PuliBO

Applicazione per la verifica della pulizia stradale nella città di Bologna

Di

Ivan Paolo Compagnoni

Introduzione

Oggi è sempre più difficile avere tutto il tempo che ci serve per fare quello che dovremmo e quindi non è raro dimenticare qualcosa. Ci sono casi in cui queste dimenticanze possono però costar caro: in una città grande come Bologna può succedere, muovendosi in auto, di dover parcheggiare in una zona non conosciuta. Nella frenesia di trovare un parcheggio (impresa tutt'altro che semplice) non è raro non accorgersi che nella strada dove stiamo parcheggiando è prevista una pulizia magari a qualche ora dal momento del parcheggio.

Con l'idea di porre rimedio a queste dimenticanze nasce "PuliBO" un'applicazione che permette di sapere se nel luogo dove si è parcheggiato è prevista o meno una pulizia stradale che ci costerebbe la rimozione del veicolo.

Gli obiettivi che ci poniamo nel realizzare questa applicazione sono molteplici. L'app deve:

- Essere in grado di reperire le informazioni sulle strade dove è prevista la pulizia
- Aggiornare in tempo reale l'utilizzatore della propria posizione
- Accorgersi in maniera automatica se l'auto è parcheggiata o meno
- Stabilire se in un range di due ore avverrà una pulizia della strada
- Ricordare all'utilizzatore di rimuovere l'auto da una strada dove avverrà la pulizia con un certo preavviso
- Comunicare a persone vicine l'imminente pulizia

Di seguito sono riportate alcune scelte fatte all'interno di questo progetto:

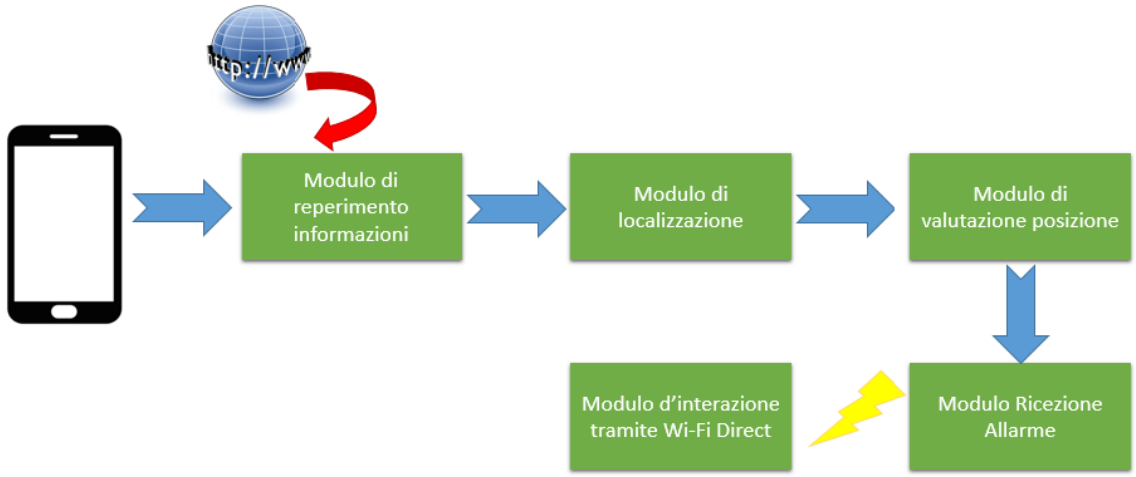
- Le informazioni riguardanti le varie pulizie stradali previste saranno reperibili tramite sito web sotto forma di file XML
- La comunicazione con utenti vicini avverrà tramite Wi-Fi Direct

Architettura logica

L'applicazione è scomposta in vari moduli che interagiscono tra di loro, creando in questa maniera un'architettura che produce il comportamento finale desiderato.

I diversi moduli sono il risultato di alcune necessità imprescindibili per la nostra applicazione. L'utente muovendosi liberamente per la città deve essere informato di dove si trova, in modo da poter avere le informazioni relative alla pulizia della strada nella quale ha parcheggiato: ecco quindi che emerge la necessità di un modulo di localizzazione che faccia in modo di recuperare l'attuale posizione dell'utente. Tale posizione può essere catturata tramite il servizio di locazione attualmente disponibile sul dispositivo, sia esso GPS o rete Wi-Fi per fare un esempio.

L'informazione sulla posizione è tipicamente espressa tramite valori di latitudine e longitudine ed è, quindi, poco "leggibile" per un utente umano. Siamo inoltre interessati a sapere più che l'esatta posizione dell'utilizzatore, la via nel quale esso si trova. Appare quindi evidente che si debba pensare di usare un modulo che, acquisita l'informazione sulla posizione corrente, restituisca l'informazione riguardante il nome della via ad esso associata. Nella figura sottostante questo modulo è indicato con il nome di "Modulo di Valutazione Posizione". Oltre a restituire il nome della via corrente questo modulo si occupa anche di controllare se in tale via sia programmata una pulizia da lì a poco.



In caso la pulizia non sia in corso in quel momento ma prevista a breve, vogliamo fare in modo che l'utente sia notificato nuovamente dell'imminente inizio della pulizia in modo che possa mobilitarsi per spostare la sua autovettura. Prevediamo quindi un modulo che sia adibito a questa funzione indicandolo con "Modulo Ricezione Allarme".

Un altro punto interessante della nostra applicazione è quello di dare all'utente la possibilità di mandare un messaggio ad altri utenti vicini a lui in quel momento in maniera di avvisarli dell'imminente pulizia. Per far ciò sfrutteremo il Wi-Fi Direct e inseriamo quindi un "Modulo d'Interazione tramite Wi-Fi Direct".

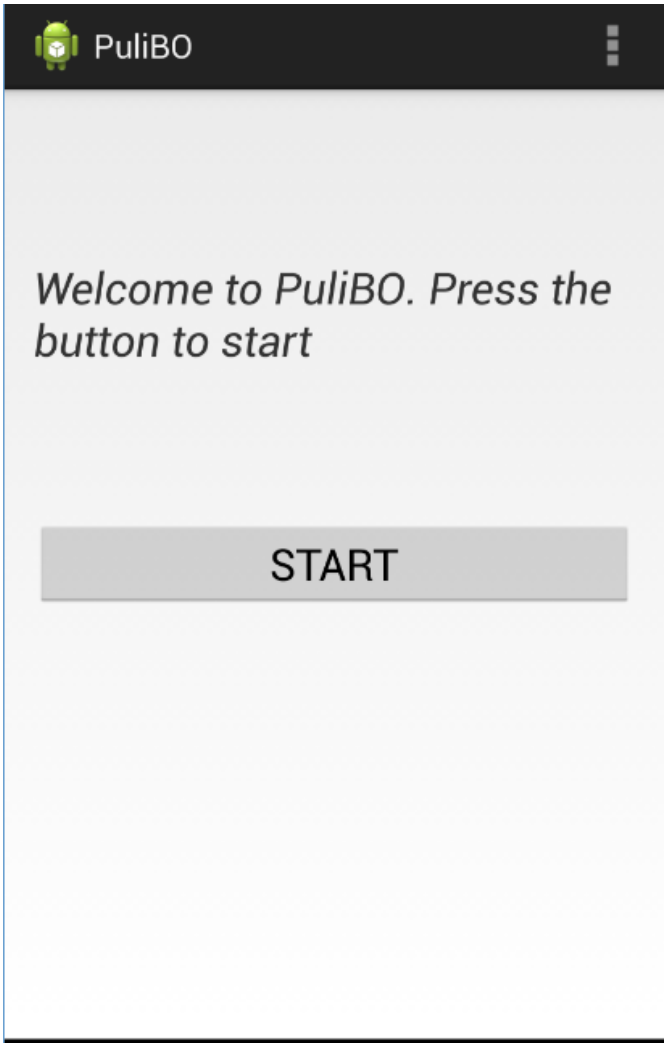
Le varie informazioni di posizione dell'utente dovranno essere confrontate con quelle presenti in un file XML disponibile per il download da Internet: tale file deve essere quindi scaricato in maniera da confrontare le informazioni di posizionamento attuale con le vie salvate in esso.

Naturalmente questi sono solo i moduli che vengono fuori da un'analisi logica del problema che vogliamo affrontare, ma non sono esaustivi di tutti gli aspetti dello stesso ed è per questo che nella versione finale sono stati previsti ulteriori moduli per curare aspetti più "pratici" della gestione dell'applicazione.

Analizzeremo di seguito l'applicazione parlando delle varie schermate di interazione con l'utente, illustrando quali moduli sono utilizzati in quest'ultime e descrivendo come sono composti.

Schermata di avvio

La prima schermata d'interazione con l'utente presenta un messaggio di benvenuto e un bottone che permette all'utente di iniziare ad utilizzare l'applicazione "PuliBO" ed è chiamata MainActivity.



Già in questo primo stadio dell'applicazione è possibile fare delle scelte progettuali importanti: infatti si è deciso di prevedere qui il download del file XML contenente le informazioni sulla pulizia delle strade. Una volta scaricato le informazioni di questo file verranno salvate su di un database SQL in maniera da non doverle reperire ogni volta. Android per far ciò utilizza SQLite una particolare libreria che permette di creare una base di dati incorporata in un unico file. Grazie alle ridotte dimensioni, SQLite è particolarmente adatto ai sistemi embedded ed è per questo utilizzata come standard in Android.

Il file XML da scaricare presenta la seguente struttura:

```
<?xml version="1.0" encoding="UTF-8"?>
<streets>
    <street>
        <name>del Timavo</name>
        <sTime>23.30</sTime>
```

```
        <eTime>04.00</eTime>
    </street>
    <street>
        <name>Ugo Bassi</name>
        <sTime>18.30</sTime>
        <eTime>20.30</eTime>
    </street>
    <street>
        <name>Emilia ponente</name>
        <sTime>01.00</sTime>
        <eTime>05.30</eTime>
    </street>
    <street>
        <name>del Risorgimento</name>
        <sTime>06.15</sTime>
        <eTime>06.45</eTime>
    </street>
</streets>
```

Viene effettuato dapprima un controllo per vedere se questo è il primo utilizzo dell'applicazione: in caso affermativo viene settato un flag che farà in modo di far partire un thread per il download del file; quando si vuole salvare una piccola collezione di valori in Android si possono utilizzare le API relative alle SharedPreferences, un oggetto che punta ad un file contente coppie del tipo dato-valore avendo così un modo pratico per accedervi. Ciò viene effettuato invocando

```
SharedPreferences sharedPref = this.getSharedPreferences(SHARED_PREF,
                    MainActivity.MODE_PRIVATE);
```

Successivamente si ha una chiamata:

```
firstUse = sharedPref.getBoolean("firstUse", true);
```

che permette di accedere al campo firstUse della nostra sharedPreference e recuperare così il suo valore. Se esso sarà true (che viene imposto come valore di default) si salvano alcune informazioni per la gestione dei successivi utlizzi e si aggiorna il campo firstUse. Nel dettaglio:

```
// First use of the application saving date info
SharedPreferences.Editor editor = sharedPref.edit();

// Putting date info in sharedPreferences
long mOldDate = GregorianCalendar.getInstance().getTime().getTime();
editor.putLong("old_date_value", mOldDate);

// Setting firstUse and updated (to force download) to false
editor.putBoolean("firstUse", false);
editor.putBoolean("updated", false);
editor.commit();
```

Per accedere e modificare i dati all'interno di una sharedPreference è bene utilizzare un editor che ci viene fornito dal metodo edit sulla preferenza stessa. Questo ci permette tramite le invocazioni di putBoolean, di inserire nuovi valori relativi ai campi firstUse, da settare a false in quanto l'applicazione sta per essere utilizzata per la prima volta, e updated. Questo secondo campo è un valore che serve per vedere se è trascorso un determinato periodo di tempo dall'ultimo download: le informazioni nel file XML possono difatti cambiare ed è quindi buona norma scaricarne una nuova copia di tanto in tanto. Occorre quindi ad ogni utilizzo controllare anche se questo periodo sia passato e tenerne traccia nella preference nel campo "old_date_value": ovviamente in caso di primo utilizzo l'informazione del tempo espressa in millisecondi sarà salvata in modo da poterla utilizzare per futuri confronti.

L'informazione relativa al periodo di update è anch'essa mantenuta nella sharedPreference

```
// Shared Preference to keep track of the update period
updatePeriod = sharedPref.getLong("updatePeriod",WEEK_IN_MILLISECONDS);
```

In caso il periodo di update sia trascorso viene creato un task per il download del file:

```
if (updated == false)
{
	// Creating a Task to downlaod the XML file
	DownloadFile downloadFile = new DownloadFile();
	downloadFile.execute(getString(R.string.download_url));

	// Updating preferences
	SharedPreferences.Editor editor = sharedPref.edit();
	editor.putBoolean("updated", true);
	editor.putLong("old_date_value", mActualDate);
	editor.commit();
}
```

La classe DownloadFile è una classe privata definita all'interno della MainActivity la cui signature è

```
private class DownloadFile extends AsyncTask<String, Integer, Void>
```

Un AsyncTask (abbreviazione di asynchronus task) è definito da un processo che gira in background ed i cui risultati sono visualizzati sul thread di UI. Un Async Task è definito da tre tipi generici chiamati Params, Progress e Result e 4 passi chiamati onPreExecute, doInBackground, onProgressUpdate e onPostExecute. In questo caso specifico il metodo onPreExecute non viene utilizzato mentre è previsto un dato di tipo stringa per il metodo doInBackground, un dato di tipo Integer per il metodo onProgressUpdate e nessun tipo di ritorno dal metodo onPostExecute. Infatti nel metodo doInBackground è svolto il download del file vero e proprio e la stringa passata in ingresso rappresenta l'indirizzo da cui reperire il file. Il metodo onProgressUpdate permette di mostrare una barra di avanzamento del download mentre in onPostExecute prevediamo il salvataggio vero e proprio delle informazioni dal file appena scaricato, che viene salvato nella memoria del dispositivo, al DB. Questa strategia è stata pensata perché in caso di DB molto grandi potrebbero essere pubblicate le sole modifiche rispetto ad una versione precedente e non l'elenco completo delle pulizie nelle strade:

```
Log.d(DEBUGTAG, "onPostExecute method starting.");
progressDialog.dismiss();
String sdCard =Environment.getExternalStorageDirectory().getAbsolutePath();
Log.d(DEBUGTAG, "Sd card path: " + sdCard);
String filePath = "/Android/data/it.unibo.android.pulibo/files/"
                  + getString(R.string.filename);
Log.d(DEBUGTAG, "File path: " + filePath);
try
{
	Log.d(DEBUGTAG, "Opening the file...");

	File file = new
	File(Environment.getExternalStorageDirectory().getAbsolutePath(),
                                filePath);
	InputStream inputStream = new FileInputStream(file);
	Log.d(DEBUGTAG, "Saving streets info...");
	saveStreetInfo(inputStream);
	Log.d(DEBUGTAG, "Streets saved");
} catch (FileNotFoundException e)
{
	Log.e(DEBUGTAG, e.getMessage());
```

```
			e.printStackTrace();
		} catch (Exception e)
		{
			Log.e(DEBUGTAG, e.getMessage());
			e.printStackTrace();
		}
```

Possiamo ottenere informazioni riguardo al path della sdCard tramite la classe Environment ed il metodo getAbsolutePath. Una volta ottenuto lo stream da cui leggere le informazioni appena scaricate invochiamo il metodo saveStreetInfo per inserirle nel database

```
private void saveStreetInfo(InputStream inputStream) throws Exception
{
	DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
	DocumentBuilder dBuilder;

	// Opening Datasource to manage DB operations
	dataSource = new StreetsDataSource(this);
	dataSource.open();

	// Deleting all entries (first version, can be an update)
	dataSource.deleteAll();

	try
	{
		dBuilder = dbFactory.newDocumentBuilder();
		Document doc = dBuilder.parse(inputStream);

		NodeList nList = doc.getElementsByTagName("street");

		for (int i = 0; i < nList.getLength(); i++)
		{
			Node node = nList.item(i);
			if (node.getNodeType() == Node.ELEMENT_NODE)
			{
				// Inserting new element into the DB
				dataSource.insertStreet(node);
			}
		}
	} catch (Exception e1)
	{
		Log.e(DEBUGTAG, e1.getMessage());
	}
	// Debug print
	dataSource.printDB();

	// Closing Datasource
	dataSource.close();
}
```

Per specificare lo schema che il nostro database utilizza è stata definita una contract-class (StreetDBHelper) che specifica esplicitamente il layout del database. A quest'ultima abbiamo agganciato una classe per eseguire le operazioni più comuni in modo da sfruttare al meglio le API forniteci da Android (classe StreetDataSource): i metodi di questa classe prevedono l'inserimento di una strada, la sua rimozione o il recupero di tutte le strade dal database. Ad esempio per recuperare le informazioni relative ad una strada a partire dal suo nome invocheremo il metodo

```
// Return a street with a specific name
public Street getStreet(String name)
{
	database = dbHelper.getReadableDatabase();
	Cursor cursor = database.query(StreetDBHelper.TABLE_NAME,
			allColumns, StreetDBHelper.COLUMN_NAME_STREET_NAME + "= \""
			+ name +"\"",
			null,null,null,null);
	if(!cursor.moveToFirst())
		return null;
	Street street = cursorToStreet(cursor);
	cursor.close();

	return street;
}
```

Il risultato di una query eseguita su un database viene restituito sotto forma di Cursor che permette tramite alcuni metodi della classe di navigare tra i vari risultati ottenuti.

Il click sul bottone di start nella MainActivity provoca l'invocazione del metodo

```
	public void startEvaluation(View view)
	{
		Intent intent = new Intent(this, LocationActivity.class);
		startActivity(intent);
	}
```

che crea un Intent esplicito per avviare una LocationActivity che come vedremo corrisponde a quello che era stato indicato come "Modulo di Localizzazione".

Schermata di localizzazione

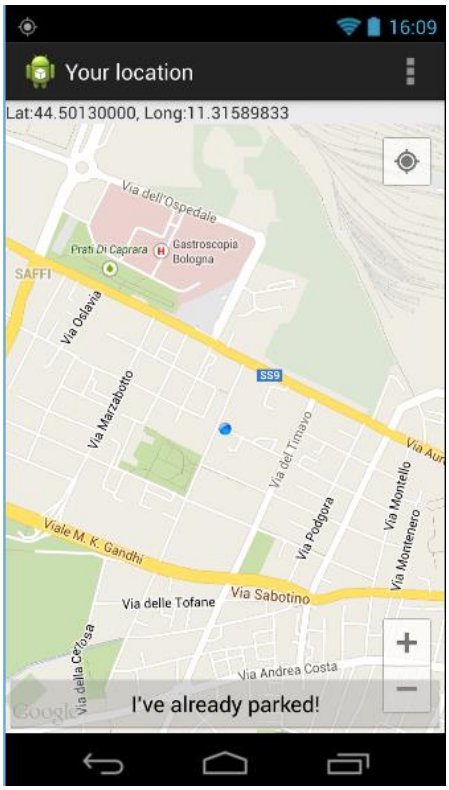
La schermata di localizzazione è costituita da una mappa che evidenzia gli spostamenti dell'utente in tempo reale. Per fare ciò vengono sfruttate le API di Google Play Services ed è quindi obbligatorio controllare che esse siano installate sul dispositivo.

```
SupportMapFragment mapFragment = (SupportMapFragment)
getSupportFragmentManager().findFragmentById(R.id.map);

// Getting GoogleMap object from the fragment
googleMap = mapFragment.getMap();

// Enabling MyLocation Layer of Google Map
googleMap.setMyLocationEnabled(true);

// Getting LocationManager object from System Service LOCATION_SERVICE
locationManager = (LocationManager) getSystemService(LOCATION_SERVICE);
```

L'oggetto googleMap è un oggetto di tipo

```
private GoogleMap googleMap;
```

che ovviamente permette di visualizzare la posizione corrente su una mappa direttamente presa dai servizi di Google. La mappa viene visualizzata in un Fragment un'entità che in genere è utilizzata per decomporre le funzionalità di un'applicazione o dell'UI (come in questo caso) in moduli riutilizzabili. In questa maniera si possono inserire diversi Fragments in una schermata evitando di dover cambiare Activity.



Per gestire gli aggiornamenti richiediamo al sistema il location service: dobbiamo quindi ricordarci di disconnetterci da tale servizio in caso di Activity non visibile o in pausa.

```
@Override
protected void onStop()
{
```

```
        super.onStop();

        //Remove the lister previously added
        locationManager.removeUpdates(this);
    }

    @Override
    protected void onPause()
    {
        super.onPause();

        //Remove the lister previously added
        locationManager.removeUpdates(this);
    }
```

Per selezionare il location provider utilizziamo una classe chiamata Criteria che restituisce un provider in base a criteri settabili (a scelta tra accuratezza, utilizzo della batteria, capacità di riferire altitudine o velocità ecc.).

```
    // Creating a criteria object to retrieve provider
    Criteria criteria = new Criteria();

    // Getting the name of the best provider
    String provider = locationManager.getBestProvider(criteria, true);
```

Una volta ottenuta la nuova posizione la mostriamo sulla mappa:

```
    // Showing the current location in Google Map

    googleMap.moveCamera(CameraUpdateFactory.newLatLng(LocationUtils.getLatLng(
    lastValidLocation)));

    // Zoom in the Google Map
    googleMap.animateCamera(CameraUpdateFactory.zoomTo(15));

    onLocationChanged(lastValidLocation);
```

e richiediamo nuovi aggiornamenti della stessa

```
    locationManager.requestLocationUpdates(provider,
    LocationUtils.UPDATE_INTERVAL_IN_MILLISECONDS, 0, this);
```

Ottenuta l'ultima locazione valida, invochiamo il metodo onLocationChanged per effettuare delle valutazioni sulla nostra attuale posizione: l'applicazione deve essere infatti in grado di capire automaticamente che l'utente ha parcheggiato. Per fare questo la nostra applicazione controlla prima che la posizione ricevuta sia migliore dell'ultima:

```
private boolean isBetterLocation(Location location, Location currentBestLocation)
{
    if (currentBestLocation == null)
    {
        // A new location is always better than no location
        return true;
    }

    if(location.getLatitude() == currentBestLocation.getLatitude() &&
       location.getLongitude() == currentBestLocation.getLongitude())
        return false;
```

```
        // Check whether the new location fix is more or less accurate
        int accuracyDelta = (int) (location.getAccuracy() -
        currentBestLocation.getAccuracy());
        boolean isLessAccurate = accuracyDelta > 0;
        boolean isMoreAccurate = accuracyDelta < 0;
        boolean isSignificantlyLessAccurate = accuracyDelta > 200;

        // Check if the old and new location are from the same provider
        boolean isFromSameProvider = isSameProvider(location.getProvider(),
                            currentBestLocation.getProvider());

        // Determine location quality using a combination of timeliness and
        // accuracy
        if (isMoreAccurate)
        {
              return true;
        }
        else if (!isLessAccurate)
        {
              return true;
        }
        else if (!isSignificantlyLessAccurate && isFromSameProvider)
        {
              return true;
        }
        return false;
}
```

Il metodo isBetterLocation si occupa di questa operazione restituendo true in caso di posizione "migliore" della precedente o false in caso contrario.

Se la locazione ottenuta è migliore della precedente si calcola la velocità dell'utente: se quest'ultima è inferiore alla soglia limite l'utente è probabilmente sceso dal veicolo e quindi quest'ultimo è parcheggiato permettendo così di passare alla fase di conversione da posizione a via.

```
        // Calculating the time difference between the last update and the old one
        int deltaT = (int) ((timeOfArrivalUpdate - prevTimeOfArrivalUpdate)/1000);
        Log.d(getString(R.string.debug),"deltaT:"+deltaT);

        if ( deltaT > 0)
        {
              // Getting distance
              float distance = location.distanceTo(lastValidLocation);
              Log.d(getString(R.string.debug),"distance:"+distance);

              // Calculating speed from distance and time
              double speed = distance / deltaT;
              Log.d(getString(R.string.debug),"speed:"+speed);

              if ( speed < LocationUtils.MIN_SPEED)
              {
                    // The user is moving on his feet
                    finished = true;
              }
        }
```

In alternativa se l'utente è fermo per diversi secondi nella stessa posizione lo riteniamo stazionario e procediamo ugualmente alla fase di valutazione della posizione. Per far ciò ci avvaliamo di una

variabile interna count che viene incrementata se non viene rilevata una nuova posizione migliore della precedente (metodo isBetterLocation ritorna false). Anche in questo caso procediamo a settare il flag finished a true in modo da procedere alla nuova fase.

```
cont++;
/*
 *  @param cont
 *     the number of interval of update already passed
 *     update interval * cont == time passed since last update with a
       different location
 *     if it's major than MAX_WAIT_TIME we consider the user stationary
 */
 if ( LocationUtils.UPDATE_INTERVAL_IN_MILLISECONDS * cont >=
      LocationUtils.MAX_WAIT_TIME_IN_MILLISECOND)
       finished = true;
```

Una volta che il flag finished viene settato a true si costruisce un Intent per lanciare la prossima activity

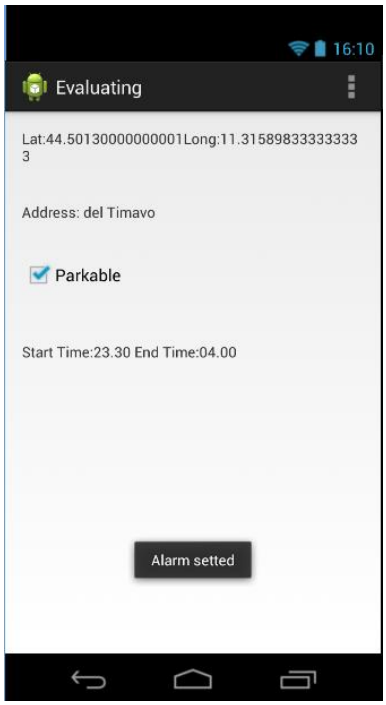
```
// Build an intent to start next activity
double[] latLngValues = new double[2];
latLngValues[0] = lastValidLocation.getLatitude();
latLngValues[1] = lastValidLocation.getLongitude();
cont = 0;

Intent intent = new Intent(this, EvaluationActivity.class);
intent.putExtra("latLng", latLngValues);
startActivity(intent);
```

Nel creare questo Intent esplicito inseriamo l'informazione relativa alla latitudine e alla longitudine che ci serviranno in seguito per ricavare il nome della via nella quale ci troviamo.

Nella schermata è inoltre presente un bottone che serve per gli utenti più frettolosi che forza il passaggio alla schermata di valutazione della posizione con l'attuale posizione.

Schermata di valutazione della posizione

La schermata è composta da varie form per la visualizzazione dei dati in maniera testuale come si può notare in figura:



Il metodo principale associato a questa schermata è getAddress() che al suo interno genera un thread per la valutazione della posizione, passando i dati recuperati dall'Intent su latitudine e longitudine

```
(new
EvaluationActivity.GetAddressTask(this)).execute(latLngValues[0],latLngValues[1])
```

Come prima il task è una classe innestata che questa volta avrà un valore di ritorno: una stringa con il nome della via corrispondente alle coordinate di latitudine e longitudine ricevute in ingresso

```
protected class GetAddressTask extends AsyncTask<Double, Void, String>
```

Per trasformare l'informazione riguardante la posizione da coordinate di latitudine e longitudine ad un nome leggibile di una via utilizzeremo un Geocoder, una classe che permette di gestire la trasformazione di un indirizzo stradale in coordinate di latitudine e longitudine e viceversa. L'utilizzo di questa classe tuttavia ha come requisito un servizio di backend che non è incluso nel core framework di Android. Bisogna quindi accertarsi che esista un'implementazione del Geocoder sul dispositivo attraverso il metodo isPresent().

Nel metodo doInBackground avremo quanto segue:

```
//Get a geocode set for localized addresses
Geocoder geocoder = new Geocoder(localContext, Locale.getDefault());

//Create a list to contain the result address
List<Address> addresses = null;

//Try to get an address for the current location
try
{
      //Get at most one result
      Log.d(getString(R.string.debug),"Lat - "+params[0]);
```

```
        Log.d(getString(R.string.debug),"Lat - "+params[1]);
        addresses = geocoder.getFromLocation(params[0], params[1],1);
        if (addresses.size()>0)
              Log.d(getString(R.string.debug),"Step 2 reached, we've a list
              of addresses");
}
```

Ottenendo al massimo un risultato nella lista di indirizzi addresses che manipoleremo in questa maniera

```
Address address = addresses.get(0);
String addressText = address.getMaxAddressLineIndex()>0 ?
                     address.getAddressLine(0) : "";
return addressText;
```

per ottenere semplicemente il nome della via con la chiamata getAddressLine(0). Abbiamo così il solo nome della via in maniera coerente con quanto salvato nel nostro DB. Questo dato sarà poi usato dal metodo onPostExecute() che controlla se tale via è presente nel DB

```
dataSource = new
         StreetsDataSource(EvaluationActivity.this.getApplicationContext());
dataSource.open();
address:"+finalAddress);
Street street = dataSource.getStreet(finalAddress);
```

In caso street sia nullo vuol dire che non abbiamo nessun riscontro nel nostro database ed è quindi possibile parcheggiare in questa via. Settiamo quindi con una spunta la casella relativa alla possibilità di parcheggiare in questa via. Se d'altro canto la via è presente nel database bisogna controllare se la pulizia sta per iniziare:

```
long actualTime = GregorianCalendar.getInstance().getTimeInMillis();
try
{
        c = getDate(street.getSTime());
        start = c.getTimeInMillis();
        c = getDate(street.getETime());
        end = c.getTimeInMillis();
        if(end <= start)
        {
              c.add(Calendar.DATE, 1);
              end = c.getTimeInMillis();
        }
}
catch(ParseException e)
{
        Log.e(getString(R.string.debug),e.getMessage());
        e.printStackTrace();
}
```

Recuperate le informazioni riguardo l'inizio della pulizia e la sua fine, controlliamo che la fine non anteceda l'inizio: se questo avviene ci troviamo nel caso in cui la pulizia inizia in un giorno e finisce il giorno dopo e quindi aggiorniamo di conseguenza il campo date incrementandolo di uno per ottenere il giusto valore in millisecondi.

Andiamo adesso a confrontare i dati ottenuti con l'ora attuale e distinguiamo i vari casi.

```
if(actualTime >= start && actualTime <= end)
{
	// The cleaning is being performed right now
	Log.d(getString(R.string.debug),"returning false");
	return false;
}
```

In questo caso la pulizia viene eseguita in questo momento e non è quindi possibile parcheggiare.

```
if(actualTime + LocationUtils.TWO_HOURS_IN_MILLISECONDS >= start &&
   actualTime + LocationUtils.TWO_HOURS_IN_MILLISECONDS <= end )
{
	// The cleaning will be performed in 2 hours
	Log.d(getString(R.string.debug),"returning false");
	return false;
}
```

Altrimenti dobbiamo controllare che la pulizia non inizi a due ore a partire da adesso.

In entrambi i casi ritorniamo il valore false, altrimenti possiamo parcheggiare in questa via al momento e ritorniamo il valore true. In quest'ultimo caso dobbiamo ricordarci di settare un allarme che ci avverta a due ore dall'inizio della pulizia per avere il tempo di andare a spostare il veicolo.

```
Calendar cleanStartTime;
// Get a Calendar initialized at the starting time of the cleaning
cleanStartTime = getDate(street.getSTime());
// Subtract two hours to get a long representing the time at which the
alarm should be sent out
cleanStartTime.add(Calendar.HOUR_OF_DAY, -2);
long alarm = cleanStartTime.getTimeInMillis();
```

Per settare l'allarme ci avvaliamo di una classe chiamata AlarmManager: tramite questa classe possiamo registrare un Intent per essere spedito ad un dato momento ed avviare così una specifica applicazione anche se non attualmente in esecuzione.

```
Intent intent = new Intent(this.localContext, AlarmReceiver.class);
intent.putExtra("alarm_message", "Cleaning of the street
                "+street.getName()+" is about to begin in 2 hours.");
PendingIntent sender = PendingIntent.getBroadcast(this.localContext,
                       ALARM_REQUEST_CODE, intent,
                       PendingIntent.FLAG_UPDATE_CURRENT);
// Get the AlarmManager service
AlarmManager am = (AlarmManager) getSystemService(ALARM_SERVICE);
am.set(AlarmManager.RTC_WAKEUP, alarm, sender);
Log.d(getString(R.string.debug),"Alarm setted");
```

Per gestire l'allarme così schedulato ci avvaliamo di un BroadcastReceiver che chiamiamo Alarm receiver:

```
public class AlarmReceiver extends BroadcastReceiver
```

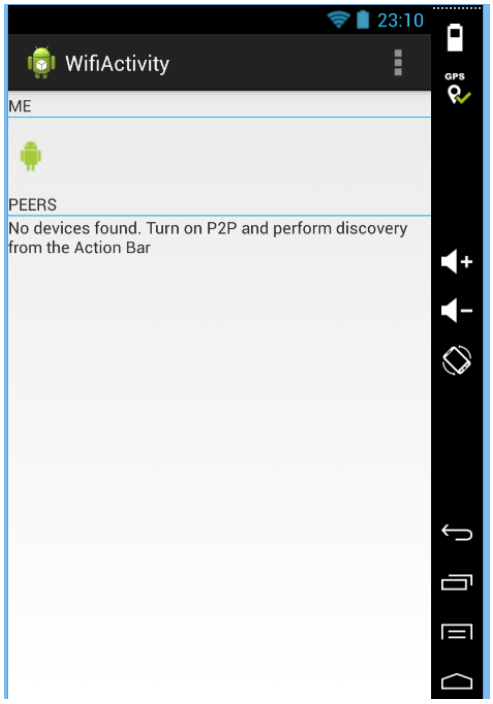
Al momento della ricezione dell'Intent invochermo il metodo onReceive che setterà una notifica via notificationBuilder, la classe responsabile in Android della costruzione delle notifiche a partire dalle API di livello 11, che permette tra le altre cose di settare varie campi della notifica.

```
public void onReceive(Context context, Intent intent)
{
	Log.d(debugTag,"Received an alarm");
```

```
            // Get the message inside the intent tagged as alarm_message
            Bundle bundle = intent.getExtras();
            String message = bundle.getString("alarm_message");

            // Getting a notification manager
            NotificationManager notificationManager = (NotificationManager)
            context.getSystemService(Context.NOTIFICATION_SERVICE);

            // Build notification
            NotificationCompat.Builder notificationBuilder = new
            NotificationCompat.Builder(context);

            // Title and text of the notification
            notificationBuilder.setContentTitle("Cleaning of the street in 2
            hours");
            notificationBuilder.setContentText(message);

            // Text that appears in the status bar along with the notification
            notificationBuilder.setTicker("Move your car!");

            // Notification time
            notificationBuilder.setWhen(System.currentTimeMillis());

            // Notification icon
            notificationBuilder.setSmallIcon(R.drawable.spazzatrice);

            // NOtification Auto-cancel after click
            notificationBuilder.setAutoCancel(true);

            // Setting sound, lights and vibrations as defaults
            notificationBuilder.setDefaults(Notification.DEFAULT_SOUND
                        | Notification.DEFAULT_LIGHTS |
                        Notification.DEFAULT_VIBRATE);

            notificationManager.notify(SIMPLE_NOTIFICATION_ID,
            notificationBuilder.build());

            // Creating the PendingIntent (launched when the notification is
            clicked)
            Intent notificationIntent = new Intent(context, WifiActivity.class);
            notificationIntent.putExtra("MessageToVisualize", message);
            PendingIntent contentIntent = PendingIntent.getActivity(context, 0,
                           notificationIntent, 0);
            notificationBuilder.setContentIntent(contentIntent);

    }
```

Possiamo notare come le ultime istruzioni siano dedicate a settare un Intent per lanciare l'Activity atta a rispondere a tale sollecitazione. Nel nostro caso vogliamo verificare se in raggio di copertura abbiamo degli amici da poter contattare tramite Wi-Fi Direct e inviare loro un messaggio. Anche in questo caso come avvenuto in precedenza inseriamo nel campo Extra dell'Intent il messaggio da passare a tale Activity associato ad un tag per recuperarlo (in questo caso il tag è "MessageToVisualize").

Schermata di comunicazione tramite Wi-Fi Direct



Nella schermata dedita alla comunicazione con il Wi-Fi Direct abbiamo una lista di utenti rilevati in raggio di copertura dal dispositivo. Anche in questo caso utilizzeremo dei Fragment in quanto la quantità di utenti rilevati è variabile e ciò ci permette di avere una visualizzazione adatta alle dimensioni del dispositivo.

Per registrarci alle modifiche relative al Wi-Fi Direct dobbiamo dichiarare un particolare Intent Filter:

```
intentFilter = new IntentFilter();
intentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION);
intentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION);
intentFilter.addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION
intentFilter.addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION);
```

Possiamo ottenere direttamente dal sistema il service dedito alle operazioni di Wi-Fi Direct, settando anche il canale ove avverrà la comunicazione. Il tutto avviene tramite la chiamata:

```
private WifiP2pManager manager;
private Channel channel;
manager = (WifiP2pManager) getSystemService(Context.WIFI_P2P_SERVICE);

channel = manager.initialize(this, getMainLooper(), null);
```

Per ricevere e gestire le notifiche che arrivano istanziamo e utilizziamo un BroadcastReceiver che gestisca solo questo tipo di notifiche:

```
receiver = new WiFiDirectBroadcastReceiver(manager, channel, this);
```

Chiaramente se l'activity finisce in pausa o viene distrutta ci dobbiamo ricordare di de-registrare tale receiver

```
unregisterReceiver(receiver);
```

L'elenco di dispositivi raggiungibili tramite Wi-Fi Direct è modellato tramite una `List<WifiP2pDevice>`: la classe che rappresenta la lista dei device trovati durante la fase di discovery estende quindi una ListFragment (lista di fragment, dato che ogni peer è rappresentato come tale) .

Il fragment che gestisce un device nello specifico fa parte della classe DeviceDetailFragment che implementa l'interfaccia `ConnectionInfoListener` responsabile dell'invocazione delle callback quando sono disponibili informazioni sulla connessione avvenuta tra due device.

Al suo interno infatti troviamo la logica relativa allo scambio di messaggi. Nello specifico viene avviato un service che si occuperà di trasferire il messaggio da un dispositivo all'altro:

```
public void onClick(View v)
{
	// Set up the message and start the service
	Intent serviceIntent = new Intent(getActivity(),
					MsgTransferService.class);

	serviceIntent.setAction(MsgTransferService.ACTION_SEND_MESSAGE);

	serviceIntent.putExtra("MessageToVisualize", message);

	serviceIntent.putExtra(MsgTransferService.EXTRAS_GROUP_OWNER_ADDRESS,

	info.groupOwnerAddress.getHostAddress());
	serviceIntent.putExtra(MsgTransferService.EXTRAS_GROUP_OWNER_PORT,
				8988);
	getActivity().startService(serviceIntent);
}
```

Nel service che avviamo, recuperiamo le informazioni relative al group owner della connessione e la porta che abbiamo inviato tramite intent e utilizziamo una socket per l'invio del messaggio

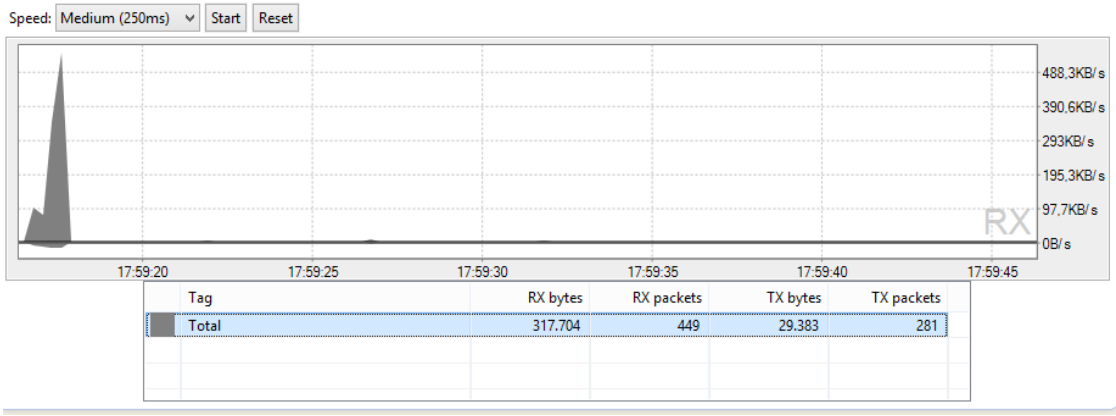
```
String messageToVisualize =  intent.getExtras().getString("MessageToVisualize");
String host = intent.getExtras().getString(EXTRAS_GROUP_OWNER_ADDRESS);
Socket socket = new Socket();
int port = intent.getExtras().getInt(EXTRAS_GROUP_OWNER_PORT);

try
{
	Log.d(getString(R.string.debug), "Opening client socket - ");
	socket.bind(null);
	socket.connect((new InetSocketAddress(host, port)), SOCKET_TIMEOUT);

	Log.d(getString(R.string.debug), "Client socket - "+socket.isConnected());
	OutputStream stream = socket.getOutputStream();
	try
	{
		BufferedWriter stdout = new BufferedWriter(new
							OutputStreamWriter(stream));
		stdout.write(messageToVisualize);
	} catch (FileNotFoundException e) {
		Log.d(getString(R.string.debug), e.toString());
	}
	Log.d(getString(R.string.debug), "Client: Data written");
	} catch (IOException e) {
		Log.e(getString(R.string.debug), e.getMessage());
	} finally {
```

```
            if (socket != null) {
                  if (socket.isConnected()) {
                        try {
                              socket.close();
                            } catch (IOException e) {
                              // Give up
                              e.printStackTrace();
                            }
                  }
            }
      }
```

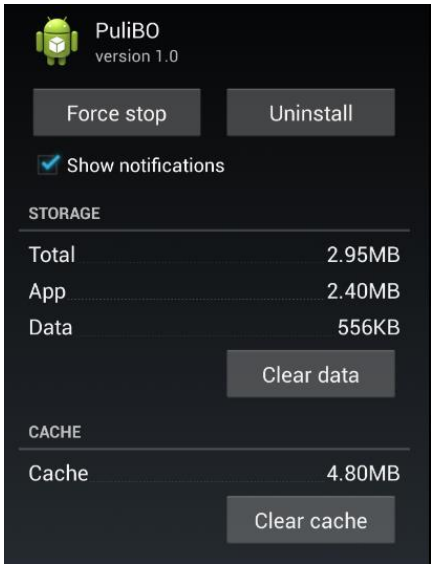
Benchmark

Alcuni benchmark ottenuti tramite DDMS durante un esecuzione dell'applicazione per 3 minuti:

Network usage (ricezione file e update locazione):



Data and cache size:



Heap usage:

Heap updates will happen after every GC for this client

ID	Heap Size	Allocated	Free	% Used	# Objects
1	13,078 MB	11,297 MB	1,782 MB	86,38%	81.096