  servent->post_type = CLOSE_ID;
  servent_send_packet(servent);
  return NULL;
 }

 return servent;
}

/**
  * Questa funzione viene utilizzata per il boot iniziale, in quanto si connette
  * alla lista dei peer conosciuti, presi dal file di init. Inoltre inizializza
  * tutte le variabili necessarie e avvia il server di ascolto nuove connessioni.
  *
  * \param init_servent Lista dei serventi necessari per il boot iniziale
  */
int servent_start(GList *init_servent) {
 //!Inizializzazione servent locale
 if(servent_init(conf_get_local_ip(), conf_get_local_port(), ONLINE_ID)<0) {
  logger(SYS_INFO, "[servent_start]Error initializing data\n");
  return -1;
 }

 //!Avvio server di ascolto richieste
 if(servent_start_server(conf_get_local_ip(), conf_get_local_port())<0) {
  logger(SYS_INFO, "[servent_start]Error starting server\n");
  return -2;
 }

 //!Fase iniziale di reperimento degli utenti iniziali
 if(init_servent!=NULL) {
  if(servent_init_connection(init_servent)<0) {
   logger(SYS_INFO, "[servent_start]Error boot connections\n");
   return -3;
  }

 }
 return 0;
}

/**
  * Tale funzione non fa altro che avviare il timer thread utilizzato per supportare
  * il meccanismo di failure detection.
  */
int servent_start_timer(void) {
 timer_thread = (pthread_t*)calloc(1, sizeof(pthread_t));
```

```
 pthread_create(timer_thread, NULL, servent_timer, NULL);
 return 0;
}

/**
  * Si connette alla lista dei peer specificati, se qualcuno non e' disponibile lo salta
      .
  */
int servent_init_connection(GList *init_servent) {
 int i;
 init_data *peer;
 for(i=0; i<g_list_length(init_servent); i++) {
  peer = (init_data*)g_list_nth_data(init_servent, i);
  logger(SOCK_INFO, "[servent_init_connection]Connecting to ip: %s, port: %d\n", peer->
      ip, peer->port);
  if(servent_start_client(peer->ip, peer->port, 0)==NULL) {
   logger(SYS_INFO, "[servent_init_connection]Unable to connect to known peer\n");
  }
 }
 return 0;
}

/**
  * Questa funzione viene chiamata alla chiusura di TorTella, serve per chiudere
  * tutti i socket aperti (non brutalmente).
  */
void servent_close_all(void) {
 int i;
 for(i=0; i<g_list_length(server_fd); i++) {
  logger(SYS_INFO, "[servent_close_all]Closing server %d\n", (int)g_list_nth_data(
      server_fd, i));
  delete_socket((int)g_list_nth_data(server_fd, i));
 }
 for(i=0; i<g_list_length(client_fd); i++) {
  logger(SYS_INFO, "[servent_close_all]Closing client %d\n", (int)g_list_nth_data(
      client_fd, i));
  delete_socket((int)g_list_nth_data(client_fd, i));
 }
 for(i=0; i<g_list_length(server_connection_fd); i++) {
  logger(SYS_INFO, "[servent_close_all]Closing server_connection %d\n", (int)
      g_list_nth_data(server_connection_fd, i));
  delete_socket((int)g_list_nth_data(server_connection_fd, i));
 }

}

/**
  * Ultima funzione chiamata alla chiusura del programma, termina tutti i thread attivi.
  */
void servent_kill_all_thread(int sig) {
 logger(SYS_INFO, "[servent_kill_all_thread]Killing threads\n");
 servent_close_supernode(); //!Viene utilizzata per il salvataggio delle chat su file (
     non utilizzata)
 logger(SYS_INFO, "[servent_kill_all_thread]Closing supernode\n");
 servent_close_all();

 int i;
 for(i=0; i<g_list_length(server_thread); i++) {
  pthread_kill((pthread_t)g_list_nth_data(server_thread, i), SIGKILL);
 }
 for(i=0; i<g_list_length(client_thread); i++) {
  pthread_kill((pthread_t)g_list_nth_data(client_thread, i), SIGKILL);
 }
 for(i=0; i<g_list_length(server_connection_thread); i++) {
```

```
  pthread_kill((pthread_t)g_list_nth_data(server_connection_thread, i), SIGKILL);
 }

 if(timer_thread!=NULL)
  pthread_kill(*timer_thread, SIGKILL);
}

/**
  * Inizializza: le variabili del peer locale, il seed, le hashtable, i lock etc...
  */
int servent_init(char *ip, u_int4 port, u_int1 status) {

 //!Inizializza la lista delle chat conosciute leggendo da un file predefinito (non
      utilizzato)
 servent_init_supernode();
 //!Inizializza il seed
 srandom(time(NULL));
 //!Recupera il numero iniziale da cui generare fake ID
 new_connection_counter = conf_get_connection_id_limit();
 logger(SYS_INFO, "[servent_init]Local Peer initialized on %s:%d\n", conf_get_local_ip()
      , conf_get_local_port());

 servent_hashtable = g_hash_table_new_full(g_str_hash, g_str_equal, free, NULL);

 //!----Routing-----
 search_packet_hashtable = g_hash_table_new(g_str_hash, g_str_equal);
 join_packet_hashtable = g_hash_table_new(g_str_hash, g_str_equal);
 leave_packet_hashtable = g_hash_table_new(g_str_hash, g_str_equal);
 //!----------------

 local_servent = (servent_data*)calloc(1, sizeof(servent_data));
 u_int8 id = local_servent->id = generate_id();
 logger(SYS_INFO, "[servent_init]Local ID: %lld\n", local_servent->id);
 local_servent->ip = ip;
 local_servent->port = port;
 local_servent->queue = g_queue_new();
 local_servent->res_queue = g_queue_new();
 local_servent->status = status;
 local_servent->nick = conf_get_nick();
 local_servent->is_online = 1;
 //!Aggiunta utente locale alle liste di utenti conosciuti
 data_add_user(local_servent->id, local_servent->nick, local_servent->ip, local_servent
      ->port);

 pthread_rwlock_init(&local_servent->rwlock_data, NULL);
 g_hash_table_insert(servent_hashtable, (gpointer)to_string(id), (gpointer)local_servent
      );

 server_fd = NULL;
 client_fd = NULL;
 server_connection_fd = NULL;

 server_thread = NULL;
 client_thread = NULL;
 server_connection_thread = NULL;

 timer_thread = NULL;

 return 0;
}

/**
  * Funzione utilizzata per il recupero delle chat conosciute da file,
  * attualmente non piu' utilizzata.
```

```
  */
void servent_init_supernode() {
 //!read_all();
}

/**
  * Funzione utilizzata per il salvataggio delle chat conosciute su file,
  * attualmente non piu' utilizzata.
  */
void servent_close_supernode() {
 //!write_all(MODE_TRUNC);
}

//!-----Gestione servent_data-----

/**
  * Restituisce il servent_data associato all'id richiesto.
  */
servent_data *servent_get(u_int8 id) {
 return (servent_data*)g_hash_table_lookup(servent_hashtable, (gconstpointer)to_string(
      id));
}

/**
  * Restituisce la lista completa delle servent_data.
  */
GList *servent_get_values(void) {
 return g_hash_table_get_values(servent_hashtable);
}

/**
  * Restituisce la lista completa delle chiavi associate alle servent_data.
  */
GList *servent_get_keys(void) {
 return g_hash_table_get_keys(servent_hashtable);
}

/**
  * Restituisce la servent_data del peer locale.
  */
servent_data *servent_get_local(void) {
 return local_servent;
}

//!-----Gestione Queue-----

/**
  * Aggiunge alla coda di pacchetti da inviare ad uno specifico peer.
  * In particolare si passa una servent_data contenete tutti i dati
  * necessari all'invio del pacchetto.
  */
void servent_send_packet(servent_data *sd) {
 if(sd != NULL)
  g_queue_push_tail(sd->queue, (gpointer)sd);
}

/**
  * Rimuove dalla coda di pacchetti da inviare ad uno specifico peer.
  * Se non ci sono pacchetti da rimuovere rimane in attesa.
  */
servent_data *servent_pop_queue(servent_data *sd) {
 servent_data *servent;
 if(sd==NULL || sd->queue==NULL) {
  logger(ALARM_INFO, "[servent_pop_queue]Queue error\n");
```

```
  return NULL;
 }
 //!Ciclo utilizzato per attendere la richiesta di invio pacchetti
 while((servent = (servent_data*)g_queue_pop_head(sd->queue))==NULL) {
  usleep(100000);        //!Attende prima di controllare di nuovo la coda
 }
 return servent;
}

/**
  * Appende alla coda di risposta di uno specifico peer.
  * In particolare aggiunge la risposta ricevuta dopo l'invio di un pacchetto.
  */
void servent_append_response(servent_data *sd, const char *response) {
 if(sd != NULL)
  g_queue_push_tail(sd->res_queue,(gpointer)strdup(response));
}

/**
  * Rimuove dalla coda di risposta un elemento se presente,
  * nel frattempo avvia un timer per rilevare eventuali timeout di risposta.
  */
char *servent_pop_response(servent_data *sd) {
 char *buf;
 if(sd==NULL || sd->res_queue==NULL) {
  logger(SYS_INFO, "[servent_append_responde]Response queue error\n");
  return NULL;
 }

 int counter = 0; //!Contatore utilizzato per dare un timeout al superamento di una
      soglia
 while((buf = (char*)g_queue_pop_head(sd->res_queue))==NULL) {
  if(counter>10) { //!Serve per il timeout
   buf = TIMEOUT;
   break;
  }
  usleep(200000);
  counter++;
 }
 return buf;
}

//!-----Routing-----

/**
  * Ritorna l'ID del pacchetto richiesto, se presente.
  * E' una sorta di verifica presenta pacchetto.
  * Utilizzato per gestire i duplicati.
  */
char *servent_get_search_packet(u_int8 id) {
 return (char*)g_hash_table_lookup(search_packet_hashtable,(gconstpointer)to_string(id))
      ;
}

/**
  * Aggiunge il pacchetto alla lista dei pacchetti ricevuti.
  */
void servent_new_search_packet(u_int8 id) {
 g_hash_table_insert(search_packet_hashtable,(gpointer)to_string(id),(gpointer)to_string
      (id));
}

/**
  * Ritorna l'ID del pacchetto richiesto, se presente.
```

```
  * E' una sorta di verifica presenta pacchetto.
  * Utilizzato per gestire i duplicati.
  */
char *servent_get_join_packet(u_int8 id) {
 return (char*)g_hash_table_lookup(join_packet_hashtable, (gconstpointer)to_string(id));
}

/**
  * Aggiunge il pacchetto alla lista dei pacchetti ricevuti.
  */
void servent_new_join_packet(u_int8 id) {
 g_hash_table_insert(join_packet_hashtable, (gpointer)to_string(id),(gpointer)to_string(
     id));
}

/**
  * Ritorna l'ID del pacchetto richiesto, se presente.
  * E' una sorta di verifica presenta pacchetto.
  * Utilizzato per gestire i duplicati.
  */
char *servent_get_leave_packet(u_int8 id) {
 return (char*)g_hash_table_lookup(leave_packet_hashtable, (gconstpointer)to_string(id))
     ;
}

/**
  * Aggiunge il pacchetto alla lista dei pacchetti ricevuti.
  */
void servent_new_leave_packet(u_int8 id) {
 g_hash_table_insert(leave_packet_hashtable, (gpointer)to_string(id),(gpointer)to_string
     (id));
}

/**
  * Pulisce la lista dei pacchetti ricevuti che serve per scartare
  * pacchetti con ID uguale a quelli ricevuti recentemente.
  */
void servent_flush_data(void) {
 g_hash_table_remove_all(search_packet_hashtable);
 g_hash_table_remove_all(join_packet_hashtable);
 g_hash_table_remove_all(leave_packet_hashtable);
}


//!---------THREAD----------------

/**
  * Thread che riceve le richieste di connessione e avvia nuovi thread.
  * Ogni nuovo peer (client) che richiede di connettersi al server locale viene
  * assegnato ad un nuovo Thread che si occupera' di rispondere alle richieste del
      client.
  */
void *servent_listen(void *parm) {
 int connFd;
 pthread_t *thread;

 while(1) {
  connFd = listen_http_packet((int)parm);
  logger(SYS_INFO, "[servent_listen]Received connection on socket: %d\n", connFd);
  if(connFd!=0) {
   thread = (pthread_t*)calloc(1, sizeof(pthread_t));
   //!Aggiunge alla lista dei socket descriptor di connessione
   server_connection_fd = g_list_prepend(server_connection_fd, (gpointer)connFd);
   //!Avvia il server thread
```

```
    pthread_create(thread, NULL, servent_respond, (void*)connFd);
    //!Aggiunge alla lista degli identificatori dei thread di connessione
    server_connection_thread = g_list_prepend(server_connection_thread, (gpointer)(*
        thread));
   }
  }
  pthread_exit(NULL);
}

/**
  * Server thread che riceve i pacchetti e risponde adeguatamente. Ne esiste uno per
      ogni
  * peer a cui si e' connessi. Questa funzione e' il vero cuore di TorTella,
  * infatti gestisce tutti i comportamente dei programma in base ai pacchetti ricevuti.
  *
  * \param parm Socket descriptor della connessione
  */
void *servent_respond(void *parm) {
 logger(SYS_INFO, "[servent_respond]Server initialized\n");
 char *buffer;
 http_packet *h_packet;
 int len;
 int fd = (int)parm;
 u_int8 user_id = 0;
 u_int4 status = 0;

 while(1) {
  logger(SYS_INFO, "[servent_respond]Waiting\n");
  //!Attesa ricezione pacchetto HTTP
  len = recv_http_packet(fd, &buffer);
  logger(PAC_INFO, "[servent_respond]Data received len: %d, buffer: \nSTART\n%s\nEND\n",
       len, dump_data(buffer, len));

  if(len>0) {
   logger(SYS_INFO, "[servent_respond]Converting\n");
   //!Riempimento della struttura dati http_packet con i valori ricevuti
   h_packet = http_char_to_bin((const char*)buffer);
   if(h_packet!=NULL) {
        logger(SYS_INFO, "[servent_respond]Http packet received, type=%d\n", h_packet->
            type);
        if(h_packet->type==HTTP_REQ_POST) {

         logger(SYS_INFO, "[servent_respond]POST received\n");

         //!Effettua le operazioni adeguatamente al tipo di pacchetto ricevuto
         if(h_packet->data==NULL || h_packet->data->header==NULL) {
          logger(SYS_INFO, "[servent_respond]Invalid HTTP packet\n");

          //!Imposta lo status del pacchetto di risposta da inviare
          status = HTTP_STATUS_CERROR;
         }
         else if(h_packet->data->header->recv_id!=local_servent->id && h_packet->data->
             header->recv_id>=conf_get_gen_start()) {
          /**
                * Entra in questa condizione se l'ID di ricezione del pacchetto e'
                    diverso dal locale,
                * ovvero il pacchetto non e' destinato al peer che l'ha ricevuto.
                    Inoltre controlla che
                * l'ID non sia falso.
                */
          status = HTTP_STATUS_CERROR;
         }
         else if(h_packet->data->header->desc_id==JOIN_ID) {
          logger(SYS_INFO, "[servent_respond]JOIN received, packet_id: %lld\n", h_packet
```

```
    ->data->header->id);

//!Invio di un pacchetto di notifica di avvenuta ricezione del JOIN
status = HTTP_STATUS_OK;
send_post_response_packet(fd, status, 0, NULL);
logger(SYS_INFO, "[servent_respond]Sending post response\n");
//!Si imposta lo status a 0 per evitare di inviare un doppio pacchetto di
    notifica
status = 0;

//!Verifica che il pacchetto ricevuto non sia un duplicato
if(servent_get_join_packet(h_packet->data->header->id) == NULL) {
 //!Aggiunge il pacchetto ricevuto all'hashtable associata al JOIN
 servent_new_join_packet(h_packet->data->header->id);

 GList *servent_list;
 servent_data *conn_servent = (servent_data*)g_hash_table_lookup(
     servent_hashtable, (gconstpointer)to_string(h_packet->data->header->
     sender_id));
 if(conn_servent==NULL) {
     logger(SYS_INFO, "[servent_responde]conn_servent entry %lld doesn't
         found\n", h_packet->data->header->sender_id);
     continue;
 }
 //!Aggiunta dell'utente che ha inviato il JOIN nelle liste contenenti gli
     utenti
 data_add_user(GET_JOIN(h_packet->data)->user_id, tortella_get_data(h_packet->
     data_string), GET_JOIN(h_packet->data)->ip, GET_JOIN(h_packet->data)->
     port);
 data_add_existing_user_to_chat(GET_JOIN(h_packet->data)->chat_id, GET_JOIN(
     h_packet->data)->user_id);
 controller_add_user_to_chat(GET_JOIN(h_packet->data)->chat_id, GET_JOIN(
     h_packet->data)->user_id);

 logger(SYS_INFO, "[servent_respond]Sending JOIN packet to others peer\n");

 //!Verifica che il ttl sia maggiore di uno per il rinvio del pacchetto agli
     altri peer
 if(GET_JOIN(h_packet->data)->ttl>1) {
     logger(SYS_INFO, "[servent_respond]TTL > 1\n");
     int i;
     servent_list = g_hash_table_get_values(servent_hashtable);
     //!Recupero di tutti i peer a cui si e' connessi per rinviare il
         pacchetto (flooding)
     for(i=0; i<g_list_length(servent_list); i++) {

      conn_servent = (servent_data*)g_list_nth_data(servent_list, i);
      /**
        * Evita di rinviare il pacchetto al peer da cui ha ricevuto il JOIN.
        * Evita di inviare il pacchetto due volte ad uno stesso peer.
        */
      if(conn_servent->id!=h_packet->data->header->sender_id && conn_servent
          ->id!=servent_get_local()->id && conn_servent->id>=
          conf_get_gen_start()) {
       RLOCK(conn_servent->id);
       servent_data *sd;
       COPY_SERVENT(conn_servent, sd);
       sd->ttl = GET_JOIN(h_packet->data)->ttl-1;
       sd->hops = GET_JOIN(h_packet->data)->hops+1;
       sd->packet_id = h_packet->data->header->id;
       sd->user_id_req = GET_JOIN(h_packet->data)->user_id;
       sd->chat_id_req = GET_JOIN(h_packet->data)->chat_id;
       sd->status_req = GET_JOIN(h_packet->data)->status;
       sd->ip_req = GET_JOIN(h_packet->data)->ip;
```

```
            sd->port_req = GET_JOIN(h_packet->data)->port;
            sd->nick_req = tortella_get_data(h_packet->data_string);
            sd->post_type = JOIN_ID;
            UNLOCK(conn_servent->id);
            //!Invio del pacchetto al peer selezionato
            servent_send_packet(sd);
            //!Attesa ricezione risposta
            servent_pop_response(sd);
            logger(SYS_INFO, "[servent_respond]Retrasmitted JOIN packet %s to %s\n
                ", sd->nick_req, sd->nick);
          }
         }
  }
 }
}
else if(h_packet->data->header->desc_id==PING_ID) {
 logger(SYS_INFO, "[servent_respond]PING received from %lld to %lld\n",
      h_packet->data->header->sender_id, h_packet->data->header->recv_id);

 status = HTTP_STATUS_OK;
 //!invio del pacchetto di OK
 send_post_response_packet(fd, status, 0, NULL);
 logger(SYS_INFO, "[servent_respond]Sending post response\n");
 status = 0;

 user_id = h_packet->data->header->sender_id;
 servent_data *conn_servent;
 u_int8 id = h_packet->data->header->sender_id;
 logger(SYS_INFO, "[servent_responde]Searching in hashtable: %lld\n", id);
 /**
       * Controlla che il peer che ha inviato il pacchetto sia conosciuto
       * e quindi viene interpretato come un semplice ping inviato per
       * gestire il meccanismo di failure detection o il cambio di status.
       */
 if((conn_servent=g_hash_table_lookup(servent_hashtable, (gconstpointer)
      to_string(id)))!=NULL) {
  logger(SYS_INFO, "[servent_respond]Found: %lld\n", id);
  WLOCK(id);
  conn_servent->status = GET_PING(h_packet->data)->status;
  conn_servent->timestamp = h_packet->data->header->timestamp;
  conn_servent->nick = h_packet->data->data;
  conn_servent->is_online = 1;
  UNLOCK(id);
  data_add_user(conn_servent->id, conn_servent->nick, conn_servent->ip,
       conn_servent->port);
  logger(SYS_INFO, "[servent_respond]Old PING, nick: %s, status: %c\n",
       conn_servent->nick, conn_servent->status);
  /**
        * notifica del cambio di status sulla gui. Presi i lock sulla gui
        * per consentire l'accesso protetto ai dati della gui.
        */
  gdk_threads_enter();
  controller_manipulating_status(id, conn_servent->status);
  gdk_threads_leave();

 }
 else {
  /**
        * Entra in questo flusso quando il peer mittente non
        * e' ancora conosciuto dal peer locale. Serve per
        * stabilire una nuova connesione tra i due peer.
        */
  logger(SYS_INFO, "[servent_respond]New PING\n");
  /**
```

```
        * controllo che l'id del mittente sia falso in modo da capire
        * che e' la richiesta di una nuova connessione
        */
    if(h_packet->data->header->recv_id < conf_get_gen_start()) {

        conn_servent = (servent_data*)calloc(1, sizeof(servent_data));
        conn_servent->ip = get_dest_ip(fd);
        conn_servent->port = GET_PING(h_packet->data)->port;
        conn_servent->timestamp = h_packet->data->header->timestamp;
        conn_servent->queue = g_queue_new();
        conn_servent->res_queue = g_queue_new();
        conn_servent->is_online = 1;

        conn_servent->status = GET_PING(h_packet->data)->status;
        logger(SYS_INFO, "[servent_respond]Status recv: %c\n", conn_servent->
            status);
        conn_servent->nick = h_packet->data->data;
        conn_servent->id = h_packet->data->header->sender_id;

        //!aggiunta dell'utente alla lista dei peer conosciuti.
        data_add_user(conn_servent->id, conn_servent->nick, conn_servent->ip,
            conn_servent->port);

        //!Si inizializza il mutex
        pthread_rwlock_init(&conn_servent->rwlock_data, NULL);

        logger(SYS_INFO, "[servent_respond]Lookup ID: %s\n", to_string(id));
        if(g_hash_table_lookup(servent_hashtable, (gconstpointer)to_string(id))
            ==NULL) {
         logger(SYS_INFO, "[servent_respond]connection %s added to hashtable\n",
              to_string(id));
         g_hash_table_insert(servent_hashtable, (gpointer)to_string(id), (
             gpointer)conn_servent);
        }

        //!creazione nuovo client thread per gestire la connessione con il nuovo
             peer.
        pthread_t *cli_thread = (pthread_t*)malloc(sizeof(pthread_t));
        pthread_create(cli_thread, NULL, servent_connect, (void*)&id);
        client_thread = g_list_prepend(client_thread, (gpointer)(*cli_thread));

        //!attesa risposta di OK (o TIMEOUT).
        servent_pop_response(conn_servent);
    }
    else {
        /**
          * Rappresenta la seconda fase della connessione ad un peer.
          * Esempio: il peer locale invia un ping con id falso ad un
          * peer con cui vuole stabilire la connessione; il peer
          * remoto invia un ping con il vero id. Connessione stabilita.
          */
        GList *users = servent_get_values ();
        int i=0;
        logger(SYS_INFO, "[servent_respond]Changing ID\n");
        for(;i<g_list_length(users);i++) {
         servent_data *tmp = (servent_data*)g_list_nth_data(users, i);

         char *nick = tmp->nick;
         char *tmp_ip = tmp->ip;
         u_int4 tmp_port = tmp->port;

         logger(SYS_INFO, "[servent_respond]old ID: %lld, new ID: %lld\n", tmp->
             id, h_packet->data->header->sender_id);
         logger(SYS_INFO, "[servent_respond]nick: %s, ip: %s, port: %d\n", nick,
```

```
                get_dest_ip(fd), GET_PING(h_packet->data)->port);
            /**
              * recupera la servent_data associata al precedente
              * fake id e sostituisce il falso id con quello reale.
              */
            if((strcmp(tmp_ip, get_dest_ip(fd))==0) && (tmp_port == GET_PING(
                h_packet->data)->port)) {
             logger(SYS_INFO, "[servent_respond]Changing old ID: %lld with new ID:
                 %lld\n", tmp->id, h_packet->data->header->sender_id);
             //!rimuove dalla hashtable la chiave con id fasullo
             g_hash_table_remove(servent_hashtable, (gpointer)to_string(tmp->id));

             tmp->id = h_packet->data->header->sender_id;
             tmp->status = GET_PING(h_packet->data)->status;
             tmp->timestamp = h_packet->data->header->timestamp;
             tmp->nick = h_packet->data->data;
             tmp->is_online = 1;
             //!aggiunge l'utente alla lista dei peer conosciuti
             data_add_user(tmp->id, tmp->nick, tmp->ip, tmp->port);

             g_hash_table_insert(servent_hashtable, (gpointer)to_string(tmp->id),(
                 gpointer)tmp);
            }
           }
  }


 }
}
else if(h_packet->data->header->desc_id==LEAVE_ID) {
 /**
       * Ricezione di un pacchetto di tipo LEAVE.
       */
 logger(SYS_INFO, "[servent_respond]LEAVE received, packet_id: %lld\n",
      h_packet->data->header->id);

 status = HTTP_STATUS_OK;
 //!invio di un pacchetto di OK che conferma l'avvenuta ricezione del LEAVE
 send_post_response_packet(fd, status, 0, NULL);
 logger(SYS_INFO, "[servent_responde]Sending post response\n");
 status = 0;

 //!controllo dei pacchetti LEAVE duplicati
 if(servent_get_leave_packet(h_packet->data->header->id) == NULL) {
  //!aggiunge il pacchetto alla lista dei pacchetti LEAVE
  servent_new_leave_packet(h_packet->data->header->id);

  GList *servent_list;
  servent_data *conn_servent = (servent_data*)g_hash_table_lookup(
       servent_hashtable, (gconstpointer)to_string(h_packet->data->header->
       sender_id));
  if(conn_servent==NULL) {
       logger(SYS_INFO, "[servent_responde]conn_servent entry %lld doesn't
            found\n", h_packet->data->header->sender_id);
       continue;
  }

  u_int8 chat_id = GET_LEAVE(h_packet->data)->chat_id;

  //!Sconnetti dalla chat l'utente
  logger(SYS_INFO, "[servent_respond]Deleting user\n");
  gdk_threads_enter();
  controller_rem_user_from_chat(chat_id, GET_LEAVE(h_packet->data)->user_id);
  gdk_threads_leave();
```

```
  logger(SYS_INFO, "[servent_respond]Deleted user: %lld\n", GET_LEAVE(h_packet
      ->data)->user_id);


  logger(SYS_INFO, "[servent_respond]Sending LEAVE packet to others peer\n");

  //!controllo che il TTL sia maggiore di uno in modo da reinviare il pacchetto
  if(GET_LEAVE(h_packet->data)->ttl>1) {
        logger(SYS_INFO, "[servent_respond]TTL > 1\n");
        int i;
        servent_list = g_hash_table_get_values(servent_hashtable);
        if(servent_list!=NULL) {
         for(i=0; i<g_list_length(servent_list); i++) {

          conn_servent = (servent_data*)g_list_nth_data(servent_list, i);
          /**
                * Evita di rinviare il pacchetto al peer da cui ha ricevuto il
                    LEAVE.
                * Evita di inviare il pacchetto due volte ad uno stesso peer.
                */
          if(conn_servent->id!=h_packet->data->header->sender_id && conn_servent
              ->id!=servent_get_local()->id && conn_servent->id>=
              conf_get_gen_start()) {
           RLOCK(conn_servent->id);
           servent_data *sd;
           COPY_SERVENT(conn_servent, sd);
           sd->ttl = GET_LEAVE(h_packet->data)->ttl-1;
           sd->hops = GET_LEAVE(h_packet->data)->hops+1;
           sd->packet_id = h_packet->data->header->id;
           sd->user_id_req = GET_LEAVE(h_packet->data)->user_id;
           sd->chat_id_req = GET_LEAVE(h_packet->data)->chat_id;
           sd->post_type = LEAVE_ID;
           UNLOCK(conn_servent->id);
           //!invio del pacchetto al peer selezionato
           servent_send_packet(sd);
           //!attesa della ricezione del messaggio di OK (o di TIMEOUT)
           servent_pop_response(sd);
           logger(SYS_INFO, "[servent_respond]Retrasmitted LEAVE packet to other
                peers\n");
          }
         }
        }
  }
 }
}
else if(h_packet->data->header->desc_id==MESSAGE_ID) {
 /**
       * Ricezione di un pacchetto di tipo MESSAGE.
       */
 logger(SYS_INFO, "[servent_respond]MESSAGE ricevuto\n");

 servent_data *conn_servent = (servent_data*)g_hash_table_lookup(
      servent_hashtable, (gconstpointer)to_string(h_packet->data->header->
      sender_id));

 WLOCK(h_packet->data->header->sender_id);
 conn_servent->msg = h_packet->data->data;
 conn_servent->msg_len = h_packet->data->header->data_len;
 conn_servent->timestamp = h_packet->data->header->timestamp;
 //!prepara il messaggio in modo da aggiornare la GUI
 char *send_msg = prepare_msg(conn_servent->timestamp, conn_servent->nick,
      conn_servent->msg, conn_servent->msg_len);
 UNLOCK(h_packet->data->header->sender_id);
 u_int8 chat_id = GET_MESSAGE(h_packet->data)->chat_id;
```

```
 if(chat_id==0) {
  logger(SYS_INFO, "[servent_respond]PM\n");
  //!aggiornamento della gui relativa ad un messaggio privato
  gdk_threads_enter();
  controller_add_msg(h_packet->data->header->sender_id, send_msg);
  gdk_threads_leave();
 }
 else {
  logger(SYS_INFO, "[servent_respond]CHAT\n");
  //!aggiornamento della gui relativa alla chat
  gdk_threads_enter();
  controller_add_msg_to_chat(chat_id, send_msg);
  gdk_threads_leave();
 }
 logger(SYS_INFO, "[servent_respond]msg: %s, msg_len: %d\n", conn_servent->msg,
       conn_servent->msg_len);


 status = HTTP_STATUS_OK;
}
else if(h_packet->data->header->desc_id==SEARCH_ID) {
 /**
       * Ricezione di un messaggio di tipo SEARCH.
       */
 logger(SYS_INFO, "[servent_respond]SEARCH ricevuto packet_id: %lld\n",
      h_packet->data->header->id);

 status = HTTP_STATUS_OK;
 //!invio di un mpacchetto di avvenuta ricezione della SEARCH
 send_post_response_packet(fd, status, 0, NULL);
 logger(SYS_INFO, "[servent_respond]Sending post response\n");
 status = 0;
 //!controllo dei pacchetti SEARCH duplicati
 if(servent_get_search_packet(h_packet->data->header->id) == NULL) {
  //!aggiunge il pacchetto alla lista dei pacchetti LEAVE
  servent_new_search_packet(h_packet->data->header->id);

  GList *res;
  GList *servent_list;
  servent_data *conn_servent = (servent_data*)g_hash_table_lookup(
       servent_hashtable, (gconstpointer)to_string(h_packet->data->header->
       sender_id));
  if(conn_servent==NULL) {
       logger(SYS_INFO, "[servent_respond]conn_servent entry %lld doesn't found
            \n", h_packet->data->header->sender_id);
       continue;
  }
  logger(SYS_INFO, "[servent_respond]conn_servent entry found\n");
  //!controllo dell'integrita' del pacchetto
  if(h_packet->data_len>0) {
       servent_data *sd;
       RLOCK(conn_servent->id);
       COPY_SERVENT(conn_servent, sd);
       UNLOCK(conn_servent->id);

       logger(SYS_INFO, "[servent_respond]Searching %s\n", tortella_get_data(
            h_packet->data_string));
       //!Ricerca nelle chat conosciute la chat richiesta dalla SEARCH
       res = data_search_all_chat(tortella_get_data(h_packet->data_string));
       logger(SYS_INFO, "[servent_respond]Results number %d\n", g_list_length(
            res));

       logger(SYS_INFO, "[servent_respond]Sending to ID: %lld\n", sd->id);
       //!Passa i risultati alla servent_data del peer remoto
```

```
        sd->chat_res = res;
        sd->packet_id = h_packet->data->header->id;
        sd->post_type = SEARCHHITS_ID;
        //!invio del pacchetto SEARCHHITS
        servent_send_packet(sd);
        logger(SYS_INFO, "[servent_respond]Sending SEARCHHITS packet to
            searching peer\n");
    }
    //!controllo che il TTL sia maggiore di uno in modo da reinviare il pacchetto
    if(GET_SEARCH(h_packet->data)->ttl>1) {
        logger(SYS_INFO, "[servent_respond]TTL > 1\n");
        int i;
        servent_list = g_hash_table_get_values(servent_hashtable);
        for(i=0; i<g_list_length(servent_list); i++) {

         conn_servent = (servent_data*)g_list_nth_data(servent_list, i);
         /**
           * Evita di rinviare il pacchetto al peer da cui ha ricevuto il LEAVE.
           * Evita di inviare il pacchetto due volte ad uno stesso peer.
           */
         if(conn_servent->id!=h_packet->data->header->sender_id && conn_servent
             ->id!=servent_get_local()->id && conn_servent->id>=
             conf_get_gen_start()) {
          RLOCK(conn_servent->id);
          servent_data *sd;
          COPY_SERVENT(conn_servent, sd);
          sd->ttl = GET_SEARCH(h_packet->data)->ttl-1;
          sd->hops = GET_SEARCH(h_packet->data)->hops+1;
          sd->title = tortella_get_data(h_packet->data_string);
          sd->title_len = h_packet->data->header->data_len;
          sd->packet_id = h_packet->data->header->id;
          sd->post_type = SEARCH_ID;
          UNLOCK(conn_servent->id);
          //!reinvio del pacchetto SEARCH al peer selezionato
          servent_send_packet(sd);
          //!Aggiunta regola di routing alla tabella
          add_route_entry(h_packet->data->header->id, h_packet->data->header->
              sender_id, conn_servent->id);
          logger(SYS_INFO, "[servent_respond]Retrasmitted SEARCH packet to other
               peers\n");
         }
        }
    }
   }
  }
 else if(h_packet->data->header->desc_id==SEARCHHITS_ID) {
  /**
        * Ricezione di un pacchetto di tipo SEARCHHITS
        */
  logger(SYS_INFO, "[servent_respond]SEARCHHITS ricevuto\n");

  status = HTTP_STATUS_OK;
  //!invio di un pacchetto di avvenuta ricezione del SEARCHHITS
  send_post_response_packet(fd, status, 0, NULL);
  logger(SYS_INFO, "[servent_respond]Sending post response\n");
  status = 0;

  /**
        * Converte la stringa dei risultati ricevuti in una lista di chat
        * con i relativi utenti.
        */
  GList *chat_list = data_char_to_chatlist(tortella_get_data(h_packet->
      data_string), h_packet->data->header->data_len);
  //!Aggiunge le chat alle liste locali
```

```
 data_add_all_to_chat(chat_list);

 //!Ritorna la regola di routing associata all'ID del pacchetto
 route_entry *entry = get_route_entry(h_packet->data->header->id);
 if(entry!=NULL) {
  //!Entra se la regola esiste
  RLOCK(entry->sender_id);
  servent_data *sd;
  servent_data *conn_servent = (servent_data*)g_hash_table_lookup(
       servent_hashtable, (gconstpointer)to_string(entry->sender_id));

  COPY_SERVENT(conn_servent, sd);
  sd->packet_id = h_packet->data->header->id;
  sd->chat_res = chat_list;
  sd->post_type = SEARCHHITS_ID;
  UNLOCK(entry->sender_id);
  //!Invia il pacchetto SEARCHHITS al peer presente nella regola di routing
  servent_send_packet(sd);
  logger(SYS_INFO, "[servent_respond]Routing packet from %lld to %lld\n",
       h_packet->data->header->sender_id, entry->sender_id);
  //!Elimina la regola o decrementa
  del_route_entry(h_packet->data->header->id);
  logger(SYS_INFO, "[servent_respond]Route entry %lld deleted\n", h_packet->
       data->header->id);
 }
 else {
  //!Se la regola non esiste vuol dire che il peer e' colui che ha fatto la
       richiesta iniziale
  int i=0;
  chat *chat_val;

  for(; i<g_list_length(chat_list); i++) {
       chat_val = (chat*)g_list_nth_data(chat_list, i);
       logger(SYS_INFO,"[servent_respond] title chat %s\n", chat_val->title);
       GList *local_chat = data_search_all_local_chat(chat_val->title);
       int j=0;
       for(; j<g_list_length(local_chat); j++) {
        chat *tmp = (chat*)g_list_nth_data(local_chat, j);
        logger(SYS_INFO, "[servent_respond] title chat %s\n", tmp->title);
        gdk_threads_enter();
        //!Aggiunge la chat alla lista dei risultati nella GUI
        gui_add_chat(tmp->id, tmp->title);
        gdk_threads_leave();
       }
  }
 }

}
else if(h_packet->data->header->desc_id == BYE_ID) {
 logger(SYS_INFO, "[servent_respond]BYE ricevuto\n");

 servent_data *conn_servent = (servent_data*)g_hash_table_lookup(
      servent_hashtable, (gconstpointer)to_string(h_packet->data->header->
      sender_id));
 WLOCK(h_packet->data->header->sender_id);
 conn_servent->timestamp = h_packet->data->header->timestamp;
 UNLOCK(h_packet->data->header->sender_id);

 status = HTTP_STATUS_OK;
 //!Invia il pacchetto di risposta
 send_post_response_packet(fd, status, 0, NULL);
 logger(SYS_INFO, "[servent_respond]Sending post response\n");
 status = 0;
```

```
            servent_data *sd;
            COPY_SERVENT(conn_servent, sd);
            sd->post_type= CLOSE_ID;
            //!Chiusura thread connessione
            servent_send_packet(sd);

            logger(SYS_INFO, "[servent_respond]Deleting user\n");

            //!Chiusura eventuali finestre PM
            gdk_threads_enter();
            controller_receive_bye(conn_servent->id);
            gdk_threads_leave();

            //!Rimozione dalle strutture dati
            g_hash_table_remove(servent_hashtable, (gconstpointer)to_string(conn_servent->
                id));
            data_del_user(conn_servent->id);
            logger(SYS_INFO, "[servent_respond]Deleted user: %lld\n", conn_servent->id);

            //!Chiusura forzata socket
            shutdown_socket(fd);
            server_fd = g_list_remove(server_fd, (gconstpointer)fd);
            server_connection_thread = g_list_remove(server_connection_thread, (
                gconstpointer)pthread_self());
            //!Esce dal server thread
            pthread_exit(NULL);
           }

           //!Invio la conferma di ricezione
           if(status>0) {
            logger(SYS_INFO, "[servent_respond]Sending post response\n");
            send_post_response_packet(fd, status, 0, NULL);
           }

          }
          else if(h_packet->type==HTTP_REQ_GET) {
           logger(SYS_INFO, "[servent_respond]GET received\n");
          }
     }
    }
    else {
     logger(SYS_INFO, "[servent_responde]Peer %lld disconnected\n", user_id);
     if(servent_get(user_id)!=NULL) {
          RLOCK(user_id);
          servent_data* srv = servent_get(user_id);
          servent_data* sd;
          COPY_SERVENT(srv, sd);
          sd->post_type = CLOSE_ID;
          //!Chiude il client thread associato al peer
          servent_send_packet(sd);
          UNLOCK(user_id);

          //!Rimuove il peer dalla GUI e dalle strutture dati
          gdk_threads_enter();
          controller_receive_bye(user_id);
          gdk_threads_leave();
          data_destroy_user(user_id);
     }

     //!Chiusura forzata socket
     shutdown_socket(fd);
     server_connection_fd = g_list_remove(server_connection_fd, (gconstpointer)fd);
     server_connection_thread = g_list_remove(server_connection_thread, (gconstpointer)(
          pthread_self()));
```

```
    pthread_exit(NULL);
   }
  }
  pthread_exit(NULL);
}

/**
  * Client thread utilizzato per gestire il canale di invio pacchetti ad un peer.
  */
void *servent_connect(void *parm) {
 int len;
 char *buffer = NULL;
 http_packet *h_packet;
 u_int8 id_dest = *((u_int8*)(parm));
 logger(SYS_INFO, "[servent_connect]ID: %lld\n", id_dest);

 //!Si prendono l'ip e la porta dalla lista degli id
 servent_data *servent_peer = (servent_data*)g_hash_table_lookup(servent_hashtable, (
      gconstpointer)to_string(id_dest));
 servent_data *servent_queue;
 if(servent_peer == NULL) {
  logger(SYS_INFO, "[servent_connect]Error\n");
  pthread_exit(NULL);
 }
 char *ip_dest = servent_peer->ip;
 u_int4 port_dest = servent_peer->port;

 //!Creazione socket client
 int fd = servent_create_client(ip_dest, port_dest);

 client_fd = g_list_prepend(client_fd, (gpointer)fd);

 if(servent_peer->queue==NULL)
  servent_peer->queue = g_queue_new();
 if(servent_peer->res_queue==NULL)
  servent_peer->res_queue = g_queue_new();

 /**
   * Aggiunta richiesta di PING nella coda del suddetto servent
   * per iniziare la connessione verso il server.
   */
 servent_data *tmp;
 COPY_SERVENT(servent_peer, tmp);
 tmp->post_type = PING_ID;
 servent_send_packet(tmp);

 u_int4 post_type;
 u_int8 user_id_req;
 u_int8 chat_id_req;
 u_int4 msg_len;
 char *msg;
 u_int4 title_len;
 char *title;
 u_int1 ttl, hops;
 u_int8 packet_id;
 u_int4 port_req;
 char *ip_req;
 char *nick_req;

 u_int1 status;
 u_int1 status_req;
 char *nick;

 //!Ora si entra nel ciclo infinito che serve per inviare tutte le richieste
```

```
while(1) {
 logger(SYS_INFO, "[servent_connect]Waiting\n");
 if(servent_peer==NULL) {
  logger(SYS_INFO, "[servent_connect]Peer error\n");
  pthread_exit(NULL);
 }
 //!Attesa richiesta di invio pacchetto
 servent_queue = servent_pop_queue(servent_peer);
 /**
       * Questo passo e' fondamentale quando si effettua la connessione iniziale,
       * Quando viene inviato il nuovo ID tramite PING.
       */
 id_dest = servent_peer->id;
 if(servent_queue==NULL) {
  logger(SYS_INFO, "[servent_connect]Queue error\n");
  continue;
 }
 logger(SYS_INFO, "[servent_connect]Signal received in id_dest %lld\n", id_dest);

 RLOCK(local_servent->id);
 WLOCK(id_dest);
 logger(SYS_INFO, "[servent_connect]Post_type %d\n", servent_queue->post_type);

 //!Si salvano tutti i dati nella struttura dati condivisa
 servent_peer->post_type = servent_queue->post_type;
 servent_peer->chat_id_req = servent_queue->chat_id_req;
 servent_peer->msg_len = servent_queue->msg_len;
 servent_peer->msg = servent_queue->msg;
 servent_peer->title = servent_queue->title;
 servent_peer->title_len = servent_queue->title_len;
 servent_peer->ttl = servent_queue->ttl;
 servent_peer->hops = servent_queue->hops;
 servent_peer->packet_id = servent_queue->packet_id;
 servent_peer->status_req = servent_queue->status_req;
 servent_peer->user_id_req = servent_queue->user_id_req;
 servent_peer->ip_req = servent_queue->ip_req;
 servent_peer->port_req = servent_queue->port_req;
 servent_peer->nick_req = servent_queue->nick_req;

 post_type = servent_peer->post_type;
 user_id_req = servent_peer->user_id_req;
 chat_id_req = servent_peer->chat_id_req;
 msg_len = servent_peer->msg_len;
 msg = servent_peer->msg;
 title = servent_peer->title;
 title_len = servent_peer->title_len;
 ttl = servent_peer->ttl;
 hops = servent_peer->hops;
 packet_id = servent_peer->packet_id;
 ip_req = servent_peer->ip_req;
 port_req = servent_peer->port_req;
 nick_req = servent_peer->nick_req;

 status = local_servent->status;
 status_req = servent_peer->status_req;
 nick = local_servent->nick;

 //!Invio dei vari pacchetti
 if(post_type==HTTP_REQ_GET) {
  //!Invio di pacchetto GET (non utilizzata)
  //!send_get_request_packet(fd, char *filename, u_int4 range_start, u_int4 range_end);
 }
 else {
  if(post_type==JOIN_ID) {
```

```
        send_join_packet(fd, packet_id, local_servent->id, id_dest, status_req,
            user_id_req, chat_id_req, nick_req, port_req, ip_req, ttl, hops);
    }
    else if(post_type==PING_ID) {
        send_ping_packet(fd, local_servent->id, id_dest, nick, local_servent->port,
            status);
    }
    else if(post_type==BYE_ID) {
        logger(SYS_INFO, "[servent_connect]Sending bye packet\n");
        send_bye_packet(fd, local_servent->id, id_dest);
    }
    else if(post_type==CLOSE_ID) {
        //!Richiesta di invio CLOSE, ovvero terminazione thread corrente
        UNLOCK_F(servent_peer);
        UNLOCK(local_servent->id);
        shutdown_socket(fd);
        client_fd = g_list_remove(client_fd, (gconstpointer)fd);
        client_thread = g_list_remove(client_thread, (gconstpointer)pthread_self());
        g_hash_table_remove(servent_hashtable, (gconstpointer)to_string(id_dest));
        pthread_exit(NULL);
    }
    else if(post_type==LEAVE_ID) {
        send_leave_packet(fd, packet_id, local_servent->id, id_dest, user_id_req,
            chat_id_req, ttl, hops);
    }
    else if(post_type==MESSAGE_ID) {
        send_message_packet(fd, local_servent->id, id_dest, chat_id_req, msg_len, msg);
    }
    else if(post_type==SEARCH_ID) {
        send_search_packet(fd, packet_id, local_servent->id, id_dest, ttl, hops,
            title_len, title);
    }
    else if(post_type==SEARCHHITS_ID) {
        int length;
        //!Converte la lista delle chat in stringa, per inviare tramite pacchetto
        char *buf = data_chatlist_to_char(servent_queue->chat_res, &length);
        if(buf==NULL) {
         length=0;
        }
        else {
         logger(SYS_INFO, "[servent_connect]Results converted in buffer: %s\n", buf);
        }
        send_searchhits_packet(fd, packet_id, local_servent->id, id_dest, g_list_length(
            servent_queue->chat_res), length, buf);
    }

    //!Attesa ricezione risposta
    logger(SYS_INFO, "[servent_connect]Listening response\n");
    len = recv_http_packet(fd, &buffer);
    logger(SYS_INFO, "[sevente_connect]Received response\n");
    if(len>0) {
        logger(SYS_INFO, "[servent_connect]buffer recv: %s\n", dump_data(buffer, len));
        h_packet = http_char_to_bin(buffer);
        if(h_packet!=NULL && h_packet->type==HTTP_RES_POST) {
         if(strcmp(h_packet->header_response->response, HTTP_OK)==0) {
          logger(SYS_INFO, "[servent_connect]OK POST received\n");
         }
         else {
          logger(SYS_INFO, "[servent_connect]Error\n");
         }
        }
    }

    if(h_packet != NULL) {
```

```
        logger(SYS_INFO, "[servent_connect]Appending response\n");
        if(servent_peer==NULL) {
         logger(SYS_INFO, "[servent_connect] Peer response NULL\n");
         pthread_exit(NULL);
        }
        if(servent_peer->post_type != SEARCH_ID && servent_peer->post_type !=
            SEARCHHITS_ID && servent_peer->post_type != CLOSE_ID) {
         //!Appende alla coda delle risposte il tipo di risposta ricevuta
         servent_append_response(servent_peer, h_packet->header_response->response);
         logger(SYS_INFO, "[servent_connect]Appended\n");
        }
    }
    else {
        //!In caso di timeout appende alla coda delle risposte l'errore
        logger(SYS_INFO, "[servent_connect]Appending response TIMEOUT\n");
        servent_append_response(servent_peer, TIMEOUT);
    }
   }
   UNLOCK_F(servent_peer);
   UNLOCK(local_servent->id);
  }
  pthread_exit(NULL);
}

/**
  * Thread utilizzato per gestire il meccanismo di failure detection e per pulire
  * la lista dei pacchetti ricevuti. L'intervallo di tempo e' impostato nel file
  * di configurazione.
  */
void *servent_timer(void *parm) {
 GList *list;
 servent_data *data, *tmp;
 char *ret;
 int i;

 while(1) {

  list=g_hash_table_get_values(servent_hashtable);

  for(i=0; i<g_list_length(list); i++) {
   data = (servent_data*)g_list_nth_data(list, i);
   if(data!=NULL && (data->id!=local_servent->id) && (data->id>=conf_get_gen_start())) {

        RLOCK(data->id);
        COPY_SERVENT(data, tmp);
        UNLOCK(data->id);
        tmp->post_type = PING_ID;
        //!Invio PING al peer selezionato
        servent_send_packet(tmp);
        //!Attende la risposta
        ret = servent_pop_response(tmp);
        if(ret!=NULL && strcmp(ret, TIMEOUT)==0) {
         //!Entra in questo flusso se c'e' stato un timeout (failure detection)
         logger(SYS_INFO, "[servent_timer]Timer expired per %lld\n", data->id);
         //!Elimina il peer dalla GUI e dalle strutture dati
         gdk_threads_enter();
         controller_receive_bye(data->id);
         gdk_threads_leave();
         data_destroy_user(data->id);

         //!Uccide il client thread associato al peer
         tmp->post_type = CLOSE_ID;
         servent_send_packet(tmp);
        }
```

```
            logger(SYS_INFO, "[servent_timer]Signaling %lld\n", data->id);
        }
      }

      //!Libera le hashtable dei pacchetti
      servent_flush_data();
      logger(SYS_INFO, "[servent_timer]Sleeping\n");
      sleep(conf_get_timer_interval());
    }
}
```

1.18 controller.h

```
#ifndef CONTROLLER_H_
#define CONTROLLER_H_

#include "servent.h"
#include "common.h"
#include "socketmanager.h"
#include "datamanager.h"
#include "confmanager.h"
#include <glib.h>
#include "gui.h"
#include "utils.h"

pthread_t gtk_main_thread;

/**
 * cambia lo status di un peer tramite l'invio di un PING a tutti gli utenti
 * e controlla che il pacchetto sia stato ricevuto correttamente dagli altri peer
 */
int controller_change_status(u_int1 status);

/**
 * notifica il cambiamento di status di un peer remoto tramite una chiamata alla gui
 */
int controller_manipulating_status(u_int8 user_id, u_int1 status);

/**
 * invia un messaggio a tutti gli utenti di una chat e controlla che il pacchetto
 * sia stato ricevuto correttamente dagli altri peer.
 */
int controller_send_chat_users(u_int8 chat_id, u_int4 msg_len, char *msg);

/**
 * invia un messaggio ad un sottoinsieme di utenti di una chat e controlla che
 * il pacchetto sia stato ricevuto correttamente dagli utenti interessati.
 */
int controller_send_subset_users(u_int8 chat_id, u_int4 msg_len, char *msg, GList *users
    );

/**
 * invia un messaggio privato ad un utente e si accerta che questo sia stato
 * ricevuto correttamente.
 */
int controller_send_pm(u_int4 msg_len, char *msg, u_int8 recv_id);

/** permette all'utente di abbandonare tutte le chat a cui era connesso */
int controller_leave_all_chat();

/**
 * stabilisce una connessione tra un peer e una lista di utenti di una chat tramite
 * l'avvio del client.
 */
int controller_connect_users(GList *users);

/**
 * funzione d'appoggio per la controller_connect_users(), verifica che il peer
 * con cui si sta cercando di stabilire una connessione sia effettivamente pronto
     .??????????
 */
int controller_check_users_con(GList *users);

/**
```

```
 * permette l'invio di un pacchetto di tipo BYE che notifica a tutti gli utenti
 * conosciuti che si sta abbandonando l'applicazione e controlla che tutti abbiano
 *ricevuto il messaggio correttamente.
 */
int controller_send_bye();

/**
 * Alla ricezione di una BYE da un utente remoto chiude tutte le conversazioni
 * private aperte (se esistenti) con quello specifico utente.
 */
int controller_receive_bye();

/**
 * Legge il file di configurazione, avvia il logger, legge il file init_data che
 * contiene ip e porta dei peer vicini, avvia il servent e infine avvia il timer
 * necessario per il meccanismo di failure detection.
 */
int controller_init(const char *filename, const char *cache);

/**
 * Uccide tutti i thread e chiude il file di logger.
 */
int controller_exit();

/**
 * Costruisce la prima finestra dell'interfaccia grafica, relativa alla creazione
 * e alla ricerca delle chat e avvia l'apposito thread per la gestione degli eventi
 */
int controller_init_gui(void);

/**
 * Controlla che la query inserita sia accettabile, dopo di che invia un pacchetto
 * di tipo SEARCH a tutti gli utenti conosciuti, evitando l'invio ai fake id e a
 * se' stesso.
 */
u_int8 controller_search(const char *query);

/**
 * Invia un pacchetto di tipo JOIN a tutti gli utenti conosciuti (ammesso che ne conosca
 * qualcuno, evitando l'invio ai fake id e a se' stesso; successivamente, ogni volta
 * che riceve un messaggio di OK da un utente, aggiunge questi alla chat; infine
 * aggiunge anche se' stesso.
 */
int controller_join_flooding (u_int8 chat_id);

/**
 * Invia un pacchetto di tipo LEAVE a tutti gli utenti conosciuti per avvertire che
 * si sta abbandonando la chat (rappresentata dal parametro chat_id) e rimuove
 * questa dall'elenco delle chat a cui si e' connessi, dopo di che attende che tutti
 * abbiano ricevuto il messaggio correttamente.
 */
int controller_leave_flooding(u_int8 chat_id);

/**
 * Consente ad un utente di creare una chat purche' abbia un nome che non sia nullo o
 * rappresentato da una stringa vuota. Successivamente viene generato un id da associare
 * alla chat e infine viene aperta la gui relativa alla chat con conseguente aggiunta
 * dell'utente alla lista dei peer partecipanti alla chat.
 */
int controller_create(const char *title);

/**
 * Chiama la funzione gui_add_user_to_chat controllando prima che l'utente che
 * si sta cercando di inserire abbia tutti i campi inizializzati correttamente. Nel
```

```
 * caso invece l'utente non sia conosciuto e quindi sia inizialmente NULL viene
 * aggiunto alla chat un utente in modo provvisorio.
 */
int controller_add_user_to_chat(u_int8 chat_id, u_int8 id);

/**
 * Funzione d'appoggio che rimuove l'utente dalla chat a cui era connesso, sia a
 * livello di gui che a livello di data_manager.
 */
int controller_rem_user_from_chat(u_int8 chat_id, u_int8 id);

/**
 * Permette ad un utente, una volta ricevuto un messaggio da un partecipante
 * alla chat, di aggiornare la text view della chat stessa.
 */
int controller_add_msg_to_chat(u_int8 chat_id, char *msg);

/**
 * Come la controller_add_msg_to_chat, ma utilizzata nel caso venga ricevuto
 * un messaggio privato.
 */
int controller_add_msg(u_int8 sender_id, char *msg);

#endif /**CONTROLLER_H_*/
```

1.19 controller.c

```
#include "controller.h"

/**
 * cambia lo status di un peer tramite l'invio di un PING a tutti gli utenti
 * e controlla che il pacchetto sia stato ricevuto correttamente dagli altri peer
 */
int controller_change_status(u_int1 status)
{
 servent_data *peer, *sd;
 char *ret;
 if(servent_get_local () == NULL) {
  logger(CTRL_INFO,"[controller_change_status] local_servent not present\n");
  return -1;
 }
 WLOCK(servent_get_local()->id);
 servent_get_local()->status = status;  //! Cambio dello status dell'utente
 UNLOCK(servent_get_local()->id);
 logger(CTRL_INFO,"[controller_change_status] sending packet\n");
 GList *users = servent_get_values();
 int i=0;
 //!preparazione e invio del pacchetto con il nuovo status a tutti gli utenti
 for(; i < g_list_length(users); i++) {
  peer = g_list_nth_data(users, i);
  peer->post_type = PING_ID;
  peer->packet_id = generate_id();
  servent_send_packet(peer);
 }
 //!fase di attesa della ricezione del messaggio di OK (o di TIMEOUT).
 for(i=0; i<g_list_length(users); i++) {
  chatclient *client = (chatclient*)g_list_nth_data(users, i);
  if(client!=NULL) {
   peer = servent_get(client->id);
   logger(CTRL_INFO, "[controller_leave_chat]pop response %lld\n", client->id);
   if(peer!=NULL && peer->id!=servent_get_local()->id) {
        COPY_SERVENT(peer, sd);
        ret = servent_pop_response(peer);
        if(strcmp(ret, TIMEOUT)==0)
         return peer->id;
   }
  }
 }
 return 0;
}

/**
  * notifica il cambiamento di status di un peer remoto tramite una chiamata alla gui
  */
int controller_manipulating_status(u_int8 user_id, u_int1 status)
{
 char *status_tmp;
 if(user_id <= 0)
  return -1;
 if(status == ONLINE_ID)
  status_tmp = ONLINE;
 else if(status == BUSY_ID)
  status_tmp = BUSY;
 else if(status == AWAY_ID)
  status_tmp = AWAY;
 //!aggiornamento della gui con il nuovo status dell'utente
 gui_change_status(user_id, status_tmp);
 return 0;
```

```
}

/**
  * invia un messaggio a tutti gli utenti di una chat e controlla che il pacchetto
  * sia stato ricevuto correttamente dagli altri peer.
  */
int controller_send_chat_users(u_int8 chat_id, u_int4 msg_len, char *msg) {
 if(chat_id != 0) {
  servent_data *data, *tmp, *sd;
  chatclient *user;
  chat *chat_elem = data_get_chat(chat_id);
  char *ret;
  if(chat_elem==NULL) {
   logger(CTRL_INFO, "[controller_send_chat_users]Chat %lld not present\n", chat_id);
   return -2;
  }
  GList *users = g_hash_table_get_values(chat_elem->users);
  if(users==NULL) {
   logger(CTRL_INFO, "[controller_send_chat_users]Users list is empty\n");
   return -2;
  }
  logger(CTRL_INFO, "[controller_send_chat_users]Users size: %d\n", g_list_length(users)
      );

  int i=0;
  //!Preparazione del pacchetto di tipo MESSAGE e invio a tutti gli utenti della chat
  for(; i<g_list_length(users); i++) {

   user = g_list_nth_data(users, i);
   data = servent_get(user->id);
   if(data!=NULL && data->id!=servent_get_local()->id ) {
        RLOCK(data->id);
        COPY_SERVENT(data, tmp);
        UNLOCK(data->id);

        tmp->msg = strdup(msg);
        tmp->msg_len = msg_len;
        tmp->chat_id_req = chat_id;
        tmp->post_type = MESSAGE_ID;

        servent_send_packet(tmp);
        logger(CTRL_INFO, "[controller_send_chat_users]Sent msg\n");
   }
   else
        logger(CTRL_INFO, "[controller_send_chat_users]Servent not present\n");
  }
  //! Attesa di ricezione dei pacchetti di OK (o di TIMEOUT) inviati da tutti gli utenti
      .
  for(i=0; i<g_list_length(users); i++) {
   chatclient *client = (chatclient*)g_list_nth_data(users, i);
   if(client!=NULL) {
        data = servent_get(client->id);
        logger(CTRL_INFO, "[controller_leave_chat]pop response %lld\n", client->id);
        if(data!=NULL && data->id!=servent_get_local()->id) {
         ret = servent_pop_response(data);
         if(strcmp(ret, TIMEOUT)==0)
          return data->id;
        }
   }
  }
  return 0;
 }
 return -1;
}
```

```
/**
  * invia un messaggio ad un sottoinsieme di utenti di una chat e controlla che
  * il pacchetto sia stato ricevuto correttamente dagli utenti interessati.
  */
int controller_send_subset_users(u_int8 chat_id, u_int4 msg_len, char *msg, GList *users
    ) {
 char *ret;
 if(chat_id != 0) {
  int i= 0;
  logger(CTRL_INFO, "[controller_send_subset_users] chat id != 0\n");
  logger(CTRL_INFO, "[controller_send_subset_users] list length %d\n", g_list_length(
      users));
  //!Preparazione del pacchetto di tipo MESSAGE e invio a tutti gli utenti della lista
  for(; i<g_list_length(users); i++) {

   chatclient *user = g_list_nth_data(users, i);
   logger(CTRL_INFO, "[controller_send_subset_users] user id %lld\n",user->id);

   servent_data *data = servent_get(user->id);
   WLOCK(data->id);

   data->msg = strdup(msg);
   data->msg_len = msg_len;
   data->chat_id_req = chat_id;
   data->post_type = MESSAGE_ID;

   UNLOCK(data->id);
   logger(CTRL_INFO, "[controller_send_subset_users] sending packet\n");
   servent_send_packet(data);
  }
  //! Attesa di ricezione dei pacchetti di OK (o di TIMEOUT) inviati dagli utenti della
      lista.
  servent_data *sd, *data;
  for(i=0; i<g_list_length(users); i++) {
   chatclient *client = (chatclient*)g_list_nth_data(users, i);
   if(client!=NULL) {
        data = servent_get(client->id);
        logger(CTRL_INFO, "[controller_leave_chat]pop response %lld\n", client->id);
        if(data!=NULL && data->id!=servent_get_local()->id) {
         ret = servent_pop_response(data);
         if(strcmp(ret, TIMEOUT)==0)
          return data->id;
        }
   }
  }
  return 0;
 }
 return -1;
}

/**
  * invia un messaggio privato ad un utente e si accerta che questo sia stato
  * ricevuto correttamente.
  */
int controller_send_pm(u_int4 msg_len, char *msg, u_int8 recv_id) {
 logger(CTRL_INFO, "[controller_send_pm] receiver id %d\n", recv_id);
 char *ret;
 servent_data *sd;

 //!Preparazione e invio del messaggio privato all'utente con id pari a recv_id;
 servent_data *data = (servent_data*)servent_get(recv_id);

 WLOCK(data->id);
```

```
 data->msg = strdup(msg);
 data->msg_len = msg_len;
 data->chat_id_req = 0;
 data->post_type = MESSAGE_ID;

 UNLOCK(data->id);
 servent_send_packet(data);
 //! Attesa di ricezione del pacchetto di OK (o di TIMEOUT)
 servent_data *peer;
 peer = servent_get(recv_id);
 logger(CTRL_INFO, "[controller_leave_chat]pop response %lld\n", recv_id);
 if(peer!=NULL && peer->id!=servent_get_local()->id) {
  ret = servent_pop_response(peer);
  if(strcmp(ret, TIMEOUT)==0)
   return peer->id;

 }

 return 0;
}

/** permette all'utente di abbandonare tutte le chat a cui era connesso */
int controller_leave_all_chat()
{
 chat *tmp;
 servent_data *sd = servent_get_local();
 GList *iter;
 if(sd == NULL) {
  logger(CTRL_INFO, "[controller_leave_all_chat] local servent not present\n");
  return -1;
 }
 GList *chat_list =  sd->chat_list;
 int i=0;
 logger(CTRL_INFO, "[controller_leave_all_chat] list length %d\n", g_list_length(
      chat_list));
 //!Chiamata alla funzione controller_leave_flooding per ogni chat a cui si e' connessi
 while((iter=g_list_last(chat_list)) != NULL) {
  tmp = (chat*)iter->data;
  controller_leave_flooding(tmp->id);
  chat_list = sd->chat_list;
 }
 return 0;
}

/**
  * stabilisce una connessione tra un peer e una lista di utenti di una chat tramite
  * l'avvio del client.
  */
int controller_connect_users(GList *users) {
 int result = 0;

 if(users!=NULL) {
  int i, counter = 3;
  chatclient *client;
  GList *users_orig = users;
  servent_data *peer, *sd;
  GList *response = NULL;
  GList *timeout = NULL;
  char *ret;
  while(counter--) {

   if(timeout!=NULL) {
        users = timeout;
```

```
        logger(CTRL_INFO, "[controller_connect_users]Retrying\n");
    }

    for(i=0; i<g_list_length(users); i++) {
        client = (chatclient*)g_list_nth_data(users, i);
        logger(CTRL_INFO, "[controller_connect_users]Connecting to client: %s\n", client
            ->nick);
        logger(CTRL_INFO, "[controller_connect_users]Get local status: %s\n", to_string(
            servent_get_local()->status));
        if(servent_get(client->id)==NULL) {
         peer = servent_start_client(client->ip, client->port, 0);
         response = g_list_append(response, (gpointer)peer);

         ret = servent_pop_response(peer);

        }
        else
         logger(CTRL_INFO, "[controller_connect_users]Already connected\n");
        result = -2;
    }

    timeout = NULL;
    for(i=0; i<g_list_length(response); i++) {
        peer = (servent_data*)g_list_nth_data(response, i);
        logger(CTRL_INFO, "[controller_connect_users]pop response %lld\n", peer->id);
        if(peer!=NULL && peer->id!=servent_get_local()->id) {

         ret = servent_pop_response(peer);
         logger(CTRL_INFO, "[controller_connect_users]ret: %s\n", ret);
         if(strcmp(ret, TIMEOUT)==0) {
          logger(CTRL_INFO, "[controller_connect_users]TIMEOUT\n");
          //!Aggiunge alla lista dei non connessi (per ritentare)
          timeout = g_list_append(timeout, (gpointer)client);
         }
         else {
          logger(CTRL_INFO, "[controller_connect_users]RECEIVED OK %s\n", ret);
         }
        }
    }
    response = NULL;

    if(timeout==NULL) {
        counter=-1; //!connessioni avvenute con successo
        logger(CTRL_INFO, "[controller_connect_users]Checking users connections\n");
        controller_check_users_con(users_orig);
        break;
    }
   }
   if(counter==-1)
    return result;
  }

  return result;
}

/**
   * funzione d'appoggio per la controller_connect_users(), verifica che il peer
   * con cui si sta cercando di stabilire una connessione sia effettivamente pronto
       .??????????
   */
int controller_check_users_con(GList *users) {
  if(users==NULL)
   return -1;
```

```
 servent_data *client;
 chatclient *user;
 int counter=3, i;
 while(counter--) {
  for(i=0; i<g_list_length(users); i++) {
   user = g_list_nth_data(users, i);
   logger(CTRL_INFO, "[controller_check_users_con]User nick: %s\n", user->nick);
   if(user!=NULL) {
         client = servent_get(user->id);
         logger(CTRL_INFO, "[controller_check_users_con]client nick: %s\n", client->nick)
             ;
         if((client!=NULL) && (!client->is_online)) {
          logger(CTRL_INFO, "[controller_check_users_con]Servent %lld not ready\n",
               client->id);
          usleep(200000);
          continue;
         }
   }
   counter=0;
  }
 }

 return 0;
}

/**
  * permette l'invio di un pacchetto di tipo BYE che notifica a tutti gli utenti
  * conosciuti che si sta abbandonando l'applicazione e controlla che tutti abbiano
  *ricevuto il messaggio correttamente.
  */
int controller_send_bye()
{
 chatclient *client;
 servent_data *tmp, *peer, *sd;
 char *ret;
 if(servent_get_local() == NULL) {
  logger(CTRL_INFO,"[controller_send_bye] local_servent not present\n");
  return -1;
 }

 logger(CTRL_INFO,"[controller_send_bye] sending packet\n");

 GList *users = servent_get_values();
 int i=0;
 //!preparazione e invio del pacchetto di tipo BYE a tutti gli utenti conosciuti
 for(; i < g_list_length(users); i++) {
  peer = g_list_nth_data(users, i);
  logger(CTRL_INFO,"[controller_send_bye] sending packet to %lld from %lld\n", peer->id,
        servent_get_local()->id);
  if(peer!=NULL && peer->id!=servent_get_local()->id && (peer->id >= conf_get_gen_start
       ())) {
   RLOCK(peer->id);
   COPY_SERVENT(peer, tmp);
   UNLOCK(peer->id);
   tmp->post_type = BYE_ID;
   logger(CTRL_INFO, "[controller_send_bye] checking post type %d\n", tmp->post_type);
   tmp->packet_id = generate_id();
   servent_send_packet(tmp);
   logger(CTRL_INFO, "[controller_send_bye] sent\n");

  }
 }
 //! Attesa di ricezione dei pacchetti di OK (o di TIMEOUT) inviati da tutti gli utenti
 for(i=0; i<g_list_length(users); i++) {
```

```
   client = (chatclient*)g_list_nth_data(users, i);
   if(client!=NULL) {
    peer = servent_get(client->id);
    logger(CTRL_INFO, "[controller_send_bye]pop response %lld\n", client->id);
    if(peer!=NULL && peer->id!=servent_get_local()->id && (peer->id >= conf_get_gen_start
         ())) {
         RLOCK(peer->id);
         COPY_SERVENT(peer, sd);
         UNLOCK(peer->id);

         ret = servent_pop_response(sd);
         logger(CTRL_INFO, "[controller_send_bye]ret: %s\n", ret);
         if(strcmp(ret, TIMEOUT)==0) {
          logger(CTRL_INFO, "[controller_send_bye]TIMEOUT\n");
          return sd->id;
         }
         logger(CTRL_INFO, "[controller_send_bye]RECEIVED OK %s\n", ret);

    }
   }
  }

 return 0;
}

/**
  * Alla ricezione di una BYE da un utente remoto chiude tutte le conversazioni
  * private aperte (se esistenti) con quello specifico utente.
  */
int controller_receive_bye(u_int8 id)
{
 pm_data *pm;
 logger(CTRL_INFO, "[controller_receive_bye] user id %lld\n", id);
 if((pm = gui_pm_data_get(id)) != NULL) {
  gui_leave_pm_event(pm->window, (gpointer)to_string(id));
 }
 return 0;
}

/**
  * Legge il file di configurazione, avvia il logger, legge il file init_data che
  * contiene ip e porta dei peer vicini, avvia il servent e infine avvia il timer
  * necessario per il meccanismo di failure detection.
  */
int controller_init(const char *filename, const char *cache) {

 //!lettura del file di configurazione
 conf_read(filename);

 //!avvio del logger
 logger_init(conf_get_verbose());

 //!inserimento dei vicini presenti nel file init_data nella lista
 GList *init_list = NULL;
 init_list = init_read_file(cache);

 //!avvio del servente
 servent_start(init_list);

 //!avvio del timer per il meccanismo di failure detection
 servent_start_timer();

 return 0;
}
```

```
/**
  * Uccide tutti i thread e chiude il file di logger.
  */
int controller_exit() {
 servent_kill_all_thread(0);

 logger_close();
 return 0;
}

/**
  * Costruisce la prima finestra dell'interfaccia grafica, relativa alla creazione
  * e alla ricerca delle chat e avvia l'apposito thread per la gestione degli eventi
  */
int controller_init_gui(void) {

 /**--Dichiarazione dei widget della finestra --*/
 GtkWidget *window;
 GtkWidget *menu;
 GtkWidget *vbox;
 GtkWidget *handlebox;
 GtkWidget *list;
 GtkWidget *searchbar;

 /**-- Creazione della finestra --*/
 window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

 /**-- Creazione del vbox e della chat_list--*/
 vbox = gtk_vbox_new(FALSE, 5);
 list = gui_create_chat_list(0);
 /**-- Creazione dell'handlebox --*/
 handlebox = gtk_handle_box_new();

 /**-- creazione della menubar --*/
 menu = gui_create_menu();

 /**-- connette la finestra all'evento gui_close_event --*/
 gtk_signal_connect(GTK_OBJECT(window), "delete_event", GTK_SIGNAL_FUNC(gui_close_event)
      , NULL);

 /**-- Aggiunge il menubar all'handlebox --*/
 gtk_container_add(GTK_CONTAINER(handlebox), menu);

 /**-- creazione dell'area della finestra relativa alla ricerca e alla creazione della
     chat --*/
 searchbar = gui_create_searchbar();

 /**-- aggiunta dei componenti handlebox, searchbar e list alla vbox --*/
 gtk_box_pack_start(GTK_BOX(vbox), handlebox, FALSE, TRUE, 5);
 gtk_box_pack_start(GTK_BOX(vbox), searchbar, FALSE, TRUE, 5);

 gtk_container_add(GTK_CONTAINER(vbox),list);

 /**-- Aggiunta del vbox alla finestra principale --*/
 gtk_container_add(GTK_CONTAINER(window), vbox);

 /**-- setting delle dimensioni e del titolo della window --*/
 gtk_container_border_width (GTK_CONTAINER (window), 0);
 gtk_window_set_default_size (GTK_WINDOW(window), 300, 400);
 gtk_window_set_title(GTK_WINDOW (window), "Lista Chat");


 /**-- Mostra i widget --*/
```

```
 gtk_widget_show_all(window);

 /**-- Start the GTK event loop --*/
 gtk_main();
 return 1;
}

/**
  * Controlla che la query inserita sia accettabile, dopo di che invia un pacchetto
  * di tipo SEARCH a tutti gli utenti conosciuti, evitando l'invio ai fake id e a
  * se' stesso.
  */
u_int8 controller_search(const char *query) {
 if(query==NULL || strcmp(query, "")==0) {
  logger(CTRL_INFO, "[controller_search]Query string not valid\n");
  return 0;
 }

 GList *servents = servent_get_values();
 logger(CTRL_INFO, "[controller_search]Query: %s\n", query);
 if(servents==NULL) {
  logger(CTRL_INFO, "[controller_search]Servents list is empty\n");
  return 0;
 }
 servent_data *servent, *tmp;
 int i=0;
 /** Preparazione e invio del pacchetto di SEARCH a tutti gli utenti conosciuti */
 for(; i<g_list_length(servents); i++) {
  servent = g_list_nth_data(servents, i);
  logger(CTRL_INFO, "[controller_search]Servent ID: %lld\n", servent->id);

  //!controllo per evitare l'auto-invio del pacchetto e l'invio ai fake id
  if(servent->id!=servent_get_local()->id && servent->id >= conf_get_gen_start ()) {

   if(servent->queue==NULL) {
        continue;
   }
   RLOCK(servent->id);
   logger(INFO, "[controller_search]Copy servent\n");
   COPY_SERVENT(servent, tmp);
   UNLOCK(servent->id);
   if(tmp->queue==NULL || tmp->res_queue==NULL) {
        continue;
   }
   logger(INFO, "[controller_search]Copy\n");
   tmp->post_type = SEARCH_ID;
   tmp->title = strdup(query);
   tmp->title_len = strlen(query);
   tmp->ttl = 3;
   tmp->hops = 0;
   tmp->packet_id = generate_id(); //!Se non ci fosse verrebbe riutilizzato l'ID di uno
        degli eventuali pacchetti SEARCH ritrasmessi
   logger(INFO, "[controller_search]Send\n");
   servent_send_packet(tmp);
   logger(INFO, "[controller_search]Sent\n");
  }
  else
   logger(INFO, "[controller_search]Local ID\n");

 }

 logger(INFO, "[controller_search]End\n");

 return 0;
```

```
}

/**
  * Invia un pacchetto di tipo JOIN a tutti gli utenti conosciuti (ammesso che ne
      conosca
  * qualcuno, evitando l'invio ai fake id e a se' stesso; successivamente, ogni volta
  * che riceve un messaggio di OK da un utente, aggiunge questi alla chat; infine
  * aggiunge anche se' stesso.
  */
int controller_join_flooding(u_int8 chat_id) {
 if(chat_id > 0) {
  GList *servents = servent_get_values();
  if(servents==NULL) {
   logger(CTRL_INFO, "[controller_join]Servents null\n");
   return 0;
  }
  servent_data *servent, *tmp, *peer;
  chat *chat_elem = data_get_chat(chat_id);
  if(chat_elem != NULL) {
   int i=0;
   int count = 0;
   //! Preparazione e invio del pacchetto di JOIN a tutti gli utenti conosciuti
   for(; i<g_list_length(servents); i++) {
        servent = g_list_nth_data(servents, i);
        logger(CTRL_INFO, "[controller_join]Servent ID: %lld;  nick: %s\n", servent->id,
             servent->nick);

        //!controllo per evitare l'auto-invio del pacchetto e l'invio ai fake id
        if(servent->id!=servent_get_local()->id && servent->id >= conf_get_gen_start ())
             {

         if(servent->queue==NULL) {
          logger(CTRL_INFO, "[controller_join]Coda Servent NULL\n");
          continue;
         }
         RLOCK(servent->id);
         logger(INFO, "[controller_join]Copy servent\n");
         COPY_SERVENT(servent, tmp);
         UNLOCK(servent->id);
         if(tmp->queue==NULL || tmp->res_queue==NULL) {
          continue;
         }
         logger(INFO, "[controller_join]Copy\n");
         tmp->post_type = JOIN_ID;
         tmp->ttl = 3;
         tmp->hops = 0;
         tmp->nick_req = servent_get_local()->nick;
         tmp->packet_id = generate_id(); //!Se non ci fosse verrebbe riutilizzato l'ID
              di uno degli eventuali pacchetti SEARCH ritrasmessi
         tmp->chat_id_req = chat_id;
         tmp->user_id_req = servent_get_local()->id;
         tmp->status_req = servent_get_local()->status;
         tmp->ip_req = servent_get_local()->ip;
         tmp->port_req = servent_get_local()->port;
         if(count == 0) {
          servent_get_local()->chat_list = g_list_append(servent_get_local()->chat_list,
               (gpointer)chat_elem);
          count = 1;
         }
         logger(INFO, "[controller_join]Send\n");
         servent_send_packet(tmp);
         logger(INFO, "[controller_join]Sent\n");
        }
        else
```

```
        logger(INFO, "[controller_join]Local ID\n");

    }
    GList *users = data_get_chatclient_from_chat(chat_elem->id);
    int j;
    //! Attesa di ricezione dei pacchetti di OK (o di TIMEOUT).
    for(i=0; i<g_list_length(servents); i++) {
        peer = (servent_data*)g_list_nth_data(servents, i);
        chatclient *client = data_get_chatclient(peer->id);
        logger(CTRL_INFO, "[controller_join_chat]Sending join to %lld\n", peer->id);
        if(client!=NULL && peer!=NULL && peer->id!=servent_get_local()->id && peer->id
            >= conf_get_gen_start ()) {
         char *ret = servent_pop_response(peer);
         if(ret==NULL) {
          return -1;
         }
         if(strcmp(ret, TIMEOUT)==0)
          return peer->id;

         for(j=0; j<g_list_length(users); j++) {
          chatclient *user = (chatclient*)g_list_nth_data(users, j);
          if(user!=NULL) {
           if(user->id == client->id) {
                //!Aggiunta dell'utente remoto alla chat e conseguente aggiornamento
                    della gui
                data_add_user_to_chat(chat_elem->id, client->id, client->nick, client->
                    ip, client->port);
                gui_add_user_to_chat(chat_elem->id, client->id, client->nick, peer->
                    status);
                break;
           }
          }
         }
        }
    }
   }
   //!Aggiunta dell'utente locale alla chat e conseguente aggiornamento della gui
   gui_add_user_to_chat(chat_elem->id, servent_get_local()->id, servent_get_local()->nick
       , servent_get_local()->status);
   data_add_user_to_chat(chat_elem->id, servent_get_local()->id, servent_get_local()->
       nick, servent_get_local()->ip, servent_get_local()->port);
   return 0;
  }
  logger(INFO, "[controller_join]End\n");

  return 0;
}

/**
  * Invia un pacchetto di tipo LEAVE a tutti gli utenti conosciuti per avvertire che
  * si sta abbandonando la chat (rappresentata dal parametro chat_id) e rimuove
  * questa dall'elenco delle chat a cui si e' connessi, dopo di che attende che tutti
  * abbiano ricevuto il messaggio correttamente.
  */
int controller_leave_flooding(u_int8 chat_id) {
 char *ret;
 logger(CTRL_INFO,"[controller_leave_chat] chat_id %lld\n", chat_id);
 if(chat_id>0) {
  logger(CTRL_INFO,"[controller_leave_chat] chat_id>0\n");
  chat *chat_elem = data_get_chat(chat_id);
  if(chat_elem!=NULL) {
   logger(CTRL_INFO,"[controller_leave_chat] chat present\n");
   GList *clients = servent_get_values();
   chatclient *client;
```

```
    servent_data *peer, *sd;
    int i;

    //! Preparazione e invio del pacchetto di LEAVE a tutti gli utenti conosciuti
    for(i=0; i<g_list_length(clients); i++) {
          client = (chatclient*)g_list_nth_data(clients, i);
          if(client!=NULL) {
           logger(CTRL_INFO,"[controller_leave_chat] client present\n");

           peer = servent_get(client->id);
           if(peer!=NULL && peer->id!=servent_get_local()->id) {
            logger(CTRL_INFO,"[controller_leave_chat] peer present\n");

            WLOCK(peer->id);
            COPY_SERVENT(peer, sd);
            sd->chat_id_req = chat_id;
            sd->user_id_req = servent_get_local()->id;
            sd->post_type = LEAVE_ID;
            sd->packet_id = generate_id();
            sd->ttl = 3;
            sd->hops = 0;

            UNLOCK(peer->id);
            servent_send_packet(sd);
           }
          }
    }
    //! Attesa di ricezione dei pacchetti di OK (o di TIMEOUT).
    for(i=0; i<g_list_length(clients); i++) {
          client = (chatclient*)g_list_nth_data(clients, i);
          if(client!=NULL) {
           peer = (servent_data*)g_list_nth_data(clients, i);
           logger(CTRL_INFO, "[controller_leave_chat]pop response %lld\n",peer->id);
           if(peer!=NULL && peer->id!=servent_get_local()->id) {
            COPY_SERVENT(peer, sd);

            ret = servent_pop_response(peer);
            if(strcmp(ret, TIMEOUT)==0)
             return peer->id;
           }
          }
    }
    servent_get_local()->chat_list = g_list_remove(servent_get_local()->chat_list, (
         gconstpointer)chat_elem);
    data_del_user_from_chat (chat_id, servent_get_local()->id);
    return 0;
   }
  }
  return -1;
}

/**
   * Consente ad un utente di creare una chat purche' abbia un nome che non sia nullo o
   * rappresentato da una stringa vuota. Successivamente viene generato un id da
       associare
   * alla chat e infine viene aperta la gui relativa alla chat con conseguente aggiunta
   * dell'utente alla lista dei peer partecipanti alla chat.
   */
int controller_create(const char *title) {
  if(title==NULL || strcmp(title, "")==0) {
   logger(CTRL_INFO, "[controller_create]Chat title is invalid\n");
   return -1;
  }
```

```
 //!Generazione degll'id e aggiunta della chat all'hashtable delle chat
 u_int8 chat_id = generate_id();
 data_add_chat(chat_id, title);

 servent_data *local = servent_get_local();
 //!aggiunta dell'utente ai client connessi alla chat
 data_add_user_to_chat(chat_id, local->id, local->nick, local->ip, local->port);
 chat *test = data_get_chat(chat_id);
 //!aggiunta della chat alla lista locale delle chat a cui si e' connessi
 local->chat_list = g_list_append(local->chat_list, (gpointer)test);

 //!apertura della finestra della chat e aggiornamento dei dati
 gui_open_chatroom(chat_id);
 data_add_existing_user_to_chat(chat_id, servent_get_local()->id);
 gui_add_user_to_chat(chat_id, local->id, local->nick, local->status);
 return 0;
}

/**
  * Chiama la funzione gui_add_user_to_chat controllando prima che l'utente che
  * si sta cercando di inserire abbia tutti i campi inizializzati correttamente. Nel
  * caso invece l'utente non sia conosciuto e quindi sia inizialmente NULL viene
  * aggiunto alla chat un utente in modo provvisorio.
  */
int controller_add_user_to_chat(u_int8 chat_id, u_int8 id) {
 servent_data *servent = servent_get(id);

 if(servent ==NULL) {
  logger(CTRL_INFO, "[controller_add_user_to_chat] servent not present\n");
  chatclient *tmp = data_get_chatclient(id);
  //!Aggiornamento provvisorio della lista degli utenti della chat a livello di GUI
  gui_add_user_to_chat(chat_id, tmp->id, tmp->nick, 0);
  return 0;
 }
 logger(CTRL_INFO, "[controller_add_user_to_chat]Adding user: %s, id: %lld, status: %c\n
     ", servent->nick, servent->id, servent->status);
 //!Aggiornamento della lista degli utenti della chat a livello di GUI
 gui_add_user_to_chat(chat_id, servent->id, servent->nick, servent->status);

 return 0;
}

/**
  * Funzione d'appoggio che rimuove l'utente dalla chat a cui era connesso, sia a
  * livello di gui che a livello di data_manager.
  */
int controller_rem_user_from_chat(u_int8 chat_id, u_int8 id) {
 gui_del_user_from_chat(chat_id, id);
 data_del_user_from_chat(chat_id, id);
 return 0;
}

/**
  * Permette ad un utente, una volta ricevuto un messaggio da un partecipante
  * alla chat, di aggiornare la text view della chat stessa.
  */
int controller_add_msg_to_chat(u_int8 chat_id, char *msg) {

 if(msg==NULL) {
  logger(CTRL_INFO, "[controller_add_msg_to_chat]Message invalid\n");
  return -1;
 }

 if(chat_id>0) {
```

```
   if(gui_add_msg_to_chat(chat_id, msg)<0) {
    logger(CTRL_INFO, "[controller_add_msg_to_chat]Message error\n");
    return -2;
   }
   return 0;
  }

 return -3;
}

/**
  * Come la controller_add_msg_to_chat, ma utilizzata nel caso venga ricevuto
  * un messaggio privato.
  */
int controller_add_msg(u_int8 sender_id, char *msg) {

 if(msg==NULL) {
  logger(CTRL_INFO, "[controller_add_msg]Message invalid\n");
  return -1;
 }

 if(sender_id>0) {
  if(gui_add_msg_pm(sender_id, msg)<0) {
   logger(CTRL_INFO, "[controller_add_msg]Adding pm to gui error\n");
   return -2;
  }
  return 0;
 }
 return -3;
}
```

1.20 gui.h

```
#ifndef GUI_H
#define GUI_H

#include <stdio.h>
#include <stdlib.h>
#include <gtk/gtk.h>
#include <glib.h>
#include <gdk/gdkkeysyms.h>
#include "utils.h"
#include "controller.h"
#include "datamanager.h"
#include "tortellaprotocol.h"

#define TOP             0x01
#define BOTTOM  0x02
#define CHAT    0x03
#define PM              0x04

#define ONLINE "Online"
#define BUSY   "Busy"
#define AWAY   "Away"

static GtkListStore *chat_model;
static GtkTreeIter   chat_iter;
static GHashTable *tree_model_hashtable = NULL;
static GHashTable *pm_data_hashtable = NULL;

GtkWidget *bar_textfield;

/**
 * Contiene i componenti della gui che vengono modificati costantemente, ovvero
 * le due text view (invio e ricezione dei messaggi) e i due componenti necessari
 * per gestire la lista degli utenti.
 */
struct tree_model
{
 GtkListStore *user_model;
 GtkTreeIter user_iter;
 GtkTextView *text_area;
 GtkTreeView *tree_view;
};
typedef struct tree_model tree_model;

/**
  * Contiene i componenti della finestra del pm che vengono modificati, ovvero
  * il widget relativo alla finestra e la text_area dei messaggi ricevuti.
  */
struct pm_data {
 GtkWidget *window;
 GtkTextView *text_area;
};
typedef struct pm_data pm_data;


//!FUNZIONI PER LA CREAZIONE DELLE GUI

/**
  * Provoca la semplice chiusura di una finestra
  */
gint gui_close_window_event(GtkWidget *widget, gpointer gdata);
```

```
/**
  * Consente l'uscita corretta dal programma, invia un messaggio di BYE e effettua
  * il leave da ogni chat a cui si e' connessi, infine chiama la controller_exit e
  * termina l'interfaccia grafica.
  */
void gui_close_event (GtkWidget *widget, gpointer gdata);

/**
  * Funzione chiamata alla selezione di una chat, chiama il controller per gestire
  * la connessione agli utenti della chat e l'invio della JOIN, inoltre lancia la
  * finestra relativa alla chat.
  */
void gui_open_chat_event (GtkTreeView *treeview, GtkTreePath *path, GtkTreeViewColumn *
    col, gpointer userdata);

/**
  * Invocata quando viene chiusa la chat, effettua la chiamata al controller per
  * fargli gestire l'invio dei pacchetti di tipo LEAVE
  */
gint gui_leave_chat_event(GtkWidget *widget, gpointer gdata);

/**
  * Chiude la finestra relativa ad una conversazione privata e la rimuove dalla
  * hashtable.
  */
gint gui_leave_pm_event(GtkWidget *widget, gpointer gdata);

/**
  * Aggiunge una chat alla lista delle chat in seguito ad una ricerca che ha prodotto
  * risultati validi
  */
gint gui_add_chat(u_int8 chat_id, char *chat_name);

/**
  * Effettua un clear della lista delle chat
  */
gint gui_clear_chat_list();

/**
  * Effettua un clear della text view in cui vengono scritti i messaggi da inviare.
  * Viene invocata ogni volta che viene inviato un messaggio.
  */
gint gui_clear_buffer(GtkTextView *widget);

/**
  * Aggiunge alla chat il messaggio appena inviato e aggiorna lo scrolling della
  * finestra in modo che l'ultimo messaggio sia ben visibile.
  */
gint gui_add_message(GtkTextView *widget, gchar *msg);

/**
  * Aggiunge una utente alla chat, effettuando un controllo su eventuali duplicati
  * in modo da poterli scartare.
  */
gint gui_add_user_to_chat(u_int8 chat_id, u_int8 id, char *user, u_int1 status);

/**
  * Scorre la lista degli utenti connessi fino a quando non viene trovato l'id
  * dell'utente da eliminare, dopo di che questo viene eliminato
  */
gint gui_del_user_from_chat(u_int8 chat_id, u_int8 user_id);

/**
  * Scorre la lista degli utenti connessi fino a quando non viene trovato il record
```

```
  * da aggiornare, dopo di che viene aggiornato il campo status.
  */
gint gui_change_status(u_int8 user_id, char *status);

/**
  * Evento che si scatena quando si effettua un doppio click su un utente della chat.
  * Permette di aprire una nuova finestra per una conversazione privata.
  */
gint gui_open_conversation_event(GtkTreeView *treeview, GtkTreePath *path,
    GtkTreeViewColumn  *col, gpointer userdata);

/**
  * Evento che permette di aprire la finestra relativa all'about.
  */
gint gui_open_about_event(GtkWidget *widget, gpointer gdata);

/**
  * Setta lo stato dell'utente su Online e invoca il controller_change_status
  * in modo che quest'ultimo possa inviare un pacchetto di PING con il nuovo status
  */
gint gui_set_to_online_event(GtkWidget *widget, gpointer gdata);

/**
  * Setta lo stato dell'utente su Busy e invoca il controller_change_status
  * in modo che quest'ultimo possa inviare un pacchetto di PING con il nuovo status
  */
gint gui_set_to_busy_event(GtkWidget *widget, gpointer gdata);

/**
  * Setta lo stato dell'utente su Away e invoca il controller_change_status
  * in modo che quest'ultimo possa inviare un pacchetto di PING con il nuovo status
  */
gint gui_set_to_away_event(GtkWidget *widget, gpointer gdata);

/**
  * Evento che si scatena alla pressione del bottone search, effettua un clear della
  * lista delle chat e invoca il controller_search in modo che quest'ultimo possa
  * inviare il pacchetto di SEARCH.
  */
gint gui_search_chat_event(GtkWidget *widget, gpointer gdata);

/**
  * Prepara il corpo del messaggio da inviare a tutta la chat o a un sottoinsieme di
  * questa, dopo di che invoca il controller affinche' questo gestisca l'invio del
  * messaggio e infine aggiorna la text view locale col nuovo messaggio.
  */
gint gui_send_text_message_event(GtkWidget *widget, GdkEventKey *event, gpointer gdata);

/**
  * Come la send_text_message_event, ma rivolta all'invio di messaggi privati
  */
gint gui_send_pm_message_event(GtkWidget *widget, GdkEventKey *event, gpointer gdata);

/**
  * Crea il componente grafico che contiene la lista degli utenti connessi ad una chat.
  */
GtkWidget *gui_create_users_list(u_int8 index);

/**
  * Crea il componente grafico che contiene la lista delle chat trovate.
  */
GtkWidget *gui_create_chat_list(u_int8 index);

/**
```

```
  * Crea la barra dei menu situata su ogni finestra. (File -- Status -- Help).
  */
GtkWidget *gui_create_menu(void);

/**
  * Crea le varie text view (di scrittura e lettura) per la chat e per il pm,
  * impostando i valori in modo opportuno.
  */
GtkWidget *gui_create_text(u_int8 chat_id, int type, int msg_type);

/**
  * Crea il componente grafico presente nella finestra iniziale in cui si puo'
  * effettuare la creazione o la ricerca di una chat.
  */
GtkWidget *gui_create_searchbar(void);

/**
  * Evento che si scatena alla pressione del bottone create, fa' una semplice
  * chiamata alla controller_create.
  */
gint gui_create_chat_button(GtkWidget *widget, gpointer gdata);

/**
  * Crea la finestra relativa ad una singola chat e assegna ai componenti gli
  * eventi appropriati
  */
int gui_open_chatroom(u_int8);

/**
  * Crea la finestra per una conversazione privata
  */
int gui_open_pm(u_int8 user_id, gchar *nickname);

/**
  * Controlla nell'hashtable la presenza o meno del tree_model corrispondente alla
  * chat.
  */
tree_model *gui_get_tree_model(u_int8 chat_id);

/**
  * Aggiunge alla text view della chat il nuovo messaggio.
  */
int gui_add_msg_to_chat(u_int8 chat_id, char *msg);

/**
  * Aggiunge alla text view della conversazione privata il nuovo messaggio.
  */
int gui_add_msg_pm(u_int8 sender_id, char *msg);

/**
  * Controlla nell'hashtable la presenza o meno della struct pm_data relativa
  * all'id dell'utente.
  */
pm_data *gui_pm_data_get(u_int8 id);

/**
  * Recupera dal tree_model associato alla gui della chat la lista degli id degli
  * utenti connessi.
  */
GList *gui_get_chat_users(u_int8 chat_id);

#endif //!GUI_H
```

1.21 gui.c

```
#include "gui.h"

/**
 * Provoca la semplice chiusura di una finestra
 */
gint gui_close_window_event(GtkWidget *widget, gpointer gdata) {
 gtk_widget_destroy(widget);
 return(FALSE);
}

/**
  * Consente l'uscita corretta dal programma, invia un messaggio di BYE e effettua
  * il leave da ogni chat a cui si e' connessi, infine chiama la controller_exit e
  * termina l'interfaccia grafica.
  */
void gui_close_event (GtkWidget *widget, gpointer gdata)
{
 controller_leave_all_chat();
 controller_send_bye();
 controller_exit();
 gtk_main_quit();
}

/**
  * Invocata quando viene chiusa la chat, effettua la chiamata al controller per
  * fargli gestire l'invio dei pacchetti di tipo LEAVE
  */
gint gui_leave_chat_event(GtkWidget *widget, gpointer gdata)
{
 char *str = (char*)gdata;
 u_int8 val = atoll((char*)gdata);
 controller_leave_flooding(val);
 return(FALSE);
}

/**
  * Chiude la finestra relativa ad una conversazione privata e la rimuove dalla
  * hashtable.
  */
gint gui_leave_pm_event(GtkWidget *widget, gpointer gdata)
{
 u_int8 val = atoll((char*)gdata);
 pm_data *pm;
 if((pm = g_hash_table_lookup(pm_data_hashtable, (gconstpointer)to_string(val)))!=NULL)
      {
  gtk_widget_destroy(GTK_WIDGET(pm->window));
  g_hash_table_remove(pm_data_hashtable, (gconstpointer)to_string(val));
 }
 return (FALSE);
}

/**
  * Aggiunge una chat alla lista delle chat in seguito ad una ricerca che ha prodotto
  * risultati validi
  */
gint gui_add_chat(u_int8 chat_id, char *chat_name)
{
 GtkTreeIter iter;
 gboolean valid = TRUE;
 gchar *id;
 //!scorre la lista delle chat visualizzate
```

```
 if(gtk_tree_model_get_iter_first(GTK_TREE_MODEL(chat_model), &iter)==TRUE) {
  while(valid) {

   gtk_tree_model_get(GTK_TREE_MODEL(chat_model), &iter, 0, &id, -1);
   logger(INFO, "[add_chat_to_list]ID to add: %s\n", id);

   if(atoll(id)==chat_id) {
        logger(INFO, "[add_chat_to_list]Exiting: %s\n", id);
        return -1;
   }

   valid = gtk_tree_model_iter_next(GTK_TREE_MODEL(chat_model), &iter);
  }
 }

 //!prepara i campi da inserire nel nuovo record
 gchar *msg = g_strdup_printf (to_string(chat_id));
 gchar *msg1 = g_strdup_printf (chat_name);
 //!inserimento del record
 gtk_list_store_append (chat_model, &chat_iter);
 gtk_list_store_set (GTK_LIST_STORE(chat_model), &chat_iter, 0, msg, 1, msg1, -1);
 g_free (msg);
 g_free (msg1);
 return 0;
}

/**
  * Aggiunge una utente alla chat, effettuando un controllo su eventuali duplicati
  * in modo da poterli scartare.
  */
gint gui_add_user_to_chat(u_int8 chat_id, u_int8 id, char *user, u_int1 status)
{
 logger(INFO, "[gui_add_user_to_chat] Adding user %lld\n", id);
 gchar *msg = g_strdup_printf(user);
 if(tree_model_hashtable==NULL) {
  logger(INFO, "[add_user_to_chat_list]User list not ready\n");
  return -1;
 }
 tree_model *mod = (tree_model*)g_hash_table_lookup(tree_model_hashtable,(gconstpointer)
      to_string(chat_id));

 if(mod!=NULL) {
  GtkTreeIter iter, iter2;
  gchar *id_tmp;
  gboolean valid = TRUE;
  gboolean valid2 = TRUE;
  //!controlla che sia presente almeno un utente
  if(gtk_tree_model_get_iter_first(GTK_TREE_MODEL(mod->user_model), &iter)==TRUE) {
   gtk_tree_model_get_iter_first(GTK_TREE_MODEL(mod->user_model), &iter2);
   valid2 = gtk_tree_model_iter_next(GTK_TREE_MODEL(mod->user_model), &iter2);
   //!ciclo su tutti i record della lista
   while(valid) {
        gtk_tree_model_get(GTK_TREE_MODEL(mod->user_model), &iter, 1, &id_tmp, -1);
        logger(INFO, "[gui_add_user_to_chat]ID: %s\n", id_tmp);
        if(atoll(id_tmp) == id) {
         logger(INFO, "[gui_add_user_to_chat] existing user\n");
         break;
        }
        //!controllo che l'id non sia gia' presente
        if(atoll(id_tmp)!= id && valid2 == FALSE ) {

         logger(INFO, "[add_user_to_chat_list]User list OK\n");
         //!inserimento del nuovo record
         gtk_list_store_append(GTK_LIST_STORE(mod->user_model), &(mod->user_iter));
```

```
          gtk_list_store_set(GTK_LIST_STORE(mod->user_model), &(mod->user_iter), 0, msg,
               -1);
          g_free(msg);

          msg = g_strdup_printf(to_string(id));
          gtk_list_store_set(GTK_LIST_STORE(mod->user_model), &(mod->user_iter), 1, msg,
               -1);
          g_free(msg);

          if(status == ONLINE_ID)
           msg = g_strdup_printf(ONLINE);
          else if(status == BUSY_ID)
           msg = g_strdup_printf(BUSY);
          else if(status == AWAY_ID)
           msg = g_strdup_printf(AWAY);
          gtk_list_store_set(GTK_LIST_STORE(mod->user_model), &(mod->user_iter), 2, msg,
               -1);
          g_free(msg);
         }

         valid = gtk_tree_model_iter_next(GTK_TREE_MODEL(mod->user_model), &iter);
         if(valid2!= FALSE);
         valid2 = gtk_tree_model_iter_next(GTK_TREE_MODEL(mod->user_model), &iter2);
    }
   }
   else
   {
    //!nel caso non siano presenti altri utenti l'iteratore non e' valido, quindi si
         inserisce il primo record
    logger(INFO, "inserting first user\n");
    gtk_list_store_append(GTK_LIST_STORE(mod->user_model), &(mod->user_iter));
    gtk_list_store_set(GTK_LIST_STORE(mod->user_model), &(mod->user_iter), 0, msg, -1);
    g_free(msg);

    msg = g_strdup_printf(to_string(id));
    gtk_list_store_set(GTK_LIST_STORE(mod->user_model), &(mod->user_iter), 1, msg, -1);
    g_free(msg);
    if(status == ONLINE_ID)
          msg = g_strdup_printf(ONLINE);
    else if(status == BUSY_ID)
          msg = g_strdup_printf(BUSY);
    else if(status == AWAY_ID)
          msg = g_strdup_printf(AWAY);
    gtk_list_store_set(GTK_LIST_STORE(mod->user_model), &(mod->user_iter), 2, msg, -1);
    g_free(msg);
   }

  }
  else
   return -2;


 return 0;
}

/**
  * Scorre la lista degli utenti connessi fino a quando non viene trovato l'id
  * dell'utente da eliminare, dopo di che questo viene eliminato
  */
gint gui_del_user_from_chat(u_int8 chat_id, u_int8 user_id)
{
 tree_model *mod = (tree_model*)g_hash_table_lookup(tree_model_hashtable,(gconstpointer)
      to_string(chat_id));
 if(mod==NULL) {
```

```
  logger(INFO, "[gui_del_user_from_chat]Not connected to chad %lld\n", chat_id);
  return -1;
 }
 GtkTreeIter iter;
 gboolean valid = TRUE;
 gchar *id;
 //!posizionamento dell'iteratore al primo record e successivo scorrimento della lista
 if(gtk_tree_model_get_iter_first(GTK_TREE_MODEL(mod->user_model), &iter)==TRUE) {
  while(valid) {

   logger(INFO, "[remove_user_from_chat_list]start\n");
   gtk_tree_model_get(GTK_TREE_MODEL(mod->user_model), &iter, 1, &id, -1);
   logger(INFO, "[remove_user_from_chat_list]ID to remove: %s\n", id);

   //!se l'utente e' presente lo si rimuove dalla lista
   if(atoll(id)==user_id) {
        logger(INFO, "[remove_user_from_chat_list]Removing: %s\n", id);
        gtk_list_store_remove(mod->user_model, &iter);
        logger(INFO, "[remove_user_from_list]Removed\n");
        return TRUE;
   }

   valid = gtk_tree_model_iter_next(GTK_TREE_MODEL(mod->user_model), &iter);
   logger(INFO, "[remove_user_from_chat_list]next user\n");

  }
 }
 return (FALSE);
}

/**
  * Scorre la lista degli utenti connessi fino a quando non viene trovato il record
  * da aggiornare, dopo di che viene aggiornato il campo status.
  */
gint gui_change_status(u_int8 user_id, char *status)
{
 if(tree_model_hashtable == NULL) {
  logger(INFO, "[gui_change_status] user list not ready\n");
  return (FALSE);
 }
 GList *chat_id_list = g_hash_table_get_keys(tree_model_hashtable);
 int i=0;
 char *id;
 for(; i < g_list_length(chat_id_list); i++) {
  u_int8 chat_id = atoll(g_list_nth_data(chat_id_list,i));
  logger(INFO, "[gui_change_status] retrieving chat_id %lld\n",chat_id);

  tree_model *chat_model_tmp = (tree_model*)g_hash_table_lookup(tree_model_hashtable, (
       gconstpointer)to_string(chat_id));
  GtkTreeIter iter;
  gboolean valid = TRUE;

  //!posizionamento dell'iteratore al primo record e successivo scorrimento della lista
  if(gtk_tree_model_get_iter_first(GTK_TREE_MODEL(chat_model_tmp->user_model), &iter)==
       TRUE) {

   while(valid) {
        gtk_tree_model_get(GTK_TREE_MODEL(chat_model_tmp->user_model), &iter, 1, &id,
             -1);
        logger(INFO, "[gui_change_status]ID: %s\n", id);

        //!trovato l'utente si aggiorna il campo status
        if(atoll(id)==user_id) {
         gtk_list_store_set(GTK_LIST_STORE(chat_model_tmp->user_model), &iter, 2, status
```

```
                , -1);
            logger(INFO, "[gui_change_status] changed status %s\n", status);

          }

          valid = gtk_tree_model_iter_next(GTK_TREE_MODEL(chat_model_tmp->user_model), &
              iter);
    }
   }
  }
  return (FALSE);
}

/**
  * Effettua un clear della lista delle chat
  */
gint gui_clear_chat_list()
{
 gtk_list_store_clear(chat_model);
 return (FALSE);
}

/**
  * Effettua un clear della text view in cui vengono scritti i messaggi da inviare.
  * Viene invocata ogni volta che viene inviato un messaggio.
  */
gint gui_clear_buffer(GtkTextView *widget)
{
 GtkTextBuffer *text;
 text = gtk_text_buffer_new(NULL);
 gtk_text_view_set_buffer(GTK_TEXT_VIEW(widget),text);
 return (FALSE);
}

/**
  * Aggiunge alla chat il messaggio appena inviato e aggiorna lo scrolling della
  * finestra in modo che l'ultimo messaggio sia ben visibile.
  */
gint gui_add_message(GtkTextView *widget, gchar *msg)
{
 GtkTextBuffer *text = gtk_text_view_get_buffer(GTK_TEXT_VIEW(widget));
 GtkTextIter iter;
 gtk_text_buffer_get_end_iter(text,&iter);
 char *first = strstr(msg, "\n");
 int len = first-msg;
 char *tmp = calloc(len+1, 1);
 strncpy(tmp, msg, len);
 /**inserisce data e nick dell'utente nel buffer, viene colorato di blue per
      differenziarlo
   *dal messaggio vero e proprio.
   */
 gtk_text_buffer_insert_with_tags_by_name(text, &iter, tmp, -1, "blue_fg", NULL);
 msg+=len;
 gtk_text_buffer_get_end_iter(text,&iter);
 //!inserimento del messaggio
 gtk_text_buffer_insert(text,&iter,msg,-1);

 //! Scrolling della finestra
 GtkTextIter new_iter;
 text = gtk_text_view_get_buffer(GTK_TEXT_VIEW(widget));
 gtk_text_buffer_get_end_iter(text, &new_iter);
 GtkTextMark *mark = gtk_text_mark_new(NULL,FALSE);
 gtk_text_buffer_add_mark(text,mark,&new_iter);
 gtk_text_view_scroll_to_mark(GTK_TEXT_VIEW(widget), mark, 0.0, FALSE, 0 ,0);
```

```
 return (FALSE);
}

/**
  * Funzione chiamata alla selezione di una chat, chiama il controller per gestire
  * la connessione agli utenti della chat e l'invio della JOIN, inoltre lancia la
  * finestra relativa alla chat.
  */
void gui_open_chat_event (GtkTreeView *treeview, GtkTreePath *path, GtkTreeViewColumn  *
     col, gpointer userdata) {
 GtkTreeModel *model;
 GtkTreeIter    iter;


 model = gtk_tree_view_get_model(treeview);

 if (gtk_tree_model_get_iter(model, &iter, path))
 {
  gchar *name;

  gtk_tree_model_get(model, &iter, 0, &name, -1);
  chat *elem = data_get_chat(atoll(name));
  if(elem!=NULL) {

   controller_connect_users(g_hash_table_get_values(elem->users));

   gui_open_chatroom(elem->id);

   controller_join_flooding(elem->id);
  }
  g_free(name);
 }
}

/**
  * Evento che si scatena quando si effettua un doppio click su un utente della chat.
  * Permette di aprire una nuova finestra per una conversazione privata.
  */
gint gui_open_conversation_event(GtkTreeView *treeview, GtkTreePath *path,
     GtkTreeViewColumn  *col, gpointer userdata) {

 GtkTreeModel *model;
 GtkTreeIter    iter;

 model = gtk_tree_view_get_model(treeview);

 if (gtk_tree_model_get_iter(model, &iter, path))
 {
  gchar *user_id;
  gchar *name;
  gtk_tree_model_get(model, &iter, 0, &name, -1);
  gtk_tree_model_get(model, &iter, 1, &user_id, -1);

  if(user_id > 0 ) {
   logger(INFO, "[open_conversation] nick length %d\n", strlen(name));
   if(g_hash_table_lookup(pm_data_hashtable, (gconstpointer)user_id) != NULL) {
        logger(INFO, "[open_conversation] conversation already open\n");
        return (FALSE);
   }
   gui_open_pm(atoll(user_id),name);

  }
 }
 return FALSE;
```

```
}

/**
  * Evento che permette di aprire la finestra relativa all'about.
  */
gint gui_open_about_event(GtkWidget *widget, gpointer gdata)
{
 GtkWidget *window;
 GtkWidget *label1;

 window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
 g_signal_connect (window, "delete_event",
                                    G_CALLBACK (gui_close_window_event), NULL);

 gtk_container_set_border_width (GTK_CONTAINER (window), 25);

 label1 = gtk_label_new("\tTorTella Chat\nCreated by TorTella Team\n");
 gtk_container_add (GTK_CONTAINER (window), label1);
 gtk_widget_show(label1);
 gtk_widget_show(window);
 gtk_main();
 return 0;
}

/**
  * Setta lo stato dell'utente su Online e invoca il controller_change_status
  * in modo che quest'ultimo possa inviare un pacchetto di PING con il nuovo status
  */
gint gui_set_to_online_event(GtkWidget *widget, gpointer gdata)
{
 controller_change_status (ONLINE_ID);
 gui_change_status(servent_get_local()->id, ONLINE);
 return(FALSE);
}

/**
  * Setta lo stato dell'utente su Busy e invoca il controller_change_status
  * in modo che quest'ultimo possa inviare un pacchetto di PING con il nuovo status
  */
gint gui_set_to_busy_event(GtkWidget *widget, gpointer gdata)
{
 controller_change_status(BUSY_ID);
 gui_change_status(servent_get_local()->id, BUSY);
 return (FALSE);
}

/**
  * Setta lo stato dell'utente su Away e invoca il controller_change_status
  * in modo che quest'ultimo possa inviare un pacchetto di PING con il nuovo status
  */
gint gui_set_to_away_event(GtkWidget *widget, gpointer gdata)
{
 controller_change_status(AWAY_ID);
 gui_change_status(servent_get_local()->id, AWAY);
 return (FALSE);
}

/**
  * Evento che si scatena alla pressione del bottone search, effettua un clear della
  * lista delle chat e invoca il controller_search in modo che quest'ultimo possa
  * inviare il pacchetto di SEARCH.
  */
gint gui_search_chat_event(GtkWidget *widget, gpointer gdata)
{
```

```
 gui_clear_chat_list();
 controller_search(gtk_entry_get_text(GTK_ENTRY(bar_textfield)));
 return(FALSE);
}

/**
  * Evento che si scatena alla pressione del bottone create, fa' una semplice
  * chiamata alla controller_create.
  */
gint gui_create_chat_button(GtkWidget *widget, gpointer gdata) {
 return controller_create(gtk_entry_get_text(GTK_ENTRY(bar_textfield)));
}

/**
  * Prepara il corpo del messaggio da inviare a tutta la chat o a un sottoinsieme di
  * questa, dopo di che invoca il controller affinche' questo gestisca l'invio del
  * messaggio e infine aggiorna la text view locale col nuovo messaggio.
  */
gint gui_send_text_message_event(GtkWidget *widget, GdkEventKey *event, gpointer gdata)
{
 //!se e' stato premuto il tasto invio, si sta inviando un messaggio
 if(event->type == GDK_KEY_PRESS && event->keyval == GDK_Return) {
  GtkTextBuffer *buf = gtk_text_view_get_buffer(GTK_TEXT_VIEW(widget));
  GtkTextIter start ;
  gtk_text_buffer_get_start_iter(buf, &start);
  GtkTextIter end;
  gtk_text_buffer_get_end_iter(buf, &end);
  //!testo del messaggio contenuto nella text view
  char *msg = gtk_text_buffer_get_text(buf, &start, &end, TRUE);
  GtkTreeSelection *selection;

  u_int8 chat_id = atoll((char*)gdata);
  logger(INFO, "[send_text_message] chat id = %lld\n",chat_id);
  /** Invio del messaggio ad un sottoinsieme di utenti*/
  tree_model *chat_list_tmp = gui_get_tree_model(chat_id);

  //!elenco degli utenti selezionati
  selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(chat_list_tmp->tree_view));
  if(gtk_tree_selection_count_selected_rows(selection) > 0) {
   GList *subset = NULL;
   logger(INFO, "[send_text_message] send to subset\n");
   GtkTreeIter tmp_iter;
   gboolean valid = TRUE;
   char *id;
   //!si setta l'iteratore al primo record della lista
   if(gtk_tree_model_get_iter_first(GTK_TREE_MODEL(chat_list_tmp->user_model), &tmp_iter
        )==TRUE) {
        //!si scorre la lista degli utenti della chat e si confronta con gli utenti
             selezionati
        while(valid) {
         if(gtk_tree_selection_iter_is_selected(selection, &tmp_iter)) {
          //!Aggiungi alla lista di utenti a cui e' rivolto il messaggio
          gtk_tree_model_get(GTK_TREE_MODEL(chat_list_tmp->user_model), &tmp_iter, 1, &
               id, -1);
          logger(INFO, "[send_text_message] user id: %lld\n", atoll(id));
          subset = g_list_prepend(subset, (gpointer)(servent_get(atoll(id))));
         }
         valid = gtk_tree_model_iter_next(GTK_TREE_MODEL(chat_list_tmp->user_model), &
              tmp_iter);
        }
        logger(INFO, "[send_text_message] sending message to subset; list length %d\n",
             g_list_length(subset));
        //!invio del messaggio tramite il controller
```

```
        if(controller_send_subset_users (chat_id,strlen(msg),msg, subset) >= 0) {
         //!clear del buffer e preparazione del messaggio da aggiungere sulla gui
         gui_clear_buffer(GTK_TEXT_VIEW(widget));
         time_t actual_time = time(NULL);
         char *send_msg = prepare_msg(actual_time, servent_get_local()->nick, msg,
              strlen(msg));
         gui_add_msg_to_chat(chat_id, send_msg);
         gtk_tree_selection_unselect_all(selection);
        }
        return TRUE;
    }
   }
   /** Fine dell'invio ad un sottoinsieme di utenti */
   else {
    //!invio del messaggio tramite il controller
    if(controller_send_chat_users(chat_id, strlen(msg), msg)>=0) {
         //!clear del buffer e preparazione del messaggio da aggiungere sulla gui
         gui_clear_buffer(GTK_TEXT_VIEW(widget));
         time_t actual_time = time(NULL);
         char *send_msg = prepare_msg(actual_time, servent_get_local()->nick, msg, strlen
              (msg));
         gui_add_msg_to_chat(chat_id, send_msg);
    }
    return TRUE;
   }
  }
  return (FALSE);
}

/**
  * Come la send_text_message_event, ma rivolta all'invio di messaggi privati
  */
gint gui_send_pm_message_event(GtkWidget *widget, GdkEventKey *event, gpointer gdata)
{
 //!se e' stato premuto il tasto invio, si sta inviando un messaggio
 if(event->type == GDK_KEY_PRESS && event->keyval == GDK_Return) {

  u_int8 user_id = atoll((char*)gdata);
  GtkTextBuffer *buf = gtk_text_view_get_buffer(GTK_TEXT_VIEW(widget));
  GtkTextIter start ;
  gtk_text_buffer_get_start_iter(buf, &start);
  GtkTextIter end;
  gtk_text_buffer_get_end_iter(buf, &end);
  //!testo del messaggio contenuto nella text view
  char *msg = gtk_text_buffer_get_text(buf, &start, &end, TRUE);
  logger(INFO, "[send_pm_message]Msg: %s to %lld\n", msg, user_id);
  //!invio del messaggio tramite il controller
  if(controller_send_pm(strlen(msg), msg, user_id)>=0) {
   //!clear del buffer e preparazione del messaggio da aggiungere sulla gui
   gui_clear_buffer(GTK_TEXT_VIEW(widget));
   time_t actual_time = time(NULL);
   char *send_msg = prepare_msg(actual_time, servent_get_local()->nick, msg, strlen(msg)
        );
   gui_add_msg_pm(user_id, send_msg);
  }
  return TRUE;
 }
 return (FALSE);
}

/**
  * Aggiunge alla text view della chat il nuovo messaggio.
  */
int gui_add_msg_to_chat(u_int8 chat_id, char *msg) {
```

```
 tree_model *model_str = gui_get_tree_model(chat_id);
 if(model_str==NULL)
  return -1;
 gui_add_message(GTK_TEXT_VIEW(model_str->text_area), msg);

 return 0;
}

/**
  * Aggiunge alla text view della conversazione privata il nuovo messaggio.
  */
int gui_add_msg_pm(u_int8 sender_id, char *msg) {
 if(msg==NULL) {
  return -1;
 }

 pm_data *pm = (pm_data*)g_hash_table_lookup(pm_data_hashtable, (gconstpointer)to_string
      (sender_id));
 if(pm==NULL) {
  logger(INFO, "[add_msg_pm]PM window not ready\n");
  logger(INFO, "[add_msg_pm] strlen nick %d\n",strlen(servent_get(sender_id)->nick));
  //!apertura della nuova conversazione privata nel caso questa non sia gia' presente
  gui_open_pm(sender_id, servent_get(sender_id)->nick);
  pm = (pm_data*)g_hash_table_lookup(pm_data_hashtable, (gconstpointer)to_string(
       sender_id));
  if(pm==NULL) {
   logger(INFO, "[add_msg_pm]PM error\n");
   return -1;
  }
 }
 logger(INFO, "[add_msg_pm]Adding msg\n");
 gui_add_message(pm->text_area, msg);
 logger(INFO, "[add_msg_pm]Added msg\n");

 return 0;
}

/**
  * Controlla nell'hashtable la presenza o meno del tree_model corrispondente alla
  * chat.
  */
tree_model *gui_get_tree_model(u_int8 chat_id) {
 if(tree_model_hashtable == NULL) {      //!PROVA
  return NULL;
 }
 return (tree_model*)g_hash_table_lookup(tree_model_hashtable, to_string(chat_id));
}

/**
  * Crea il componente grafico che contiene la lista degli utenti connessi ad una chat.
  */
GtkWidget *gui_create_users_list(u_int8 index )
{

 GtkWidget *scrolled_window;
 GtkWidget *tree_view;
 GtkListStore *model = gtk_list_store_new(3, G_TYPE_STRING, G_TYPE_STRING, G_TYPE_STRING
      );
 GtkCellRenderer *cell;
 tree_model *model_str = calloc(1, sizeof(tree_model));
 model_str->user_model = model;

 //! crea una nuova scrolled window con lo scrolling abilitato solo se necessario
 scrolled_window = gtk_scrolled_window_new (NULL, NULL);
```

```
 gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                                                    GTK_POLICY_AUTOMATIC,
                                                                    GTK_POLICY_AUTOMATIC);

 //! Definizione della tree view e dei sotto componenti
 tree_view = gtk_tree_view_new ();
 gtk_scrolled_window_add_with_viewport (GTK_SCROLLED_WINDOW (scrolled_window),
                                                                                    tree_view
                                                                                        );
 gtk_tree_view_set_model (GTK_TREE_VIEW (tree_view), GTK_TREE_MODEL (model));
 gtk_widget_show (tree_view);

 cell = gtk_cell_renderer_text_new ();
 gtk_tree_view_insert_column_with_attributes (GTK_TREE_VIEW(tree_view), -1, "Nickname",
      cell, "text", 0, NULL);
 cell = gtk_cell_renderer_text_new ();
 gtk_tree_view_insert_column_with_attributes (GTK_TREE_VIEW(tree_view), -1, "ID", cell,
      "text", 1, NULL);
 cell = gtk_cell_renderer_text_new ();
 gtk_tree_view_insert_column_with_attributes (GTK_TREE_VIEW(tree_view), -1, "Status",
      cell, "text", 2, NULL);

 //!modalita' di selezione dei record della lista
 GtkTreeSelection *select;
 select = gtk_tree_view_get_selection(GTK_TREE_VIEW(tree_view));
 gtk_tree_selection_set_mode(select, GTK_SELECTION_MULTIPLE);

 //!evento che si scatena al doppio click su un record della lista
 g_signal_connect(G_OBJECT(tree_view),
                                    "row-activated",
                                    G_CALLBACK(gui_open_conversation_event),
                                    to_string(index));
 if(tree_model_hashtable == NULL) {
  logger(INFO, "[create_users_list] created user list\n");
  tree_model_hashtable = g_hash_table_new(g_str_hash, g_str_equal);
 }

 logger(INFO, "[create_users_list]chat ID: %lld\n", index);
 model_str->tree_view = GTK_TREE_VIEW(tree_view);
 g_hash_table_insert(tree_model_hashtable, (gpointer)to_string(index), (gpointer)
      model_str);

 return scrolled_window;
}

/**
  * Crea il componente grafico che contiene la lista delle chat trovate.
  */
GtkWidget *gui_create_chat_list(u_int8 index )
{

 GtkWidget *scrolled_window;
 GtkWidget *tree_view;
 GtkCellRenderer *cell;

 //! crea una nuova scrolled window con lo scrolling abilitato solo se necessario
 scrolled_window = gtk_scrolled_window_new (NULL, NULL);
 gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                                                    GTK_POLICY_AUTOMATIC,
                                                                    GTK_POLICY_AUTOMATIC);

 //! Definizione della tree view e dei sotto componenti
 chat_model = gtk_list_store_new (2, G_TYPE_STRING, G_TYPE_STRING);
 tree_view = gtk_tree_view_new ();
```

```
 gtk_scrolled_window_add_with_viewport (GTK_SCROLLED_WINDOW (scrolled_window),
                                                                    tree_view
                                                                      );
 gtk_tree_view_set_model (GTK_TREE_VIEW (tree_view), GTK_TREE_MODEL (chat_model));
 gtk_widget_show (tree_view);

 cell = gtk_cell_renderer_text_new ();
 gtk_tree_view_insert_column_with_attributes (GTK_TREE_VIEW(tree_view), -1, "ID", cell,
      "text", 0, NULL);
 cell = gtk_cell_renderer_text_new ();
 gtk_tree_view_insert_column_with_attributes (GTK_TREE_VIEW(tree_view), -1, "Title",
      cell, "text", 1, NULL);

 //!evento che si scatena al doppio click su un record della lista
 g_signal_connect(tree_view, "row-activated", (GCallback) gui_open_chat_event, NULL);

 pm_data_hashtable = g_hash_table_new(g_str_hash, g_str_equal);

 return scrolled_window;
}

/**
  * Crea la barra dei menu situata su ogni finestra. (File -- Status -- Help).
  */
GtkWidget *gui_create_menu() {
 GtkWidget *menubar;
 GtkWidget *menuFile;
 GtkWidget *menuEdit;
 GtkWidget *menuHelp;
 GtkWidget *menuitem;
 GtkWidget *menu;
 /**-- Crea la menu bar --*/
 menubar = gtk_menu_bar_new();
 /**----------------- Crea File menu items --------------------*/

 menuFile = gtk_menu_item_new_with_label ("File");
 gtk_menu_bar_append (GTK_MENU_BAR(menubar), menuFile);
 gtk_widget_show(menuFile);

 /**-- Crea File submenu --*/
 menu = gtk_menu_new();
 gtk_menu_item_set_submenu(GTK_MENU_ITEM(menuFile), menu);

 /**-- Crea un NEW menu item da collocare nel File submenu --*/
 menuitem = gtk_menu_item_new_with_label ("New");
 gtk_menu_append(GTK_MENU(menu), menuitem);
 gtk_widget_show (menuitem);

 /**-- Crea un OPEN menu item da collocare nel File submenu --*/
 menuitem = gtk_menu_item_new_with_label ("Open");
 gtk_menu_append(GTK_MENU(menu), menuitem);
 gtk_widget_show (menuitem);

 /**-- Crea un Exit menu item da collocare nel File submenu --*/
 menuitem = gtk_menu_item_new_with_label ("Exit");
 gtk_menu_append(GTK_MENU(menu), menuitem);
 gtk_signal_connect(GTK_OBJECT (menuitem), "activate", GTK_SIGNAL_FUNC (gui_close_event)
      , NULL);
 gtk_widget_show (menuitem);
 /**----------------- Fine dichiarazione File menu ------------------*/

 /**----------------- Crea Edit menu items ---------------------*/

 menuEdit = gtk_menu_item_new_with_label ("Stato");
```

```
 gtk_menu_bar_append (GTK_MENU_BAR(menubar), menuEdit);
 gtk_widget_show(menuEdit);

 /**-- Crea submenu --*/
 menu = gtk_menu_new();
 gtk_menu_item_set_submenu(GTK_MENU_ITEM(menuEdit), menu);

 /**-- Crea Online menu item da collocare in Stato submenu --*/
 menuitem = gtk_menu_item_new_with_label ("Online");
 gtk_menu_append(GTK_MENU(menu), menuitem);
 gtk_signal_connect(GTK_OBJECT (menuitem), "activate", GTK_SIGNAL_FUNC (
      gui_set_to_online_event), NULL);
 gtk_widget_show (menuitem);

 /**-- Crea Busy menu item da collocare in Stato submenu --*/
 menuitem = gtk_menu_item_new_with_label ("Busy");
 gtk_menu_append(GTK_MENU(menu), menuitem);
 gtk_signal_connect(GTK_OBJECT (menuitem), "activate", GTK_SIGNAL_FUNC (
      gui_set_to_busy_event), NULL);
 gtk_widget_show (menuitem);

 /**-- Crea Away menu item da collocare in Stato submenu --*/
 menuitem = gtk_menu_item_new_with_label ("Away");
 gtk_menu_append(GTK_MENU(menu), menuitem);
 gtk_signal_connect(GTK_OBJECT (menuitem), "activate", GTK_SIGNAL_FUNC (
      gui_set_to_away_event), NULL);
 gtk_widget_show (menuitem);
 /**----------------- Fine dichiarazione Edit menu -----------------*/

 /**----------------- Start Help menu -----------------*/
 menuHelp = gtk_menu_item_new_with_label ("Help");
 gtk_menu_bar_append (GTK_MENU_BAR(menubar), menuHelp);
 gtk_widget_show(menuHelp);

 /**-- Crea Help submenu --*/
 menu = gtk_menu_new();
 gtk_menu_item_set_submenu(GTK_MENU_ITEM(menuHelp), menu);

 /**-- Crea About menu item da collocare in Help submenu --*/
 menuitem = gtk_menu_item_new_with_label ("About");
 gtk_menu_append(GTK_MENU(menu), menuitem);
 gtk_signal_connect(GTK_OBJECT (menuitem), "activate", GTK_SIGNAL_FUNC (
      gui_open_about_event), NULL);
 gtk_widget_show (menuitem);
 /**----------------- Fine Help menu -----------------*/
 return menubar;
}

/**
  * Crea le varie text view (di scrittura e lettura) per la chat e per il pm,
  * impostando i valori in modo opportuno.
  */
GtkWidget *gui_create_text(u_int8 chat_id, int type, int msg_type)
{
 logger(INFO, "[create_text]chat ID: %lld\n", chat_id);
 GtkWidget *scrolled_window;
 GtkWidget *view = gtk_text_view_new ();
 GtkTextBuffer *buffer;

 buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (view));
 //!creazione della scrolled window
 scrolled_window = gtk_scrolled_window_new (NULL, NULL);
 gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                                    GTK_POLICY_AUTOMATIC,
```

```
                                                                    GTK_POLICY_AUTOMATIC);
 gtk_text_view_set_wrap_mode(GTK_TEXT_VIEW(view), TRUE);
 gtk_container_add (GTK_CONTAINER (scrolled_window), view);

 /** da qui parte la differenziazione tra le text view relative alla chat e al pm
   * con i relativi eventi, che si scatenano alla pressione di un tasto
   */
 if(type == BOTTOM) {
  gtk_text_view_set_editable(GTK_TEXT_VIEW(view),TRUE);
  logger(INFO, "[create_text]signal connect\n");
  if(msg_type == PM) {
   g_signal_connect(GTK_OBJECT(view),"key_press_event", G_CALLBACK(
        gui_send_pm_message_event), (gpointer)to_string(chat_id));
  }
  else if(msg_type == CHAT) {
   g_signal_connect(GTK_OBJECT(view),"key_press_event", G_CALLBACK(
        gui_send_text_message_event), (gpointer)to_string(chat_id));
  }
  gtk_text_view_set_editable(GTK_TEXT_VIEW(view),TRUE);
  logger(INFO, "[create_text]signal connected\n");
 }
 else if (type == TOP){
  if(msg_type == CHAT) {
   gtk_text_view_set_editable(GTK_TEXT_VIEW(view),FALSE);
   gtk_text_buffer_create_tag(buffer, "blue_fg", "foreground", "blue", NULL);
   tree_model *model_str = gui_get_tree_model(chat_id);
   if(model_str!=NULL) {
        model_str->text_area = GTK_TEXT_VIEW(view);
   }
   else {
        return NULL;
   }
  }
  else if(msg_type == PM) {
   pm_data *pm = (pm_data*)g_hash_table_lookup(pm_data_hashtable, (gconstpointer)
        to_string(chat_id));
   if(pm==NULL) {
        pm = calloc(1, sizeof(pm_data));
        pm->text_area = GTK_TEXT_VIEW(view);
        gtk_text_view_set_editable(GTK_TEXT_VIEW(pm->text_area),FALSE);
        gtk_text_buffer_create_tag(buffer, "blue_fg", "foreground", "blue", NULL);
        logger(INFO, "[create_text]Allocating pm window\n");
        g_hash_table_insert(pm_data_hashtable, (gpointer)to_string(chat_id), (gpointer)
             pm);
        logger(INFO, "[create_text]Allocated pm window\n");
   }
  }
 }
 gtk_widget_show_all (scrolled_window);

 return scrolled_window;
}

/**
  * Crea la finestra relativa ad una singola chat e assegna ai componenti gli
  * eventi appropriati
  */
int gui_open_chatroom(u_int8 chat_id) {

 GtkWidget *window;
 GtkWidget *menu;
 GtkWidget *vbox;
 GtkWidget *hbox;
 GtkWidget *vbox2;
```

```
 GtkWidget *handlebox;
 GtkWidget *text;
 GtkWidget *list;
 GtkWidget *chat;

 //! Crea i vari componenti
 window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

 menu = gui_create_menu();
 list = gui_create_users_list (chat_id);

 text = gui_create_text(chat_id, TOP, CHAT);
 chat = gui_create_text(chat_id, BOTTOM, CHAT);

 vbox = gtk_vbox_new(FALSE, 5);

 hbox = gtk_hbox_new(FALSE, 5);
 vbox2 = gtk_vbox_new(FALSE, 5);

 handlebox = gtk_handle_box_new();

 logger(INFO,"[open_chatroom_gui] chat_id %lld\n", chat_id);

 //!aggiunge i vari componenti ai box
 gtk_container_add(GTK_CONTAINER(handlebox), menu);
 gtk_box_pack_start(GTK_BOX(vbox), handlebox, FALSE, TRUE, 0);
 gtk_container_add(GTK_CONTAINER(vbox),hbox);
 gtk_container_add(GTK_CONTAINER(hbox),vbox2);
 gtk_container_add (GTK_CONTAINER (hbox), list);
 gtk_container_add(GTK_CONTAINER(vbox2),text);
 gtk_box_pack_end(GTK_BOX(vbox2), chat, FALSE, TRUE, 0);
 //!aggiunge il vbox alla finestra principale
 gtk_container_add(GTK_CONTAINER(window), vbox);

 //!setta le caratteristiche della finestra
 gtk_container_border_width (GTK_CONTAINER (window), 0);
 gtk_window_set_default_size (GTK_WINDOW(window), 640, 400);
 gtk_window_set_title(GTK_WINDOW (window), "Chat Room");
 logger(INFO, "[open_chatroom_gui]chat_id created: %lld\n", chat_id);

 /**-- Display the widgets --*/
 gtk_widget_show(handlebox);
 gtk_widget_show(hbox);
 gtk_widget_show(vbox);
 gtk_widget_show(vbox2);
 gtk_widget_show(text);
 gtk_widget_show(list);
 gtk_widget_show(chat);
 gtk_widget_show(menu);
 gtk_widget_show(window);


 logger(INFO, "[open gui] to string %s\n", to_string(chat_id));
 //!evento che si scatena alla chiusura della finestra
 g_signal_connect(GTK_OBJECT(window), "destroy", G_CALLBACK(gui_leave_chat_event), (
      gpointer)to_string(chat_id));
 return 0;
}

/**
  * Crea la finestra per una conversazione privata
  */
int gui_open_pm(u_int8 user_id, gchar *nickname) {
```

```
 GtkWidget *window;
 GtkWidget *menu;
 GtkWidget *vbox;
 GtkWidget *vbox2;
 GtkWidget *handlebox;
 GtkWidget *text;
 GtkWidget *chat;

 /**-- Crea la nuova finestra e i vari componenti --*/
 window = gtk_window_new(GTK_WINDOW_TOPLEVEL);


 vbox = gtk_vbox_new(FALSE, 5);
 vbox2 = gtk_vbox_new(FALSE, 5);
 text = gui_create_text(user_id, TOP, PM);
 chat = gui_create_text(user_id, BOTTOM, PM);
 handlebox = gtk_handle_box_new();
 menu = gui_create_menu();

 /**-- Aggiunge i componenti ai box --*/
 gtk_container_add(GTK_CONTAINER(handlebox), menu);

 gtk_box_pack_start(GTK_BOX(vbox), handlebox, FALSE, TRUE, 0);
 gtk_container_add(GTK_CONTAINER(vbox),vbox2);
 gtk_container_add(GTK_CONTAINER(vbox2),text);
 gtk_box_pack_end(GTK_BOX(vbox2), chat, FALSE, TRUE, 0);
 gtk_container_add(GTK_CONTAINER(window), vbox);

 /**-- Setta le caratteristiche della finestra --*/
 gtk_container_border_width (GTK_CONTAINER (window), 0);
 gtk_window_set_default_size (GTK_WINDOW(window), 640, 400);
 gtk_window_set_title(GTK_WINDOW (window), nickname);

 gtk_widget_show(handlebox);
 gtk_widget_show(vbox);
 gtk_widget_show(vbox2);
 gtk_widget_show(text);
 gtk_widget_show (chat);
 gtk_widget_show(menu);
 gtk_widget_show(window);

 /**-- evento che si scatena alla chiusura della finestra --*/
 g_signal_connect(GTK_OBJECT(window), "destroy", G_CALLBACK(gui_leave_pm_event), (
      gpointer)to_string(user_id));

 pm_data *pm = (pm_data*)g_hash_table_lookup(pm_data_hashtable, (gconstpointer)to_string
      (user_id));
 if(pm!=NULL) {
  pm->window = GTK_WIDGET(window);
 }
 return 0;
}

/**
  * Crea il componente grafico presente nella finestra iniziale in cui si puo'
  * effettuare la creazione o la ricerca di una chat.
  */
GtkWidget *gui_create_searchbar(void) {
 GtkWidget *bar_container = gtk_hbox_new(FALSE, 5);

 bar_textfield = gtk_entry_new();
 GtkWidget *bar_button = gtk_button_new();
 GtkWidget *bar_create_button = gtk_button_new();
```

```
 gtk_entry_set_width_chars(GTK_ENTRY(bar_textfield), (gint)40);
 gtk_button_set_label(GTK_BUTTON(bar_button), (const gchar*)"Search");
 g_signal_connect (G_OBJECT (bar_button), "clicked",
                                    G_CALLBACK (gui_search_chat_event), NULL);

 gtk_button_set_label(GTK_BUTTON(bar_create_button), (const gchar*)"Create");
 g_signal_connect(G_OBJECT(bar_create_button), "clicked", G_CALLBACK(
      gui_create_chat_button), NULL);

 gtk_container_add(GTK_CONTAINER(bar_container), bar_textfield);
 gtk_box_pack_start(GTK_BOX(bar_container), bar_button, FALSE, TRUE, 5);
 gtk_box_pack_start(GTK_BOX(bar_container), bar_create_button, FALSE, TRUE, 5);

 gtk_widget_show(bar_create_button);
 gtk_widget_show(bar_textfield);
 gtk_widget_show(bar_button);

 return bar_container;
}

/**
  * Controlla nell'hashtable la presenza o meno della struct pm_data relativa
  * all'id dell'utente.
  */
pm_data *gui_pm_data_get(u_int8 id) {
 if(pm_data_hashtable!=NULL)
  return g_hash_table_lookup(pm_data_hashtable, (gconstpointer)to_string(id));
 else
  return NULL;
}

/**
  * Recupera dal tree_model associato alla gui della chat la lista degli id degli
  * utenti connessi.
  */
GList *gui_get_chat_users(u_int8 chat_id) {
 tree_model *mod = gui_get_tree_model(chat_id);
 GList *id_list = NULL;

 if(mod == NULL) {
  logger(INFO, "[gui_get_chat_users] users list not ready\n");
  return id_list;
 }
 GtkTreeIter iter;
 gchar *id;
 gboolean valid = TRUE;
 if(gtk_tree_model_get_iter_first(GTK_TREE_MODEL(mod->user_model), &iter)==TRUE) {

  while(valid) {
   gtk_tree_model_get(GTK_TREE_MODEL(mod->user_model), &iter, 1, &id, -1);
   logger(INFO, "[gui_get_chat_users]user id: %s\n", id);
   id_list = g_list_append(id_list, (gpointer)id);


   valid = gtk_tree_model_iter_next(GTK_TREE_MODEL(mod->user_model), &iter);
  }
 }

 return id_list;
}
```

1.22 routemanager.h

```
#ifndef ROUTEMANAGER_H
#define ROUTEMANAGER_H

#include "common.h"
#include "packetmanager.h"
#include "utils.h"
#include "datamanager.h"
#include "init.h"
#include "routemanager.h"
#include <pthread.h>
#include <signal.h>
#include <unistd.h>
#include <glib.h>

struct route_entry {
 u_int8 sender_id;
 u_int8 recv_id;
 u_int4 counter;
};
typedef struct route_entry route_entry;

static GHashTable *route_hashtable = NULL;

//!----------------------------------

/**
  * Aggiunge una regola di routing alla tabella di routing. Se la regola e' gia'
      presente
  * incrementa il contatore associato alla regola.
  */
int add_route_entry(u_int8 packet_id, u_int8 sender_id, u_int8 recv_id);

/**
  * Decrementa il contatore della regola. Elimina la regola se il contatore e' 0
  */
int del_route_entry(u_int8 id);

/**
  * Ritorna una la regola di routing associata all'id del pacchetto
  */
route_entry *get_route_entry(u_int8 packet_id);

/**
  * Ritorna l'id destinazione associato all'id del pacchetto
  */
u_int8 get_iddest_route_entry(u_int8 id);


#endif //!ROUTEMANAGER_H
```

1.23 routemanager.c

```
#include "routemanager.h"

/**
 * Aggiunge una regola di routing alla tabella di routing. Se la regola e' gia' presente
 * incrementa il contatore associato alla regola.
 */
int add_route_entry(u_int8 packet_id, u_int8 sender_id, u_int8 recv_id) {
 if(route_hashtable==NULL) { //!Alloca l'hashtable solo la prima volta
  route_hashtable = g_hash_table_new(g_str_hash, g_str_equal);
 }

 route_entry *entry;
 char *key = to_string(packet_id);
 if((entry=get_route_entry(packet_id))!=NULL) { //!Incrementa il contatore se la regola
      e' gia' presente
  entry->counter++;
 }
 else { //!Aggiunge una nuova regola
  entry = (route_entry*)calloc(sizeof(route_entry), 1);
  entry->sender_id = sender_id;
  entry->recv_id = recv_id;
  entry->counter = 0;
  g_hash_table_insert(route_hashtable, (gpointer)key, (gpointer)entry);
 }

 return 1;
}

/**
  * Decrementa il contatore della regola. Elimina la regola se il contatore e' 0
  */
int del_route_entry(u_int8 id) {
 if(route_hashtable==NULL) {
  route_hashtable = g_hash_table_new(g_str_hash, g_str_equal);
 }

 route_entry *entry;
 char *key = to_string(id);
 if((entry=get_route_entry(id))!=NULL) { //!Decrementa la regola di routing
  entry->counter--;
  if(entry->counter==0)
   g_hash_table_remove(route_hashtable, (gconstpointer)key);
  return 0;
 }

 return -1;
}

/**
  * Ritorna una la regola di routing associata all'id del pacchetto
  */
route_entry *get_route_entry(u_int8 packet_id) {
 if(route_hashtable==NULL) {
  route_hashtable = g_hash_table_new(g_str_hash, g_str_equal);
 }

 char *key = to_string(packet_id);
 route_entry *entry = (route_entry*)g_hash_table_lookup(route_hashtable, (gconstpointer)
      key);
 return entry;
```

```
}

/**
  * Ritorna l'id destinazione associato all'id del pacchetto
  */
u_int8 get_iddest_route_entry(u_int8 id) {

 return ((route_entry*)get_route_entry(id))->sender_id;
}
```

1.24 datamanager.h

```
#ifndef SUPERNODEDATA_H
#define SUPERNODEDATA_H

#include "common.h"
#include "utils.h"
#include <glib.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <pthread.h>
#include <dirent.h>
#include "logger.h"

#define MODE_TRUNC 0x900
#define MODE_APPEND 0x901

/** utente connesso ad una chat */
struct chatclient {
 u_int8 id;
 char *nick;
 char *ip;
 u_int4 port;
};
typedef struct chatclient chatclient;

/**
  * struttura della chat, contiene l'id, il nome della chat, una hashtable degli
  * utenti connessi e un mutex per gestire gli accessi concorrenti alla chat.
  */
struct chat {
 u_int8 id;
 char *title;
 GHashTable *users;
 pthread_mutex_t mutex;
};
typedef struct chat chat;

static GHashTable *chatclient_hashtable;
static GHashTable *chat_hashtable;

/**
  * Scrive la struttura dati 'chat' in un file, nel seguente modo:
  * chat_id;chat_title;
  * user_id;user_nick;user_ip;user_port;
  * ...
  *
  * In modalita' TRUNC crea ogni volta un nuovo file, mentre
  * nella modalita' APPEND modifica il file esistente.
  * Non utilizzata.
  */

int write_to_file(const char *filename, chat *chat_str, u_int4 mode);

/**
  * Salva tutte le chat su un file in modalita' TRUNC. Chiama la funzione write_to_file
  * per ogni chat contenuta nella hashtable.
  * Non utilizzata.
  */
```

```
int write_all(u_int4 mode);

/**
  * Legge le informazioni delle chat e degli utenti dal file specificato.
  * Aggiunge i dati sulla chat alla hashtable relativa, inoltre i dati degli utenti
  * alla hashtable relativa.
  * Non utilizzata.
  */
int read_from_file(const char *filename);

/**
  * Legge tutte le chat contenute nella directory specificata nel file di configurazione
      .
  * Non utilizzata.
  */
int read_all(void);

/**
  * Inserisce la chat rappresentata dai parametri id e titolo nella hashtable
  * relativa alle chat.
  */
int data_add_chat(u_int8 id, const char *title);

/**
  * Aggiunge la lista di chat nella hashtable relativa alle chat. Per ognuna di
  * queste recupera la lista degli utenti connessi e li inserisce nella hashtable
  * relativa ai chatclient.
  */
int data_add_all_to_chat(GList *chats);

/**
  * Rimuove la chat con l'id specificato dalla hashtable relativa alle chat.
  */
int data_del_chat(u_int8 id);

/**
  * Crea un nuovo chatclient settandone i campi con i valori dei parametri e lo
  * inserisce nella hashtable contenente tutti i chatclient
  */
int data_add_user(u_int8 id, const char *nick, const char *ip, u_int4 port);

/**
  * Aggiunge un utente presente nella hashtable dei client alla hashtable di utenti
  * della chat poiche' non ancora presente all'interno di quest'ultima.
  */
int data_add_existing_user_to_chat(u_int8 chat_id, u_int8 id);

/**
  * Aggiunge una lista di utenti alla chat specificata da chat_id. Per ogni
  * elemento della lista viene invocata la funzione data_add_user_to_chat.
  */
int data_add_users_to_chat(u_int8 chat_id, GList *users);

/**
  * Crea un nuovo chatclient settando i campi con i valori dei parametri e lo aggiunge
  * alla hashtable degli utenti connessi alla chat con id pari a chat_id.
  */
int data_add_user_to_chat(u_int8 chat_id, u_int8 id, const char *nick, const char *ip,
    u_int4 port);

/**
  * Rimuove un utente dalla hashtable dei chatclient.
  */
int data_del_user(u_int8 id);
```

```
/**
  * Rimuove l'utente con lo specifico id dalla hashtable dei chatclient e dalla
  * hashtable degli utenti connessi alla chat con id pari a chat_id.
  */
int data_del_user_from_chat(u_int8 chat_id, u_int8 id);

/**
  * Funzione wrapper non piu' utilizzata. Invoca la funzione data_search_chat
  */
chat *data_search_chat_local(const char *title);

/**
  * Cerca nella hashtable chat_table l'occorrenza della chat title
  * Ritorna la struttura dati della chat
  */
chat *data_search_chat(const char *title);

/**
  * Cerca nella hashtable chat_table tutte le chat che hanno come titolo *title*
  * Ritorna le chat in una slist
  */
GList *data_search_all_chat(const char *title);

/**
  * Funzione wrapper non piu' utilizzata.
  */
GList *data_search_all_local_chat(const char *title);

/**
  * Cerca un chatclient nella hashtable dei chatclient. Non piu' utilizzata.
  */
chatclient *data_search_chatclient(const char *nick);

/**
  * Converte la lista di chat in una stringa del tipo:
  * 111;test;
  * 22;pippo;127.0.0.1;2110;
  * 33;pluto;127.0.0.1;2111;
  */
char *data_chatlist_to_char(GList *chat_list, int *len);

/**
  * Converte una stringa in una lista di chat con i relativi utenti
  * 111;test;
  * 22;pippo;127.0.0.1;2110;
  * 33;pluto;127.0.0.1;2111;
  * |222;test;
  * 333;si.......
  */
GList *data_char_to_chatlist(const char *buffer,int len);

/**
  * Converte la lista di utenti in una stringa del tipo:
  * 22;pippo;127.0.0.1;2110;
  * 33;pluto;127.0.0.1;2111;
  */
char *data_userlist_to_char(GList *user_list, int *len);

/**
  * Converte una stringa in una lista di utenti.
  */
GList *data_char_to_userlist(const char *buffer,int len);
```

```
/**
  * Ritorna una lista di tutti i client della chat specificata.
  */
GList *data_get_chatclient_from_chat(u_int8 id);

/**
  * Ritorna la chat con lo specifico chat_id.
  */
chat *data_get_chat(u_int8 chat_id);

/**
  * Ritorna il chatclient con lo specifico id.
  */
chatclient *data_get_chatclient(u_int8 id);

/**
  * Rimuove un utente da tutte le hashtable delle chat in cui e' presente e infine
  * lo rimuove dalla hashtable dei chatclient.
  */
int data_destroy_user(u_int8 id);

#endif //!SUPERNODEDATA_H
```

1.25 datamanager.c

```
#include <string.h>
#include "datamanager.h"

/**
 * Scrive la struttura dati 'chat' in un file, nel seguente modo:
 * chat_id;chat_title;
 * user_id;user_nick;user_ip;user_port;
 * ...
 *
 * In modalita' TRUNC crea ogni volta un nuovo file, mentre
 * nella modalita' APPEND modifica il file esistente.
 * Non utilizzata.
 */
int write_to_file(const char *filename, chat *chat_str, u_int4 mode) {
 if(chat_str==NULL || filename==NULL || strcmp(filename, "")==0)
  return -1;

 int fd;
 if(mode==MODE_TRUNC) {

  pthread_mutex_lock(&chat_str->mutex);

  if((fd=open(filename, O_TRUNC|O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH))<0)
   return 0;

  GList *listclient;
  chatclient *chatclient_str;
  char buffer[2048];
  char bufferclient[1024];
  int j;

  sprintf(buffer, "%lld;%s;\n", chat_str->id, chat_str->title);

  listclient = g_hash_table_get_values(chat_str->users);
  for(j=0; j<g_list_length(listclient); j++) {
   chatclient_str = (chatclient*)g_list_nth_data(listclient, j);
   sprintf(bufferclient, "%lld;%s;%s;%d;\n", chatclient_str->id, chatclient_str->nick,
        chatclient_str->ip, chatclient_str->port);

   strcat(buffer, bufferclient);
  }

  int len=0;
  if((len=write(fd, buffer, strlen(buffer)))<0)
   return -1;

  close(fd);

  pthread_mutex_unlock(&chat_str->mutex);

  return len;
 }

 return 0;
}

/**
  * Salva tutte le chat su un file in modalita' TRUNC. Chiama la funzione write_to_file
  * per ogni chat contenuta nella hashtable.
  * Non utilizzata.
  */
```

```
int write_all(u_int4 mode) {

 if(mode==MODE_TRUNC) {
  GList *list = g_hash_table_get_values(chat_hashtable);
  printf("[write_all]size list: %d\n", g_list_length(list));
  chat *chat_str;
  int i;
  for(i=0; i<g_list_length(list); i++) {
   chat_str = (chat*)g_list_nth_data(list, i);
   logger(SYS_INFO, "[write_all]Writing file: %lld\n", chat_str->id);
   char *path = calloc(strlen(to_string(chat_str->id))+strlen(conf_get_datadir())+2, 1);
   strcpy(path, conf_get_datadir());
   strcat(path, "/");
   strcat(path, to_string(chat_str->id));
   write_to_file(path, chat_str, mode);
  }
 }

 return 0;
}

/**
  * Legge le informazioni delle chat e degli utenti dal file specificato.
  * Aggiunge i dati sulla chat alla hashtable relativa, inoltre i dati degli utenti
  * alla hashtable relativa.
  * Non utilizzata.
  */
int read_from_file(const char *filename) {
 char *saveptr;
 if(filename==NULL || strcmp(filename, "")==0) {
  printf("[read_from_file]Errore filename\n");
  return -1;
 }

 int fd;
 if((fd=open(filename, O_RDONLY|O_EXCL))<0) {
  printf("[read_from_file]Errore apertura file\n");
  return -2;
 }

 char ch;
 char line[256];
 char *buf;
 int line_count=0;
 int index=0;

 char *chat_id=NULL;
 char *title;
 char *id;
 char *nick;
 char *ip;
 char *port;
 while(read(fd, &ch, 1)>0) {
  if(ch=='\n') {
   if(line_count==0) {
        buf = strdup(line);
        chat_id = strtok_r(buf, ";",&saveptr);
        title = strtok_r(NULL, ";",&saveptr);
        data_add_chat(strtoull(chat_id, NULL, 10), strdup(title));
        memset(line, 0, 256);;
        index=0;
   }
   else if(line_count>0) {
        buf = strdup(line);
```

```
        id = strtok_r(buf, ";",&saveptr);
        nick = strtok_r(NULL, ";",&saveptr);
        ip = strtok_r(NULL, ";",&saveptr);
        port = strtok_r(NULL, ";",&saveptr);
        data_add_user_to_chat(strtoull(chat_id, NULL, 10), strtoull(id, NULL, 10),
             strdup(nick), strdup(ip), strtod(port, NULL));
        memset(line, 0, 256);
        index=0;
    }
    line_count++;
   }
   else {
    line[index++]=ch;
   }
  }

  return 0;
}

/**
  * Legge tutte le chat contenute nella directory specificata nel file di configurazione
      .
  * Non utilizzata.
  */
int read_all(void) {
 DIR *dir=opendir(conf_get_datadir());
 struct dirent *ent;
 char buf[100];
 GList *dir_list = NULL;
 while (0!=(ent=readdir(dir))) {
  printf("[read_all]Opening %s\n",ent->d_name);
  if(strcmp(ent->d_name, ".")!=0 && strcmp(ent->d_name, "..")!=0 && strcmp(ent->d_name,
       ".svn")!=0 && strstr(ent->d_name, "init_data")==NULL) {
   sprintf(buf, "%s/%s", conf_get_datadir(), ent->d_name);
   dir_list = g_list_append(dir_list, (gpointer)strdup(buf));
  }
 }
 closedir(dir);

 int i;
 for(i=0; i<g_list_length(dir_list); i++) {
  read_from_file((char*)g_list_nth_data(dir_list, i));
 }
 return 1;
}

/**
  * Inserisce la chat rappresentata dai parametri id e titolo nella hashtable
  * relativa alle chat.
  */
int data_add_chat(u_int8 id, const char *title) {
 if(chat_hashtable==NULL) {
  chat_hashtable = g_hash_table_new(g_str_hash, g_str_equal);
 }
 if(g_hash_table_lookup(chat_hashtable, (gpointer)to_string(id))!=NULL) {
  return -1;
 }
 //!creazione della nuova chat
 chat *chat_str = (chat*)calloc(1, sizeof(chat));
 chat_str->id = id;
 chat_str->title = (char*)strdup(title);
 chat_str->users = g_hash_table_new(g_str_hash, g_str_equal);
 pthread_mutex_init(&chat_str->mutex, NULL);
```

```
 logger(SYS_INFO, "[add_chat]Table created\n");

 g_hash_table_insert(chat_hashtable, (gpointer)to_string(id), (gpointer)chat_str); //!
      Aggiunta la chat alla hashtable

 return 0;
}

/**
  * Aggiunge la lista di chat nella hashtable relativa alle chat. Per ognuna di
  * queste recupera la lista degli utenti connessi e li inserisce nella hashtable
  * relativa ai chatclient.
  */
int data_add_all_to_chat(GList *chats) {
 if(chats==NULL) {
  return -1;
 }

 int i=0, j=0;
 chat *elem;
 chatclient *user;
 GList *user_list;
 for(; i<g_list_length(chats); i++) {
  elem = (chat*)g_list_nth_data(chats, i);
  //!aggiunta della chat alla hashtable chat_hashtable
  data_add_chat(elem->id, elem->title);
  if(elem->users!=NULL) {
   logger(SYS_INFO, "[add_all_to_chat]User list not empty\n");
   user_list = g_hash_table_get_values(elem->users);
   for(j=0; j<g_list_length(user_list); j++) {
        user = (chatclient*)g_list_nth_data(user_list, j);
        //!aggiunta degli utenti alla chat
        data_add_user_to_chat(elem->id, user->id, user->nick, user->ip, user->port);
   }
  }
 }
 return 0;
}

/**
  * Rimuove la chat con l'id specificato dalla hashtable relativa alle chat.
  */
int data_del_chat(u_int8 id) {
 if(chat_hashtable==NULL)
  return -1;

 g_hash_table_remove(chat_hashtable, (gconstpointer)to_string(id));

 return 0;
}

/**
  * Aggiunge un utente presente nella hashtable dei client alla hashtable di utenti
  * della chat poiche' non ancora presente all'interno di quest'ultima.
  */
int data_add_existing_user_to_chat(u_int8 chat_id, u_int8 id) {
 if(chat_hashtable==NULL || chatclient_hashtable==NULL)
  return -1;

 chat *chat_elem = (chat*)g_hash_table_lookup(chat_hashtable, (gconstpointer)to_string(
      chat_id));
 if(chat_elem!=NULL) {
  chatclient *chatclient_elem = (chatclient*)g_hash_table_lookup(chatclient_hashtable,
       to_string(id));
```

```
  if(chatclient_elem!=NULL) {
   logger(SYS_INFO, "[data_add_existing_user_to_chat] adding user nick: %s\n",
        chatclient_elem->nick);
   //!inserimento del chatclient nella hashtable degli utenti della chat chat_elem.
   g_hash_table_insert(chat_elem->users, (gpointer)to_string(chatclient_elem->id), (
        gpointer)chatclient_elem);
   return 1;
  }
 }
 return 0;
}

/**
  * Aggiunge una lista di utenti alla chat specificata da chat_id. Per ogni
  * elemento della lista viene invocata la funzione data_add_user_to_chat.
  */
int data_add_users_to_chat(u_int8 chat_id, GList *users) {
 if(users==NULL) {
  logger(SYS_INFO, "[add_users_to_chat]Users NULL\n");
  return -1;
 }
 int i=0;
 for(; i < g_list_length(users); i++) {
  chatclient *user = g_list_nth_data(users, i);
  if(user == NULL) {
   logger(SYS_INFO, "[add_users_to_chat] users null\n");
  }
  logger(SYS_INFO, "[data_add_users_to_chat] adding user %s\n", user->nick);
  data_add_user_to_chat (chat_id, user->id, user->nick, user->ip, user->port);
 }
 return 0;
}

/**
  * Crea un nuovo chatclient settando i campi con i valori dei parametri e lo aggiunge
  * alla hashtable degli utenti connessi alla chat con id pari a chat_id.
  */
int data_add_user_to_chat(u_int8 chat_id, u_int8 id, const char *nick, const char *ip,
     u_int4 port) {

 if(chat_hashtable==NULL)
  return -1;

 if(chatclient_hashtable==NULL)
  chatclient_hashtable = g_hash_table_new(g_str_hash, g_str_equal);

 data_add_user(id, nick, ip, port);

 //!creazione del nuovo chatclient
 chatclient *chatclient_str = (chatclient*)calloc(1, sizeof(chatclient));
 chatclient_str->id = id;
 chatclient_str->nick = (char*)strdup(nick);
 chatclient_str->ip = (char*)strdup(ip);
 chatclient_str->port = port;
 g_hash_table_insert(chatclient_hashtable, (gpointer)to_string(id), (gpointer)
      chatclient_str);

 chat *chat_str = (chat*)g_hash_table_lookup(chat_hashtable, (gconstpointer)to_string(
      chat_id));
 if(chat_str==NULL) {
  return -1;
 }

 //!inserimento del chatclient nella hashtable degli utenti della chat.
```

```
 pthread_mutex_lock(&chat_str->mutex);
 if(g_hash_table_lookup(chat_str->users, (gpointer)to_string(id))==NULL) {
  g_hash_table_insert(chat_str->users, (gpointer)to_string(id), (gpointer)chatclient_str
      );
 }
 pthread_mutex_unlock(&chat_str->mutex);

 return 0;
}

/**
  * Crea un nuovo chatclient settandone i campi con i valori dei parametri e lo
  * inserisce nella hashtable contenente tutti i chatclient
  */
int data_add_user(u_int8 id, const char *nick, const char *ip, u_int4 port) {

 //!istanzia l'hashtable qualora non sia presente.
 if(chatclient_hashtable==NULL)
  chatclient_hashtable = g_hash_table_new(g_str_hash, g_str_equal);

 //!creazione del nuovo chatclient e inserimento nella relativa hashtable
 chatclient *chatclient_str = (chatclient*)malloc(sizeof(chatclient));
 chatclient_str->id = id;
 chatclient_str->nick = (char*)nick;
 chatclient_str->ip = (char*)ip;
 chatclient_str->port = port;
 g_hash_table_insert(chatclient_hashtable, (gpointer)to_string(id), (gpointer)
     chatclient_str);

 return 0;
}

/**
  * Rimuove un utente dalla hashtable dei chatclient.
  */
int data_del_user(u_int8 id) {
 if(chatclient_hashtable==NULL)
  return -1;

 g_hash_table_remove(chatclient_hashtable, (gconstpointer)to_string(id));

 return 0;
}

/**
  * Rimuove l'utente con lo specifico id dalla hashtable dei chatclient e dalla
  * hashtable degli utenti connessi alla chat con id pari a chat_id.
  */
int data_del_user_from_chat(u_int8 chat_id, u_int8 id) {
 if(chat_hashtable==NULL || chatclient_hashtable==NULL)
  return -1;

 g_hash_table_remove(chatclient_hashtable, (gconstpointer)to_string(id));
 chat *chat_str = (chat*)g_hash_table_lookup(chat_hashtable, (gconstpointer)to_string(
     chat_id));
 g_hash_table_remove(chat_str->users, (gconstpointer)to_string(id));

 return 0;
}

/**
  * Funzione wrapper non piu' utilizzata. Invoca la funzione data_search_chat
  */
chat *data_search_chat_local(const char *title) {
```

```
 return data_search_chat(title);
}

/**
  * Cerca nella hashtable chat_table l'occorrenza della chat title
  * Ritorna la struttura dati della chat
  */
chat *data_search_chat(const char *title) {
 if(title==NULL || chat_hashtable==NULL)
  return NULL;

 GList *listchat = g_hash_table_get_values(chat_hashtable);
 chat *chatval;
 int j;
 for(j=0; j<g_list_length(listchat); j++) {
  chatval = (chat*)g_list_nth_data(listchat, j);
  if(strcmp(chatval->title, title)==0)
   return chatval;
 }
 return NULL;
}

/**
  * Cerca nella hashtable chat_table tutte le chat che hanno come titolo *title*
  * Ritorna le chat in una slist
  */
GList *data_search_all_chat(const char *title) {
 if(title==NULL) {
  return NULL;
 }
 if(chat_hashtable==NULL) {
  return NULL;
 }

 GList *listallchat=NULL;
 GList *listchat = g_hash_table_get_values(chat_hashtable);
 if(listchat==NULL) {
  return NULL;
 }
 chat *chatval;
 int j;
 for(j=0; j<g_list_length(listchat); j++) {
  chatval = (chat*)g_list_nth_data(listchat, j);
  //!controllo che ci sia almeno una sottostringa in comune tra il titolo inserito e le
       chat presenti.
  if(strstr(chatval->title, title)!=NULL) {
   //!aggiunge la chat alla lista.
   listallchat = g_list_prepend(listallchat, (gpointer)chatval);
  }
 }
 return listallchat;
}

/**
  * Funzione wrapper non piu' utilizzata.
  */
GList *data_search_all_local_chat(const char *title) {
 return data_search_all_chat(title);
}

/**
  * Cerca un chatclient nella hashtable dei chatclient. Non piu' utilizzata.
  */
chatclient *data_search_chatclient(const char *nick) {
```

```
 if(nick==NULL || chatclient_hashtable==NULL) {
  return NULL;
 }

 GList *listclient = g_hash_table_get_values(chatclient_hashtable);
 chatclient *clientval;
 int j;
 for(j=0; j<g_list_length(listclient); j++) {
  clientval = (chatclient*)g_list_nth_data(listclient, j);
  if(strcmp(clientval->nick, nick)==0)
   return clientval;
 }
 return NULL;
}

/**
  * Converte la lista di chat in una stringa del tipo:
  * 111;test;
  * 22;pippo;127.0.0.1;2110;
  * 33;pluto;127.0.0.1;2111;
  */
char *data_chatlist_to_char(GList *chat_list, int *len) {
 if(chat_list==NULL) {
  return NULL;
 }
 if(g_list_length(chat_list)==0)
  return NULL;

 chat *chat_elem;
 chatclient *chatclient_elem;
 int cur_size = 512;
 int cur = 0;
 char *ret = (char*)calloc(cur_size, 1);
 char *line = (char*)calloc(512, 1);
 int i, j;
 GList *chatclient_list;
 for(i=0; i<g_list_length(chat_list); i++) {
  chat_elem = (chat*)g_list_nth_data(chat_list, i);
  sprintf(line, "%lld;%s;\n", chat_elem->id, chat_elem->title);
  cur += strlen(line);
  if(cur>=cur_size) {
   cur_size *= 2;
   ret = realloc(ret, cur_size);
  }
  strcat(ret, line);

  chatclient_list = g_hash_table_get_values(chat_elem->users);
  for(j=0; j<g_list_length(chatclient_list); j++) {
   chatclient_elem = (chatclient*)g_list_nth_data(chatclient_list, j);
   sprintf(line, "%lld;%s;%s;%d;\n", chatclient_elem->id, chatclient_elem->nick,
        chatclient_elem->ip, chatclient_elem->port);
   cur += strlen(line);
   if(cur>=cur_size) {
        cur_size *= 2;
        ret = realloc(ret, cur_size);
   }
   strcat(ret, line);
  }

  cur++;
  if(cur>=cur_size) {
   cur_size *= 2;
   ret = realloc(ret, cur_size);
  }
```

```
  strcat(ret, "|");
 }
 *len = cur;

 return ret;
}

/**
  * Converte una stringa in una lista di chat con i relativi utenti
  * 111;test;
  * 22;pippo;127.0.0.1;2110;
  * 33;pluto;127.0.0.1;2111;
  * |222;test;
  * 333;si.......
  */
GList *data_char_to_chatlist(const char *buffer, int len) {
 char *saveptr, *saveptr2;
 char *buffer2 = strdup(buffer);
 char *token;
 int i=0;
 int line=-1;

 GList *chat_list=NULL;
 while((token = strtok_r(buffer2, "|", &saveptr))!=NULL) {
  line = -1;
  for(i=0;i<strlen(token);i++) {
   if(token[i]=='\n')
        line++;
  }
  if(line>-1) {
   chat *chat_elem=(chat *)calloc(1, sizeof(chat));
   chat_elem->id= atoll(strtok_r(strdup(token), ";", &saveptr2));
   chat_elem->title=strdup(strtok_r(NULL, ";", &saveptr2));
   chat_elem->users=g_hash_table_new(g_str_hash, g_str_equal);
   for(i=0;i<line;i++) {
        chatclient *chat_client=(chatclient *)calloc(1,sizeof(chatclient));
        chat_client->id=atoll(strtok_r(NULL, ";", &saveptr2));
        chat_client->nick=strdup(strtok_r(NULL, ";", &saveptr2));
        chat_client->ip=strdup(strtok_r(NULL, ";", &saveptr2));
        chat_client->port= atoi(strtok_r(NULL, ";", &saveptr2));
        g_hash_table_insert(chat_elem->users,(gpointer)to_string(chat_client->id),(
            gpointer)chat_client);
   }
   chat_list=g_list_append(chat_list,(gpointer)chat_elem);
   buffer2 = NULL;
   line = 0;
  }
 }
 return chat_list;
}

/**
  * Converte la lista di utenti in una stringa del tipo:
  * 22;pippo;127.0.0.1;2110;
  * 33;pluto;127.0.0.1;2111;
  */
char *data_userlist_to_char(GList *user_list, int *len) {
 if(user_list==NULL) {
  return NULL;
 }
 if(g_list_length(user_list)==0) {
  return NULL;
 }
```

```
 chatclient *chatclient_elem;
 int cur_size = 512;
 int cur = 0;
 char *ret = (char*)calloc(cur_size, 1);
 char *line = (char*)calloc(512, 1);
 int j;
 for(j=0; j<g_list_length(user_list); j++) {
  chatclient_elem = (chatclient*)g_list_nth_data(user_list, j);
  sprintf(line, "%lld;%s;%s;%d;\n", chatclient_elem->id, chatclient_elem->nick,
       chatclient_elem->ip, chatclient_elem->port);
  cur += strlen(line);
  if(cur>=cur_size) {
   cur_size *= 2;
   ret = realloc(ret, cur_size);
  }
  strcat(ret, line);
 }

 cur++;
 if(cur>=cur_size) {
  cur_size *= 2;
  ret = realloc(ret, cur_size);
 }
 ret = realloc(ret, cur);
 *len = cur;

 return ret;
}

/**
  * Converte una stringa in una lista di utenti
  */
GList *data_char_to_userlist(const char *buffer, int len) {
 char *saveptr, *saveptr2;
 char *buffer2 = strdup(buffer);
 char *token;
 GList *user_list = NULL;
 while((token = strtok_r(buffer2, "\n", &saveptr))!=NULL) {

  //!creazione del nuovo chatclient e settaggio dei parametri tramite tokenizzazione del
       buffer
  chatclient *chat_client=(chatclient *)calloc(1, sizeof(chatclient));
  chat_client->id=atoll(strtok_r(token, ";", &saveptr2));
  chat_client->nick=strdup(strtok_r(NULL, ";", &saveptr2));
  chat_client->ip=strdup(strtok_r(NULL, ";", &saveptr2));
  chat_client->port= atoi(strtok_r(NULL, ";", &saveptr2));
  //!inserimento del nuovo chatclient nella lista
  user_list = g_list_append(user_list, (gpointer)chat_client);
  buffer2 = NULL;
 }
 return user_list;
}

/**
  * Ritorna una lista di tutti i client della chat specificata
  */
GList *data_get_chatclient_from_chat(u_int8 id) {
 chat *chatval;
 if((chatval=data_get_chat(id))!=NULL)
  return g_hash_table_get_values(chatval->users);
 return NULL;
}

/**
```

```
  * Ritorna la chat con lo specifico chat_id.
  */
chat *data_get_chat(u_int8 chat_id) {
 if(chat_hashtable == NULL)
  return NULL;
 return g_hash_table_lookup(chat_hashtable, to_string(chat_id));
}

/**
  * Ritorna il chatclient con lo specifico id.
  */
chatclient *data_get_chatclient(u_int8 id) {
 return g_hash_table_lookup(chatclient_hashtable, to_string(id));
}

/**
  * Rimuove un utente da tutte le hashtable delle chat in cui e' presente e infine
  * lo rimuove dalla hashtable dei chatclient.
  */
int data_destroy_user(u_int8 id) {

 if(id==0)
  return -1;

 GList *chats = g_hash_table_get_values(chat_hashtable);
 int i=0;
 //!per ogni chat rimuove l'utente dalla hashtable degli utenti connessi (qualora
      presente)
 for(; i<g_list_length(chats); i++) {
  chat *chat_val = (chat*)g_list_nth_data(chats, i);
  if(chat_val!=NULL) {
   g_hash_table_remove(chat_val->users, (gconstpointer)to_string(id));
  }
 }
 //!rimuove l'utente dalla hashtable dei chatclient
 data_del_user(id);
 return 0;
}
```

1.26 tortella.c

```
#include "controller.h"

/**
 * Lancia l'applicazione, inizializzando il controller in due modalita' differenti
 * a seconda dei parametri passati da riga di comando. Il primo parametro dev'essere
 * il path del file di configurazione, mentre il secondo (opzionale nel caso non si
 * conosca alcun vicino) dev'essere il path del file in cui sono inseriti ip e
 * porta dei peer vicini conosciuti. La seconda fase consiste nell'inizializzazione
 * dei thread necessari a gestire la gui e infine si avvia l'interfaccia grafica
 * tramite la chiamata a controller_init_gui().
 */
int main(int argc, char *argv[])
{

 if(argc<2) {
  printf("Usage: <conf_path> [cache_path] [filename]\n");
  return 0;
 }

 //!inizializzazione del controller
 if(argc==2) {
  controller_init(argv[1], NULL);
 }
 else {
  controller_init(argv[1], argv[2]);
 }

 //!inizializzazione dei thread
 g_thread_init(NULL);
 gdk_threads_init();

 //!inizializzazione della gui
 gtk_init (&argc, &argv);
 controller_init_gui();

 return (0);
}
```

Listings