Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Progettazione ed implementazione di un sistema Smart Parking basato su comunicazione Device-To-Device

Relatore:
Chiar.mo Prof.
Marco Di Felice

Presentata da:
Andrea Sghedoni

Co-relatore:
Dott. Federico Montori

Sessione III
Anno Accademico 2015/2016

Alla mia famiglia...

Introduzione

Oggigiorno, il numero di veicoli presenti nelle metropoli ha raggiunto quantità tali da portare problemi di congestione del traffico. Si stima che una buona parte della congestione sia causata da automobilisti alla ricerca costante di un parcheggio libero. Questa problematica causa una serie di conseguenze negative, danneggiando progressivamente gli utenti e l'ambiente. La ricerca prolungata di uno slot libero comporta all'utente perdita di tempo e di denaro, contribuendo ad un peggioramento, anche se leggero, della qualità di vita nelle grandi città. Inoltre, sono evidenti le conseguenze negative sull'impatto ambientale, dove la qualità dell'aria viene messa costantemente a rischio dall'enorme quantità di carburante che viene consumato, producendo emissioni di agenti inquinanti nell'atmosfera. Enti governativi stanno iniziando a prendere in considerazione il tema del parcheggio, cercando di trovare le giuste soluzioni per migliorare la situazione. In questo senso, numerose proposte sono già presenti in letteratura, basandosi sui concetti innovativi di Smart Parking, IoT(Internet of Things) e Smart City. Infrastrutture informatiche, comunicazione tra dispositivi eterogenei e sensoristica sono tutti elementi che possono favorire un miglior utilizzo dei parcheggi, fornendo agli utenti importanti indicazioni sullo stato degli slot, in una determinata zona. I sistemi di Smart Parking diventano fondamentali soprattutto se si considerano parcheggi su strada (on-street parking), i quali sono limitati e non circoscritti in precisi punti (al contrario di parcheggi-garage e sistemi a circuito chiuso, in cui tramite semplici apparecchiatura è possibile contare il numero di veicoli in entrata ed in uscita). Molte soluzioni proposte

prevedono l'utilizzo di sensori su strada o telecamere per comprendere l'occupazione o meno di un determinato parcheggio. Mentre per la disseminazione delle informazioni raccolte, si propongono sistemi decentralizzati basati sull'IoT e WSN(Wireless Sensor Network), oppure classici sistemi centralizzati. Tutti i componenti citati comportano un investimento iniziale enorme per apparecchiature ed installazione, senza contare i futuri e necessari costi di manutenzione. L'investimento è ovviamente un grosso limite per i governi che spesso non sono intenzionati ad utilizzare ingenti fondi per questi scopi. Il progetto riguarda l'implementazione prototipale di un sistema Smart Parking in ambiente Android, dove le informazioni sui parcheggi vengono propagate in modalità Device-To-Device tra gli Smartphone utente. Il sistema che viene proposto non necessita di alcuna infrastruttura particolare e si pone l'obiettivo di fornire all'utente una stima sulle zone più dense di parcheggi liberi della città. Grazie al concetto di Crowdsensing, gli utenti condividono le proprie rilevazioni con altri utenti, cercando di restare aggiornati sulla situazione parcheggi in città. Ogni Smartphone, avendo la possibilità di rilevare il tipo di mobilità dell'utente tramite i propri sensori, riesce a dedurre in probabilità un evento di parcheggio, piuttosto che di rilascio. I dati vengono disseminati localmente nell'ambiente ad altri device in maniera totalmente decentralizzata, permettendo agli utenti di restare informati sulla situazione parcheggio nelle zone limitrofe. L'utilizzo della tecnologia *WiFi Direct* permette la comunicazione wireless tra dispositivi mobili. Il servizio, una volta avviato, non avrà bisogno di alcuna interazione utente, in quanto il processo di rilevazione e di comunicazione sono stati automatizzati. Oltre all'implementazione dell'applicazione è stato necessario studiare l'efficacia del processo di spreading. La logica di disseminazione è stata testata su una porzione del centro di Bologna, grazie ai simulatori *OMNeT++*, *SUMO* ed al framework *Veins*. Per ottenere risultati attendibili, sono stati sviluppati moduli che replicassero fedelmente il comportamento dell'applicazione Android in uno scenario urbano.

Il documento viene organizzato in quattro capitoli, secondo la seguente strut-

tura. Nel primo capitolo viene analizzato lo stato dell'arte per quanto riguarda l'argomento in questione. Il secondo capitolo illustra i passi di progettazione necessari alla realizzazione del sistema. Il terzo capitolo riguarda l'implementazione dell'applicazione Android, entrando nei dettagli più tecnici. Il quarto ed ultimo capitolo mostra la valutazione del sistema, dove i risultati sono stati ottenuti simulando il funzionamento del sistema in una città metropolitana.

Indice

Elenco delle figure

Elenco delle tabelle

Parte I

Stato dell'arte

Capitolo 1

Stato dell'arte

Il seguente capitolo illustra il contesto in cui si colloca il progetto di tesi ed una serie di lavori correlati sull'argomento dello Smart Parking.

1.1 Sistemi di Trasporto Intelligenti

Il Ministero delle Infrastrutture e dei Trasporti definisce gli Intelligent Transport System (ITS, in italiano Sistemi di Trasporto Intelligenti) come *"...range di strumenti per la gestione delle reti di trasporto, così come i servizi per i viaggiatori. Gli strumenti ITS sono basati su tre aspetti centrali: acquisizione, elaborazione e diffusione delle informazioni e da un minimo comune denominatore quale è l'integrazione. I processi di acquisizione dei dati, l'elaborazione e l'integrazione degli stessi e la catena delle informazioni da fornire agli utenti del sistema di trasporto sono il cuore dei sistemi ITS"*[1]. Lo sviluppo esponenziale dell'informatica e delle reti di comunicazione ha portato alla nascita di numerose applicazione, ad alto contenuto tecnologico, per la gestione ed il controllo nell'ambito trasporti. La gestione del traffico veicolare (acquisizione dei dati di traffico, controllo semaforico, controllo degli accessi, gestione dei parcheggi), del trasporto collettivo e della logistica (indicazioni di percorso e instradamento parcheggi, sistemi di distribuzione delle merci, sistemi misti merci/passeggeri, ecc...) sono esempi tangibili di

come i sistemi ITS stiano diventando uno strumento prezioso per il controllo della mobilità. Le città metropolitane sono sovraffollate di veicoli, portando a costanti situazioni di inefficienza dei trasporti, congestione del traffico ed un aumento dei tempi di percorrenza per giungere a destinazione. L'urbanizzazione e la crescente densità di popolazione nelle grandi città potrà soltanto peggiorare la situazione. Il 54% della popolazione mondiale risiede in città metropolitane e si stima che possa diventare il 66% entro il 2050[2]. I sistemi

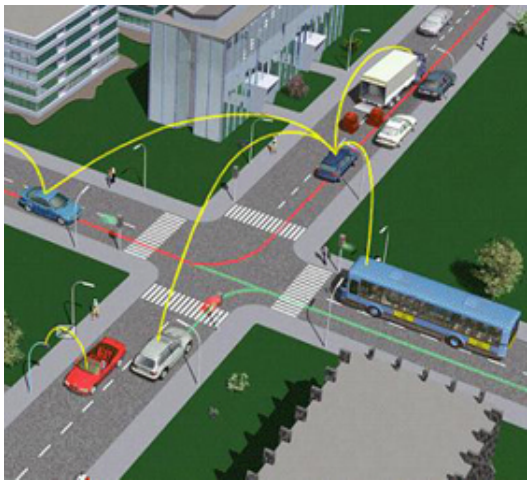

Figura 1.1: Esempio di ITS

ITS si pongono l'obiettivo di migliorare la qualità dei trasporti in termini di riduzione delle congestioni, rischi di incidente e situazioni di emergenza. Per fare ciò vengono sfruttati meccanismi intelligenti, in grado di fare determinate assunzioni sulla base delle informazione scambiate tra veicoli che si trovano in una determinata zona. Gli utenti, consultando i dati sul traffico forniti da sistemi ITS, potrebbero variare il loro comportamento, permettendo così di smaltire le congestioni e favorire la viabilità generale.

1.2 Crowdsensing

Per Crowdsensing si definisce quell'attività di condivisione e raccolta dati, utilizzando i vari sensori presenti all'interno dei dispositivi mobili. Lo scopo

di ogni device deve essere quello di condividere con molti utenti le proprie informazioni. In questa maniera, si potrà giungere ad una conoscenza condivisa tra gli utenti ed ottenere, in maniera intelligente, i risultati prefissati. Il Crowdsensing si suddivide principalmente in 2 categorie[3]:

- *Partecipatory Crowdsensing*: l'utente viene coinvolto direttamente nella fase di propagazione dei dati.

- *Opportunistic Crowdsensing*: la propagazione avviene in maniera automatica e trasparente all'utente.

La seconda categoria, dove l'utente deve solo accettare la condivisione dei propri dati, è quella maggiormente utilizzata nelle applicazioni mobili, in quanto è quella che garantisce una buona quantità di raccolta dati.
Il Crowdsensing può essere uno strumento molto potente per ottenere risultati che possono essere ricavati soltanto dall'aggregazione di dati collettivi. Numerosi contributi, in letteratura, si basano sulla condivisione massiva di dati. Ad esempio, il progetto in [4] utilizza il concetto di Crowdsensing per la costruzione intelligente di mappe indicanti il numero di parcheggi disponibili per una determinata strada. Prendendo un numero elevato di utenti in una determinata zona e condividendo i dati di dove essi parcheggiano abitualmente, è possibile tracciare le zone adibite al parcheggio di una determinata strada. Il sistema presentato in [8], vuole fornire agli utenti la situazione di occupazione del parcheggio universitario tramite i dati raccolti dagli utenti. Ogni studente, quando lascia il parcheggio, contribuisce alla conoscenza globale con la propria visione del parcheggio (pieno, semi-pieno, vuoto). I lavori proposti sul tema dello Smart Parking in [6][9], mostrano quanto sia fondamentale il tasso di partecipazione utente. Più utenti condividono le proprie informazioni, più l'accuratezza dei risultati sarà precisa. In particolare viene mostrato che un alto tasso di partecipazione corrisponde ad una diminuzione notevole del tempo di ricerca di un parcheggio. Ogni utente condivide i propri dati sapendo di contribuire al risultato generale, avendo l'interesse nel

migliorarne l'accuratezza. Le applicazioni Crowdsensing rappresentano una possibilità di migliorare la quotidianità dell'individuo.

1.3 Smart City

La *IEEE* definisce la Smart City come ambiente sostenibile in grado di ridurre i consumi e fornire una qualità di vita migliore ai cittadini[18]. Il mondo tecnologico, i governi e la società dovrebbero incidere in maniera positiva nel processo di urbanizzazione, riuscendo a proporre le seguenti caratteristiche nella città intelligente:

- smart mobility
- smart economy
- smart environment
- smart people
- smart living
- smart governance

La Smart City si pone l'obiettivo di semplificare ogni operazione della città, grazie agli ultimi sviluppi tecnologici. Soprattutto la recente espansione del paradigma *IoT* (Internet of Things, in italiano Internet delle Cose) ha permesso l'apertura verso nuove applicazioni, in grado di portare miglioramenti nella vita metropolitana. In [19], si definisce l'Internet delle Cose come *"l'interconnessione in rete di oggetti di uso quotidiano, formando una rete distribuita di dispositivi, comunicanti con l'uomo o con altri dispositivi"*.
Ogni dispositivo potrà aumentare il proprio livello di intelligenza grazie all'interconnessione ed alle informazioni derivanti da altri device.
Il progetto di tesi può essere visto come un primo passo verso il tema dello Smart City, dove numerosi dispositivi vengono coinvolti per migliorare un aspetto quotidiano dei cittadini.

1.4 Smart Parking

Il problema del parcheggio può essere visto come una grossa sfida all'interno delle città intelligenti, definite nella sezione 1.3. Con il termine Smart Parking si intendono tutte quelle soluzioni che potrebbero favorire l'utente nell'attività di parcheggio. Secondo recenti studi, la maggior parte della congestione veicolare nelle grandi città è causata dai parcheggi. Gli utenti passano molto tempo nel cercare uno slot libero, soprattutto se si considerano parcheggi su strada (on-street parking).
Le conseguenze negative di tutto ciò si possono riassumere in:

- perdita di tempo e denaro
- inquinamento ambientale
- peggioramento della qualità di vita

Lo studio in [4], quantifica gli effetti negativi evidenziando come in una porzione di Los Angeles, nel periodo di un anno, le macchine abbiano compiuto una distanza pari a 37 giri intorno al mondo, soltanto per le attività di ricerca parcheggio. Con il conseguente consumo di 47000 galloni di gasolio, tramutati in 730 tonnellate di CO_2 (diossido di carbonio) emesso nell'aria. Un'altra analisi molto interessante viene fatta in [5], dove si dimostra come gli utenti, non trovando parcheggio in un tempo ragionevole, possano assumere comportamenti negativi. Molte persone sarebbero disposte a parcheggiare illegalmente o addirittura rinunciare all'attività da compiersi. Lo stato dell'arte presenta numerosi lavori sul tema dello Smart Parking, proponendo idee sempre più innovative per limitare gli effetti negativi appena descritti.
I sistemi di Smart Parking possono essere suddivisi in 3 macrocategorie:

- *Sistemi basati su reti di sensori*:
  I progetti mostrati in [5][10][11][14][15] si affidano a Wireless Sensor Network (WSN), dove ogni sensore viene posizionato in prossimità del parcheggio (come mostrato in figura 1.2) ed instrada le proprie informazioni verso una struttura centralizzata. L'efficienza di questi sistemi

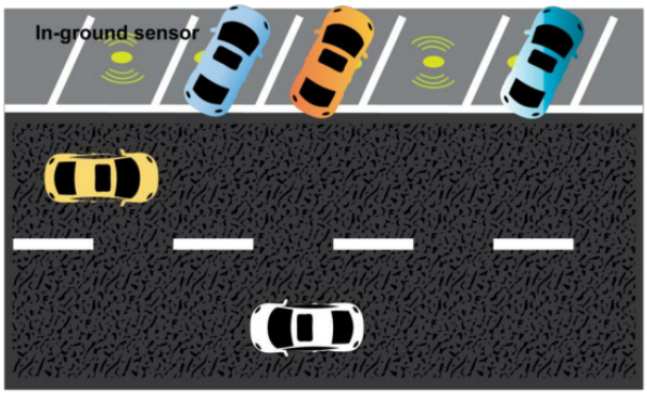

Figura 1.2: Sistema di Smart parking con sensori in-ground

è molto alta, ma il costo per l'installazione e la manutenzione (come riportato anche nell'introduzione) sono un limite per la loro diffusione.

- *Sistemi basati su computer vision*:
  Come citato in [16] molti sistemi si basano su tecniche di Video Image Processing (VIP) per dedurre l'occupazione o meno dei parcheggi. I risultati vengono ottenuti processando le immagini (ricavate da telecamere) con algoritmi basati sull'Intelligenza Artificiale. Anche per questi sistemi il costo dell'infrastruttura può essere notevole.

- *Sistemi di Crowdsensing e Crowdsourcing*:
  Questi sistemi permettono di ottenere risultati soddisfacenti partendo dal contributo degli utenti stessi. In [4] si cerca di fornire all'utente una mappa dettagliata della quantità di parcheggi per strada. La mappa può essere un'informazione utile per l'utente, il quale riesce quantomeno a farsi un'idea sulle strade più dense di parcheggi on-street. I progetti in [2][7][8][9] forniscono all'utente, tramite l'ausilio di mappe, indicazioni riguardo la situazione parcheggi nelle vicinanze. Gli utenti che compiono un'attività di parcheggio, condividono i propri dati (ottenuti tramite sensori dello Smartphone o altro) con la comunità, evitando congestione del traffico stradale. Questi lavori possono essere correlati all'attività di tesi presentata in questo documento, dove si

cercherà di fornire agli utenti preziose informazioni sui parcheggi vicini, senza l'aiuto di architetture centralizzate o WSN.

1.4.1 Propagazione dell'informazione

Una parte fondamentale all'interno dei sistemi Smart Parking riguarda la propagazione dei dati tra i diversi utenti. Innanzitutto è bene distinguere la comunicazione secondo due paradigmi:

- *centralizzata*: classica struttura Client-Server, dove una struttura centralizzata raccoglie tutti i dati derivanti da diverse fonti della Smart City e fornisce i risultati ai client.
- *distribuita*: ogni dispositivo comunica con altri dispositivi cercando di propagare efficacemente l'informazione. L'assenza di una struttura centralizzata porta ogni nodo ad avere risultati parziali nel tempo. È necessario un meccanismo di propagazione efficiente perchè i device restino correttamente aggiornati.

Il secondo paradigma è quello che verrà utilizzato nel progetto di tesi, dove gli attori della rete saranno solamente dispositivi end-point. Risparmiando l'utilizzo di agenti centralizzati, si evitano importanti costi di installazione e gestione. Per propagare informazioni in maniera distribuita all'interno della città si considerano due tecnologie come il Vehicle-To-Vehicle[31] ed il Device-To-Device[32][13].

Sistemi Vehicle-To-Vehicle

Il tema dello Smart Parking porta a considerare la Smart City come una rete veicolare distribuita ed in queste condizioni è normale pensare ad una comunicazione Vehicle-To-Vehicle (V2V). Il V2V riguarda comunicazioni tra veicoli in movimento portanto alla creazione di reti spontanee, denominate Vehicular Ad-Hoc Networks (VANETs)[31]. Una VANET è un caso particolare di Mobile Ad-Hoc Network (MANET). La topologia della rete è

altamente dinamica in quanto i veicoli possono muoversi a velocità considerevoli ed in senso opposto, di conseguenza la comunicazione deve avvenire in pochi istanti. Le disconnessioni possono essere frequenti, considerando che i veicoli possono allontanarsi velocemente l'uno dall'altro proprio durante uno scambio dati. Il mondo del V2V prevede 3 categorie principali di comunicazione:

- *Vehicle-to-Vehicle (V2V)*: prevede la sola comunicazione tra veicoli.
- *Vehicle-to-Infrastructure (V2I)*: prevede la comunicazione tra veicoli ed infrastrutture stradali.
- *Hybrid (V2X)*: prevede comunicazioni sia V2V che V2I.

La comunicazione veicolare apre le porte a numerosi campi di applicazione. La sicurezza è sicuramente l'obiettivo principale, segnalando all'utente eventuali incidenti o situazioni di pericolo. Inoltre l'efficienza (ad esempio evitare percorsi congestionati), il comfort, l'intrattenimento (ad esempio condivisione di musica e file) e l'urban sensing (ad esempio monitoraggio smog) sono tutti possibili traguardi per il V2V. Le case automobilistiche stanno iniziando a predisporre le auto con sensori e sistemi di navigazione. Il prossimo step sarà integrare la comunicazione V2V, in quanto la maggior parte delle automobili in commercio non sono ancora idonee in questo senso.

Sistemi Device-To-Device

Un altro modo di propagazione distribuita può avvenire invece tramite Smartphone, utilizzando il paradigma Device-To-Device (D2D)[32][13]. Il D2D permette comunicazioni wireless tra device che sono nelle vicinanze. L'approccio non prevede alcuna infrastrutture per lo scambio dati al di fuori dei dispositivi mobili (come ad esempio Smartphone e Tablet). Il D2D è l'approccio preso in considerazione nel lavoro di tesi, sfruttando i dispositivi utente per la propagazione di informazioni, relativa alla situazione parcheggi, all'interno della Smart City.

Parte II

Progettazione ed Implementazione

Capitolo 2

Progettazione

Il seguente capitolo ha lo scopo di illustrare in dettaglio il goal del progetto, le problematiche che vi stanno dietro e la fase di progettazione su cui è stato necessario soffermarsi, per la successiva fase di implementazione. La mission, descrivendola in pochi concetti, è sostanzialmente quella di determinare quando un utente parcheggia o rilascia uno slot e disseminare queste informazioni in maniera totalmente distribuita, 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 la comunicazione avverrà esclusivamente Device-To-Device, utilizzando tecnologie presenti nei più comuni device in commercio. Quello che si vuole fornire all'utente sono le probabilità di trovare parcheggio nelle diverse zona della città. Dettagli tecnologici ed implementativi saranno comunque discussi nel prossimo capitolo. Di seguito verranno illustrate le principali strutture che compongono il sistema, descrivendone funzionalità, utilità ed integrazione con le restanti componenti.

2.1 Scopo e problematiche

Come già accennato 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, rileva un evento di parcheggio (o rilascio) e ne tiene traccia, salvandolo in memoria locale. Le informazioni sopradescritte 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 scelta progettuale, 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 maniera totalmente autonoma. Questo significa che l'utente, senza nessuna interazione con il device (per esempio mantenendolo in tasca), sincronizza comunque i propri dati con quelli di utenti circostanti. La funzionalità appena descritta, per renderla robusta, ha richiesto una fase implementativa consistente, la quale verrà illustrata tecnicamente nel Cap.3.

2.2 Scenario generale

Lo scenario che si va a considerare per l'utilizzo dell'applicazione è una città metropolitana. La città viene interpretata e suddivisa logicamente in una griglia composta da celle (quadranti), dove ognuna di queste viene identificata univocamente da una coppia di coordinate. Le coordinate possono essere considerate come indici matriciali che individuano una cella all'interno della matrice griglia della città. Si presuppone che il numero di parcheggi disponibili in una data cella sia noto a priori. Si considerano tutti gli slot disponibili nelle strade che sono raggruppate all'interno del quadrante considerato. Si ricorda che si fa riferimento a parcheggi pubblici su strada e non a strutture private o a circuito chiuso. L'applicazione, riconoscendo un evento di parcheggio o rilascio, identifica la cella in base alla posizione corrente e registra il dato nella propria memoria locale. Le assunzioni sul-

la probabile quantità di parcheggi liberi vengono calcolate sulla granularità della cella. In fase di progettazione, si è giunti alla conclusione che non è sufficiente scambiarsi il numero di parcheggi liberi/occupati derivanti da dati statistici, in quanto ogni 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. 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 entry 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. Per evitare che questo succeda, viene controllata la dimensione del database locale, ed eventualmente eliminate le entry meno aggiornate. Inoltre i peer possono effettuare sincronizzazioni soltanto dopo una certa threshold, espressa in secondi. Questi concetti sono stati parametrizzati in fase di implementazione, quindi possono essere giustamente settati in base all'ambiente in cui si intende utilizzare l'app.

2.3 Ruoli dei dispositivi

Il sistema progettato 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 esclusivamente due ruoli:

- **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.

- **ROLE CLIENT**:
  Se un device si accorge della presenza di un Hotspot nelle vicinanze, grazie al beacon sopracitato, può tentare di connettersi come client all'Access Point. Qualora riesca ad agganciarsi in modo stabile verranno aperti i canali di comunicazione, su cui effettuare una sincronizzazione tra i due device.

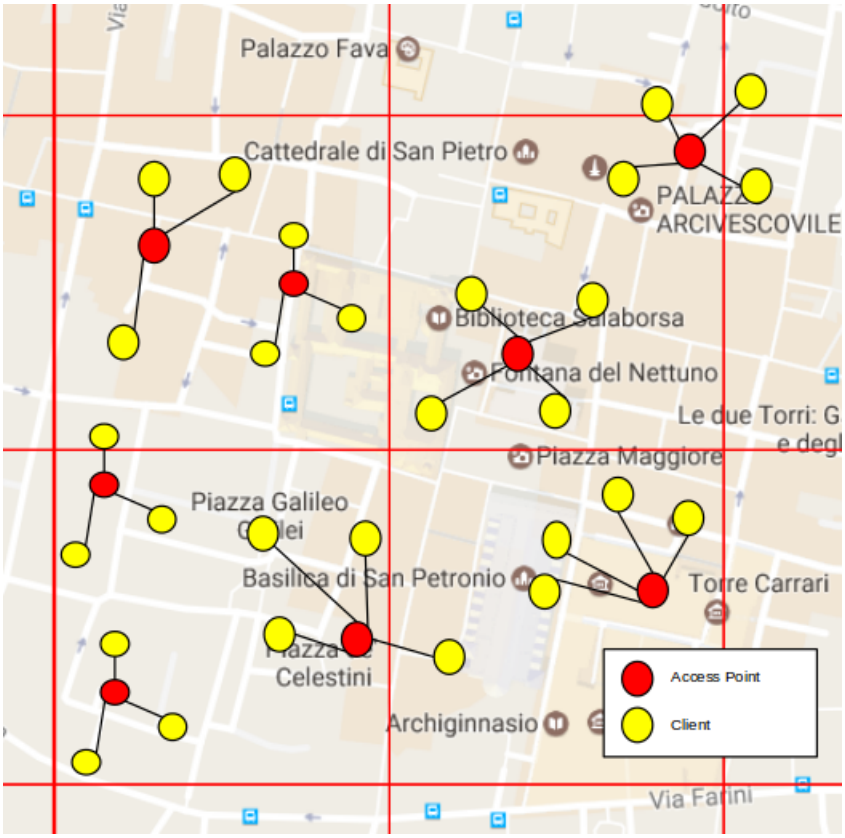

Figura 2.1: Rappresentazione dei diversi ruoli e della divisione in celle di una porzione di Bologna

A questo punto la topologia della rete generale può essere vista come hub di hub, ovvero tante piccole star, dove grazie al continuo movimento dei device si riuscirà ad ottenere una propagazione dell'informazione, interconnettendo appunto diverse star tra di loro, ad istanti di tempo differenti. Come si può comprendere dalla figura 2.1, i device muovendosi continuamente all'interno dello scenario (identificato in questo caso con la griglia città), formeranno continuamente nuove star, con attori sempre diversi. Questo fattore è determinante per il processo di propagazione delle informazioni, in quanto dati presenti in una star, verranno sincronizzati, in istanti successivi, con star diverse. L'analisi dello spreading delle informazioni, la sua efficacia ed accuratezza sono illustrate nel Cap.4, nel quale si cerca di capire se questo meccanismo è attuabile nella realtà.

2.4 Architettura software

Prima della fase di implementazione, è stato necessaria una fase di progettazione dell'architettura software. L'applicazione, per essere efficace nel meccanismo di spreading, necessita di numerosi componenti e soprattutto dell'interazione tra essi. Nella figura 2.2 sono stati messi i componenti principali che concettualmente compongono l'architettura e come questi collaborano. Di seguito vengono spiegati i compiti che ogni struttura deve compiere per la corretta esecuzione dell'applicativo:

- **Componente di Activity Recognition:**
  Il componente di activity recognition si preoccupa di rilevare gli eventi di parcheggio e rilascio. 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 fase 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 nel-

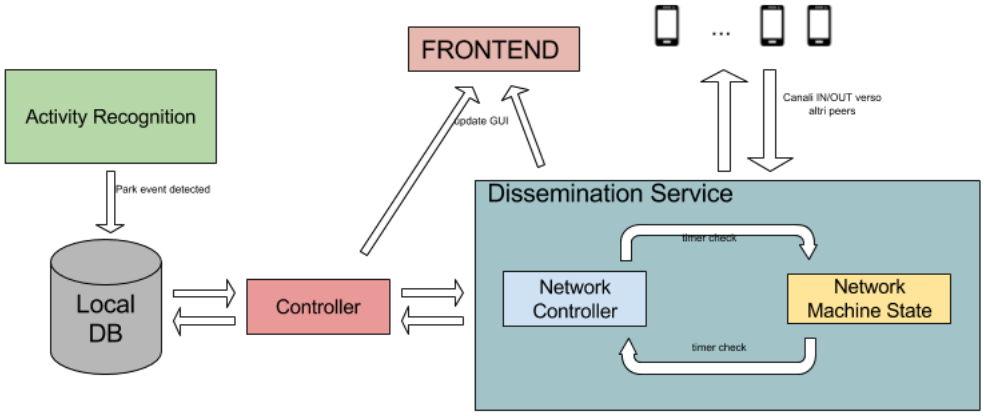

Figura 2.2: Rappresentazione dell'Architettura software

la prima riuscirà a fare assunzione sul tipo di attività che l'utente sta svolgendo. L'algoritmo *Random Forest* permette di classificare i dati grezzi in una precisa classe di movimento, tramite il massiccio utilizzo di alberi decisionali. 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 database locale. L'implementazione di questa parte non è stata necessaria perchè è stato possibile usufruire interamente del sistema descritto in [2], dove si mostra come l'algoritmo di recognition riesca a raggiungere una precisione di circa 90%.

- **Componente Database locale:**
  Il database locale si preoccupa di raccogliere 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 nella sezione 2.4.1.

- **Componente Controller:**
  Il Controller si preoccupa di scrivere, aggiornare e dare accesso alle in-

formazioni 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 Model-View-Controller (MVC), gestendo le classi del Modello.

- **Componente Frontend:**
  Il frontend è composto dalle view che verranno mostrate sul device Android dell'utente. 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 nelle vicinanze. Nella sezione 3.4 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 Cap.3. Il disseminatore è un servizio che gira in background e la logica è scandita da una Finite State Machine (FSM), dove in base allo stato in cui si trova verranno compiute azioni, piuttosto che altre. Quando viene rilevata una nuova connessione ad un peer, il componente si preoccupa anche di gestire il corretto svolgimento della sincronizzazione.

2.4.1 Organizzazione dell'informazione

L'organizzazione dei city data è ovviamente una fase determinante del progetto. 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 utente 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 Cap.3 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*: identifica univocamente la cella all'interno della griglia città. Informazione di tipo geografica.
- *event*: evento che può essere esclusivamente PARKED o RELEASED. Indica se l'entry fa riferimento ad una sosta o al rilascio di un parcheggio.
- *timestamp*: datetime che indica il momento in cui è stato performato l'evento in questione.
- *mac*: identificativo univoco del device che ha performato l'evento di parcheggio o di rilascio.

La PRIMARY KEY è composta da <u><*timestamp, mac*></u>. 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*: identificativo univoco del 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 settare 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.

2.4.2 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 Access Point nelle vicinanze.
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 in 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

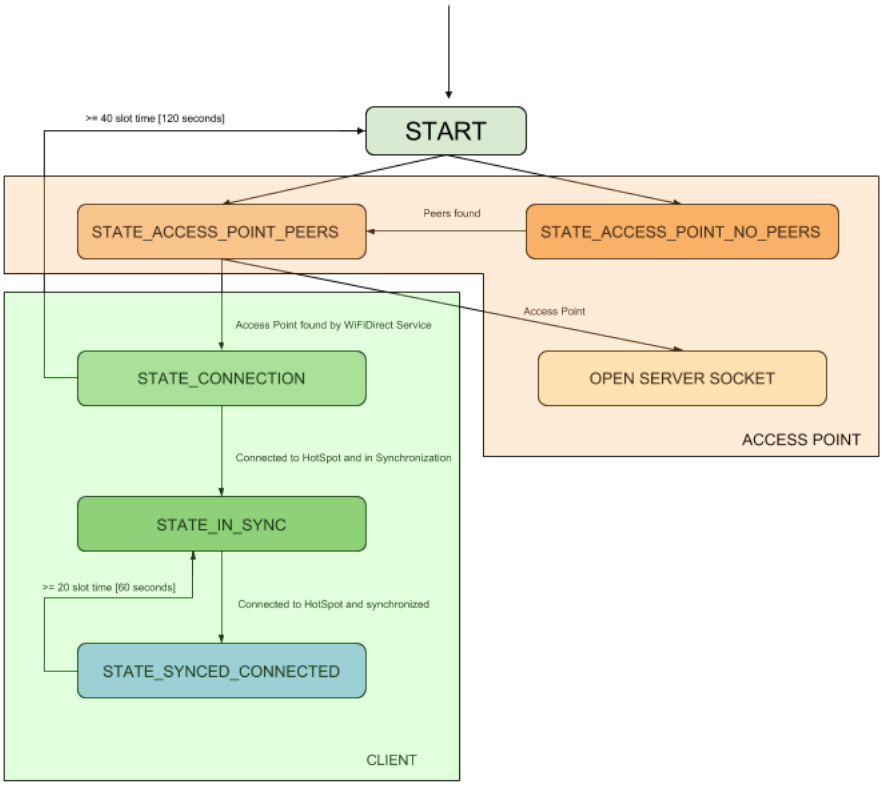

Figura 2.3: Network State Machine

indirizzo IP, verrà automaticamente aperto un canale di comunicazioni bidirezionali tra il client ed il server, iniziando così la fase di sincronizzazione (STATE_IN_SYNC).
Come si può notare dallo schema in figura 2.3, da STATE_CONNECTION vi è anche la possibilità di ripartire dall'inizio, nel caso in cui, il device non riesca a connettersi all'Hotspot entro una certa 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, si dà l'opportunità al client (dopo una certa threshold) di richiedere nuovamente una sincronizzazione.

2.5 Probabilità di occupazione di una cella

Man mano che le informazioni giungono nel database locale, si calcolano le probabilità di trovare parcheggio nelle celle della città. 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. 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'affidabilità delle probabilità fornite agli utenti. Il numero di slot totali in una generica cella i ($N_{\mathrm{i}}^{\mathrm{t}}$), noto a priori, lo si può interpretare anche come la somma tra numero di slot liberi ($N_{\mathrm{i}}^{\mathrm{f}}$) e numero di slot occupati ($N_{\mathrm{i}}^{\mathrm{o}}$). Il numero di slot occupati $N_{\mathrm{i}}^{\mathrm{o}}$ può essere ricavato dagli eventi di parcheggio e rilascio presenti in memoria locale.
In particolare, il numero dei parcheggi occupati $N_{\mathrm{i}}^{\mathrm{o}}$ è dato dalla differenza tra il numero di eventi parcheggio $E_{\mathrm{i}}^{\mathrm{p}}$ ed il numero di eventi rilascio $E_{\mathrm{i}}^{\mathrm{r}}$.

$$N_{\mathrm{i}}^{\mathrm{o}} = E_{\mathrm{i}}^{\mathrm{p}} - E_{\mathrm{i}}^{\mathrm{r}}$$

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

$$p_{\mathrm{i}}^{\mathrm{o}} = \frac{N_{\mathrm{i}}^{\mathrm{o}}}{N_{\mathrm{i}}^{\mathrm{t}}}$$

Infine calcoliamo la probabilità di trovare parcheggio nella cella i $p_{\mathrm{i}}^{\mathrm{f}}$, serven-

dosi del tasso di occupazione appena calcolato:

$$p_{\mathrm{i}}^{\mathrm{f}} = 1 - p_{\mathrm{i}}^{\mathrm{o}}$$

La probabilità $p_{\mathrm{i}}^{\mathrm{f}}$, messa in percentuale, sarà l'informazione a cui l'utente avrà accesso una volta aperta l'applicazione. 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 3.2, 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 centralizzata. 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 di spreading nel Sistema Operativo Android, con l'ausilio della tecnologia *WiFi Direct*. La tecnologia, 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 in sezione 2.4.

3.1 Componente di disseminazione

Il paragrafo illustra tutti gli attori principali che compongono il servizio di disseminazione delle informazioni. Ogni sottoparagrafo spiega 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.

3.1.1 Strategia di connessione tramite WiFi Direct

Il *WiFi Direct* è una nuova tecnologia definita dalla *WiFi Alliance* che permette comunicazioni wireless Device-To-Device[30]. La tecnologia si basa sullo standard 802.11, ereditandone caratteristiche e sviluppi degli ultimi anni. L'architettura classica WiFi prevede una netta distinzione tra l'Access Point e i client, mentre il *WiFi Direct* introduce una certa dinamicità in questo senso. Ogni device che supporta la tecnologia può fungere sia da Access Point che da semplice client. I cosiddetti *P2P Group* possono essere comparati ad infrastrutture WiFi e sono creati al termine di una fase di *Group Owner (GO) negotiation*, dove viene stabilito il ruolo (Access Point o client) per ogni device coinvolto[23]. 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). I client vengono identificati all'interno del gruppo tramite un IP, assegnato dal *DHCP* del dispositivo Access Point. La creazione del gruppo può avvenire in 3 modalità distinte:

- *standard*: dopo una fase di discovery, i dispositivi iniziano la fase di *GO negotiation* per definire i ruoli all'interno del gruppo. Una volta definito l'Access Point, il *DHCP* identifica i client tramite l'assegnazione degli indirizzi IP.

- *autonoma*: un dispositivo si autoelegge spontaneamente a *P2P Group Owner* ed inizia a notificare la propria presenza tramite beacon.

- *persistente*: durante la creazione del gruppo vi è la possibilità di tener traccia delle credenziali e dei ruoli. Questo dovrebbe favorire il processo di ricreazione del gruppo.

Per realizzare lo scopo del progetto, la tecnologia *WiFi Direct* non è stata utilizzata in maniera tradizionale, bensì in modalità legacy.
Questo perchè, almeno nel primo incontro tra due device, la nascita della connessione necessità dell'interazione utente (come mostrato in figura 3.1).
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 in modalità autonoma 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 in sezione 2.3.

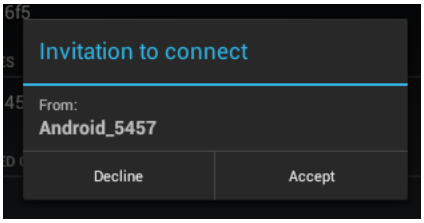

Figura 3.1: Interazione utente per il consenso alla connessione WiFi Direct

3.1.2 Il servizio background

Dato che il processo di comunicazione tra peer deve avvenire in maniera completamente trasparente all'utente, 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 Andorid, sprovvisto di interfaccia grafica, che si preoccupa di compiere procedure in background. Questo componente è stato denominato *NetworkAutoService*, ed estende appunto la classe *Service* di Android. I metodi che gestiscono il servizio sono:

- *onCreate*: crea il servizio e lo notifica all'utente.
- *onStartCommand*: metodo che inizia l'esecuzione vera e propria del servizio di disseminazione, in background. Qui, sostanzialmente, espone in beaconing la possibilità di fungere da Access Point e si mette alla ricerca di preesistenti Hotspot nei paraggi. Queste funzionalità verranno descritte meglio in sezione 3.1.5 e 3.1.6. Si ricorda che l'attivazione

del servizio è sancita dall'utente tramite la pressione di un pulstante da interfaccia grafica, come mostrano gli screenshot in figura 3.3.

- *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 stop del servizio, come mostrano gli screenshot in figura 3.3.

- *startCheckNetworkTimer*: metodo privato che inizializza un oggetto *Timer* e lo schedula ogni 3s. I 3s possono essere interpretati logicamente come un time slot. L'azione schedulata, prevede l'invio di un *Intent*, in modo tale che il *BroadcastReceiver* (definito nella sezione 3.1.8) lo possa catturare e controllare lo stato della Network State Machine (definita in sezione 2.4.2).

3.1.3 Il NetworkController

La classe *NetworkController* si preoccupa di gestire gli attori coinvolti nel processo di networking e disseminazione locale delle informazioni. Questo oggetto è stato implementato seguendo le linee guida del pattern Singleton. Questo pattern permette di creare una ed una sola istanza di una particolare classe, rendendola disponibile in ottica globale. Il concetto è quello di avere controllo, in ogni punto del progetto, sulle componenti fondamentali dell'architettura implementata. Questo componente dà accesso agli oggetti delle classi *NetworkBeaconPeerSearcher*, *NetworkAccessPoint* e *NetworkConnection*. Il primo 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 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:

- *startNetworkAccessPoint* / *stopNetworkAccessPoint*: metodi che decretano l'inizio, o la fine, del device come Access Point.
- *startNetworkConnection* / *stopNetworkConnection*: metodi che cercano di connettersi, o scollegarsi, da un Hotspot.
- *startNetworkBeaconPeerSearcher* / *stopNetworkBeaconPeerSearcher*: metodi che sanciscono l'inizio, o la fine, della ricerca di beacon nelle vicinanze.

3.1.4 Connessione ad un Access Point

La classe *NetworkConnection* detiene la procedura necessaria per connettersi in maniera automatica ad un Access Point (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. Assumendo il ruolo di client si cercherà di agganciarsi all'Hotspot individuato.
È necessario ricordare che, per effettuare queste procedure in maniera completamente automatica, vi è la necessità di conoscere la passphrase per connettersi. Questa come vedremo in sezione 3.1.8, sarà ricavata dai beacon di servizio. Di seguito viene mostrato il codice discusso:

```
/* connection to Access Point */
public class NetworkConnection {
  ...
  public NetworkConnection(context, ssid, passphrase) {
    ...
    /* disable other networks */
    ...
    wifiConfig = new WifiConfiguration();
    wifiConfig.SSID = String.format("\"%s\"", ssid);
    /* set passphrase */
    wifiConfig.preSharedKey = String.format("\"%s\"", passphrase);

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

3.1.5 Esplorazione dell'ambiente circostante

Il sistema ha la necessità di capire la situazione nell'ambiente esterno, scoprendo quanti e quali attori vi sono nelle vicinanze e comportarsi di conseguenza. Tutto quello che verrà illustrato in questa sezione è stato implementato nella classe *NetworkBeaconPeerSearcher*.

Scoperta di peer

La classe *WiFiP2PManager* di Android, permette di gestire tramite chiamate API la connessione peer-to-peer tra device. Molte di queste fungono da listener per gli eventi asincroni della rete, in modo tale da proporre determinate logiche al succedersi di determinati eventi. Tramite il metodo *discoverPeers* è possibile mettersi alla ricerca di eventuali device Android nell'ambiente circostante. Per catturare le notifiche relative alla presenza di un device è stato necessario implementare un custom *PeerListListener*, il quale grazie al metodo *onPeersAvailable* restituisce la lista dei peer 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[30]. La documentazione per developer di Android[21] specifica che questo meccanismo è stato fatto per individuare peer con determinati servizi. I Bonjour service vengono creati grazie alla classe *WifiP2pDnsSdServiceInfo*. 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 includere 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 può essere inserito in questi beacon. Lo standard consiglia di tenere questo contenuto sotto i 200 bytes e non è raccomandato superare i 1300 bytes[22]. Provando empiricamente, si è potuto confermare la restrinzione 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 Access Point e connessioni automatiche. Questi pacchetti risultano comunque determinanti, in quanto incapsulano la stringa passphrase, necessaria ai client per connettersi in maniera automatica agli Access Point. 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 beacon appena scoperto (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 peer 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 (indicante la presenza di un Access Point) 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 molto ampia, per questo si mostra il solo codice relativo ai listener citati.

```
...
/* peer discover listener */
peerListListener = new WifiP2pManager.PeerListListener() {
  public void onPeersAvailable(peers) {
    for (peer : peers) {
      nPeers++; /* peer found! */
    }
    /* if there is someone, looking for D2DSP service */
    if(nPeers > 0){
      /* change NSM state */
      networkStateMachine = STATE_ACCESS_POINT_PEERS;
      startBeaconDiscovery();
%      } else { /* else search peers */
      /* change NSM state */
      networkStateMachine = STATE_ACCESS_POINT_NO_PEERS;
      startPeerDiscovery();
    }
  }
};
...
/* service discover listener */
serviceListener = new WifiP2pManager.DnsSdServiceResponseListener()
    {
  public void onDnsSdServiceAvailable(beacon, type, device) {
    if (serviceType.startsWith("D2DSP")) {
      ...
      /* send intent with beacon payload */
      context.sendBroadcast(intent);
    } else {
      /* no service D2DSP */
    }
    /* continue to search peers */
    startPeerDiscovery();
  }
};
...
```

3.1.6 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 gruppo. In particolare, l'interfaccia *WifiP2pManager.GroupInfoListener* permette di essere notificati quando il gruppo è stato creato. Questo è permesso dal metodo *onGroupInfoAvaila-*

ble, 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. Oltre a questa operazione viene fatto partire il meccanismo per ricevere connessioni in input tramite Socket. Come citato in precedenza (sezione 3.1.5), 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 WiFi Direct Group */
public void start() {
  managerP2P.createGroup(channel,new
      WifiP2pManager.ActionListener() {
    public void onSuccess() {
      /* Access Point created! */
    }
  });
}
...
@Override
public void onGroupInfoAvailable(WifiP2pGroup group) {
  try {
    ...
    /* put passphrase in Bonjour beacon */
    startBeaconAP("D2DSP:" + group.getNetworkName() + ":" +
        group.getPassphrase() + ":" + netAddress);
    /* Start Socket Server for synchronizations with clients */
    new Server();
  } catch(Exception e) {
    /* failure */
  }
}
...
/* expose beacon that notifies clients the Access Point presence */
private void startBeaconAP(String instance) {
  beacon = WifiP2pDnsSdServiceInfo.newInstance(instance, type,
      record);
  /* broadcast of WifiP2pDnsSdServiceInfo beacon */
  managerP2P.addLocalService(channel, beacon, new
      WifiP2pManager.ActionListener() {
    public void onSuccess() {
      Log.d(TAG, "Beacon is in air!");
    }
  });
}
...
```

3.1.7 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. I canali di comunicazione si basano su *Socket* serializzate, permettendo di inviare e ricevere, con un semplice casting, direttamente degli *ArrayList<ParkEvent>*. L'Access Point, appena notificato dell'avvenuta creazione del gruppo, richiama la creazione di un oggetto della classe *Server*, il quale istanzia una *ServerSocket*, mettendosi in ascolto di eventuali richieste, con il metodo *accept()*. I client invece, quando notificati dal *BroadcastReceiver* dell'avvenuta connessione all'Hotspot, creano un oggetto *Socket* con IP e porta idonea. Ora che il canale di comunicazione bidirezionale è stato messo in piedi, l'Access Point invierà il proprio *ArrayList* nell'*ObjectOutputStream*, 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 entry ricevute 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. 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, PORT);
      while (true) {
        ...
        try {
          in = new ObjectInputStream(socket.getInputStream());
          /* get data from Socket channel */
          toSync = (ArrayList<ParkEvent>) in.readObject();
          ...
          break;
        } catch (Exception e) {
        }
      }
      ...
```

```
      while(run) {
       try {
        out = new ObjectOutputStream(socket.getOutputStream());
        /* write data in Socket channel */
        out.writeObject(myParkEvents);
        break;
       } catch (IOException e) {
        run = false;
       }
      }
      socket.close();
      /* contact Controller - merge new entries */
      mergeEntries(toSync);
      /* contact Controller - register sync */
      insertSynchronizationOnSQLiteDB(false, macWithSync);
      /* change state in NSM */
      networkStateMachine = 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(PORT);
       while (true) {
         Socket socket = serverSocket.accept();
         /* thread for socket sync with client */
         socketServerReplyThread.run();
       }
     } catch (IOException e) {
     }
   }
}
...
```

3.1.8 L'ascoltatore BroadcastReceiverManagement

Un componente fondamentale, a livello implementativo, è senza dubbio il *BroadcastReceiverManagement*. Estendendo la classe Android *BroacastReceiver* si 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. I *BroacastReceiver* possono tranquillamente lavorare in background e ricevere aggiornamenti anche se l'applicazio-

ne è chiusa. 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 Access Point attivo nelle vicinanze e cerca di connettersi ad esso, utilizzando la passphrase ricavata dal beacon di Bonjour.

```
if(action.equals(INTENT_D2D_AP_FOUND)) {
  /* get beacon data */
  String beacon =
      intent.getStringExtra(INTENT_D2D_AP_ACCESSDATA);
  String[] payload = beacon.split(":");
  if(networkConnection == null) {
    /* stop my Access Point functionalities */
    NetworkController.getInstance().stopAccessPoint();
    /* Now I'm a simple Client */
    ssid = payload[1];
    passphrase = payload[2];  /* passprhase from beacon */
    /* start connection mechanism */
    networkConnection = new
        NetworkConnection(ctx,ssid,passphrase);
    /* change state of NSM */
    networkStateMachine = 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(action.equals(WifiManager.NETWORK_STATE_CHANGED_ACTION)) {
  ... check connection state ...
  /* looking for connection @ Access point */
```

```
    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 ha la possibilità di effettuare un successivo scambio dati, dopo una certa threshold. Il componente quindi controlla periodicamente 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(action.equals(INTENT_D2D_CHECK_MACHINE_STATE)) {
    ...
    /* check for threshold */
    boolean check = Controller.getInstance().isPossibleSync();
    if((networkState.equals(STATE_SYNCED_CONNECTED))&&(check)) {
        /* re-synchronized client side */
        this.runThreadSync(context);
    }
    ...
}
```

3.2 Gestione dell'informazione

Ogni device Android deve essere in grado di registrare tutte le informazioni sincronizzate nella propria memoria locale. Le informazioni vengono organizzate in un database di tipo relazionale basandosi sullo standard SQL e mappate in oggetti Java, in modo da semplificarne la gestione.

3.2.1 Il Modello

Il modello si compone di due entità, definite come *Synchronization* e *ParkEvent*. Gli attributi che compongono queste due classi sono i medesimi

definiti in fase di progettazione (sezione 2.4.1) ed i metodi sono i tradizionali getter e setter, i quali permettono di leggere informazioni e settare attributi delle istanze Java in questione. All'interno dell'architettura, queste istanze, vengono gestite e rese accessibili dal *Controller* tramite degli *ArrayList<*>*.

3.2.2 Il database SQLite

Salvare le informazioni dei parcheggi in maniera strutturata è sembrato la soluzione migliore fin dalla progettazione. Nei sistemi mobili Android è possibile lavorare con basi di dati tramite l'utilizzo di *SQLite*[24]. Si tratta di una serie di API, presenti nel package *android.database.sqlite*, che permettono di gestire un file come un database transazionale a tutti gli effetti. All'avvio dell'applicazione viene istanziato un oggetto *CustomSQLiteHelper* (estensione della classe *SQLiteOpenHelper*) che controlla se esiste un database SQLite e ne apre la connessione. Se è la prima volta che l'utente avvia l'applicazione, dopo l'installazione, viene automaticamente creato il database con le tabelle *synchronizations* e *park_events*. La classe che effettua le query sql e che viene contatta dal *Controller*, per recuperare informazione dal database, è *SQLiteDBManager*. I metodi che fornisce questa classe sono i necessari per operazioni CRUD (Create Read Update Delete) su una base di dati. La chiamata *rawQuery* dell'oggetto *SQLiteDatabase* permette di effettuare del codice sql puro, in modo da facilitare le query più particolari. Di seguito viene mostrato il codice di alcune funzionalità.

```
/* INSERT a new park event in SQLite DB */
public boolean insert(cellId, event, timestamp, mac) {
  values.put(COLUMN_PARK_CELL_ID, cellId);
  values.put(COLUMN_PARK_EVENT, event);
  ...
  long idRow = this.db.insert(TABLE_NAME, null, values);
  /* if entity already exists -> return FALSE */
  if(idRow < 0) return false;
  return true;
}
...
/* READ all Synchronizations from DB, return the ArrayList */
public ArrayList<Synchronization> getSync(apRole, timeLimit) {
  /* filter query on time and role */
  c = this.db.rawQuery(
    "SELECT * " +
```

```
    "FROM " + TABLE_NAME + " " +
    "WHERE " + COLUMN_SYNC_TIMESTAMP + " >= '" + timeLimit + "' " +
    "AND WHERE " + COLUMN_SYNC_ROLE + " = " + apRole + " " +
    "ORDER BY " + COLUMN_SYNC_TIMESTAMP + " DESC;", null
  );
  //return results in ArrayList<Synchronization>
}
```

3.2.3 Il Controller

L'oggetto *Controller* si avvicina concettualmente al *NetworkController*, illustrato in 3.1.3. Implementato anch'esso con il pattern Singleton, ha l'obiettivo di fornire agli altri componenti dell' architettura le informazioni presenti nel database *SQLite*. Quando si deve effettuare una query sulla base di dati, non si crea un oggetto *SQLiteDBManager* ma si contatta direttamente il *Controller*, il quale fornirà la risposta. Oltre a fungere da interfaccia verso il database, implementa altri metodi utili:

- *isMyServiceRunning(Class<?>class, Context ctx)*: metodo che controlla se un determinato servizio background è attivo.
- *mergeEntries(ArrayList<ParkEvent>pe)*: metodo che unisce le nuove entry in input con quelle preesistenti nella memoria locale.
- *getMacAddr()*: metodo che ritorna l'indirizzo MAC del device.
- *isPossibleSync()*: metodo che sancisce se è possibile effettuare una sincronizzazione. I client, prima di effettuare una sincronizzazione con l'Access Point, controllano che sia passata una certa threshold dall'ultimo scambio dati.

```
/* Return if is possible a synchronization */
public boolean isPossibleSync() {
  /* get last sync datetime */
  date = dbManager.selectLastSyncTimestamp();
  /* get difference from now and last sync in seconds */
  long diff = (now.getTime() - date.getTime())/1000;
  /* check threshold in seconds */
  if(diff < TIME_FOR_NEXT_SYNC)
    return false;
  return true;
}
```

3.2.4 L'informazione utente

La classe *ProbabilityPark* tramite il metodo *getProbabilityToParkByCell* implementa i calcoli relativi alla probabilità di trovare parcheggio in una determinata cella della città. Sostanzialmente si tratta della traduzione in codice sorgente di quanto detto in sezione 2.5.

3.3 Interfaccia grafica

L'interfaccia grafica è gestita da un'estensione della classe Android *AppCompactActivity*. L'utente tramite il movimento di swipe laterale ha la possibilità di spostarsi da una schermata all'altra. La *MainActivity* utilizza un *FragmentPagerAdapter* per catturare l'evento di swipe e mostrare il *Fragment* corretto. Ogni schermata viene gestita da un'estensione della classe *Fragment*, fornendo all'utente una determinata interfaccia grafica in base ai file *xml* di layout che vengono settati nel metodo *onCreateView*. L'applicazione è composta da 5 schermate:

- *MapFragment*: schermata che mostra la città divisa in celle, colorate in base alla probabilità di occupazione. La pressione di una cella fornisce un *Toast* indicante la probabilità in numero. La figura 3.2 mostra lo screenshot della schermata.

- *DisseminationFragment*: schermata in cui si può attivare/disattivare il servizio di disseminazione tramite *WiFi Direct*. Inoltre si forniscono all'utente informazioni sullo stato in cui si trova. Le figure 3.3 e 3.4 mostrano lo screenshot della schermata.

- *RecognitionFragment*: schermata in cui si può attivare/disattivare le fasi di training e recognition della mobilità. Se la fase di recognition è attiva viene mostrata la predizione corrente. La figura 3.5 mostra lo screenshot della schermata.

- *SyncListFragment*: schermata in cui viene mostrato all'utente la lista delle ultime sincronizzazione. È possibile filtrare per tempo e ruolo (se il device fungeva da Access Point o no al momento della sincronizzazione). La figura 3.6 mostra lo screenshot della schermata.

- *ParkEventListFragment*: schermata in cui viene mostrato all'utente la lista degli ultimi eventi di parcheggio. È possibile filtrare per cella e tipo (parcheggio o rilascio). La figura 3.6 mostra lo screenshot della schermata.

3.4 Screenshot

Di seguito sono mostrati gli screenshot dell'applicazione *D2DSmartParking*:

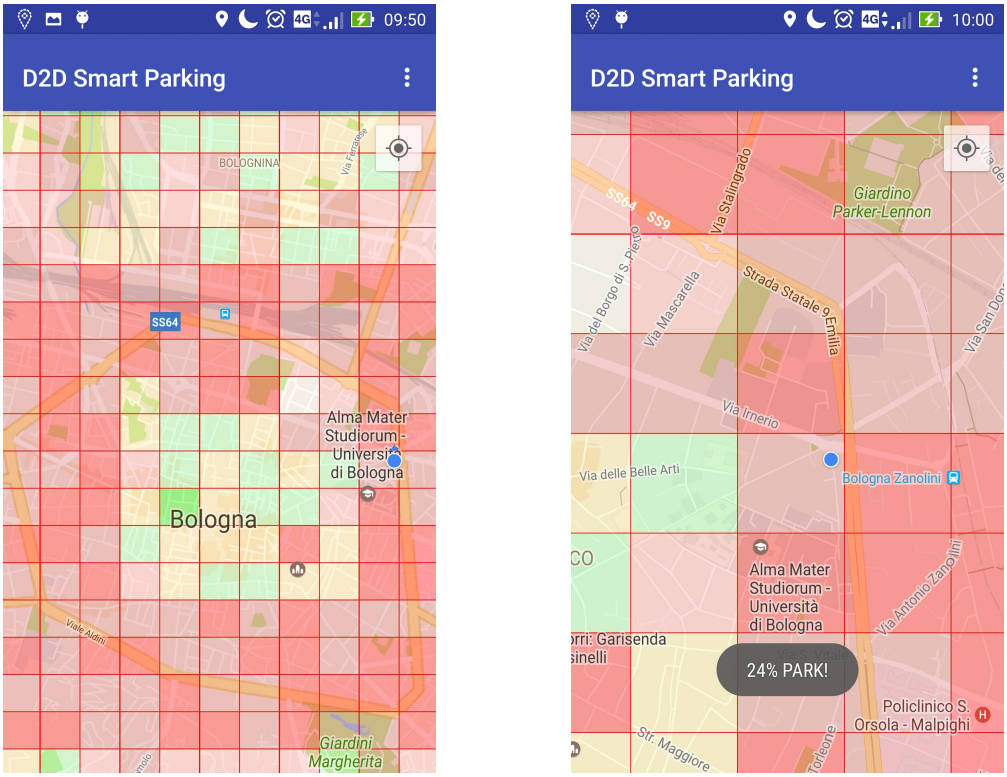

Figura 3.2: Mappa del centro di Bologna divisa in celle, ognuna colorata con la relativa probabilità di trovare parcheggio. Le probabilità sono state stimate grazie a [29], dove viene mostrato uno studio sulle strade più congestionate di Bologna

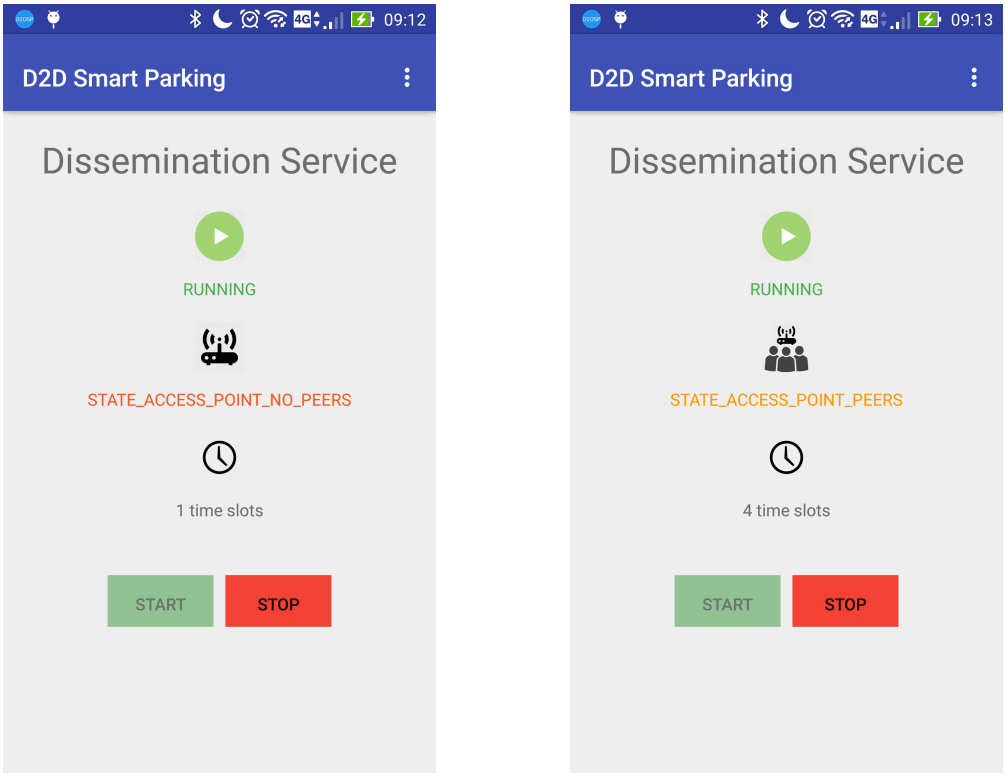

Figura 3.3: Servizio di disseminazione attivo negli stati STATE_ACCESS_POINT_NO_PEERS e STATE_ACCESS_POINT_PEERS

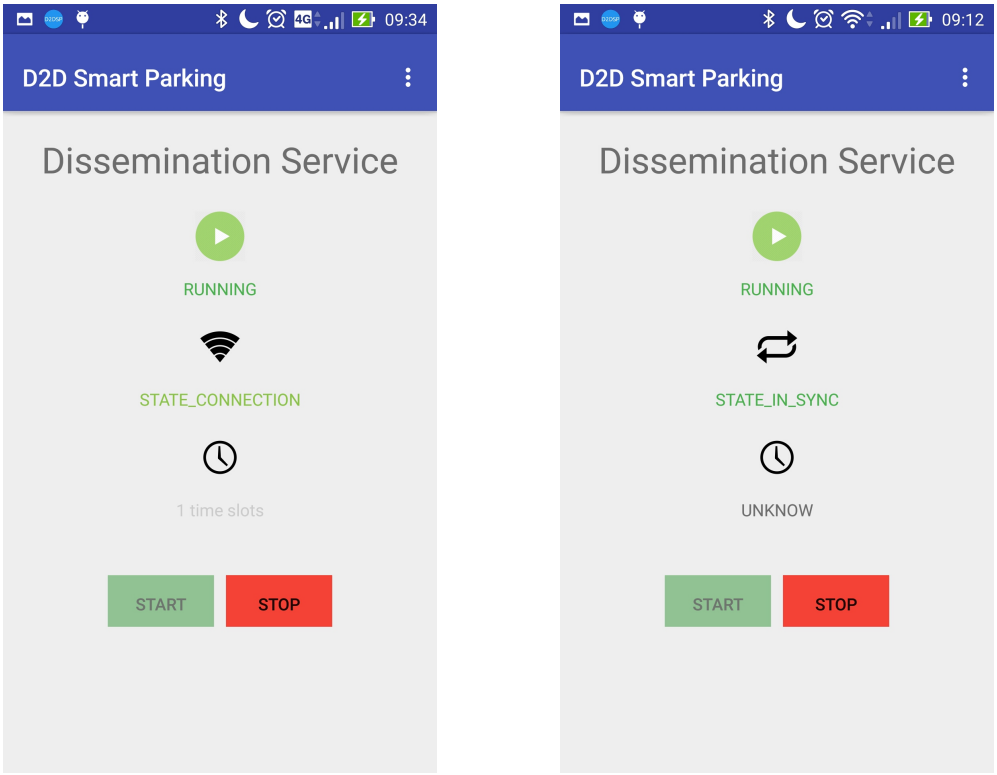

Figura 3.4: Servizio di disseminazione attivo negli stati STATE_CONNECTION e STATE_IN_SYNC

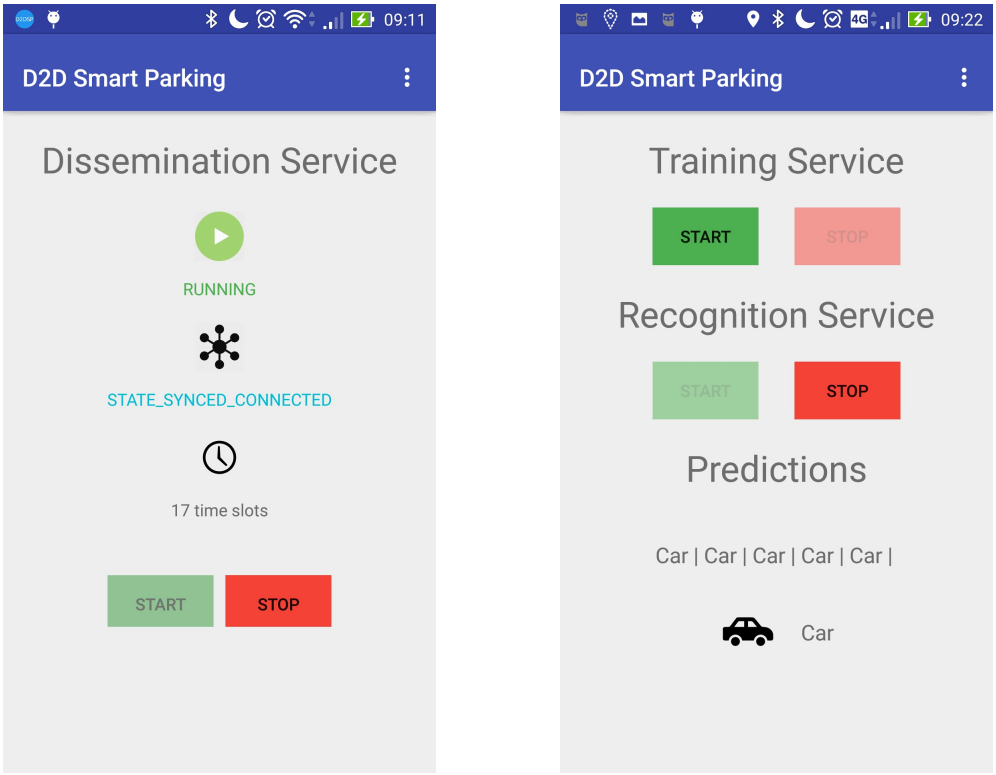

Figura 3.5: Servizio di disseminazione attivo negllo stato STATE_SYNCED_CONNECTED e schermata del componente di Activity Recognition

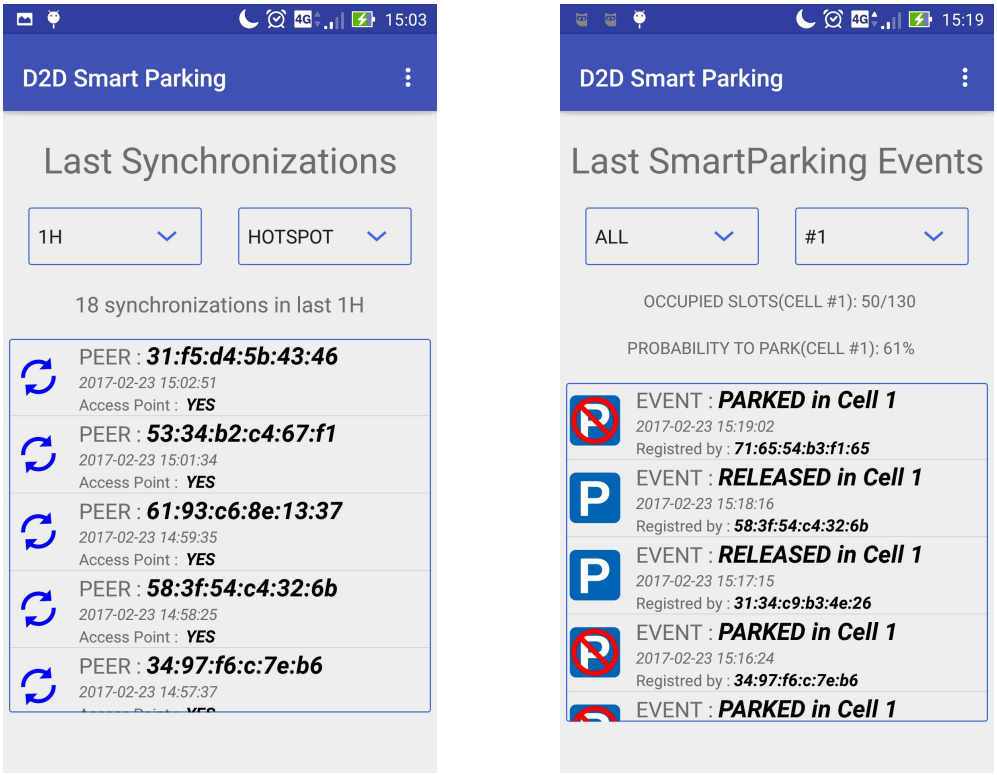

Figura 3.6: Schermate che mostrano all'utente le ultime sincronizzazioni effettuate e gli eventi parcheggio, con possibilità di filtro

Parte III

Valutazione

Capitolo 4

Valutazione

L'ultima fase riguarda la valutazione del processo di disseminazione delle informazioni, grazie alla quale riusciamo a studiare come si propagano i dati all'interno del mondo che si va a considerare.
È importante comprendere che non si andrà a valutare la precisione delle probabilità di parcheggi liberi, bensì si andrà a verificare la convergenza sulla conoscenza dello scenario. Infatti è fondamentale che l'utente riesca a sincronizzarsi efficacemente con gli altri attori circostanti, in maniera tale da restare aggiornato, durante il movimento, sulla cella corrente e su quelle adiacenti.

4.1 Strumenti

Ovviamente non è pensabile testare lo spreading delle informazioni in un ambiente reale, di conseguenza ci si affida ad un simulatore ad eventi discreti. In ambito accademico lo strumento più efficace ed utilizzato per fare simulazione è senza dubbio OMNeT++[26]. Il software, disponibile in open source dal 2003, viene utilizzato in fase di progettazione, ricerca, testing e analisi di protocolli di rete, architetture software e hardware. L'ambiente di sviluppo è un surrogato di Eclipse ed il linguaggio principale è il C++.
La logica del simulatore si basa sul concetto fondamentale di Modulo, i quali

possono essere aggregati e customizzati a seconda delle proprie esigenze. In particolare, per la valutazione del progetto e dell'architettura pensata è stato necessario sviluppare un componente *SmartParking* che riproponesse la logica pensata in fase di progettazione ed implementata conseguentemente. Come verrà illustrato in sezione 4.2, sono stati parametrizzati gli aspetti che influenzano maggiormente lo spreading dell'informazione, in modo tale da comprendere le differenze al variare delle condizioni.
L'utilizzo di OMNeT viene integrato dall'apporto di SUMO e di Veins[27]. Il primo è un simulatore di mobilità urbana, anch'esso ottenibile in open source, che si preoccupa di emulare reti stradali, anche di grandi dimensioni. Tutte le configurazioni necessarie per il corretto funzionamento di questo, definite in file xml, riguardano la definizione delle strade, delle costruzioni presenti nell'ambiente e delle macchine (di cui, per ognuna, viene specificato il tempo di partenza ed il percorso da effettuare). La gestione di questi file risulta complicata quando si vanno a simulare intere città, in quanto la mole di dati diventa importante. Veins è un framework open source, scritto anch'esso in C++, che fornisce una grossa quantità di moduli preimplementati, per la simulazione di protocolli wireless in ambito veicolare. Inoltre vengono fornite API per interfacciarsi a TraCi, un server di SUMO che permette l'interazione tra i diversi simulatori.
I risultati delle simulazioni possono essere visualizzati direttamente da un apposita interfaccia grafica fornita da OMNeT ma possono anche essere esportati in vari formati. Si è scelto di esportare i risultati in formato CSV, per poi analizzarli tramite degli script PHP.

4.2 Modellazione

La città presa in considerazione per l'analisi simulata è Bologna. In realtà, viene presa in considerazione soltanto una porzione di questa, più precisamente la zona nord-est del centro storico fino ad arrivare in zona Bologna fiere, fuori dall'anello del centro. Quantificandola, l'area appare come un ret-

tangolo di 2.5 km per 1.5 km, per un totale di 3.75 km^2. Comprendendo le strutture universitarie e la stazione dei treni, si tratta di una zona abbastanza movimentata e del tutto idonea al testing dell'architettura implementata.

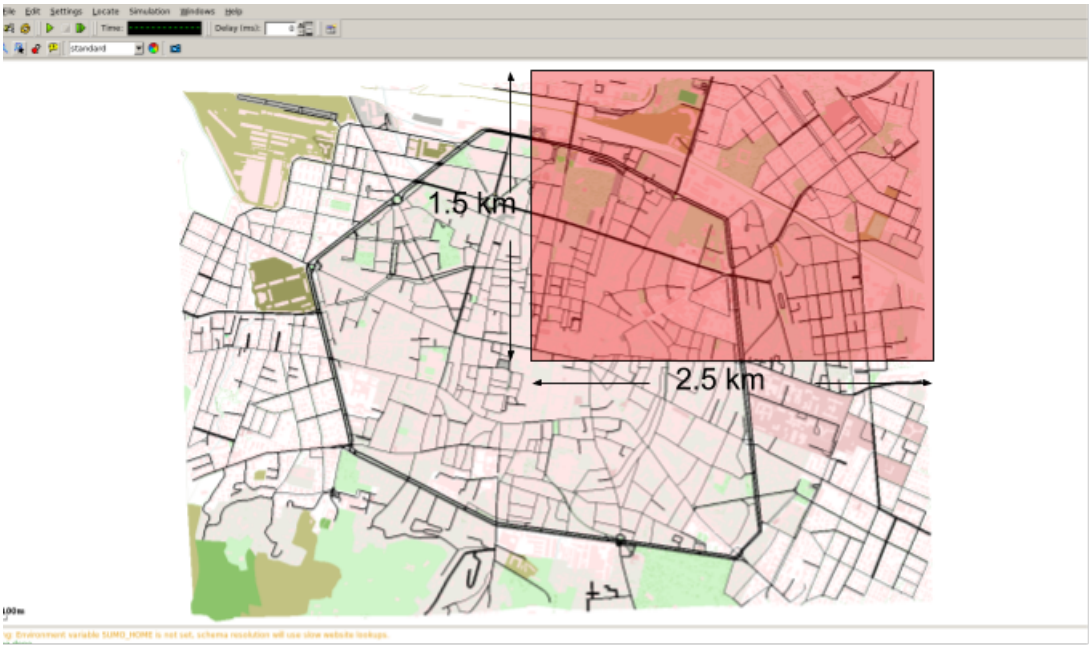

Figura 4.1: Interfaccia grafica di SUMO che mostra la porzione di Bologna per la simulazione

La struttura implementata permette di parametrizzare determinate metriche in fase di configurazione, le quali vengono citate in elenco:

- *range*: quando i due dispositivi risiedono l'uno nel range trasmissivo dell'altro la comunicazione avrà esito positivo. Misura espressa in metri.
- *min_latency*: latenza minima che un utente aspetta per effettuare una sincronizzazione. Misura espressa in secondi.
- *max_latency*: latenza massima che un utente può verificare prima che avvenga una sincronizzazione o un fallimento. Misura espressa in secondi.

- *sensor_accuracy*: accuratezza dei device Android nel performare l'evento di activity recognition rilascio o parcheggio. Misura espressa in probabilità da 0 a 1, compresi.

- *synchronization_time*: tempo minimo per cui un utente deve aspettare prima di poter ri-sincronizzarsi. Misura espressa in secondi.

- *cell_size*: lunghezza del lato di una cella, espresso in metri.

La logica progettata per la simulazione è una versione semplificata di quella implementata nell'applicazione Android ma rappresenta comunque fedelmente il modello, in modo che i risultati ottenuti possano essere ritenuti affidabili. I dispositivi si possono sincronizzare continuamente nel tempo ma, per problemi di overhead già citati, tra due scambi di dati deve passare una certa threshold, rappresentata nel modello dal parametro *synchronization_time*.
I parametri di *min_latency* e *max_latency* rappresentano il tempo che gli Access Point e i client ci mettono per identificare il loro ruolo, scoprire peer, rilevare beacon e connettersi ad un *WiFiDirect Group*.

4.3 Metriche di prestazione

In fase di simulazione, si vuole dimostrare che il movimento degli utenti e la strategia di disseminazione locale, permettono di restare sincronizzati sugli eventi che effettivamente accadono. In particolare si cercherà di analizzare l'accuratezza dei dati presenti nei database locali ai device, rispetto a ciò che è accaduto realmente.
Per definire meglio il concetto dell'accuratezza, si prenda di riferimento una cella i dove, nel lasso di tempo t, sono stati performati 100 eventi parcheggio o rilascio E_i^{tot}. A questo punto, all'istante *t+1*, se il device ha nella propria conoscenza locale (E_i^{loc}) 10 eventi dei 100 totali, avrà un'accuratezza del 10%, se ne ha 50 avrà accuratezza del 50% e così fino al limite superiore del 100%,

il quale può essere raggiunto solo da una struttura di tipo centralizzata.

$$accuracy_{\mathrm{i}}^{\mathrm{t+1}} = \frac{E_{\mathrm{i}}^{\mathrm{loc}}}{E_{\mathrm{i}}^{\mathrm{tot}}} * 100$$

L'accuratezza, appena descritta, verrà calcolata e registrata nel corso del tempo per comprendere il relativo andamento, col trascorrere dei secondi. Inoltre verrà stabilita l'accuratezza in relazione alla distanza, per capire il grado di allineamento non solo sulle celle strettamente adiacenti, ma anche su quelle progressivamente più lontane. Infine verrà presa in considerazione l'accuratezza media dopo 1800 secondi in base al numero di veicoli presenti nella porzione di Bologna considerata.
Tutte queste misurazioni verranno comparate simulando diverse tecnologie, in modo tale da capire dove si colloca l'implementazione tramite *WiFi Direct*.

4.4 Configurazioni e run

Si vogliono comparare le metriche sopradescritte, simulando l'architettura con l'utilizzo di tre tecnologie differenti quali *VANET V2V*, *WiFi Direct* e *Bluetooth*. Il tempo per ogni run di simulazione è stato settato a 1800 simsec (mezzora simulata) e per ogni tecnologia si considera lo scenario target con 3000 veicoli. Il numero di veicoli fa riferimento a rilevamenti reali della porzione di Bologna considerata, in orari centrali della giornata. Per ottenere una stima dell'accuratezza media in base al numero di veicoli è stato necessario effettuare simulazioni riducendo le autovetture anche a 1500 e 500. L'accuratezza dei sensori è stata settata per tutte le simulazioni a 90%. La scelta deriva dallo studio in [2], dove viene stimata l'accuratezza del componente di activity recognition presente nell'architettura.
Per ottenere un'analisi statistica che abbia un senso, si effettuano 10 run indipendenti per ogni tecnologia e numero di veicoli, per un totale di 90 run indipendenti. I risultati saranno frutto delle medie dei vari run con la stessa configurazione.

Simulation parameters			
technology	V2V	WiFi Direct	Bluetooth
range(m)	500	100	20
min_latency(s)	0	2	5
max_latency(s)	0	10	15
sensor_accuracy(%)	0.90	0.90	0.90
synchronization_time(s)	20	20	20
cell_size(m)	250	250	250
sim time(simsec)	1800	1800	1800

Tabella 4.1: Parametri di simulazione

VANET V2V

Per simulare la tecnologia presente nei sistemi V2V, basato sull'802.11p, si setta il *range* di comunicazione sui 500m. Il range di comunicazione tra veicoli, in condizioni ottimali di propagazione, può raggiungere i 1000m[28], ma la distanza settata appare più idonea per testare la tecnologia. I parametri di *min_latency* e *max_latency* sono azzerati in quanto la logica descritta in fase di progettazione ed implementata sull'app Android, su questi sistemi, non ha senso di esistere. I canali di comunicazione V2V non necessitano di un meccanismo di esposizione dell'Hotspot e connessione a questo da parte dei client.

WiFi Direct

La seguente configurazione rispecchia l'architettura progettata ed implementata sull'applicazione Android. La documentazione ufficiale dello standard *WiFi Direct* specifica che il range di comunicazione potrebbe raggiungere i 200m[20]. In realtà 100m sembra un'assunzione più realistica per garantire la maggior parte delle connessioni. La latenza di apertura dei canali di comunicazione viene settata dai 2 (*min_latency*) ai 10 secondi (*max_latency*).

Si ricorda che i parametri di *min_latency* e *max_latency* modellano la fase di discovery e connessione all'Access Point, come già accennato in sezione 4.2.

Bluetooth

La terza tecnologia che si vuole considerare è il *Bluetooth*, il quale ha un range di comunicazione sui 20m ed una latenza di connessione che va dai 5 ai 15 secondi. Quest'ultimi parametri possono essere visti come un approccio ottimistico, in quanto il protocollo Bluetooth può impiegare più tempo per scoprire peer nelle vicinanze.

4.5 Risultati

L'accuratezza della cella corrente

Il primo grafico (figura 4.2) misura l'accuratezza delle informazioni sulla cella corrente nel tempo. Ad intervalli di tempo costante e per ogni veicolo, viene calcolata la percentuale di informazione disponibile per la cella corrente, rispetto al numero di eventi totali che sono stati compiuti in quel quadrante. L'andamento delle curve nel grafico 4.2 mostra come la tecnolo-

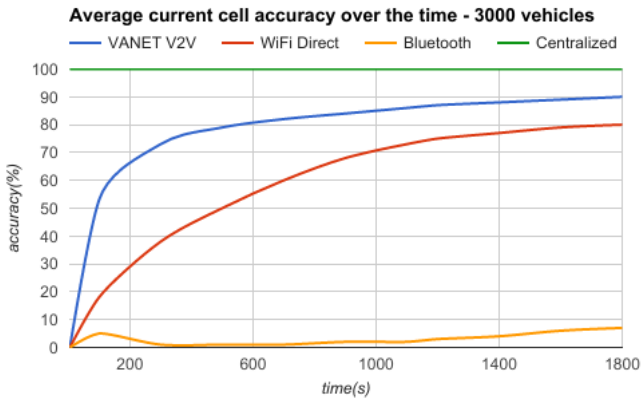

Figura 4.2: Grafico dell'accuratezza media della cella corrente al trascorrere del tempo - 3000 veicoli

gia *V2V*, dopo una prima fase di apprendimento di circa 400 simsec, riesca

a mantenere un livello di accuratezza sull'80-85%, raggiungendo il 90% dopo la mezzora di simulazione. Si ricorda che i 3000 veicoli non vengono creati tutti nell'istante 0, ma ciascuno viene inserito nello scenario in un istante che va da 0 a 1800 simsec. Nella fase iniziale, detta anche transiente, il numero di veicoli è basso e per questo l'informazione non riesce a raggiungere i nodi già presenti nella città. Man mano che altri veicoli vengono inseriti, l'accuratezza sale grazie all'aumentare delle sincronizzazioni possibili, rendendo il processo di spreading più efficace. La tecnologia *WiFi Direct* cresce meno rapidamente del *V2V* a causa del minor range di comunicazione e della latenza di connessione. Si ottiene comunque un buon livello di accuratezza già dopo 20 minuti, mantenendosi sul 75% e raggiungendo anche l'80% a fine simulazione. La tencologia *Bluetooth* non riesce a raggiungere livelli di accuratezza soddisfacenti. La latenza di connessione e soprattutto il basso range di comunicazione incidono negativamente sulla disseminazione. Inoltre, come si può notare dall'andamento delle curve, l'indice di accuratezza non è ancora completamente stazionario dopo i 1800 simsec di simulazione. Il livello di accuratezza continua leggermente a salire negli ultimi istanti. Non si esclude che si potrebbero ottenere risultati sempre migliori, con il trascorrere del tempo.

L'accuratezza dello scenario generale

Il secondo grafico (figura 4.3) misura l'accuratezza delle informazioni rispetto a tutta la città e non solo rispetto alla cella corrente in cui si trova il veicolo. L'andamento delle curve nel grafico 4.3 può essere comparabile con quello nel grafico 4.2 con un abbassamento generale dell'accuratezza di circa 5-10%. Col il passare dei secondi e con l'inserimento di nuovi veicoli, la tecnologia *WiFi Direct* riesce a fornire all'utente un buon livello di accuratezza generale (65-70% dopo i primi 20 minuti, fino ad un 75% a fine simulazione) nonostante il range di comunicazione sia ridotto rispetto ad una tecnologia *V2V*.

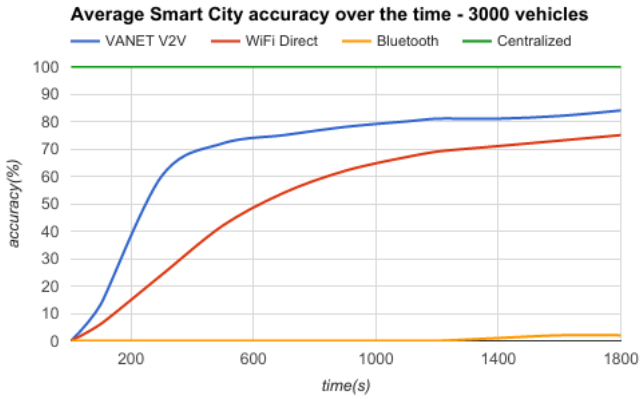

Figura 4.3: Grafico dell'accuratezza media della porzione di città al trascorrere del tempo - 3000 veicoli

L'accuratezza in base alla distanza

Il grafico in figura 4.4 mostra l'accuratezza delle informazioni in base alla distanza. Ricordando che i device si sincronizzano sulle informazioni della cella corrente e su quelle strettamente adiacenti, è normale pensare che l'accuratezza decresca con la distanza. L'ipotesi viene dimostrata dall'an-

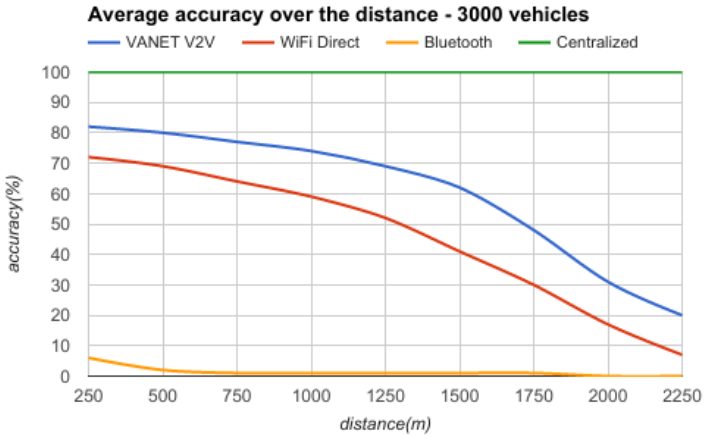

Figura 4.4: Grafico dell'accuratezza media in base alla distanza - 3000 veicoli

damento delle curve nel grafico 4.4, dove per ogni tecnologia l'accuratezza

migliore la si ottiene per la cella corrente e per quelle presenti nel raggio di 250-500m.

L'accuratezza in base al numero di veicoli

Il grafico in figura 4.5 mostra l'accuratezza media dopo 1800 simsec, per ogni tecnologia, in scenari con 3000, 1500 e 500 veicoli. Come citato in sezione 1.2, il tasso di partecipazione in sistemi Crowdsensing è un fattore determinante per un efficace processo di spreading dell'informazione. La tecnologia *V2V* riesce a mantenere livelli di accuratezza soddisfacenti anche con 1500 e 500 veicoli. Questo è determinato dall'alto range di comunicazione che il protocollo 802.11p riesce a garantire nella creazione di VANETs. L'accuratezza della tecnologia *WiFi Direct*, invece, decresce notevolmente con la diminuzione dei veicoli nella porzione di città considerata. Si passa da 70-72% con 3000 veicoli, a 35% con 1500 veicoli fino ad un 5% con 500 veicoli. La decrescità notevole dell'accuratezza, denota che il sistema tramite l'applicazione Android può funzionare in modo soddisfacente se il numero di utenti rimane elevato. La tecnologia *Bluetooth* non può essere considerata efficace in alcuna situazione.

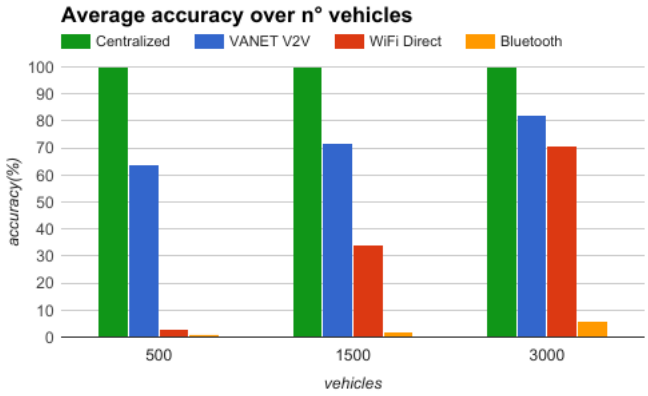

Figura 4.5: Grafico dell'accuratezza media in base al numero di veicoli

Conclusioni

La tesi descrive un sistema di Smart Parking, cercando di favorire l'attività di parcheggio dell'utente. Il sistema fornisce una mappa della città suddivisa in quadranti e per ognuno di questi viene indicata la probabilità di trovare parcheggio. L'utente, durante la fase di parcheggio, può individuare le zone che forniscono un'alta probabilità di trovare uno slot libero. L'applicazione favorisce l'utente stesso, ma aiuta anche a smaltire la congestione del traffico nella metropoli. Tramite un componente di activity recognition (basato su sensori) ogni Smartphone riesce a determinare quando l'utente parcheggia o rilascia la propria autovettura. Il sistema si basa sul concetto di Crowdsensing, dove ogni individuo condivide i propri eventi di parcheggio e rilascio con la comunità. In base alle informazioni condivise è possibile dare una stima delle zone più dense di parcheggi liberi della città. La propagazione delle informazioni non si basa su una struttura centralizzata, ma l'approccio è completamente distribuito tra i dispositivi utente. Gli Smartphone Android, a partire dalla versione 4.0, danno la possibilità di comunicare tra loro secondo il paradigma Device-To-Device (D2D). Il *WiFi Direct* è la tecnologia che permette lo spreading dell'informazione tra peer. Basandosi sullo standard 802.11, dà la possibilità di creare vere e proprie infrastrutture WiFi utilizzando soltanto i dispositivi mobile. Il meccanismo di disseminazione è stato reso totalmente automatico e trasparente all'utente. Una volta attivato permette di restare aggiornati sulla situazione parcheggi, sincronizzando spontanemante i propri dati con gli altri utenti nelle vicinanze. I risultati ottenuti in fase di valutazione, mostrano come l'efficacia del processo di

spreading cresca all'aumentare del tasso di partecipazione. Più utenti utilizzano l'applicazione, condividendo le proprie informazioni, più resteranno aggiornati sulla situazione parcheggi in città. Con una buona partecipazione si possono ottenere risultati di poco inferiori a tecnologie molto più costose ed articolate (come ad esempio sistemi Vehicle-To-Vehicle). Il meccanismo di disseminazione locale, oltre al problema del parcheggio, può essere facilmente adottato in altri ambiti (ad esempio il marketing di prossimità). In conclusione, è possibile pensare all'utilizzo del sistema in una città metropolitana in cui numerosi utenti contribuiscono al bene comune, sfruttando i servizi dell'applicazione.
Possibili sviluppi futuri potrebbero riguardare:

- un metodo per garantire l'esclusione di parcheggi privati (l'applicazione si basa su parcheggi on-street pubblici);
- un meccanismo di risparmio energetico e ponderare le attività di disseminazione in base allo stato della batteria;
- ottimizzare lo scambio dati inviando solo le entry necessarie, evitando così numerosi controlli;
- individuare automaticamente la cella con maggior quantità di slot liberi, in base alla destinazione dell'utente;
- dare priorità alle sincronizzazioni con device sempre diversi per aumentare lo spreding dell'informazione;
- introdurre un semplice sistema centralizzato che fornisca all'utente le probabilità di parcheggio nel caso in cui le connessioni *WiFi Direct* non risultino efficienti;
- migliorare la grafica dell'applicazione in modo tale da renderla più appetibile sul mercato.

Bibliografia

[1] "I Sistemi di Trasporto Intelligenti (ITS)", Ministero delle Infrastrutture e dei Trasporti, 2010, http://www.mit.gov.it.

[2] Rosario Salpietro, Luca Bedogni, Marco Di Felice, Luciano Bononi, "Park Here! A Smart Parking System based on Smartphones' Embedded Sensors and Short Range Communication Technologies", University of Bologna, 2015.

[3] Simone Masini, "Sviluppo di una Piattaforma di Crowdsensing per l'Analisi di Dati", Università di Bologna, 2014.

[4] Vladimir Coric, Marco Gruteser, "Crowdsensing Maps of On-street Parking Spaces", IEEE, 2013.

[5] Elena Polycarpou, Lambros Lambrinos, Eftychios Protopapadakis, "Smart parking solutions for urban areas", IEEE, 2013.

[6] Xiao Chen, Elizeu Santos-Neto, Matei Ripeanu, "Crowdsourcing for on-street smart parking", ACM, 2012.

[7] Rinne, Mikko, Törmä, Seppo, "Mobile crowdsensing of parking space using geofencing and activity recognition", Aalto University, 2014.

[8] Alessandro Grazioli, Marco Picone, Francesco Zanichelli, Michele Amoretti, "Collaborative Mobile Application and Advanced Services for Smart Parking", IEEE, 2014.

[9] Károly Farkas, Imre Lendák, "Simulation Environment for Investigating Crowd-sensing Based Urban Parking", IEEE, 2015.

[10] S.V. Srikanth, Pramod P.J., Dileep K.P., Tapas S., Mahesh U. Patil, Sarat Chandra Babu N., "Design and Implementation of a Prototype Smart PARKing (SPARK) System Using Wireless Sensor Networks", IEEE, 2009.

[11] Vanessa W.S. Tang, Yuan Zheng, Jiannong Cao, "An Intelligent Car Park Management Systembased on Wireless Sensor Networks", IEEE, 2006.

[12] Qing Li, Hongkun Li, Paul Russell Jr., Zhuo Chen, and Chonggang Wang, "CA-P2P: Context-Aware Proximity-Based Peer-to-Peer Wireless Communications", IEEE, 2014.

[13] Emad Abd-Elrahman, Adel Mounir Said, Thouraya Toukabri, Hossam Afifi, Michel Marot, "A Hybrid Model to Extend Vehicular Intercommunication V2V through D2D Architecture", IEEE, 2015.

[14] Jihoon Yang, Jorge Portilla and Teresa Riesgo, "Smart Parking Service based on Wireless Sensor Networks", IEEE, 2012.

[15] Muhammad Alam, Bruno Fernandes, João Almeida, Joaquim Ferreira, José Fonseca, "Integration of Smart Parking in Distributed ITS Architecture", IEEE, 2016.

[16] G. Revathi, V.R.Sarma Dhulipala, "Smart Parking Systems and Sensors: A Survey", IEEE, 2012.

[17] Rongxing Lu, Xiaodong Lin, Haojin Zhu, Xuemin Shen, "SPARK: A New VANET-based Smart Parking Scheme for Large Parking Lots", IEEE, 2009.

[18] IEEE Smart Cities, http://smartcities.ieee.org/about.html.

[19] Feng Xia, LaurenceT.Yang, LizheWang, Alexey Vinel, "Internet of Things", IEEE, 2012.

[20] Documentazione WiFi Direct, http://www.wi-fi.org/discover-wi-fi/wi-fi-direct.

[21] Documentazione ufficiale Android, https://developer.android.com/index.html.

[22] Standard Bonjour beacon, http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt.

[23] Documentazione Android sul D2D, https://developer.android.com/training/connect-devices-wirelessly/wifi-direct.html.

[24] Documentazione dello standard SQLite, https://www.sqlite.org/.

[25] Progetto fornito da Google su WiFi Direct e Bonjour beacon, https://android.googlesource.com/.

[26] Documentazione del simulatore OMNeT, https://omnetpp.org/.

[27] Documentazione di Veins, http://veins.car2x.org/.

[28] Vaishali D. Khairnar, Ketan Kotecha, "Performance of Vehicle-to-Vehicle Communication using IEEE 802.11p in Vehicular Ad-hoc Network Environment", IEEE, 2013.

[29] Luca Bedogni, Marco Gramaglia, Andrea Vesco, Marco Fiore, Jérôme Härri, Francesco Ferrero, "The Bologna Ringway dataset: improving road network conversion in SUMO and validating urban mobility via navigation services", IEEE, 2015.

[30] Daniel Camps-Mur, Andres Garcia-Saavedra, Pablo Serrano, "Device-to-device communications with Wi-Fi Direct: overview and experimentation", IEEE, 2013.

[31] Felipe Domingos da Cunha, Azzedine Boukerche, Leandro Villas, Aline Carneiro Viana, Antonio A. F. Loureiro, "Data Communication in VANETs: A Survey, Challenges and Applications[Research Report] RR-849", INRIA, 2014.

[32] Gábor Fodor, Erik Dahlman, Gunnar Mildh, Stefan Parkvall, Norbert Reider, György Miklós, Zoltán Turányi, "Design aspects of network assisted device-to-device communications", IEEE, 2012.