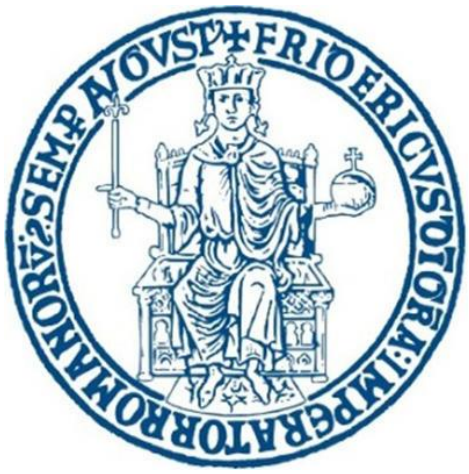


UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

ANNO ACCADEMICO 2021/2022

Documentazione relativa al progetto Potholes del gruppo LSO_2122_26

ANGELO PIO AMIRANTE	N86003932
VINCENZO BRANCACCIO	N86003944
FRANCESCO SCARFATO	N86003789

Sommario

Introduzione 3

Parte I 5

Guida all'uso 5

1 - Guida alla compilazione e all'uso del client 5

2 - Guida alla compilazione e all'uso del server 9

Parte II 11

Illustrazione del protocollo di comunicazione a livello di applicazione tra client e server 11

1 – Primo accesso 12

2 – Richiesta del valore soglia 13

3 – Salvataggio della buca rilevata 14

4 – Visualizzazione delle buche in un raggio stabilito 15

Parte III 17

Implementazione del server 17

Parte IV 23

Implementazione del client 23

Introduzione

La richiesta del progetto era lo sviluppo di un sistema *client-server* che consenta la raccolta e l'interrogazione di informazioni riguardanti la presenza di irregolarità (buche) sul manto stradale.

Il sistema deve consentire agli utenti di:

- Registrarsi con uno username identificativo
- Avviare una sessione di registrazione in cui si rilevano le buche
- Visualizzare tutte le buche rilevate in un certo raggio dalla sua posizione corrente, anche tramite mappa

Il client deve essere realizzato

- In linguaggio Java
- Su piattaforma Android
- Deve far uso dei sensori di accelerazione

Il server deve essere realizzato

- In linguaggio C
- Essere ospitato su public cloud
- Deve poter gestire l'accesso concorrente di un numero arbitrario di client

In aggiunta gestisce

- La creazione di un database MySQL

- Gli inserimenti sul database
- Le interrogazioni al database

La comunicazione tra client e server deve essere realizzata facendo uso di socket TCP o UDP.

La rilevazione della buca avviene comunicando un valore soglia al client, il quale durante la sessione di registrazione verifica se il valore dell'accelerometro lungo l'asse Y supera il valore soglia, in tal caso la buca è stata rilevata, viene mandata al server e memorizzata nel database.

Il valore soglia rappresenta l'accelerazione sull'asse Y che deve raggiungere il client per registrare una buca ed è stato impostato a 10.0 m/s^2.

La visualizzazione delle buche avviene comunicando al server di voler visualizzare le buche nelle vicinanze e mandandogli la posizione corrente; il server recupera le buche, controlla se sono comprese in un certo raggio (che è stato impostato a 30 km) e le invia al client.

Parte I

Guida all'uso

1 - Guida alla compilazione e all'uso del client

L'applicativo è stato realizzato usando l'ambiente di sviluppo Android Studio. Una volta definite tutte le risorse da usare, definiti i layout per ciascun schermata e il relativo codice Java, la compilazione si esegue cliccando sul bottone verde a forma di triangolo in alto sulla barra di navigazione (Fig 1.1) che rappresenta il "Run", infatti questo pulsante permette anche di installare e eseguire l'applicazione su un emulatore (che deve essere dotato di un account Google per usare i servizi di Google Play Service) o su un dispositivo, se quest'ultimo è collegato.

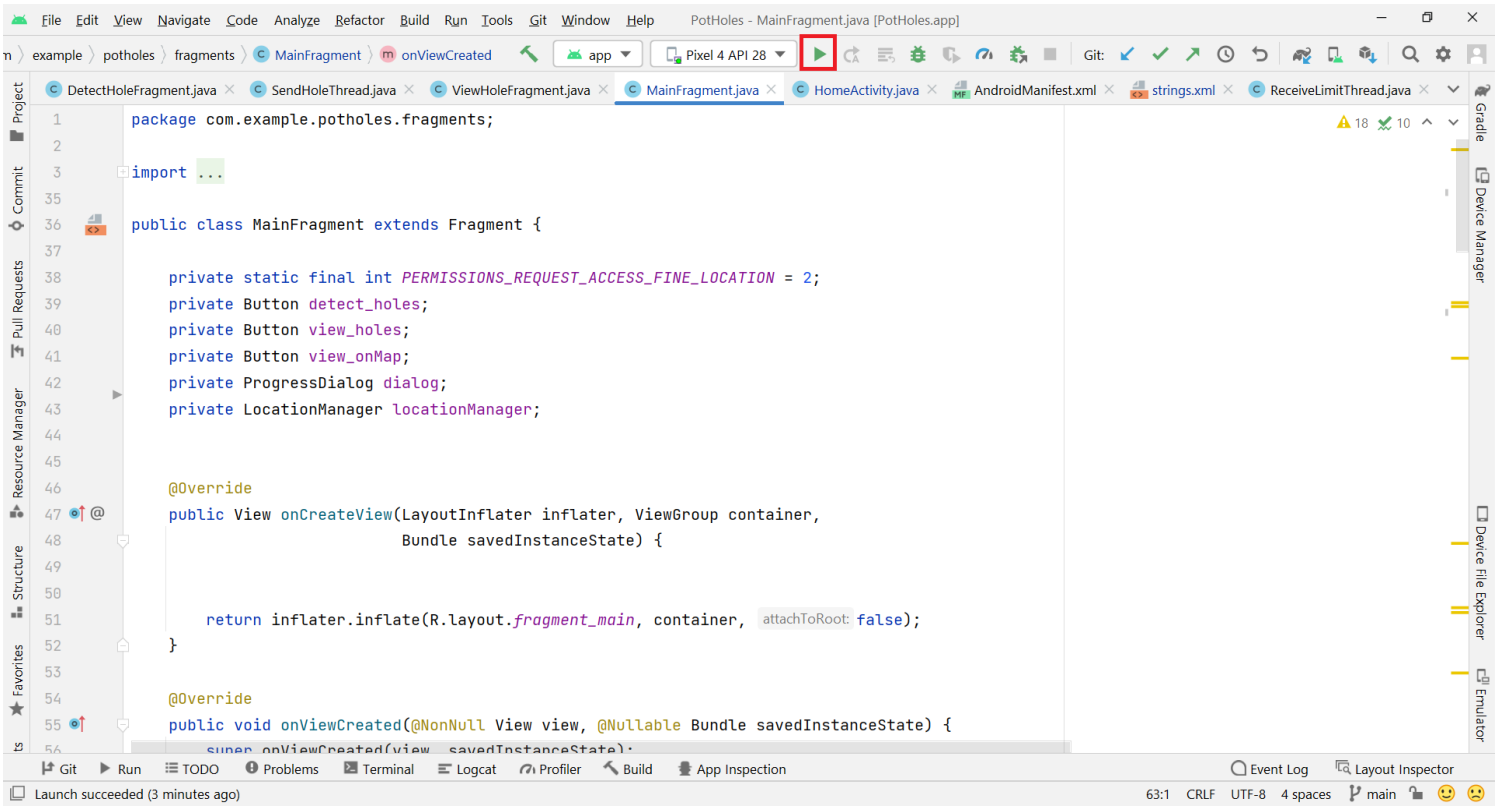


Fig 1.1 Nel quadrato rosso è evidenziato il tasto ”Run”

Successivamente, dopo aver compilato e installato l’app, quest’ultima si avvia e alla prima esecuzione, si apre la schermata di registrazione (Fig1.2) che consente all’utente di immettere il proprio username, che verrà salvato nelle shared preferences, quindi non verrà richiesto ai successivi avvii. L’utente dovrà però prima autorizzare l’utilizzo, da parte del dispositivo, della propria posizione. Se viene inserito un username duplicato, si ha un errore altrimenti si va alla schermata successiva (schermata Home) che è una schermata che permette di gestire l’applicazione (Fig 1.3). Da qui l’utente può decidere l’azione da intraprendere:

- Avviare una sessione di rilevazione (Fig 1.4) dove è anche possibile visualizzare le buche rilevate durante la sessione

- Visualizzare le buche in un elenco oppure su una mappa (Fig 1.5). Le buche visualizzate su mappa vengono filtrate in base alla distanza, vengono infatti mostrate sole le buche entro un certo raggio dalla propria posizione.

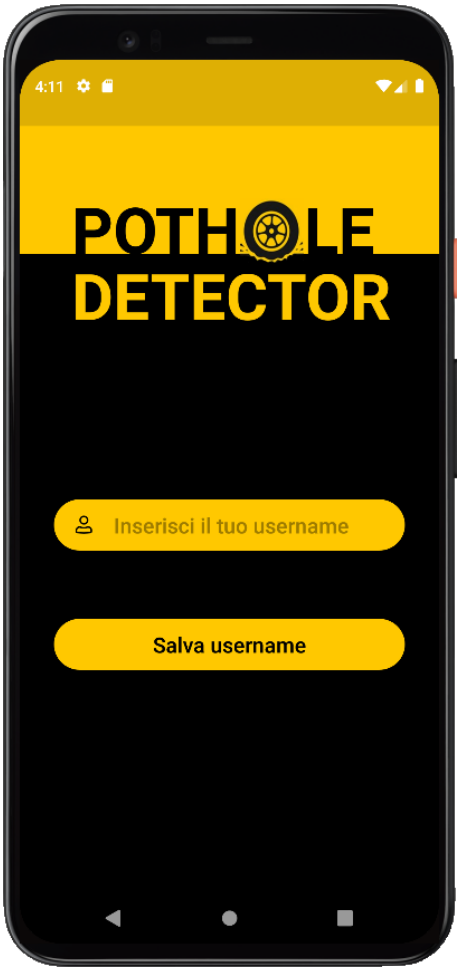


Fig 1.2 *Schermata di registrazione*

Fig 1.3 *Schermata Home*

Fig 1.4 *Rilevazione buche*

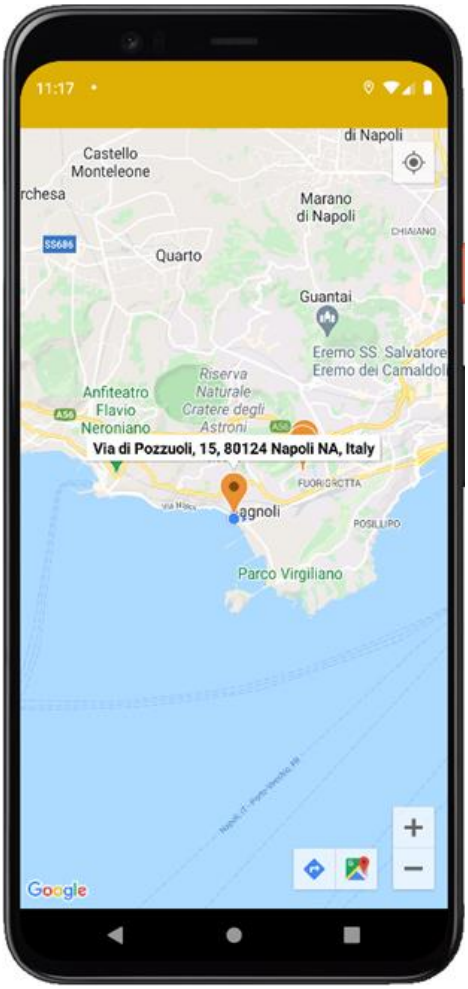


Fig 1.5 *Visualizzazione su mappa*

2 - Guida alla compilazione e all'uso del server

La compilazione del server avviene tramite seguente comando:

```
root@Ubuntu:/home/lso/Unix# gcc -pthread  -o server -g *.c -lm  -lmysqlclient
```

Fig 1.6

- -pthread aggiunge il supporto per il multithreading con le librerie pthreads
- -lm permette l'utilizzo delle librerie matematiche
- -lmysqlclient permette l'utilizzo delle API di MySql

All'avvio del server viene creata una socket che si pone in ascolto di una connessione da parte di un client (Fig 1.7)

```
root@Ubuntu:/home/lso/Unix# ./server
[#] [03-03-2022 09:10:08] Socket created.
[#] [03-03-2022 09:10:08] Connected to the port.
[#] [03-03-2022 09:10:08] Listen to incoming connections.
```

Fig 1.7 Avvio del server

In seguito alla connessione di un client esso può effettuare diverse operazioni, le operazioni sono identificate da un intero (vedere la parte 2 per maggiori dettagli)
Nel caso l'utente provi a registrarsi esso effettua una richiesta passando al server l'intero 3, nel caso l'username non sia già esistente viene salvato nel database (Fig 1.8) altrimenti riceve un errore (Fig 1.9).

```
[#] [03-03-2022 11:35:47] Accepted connection from the server to client 93.70.241.243 with socket descriptor 4.
[#] [03-03-2022 11:35:47] Thread -1573251328 handle client connection 4.
[#] [03-03-2022 11:35:48] Client 4 request is type 3: client wants to access.
[#] [03-03-2022 11:35:48] Connected to database.
[#] [03-03-2022 11:35:48] Username tarantell correctly saved.
[#] [03-03-2022 11:35:48] Sent check value 0 to client 4.
```

Fig 1.8 Il client con ip 93.70.241.243 si connette al server e richiede di salvare l'username "tarantell" che viene correttemente salvato

```
[#] [03-03-2022 11:33:53] Accepted connection from the server to client 93.70.241.243 with socket descriptor 4.
[#] [03-03-2022 11:33:53] Thread -974178560 handle client connection 4.
[#] [03-03-2022 11:33:54] Client 4 request is type 3: client wants to access.
[#] [03-03-2022 11:33:54] Connected to database.
[***] [03-03-2022 11:33:54] Error! Can't complete the query on the database. For more information: Duplicate entry 'angelo' for key 'PR
IMARY'
[#] [03-03-2022 11:33:54] Sent check value 1 to client 4.
```

Fig 1.9 Il client con ip 93.70.241.243 si connette al server e richiede di salvare l'username "angelo" che non viene correttemente salvato in quanto esiste già ed è quindi un duplicato

Successivamente l'utente dalla schermata Home può effettuare diverse azioni, nel caso volesse avviare una sessione di rilevazione esso richiede prima la soglia limite (richiesta effettuata attraverso il passaggio dell'intero 0 ,**Fig 1.10**), se il client rileva la presenza di una buca la segnala comunicando al server l'intero 1, le coordinate dove è presente la buca e la variazione di accelerazione lungo l'asse Y (**Fig 1.11**), qualora la buca sia già stata rilevata da un'altro utente il server non memorizza il duplicato (**Fig 1.12**).

```
[#] [03-03-2022 11:36:42] Accepted connection from the server to client 93.70.241.243 with socket descriptor 5.
[#] [03-03-2022 11:36:42] Thread -1573251328 handle client connection 5.
[#] [03-03-2022 11:36:43] Client 5 request is type 0: client requires limits value.
[#] [03-03-2022 11:36:43] Sent limit value to client 5.
```

Fig 1.10 Il client con indirizzo ip 93.70.241.243 richiede ed ottiene il valore soglia

```
[#] [03-03-2022 11:42:33] Accepted connection from the server to client 93.70.241.243 with socket descriptor 5.
[#] [03-03-2022 11:42:33] Thread 1411118848 handle client connection 5.
[#] [03-03-2022 11:42:34] Client 5 request is type 1: client is sending his position beacause has exceeded the limits values.
[#] [03-03-2022 11:42:34] These are client 5 coordinates: Latitude: 40.816123; Longitude: 14.174437. Variations in relation to limit va
lue: 1.428850.
[#] [03-03-2022 11:42:34] Connected to database.
```

Fig 1.11 Il client registra la presenza di una buca e trasmette al server la sua posizione e la variazione rilevata

```
[#] [03-03-2022 11:43:25] Client 4 request is type 1: client is sending his position beacause has exceeded the limits values.
[#] [03-03-2022 11:43:25] These are client 4 coordinates: Latitude: 40.816123; Longitude: 14.174437. Variations in relation to limit va
lue: 1.294600.
[#] [03-03-2022 11:43:25] Connected to database.
[***] [03-03-2022 11:43:25] Error! Can't complete the query on the database. For more information: Duplicate entry '40.816123-14.174437
' for key 'PRIMARY'
```

Fig 1.12 Il client rileva una buca che però è già presente nel database

L'utente può anche richiedere di visualizzare tutte le buche rilevate entro un certo raggio dalla propria posizione inviando l'intero 2 al server (Fig 1.13).

```
[#] [03-03-2022 11:44:15] Accepted connection from the server to client 93.70.241.243 with socket descriptor 5.
[#] [03-03-2022 11:44:15] Thread 1411118848 handle client connection 5.
[#] [03-03-2022 11:44:16] Client 5 request is type 2: client requires potholes in his zone.
[#] [03-03-2022 11:44:16] Connected to database.
[#] [03-03-2022 11:44:16] Query result successfully stored.
[#] [03-03-2022 11:44:16] Sent nearby holes to client 5.
```

Fig 1.13 Il client richiede la lista di tutte le buche nelle sue vicinanze

Parte II

Illustrazione del protocollo di comunicazione a livello di applicazione tra client e server

La comunicazione tra client – server viene gestita con l'implementazione di una socket TCP e di un protocollo di comunicazione a livello applicativo. Possiamo sezionare la comunicazione in 4 parti:

1. Il client effettua il suo primo accesso
2. Il client vuole avviare una sessione di registrazione per le buche e richiede il valore soglia
3. Il client rileva una buca che supera il valore soglia e vuole salvarla
4. Il client vuole visualizzare tutte le buche nelle sue vicinanze

Essendo richieste di tipo diverso, per ognuna il client formatta una stringa ben precisa. È possibile dividerla in 3 parti

- Flag, numero intero per riconoscere il tipo di richiesta
- Payload, dati che il client invia al server
- Carattere di terminazione, per tutti i tipi di richiesta è "#"

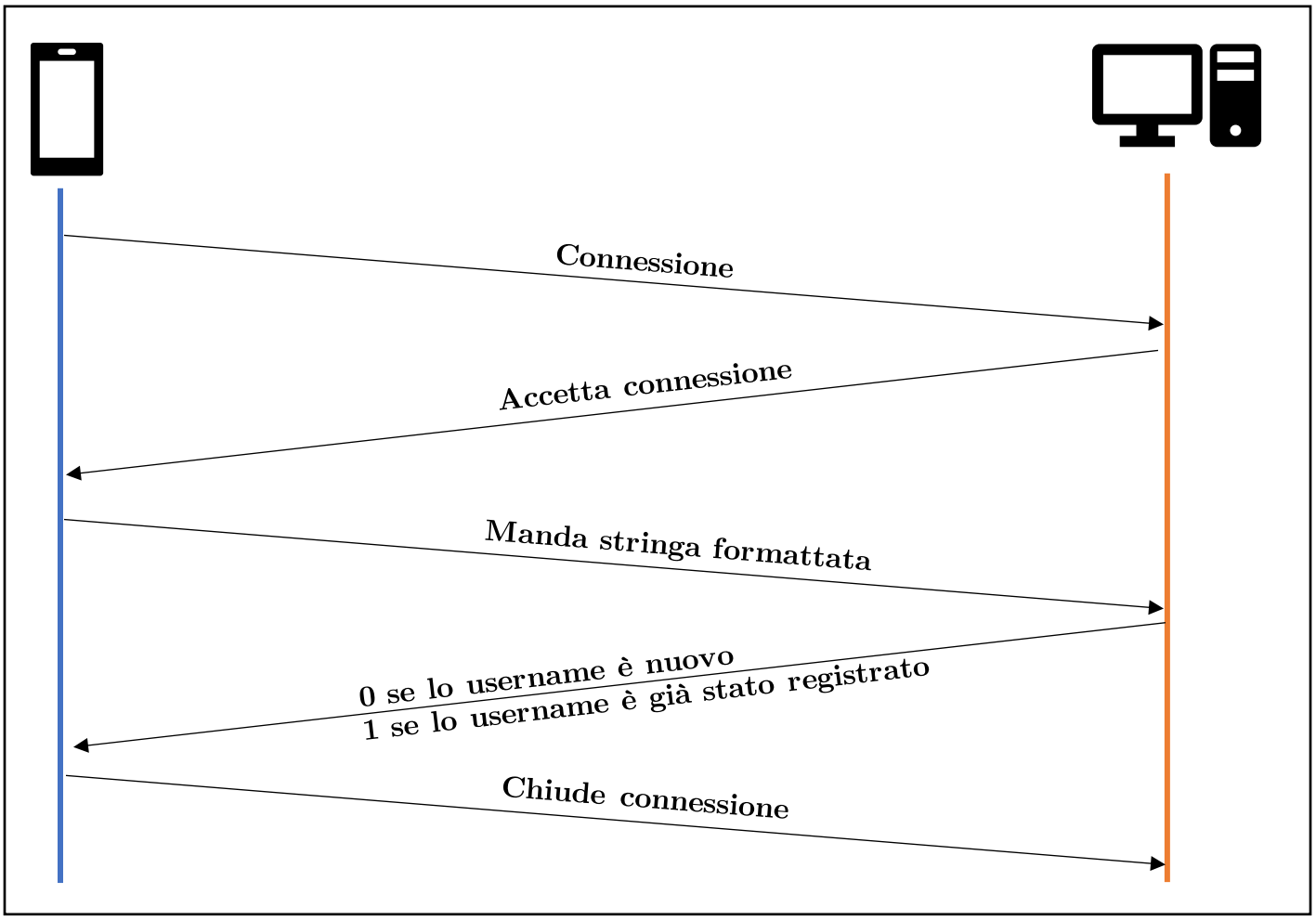
La flag può essere

- 0, se è una richiesta di tipo 2
- 1, se è una richiesta di tipo 3
- 2, se è una richiesta di tipo 4
- 3, se è una richiesta di tipo 1

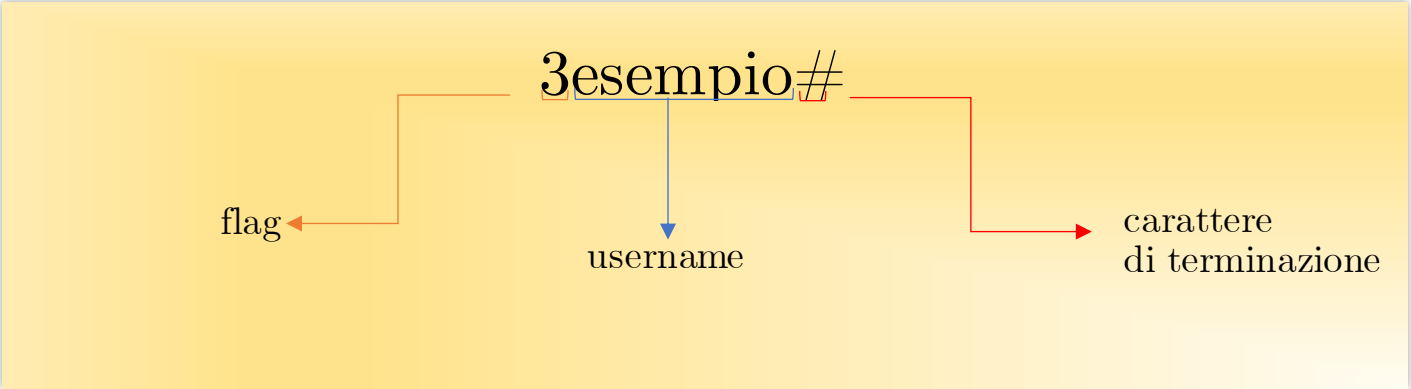
Per ogni richiesta, il server riconosce la flag e gestisce il tipo di richiesta.

N.B: Per le funzioni esplicite che formattano la stringa lato client e quelle che estraggono i dati da quest'ultima lato server, si trovano rispettivamente nei capitoli 3 e 4

1 – Primo accesso



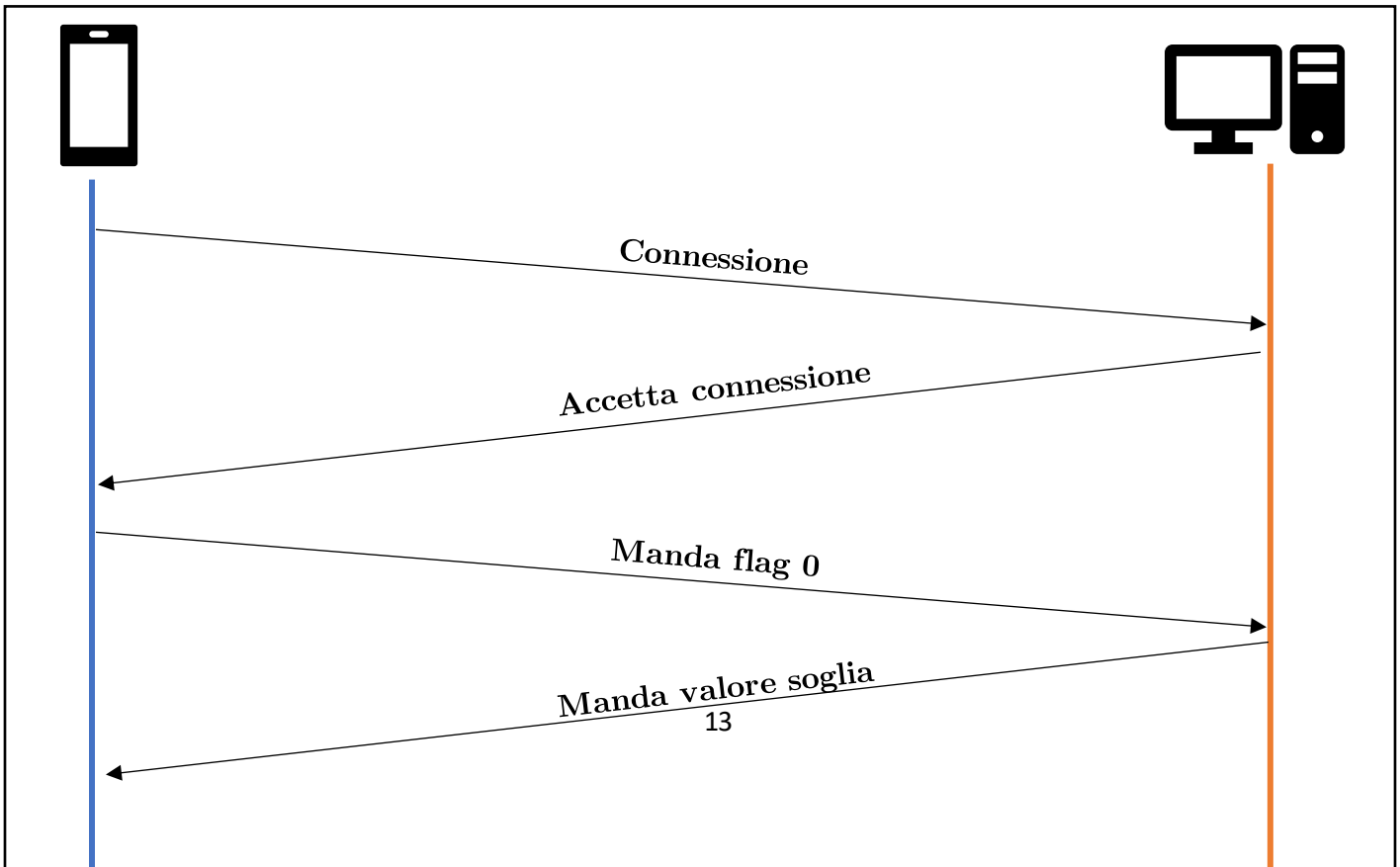
Nel caso del primo accesso, il client manda una stringa formattata in questo modo

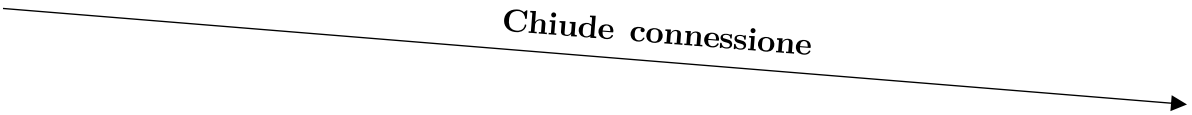


In cui

- Flag: di tipo 3 per indicare che il client sta eseguendo il primo accesso/registrazione al sistema
- Payload: username che sceglie l'utente

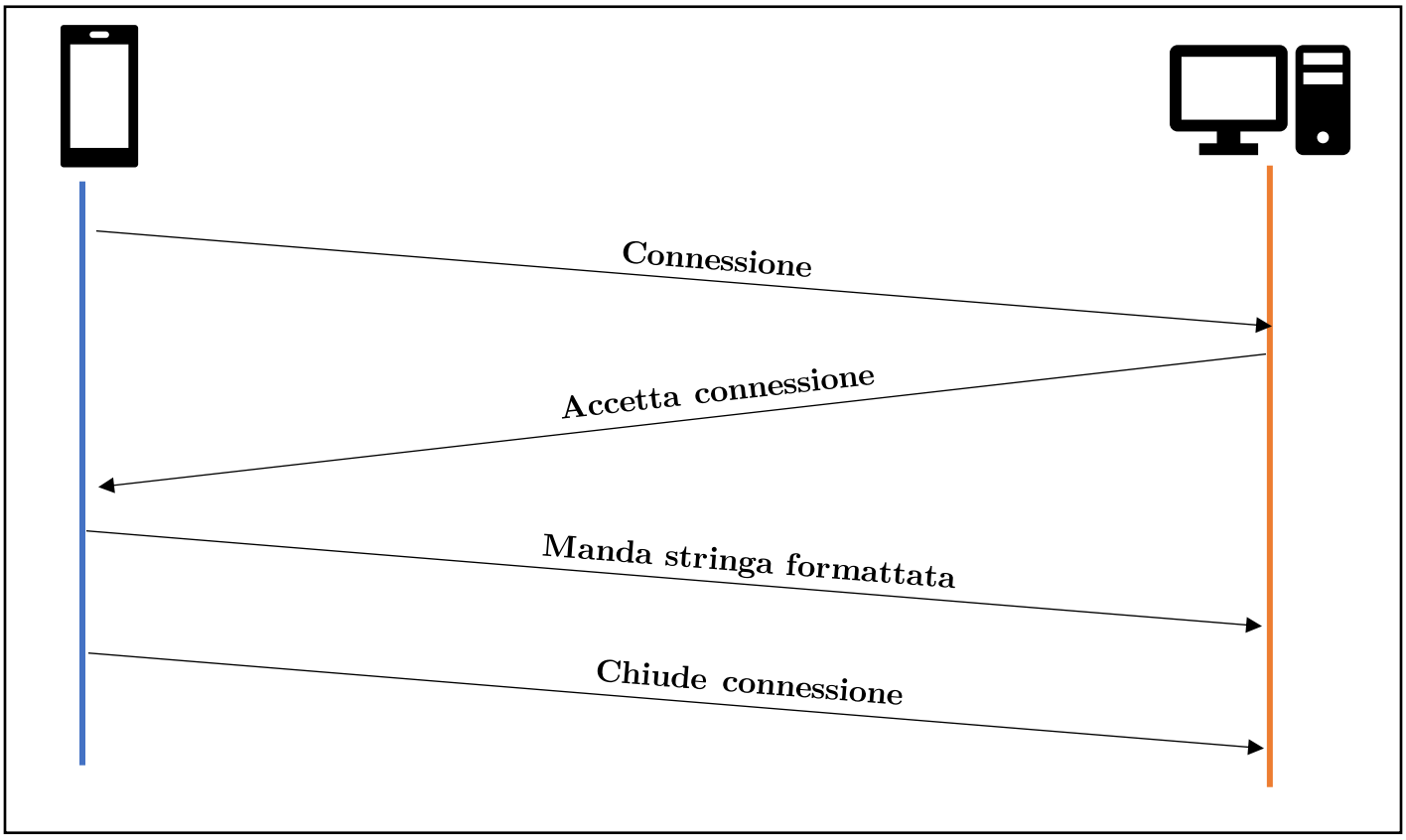
2 – Richiesta del valore soglia



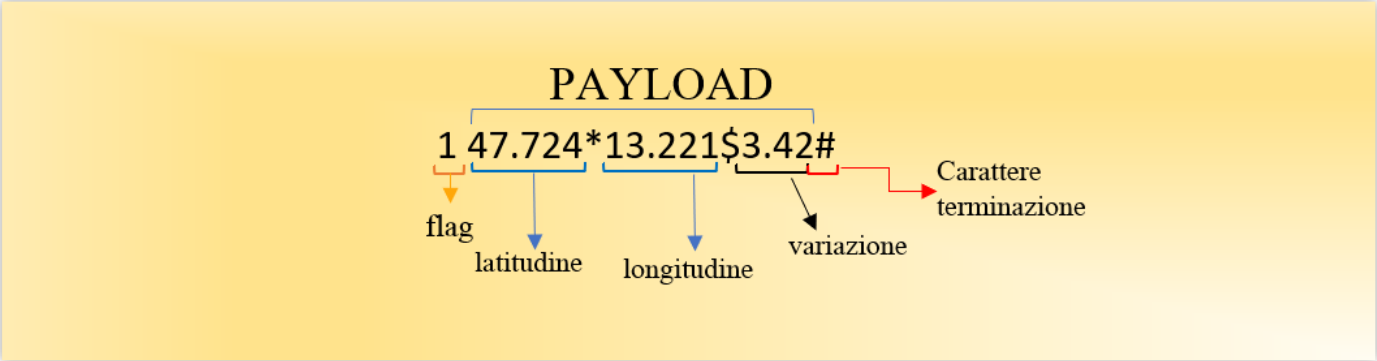


Nel caso in cui il client volesse avviare una sessione di registrazione per rilevare le buche, si instaura la connessione e si invia solamente la flag 0. Il payload è vuoto, poiché al server non servono ulteriori informazioni, e il carattere di terminazione è omesso.

3 – Salvataggio della buca rilevata



Nel caso in cui si rilevasse una buca (quindi che il valore soglia sia superato), il client manda una stringa formattata in questo modo



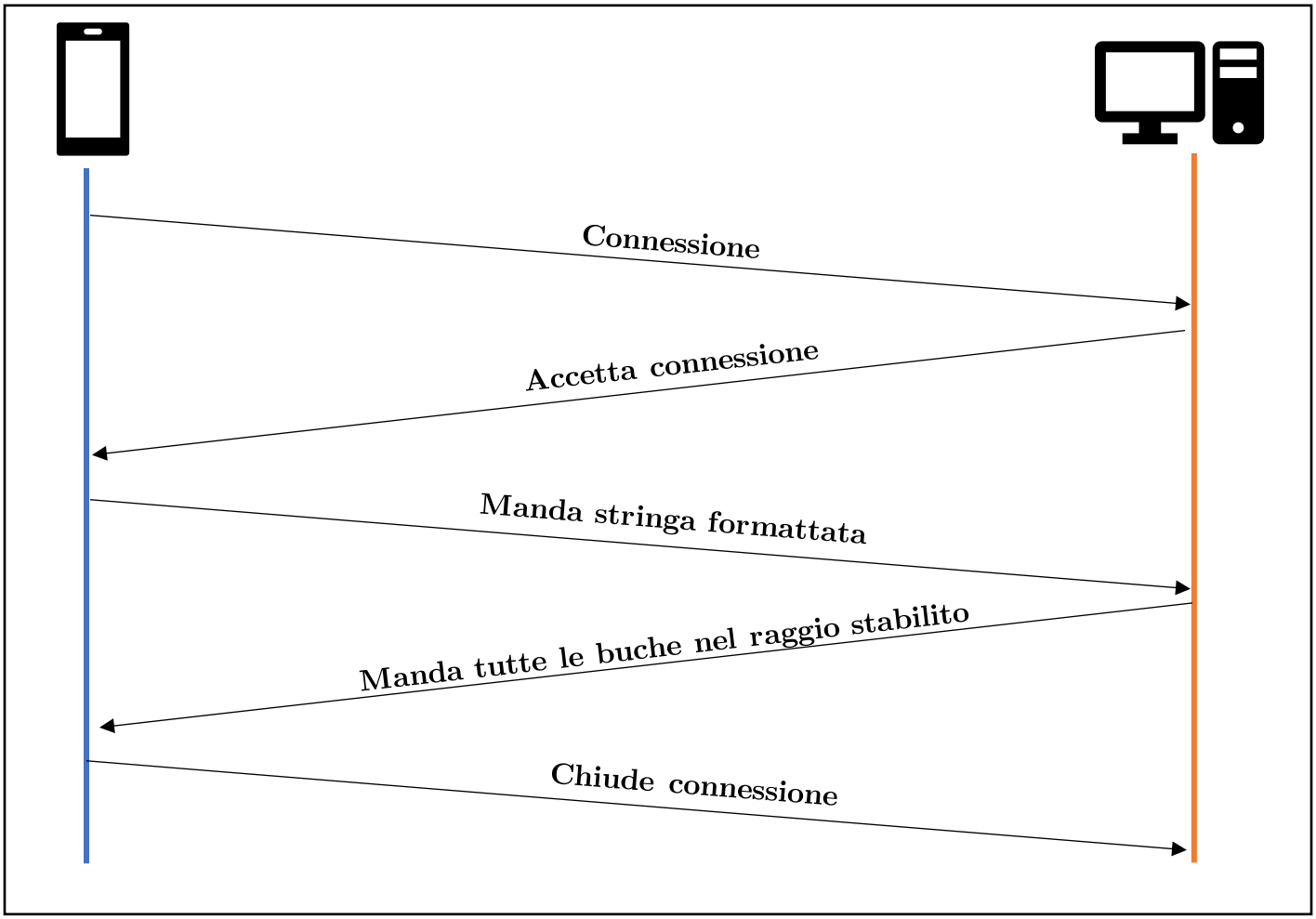
In cui

- Flag: di tipo 1 per indicare che il client sta mandando le informazioni di una buca
- Payload: composto da 3 elementi descrittivi della buca
  - Latitudine
  - Longitudine
  - Variazione di accelerazione del sensore

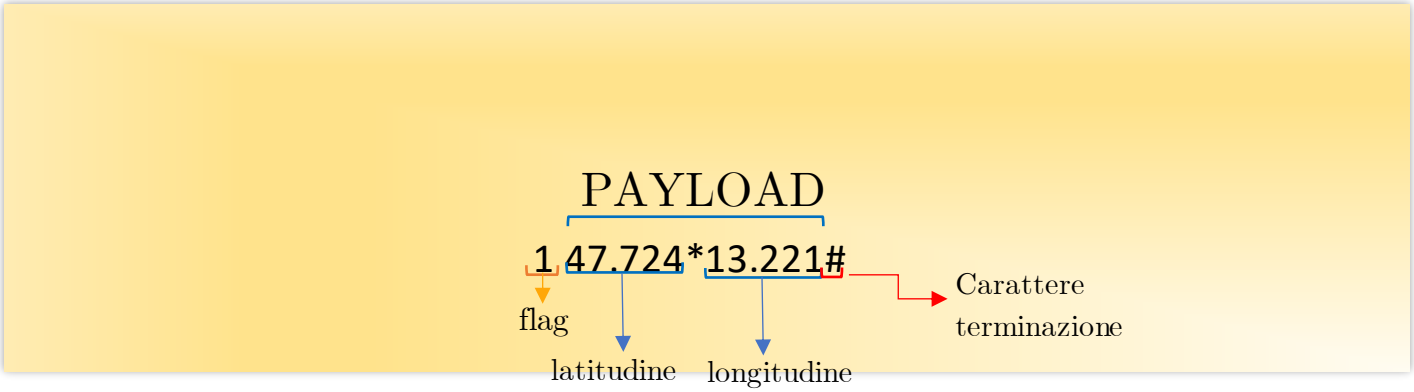
  Per distinguere i vari dati, vengono posti dei separatori

  - * tra latitudine e longitudine
  - $ tra longitudine e variazione
  - # come carattere di terminazione

4 – Visualizzazione delle buche in un raggio stabilito



Nel caso in cui il client richiedesse di visualizzare le buche nelle vicinanze, il client manda una stringa formattata in questo modo:



In cui

- Flag: di tipo 2 per indicare che il client sta mandando le informazioni sulla sua posizione attuale
- Payload: composto da 2 elementi per localizzare la posizione dell'utente
  - Latitudine
  - Longitudine

  Per distinguere i vari dati, viene posto un separatore

  - * tra latitudine e longitudine
  - # come carattere di terminazione

In questa particolare situazione anche il server risponde con una stringa formattata in JSON (come nella figura sotto) per passare efficientemente più buche al client.

```
{
    "potholes": [
        {
            "latitude": 43.724,
            "longitude": 13.221,
            "variation": 3.42
        },
        ...
    ]
}
```

Parte III

Implementazione del server

Come descritto nella sezione precedente il client e il server comunicano tramite una stringa strutturata in cui è possibile individuare diversi campi, ciò avviene tramite diverse funzioni. La funzione *void handleRequest(int, int *, char [])* (definita in utils.c e usata in handling.c) cattura la richiesta effettuata dal client (il

primo parametro della funzione) che gli comunica l'azione da intraprendere tramite la flag (il secondo parametro) che viene settato in base al primo carattere del buffer (il terzo parametro). Anche il buffer è modificato e aggiornato secondo i nuovi valori da comunicare al client.

Analizziamo ora nel dettaglio il codice della funzione che è riportato sotto

```
void handleRequest(int client_sd, int *flag, char buf[])
2   {
3       int old_index = 1, new_index = 0;
4       if(buf[0] == '0')
5          *flag = 0, printf("[#] [%s] Client %d request is type 0: client requires limits value.\n", getLogTime(),
client_sd);
6       else if(buf[0] == '1')
7       {
8           printf("[#] [%s] Client %d request is type 1: client is sending his position beacause has exceeded the
limits     values.\n", getLogTime(), client_sd);
9           *flag = 1;
10          while(buf[old_index] != '#')
11          {
12              buf[new_index] = buf[old_index];
13              new_index++;
14             old_index++;
15          }

16          buf[new_index++] = '#';
17         buf[new_index] = '\0';
18      }
19      else if(buf[0] == '2')
20    {
21          *flag = 2;
22          printf("[#] [%s] Client %d request is type 2: client requires potholes in his zone.\n", getLogTime(),
client_sd);
23         while(buf[old_index] != '#')
24          {
25              buf[new_index] = buf[old_index];
26              new_index++;
27              old_index++;
```

```
28          }
29          buf[new_index++] = '#';
30          buf[new_index] = '\0';
31      }
32      else if(buf[0] == '3')
33      {
34          *flag = 3;
35          printf("[#] [%s] Client %d request is type 3: client wants to access.\n", getLogTime(), client_sd);
36          while(buf[old_index] != '#')
37         {
38              buf[new_index] = buf[old_index];
39              new_index++;
40             old_index++;
41         }
42         buf[new_index++] = '\0';
43   }
44   else
45       fprintf(stderr, "[***] [%s] Error! String is not formatted properly. Try again.\n", getLogTime());
46   }
```

Come parametri la funzione riceve un intero per identificare il client, la flag e il buffer dal quale leggere la stringa per decodificare l'azione da intraprendere e nel quale memorizzare i valori da passare al client. Vengono quindi definite delle variabili locali che fungono da indici per analizzare il buffer (riga 3). La funzione controlla il primo carattere della stringa per stabilire la richiesta al server (righe 4 - 6 – 19 – 32)

se il primo carattere è 0 allora il client richiede il valore soglia poiché vuole avviare la sessione di rilevamento buche, viene quindi settata la flag a 0 (riga 5).

Se, invece, il primo carattere della stringa è 1, il client vuole inviare al server i dati di una buca appena rilevata, quindi entra in un ciclo while dove copia nello stesso buffer la posizione della buca e la variazione rilevata. La funzione copia i caratteri uno ad uno finchè non raggiunge il carattere finale della stringa da

copiare che viene successivamente posto a '#', mentre al carattere finale viene assegnato '\0' (righe 6-18).

Nel caso in cui il carattere iniziale sia 2 o 3, la richiesta è diversa ma i meccanismi di soddisfacimento sono gli stessi, se invece non è nessuno dei precedenti la funzione dà un errore.

Dopo aver settato la flag in *handling.c* verrà richiamata una funzione specifica. Ad esempio se il client vuole ricevere le buche da visualizzare (flag = 2) viene richiamata la funzione *void sendHoles(int, char [])* che riceve la posizione del client, recupera dal database tutte le buche nelle vicinanze e le mostra al client, ciò avviene grazie alle funzioni richiamate all'interno di quest'ultima, le più interessanti sono *void extractPosition(clientData **, char [])* e *void createJSON(clientData **, char [])* utili per comunicare al server la posizione del client e per creare la stringa Json da trasmettere al client da cui legge le buche.

```
1 void extractPosition(clientData **position, char buf[])
2 {
3      int buf_index = 0, i;
4       char latitude[MAXBUFF], longitude[MAXBUFF];
5       for(i = 0; buf[buf_index] != '*'; i++, buf_index++)
6           latitude[i] = buf[buf_index];

7       latitude[i] = '\0';
8       buf_index++;

9       for (i = 0; buf[buf_index] != '#'; i++, buf_index++)
10          longitude[i] = buf[buf_index];

11      longitude[i] = '\0';
12      *position = createNode(latitude, longitude, "0");
13  }
```

Come parametri la funzione riceve in ingresso un doppio puntatore a un nodo sul quale scrivere i valori di latitudine e longitudine che permettono la localizzazione del client, e una stringa buffer da cui è possibile ricavarne i valori. Vengono quindi create delle variabili locali per memorizzare i valori di latitudine e longitudine e per tenere l'indice della stringa ricevuta come parametro (righe 3 e 4). Come illustrato nella sezione precedente la stringa è strutturata in modo che i diversi valori del payload sono separati da un asterisco ('*') e la fine è segnalata dal carattere '#'. Quindi per estrarre latitudine e longitudine si implementano due cicli for. Il primo copia nella variabile locale *latitude* il valore della latitudine ciclando dal primo carattere la stringa ricevuta in ingresso, e incrementando entrambi gli indici, quindi viene posto l'ultimo carattere di *latitude* uguale a '\0' (righe 5-8) . Allora inizia il secondo ciclo che funziona allo stesso modo del primo ma assegna i valori della logitudine ricavati dalla stringa in ingresso alla variabile locale dedicata, con la differenza che il ciclo si interrompe quando il carattere della stringa in ingresso che si sta ciclando corrisponde a '#' (righe 9-11) . Dopodichè viene creato un nodo in cui si salvano i valori di latitudine e longitudine. Questi cicli vengono usati in tutte le funzioni in cui si devono estrarre i valori dalla stringa in ingresso, ad esempio la funzione *void extractHole(clientData **, char [])* usata per inviare i dati delle buche rilevate al server con l'aggiunta del valore della variazione che è diviso dalla longitudine con il carattere '$'.

È stato scelto il Json come formato del messaggio da inviare al client per facilitare l'invio e l'estrazione di molti elementi dello stesso tipo.

Vediamo ora il codice della funzione *createJson:*

```
void createJSON(clientData **holes, char json_string[])
2   {
3       char tmp[1000];
4      if(*holes != NULL)
5      {
       //Salva una buca come elemento {"lat":??,"lon":??,"variation":??}
6        sprintf(tmp, "{\"lat\":%f,\"lon\":%f,\"var\":%f}", (*holes)->latitude, (*holes)->longitude, (*holes)->variation);

        //Se il prossimo non è l'ultimo elemento della lista continua a concatenare elementi con la virgola
7        if((*holes)->next != NULL)
8              strcpy(tmp, strcat(tmp, ","));
9          else
10         {
11             strcpy(tmp, strcat(tmp, "]}")); //Se prossimo elemento della lista è ultimo allora chiude la stringa JSON
12             tmp[strlen(tmp)] = '\0';
13        }
14         strcpy(json_string, strcat(json_string, tmp));
15         createJSON(&((*holes)->next), json_string);
16     }
17  }
```

La funzione riceve in ingresso un doppio puntatore a una lista di nodi dal quale ricavare i valori delle buche e una stringa nella quale sarà memorizzata la stringa Json finale. La funzione controlla che il nodo a cui sta puntando non sia nullo (riga 4) se non lo è, copia i valori di latitudine, longitudine e variazione nella stringa *tmp* formattandola in Json (riga 6). Fa poi un controllo sul valore dell'elemento successivo puntato dal nodo corrente(riga 7): Se esso non è nullo, concatena una virgola poichè sarà aggiunto un altro oggetto al Json ed è

necessario separarli (riga 8), altrimenti concatena '] }' che indica la fine del Json e aggiunge '\0' (righe 11-12). Concatena poi l'elemento salvato nel *tmp* alla stringa *json_string*, e salva la stringa risultante in *json_string* (riga 14), la funzione richiama quindi se stessa passando come parametri il nodo successivo e *json_string*.

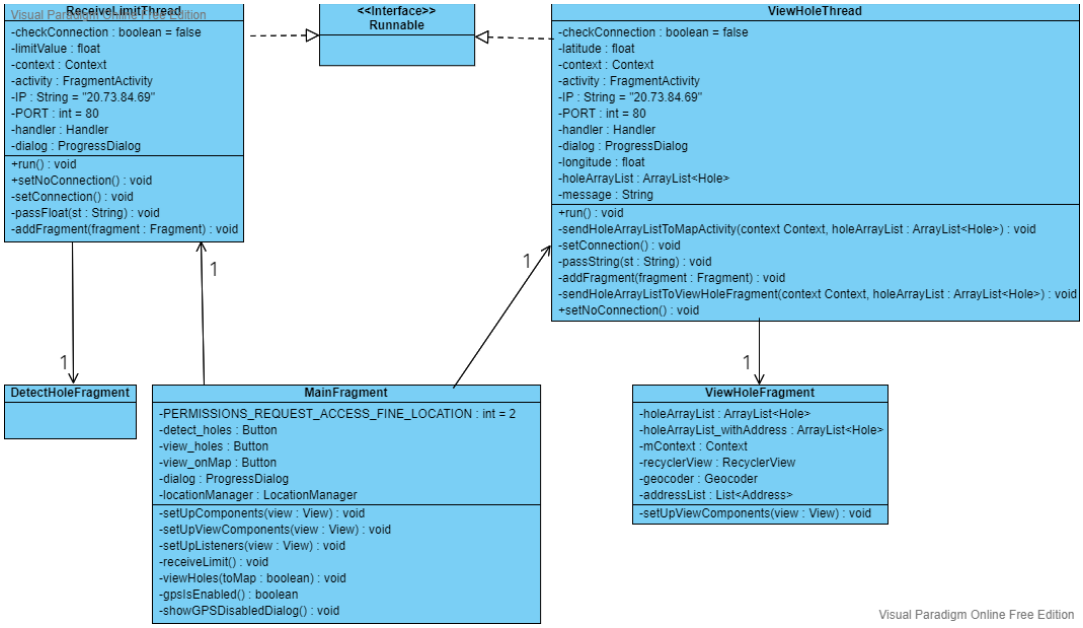
Parte IV

Implementazione del client

I dettagli implementativi più interessanti per quanto riguarda il client sono:

- Thread per gestire le comunicazioni tramite le socket
- Sensoristica di Android
- Servizi di geocoding e localizzazione
- Utilizzo del Json per facilitare l'estrazione di informazioni ricevute dal server
- Utilizzo del RecyclerView

Tutte le comunicazioni vengono gestite da tre classi: *ReceiveLimitThread, SendHoleThread, ViewHolesThread.* Tutte queste classi implementano l'interfaccia *Runnable.*



ReceiveLimitThread richiede il valore soglia al server in questo modo

```
@Override
   public void run() {
       try {
           Socket s = new Socket();
           s.connect(new InetSocketAddress(IP, PORT), 1000);
           setConnection();
           PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(s.getOutputStream())), true);
           out.println("0");
           BufferedReader fromServer = new BufferedReader(new InputStreamReader(s.getInputStream()));
           final String buffer = fromServer.readLine();
           passFloat(buffer);
           s.close();
       } catch (IOException /*| InterruptedException*/ e) {
           setNoConnection();
           e.printStackTrace();
       }
```

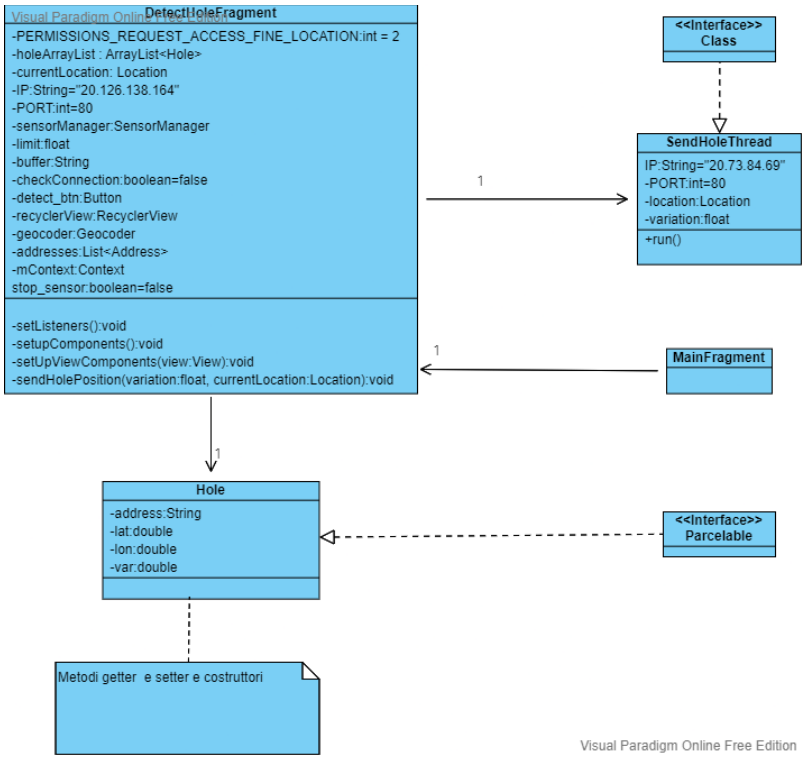
Una volta effettuata la connessione al server tramite l'utilizzo della classe *Socket* , invia la richiesta per ottenere il valore soglia (cioè il flag 0), e tramite il *BufferedReader* salva in una stringa il valore soglia ricevuto dal server che verrà poi salvato. Le azioni da intraprendere se si è ricevuta la soglia o in caso di errore sono gestite da un *handler*

```
 handler.post(new Runnable() {
            @Override
            public void run() {
                //Se la connessione al server è stata effettuata correttamente allora passa a rilevare le buche
                if(checkConnection) {
                    MotionToast.Companion.darkToast(activity, "","Soglia ricevuta: "+limitValue,
MotionToastStyle.SUCCESS,MotionToast.GRAVITY_BOTTOM, MotionToast.LONG_DURATION, ResourcesCompat.getFont(context,
R.font.helveticabold));

                    dialog.dismiss();
                    Bundle bundle = new Bundle();
                    bundle.putFloat("limit", limitValue);
                    DetectHoleFragment detectHoleFragment = new DetectHoleFragment();
                    detectHoleFragment.setArguments(bundle);
                    addFragment(detectHoleFragment);
                } else {
                    dialog.dismiss();
                    MotionToast.Companion.darkToast(activity, "Errore","Connessione al server non riuscita.\nRiprova più
tardi.", MotionToastStyle.ERROR,MotionToast.GRAVITY_BOTTOM, MotionToast.LONG_DURATION, ResourcesCompat.getFont(context,
R.font.helveticabold));
                }
            }
        });
    }
```

Il valore soglia ricevuto si invia tramite Bundle al DetectHoleFragment che si occupa di gestire l'accelerometro per la rilevazione delle buche. Se ci sono stati errori,invece,verrà mostrato un *MotionToast* (libreria di GitHub per mostrare Toast colorati).

DetectHoleFragment si occupa della rilevazione delle buche e invia i dati riguardo ad esse al *SendHoleThread.*



Il metodo che resta in ascolto di cambiamenti di accelerazione del sensore è questo:

```
@Override
    public void onSensorChanged(SensorEvent sensorEvent) {
        float y = sensorEvent.values[1];
        if (!stop_sensor) {
            if (y > limit) {
                try {
                   Task<Location>taskLocation=LocationServices.getFusedLocationProviderClient(mContext).getLastLocation();
                   taskLocation.addOnCompleteListener(task -> {
                        //sensorManager.unregisterListener(this);
```

```
                        currentLocation = task.getResult();
                        if (currentLocation != null) {
                            sendHolePosition((z - limit), currentLocation);

                            try {
                                addresses = geocoder.getFromLocation(currentLocation.getLatitude(),
                                            currentLocation.getLongitude(), 1);
                            } catch (IOException e) {
                                e.printStackTrace();
                            }
                            hole = new Hole(addresses.get(0).getAddressLine(0), z - limit);
                            holeArrayList.add(hole);
                            RecyclerAdapter recyclerAdapter = new RecyclerAdapter(holeArrayList);
                            recyclerView.setItemAnimator(new DefaultItemAnimator());
                            recyclerView.setAdapter(recyclerAdapter);

                        } else {
                            MotionToast.Companion.darkToast(activity, "Errore","La posizione corrente è nulla",
MotionToastStyle.ERROR,MotionToast.GRAVITY_BOTTOM, MotionToast.LONG_DURATION, ResourcesCompat.getFont(context,
R.font.helveticabold))
}});
        } catch (SecurityException e) {e.printStackTrace();
}
}
```

Il metodo *onSensorChanged* resta in ascolto per eventuali cambiamenti di accelerazione sull'asse y. Se il sensore rileva un accelerazione che supera il valore soglia precedentemente comunicato dal server , viene richiesta la posizione corrente con il *FusedLocationProvider* , e con il metodo *sendHolePosition* verrà inviata la variazione sull'asse y e la posizione corrente della buca. Il *RecyclerView* posizionerà la buca appena rilevata secondo un pattern definito in precedenza e ottimizza l'utilizzo della memoria e di conseguenza della batteria del dispositivo, perchè riciclerà le *view* nelle quali saranno posizionate le buche. Tramite la

posizione corrente e il *Geocoder* si può trovare l'indirizzo di dove è stata rilevata la buca.

Il metodo sendHolePosition che invierà la buca al *SendHoleThread è il seguente:*

```
private void sendHolePosition(float variation, Location currentLocation) {
        Thread rec = new Thread(new SendHoleThread(currentLocation, variation));
        rec.start();
    }
```

Il metodo principale della classe *SendHoleThread è il seguente:*

```
@Override
    public void run() {
        try {
            Socket s = new Socket(IP, PORT);
            PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(s.getOutputStream())), true);
            out.println("1" + location.getLatitude() + "*" + location.getLongitude() + "$" + variation + "#");
            s.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Invia al server la buca secondo il pattern definito nel protocollo di comunicazione.

Quando l'utente vuole visualizzare tutte le buche nelle sue vicinanze viene richiamata la classe *ViewHolesThread.*

```
@Override
    public void run() {
        try {
```

```
            Socket s = new Socket();
            //Creo la socket con un timeout di 2 secondi per accorciare i tempi di ricezione di un'eventuale eccezione
            s.connect(new InetSocketAddress(IP, PORT), 1000);
            //Connessione effettuata correttamente
            setConnection();
            //Prima connessione e invio della richiesta da parte del client
            PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(s.getOutputStream())), true);
            //Formattazione richiesta
            out.println("2"+latitude+"*"+longitude+"#");

            BufferedReader fromServer = new BufferedReader(new InputStreamReader(s.getInputStream()));
            final String buffer = fromServer.readLine();
            passString(buffer);
            s.close();
        } catch (IOException /*| InterruptedException*/ e) {
            e.printStackTrace();
            setNoConnection();
}
```

Effettuata la connessione verrà inviata la richiesta al server per ottenere tutte le buche entro il raggio prefissato di 30 km dalla propria posizione. Riceverà la risposta dal server che sarà una stringa formattata in json.

```
handler.post(new Runnable() {
            @Override
            public void run() {
                //Se la connessione al server è stata effettuata correttamente allora passa a rilevare le buche
                if(checkConnection) {
```

```
                    holeArrayList = new ArrayList<>();
                    holeArrayList = parseJSON();
                    if (toMap == false)
                        sendHoleArrayListToViewHoleFragment(context,holeArrayList);
                    else
                        sendHoleArrayListToMapActivity(context,holeArrayList);
                    dialog.dismiss();
                } else {
                    MotionToast.Companion.darkToast(activity, "Errore","Connessione al server non riuscita.\nRiprova più
tardi.", MotionToastStyle.ERROR,MotionToast.GRAVITY_BOTTOM, MotionToast.LONG_DURATION, ResourcesCompat.getFont(context,
R.font.helveticabold));

                }
            }
        });
```

Nell'handler si gestiscono le operazioni una volta ricevuto il json. Verrà costruito un *ArrayList* di oggetti di tipo *Hole* che verrà poi passato, con *Intent* o *Bundle*, tramite due metodi, all'activity che gestisce la visualizzazione di buche su mappa o al fragment che tramite *RecyclerView* mostrerà le buche ricevute dal server.

Il parsing della stringa Json ritornata dal server viene effettuato tramite il metodo parseJson che ritorna una ArrayList di Hole:

```
private ArrayList<Hole> parseJSON() {
        ArrayList<Hole> holes = new ArrayList<>();
```

```
        try {
            JSONObject myJSON = new JSONObject(message);
            JSONArray holesJSON = myJSON.getJSONArray("potholes");
            int size = holesJSON.length();

            for (int i = 0; i < size; i++) {
                JSONObject singleHole = holesJSON.getJSONObject(i);
                holes.add(new Gson().fromJson(String.valueOf(singleHole), Hole.class));
            }
        } catch (JSONException e) {
            e.printStackTrace();
        }
        return holes;
    }
```

Estrae le informazioni necessarie per costruire un oggetto di tipo Hole dopodichè aggiunge quest'ultimo nell'ArrayList.