Università degli Studi di Napoli Federico II
Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione
Corso di Laurea Triennale in Informatica

Relazione "LocationHub"

Autori - Gruppo N13
Davide Pio FAICCHIA N86003018
Mauro GUIDA N86002889

Anno Accademico 2020–2021

Indice

1 Compilazione ed utilizzo del server **2**
1.1 Compilazione ed Esecuzione . . . 2
1.2 Utilizzo . . . 2

2 Architettura Applicazione Android **3**
2.1 Activity . . . 3
2.1.1 Login . . . 3
2.1.2 Main . . . 4
2.2 ViewModel . . . 4
2.2.1 Login . . . 4
2.2.2 Main . . . 4
2.3 Repository . . . 5

3 Utilizzo del client **7**
3.1 Login . . . 7
3.2 Home . . . 8
3.3 People . . . 9
3.4 Settings . . . 10
3.5 Perdita di connessione o accesso GPS . . . 11

4 Protocollo Comunicazione Client-Server **12**
4.1 SIGN_UP . . . 12
4.2 GET_LOCATIONS . . . 12
4.3 SEND_LOCATION . . . 13
4.4 SET_PRIVACY . . . 13

5 Dettagli implementativi del server **14**
5.1 Gestione pool di utenti . . . 14
5.2 Gestione richieste client . . . 15
5.3 Gestione dei segnali . . . 17
5.4 Gestione dei log . . . 18

1. Compilazione ed utilizzo del server

1.1 Compilazione ed Esecuzione

Per semplificare il processo di compilazione si è preferito fare uso di un Makefile, presente nella directory contenente il codice sorgente del server.

Per tanto la procedura di compilazione si riduce ad un unico comando:

- `make`: Effettua la compilazione del server, generando un eseguibile `myserver`
- `make clean`: Rimuove tutti i file oggetto

1.2 Utilizzo

Per eseguire il server è necessario lanciare il seguente comando: `./myserver <port_num>`
Si ricorda che `<port_num>` è un intero nell'intervallo 1024-65535.
Durante il suo ciclo di vita il server stamperà su stdout le varie righe di log. Inoltre, sarà generato un file `location_hub.log` che conserverà tali righe, permettendo dunque una successiva revisione delle stesse.
Verranno generati delle righe di log per i seguenti eventi:

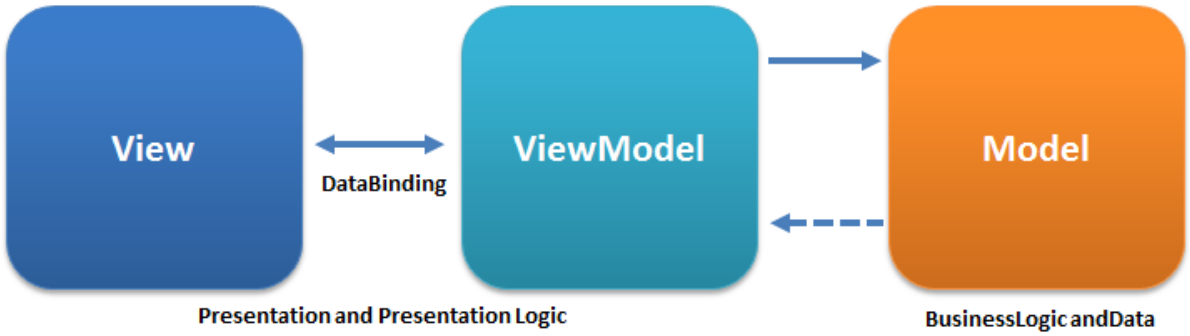
- Nuova connessione client
- Disconnessione client
- Registrazione
- Richiesta posizioni
- Invio posizione del client
- Richiesta cambio privacy
- Invio messaggio non valido

Le righe di log avranno il seguente formato:

```
dd-mm-yyyy HH:MM:SS <ipaddr>:<port> <riga_di_log>
```

2. Architettura Applicazione Android

L'applicazione Client sviluppata per Android si basa sul pattern architetturale MVVM; tale pattern prevede l'associazione di apposite viste (dette View) a relativi controller (detti ViewModel) al fine di gestire ed aggiornare i dati mostrati. Il dialogo con il server è gestito su richiesta del ViewModel da un apposito Repository, impegnato nel recupero e creazione dei Modelli che rappresentano l'informazione.



Sull'Activity Main poggeranno Tre Fragment distinti con il compito di visualizzare la posizione degli utenti (Schermata Home e People) e di modificare il comportamento dell'applicazione attraverso impostazioni di Privacy e raggio di visualizzazione Client terzi.

2.1 Activity

In questa sezione verranno specificati i servizi offerti dalle Activity, tralasciando il loro funzionamento come View, la reazione di quest'ultime sarà infatti discusso nell'apposito capitolo (3).

2.1.1 Login

L'Activity Login avrà il compito di garantire il corretto funzionamento dell'applicazione verificando che l'utente abbia fornito i permessi di accesso alla Posizione.

Nel caso in cui tale requisito non sia soddisfatto, l'activity impedirà l'accesso dell'utente, segnalando il problema riscontrato mediante l'apposita View.

2.1.2 Main

L'Activity Main avrà il compito di fornire i servizi necessari al funzionamento dell'applicazione, quali:

- Aggiornamento dei dati visualizzati
- Controllo problemi di Connessione o Localizzazione
- Aggiornamento posizione del Client in uso per gli altri utenti connessi

Tali servizi saranno garantiti durante tutta la vita dell'applicazione, nel caso in cui si verifichino problemi di Connessione o Localizzazione l'utente sarà avvisato mediante Dialog (Capitolo 3 Sezione 5) e gli sarà data la possibilità di tentare di risolvere il problema o terminare l'applicazione.

L'aggiornamento della posizione del Client in uso e degli utenti connessi, sarà effettuata ogni 10 secondi.

2.2 ViewModel

2.2.1 Login

Il LoginViewModel ha il compito di controllare la validità del Nickname inserito nell'apposita View, controllando l'assenza di caratteri speciali, oltre alla lunghezza minima e massima, segnalando alla View eventuali errori commessi dall'utente.

Una volta superata la verifica locale della validità del Nickname, il ViewModel si interfaccerà con il Repository al fine di verificare la disponibilità dei tale Nickname sul Server; anche in questa circostanza in caso di fallimento, l'utente sarà avvistato tramite l'apposita View (Capitolo 3 Sezione 1).

2.2.2 Main

Il MainViewModel ha il compito di interfacciarsi con il Repository per recuperare ed inviare i dati al server, verificando eventuali problemi.

L'Activity Main si interfaccerà con questo ViewModel per richiedergli l'aggiornamento dei dati da esso mantenuti (quali la posizione corrente dei Client terzi) e fornendogli l'attuale posizione del client corrente.

- La posizione corrente del Client sarà quindi conservata dal MainViewModel ed inviata al server.
- Il MainViewModel aggiornerà la posizione dei Client Terzi e notificherà le modifiche alle Views.

Il MainViewModel sarà anche responsabile di gestire le richieste effettuate dall'utente tramite le View, quali:

- La modifica della Privacy
- La modifica del Range di visualizzazione dei Client Terzi

2.3 Repository

Il Repository è il Cuore della comunicazione dell'applicazione con l'esterno, esso consiste banalmente in una Socket Java impegnata nel recupero e nell'invio di dati da e verso il Server.

Tale Socket sarà istanziata durante la fase di login, stabilendo una Connessione TCP/IP con il server, che resterà attiva fino a chiusura dell'applicazione.

La Socket in quanto Repository ha il compito di inviare in modo grezzo le informazioni del Client in uso, al server, oltre a recuperare le informazioni grezze fornite dal server, al fine di costruire oggetti utilizzabili dai sovrastanti ViewModel.

Le possibili operazioni della Socket sono specificate nel dettaglio al Capitolo 4 di questa Relazione.

```
1   public void login(String username) throws UsernameAlreadyInUseException,
        IOException {
2       String response = sendMessage(SIGN_UP + username);
3
4       if (response == null || response.isEmpty() || !response.equals(OK_RESPONSE))
5           throw new UsernameAlreadyInUseException();
6   }
7
8   public void sendClientPosition(Position p) throws NoInternetConnectionException,
        ServerResponseException {
9       try {
10          String response = sendMessage(SEND_LOCATION + p.getLatitude() + " " +
                p.getLongitude());
11
12          if (response == null || response.isEmpty() ||
                !response.equals(OK_RESPONSE))
13              throw new ServerResponseException();
14
15      } catch (IOException e) {
16          throw new NoInternetConnectionException();
17      }
18  }
19
20  private void updateUsersLocation() throws IOException, ServerResponseException {
21      String response = sendMessage(GET_LOCATIONS);
22
23      if (response == null || response.isEmpty() ||
            !response.startsWith(OK_RESPONSE))
24          throw new ServerResponseException();
25
26      userSet.clear();
27      userSet.addAll(StringParser.usersParser(response));
28  }
29
30
```

```
31      public List<User> getAllConnectedUsers() throws NoInternetConnectionException,
            ServerResponseException {
32          try {
33              updateUsersLocation();
34          } catch (IOException e) {
35              throw new NoInternetConnectionException();
36          }
37
38          return userSet;
39      }
40
41      public void setUserPrivacy(boolean b) throws ServerResponseException,
            NoInternetConnectionException {
42          try {
43              String response = sendMessage(SET_PRIVACY + (b ? 1 : 0));
44
45              if (response == null || response.isEmpty() ||
                    !response.equals(OK_RESPONSE))
46                  throw new ServerResponseException();
47
48          } catch (IOException e) {
49              throw new NoInternetConnectionException();
50          }
51      }
```

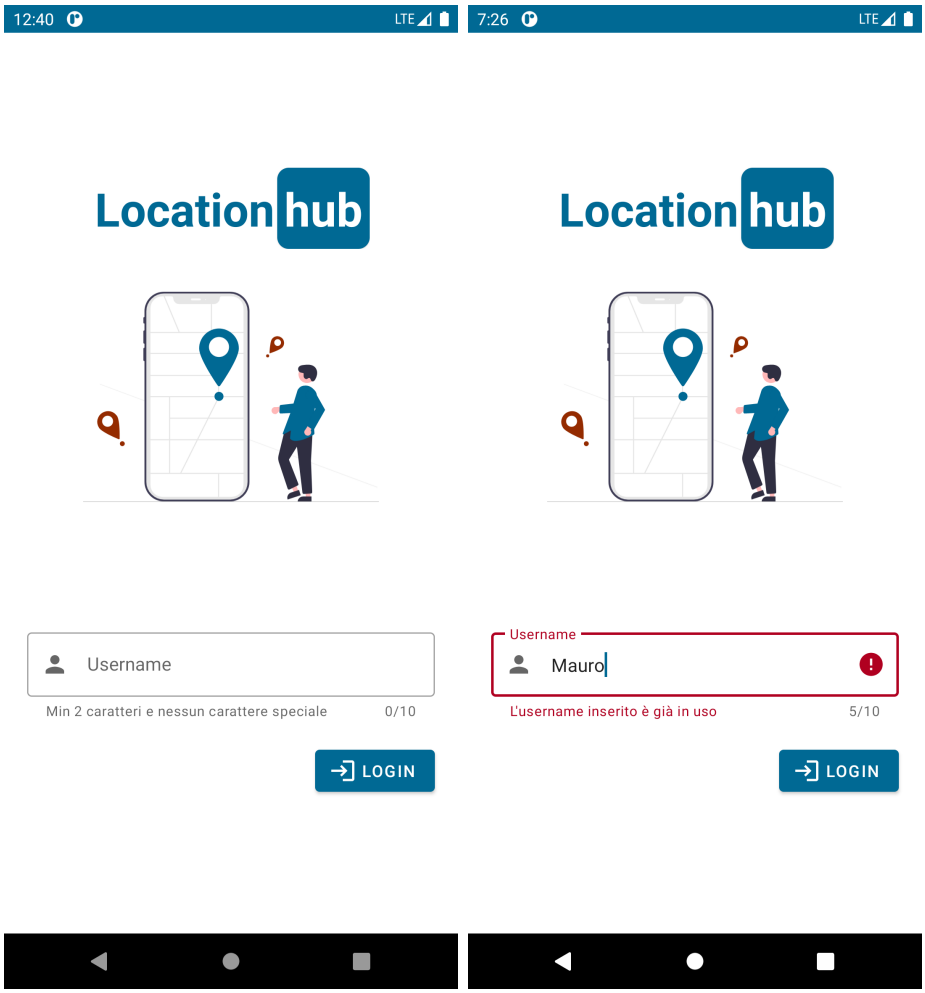
3. Utilizzo del client

Il client aggiornerà automaticamente la propria posizione e la posizione corrente dei client connessi al server, ogni 10 secondi; le schermate saranno aggiornate di conseguenza.

3.1 Login

Con la pressione del bottone Login, il Nickname inserito sarà controllato ed inviato al server al fine di verificare la sua univocità nel sistema, nel caso in cui la verifica abbia esito negativo tale problematica verrà segnalata all'utente mediante messaggio di errore sulla `EditText` apposita.

Seguono la UI della schermata Login:



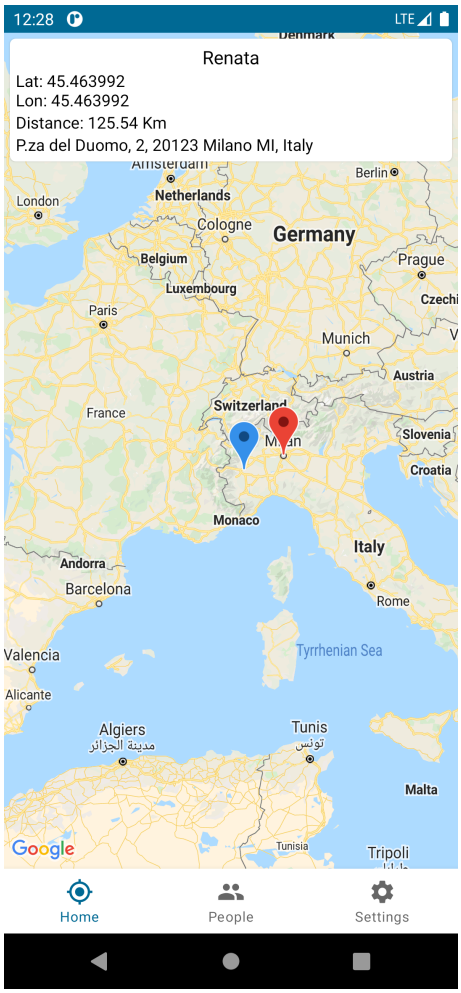
3.2 Home

La schermata Home mostra su mappa la posizione corrente di tutti i client connessi e nel range stabilito nelle Impostazioni.

- Il marker blu indica la posizione corrente del Client in uso
- Il marker rosso indica la posizione degli altri client connessi

Alla pressione di un marker apparirà una `CardView` riportante le seguenti informazioni:

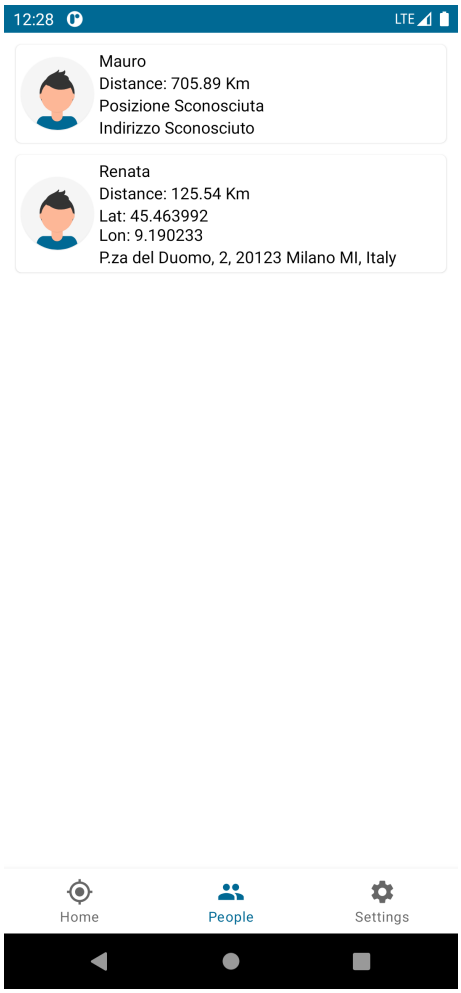
- Nickname
- Latitudine
- Longitudine
- Distanza (in Km)
- Indirizzo



3.3 People

La schermata presenta una RecyclerView contenente CardView rappresentative di ogni Client connesso con le annesse informazioni.

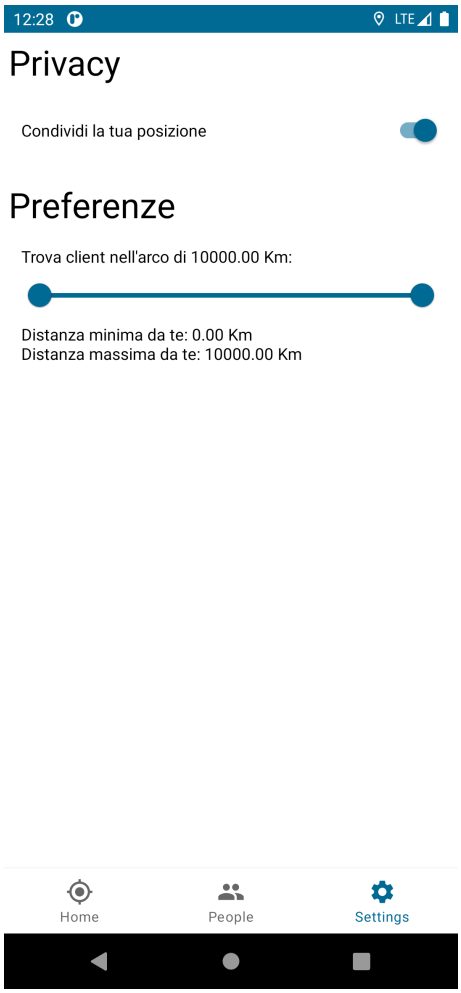
Questa schermata mostrerà anche i client con Privacy attiva, non riportandone le coordinate precise e l'indirizzo; la distanza sarà comunque consultabile.



3.4 Settings

La schermata impostazioni influenza i dati visualizzati localmente attraverso lo slider Range. Tale slider consente di impostare la distanza minima e massima di visualizzazione di client terzi.

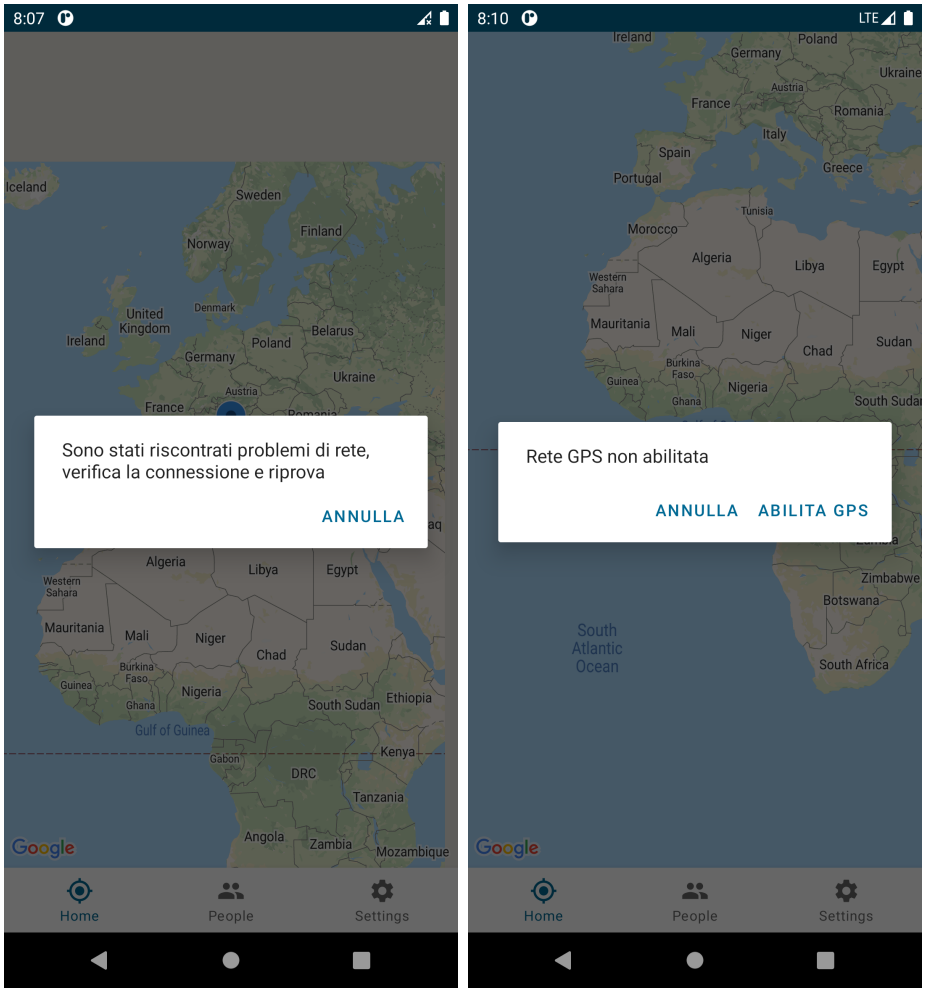
Lo switch "condividi la tua posizione" consente di impostare la propria Privacy per la sessione corrente. Di default la condivisione della propria posizione a client terzi è disattivata.



3.5 Perdita di connessione o accesso GPS

Nel caso in cui si verifichi un errore di connessione lato server oppure lato client la seguente Dialog sarà visualizzata a schermo, consentendo unicamente la chisura dell'app.

Altra possibile evenienza è la disattivazione del segnale GPS, tale errore mostrerà una Dialog che consentirà all'utente o la chiusura dell'applicazione oppure l'attivazione del GPS.



4. Protocollo Comunicazione Client-Server

Il protocollo di comunicazione progettato è reso quanto più funzionale e minimale , rendendo così la comunicazione tra client e server semplice e veloce.

Esso si basa su una serie di token, i quali identificano il servizio richiesto dai client. Per ciascuna richiesta inviata dal client, il server risponde con un opportuno messaggio di notifica.

4.1 SIGN_UP

Per poter utilizzare l'applicazione nella sua interezza, il client dovrà superare una prima fase di registrazione.

Provvederà dunque ad inviare al server la seguente stringa: `"SIGN_UP nickname"`.
La stringa `nickname` rappresenta l'identificativo scelto dall'utente, col quale verrà riconosciuto dagli utente connessi.

A tale richiesta il server risponderà con la stringa `"ERR"` se dovesse già esistere un utente con quel nickname. Questa verrà interpretata lato client modificando opportunamente la UI.

Se invece non dovesse esistere alcun utente con quel nickname, il server invierà la stringa `"OK"` completando la fase di registrazione, permettendo dunque all'utente di accedere ai vari servizi.

4.2 GET_LOCATIONS

Per richiedere le posizioni di tutti gli utente attualmente connessi, il client provvederà all'invio della seguente stringa: `"GET_LOCATIONS"`.

Il server fornirà le posizioni con il seguente formato:

```
"OK - {  nickname distance privacy [latitude;longitude]@
         nickname distance privacy [latitude;longitude]@ }"
```

Per ciascun utente, diverso dal richiedente, è presente una riga nel payload che manterrà le seguenti informazioni: nickname, distanza in km dal client richiedente, visibilità della posizione, latitudine e longitudine.

Le informazioni di ciascun utente sono separate dal carattere `@`.

Ulteriori informazioni relative la posizione degli utenti quali indirizzo o nazione verranno recuperate lato client tramite le API offerte da Google.

4.3 SEND_LOCATION

Ogni 10 secondi il client provvederà a mantenere aggiornato il server riguardo la sua posizione.

Per fare ciò invierà la seguente stringa: `"SEND_LOCATION latitudine longitudine"`.
Il server aggiornerà la posizione ed invierà al client la stringa `"OK"`.

4.4 SET_PRIVACY

L'utente in qualsiasi istante può decidere di modificare la visibilità della propria posizione inviando al server la seguente stringa: `"SET_PRIVACY 0/1"`.

Il parametro `1` renderà la visibilità privata; diversamente, il parametro `0` renderà la visibilità pubblica.
Il server aggiornerà la visibilità della posizione ed invierà al client la stringa `"OK"`.

Si ricordi che la visibilità di una posizione è unicamente relativa all'indirizzo; qualora la posizione fosse privata, tutti gli utente connessi saranno a conoscenza della sola distanza in km che li separa dall'utente in questione.

5. Dettagli implementativi del server

5.1 Gestione pool di utenti

Il server gestisce il pool di utenti mediante un albero bilanciato, struttura adatta a questo caso d'uso data la possibilità di effettuare ricerche, eliminazioni ed inserimenti in tempo logaritmico. Poichè diversi threads dovranno interagire con l'albero, questo sarà dotato di un fast mutex, così da evitare problemi di sincronizzazione come race condition.

```
1  struct client_location_t
2  {
3      double latitude;
4      double longitude;
5  };
6  typedef struct client_location_t client_location_t;
7
8  struct node_t
9  {
10     char              *nickname;
11     client_location_t *client_location;
12     bool              is_private;
13
14     int               height;
15     struct node_t     *left;
16     struct node_t     *right;
17 };
18 typedef struct node_t node_t;
19
20 struct avl_t
21 {
22     node_t          *root;
23     comparator      comp;
24     pthread_mutex_t lock;
25 };
26 typedef struct avl_t avl_t;
```

Per ciascun utente connesso il server provvederà alla creazione di un nodo che manterra il seguente payload:

- Nickname (Chiave di ricerca dell'albero)
- Posizione (Coppia latitudine-longitudine)
- Visibilità della posizione

Le operazioni di modifica della posizione e visibilità della stessa si riducono essenzialmente ad una ricerca del nodo ed una successiva modifica dell'informazione, il tutto in maniera sincronizzata.

5.2 Gestione richieste client

Il blocco di codice principale dei thread associati ai rispettivi client è il seguente:

```
for (;;)
{
    timeout.tv_sec = 15;
    timeout.tv_usec = 0;

    FD_ZERO(&read_fds);
    FD_SET(client_sockfd, &read_fds);

    if (select(client_sockfd + 1, &read_fds, NULL, NULL, &timeout) > 0)
    {
        n = read(client_sockfd, buf, BUF_SIZE);
        if (n > 0)
        {
            buf[n] = '\0';

            req = extract_request(buf);

            switch (req)
            {
            case SIGN_UP:
                log_print(server->logger, LOG_SIGN_UP, client_ip_addr,
                    client_port_num);
                sign_up(server, &nickname, buf, client_sockfd);
                break;

            case GET_LOCATIONS:
                log_print(server->logger, LOG_GET_LOCATIONS, client_ip_addr,
                    client_port_num);
                send_locations_to_client(server, nickname, client_sockfd);
                break;

            case SEND_LOCATION:
                log_print(server->logger, LOG_SEND_LOCATION, client_ip_addr,
                    client_port_num);
                set_client_location(server, nickname, buf, client_sockfd);
                break;

            case SET_PRIVACY:
                log_print(server->logger, LOG_SET_PRIVACY, client_ip_addr,
                    client_port_num);
                set_client_privacy(server, nickname, buf, client_sockfd);
```

```
38                  break;
39
40              default:
41                  log_print(server->logger, LOG_INVALID_MSG, client_ip_addr,
                        client_port_num);
42                  char *msg = "ERR\n";
43                  write(client_sockfd, msg, strlen(msg));
44                  break;
45              }
46          }
47          else if (n == 0 || n == -1)
48          {
49              log_print(server->logger, LOG_DISCONNECTION, client_ip_addr,
                    client_port_num);
50              avl_remove(server->avl, nickname);
51              free(nickname);
52              break;
53          }
54      }
55      else
56      {
57          log_print(server->logger, LOG_DISCONNECTION, client_ip_addr, client_port_num);
58          avl_remove(server->avl, nickname);
59          free(nickname);
60          break;
61      }
62  }
```

Come è possibile notare, la lettura del messaggio scritto sulla socket dal client viene seguita da uno switch-case che, a seconda della richiesta del client, provvederà a eseguire il giusto blocco di codice, stampando la riga di log opportuna.

Se la system call `read` legge 0 (nel caso in cui il client chiude la socket, ovvero alla chiusura dell'app mobile) oppure -1 (in caso di errore), allora si provvede alla deallocazione delle risorse e alla successiva chiusura del thread, accompagnate dall'opportuno log.

L'utilizzo della funzione `select` è stato necessario per rilevare un'eventuale perdita di connessione internet da parte del client. La `select` è una funzione che permette di monitorare il verificarsi di un evento su uno o più insiemi di File Descriptors entro un numero di secondi definito tramite una struct `timeval`.

Nel nostro caso d'uso la `select` rileva lo stato del Socket Descriptor del client. Se non dovesse rilevare alcuna scrittura da parte del client entro 15 secondi (ergo, il client ha perso accesso alla connessione internet), allora restituirà 0, procedendo con la deallocazione e con la chiusura del thread.

5.3 Gestione dei segnali

Il sollevamento dei segnali `SIGINT` e `SIGTERM` provocano la chiusura del server. Un'opportuna funzione di cleanup `server_destroy` provvederà alla deallocazione dell'`albero AVL` e del `logger`.

```
struct server_t
{
    int               sockfd;
    struct sockaddr_in sockaddr;
    int               port_num;

    logger_t          *logger;
    avl_t             *avl;
};
typedef struct server_t server_t;

if (signal(SIGINT, sig_handler) == SIG_ERR ||
     signal(SIGTERM, sig_handler) == SIG_ERR)
{
    perror("signal");
    fprintf(stderr, "Failed to install termination signal handler.\n");
    exit(errno);
}

void sig_handler(int sig_num)
{
    server_destroy(server);
    exit(0);
}

void server_destroy(server_t *server)
{
    if (server)
    {
        logger_destroy(server->logger);
        avl_destroy(server->avl);

        close(server->sockfd);
        free(server);

        printf("\nServer shutdown.\n");
    }
}
```

5.4 Gestione dei log

Ogniqualvolta dovesse essere necessario effettuare un log verrà invocata la funzione `log_print`. Tale funzione richiede 4 parametri: il logger condiviso dai thread col quale si ha accesso al file di log `location_hub.log`, l'evento che scaturisce il log, indirizzo ip e numero di porta del client.

Attraverso la struct `tm` e alle funzioni della libreria `<time.h>` è possibile recuperare l'ora e la data in cui si stampa la riga di log.

```
enum log_type_t
{
    LOG_NEW_CONNECTION,
    LOG_DISCONNECTION,
    LOG_SIGN_UP,
    LOG_GET_LOCATIONS,
    LOG_SEND_LOCATION,
    LOG_SET_PRIVACY,
    LOG_INVALID_MSG
};
typedef enum log_type_t log_type_t;

static const char *log_strings[] =
{
    [LOG_NEW_CONNECTION] = "************ NEW CONNECTION ***********",
    [LOG_DISCONNECTION]  = "************ DISCONNECTED ***********",
    [LOG_SIGN_UP]        = "Client requested registration",
    [LOG_GET_LOCATIONS]  = "Client requested all locations",
    [LOG_SEND_LOCATION]  = "Client requested to sent his location",
    [LOG_SET_PRIVACY]    = "Client requested to change in privacy settings",
    [LOG_INVALID_MSG]    = "Client sent an invalid message"
};

void log_print(logger_t *logger, log_type_t log_type, char *ip_addr, int port_num)
{
    time_t rawtime;
    struct tm *timeinfo;
    char log[512] = {'\0'};

    if (logger && ip_addr)
    {
        pthread_mutex_lock(&logger->lock);

        if ((logger->logfile = fopen(LOGFILE, "a+")) == NULL)
        {
            pthread_mutex_unlock(&logger->lock);
            return;
        }

        time(&rawtime);
        timeinfo = localtime(&rawtime);

```

```
43
44          sprintf(log, "%02d-%02d-%d %02d:%02d:%02d %s:%d %s", timeinfo->tm_mday,
45                                                               timeinfo->tm_mon + 1,
46                                                               timeinfo->tm_year + 1900,
47                                                               timeinfo->tm_hour,
48                                                               timeinfo->tm_min,
49                                                               timeinfo->tm_sec,
50                                                               ip_addr,
51                                                               port_num,
52                                                               log_strings[log_type]);
53
54          printf("%s\n", log);
55          fprintf(logger->logfile, "%s\n", log);
56
57          fclose(logger->logfile);
58          logger->logfile = NULL;
59
60          pthread_mutex_unlock(&logger->lock);
61      }
62  }
```