

*Questa è la DEDICA:  
ognuno può scrivere quello che vuole,  
anche nulla ...*



# Introduzione

Questa è l'introduzione.



# Indice

|   |           |
|---|-----------|
| <b>Introduzione</b>   | <b>i</b>  |
| <b>1 Stato dell'arte</b>                                    | <b>1</b>  |
| <b>2 Progettazione</b>                                      | <b>3</b>  |
| 2.1 Scopo e problematiche . . . . .                         | 3         |
| 2.2 Scenario generale . . . . .                             | 4         |
| 2.3 Architettura generale . . . . .                         | 5         |
| 2.4 Architettura software . . . . .                         | 7         |
| 2.5 Organizzazione dell'informazione . . . . .              | 9         |
| 2.6 Network State Machine . . . . .                         | 11        |
| 2.7 Statistica . . . . .                                    | 13        |
| <b>3 Implementazione</b>                                    | <b>15</b> |
| 3.1 Componente di disseminazione . . . . .                  | 15        |
| 3.1.1 Strategia di connessione tramite WiFiDirect . . . . . | 16        |
| 3.1.2 Il NetworkController . . . . .                        | 17        |
| 3.1.3 NetworkConnection . . . . .                           | 18        |
| 3.1.4 Esplorazione dell'ambiente circostante . . . . .      | 19        |
| 3.1.5 Funzionalità Access Point-like . . . . .              | 23        |
| 3.1.6 Canali di comunicazione . . . . .                     | 26        |
| 3.1.7 L'ascoltatore BroadcastReceiverManagement . . . . .   | 29        |
| 3.1.8 Il servizio background . . . . .                      | 32        |
| 3.2 Screenshot . . . . .                                    | 33        |

|          |                                 |           |
|----------|---------------------------------|-----------|
| <b>4</b> | <b>ALTRA SEZ</b>                | <b>39</b> |
| 4.0.1    | Prima SottoSezione . . . . .    | 39        |
| 4.1      | Terza Sezione . . . . .         | 40        |
| 4.2      | Altra Sezione . . . . .         | 41        |
| 4.2.1    | Altra SottoSezione . . . . .    | 41        |
| 4.3      | Altra Sezione . . . . .         | 41        |
| 4.4      | Altra Sezione . . . . .         | 42        |
| 4.4.1    | Listati dei programmi . . . . . | 42        |
|          | <b>Conclusioni</b>              | <b>43</b> |
|          | <b>A Prima Appendice</b>        | <b>45</b> |
|          | <b>B Seconda Appendice</b>      | <b>47</b> |
|          | <b>Bibliografia</b>             | <b>49</b> |

# Elenco delle figure

|     |   |    |
|-----|---|----|
| 2.1 | Rappresentazione dell'Architettura generale e della divisione<br>in celle di una porzione di Bologna . . . . .              | 6  |
| 2.2 | Rappresentazione dell'Architettura software . . . . .   | 7  |
| 2.3 | Rappresentazione dell'Architettura software . . . . .   | 12 |
| 3.1 | Interazione utente per il consenso alla connessione . . . . .   | 16 |
| 3.2 | Mappa del centro di Bologna divisa in celle, ognuna colorata<br>con la relativa probabilità di trovare parcheggio . . . . . | 34 |
| 3.3 | Servizio di disseminazione attivo negli stati STATE_ACCESS_POINT_NO_PEERS<br>e STATE_ACCESS_POINT_PEERS . . . . .           | 35 |
| 3.4 | Didascalia comune alle due figure . . . . .   | 36 |
| 3.5 | Didascalia comune alle due figure . . . . .   | 37 |
| 3.6 | Didascalia comune alle due figure . . . . .   | 38 |
| 4.1 | legenda elenco figure . . . . .   | 40 |





# Elenco delle tabelle

|     |                                  |    |
|-----|----------------------------------|----|
| 4.1 | legenda elenco tabelle . . . . . | 42 |
|-----|----------------------------------|----|



# Capitolo 1

## Stato dell'arte

trtrtrtrr



# Capitolo 2

## Progettazione

Il seguente capitolo ha lo scopo di illustrare in dettaglio il goal del progetto, le problematiche che vi stanno dietro ed il processo di progettazione su cui è stato necessario soffermarsi, per la successiva realizzazione ed implementazione. La mission, descrivendola in pochi concetti, è sostanzialmente quella di disseminare le informazioni, relative al concetto di Smart Parking, in maniera totalmente distribuita e senza alcun appoggio di una struttura centralizzata. Questo concetto diventa innovativo dal momento in cui non si passa dalla rete cellulare o da qualche gateway verso Internet per la comunicazione tra dispositivi, ma verranno utilizzate soltanto le tecnologie e apparecchiature presenti nei più comuni smartphone in commercio. Dettagli tecnologici ed implementativi saranno comunque discussi nel prossimo capitolo. Di seguito verranno illustrati soprattutto le principali componenti che compongono il sistema, descrivendone funzionalità, utilità ed integrazione con le restanti componenti.

### 2.1 Scopo e problematiche

Come già accennato e discusso in precedenza lo scopo principale è quello di propagare informazioni, tra dispositivi mobili, in maniera completamente distribuita, senza l'appoggio di agenti centralizzati. Il device utente, tramite

la componente di Activity Recognition presente nel software, compie un evento di parcheggio o rilascio, dopo di chè tiene traccia dell'evento salvandolo in locale nella memoria. Le informazioni campionate sopra descritte, saranno l'oggetto della comunicazione tra i device che potranno fare assunzioni su eventuali zone dense di parcheggi liberi. Più gli utenti restano allineati e sincronizzati su questi dati, più restano aggiornati e consapevoli della situazione parcheggi nei dintorni. Per semplicità concettuale la città viene divisa in celle(identificate con id univoci), permettendo così alla logica di disseminare informazioni relative alla cella in questione e a quelle adiacenti. Un altro aspetto non meno importante di quelli già illustrati, è il fatto che il servizio deve funzionare in modo continuativo nel tempo e soprattutto in modo totalmente autonomo. Questo significa che l'utente, senza nessuna interazione con il device(mantenendolo per esempio in tasca), sincronizza comunque i propri dati con quelli di utenti circostanti, nei paraggi. La funzionalità appena descritta, per renderla robusta, ha richiesto una fase implementativa consistente, la quale verrà illustrata tecnicamente in seguito.

## 2.2 Scenario generale

Lo scenario che si va a considerare per l'utilizzo di un'applicazione di questo genere è una città metropolitana. Questa può essere diversamente interpretata e suddivisa logicamente in una griglia composta da celle(quadranti), dove ognuna di queste viene identificata univocamente da una coppia di coordinate X e Y. Si presuppone che il numero di parcheggi disponibili in una data cella sia un dato noto a priori, il quale comprende tutti gli slot disponibili nelle strade che sono logicamente raggruppate, all'interno del quadrante considerato. Il device dell'utente, tramite l'implementazione presente nell'app, performando un evento di parcheggio o rilascio, riesce a recuperare la cella in cui risiede logicamente e registrerà nel proprio database locale che l'evento si è verificato in quello specifico quadrante. Le assunzioni sulla probabile quantità, in percentuale, di parcheggi liberi vengono calcolate sulla

granularità della cella. Quando due peer cercheranno di sincronizzarsi verranno scambiate le informazioni relative alla cella corrente e quelle adiacenti. In questo modo, si cerca di fornire all'utente, previsioni più precise sulle celle che sono nei pressi della posizione corrente del device. Questo evita overhead nella comunicazione e sincronizzazione di entries che non sono utili all'utente. Rimanendo nel discorso di overhead, è normale pensare che questo modello possa “esplodere“, in quanto la mole di dati potrebbe diventare enorme con il passare del tempo, così come le opportunità di sincronizzazione. Per evitare che questo succeda, viene controllata la dimensione del db locale, ed eventualmente eliminate le entries meno aggiornate. Inoltre i peers possono effettuare sincronizzazioni soltanto dopo una certa threshold, espressa in secondi. Questi concetti sono stati parametrizzati in fase di implementazione, quindi possono essere giustamente calibrati in base all'ambiente in cui si intende utilizzare l'app.

## 2.3 Architettura generale

L'architettura generale, analizzandola da un punto di vista esterno, prevede come unici attori i device utenti. Questi muovendosi all'interno dell'ambiente città restano in attesa di possibili sincronizzazioni. Ogni dispositivo, che presenta il servizio di comunicazione e sincronizzazione attivo, può interpretare fondamentalmente due ruoli concettuali:

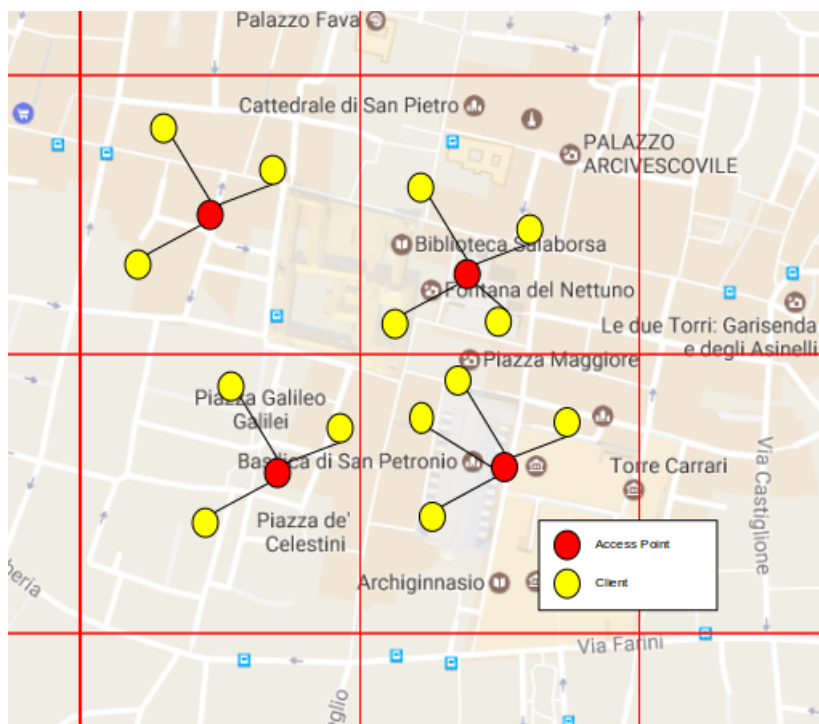
- **ROLE ACCESS POINT:**

Il device funge da access point, offrendo la possibilità a device presenti in prossimità di agganciarsi all'HotSpot esposto. Una volta che un Client si connette stabilmente ad esso avrà l'opportunità di aprire un canale di comunicazione e sincronizzare i propri dati con quelli del Access Point, denominato anche come Server. Inoltre l'Access Point notifica la propria presenza e informazioni utili attraverso la propagazione di un beacon di servizio.

- **ROLE CLIENT:**

Se un device si accorge della presenza di un HotSpot nelle vicinanze, grazie al beacon sopra citato, può tentare di connettersi come Client all'Access Point. Qualora riesca ad agganciarsi in modo stabile all'HotSpot verranno avviati i canali di comunicazione, su cui effettuare una sincronizzazione tra i device Client e Server.

A questo punto la topologia della rete generale può essere vista come hub di hub, ovvero tante piccole star, dove grazie al movimento inerziale dei device si riuscirà ad ottenere una propagazione dell'informazione, interconnettendo appunto le informazioni di diversi hub tra di loro, ad istanti di tempo differenti.



**Figura 2.1:** Rappresentazione dell'Architettura generale e della divisione in celle di una porzione di Bologna

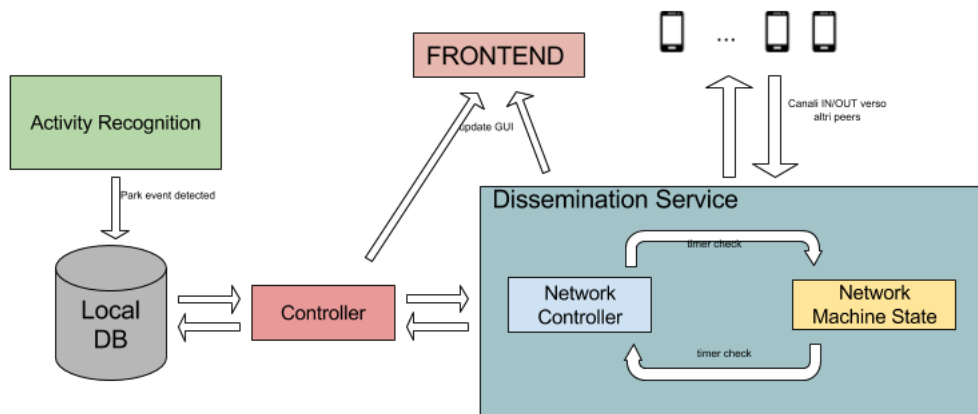
Come si può comprendere dalla figura i device muovendosi continuamente



all'interno dello scenario (identificato in questo caso con la griglia città), formeranno continuamente nuovi Hub, con attori sempre diversi. Questo fattore è determinante per il processo di propagazione delle informazioni, in quanto dati presenti in un Hub, verranno sincronizzati, in istanti successivi, con Hub diversi. L'analisi dello spreading delle informazioni, la sua efficacia ed accuratezza sono illustrate nel capitolo di valutazione, nel quale si cerca di capire se questo meccanismo è attuabile nella realtà.

## 2.4 Architettura software

Prima della fase di implementazione, è stata necessaria una fase di progettazione dell'architettura software. L'applicazione, per essere considerata utilizzabile dagli utenti ed efficace nel meccanismo di spreading dell'informazione, necessita di numerosi componenti e soprattutto dell'interazione tra essi. Nella figura sono stati messi i componenti principali che concettualmente compongono l'architettura e come questi collaborano.



**Figura 2.2:** Rappresentazione dell'Architettura software

- **Componente di Activity Recognition:**

Il componente di activity recognition si preoccupa di rilevare gli eventi di parcheggio e rilascio, che un determinato utente compie, col trascorrere del tempo. Questo è possibile tramite l'utilizzo dei sensori, i

quali campionano periodicamente lo stato di accelerometro e giroscopio, mappandoli tramite un algoritmo decisionale in un'attività ben definita. Questo processo, in realtà, si compone di due fasi come training e recognition. La prima è necessaria alla seconda e riguarda il campionamento dei dati sensore in una attività definita dall'utente di CAR o NO\_CAR. La seconda, invece, grazie ai campioni rilevati nella prima riuscirà a fare assunzione sul tipo di attività che l'utente sta svolgendo. Quando il componente capisce che si è passato da uno stato CAR ad uno stato NO\_CAR, significa che l'utente ha parcheggiato l'autovettura, mentre, dualmente, se si passa da uno stato NO\_CAR a CAR, significa che l'utente ha rilasciato uno slot. Quando una di queste due situazioni accade, si registra l'evento nel local DB.

- **Componente local DataBase:**

Il Database locale si preoccupa di persistere le informazioni sul device utente, le quali poi verranno processate per calcolare la probabilità di parcheggio in una determinata zona. Le informazioni che vengono memorizzate sono illustrate più precisamente nel paragrafo XYZ.

- **Componente Controller:**

Il Controller si preoccupa di scrivere, aggiornare e dare accesso alle informazioni presenti nel database locale. Inoltre, è stato pensato come semplice meccanismo di Object-Relational Mapping(ORM), per integrare al meglio le logiche ad oggetti con i dati presenti nel DBMS. Sostanzialmente, può essere interpretato come un oggetto Controller, del pattern MVC(Model-View-Controller), gestendo ed implementando le classi del Modello.

- **Componente Frontend:**

Il frontend sono sostanzialmente le views che verranno mostrate all'utente, sul proprio device Android. Si è optato per un frontend semplice, per rendere utilizzabile l'app a qualsiasi tipo di utente, pur fornendo ad esso tutte le informazioni necessarie sullo stato dei parcheggi nel-

le vicinanze. Nella sezione Screenshot è possibile prendere visione del layout grafico pensato per l'applicazione.

- **Componente Dissemination Service:**

Il componente di disseminazione delle informazioni è il cuore del progetto e l'elemento più articolato all'interno dell'architettura. Questo è composto da diversi sottocomponenti, la maggior parte dei quali verranno illustrati con precisione nel capitolo di implementazione. Il disseminatore è un servizio che gira in background e la logica è scandita da una FSM(Finite State Machine), dove in base allo stato in cui ci si trova verranno compiute azioni, piuttosto che altre. Quando viene rilevata una nuova connessione ad un peer, essendo questo nel ruolo di client o server, il componente si preoccupa anche di gestire il corretto funzionamento della comunicazione, gestendo i canali di INPUT e OUTPUT verso gli altri device, nel range di comunicazione.

## 2.5 Organizzazione dell'informazione

L'organizzazione e persistenza dei city data è ovviamente una fase determinante del progetto. In fase di progettazione, si è giunti alla conclusione che non è sufficiente scambiarsi il numero di parcheggi liberi/occupati/unknown derivanti da dati statistici, in quanto ogni device/utente potrebbe avere una parziale, ma diversa, visione dello scenario. Da ciò deriva la necessità di sincronizzarsi su tutti gli eventi che sono successi recentemente in una determinata zona(cella) e quelle adiacenti. Si è optato per la realizzazione di un database di tipo relazionale composto principalmente da due tabelle, indipendenti l'una dall'altra. La prima tabella, denominata, `park_events` colleziona tutti gli eventi che gli utenti compiono o ricevono in seguito a sincronizzazioni con altri utenti. La seconda tabella `synchronizations` colleziona invece tutte le sincronizzazioni che hanno coinvolto il device in questione. Come verrà illustrato nel capitolo relativo all'implementazione, gli strumenti utilizzati si baseranno completamente sullo standard SQL, i quali permetteranno una

semplice gestione di query di filtraggio, creazione, modifica, cancellazione. Entrando più in dettaglio nella composizione delle tabelle e degli attributi salvati, si mostra di seguito uno schema riassuntivo di ciò che viene messo a database.

Tabella **park\_events**:

- **cell\_id** : identificativo univoco della cella che la identifica all'interno della grid city. Informazione di tipo geografica.
- **event** : evento che può essere limitatamente PARKED o RELEASED. Indica se l'entry fa riferimento ad una sosta o alla liberazione di un parcheggio.
- **timestamp** : datetime che indica il momento in cui è stato performedo l'evento in questione.
- **mac** : identificativo univoco del device che ha performedo l'evento di parcheggio o rilascio.

La PRIMARY KEY è composta da <timestamp, mac >. L'idea che vi sta dietro è quella che un utente può performare, in un dato istante, al più un unico evento di rilascio o parcheggio.

Tabella **synchronizations**:

- **timestamp** : datetime in cui è avvenuta la sincronizzazione
- **mac** : id univoco dell'altro device con cui è stata possibile la sincronizzazione
- **ap\_role** : booleano che indica se ero Access Point o meno al momento della sincronizzazione

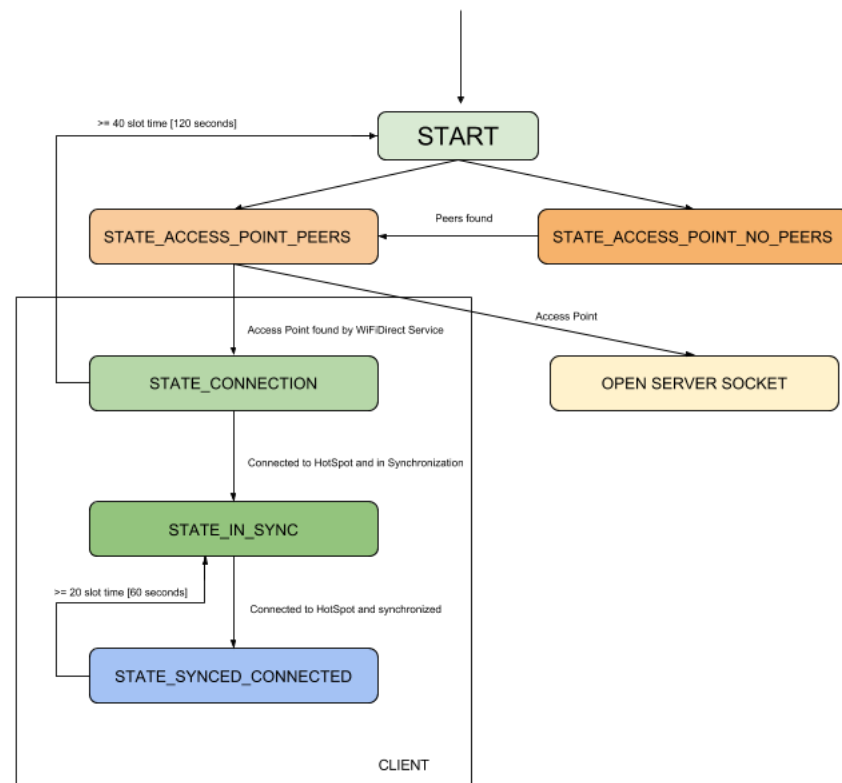
Questa tabella oltre a fini statistici, è importante perché permette di evitare che due device si sincronizzino in continuazione. Grazie all'attributo

timestamp, si riesce a risalire all'ultima sincronizzazione che ha coinvolto il device e di conseguenza, è possibile istanziare una time threshold, sotto la quale non è possibile iniziare una nuova sincronizzazione. Le logiche implementate a backend non permettono lo scambio di dati in istanti troppo vicini tra loro. Questo meccanismo può esser visto come un minimo di salvaguardia da cicli continui o sovrapposizione di sincronizzazioni.

## 2.6 Network State Machine

Come già accennato nell'architettura software, il processo di disseminazione delle informazioni è scandito da una macchina a stati finiti. Si è deciso di approfondire questo componente per far comprendere meglio come vengono gestite le sincronizzazioni tra i device utente coinvolti.

Alla partenza del servizio, il device, non ha conoscenza dell'ambiente circostante, di conseguenza si rende disponibile come ruolo di Access Point ed entra nello stato `STATE_ACCESS_POINT_NO_PEERS`. Quando si riesce a scoprire la presenza di altri peer, nei paraggi, si entra nello stato `STATE_ACCESS_POINT_PEERS`. A questo punto, il device, si mette all'ascolto di eventuali beacon di servizio, i quali potrebbero notificare la presenza di un preesistente AccessPoint. Se questo non accade, il device rimane disponibile come ruolo di Access Point, notificando la propria presenza in beaconing e fornendo opportunità di sincronizzazioni ad eventuali client in arrivo (stato `OPEN_SERVER_SOCKET`). Diversamente, se il device comprende la presenza di un Access Point già consolidato in zona, smette di svolgere il ruolo di HotSpot e comincia a comportarsi come Client, entrando, di conseguenza, nello stato `STATE_CONNECTION`. Ora il device cerca di connettersi all'HotSpot, esposto grazie alla tecnologia WiFi Direct. Se il processo porta ad una connessione stabile e ad un assegnazione di un indirizzo IP, verrà automaticamente aperto un canale di comunicazioni bidirezionali tra il Client e Server, iniziando così la fase di sincronizzazione (stato `IN_SYNC`). Come si può notare dallo schema in figura XY, dallo stato `STATE_CONNECTION`



**Figura 2.3:** Rappresentazione dell'Architettura software

vi è anche la possibilità di ripartire dall'inizio, nel caso in cui, il device non riesca a connettersi all'HotSpot in un tempo ragionevole, impostando una threshold. Questo è stato fatto per evitare situazioni di stallo e cercare di ripartire da un punto più stabile. Supponendo che la sincronizzazione sia andata a buon fine, si passa allo stato `STATE_SYNCED_CONNECTED`, il quale significa che la sincronizzazione è terminata ma si è ancora agganciati all'HotSpot del Server. A questo punto, gli attori coinvolti, potrebbero allontanarsi a tal punto da rompere il legame creato e quindi il servizio ripartirebbe dallo stato iniziale di `START`. Altrimenti, se gli utenti restano nelle vicinanze l'uno dell'altro, in modo tale da mantenere attiva la connes-

sione, si dà l'opportunità al Client(dopo una certa threshold) di richiedere nuovamente una sincronizzazione.

## 2.7 Statistica

La parte di statistica si preoccupa, man mano che le informazioni giungono nel database locale, di calcolare le probabilità di trovare parcheggio in una determinata cella della città, piuttosto che in altre.

Si ricorda che il dato che si fornisce all'utente è una probabilità e non un valore certo, così da dare indicazioni agli utenti, sulle zone, che statisticamente saranno meno congestionate, in termini di occupazione.

Un'altra precisazione riguarda il fatto che più utenti utilizzeranno l'applicazione, come supporto al parcheggio, e più dati verranno disseminati tra i device, aumentando progressivamente l'accuratezza delle probabilità fornite agli utenti.

Supponendo che il numero di slot totali in una generica cella  $i$ , identificato con la sigla  $N_i^t$ , sia noto a priori, lo si può interpretare anche come la somma tra numero di slot liberi,  $N_i^f$ , e numero di slot occupati  $N_i^o$ . Questi ultimi due, possono essere ricavati a partire dagli eventi di parcheggio e rilascio, i quali sono presenti nel database locale citato in precedenza. In particolare, il numero dei parcheggi occupati  $N_i^o$  può essere ricavato dalla differenza tra il numero di eventi parcheggio  $E_i^p$  ed il numero di eventi rilascio  $E_i^r$ .

$$N_i^o = E_i^p - E_i^r$$

A questo punto, conoscendo già il numero totale degli slot, è facile calcolare il tasso di occupazione di una determinata cella:

$$p_i^o = \frac{N_i^o}{N_i^t}$$

Infine calcoliamo la probabilità di trovare parcheggio, servendosi del tasso di

occupazione appena calcolato:

$$p_i^f = 1 - p_i^o$$

Quest'ultima sarà l'informazione a cui l'utente avrà accesso una volta aperta l'applicazione, dove verrà visualizzata la mappa della città suddivisa in griglia ed ogni quadrante sarà colorato in base alla probabilità di trovare parcheggio. Nella sezione screenshot, figura XZ, è possibile vedere quanto descritto.



# Capitolo 3

## Implementazione

Come ribadito nel Cap. 2 l'obiettivo del progetto è quello di disseminare informazioni di smart parking in modalità Device-to-Device, senza l'ausilio di alcuna infrastruttura di rete intermediaria. In base alle informazioni ricevute, si riuscirà dunque a fare assunzioni sulle quantità di parcheggi disponibile in determinate zone della città.

L'implementazione riguarda la realizzazione del processo sopradescritto nel Sistema Operativo Android, con l'ausilio della tecnologia WiFiDirect. Quest'ultima, presente a partire dalle librerie API 14 e device con Android 4.0 o superiori, permette la comunicazione one-to-one tra dispositivi utente.

Il capitolo seguente mostra i passi necessari alla realizzazione, dettagliando i componenti software realizzati partendo dalla base di progettazione, illustrata precedentemente.

### 3.1 Componente di disseminazione

Il paragrafo illustra tutti gli attori principali che compongono il servizio di disseminazione delle informazioni. Si è cercato di spiegare, in ogni sottoparagrafo, le funzionalità del componente e l'interazione con gli altri. Questa è la parte che, in fase di implementazione, ha richiesto maggior sforzo ed

essendo il cuore del progetto, verrà maggiormente descritta, addentrandosi anche nell'analisi del sorgente.

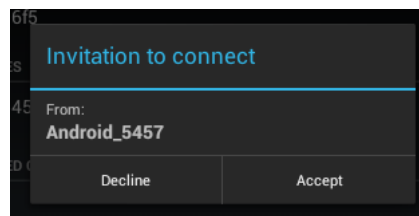
### 3.1.1 Strategia di connessione tramite WiFiDirect

La tecnologia WiFi Direct rende possibile la creazione di cosiddetti P2P Groups, i quali possono essere comparati ad infrastrutture WiFi. Il device che detiene le funzionalità di Access Point viene definito P2P Group Owner, il quale può comunque mantenere attive connessioni cellulari (come ad esempio il 4G). Per realizzare lo scopo del progetto il WiFi Direct non è stato utilizzato in maniera tradizionale, bensì in modalità legacy.

Questo perchè, almeno nel primo incontro tra due device, la nascita della connessione necessita dell'interazione utente (come mostrato in figura XX).

Il nostro processo di comunicazione peer-to-peer non prevede però alcuna interazione dell'utente, in quanto i dispositivi devono scoprirsi e comunicare in maniera del tutto autonoma e trasparente.

Utilizzando la modalità legacy, un dispositivo crea un P2P Group fungendo da Access Point, mentre gli altri lo percepiscono e si connettono ad esso come un normale HotSpot 802.11. Questo dà la possibilità di connettere più client per Access Point, formando logicamente le topologie a stella, citate nel paragrafo di Architettura generale, nel Cap. 2.



**Figura 3.1:** Interazione utente per il consenso alla connessione

### 3.1.2 Il NetworkController

Il NetworkController si preoccupa di fornire gli strumenti necessari alla gestione degli attori coinvolti nel processo di networking e disseminazione locale delle informazioni. Anche questo componente è stato implementato seguendo il pattern Singleton, in modo tale da avere un unico e ben definito accesso alle risorse network. Infatti, la mansione principale di questa classe è proprio quella di esporre le istanze degli oggetti che permettono la realizzazione del processo di disseminazione locale. Tramite questa è possibile accedere all'oggetto NetworkServiceSearcher, NetworkAccessPoint e NetworkConnection. Il primo, come verrà illustrato in seguito, si preoccupa di segnalare la presenza di beacon di servizio nelle vicinanze, il secondo governa eventuali funzionalità da Access Point del device, mentre il terzo implementa funzionalità client per connettersi ad HotSpot nei paraggi. Inoltre, il NetworkController, registra in una variabile lo stato corrente della Network Machine State, la quale scandisce logicamente il processo di spreading. Infine, espone metodi per la registrazione e rilascio del BroadcastReceiver. Di seguito vengono elencate i metodi principali che la classe espone:

- startAccessPoint / stopAccessPoint : Metodi che decretano l'inizio, o la fine, del device come Access Point.
- startConnection / stopConnection : Metodi che cercano di connettersi, o scollegarsi, da un HotSpot.
- startServiceSearcher / stopServiceSearcher : Metodi che sanciscono l'inizio, o la fine, della ricerca di beacon di servizio nelle vicinanze.

### 3.1.3 NetworkConnection

Questa classe, concettualmente molto semplice, detiene la procedura necessaria per connettersi in maniera automatica ad un AccessPoint, presente nella lista ricavata dallo scan delle reti circostanti. Questa classe viene utilizzata quando un device, tramite l'arrivo di un beacon, capisce che vi è un

Access Point attivo e disponibile a soddisfare richieste di sincronizzazione. Infatti, questo, assumerà il ruolo di Client cercando di agganciarsi all'HotSpot individuato.

È necessario ricordare che, per effettuare queste procedure in maniera completamente automatica e trasparente all'utente, vi è la necessità di conoscere la passphrase per connettersi. Questa come vedremo, sarà ricavata dai beacon di servizio, già accennati in precedenza.

Di seguito viene mostrato il codice discusso:

...

```
public class NetworkConnection {

    public static String TAG= "[DEBAPP]NetworkConnection";

    WifiManager wifiManager = null;
    WifiConfiguration wifiConfig = null;
    Context context = null;

    public NetworkConnection(Context Context, String SSIS, String passphrase) {
        Log.d(TAG, "New WiFiConnection @ AP SSID: " + SSIS +
            " with psw: " + passphrase + " ..... ");
        this.context = Context;

        this.wifiManager = (WifiManager)this.context.getSystemService(
            this.context.WIFI_SERVICE);

        /* disable others */
        for (WifiConfiguration cnf:this.wifiManager.getConfiguredNetworks()) {
            this.wifiManager.disableNetwork(cnf.networkId);
        }
    }
}
```

```
/* set passphrase */
this.wifiConfig = new WifiConfiguration();
this.wifiConfig.SSID = String.format("\"%s\"", SSIS);
this.wifiConfig.preSharedKey = String.format("\"%s\"", passphrase);

/* try to reconnect with new config */
int id = this.wifiManager.addNetwork(this.wifiConfig);
this.wifiManager.enableNetwork(id, false);
this.wifiManager.reconnect();
}

public void Stop(){
    Log.d(TAG, "Remove connection with AP!!");
    this.wifiManager.disconnect();
}
}
```

### 3.1.4 Esplorazione dell'ambiente circostante

Questa classe ha lo scopo di capire la situazione nell'ambiente esterno, scoprendo quanti e quali attori vi sono nelle vicinanze e comportarsi di conseguenza. Ogni device, per decidere se fungere da AccessPoint o se connettersi come client ad un HotSpot già esistente, ha bisogno di comprendere ed essere avvisato della presenza o meno di altri host nell'ambiente locale.

#### Scoperta di peers

La classe `WiFiP2PManager` di Android, permette di gestire tramite chiamate API la connessione peer-to-peer tra device. Molte chiamate implementate fanno riferimento ad eventi asincroni, di conseguenza è stato necessario implementare Listener in modo tale da porre determinate logiche, al succedersi di determinati eventi. Uno di questi può essere individuato nella scoperta di peers, dove si prende nota dell'eventuale presenza di altri device

Android nell'ambiente circostante. Questo può essere ritrovato nella funzione `startPeerDiscovery` della classe `NetworkBeaconPeerSearcher`, dove viene richiamato il metodo `discoverPeers` del `WiFiP2PManager`. A questo punto il device è pronto a notificare la presenza di eventuali peers nelle vicinanze. Per catturare queste notifiche è stato necessario implementare un personale `PeerListListener`, il quale grazie al metodo `onPeersAvailable(WifiP2pDeviceList peers)` restituisce la lista dei peers presenti nei paraggi.

### Beacon di servizio

Sempre nell'ambito peer-to-peer, la tecnologia Android, permette di pubblicizzare un servizio ad altri device prima di un'effettiva connessione tra i dispositivi. La documentazione per developer di Android, specifica che questo meccanismo è stato fatto per individuare, in una fase iniziale, peer che esponessero determinati servizi, piuttosto che altri. Questa funzionalità è stata fondamentale per la realizzazione del progetto, in quanto permette di fare assunzioni prima di effettuare tentativi di connessione tra peer. In un primo momento si è pensato di inserire le informazioni relative agli eventi parcheggio direttamente all'interno di questi pacchetti, in quanto vi è la possibilità di inserire un tipo di dato `Map<String, String >`. Con questo tipo di meccanismo si sarebbe potuto attuare il cosiddetto piggybacking, inserendo il vero payload direttamente dentro a questi pacchetti di servizio. Questa strategia, oltre ad abbassare notevolmente la complessità, avrebbe permesso di scambiare dati in broadcast tra device, senza che questi instaurassero una connessione. L'implementazione però non è resa possibile a causa del basso contenuto, in termini di dimensione, che il tipo di dato `Map<String, String >` può contenere. Lo standard consiglia di tenere questo contenuto sotto i 200 bytes e non è raccomandato superare i 1300 bytes. Provando empiricamente, si è potuto confermare la restrizione appena citata. Gli `ArrayList` degli eventi che vogliamo sincronizzare hanno un contenuto maggiore del limite tecnologico imposto, per questo motivo si è dovuto virare sul meccanismo di `AccessPoint` e connessioni automatiche. Questi pacchetti risultano

comunque determinanti, in quanto incapsuleranno la stringa passphrase, necessaria ai client per connettersi in maniera automatica all'HotSpot esposto. Tornando al progetto, è possibile scoprire questi servizi tramite il metodo `startBeaconDiscovery`, il quale chiama a sua volta il metodo `discoverServices` del `WifiP2PManager`. Il Listener, relativo alla scoperta di un servizio, riguarda la creazione di un oggetto `DnsSdServiceResponseListener`, il quale implementando il metodo `onDnsSdServiceAvailable`, fornisce in input tutte le informazioni necessario sul servizio appena scoperto (ad esempio nome, tipo, device).

### Logiche di scoperta

Nei precedenti due sottoparagrafi si è cercato di illustrare le funzionalità principali che si ritrovano nella classe `NetworkBeaconPeerSearcher`, illustrando le funzioni API chiave utilizzate. Ora si cercherà di spiegare la logica che è stato necessario implementare e come la scoperta di peers e servizi si interfacciano tra loro. Quando un device scopre la presenza di almeno un peer attorno a lui, si mette immediatamente all'ascolto di eventuali servizi esposti da altri utenti. Nel momento in cui si scoprisse un servizio idoneo che indichi la presenza di un `D2DSmartParking AccessPoint`, in attesa di possibili sincronizzazioni, viene mandato un broadcast Intent tale da notificare al BroadcastReceiver di iniziare il processo di connessione all'HotSpot trovato. La classe che implementa tutte queste funzionalità è abbastanza ampia, per questo, di seguito, si mostra il solo codice relativo ai Listeners citati nel paragrafo.

```
...  
peerListListener = new WifiP2pManager.PeerListListener() {  
    public void onPeersAvailable(WifiP2pDeviceList peers) {  
        int nPeers = 0;  
        for (WifiP2pDevice peer : peers.getDeviceList()) {  
            nPeers++;  
            Log.d(TAG, "PEER FOUND: " + peer.deviceName);  
        }  
    }  
}
```

```
}
/* if there is someone, looking for D2DSP service */
if(nPeers > 0){
    Log.d(TAG, "Search for a D2DSP service...");
    NetworkController.getInstance().networkStateMachine =
        Constants.STATE_ACCESS_POINT_PEERS;
    startBeaconDiscovery();
} else { /* else search peers */
    Log.d(TAG, "Search for peers...");
    NetworkController.getInstance().networkStateMachine =
        Constants.STATE_ACCESS_POINT_NO_PEERS;
    startPeerDiscovery();
}
}
};
...
serviceListener = new WifiP2pManager.DnsSdServiceResponseListener() {
    public void onDnsSdServiceAvailable(String serv, String type,
                                       WifiP2pDevice dev) {
        if (serviceType.startsWith(NetworkTestFragment.SERVICE_TYPE)) {
            if(broadcaster != null) {
                Log.d(TAG, "find a D2DSP beacon!");
                Intent intent = new Intent(Constants.INTENT_D2D_AP_FOUND);
                intent.putExtra(Constants.INTENT_D2D_AP_ACCESSSDATA, serv);
                context.sendBroadcast(intent);
            }
        } else {
            Log.d(TAG, "This service is not for me!");
        }
    }
};
/* continue search peers */
startPeerDiscovery();
```



```
    }  
};  
...
```

### 3.1.5 Funzionalità Access Point-like

Le funzionalità necessarie ad un device per interpretare il ruolo di Access Point, sono definite nella classe `NetworkAccessPoint`. Queste, sostanzialmente, riguardano la creazione spontanea e autonoma di un WiFi Direct Group, con la conseguente autoelezione a Group Owner. La classe implementa delle interfacce astratte Android, le quali permettono di catturare segnali relativi alla gestione del WiFiDirect Group. In particolare, l'interfaccia `WifiP2pManager.GroupInfoListener` permette di essere notificati quando il Gruppo è stato creato. Questo è permesso dal metodo `onGroupInfoAvailable(WifiP2pGroup group)`, dove in input vengono date tutte le informazioni necessarie sul gruppo appena formato. A questo punto all'interno del metodo si richiama una procedura `startBeaconAP` per creare ed esporre il servizio di notifica, in modo tale da che gli utenti esterni riescano, tramite questo beacon, a connettersi automaticamente all'HotSpot. Oltre a questa operazione viene fatto partire il meccanismo per ricevere connessioni in input tramite Socket, operazione che verrà approfondita in seguito. Come citato in precedenza, il beacon incorpora l'informazione di passphrase in modo tale che la connessione possa avvenire in maniera automatica tramite la classe `NetworkConnection`. Il codice sottostante mostra la creazione del gruppo tramite il metodo `createGroup` e l'esposizione del servizio in merito alla notifica della corretta creazione della rete:

```
...  
/* Creation of Group */  
public void start() {  
  
    p2p = (WifiP2pManager) NetworkController.getInstance().managerP2P;
```

```

if (p2p == null) {
    Log.d(TAG, "Wi-Fi Direct NOT SUPPORTED, BYE!");
} else {
    channel = NetworkController.getInstance().channel;
    /* WiFi Direct Group creation */
    p2p.createGroup(channel, new WifiP2pManager.ActionListener() {
        public void onSuccess() {
            Log.d(TAG, "WiFiDirect Group created!!");
        }
        public void onFailure(int reason) {
            Log.d(TAG, "Error in creation, error code " + reason);
        }
    });
}
}
...
@Override
public void onGroupInfoAvailable(WifiP2pGroup group) {
    try {
        Log.d(TAG, "WiFi Direct Net is created, expose beacon service...");
        ...
        startBeaconAP("D2DSP:" + group.getNetworkName() + ":" +
            group.getPassphrase() + ":" + netAddress);
        ...
        /* Start Socket Server for synchronizations with clients */
        if(NetworkController.getInstance().server == null) {
            NetworkController.getInstance().server = new Server();
        }
    } catch(Exception e) {
        Log.d(TAG, "onGroupInfoAvailable, error: " + e.toString());
    }
}

```

```
}  
...  
private void startBeaconAP(String instance) {  
    /* beacon fot notify client of Access Point presence */  
    Map<String, String> record = new HashMap<String, String>();  
    record.put("D2DSP", "apactive");  
  
    WifiP2pDnsSdServiceInfo service =  
        WifiP2pDnsSdServiceInfo.newInstance(  
            instance, Constants.SERVICE_TYPE, record);  
  
    p2p.addLocalService(channel, service,  
        new WifiP2pManager.ActionListener() {  
        public void onSuccess() {  
            Log.d(TAG, "Beacon is in air!");  
        }  
        public void onFailure(int reason) {  
            Log.d(TAG, "Adding local service failed, error code " + reason);  
        }  
    });  
}  
...  

```

### 3.1.6 Canali di comunicazione

Una volta che l'Access Point crea la rete ed altri peer riescono a connettersi, identificandosi con un indirizzo IP, si può passare all'apertura dei canali di comunicazione ed alla sincronizzazione degli eventi di parcheggio/rilascio. I canali di comunicazione si basano su Socket serializzate, permettendo di inviare e ricevere, con un semplice casting, direttamente degli `ArrayList<ParkEvent>`. L'AccessPoint, appena notificato dell'avvenuta creazione del gruppo, richiama la creazione di un oggetto di tipo Server, il

quale istanzia una `ServerSocket`, mettendosi in ascolto di eventuali richieste Socket dai client, con il metodo `accept()`. I clients invece, quando notificati dal `BroadcastReceiver` dell'avvenuta connessione all'`HotSpot`, creano un oggetto `Socket` con l'indirizzo IP del Server e la porta idonea. Ora che il canale di comunicazione bidirezionale è stato messo in piedi, il Server invierà il proprio `ArrayList` nell' `ObjectOutputStream` della `Socket`, grazie al metodo `writeObject`. Il Client coinvolto, vedendosi arrivare le informazioni nel canale `ObjectInputStream`, le memorizza in una variabile ed invia le proprie, ripetendo il procedimento sopradescritto a parti inverse. Una volta chiuso il canale, viene fatto il merge delle entries ricevuto con quelle già presenti in memoria. Si precisa che tutti i meccanismi descritti vengono gestiti tramite `Thread`, in modo tale che il processo di sincronizzazione non risulti bloccante. Anche le classi `Client` e `Server` sono corpose dal punto di vista del codice, di conseguenza si illustra il codice dei passi salienti.

### CLIENT SIDE ###

```
public class Client implements Runnable {  
    ...  
    public void run() {  
        try {  
            ...  
            Socket socket = new Socket(serverAddr, Constants.PORT);  
            ArrayList<ParkEvent> parkEventsFromOther;  
  
            while (true) {  
                ...  
                try {  
                    ObjectInputStream objectInStream =  
                        new ObjectInputStream(socket.getInputStream());  
                    parkEventsFromOther =  
                        (ArrayList<ParkEvent>) objectInStream.readObject();  
                    ...  
                }  
            }  
        }  
    }  
}
```

```
        break;
    } catch (Exception e) {
    }
}
...
while(run) {
    try {
        ObjectOutputStream objectOutputStream =
            new ObjectOutputStream(socket.getOutputStream());
        objectOutputStream.writeObject(Controller.getInstance().parkEvents);
        break;
    } catch (IOException e) {
        run = false;
    }
}
socket.close();
/* merge new entries*/
Controller.getInstance().mergeEntries(parkEventsFromOther);
/* register sync */
Controller.getInstance().insertSynchronizationOnSQLiteDB(
    false, macWithSync);
/* change stati in NSM*/
NetworkController.getInstance().networkStateMachine =
    Constants.STATE_SYNCED_CONNECTED;
} catch (Exception e) {
}
}
}
### SERVER SIDE ###
...
/* Thread that attend for client socket requests */
```

```
private class SocketServerThread extends Thread {

    @Override
    public void run() {
        try {
            /* create ServerSocket using specified port */
            serverSocket = new ServerSocket(Constants.PORT);
            while (true) {
                Socket socket = serverSocket.accept();
                /* thread for socket sync with client - logic of
                 socketServerReplyThread are same of Client Side */
                SocketServerReplyThread socketServerReplyThread =
                    new SocketServerReplyThread(socket);
                socketServerReplyThread.run();
            }
        } catch (IOException e) {
        }
    }
}

...
```

### 3.1.7 L'ascoltatore BroadcastReceiverManagement

Un componente fondamentale, a livello implementativo, è senza dubbio il BroadcastReceiverManagement. Questo estende la classe Android BroadcastReceiver ed è in grado di ricevere avvisi sia dall'app stessa, che da componenti esterni di sistema. Per avvisi si intendono Intent, i quali vengono catturati dal metodo onReceive della classe. Questi oggetti possono tranquillamente lavorare in background e ricevere aggiornamenti anche se l'applicazione è chiusa. Si è definito fondamentale in quanto qui vengono prese tutte le decisioni riguardanti il networking, in base ai segnali ricevuti automaticamente in input. La classe può essere interpretata anche come un decisore,

che in base al succedersi di determinati eventi, compie precise azioni. Di seguito vengono illustrati i comportamenti fondamentali, in base a determinati Intent in input. Le istruzioni condizionali citate sono tutte all'interno del metodo onReceive.

- Tentativi di connessione :

All'arrivo di un custom Intent `INTENT_D2D_AP_FOUND`, il sistema capisce che è stato scoperto un `AccessPoint` attivo nelle vicinanze e cerca di connettersi ad esso, utilizzando la passphrase ricavata dal beacon di servizio.

```
...
if (Constants.INTENT_D2D_AP_FOUND.equals(action)) {
    /* get beacon data */
    String serv =
        intent.getStringExtra(Constants.INTENT_D2D_AP_ACCESSSDATA);
    String[] separated = serv.split(":");

    /* stop my Access Point functionalities */
    if(NetworkController.getInstance().networkConnection == null) {
        NetworkController.getInstance().stopNetworkAccessPoint();
        NetworkController.getInstance().stopNetworkBeaconPeerSearcher();
        /* Now I'm a simple Client */
        String networkSSID = separated[1];
        String networkPass = separated[2];
        String ipAddress   = separated[3];

        Log.d(TAG, "start new connection @ " + networkSSID);
        /* start connection mechanism */
        NetworkController.getInstance().networkConnection =
            new NetworkConnection(ctx, networkSSID, networkPass);
        /* networkStateMachine update */
    }
```

```

        NetworkController.getInstance().networkStateMachine =
            Constants.STATE_CONNECTION;
    }
}
...

```

- Gestione dello stato di connessione :

L'intent `WifiManager.NETWORK_STATE_CHANGED_ACTION` decreta un cambiamento di stato della connessione WiFi del device. Se è avvenuta una disconnessione il device torna disponibile ad interpretare il ruolo di Access Point, mentre se è decretata la connessione ad un HotSpot inizierà il procedimento di sincronizzazione, lato Client.

```

...
if (WifiManager.NETWORK_STATE_CHANGED_ACTION.equals(action)) {
    NetworkInfo info =
        intent.getParcelableExtra(WifiManager.EXTRA_NETWORK_INFO);
    if(info != null) {
        if (info.isConnected()) {
            ...
        }else {
            /* connection with AP is lost */
            if(this.connetionState.equals(Constants.CONN_STATE_CONNECTED)){
                Log.d(TAG, "WiFi DIS-Connecting...");
                this.connetionState = Constants.CONN_STATE_DISCONNECTED;
            }
        }
        if(this.connetionState.equals(Constants.CONN_STATE_DISCONNECTED)){
            /* start access point funzionalities */
            ...
        }
    }
}

```



```

WifiInfo wiffo =
    intent.getParcelableExtra(WifiManager.EXTRA_WIFI_INFO);
if(wiffo != null){/* I'm now connected! */
    /* open client socket channel*/
    this.runThreadSync(context);
}
}
...

```

- Sincronizzazioni successive :

Come ribadito in fase di progettazione, se un Client mantiene la propria connessione attiva con l'Access Point per un certo periodo, anche dopo aver effettuato una prima sincronizzazione con esso, ha la possibilità di effettuare un successivo scambio dati, dopo una certa threshold. Il componente quindi, periodicamente, controlla lo stato della Network State Machine e se il device continua a trovarsi nello stato STATE\_SYNCED\_CONNECTED anche dopo la soglia stabilita, può effettuare un'ulteriore comunicazione con l'HotSport Server.

```

...
if(Constants.INTENT_D2D_CHECK_MACHINE_STATE.equals(action)) {
...
    /* check for threshold */
    boolean isPossibleSync = Controller.getInstance().isPossibleSync();
    String state = NetworkController.getInstance().networkStateMachine;
    if((state.equals(Constants.STATE_SYNCED_CONNECTED))&&(isPossibleSync)) {
        ...
        /* re-synchronized client side */
        this.runThreadSync(context);
    }
...
}

```

...

### 3.1.8 Il servizio background

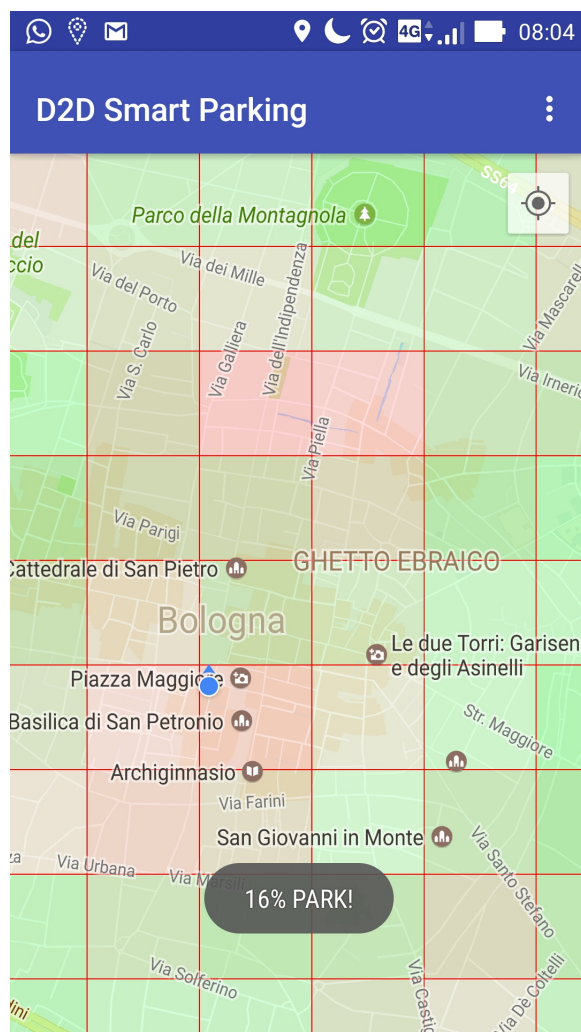
Il processo di comunicazione tra peers avviene in maniera completamente trasparente all'utente, per questo è stato necessario implementare un servizio che svolgesse il meccanismo di invio e ricezione dati, anche nel caso in cui l'applicazione venisse chiusa. Un Service è un componente Android, sprovvisto di interfaccia grafica, che si preoccupa di compiere procedure totalmente in background. Questo componente, presente nel package services, viene denominato NetworkAutoService, ed estende appunto la classe Service di Android.

- onCreate : Si preoccupa di creare il servizio e notificare l'utente di ciò con l'apparizione di un Toast.
- onStartCommand : Metodo che inizia l'esecuzione vera e propria del servizio in background. Qui, sostanzialmente, espone in beaconing la possibilità di fungere da AccessPoint e si mette alla ricerca di preesistenti HotSpot nei paraggi. Queste funzionalità verranno descritte meglio in seguito. Inoltre richiama il metodo privato startCheckNetworkTimer, illustrato tra poche righe. Richiamando tale funzione il servizio è attivo e pronto per il processo di sincronizzazione e disseminazione. Si ricorda che la partenza di questo è sancita dall'utente tramite interfaccia grafica, alla pressione di un pulsante.
- onDestroy : Metodo che ferma il servizio e libera le strutture dati necessarie al processo di networking. L'utente richiama implicitamente questo metodo alla pressione del bottone di Stop del servizio.
- startCheckNetworkTimer : Metodo privato che inizializza un oggetto Timer e lo schedula ogni 30s. Questi 30s possono essere interpretati logicamente come un time slot. L'azione schedulata, prevede l'invio di un Broadcast Intent, in modo tale che il Broadcast Receiver lo possa

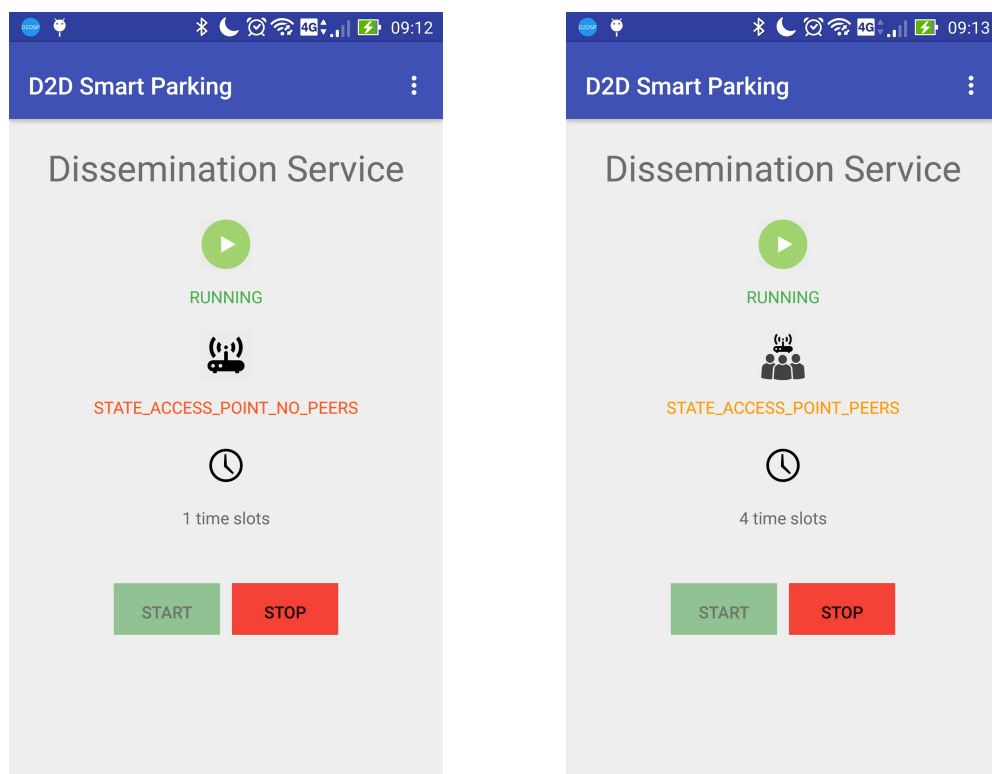
catturare e controllare lo stato della macchina a stati finiti, definita in fase di progettazione.

## 3.2 Screenshot

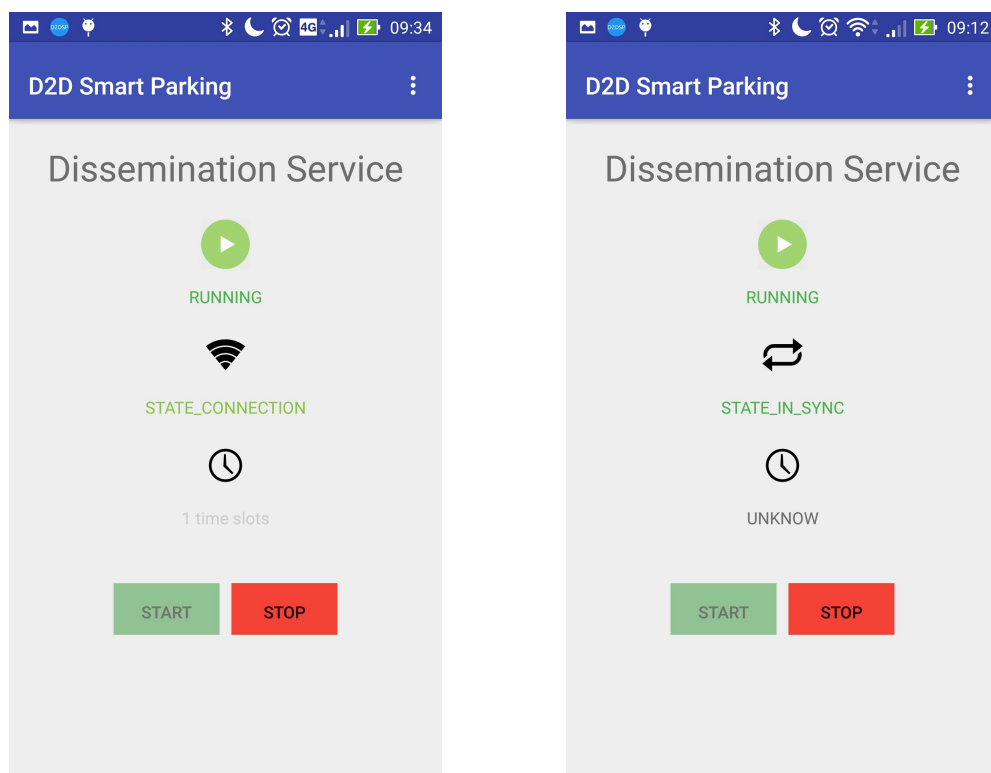
Di seguito sono mostrati gli screenshot dell'applicazione D2DSP:



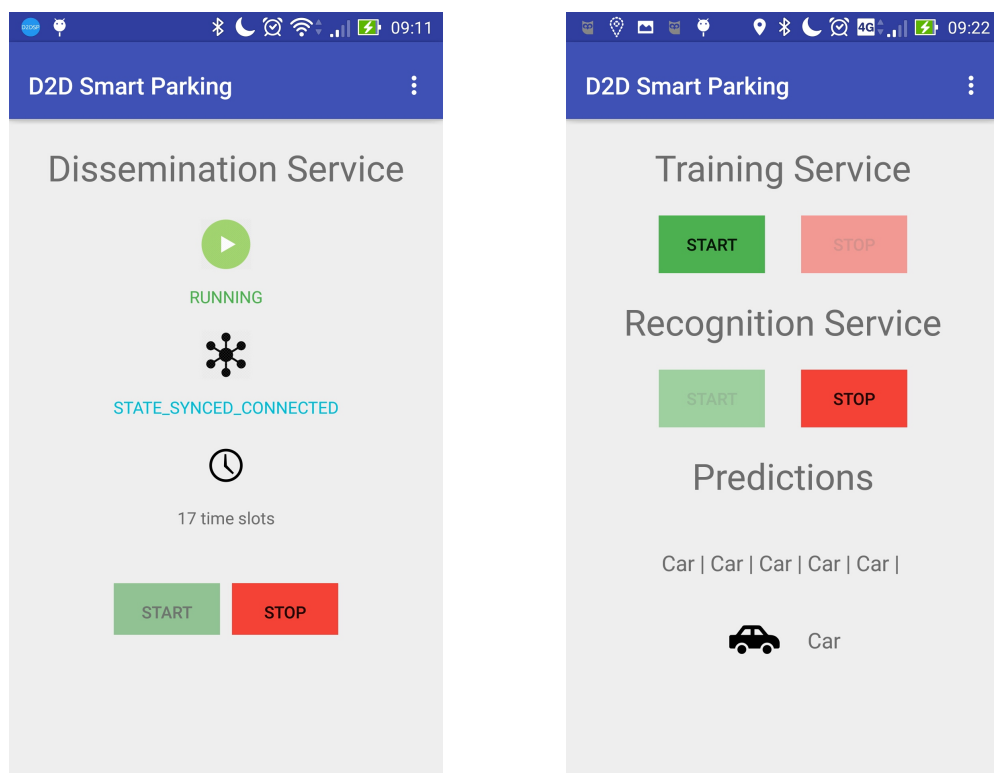
**Figura 3.2:** Mappa del centro di Bologna divisa in celle, ognuna colorata con la relativa probabilità di trovare parcheggio



**Figura 3.3:** Servizio di disseminazione attivo negli stati STATE\_ACCESS\_POINT\_NO\_PEERS e STATE\_ACCESS\_POINT\_PEERS



**Figura 3.4:** Didascalia comune alle due figure



**Figura 3.5:** Didascalia comune alle due figure

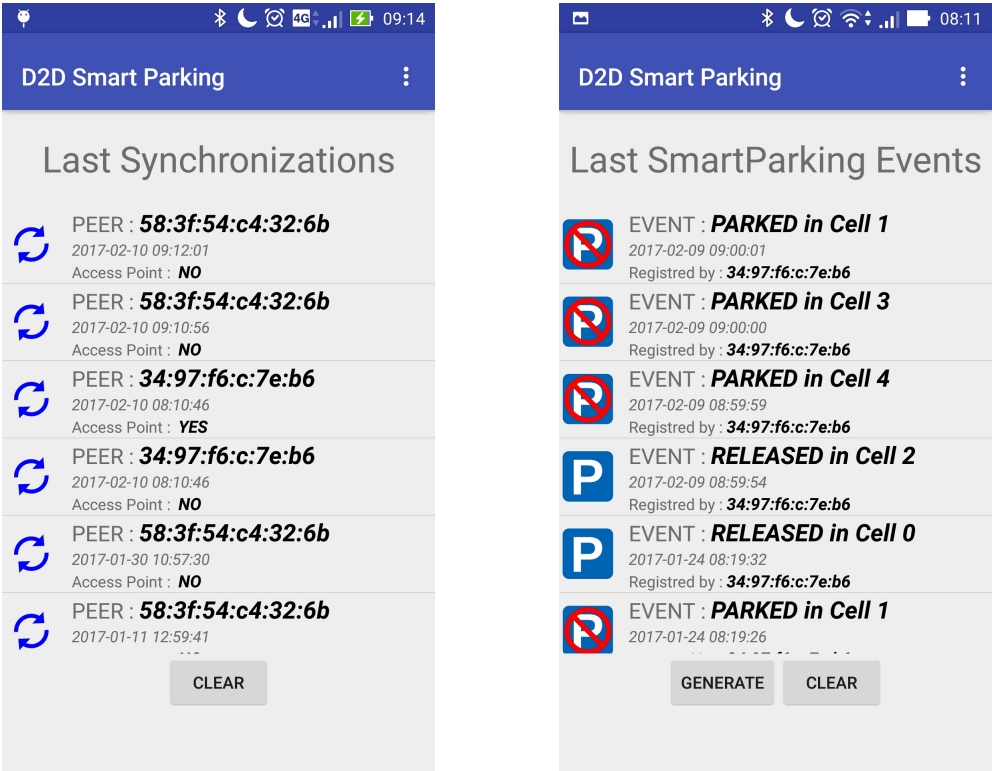


Figura 3.6: Didascalia comune alle due figure





# Capitolo 4

## ALTRA SEZ

### 4.0.1 Prima SottoSezione

**Teorema 4.0.1** (nome teorema).

*Primo Teorema.*

$$\frac{\partial}{\partial x} \boldsymbol{f}(x, y) = \frac{9}{10}$$

*Dimostrazione.* Il teorema è dimostrato

□

*Osservazione 1.* Prima Osservazione.

*Osservazione 2.* Seconda Osservazione.

**Definizione 4.1.** Altra definizione.

Ora vediamo un elenco numerato:

1. primo oggetto
2. secondo oggetto
3. terzo oggetto
4. quarto oggetto

**Esempio 4.1.** In questo primo esempio vediamo come si inserisce una figura:

**Figura 4.1:** legenda sotto la figura

**Teorema 4.0.2.**

*Secondo Teorema.*

*Dimostrazione.* Il teorema è dimostrato

□

## 4.1 Terza Sezione

**Esempio 4.2** (nome esempio).

Secondo Esempio.

Ora vediamo un elenco puntato:

- primo oggetto
- secondo oggetto

## 4.2 Altra Sezione

Vediamo un elenco descrittivo:

**OGGETTO1** prima descrizione;

**OGGETTO2** seconda descrizione;

**OGGETTO3** terza descrizione.

### 4.2.1 Altra SottoSezione

**SottoSubSez**

**SottoSottoSezione**

Questa sottosottosezione non viene numerata, ma è solo scritta in grassetto.

## 4.3 Altra Sezione

Vediamo la creazione di una tabella; la tabella 4.1 (richiamo il nome della tabella utilizzando la label che ho messo sotto): la facciamo di tre righe e tre colonne, la prima colonna “incolonnata” a destra (r) e le altre centrate (c):

|        |        |        |
|--------|--------|--------|
| (1, 1) | (1, 2) | (1, 3) |
| (2, 1) | (2, 2) | (2, 3) |
| (3, 1) | (3, 2) | (3, 3) |

**Tabella 4.1:** legenda tabella

## 4.4 Altra Sezione

### 4.4.1 Listati dei programmi

#### Primo Listato

In questo ambiente posso scrivere come voglio,  
lasciare gli spazi che voglio e non % commentare quando voglio  
e ci sarà scritto tutto.

Quando lo uso è meglio che disattivi il Wrap del WinEdt

# Conclusioni

Queste sono le conclusioni.

In queste conclusioni voglio fare un riferimento alla bibliografia: questo è il mio riferimento [3, 4].



# Appendice A

## Prima Appendice

In questa Appendice non si è utilizzato il comando:  
`\clearpage{\pagestyle{empty}\cleardoublepage}`, ed infatti l'ultima pagina 8 ha l'intestazione con il numero di pagina in alto.





## Appendice B

### Seconda Appendice



# Bibliografia

- [1] Primo oggetto bibliografia.
- [2] Secondo oggetto bibliografia.
- [3] Terzo oggetto bibliografia.
- [4] Quarto oggetto bibliografia.



# Ringraziamenti

Qui possiamo ringraziare il mondo intero!!!!!!!!!!  
Ovviamente solo se uno vuole, non è obbligatorio.