ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica

INGEGNERIZZAZIONE DI UN SISTEMA DI FIXED-COST REAL-TIME RIDE-SHARING

Elaborata nel corso di:
Sistemi Multi-Agente LS

Tesi di Laurea di:
IVANO MANCA

Relatore:
Prof. ALESSANDRO RICCI

Co-relatori:
Dott. Ing. ANDREA SANTI

ANNO ACCADEMICO 2010–2011
SESSIONE III

PAROLE CHIAVE

Real-time Ridesharing
Fixed-Cost Ridesharing
Carpooling
JaCa-Android
Mobile Application

Alla mia famiglia ed alla mia insegnante di canto.

Indice

Introduzione

L'intenso traffico automobilistico che caratterizza il mondo dei trasporti degli ultimi decenni ha una serie di effetti negativi ad alto impatto ambientale, sociale ed economico. Problemi come l'inquinamento atmosferico ed acustico, la congestione stradale e l'alto rischio di incidenti sono dovuti principalmente dall'aumento molto rapido dei veicoli privati in circolazione. La promozione di modalità di spostamento alternative e di un uso "intelligente" dei veicoli privati rappresentano oggi la strada maestra per affrontare i problemi legati al traffico e costituiscono le basi per organizzare e sviluppare una mobilità sostenibile.

Recentemente, il rapido aumento del costo dei carburanti utilizzati dai veicoli privati sta inevitabilmente cambiando le abitudini degli automobilisti. La necessità di risparmiare sulle spese del carburante porta oggi l'automobilista a cercare soluzioni utili al fine di ottimizzare l'uso del proprio veicolo. Una delle soluzioni più intuitive ed immediate adottate dagli automobilisti consiste nel praticare il *Carpooling* (o più in generale *Ridesharing*). La pratica consiste nel condividere il veicolo privato per un determinato viaggio, dividendone le spese con altri passeggeri spesso sconosciuti. Sebbene il Carpooling, sia per motivi economici che ecologici, stia avendo una discreta diffusione in determinati contesti, è quasi del tutto ignorato invece in alcuni paesi. Le barriere principali alla diffusione della condivisione del mezzo privato sono: il difficile rapporto di fiducia tra sconosciuti (guidatore e passeggero); il sistema finanziario adottato nel ripartire le spese di viaggio basato sulla spesa reale per sostenerlo e realizzato sulla base di transazioni bancarie o con denaro contante; la necessità di programmare i viaggi con sufficiente anticipo.

Questa tesi ha un duplice obiettivo. In primo luogo proporre un

sistema di Carpooling alternativo a quelli finora conosciuti, che prenderà il nome di *Fixed-Cost Real-time Ridesharing*, in secondo luogo progettare la piattaforma software, in particolare l'applicazione mobile, che ne rende possibile l'attuazione. Il sistema alternativo proposto mette in campo semplici soluzioni per migliorare i punti deboli dei sistemi classici riguardanti la politica finanziaria e la programmazione dei viaggi. Dal lato finanziario propone una suddivisione dei costi di viaggio basata sul numero di chilometri percorsi (fixed-cost) e sull'uso dell'unità di misura virtuale del *Credito Chilometrico* trasferita tra i viaggiatori attraverso transazioni digitali. Dal lato della programmazione dei viaggi prospetta la possibilità di realizzare l'Istant-pooling (prenotazione del viaggio in tempo reale) ed il monitoraggio in tempo reale del viaggio con la possibilità di localizzare su una mappa digitale le posizioni iniziali dei viaggiatori. Per realizzare concretamente le soluzioni proposte, la piattaforma software di supporto al *Fixed-Cost Real-time Ridesharing* deve rispondere pienamente a determinate caratteristiche dinamiche. In particolare deve poter localizzare in tempo reale i partecipanti al viaggio condiviso e tener traccia dell'evoluzione del viaggio per poi ripartirne automaticamente le spese secondo la politica del costo fisso proposta. Oggi è possibile rispondere a questo tipo di requisiti grazie alle moderne tecnologie mobile, alle loro funzionalità di geolocalizzazione e alla diffusione capillare della rete internet. In ultima analisi, la progettazione di sistemi software come quello proposto, può essere guidata da un approccio ad alto livello reso possibile oggi da una nuova generazione di piattaforme per sistemi mobile basate sul paradigma ad agenti come JaCa-Android.

Questo lavoro di tesi è sviluppato in cinque capitoli:

- Nel primo capitolo verrà descritto il sistema ***F**ixed-**C**ost **R**eal-time **R**idesharing* (*FCRR*) alternativo ai sistemi di condivisione del viaggio esistenti oggi. La trattazione partirà dalla definizione dei due sistemi conosciuti, il *Carpooling Classico* ed il *Real-time Ridesharing*, e dalla descrizione dello stato dell'arte sulle piattaforme software esistenti per i sistemi classici. Successivamente verrà illustrato il sistema finanziario a costo fisso (***C**arpooling a **C**osto **F**isso*) ed il risultato finale della sua integrazione con il Real-time Ridesharing, il *FCRR* appunto.

- Il secondo capitolo tratta l'analisi dei requisiti e l'analisi del problema relative al software di supporto al sistema ***F**ixed-**C**ost **R**eal-time **R**idesharing*. In particolare, verrà data una descrizione dettagliata dei requisiti del software, un'analisi dei requisiti e delle funzionalità attraverso i casi d'uso, una descrizione del modello del dominio del sistema ed infine verrà delineata l'architettura logica.

- All'interno del terzo capitolo è illustrata la progettazione del sistema software ***F**ixed-**C**ost **R**eal-time **R**idesharing* focalizzandosi principalmente sul sottosistema mobile ***F**CRR **M**obile **A**pplication* (*FMA*). In primo luogo è descritta l'architettura generale, in seguito vi è una breve trattazione del ***F**CRR **S**ervice* e la progettazione della *FMA*. In quest'ultima parte verranno preventivamente motivate le ragioni dell'approccio agent-oriented, introdotto il modello A&A adottato e la tecnologia mobile JaCa-Android , dopodichè verrà presentata l'architettura del sistema multi-agente della FCRR Mobile Application.

- Nel quarto capitolo verranno riportate parti significative dell'implementazione dell'applicazione mobile appartenente al sistema ***F**ixed-**C**ost **R**eal-time **R**idesharing* progettato nel capitolo precedente. Verrà studiato il comportamento degli agenti introdotti in fase di progettazione descrivendone in maniera separata ogni macro-attività svolta abbinandone un frammento del codice Jason che evidenzia i goal, eventuali sub-goal ed i piani reattivi più significativi. Verrà inoltre riportato un frammento di codice significativo per ogni artefatto definito ed infine illustrato il test del monitoring di un viaggio nel sistema *FCRRS*.

- Il quinto capitolo chiude questa tesi riportandone le conclusioni ed i possibili futuri sviluppi.

Capitolo 1

Fixed-Cost Real-time Ridesharing

In questo primo capitolo verrà presentato il sistema di condivisione dei viaggi su cui si basa il software progettato in questa tesi. Il sistema in oggetto si chiama ***F**ixed-**C**ost **R**eal-time **R**idesharing* e costituisce una alternativa ai sistemi di condivisione del viaggio noti ad oggi. Inizialmente verranno definiti i due principali sistemi di Carpooling (o Ridesharing) conosciuti, il primo nominato ***C**arpooling **C**lassico* (*CC*) ed il secondo, più avanzato tecnologicamente, denominato ***R**eal-time **R**idesharing*. Successivamente verrà rapidamente descritto lo stato dell'arte trattando le piattaforme software conosciute che supportano i sistemi citati sopra ed un cenno ai progetti di ricerca esistenti. Con l'obiettivo di migliorare i punti deboli dei primi due sistemi ed incentivare la diffusione della pratica del Carpooling, verrà proposto un terzo sistema, il ***C**arpooling a **C**osto **F**isso* (*CCF*), alternativo al Carpooling Classico ed in particolare al suo sistema finanziario di ripartizione dei costi. Infine, il sistema *CCF*, vedrà la sua naturale evoluzione nel sistema **F**ixed-**C**ost **R**eal-time **R**idesharing che abbina la dinamicità del sistema Real-time Ridesharing alle caratteristiche finanziarie del sistema *CCF*.

1.1 Carpooling Classico e Real-time Ridesharing

Il Carpooling Classico

Il *Carpooling* o *Ridesharing*[17] (termine inglese dal significato letterale di "auto di gruppo") è la condivisione di un viaggio tra due o più persone in una automobile privata. L'obiettivo principale è quello di ridurre il costo del viaggio (in temini di carburante, pedaggi) dividendone le spese ed eventualmente anche lo stress del viaggio alternandosi alla guida.

Nel Carpooling Classico, i *carpooler*, ovvero coloro i quali promuovono e praticano il carpooling, spesso utilizzano piattaforme web o applicazioni mobile (par.1.3) che favoriscono l'incontro tra la domanda e l'offerta dei passaggi. Il carpooler proprietario del mezzo, decide di condividere con altri carpooler una determinata tratta al fine di risparmiare sul costo del viaggio dividendolo poi in parti uguali con gli altri passeggeri.

Il passeggero che desidera viaggiare in carpooling, cerca mediante la piattaforma software (che ne offre il servizio) un'offerta di pooling relativa alla stessa tratta che vuole percorrere. Il software presenta al passeggero la lista delle offerte di pooling in precedenza inserite dai carpooler proprietari delle autovetture. Il carpooler passeggero, scelta l'offerta di suo interesse, richiede formalmente il passaggio in pooling al proprietario, il quale, qualora lo ritenga opportuno, lo accetta come passeggero. Nei sistemi di carpooling più avanzati, il software supporta il carpooler passeggero nella realizzazione di pooling a tappe (*multiple drivers per trip*) integrando nei suggerimenti l'eventualità di effettuare tappe tramite trasporto pubblico.

Normalmente è il proprietario del mezzo che fissa il costo del viaggio (*handled by provider*) secondo una sua personale stima. Per questa ragione, un passeggero, oltre a dover cercare l'offerta del passaggio in pooling sulla tratta interessata, solitamente va anche alla ricerca del costo di viaggio, stimato e proposto dal proprietario, per lui più conveniente.

Il carpooling ha anche degli interessanti risvolti sociali. Capita spesso infatti, che la condivisione del percorso avvenga tra sconosciuti

che scoprono di coltivare interessi comuni. La stessa destinazione del viaggio infatti, crea fin troppo spesso l'occasione per far conoscere persone che hanno l'interesse di partecipare ad uno stesso evento di carattere sportivo, culturale, in ambito lavorativo o semplicemente per turismo o vacanza. Il carpooling assume poi una certa importanza nel *commuting*, ovvero nella condivisione del viaggio per i pendolari che compiono regolarmente la stessa tratta per raggiungere il posto di lavoro o di studio. In realtà, il carpooling finalizzato al *commuting* è stato il più conosciuto e diffuso fino ad oggi.

L'aspetto sociale però presenta anche delle problematiche relative alla fiducia negli sconosciuti. La mancanza di fiducia è sempre stata un freno alla diffusione di questa pratica. Oggi però, possediamo gli strumenti per sbloccare almeno in parte questa paura. La diffusione dei social network negli ultimi anni, ha reso possibile qualcosa di impensabile fino a non troppo tempo fa: l'esplorazione della rete di conoscenze tra persone. Oltre alla possibilità di recuperare delle informazioni private su una persona sconosciuta, ammesso che siano state condivise, si possono conoscere eventuali amicizie comuni. Le amicizie comuni costituiscono una garanzia molto forte sull'affidabilità della persona, in virtù del fatto che spesso ci si fida delle informazioni che una persona amica fornisce su una a noi del tutto sconosciuta. Per aumentare ulteriormente la fiducia nell'altro, molto di frequente, i carpooler preferiscono restringere ad una sola categoria di persone la possibilità di partecipare al viaggio, come ad esempio avviene nel "carpooling rosa", che per ragioni molto semplici da intuire, riguarda soltanto carpooler di sesso femminile. In alcuni casi, con l'obiettivo di premiare gli utenti affidabili, le piattaforme software di supporto al carpooling consentono lo scambio di feedback tra i carpooler imitando un po' ciò che avviene per alcune piattaforme per il commercio elettronico (*e-commerce*).

Il Real-time RideSharing

Il *Real-time Ridesharing*[18] (termine inglese dal significato letterale di "corsa/viaggio condiviso in tempo reale") è una forma estremamente dinamica di car pooling. La definizione che ne dà Andrew Amey[1]

> *"[...] a single, or recurring rideshare trip with no fixed schedule, organized on a one-time basis, with matching of participants occurring as little as a few minutes before departure or as far in advance as the evening before a trip is scheduled to take place."*
>
> *"[...] un singolo, o ricorrente viaggio condiviso non programmato preventivamente, organizzato di volta in volta, in cui il matching tra i partecipanti avviene pochi minuti prima della partenza o al massimo la sera prima del viaggio stesso."*

riassume egregiamente le caratteristiche principali del Real-time Ridesharing, ma può essere estesa dinamicizzandola ulteriormente. Oggi il Real-time Ridesharing permette ai carpooler proprietari del mezzo di organizzare il viaggio sottoponendo le informazioni dell'offerta di pooling mediante applicazioni mobile (ad esempio su smartphone) o desktop fornite di connessione internet. La possibilità di rilevare costantemente la posizione esatta di un dispositivo (quindi del carpooler), grazie alle tecnologie di geolocalizzazione, apre nuovi scenari mai pensati prima. Di norma, nel Real-time Ridesharing, il carpooler proprietario della vettura su cui offre il passaggio, preleva i carpooler passeggeri dopo averli raggiunti seguendo la loro posizione sulla mappa digitale. Oltre ad avere il vantaggio di tenere informati i carpooler sullo stato del viaggio in tempo reale e sulla la posizione degli altri carpooler, permette di gestire eventuali richieste di pooling aggiuntive che intercorrono durante il viaggio (*multiple riders per trip*).

1.2 Vantaggi del Carpooling

Per dare un'idea più precisa di quanto sia importante il carpooling, elenchiamo i vantaggi principali che ne derivano dal praticarlo in qualsiasi contesto:

- Riduzione dei costi di trasporto pro-capite
- Riduzione del traffico
- Riduzione dell'occupazione di parcheggio pubblico
- Riduzione delle emissioni di CO_2 e di altre sostanze inquinanti (in conseguenza al minore traffico)
- Transito sulle **HOV** (***H**igh-**O**ccupancy **V**ehicle lane*) [fig. 1.1]
- Sconti sui pedaggi autostradali
- Miglioramento dei rapporti sociali
- Accessibilità a luoghi non raggiungibili attraverso i trasporti pubblici

Figura 1.1: Segnale stradale HVO in Norvegia [19]

1.3 Stato dell'arte

La pratica del Carpooling iniziò (in totale assenza di tecnologie informative) durante la Prima Guerra Mondiale con i Ridesharing Clubs per poi riemergere negli USA agli inizi degli anni settanta spinto dalla crisi energetica.[6] Negli ultimi anni il Carpooling è diventato un fenomeno di massa in paesi come gli Stati Uniti ed il Canada, inoltre soprattutto nelle metropoli è incentivato da numerose iniziative e fondi governativi. In Europa è abbastanza diffuso in Germania ed in Francia, si pensi soltanto che nel tratto Berlino-Amburgo l'attività di Carpooling è così frequente che un passeggero deve attendere in media soltanto un quarto d'ora per trovare un automobilista che offra il viaggio in condivisione. Molti paesi del Nord Europa hanno da tempo fornito le autostrade a traffico più intenso di corsie riservate alle sole vettura con più di un passeggero a bordo.

Piattaforme software di supporto

Il Carpooling in Italia stenta a diffondersi. Tuttavia, esistono una molteplicità di piattaforme web che supportano i carpooler nel programmare il viaggio condiviso. *Tandemobility*[1], *RoadShare*[2], *Carpooling.it*[3], *Postoinauto.it* [4], sembrano essere le più utilizzate. Esse però, offrono supporto ad un servizio di Carpooling di tipo classico, nessun Real-time Ridesharing, forniscono soltanto una funzionalità di incontro tra domanda ed offerta (*matching*) curando maggiormente l'aspetto social con l'obiettivo di aumentare la fiducia tra carpooler sconosciuti.

Molte delle piattaforme software più avanzate implementano il Real-time Ridesharing e sono accessibili dai dispositivi mobile (in particolare smartphone), sfruttandone al meglio le funzionalità di geolocalizzazione. Di seguito una descrizione delle piattaforme più conosciute ed avanzate tecnologicamente presenti oggi:

[1] `http://m.tandem.imginternet.it`
[2] `http://www.roadsharing.com`
[3] `http://www.carpooling.it`
[4] `http://www.postoinauto.it`

Avego [2], è la piattaforma sviluppata dall'omonima compagnia irlandese (spinoff della Mapflow, UK). Realizza un servizio di Real-time Ridesharing avanzato con gestione dei pagamenti, informazioni real-time ai carpooler, meccanismi di sicurezza per la fiducia verso gli altri carpooler.

Dopo il matching guidatore-passeggero, Avego calcola il costo da far pagare al passeggero proporzionandolo ai chilometri della tratta da percorrere. Se entrambi i carpooler accettano la cifra proposta da Avego allora ha inizio il Real-time RideSharing, il guidatore sa dove andare per effettuare il pick-up ed il passeggero visualizza in tempo reale la posizione del guidatore. Quando è in prossimità del passeggero, il guidatore viene informato da un segnale acustico. Il guidatore autentica il passeggero digitando nella sua applicazione il PIN comunicatogli da quest'ultimo. Al termine del viaggio, Avego effettua la transazione della cifra dal conto del passeggero a quello del guidatore. Entrambi i carpooler rilasciano un feedback l'uno dell'altro che va da una a cinque stelle. L'applicazione è disponibile soltanto per *iPhone* e *Windows Phone 7*.

Tra i **punti deboli**: indisponibilità per *Android*; costi predefiniti mediamente alti (superiori ai $0,30$€/Km) ed obbligo di pagamento anticipato (prima ancora di una prenotazione) in "ricariche" da 10, 50, 100 euro; la parte social fa affidamento soltanto sui feedback, nessun social network è integrato.

Zimride [20], è la piattaforma web ideata da John Zimmer. Non realizza un Real-time Ridesharing avanzato, ma un servizio di Carpooling classico, pertanto non è disponibile come applicazione mobile. Gestisce separatamente il *commuting* (viaggio ricorsivo, tipicamente per lavoro) dal *one-time travel* (viaggio occasionale).

Contestualmente al matching guidatore-passeggero, realizzato anche in base al rimborso richiesto dal carpooler proprietario, Zimride fa visualizzare al passeggero il punto di pick-up dal quale inizierà il viaggio in pooling. Per le transazioni Zimride fa affidamento a *Paypal*[5]. La transazione non viene effettuata al termine

[5] `https://cms.paypal.com`

del viaggio, ma soltanto il giorno successivo. Per il lato social, Zimride ha l'approccio più innovativo. Fa affidamento alla mole enorme di informazioni garantita da *Facebook*[6]. Facebook, oltre a far conoscere meglio la persona che c'è dietro al carpooler, ha la possibilità di far sapere ai carpooler se si hanno conoscenti in comune e questo è il massimo dal punto di vista della garanzia percepita.

Tra i **punti deboli**: non realizza il Real-time Ridesharing; costo del viaggio altamente variabile poichè la stima è a carico del carpooler proprietario (anche se in alcuni casi consente la contrattazione).

iCarpool [9], è la startup vincitrice del *ITS Congestion Challenge* di IBM[7] fondata da Lakshmi Krishnamurthy e Amol Brahme. E' un sistema studiato per promuovere il carpooling nel *commuting*. In tal proposito, iCarpool annovera un elenco di aziende ed organizzazioni pubbliche e private le quali si affidano alla piattaforma per promuovere il pooling ai loro dipendenti o associati. Realizza un servizio di Real-time Ridesharing avanzato con informazioni real-time ai carpooler e una forte dinamicità nel last-minute carpooling. Implementa un meccanismo di matching in cascata per cui un passeggero che è in ritardo per il pick-up, può al volo riprogrammare il pooling con una vettura che percorrerà la stessa tratta poco tempo dopo. L'aspetto social è tutto incentrato sul profilo dei dipendenti o associati, per cui vi è una forte fiducia nei carpooler poiché spesso il compagno di viaggio è un collega di lavoro o lavora per una compagnia collocata nelle vicinanze della propria. L'applicazione è disponibile per *iPhone* e a breve lo sarà anche per *Android*.

Tra i **punti deboli**: indisponibilità per *Windows Phone*; l'aspetto finanziario e social è gestito in collaborazione con l'azienda o l'organizzazione cliente di iCarpool.

[6] `http://www.facebook.com`
[7] `http://www-03.ibm.com/press/us/en/pressrelease/28467.wss`

Ricerca

Nel 2009 il *Massachusetts Institute of Technology* (*MIT*) con la supervisione del prof. John Attanucci ha avviato un progetto di ricerca denominato *Real-Time Rideshare Research*[10] che mira ad

> (*"identify barriers to improved rideshare participation and develop some strategies on how to overcome them"*).
>
> *"identificare le barriere al miglioramento della partecipazione al viaggio condiviso e sviluppare delle strategie su come superarle"*

Uno studio[8] del *Nokia Research Center* del 2007, stima che nel mondo che ci siano in circolazione più 500 milioni di veicoli. Inoltre, calcola per gli Stati Uniti, una stima del numero medio di passeggeri a bordo di un veicolo, ricavandone un dato molto significativo: 1.5 passeggeri per auto. Questo significa avere in media soltanto 3 passeggeri ogni 2 auto. É facile intuire che sotto i nostri occhi, quotidianamente assistiamo ad un enorme spreco di carburante, quindi di energia in genere. Allo spreco dell'energia va ad aggiungersi lo spreco di denaro pubblico per progettare e costruire infrastrutture per i trasporti con capacità di gran lunga superiori alle reali necessità della popolazione.

1.4 Sistema *FCRR*

1.4.1 Il costo del viaggio nel Carpooling Classico

Nel Carpooling Classico (par.1.1) il costo reale del viaggio viene equamente ripartito tra i carpooler. Il costo reale pro-capite del viaggio condiviso dipende generalmente da tre fattori:

1. **Costo del carburante o dell'energia consumati**, variabile a seconda del mercato e del combustibile o dell'energia; utilizzati dal mezzo.

2. **Quantità di carburante o dell'energia consumati**, che a sua volta dipende da altri tre fattori:

   - **Numero dei chilometri percorsi**;
   - **Efficienza del mezzo**, ovvero quanti chilometri è in grado di percorrere la vettura consumando un certo quantitativo di carburante o energia;
   - **Stile di guida** del conducente;

3. **Costo del pedaggio**, che può essere nullo, fisso o variabile (in base al mezzo, al possesso di licenze sconto, ecc...).

Come è evidente dall'elenco dei fattori coinvolti, i costi di un viaggio (escludendo il numero dei chilometri percorsi) dipendono fortemente dal mezzo utilizzato e dal conducente/proprietario. Pertanto, nel Carpooling Classico, il passeggero ha un forte interesse nel trovare di volta in volta il viaggio del carpooler che pubblica il più basso costo stimato.

Dal mio personale punto di vista, poiché la responsabilità del costo di viaggio ricade quasi esclusivamente sul carpooler proprietario del mezzo, è su quest'ultimo che dovrebbero ricadere onori ed oneri delle spese di viaggio. Per avallare questa tesi, mettiamo a confronto con la tabella 1.1 i costi medi di un viaggio di 200 chilometri percorsi da quattro carpooler su tre mezzi differenti: un *SUV diesel*, un'*auto a metano* ed una *elettrica*.

Consumi/Auto	**SUV**	**Panda N.P.**	**Nissan Leaf**
Carburante/Energia	Diesel	Metano	En. Elettrica
Prezzo carburante*	1.75 €/L	0.95 €/Kg	0.18 €/KWh
Consumo medio	10 Km/L	25 Km/Kg	4.9 Km/KWh⋆
€/Km	0.17	0.038	0.037
€/200 Km	35	7.6	7.4
€ pro-capite (4 p.)	8.75	1.90	1.85
€ di Rimborso	26.25	5.7	5.55

* Prezzo medio in Italia al 28/01/2012

⋆ 20.39 kWh per percorrere 100 Km

Tabella 1.1: Confronto consumi medi auto

Osservazioni sui consumi medi

La voce **€ pro-capite (4 p.)** in tabella 1.1 evidenzia il costo sostenuto da ogni passeggero per rimborsare i 200 Km del viaggio effettuato in pooling. É evidente come i passeggeri che hanno percorso la tratta a bordo del SUV debbano sostenere una spesa 4.7 volte maggiore dei passeggeri a bordo dell'auto elettrica. Questo dato suggerisce che, molto probabilmente, i passeggeri in un sistema di Carpooling Classico, avranno tutto l'interesse ad evitare il pooling su un mezzo così poco conveniente come il SUV. Magari, in assenza di alternative, sceglieranno di fare affidamento ad altri trasporti più convenienti e garantiti come i treni. Il carpooler proprietario del SUV, avrà grandi difficoltà ad occupare tutti i posti disponibili nel proprio mezzo, perché il suo servizio di pooling è assolutamente sconveniente. Le conseguenze che ne derivano dal punto di vista ecologico e logistico sono tutt'altro che buone. Il SUV, oltre ad essere svantaggioso per i carpooler, a parità di posti passeggero è uno dei mezzi più inquinanti ed anche il mezzo che occupa maggiore spazio nei parcheggi pubblici.

Adesso prendiamo in considerazione il dato **€ pro-capite (4 p.)** minore, quello relativo all'auto elettrica. Indubbiamente, 1.85€ sono una cifra più che conveniente per i passeggeri, poiché inferiore al costo medio di qualsiasi altro mezzo di trasporto utilizzabile. Pertanto è facilmente prevedibile che, a fronte dell'offerta di pooling su un'auto

elettrica, si possa scatenare tra i carpooler passeggeri una vera e propria "corsa all'acquisto del posto". Consideriamo però la situazione dal punto di vista del proprietario dell'auto elettrica. Sicuramente avrà poche difficoltà nell'occupare tutti i posti disponibili nella sua auto, ma non dimentichiamo che, oltre a ragioni di carattere ecologico il pooling viene praticato dai proprietari principalmente per risparmiare sulle spese di viaggio. Il carpooler proprietario dell'auto elettrica, raggiunge un risparmio complessivo di 5.55€ (visibile alla voce **€ di Rimborso**) su ben 200 Km di viaggio e dopo aver prelevato (da location diverse nel caso di Real-Time Ride Sharing) ben altri tre passeggeri (ed altrettanti bagagli). É facile intuire come il proprietario di un'auto elettrica sia molto meno motivato rispetto al proprietario di un SUV nel praticare il carpooling. Un esiguo risparmio di soli 5.55€ sommato al "fastidio" di dover accontentare altri tre passeggeri potrebbero essere motivazioni sufficienti per inibire il proprietario di una auto con bassissimi costi nel praticare in modo costante l'attività di carpooling.

1.4.2 Carpooling a Costo Fisso (*CCF*)

In un sistema *CCF* (***C**arpooling a **C**osto **F**isso*) i passeggeri non rimborsano il proprietario in base al costo reale del viaggio, ma in base ad un costo fisso per chilometro percorso. Introduciamo il concetto di ***Credito Chilometrico*** (cc) definito come l'unità di misura del *credito di pooling* acquisibile ed accumulabile da un carpooler nel suo conto privato fornitogli dal sistema. Il credito di pooling può essere acquisito in una delle seguenti modalità:

- Acquistandolo dal sistema a fronte di un pagamento in denaro
- In seguito ad un trasferimento dall'account di un altro carpooler

Il costo da pagare al sistema per l'acquisto di un singolo cc deve essere inferiore al costo medio per chilometro di un trasporto pubblico come ad esempio il treno [8] e molto vicino ad un consumo medio delle autovetture in genere, per cui lo fisseremo empiricamente ad un valore di 0.07 €/km.

[8] Circa 0.105 €/Km per un treno InterCity, vedi tabella 1.2

Costo/Mezzo	**SUV**	**Panda N.P.**	**Nissan Leaf**	**TrenoIC 2Cl**
€/200 Km*	35	7.6	7.4	21
€/Km*	0.17	0.038	0.037	0.105

* Nel calcolo non si tiene conto degli eventuali costi di pedaggio.

Tabella 1.2: Confronto costi medi dei mezzi

Infine, i passeggeri non rimborsano il proprietario direttamente in denaro, ma lo fanno a fine viaggio in maniera automatica e digitale mediante i propri cc presenti nel conto e probabilmente acquisiti dal sistema direttamente o non (tramite trasferimento) al *costo di riferimento per l'acquisto* (CRA) di 0.07€ per cc.

In questo sistema di carpooling, chiamato **"a costo fisso"**, i tre passeggeri dell'esempio precedente, sia che abbiano viaggiato sul SUV diesel o nell'auto a metano o nell'auto elettrica trasferiscono automaticamente a fine viaggio un credito chilometrico di 200 cc (uno per ogni chilometro percorso) dal proprio conto al conto del carpooler proprietario del mezzo. Pertanto, qualunque mezzo abbia usato e qualunque sia stata la spesa reale del viaggio per il carpooler proprietario, il suo personale credito chilometrico acquisito sarà di 200 cc per ogni passeggero, per un totale quindi di 600 cc.

Nel sistema *CCF*, il proprietario può scegliere in qualsiasi momento di conservare i cc acquisiti o di cederli al sistema. La cessione (o restituzione) dei cc al sistema, ha come contropartita un rimborso in denaro che il sistema accredita al carpooler ad un prezzo fissato preventivamente e di poco inferiore al costo di riferimento per l'acquisto dei cc. Prendiamo come *costo di riferimento per la cessione* (CRC) dei cc il valore di 0.06 €/km. In caso di cessione dei cc acquisiti da parte del carpooler proprietario dell'esempio discusso, il sistema *CCF* accredita, in denaro, sul conto corrente bancario del carpooler la somma di 12€ (= 0.06€∗200). Così facendo, al termine dell'operazione, il sistema trattiene la differenza tra il denaro versato dai passeggeri per acquistare i 200 cc e il denaro ricavato dal proprietario in seguito alla cessione dei cc acquisiti. Questa differenza è il *ricavo del sistema* ed in questo caso ammonta a 2 €.

In aggiunta, per aumentare la fiducia nel sistema e verso gli altri carpooler, il *CCF* implementa un meccanismo di prenotazione. Il meccanismo prevede l'obbligo del trasferimento di un anticipo di una quota di cc (proporzionata ai chilometri della tratta da prenotare) da parte dei passeggeri per assicurarsi il proprio posto in vettura. La prenotazione con acconto disincentiva i forfait da parte dei passeggeri ed assicura in questi casi un indennizzo al carpooler proprietario. Inoltre, nel caso in cui un passeggero volesse cancellare una prenotazione con poco anticipo rispetto alla partenza del viaggio, perderebbe la quota di cc versati in anticipo.

Nelle tratte brevi (meno di 50Km percorsi), il rimborso in cc non costituisce una cifra motivante per il carpooler proprietario, per questo motivo il sistema prevede altre proporzioni (ad esempio un rimborso doppio rispetto ai chilometri percorsi).

1.4.3 *CCF* vs *CC*

Ponendo l'attenzione sui carpooler proprietari nell'esempio delle tre auto, consideriamo la differenza tra il **rimborso** totale versato dai passeggeri ed il **costo reale** sostenuto dal carpooler proprietario, differenza che chiameremo **bilancio**. Verifichiamo i valori di bilancio per i carpooler proprietari nel caso di Carpooling Classico (*CC*) e nel caso di Carpooling a Costo Fisso (*CCF*) (vedi tabella 1.3).

Sistema/Mezzo	**SUV**	**Panda N.P.**	**Nissan Leaf**
***CC*(€)**	−8.75	−1.9	−1.85
***CCF*(€)**	−23	+4.4	+4.6
***CC*(€/Km)**	−0.044	−0.0095	−0.0092
***CCF*(€/Km)**	−0.115	+0.022	+0.023

Tabella 1.3: Bilanci (rimborso-costo reale)

Osservazioni sui bilanci

Considerando la tabella 1.3 che illustra i bilanci di viaggio dei tre diversi proprietari nei due sistemi di carpooling (*CC* e *CCF*) emergono una serie di aspetti interessanti che andiamo ad esaminare:

- Il bilancio del carpooler proprietario del **SUV** è negativo sia nel caso che operi in un sistema *CC* che in un sistema *CCF*. Questo significa che praticare un'attività di carpooling, in entrambi i sistemi corrisponde ad un risparmio sulle spese di viaggio. Il risparmio è maggiore nel sistema *CC* (bilancio a -8.75€) rispetto al *CCF* (-23€), ma non bisogna dimenticare che il minore risparmio nel caso del sistema *CCF* potrebbe essere compensato (se non superato) a lungo termine, grazie alla maggiore probabilità nel sistema *CCF* di occupare interamente i posti disponibili nella vettura (vedi par. 1.4.1).

- Il bilancio del carpooler proprietario dell'**auto elettrica**, ma anche quello del carpooler proprietario dell'**auto a metano**, è negativo nel caso in cui operi in un sistema *CC* (-1.85€ e -1.9€). Questi due valori indicano che l'attività di carpooling in un sistema CPC corrisponde soltanto ad un risparmio sulle spese. Se invece i due proprietari in questione svolgono attività di carpooling in un sistema *CCF* i loro bilanci hanno segno positivo ($+4.4$€ nel caso dell'auto a metano e $+4.6$€ nel caso dell'auto elettrica). I due bilanci di segno positivo presentano ai carpooler proprietari di vetture a bassissimi consumi operanti in un sistema di tipo *CCF*, la possibilità non solo di viaggiare gratis (che si otterrebbe con un valore 0€ del bilancio), ma addirittura di conseguire un guadagno in seguito all'attività di condivisione. Nel sistema *CCF* quindi, i proprietari di vetture a bassissimi consumi, trovano una fortissima motivazione nel praticare il carpooling grazie alla possibilità di guadagnare semplicemente condividendo la propria auto.

Riepilogo vantaggi e svantaggi del *CCF*

Ai vantaggi dei sistemi di Carpooling Classico elencati nella Sezione 1.2 si aggiungono quelli del Carpooling a Costo Fisso, che rafforza molti

dei punti deboli dei sistemi *CC* e incentiva notevolmente la pratica del carpooling con mezzi di trasporto a basso consumo ed ecologici.

Riassumiamo i vantaggi del sistema *CCF* dal punto di vista dei **carpooler proprietari**:

- Facilita l'occupazione dei posti liberi nei mezzi con un alto costo reale del viaggio;
- Consente di viaggiare gratuitamente o guadagnando nei mezzi con basso costo reale del viaggio;
- Elimina il fastidio di dover calcolare la stima del costo di viaggio;
- Elimina la spiacevole pratica del dover *chiedere* denaro contante ai passeggeri;
- Elimina la pratica di avere i rimborsi in contanti alla fine del viaggio ed i problemi che ne derivano (ad esempio l'eventuale non disponibilità di resto contante);
- Consente di monetizzare i cc acquisiti cedendoli al sistema;
- Consente di accettare una prenotazione di pooling richiesta da un carpooler passeggero sotto l'assicurazione dell'anticipo di una parte dei cc necessari per il rimborso del viaggio;
- Migliora la fiducia nel sistema, quindi anche nelle relazioni sociali per via dell'assenza di scambi di denaro contante;
- Consente di accumulare cc e di riusarli in qualità di carpooler passeggero.

Dal punto di vista dei **carpooler passeggeri**:

- Elimina la preoccupazione di dover cercare l'offerta di pooling a costo minimo sulla tratta interessata;
- Consente di acquistare in anticipo i cc con cui rimborsare le spese di viaggio senza l'uso di contanti ed in maniera automatica e digitale;

- Migliora la fiducia nel sistema, quindi anche le relazioni sociali per via dell'assenza di scambi di denaro contante;
- Consente, qualora lo si voglia, di cedere al sistema i cc acquistati recuperando la quasi totalità del corrispettivo in denaro.

Dal punto di vista **ecologico/ambientale**:

- Incentiva fortemente l'acquisto e l'utilizzo di mezzi a bassi consumi e costi di viaggio, poiché in questi casi, il carpooling consentirebbe di ammortizzarne la spesa d'acquisto degli stessi per poi a lungo termine procurare del guadagno;
- Favorisce il carpooling nei mezzi con alto costo reale di viaggio riducendo ulteriormente le emissioni di CO_2 e di altre sostanze inquinanti.

Dal punto di vista del **sistema che offre il servizio *CCF***:

- Genera un ricavo del $14,3\%$ sul prezzo di ogni cc acquistato dai carpooler a fronte del servizio informativo offerto;
- Fornisce una grande quantità di dati sulle abitudini e sugli spostamenti di automobilisti (e non) in una area geografica di interesse;

Lo svantaggio più evidente in un sistema *CCF* è l'obbligo per un carpooler proprietario di possedere un dispositivo mobile che gli garantisca l'accesso continuo al sistema. Questo vincolo in futuro potrà sicuramente essere superato dalla probabile presenza di un computer di bordo in ogni nuovo mezzo prodotto. Volendo individuare altri **svantaggi** presenti nel sistema *CCF* potremmo prevedere una certa difficoltà nel far crescere l'utenza del sistema nella fase iniziale. Il problema dello *slow start* è tipico dei sistemi di car pooling in genere. Non è difficile immaginare come una scarsa presenza di carpooler in una certa area geografica renderebbe difficile trarre grossi benefici nel praticare questo tipo di attività. Tuttavia, decidere di praticare il carpooling in sistemi *CCF* non comporta, se l'accesso al sistema è gratuito, alcun tipo di spesa iniziale. Una minima perdita si avrebbe nei casi di carpooler passeggeri che acquistano e restituiscono crediti

senza sfruttarli. Nella restituzione dei cc al sistema il carpooler perde solo una piccola percentuale di denaro, ma poichè la donazione dei cc tra carpooler in questo sistema è consentita, nessuno vieta ad un carpooler di limitare ulteriormente la perdita di denaro rivendendo i suoi cc ad altri carpooler ad un costo minore rispetto al CRA.

Costi di riferimento dei *Crediti Chilometrici*

I costi di riferimento **d'acquisto** (CRA) e di **cessione** (CRC) dei cc indicati nella Sezione 1.4.2 sono stati identificati empiricamente. Per programmarne la variazione nel tempo, nel caso del costo d'acquisto si deve seguire il valore del costo medio chilometrico delle vetture in genere. Il costo di cessione può essere agganciato al costo d'acquisto attraverso un valore percentuale fisso o può essere modulato in base alle esigenze del momento.

1.4.4 Sistema Fixed-Cost Real-time RideSharing

Un sistema denominato ***F**ixed **C**ost **R**eal-time **R**ideSharing* (*FCRR*) abbina le caratteristiche finanziarie dei sistemi di Carpooling a Costo Fisso (*CCF*) (vedi Sezione 1.4.2) a quelle dinamiche dei sistemi di Real-time RideSharing (vedi Sezione 1.1).

Per implementare via software un sistema *FCRR* è necessario disporre di tecnologie in grado di rilevare costantemente la posizione geografica dei dispositivi di accesso al sistema e di rispondere efficacemente ai requisiti dinamici richiesti in questo particolare tipo di pooling organizzato. In effetti, i sistemi Real-time RideSharing sono di norma supportati da tecnologie *mobile* mediante le quali è possibile rilevare in tempo reale la variazione della posizione dei carpooler all'interno di un'area geografica.

Essendo un semplice merge di due sistemi di pooling complementari come *CCF* e Real-time RideSharing, il *FCRR* presenta i vantaggi di entrambi i sistemi di origine, semplificando e migliorando alcuni aspetti che nei rispettivi di origine erano trascurati. Pertanto, evitando di rielencare i vantaggi che emergono nell'impiego dei sistemi *CCF* (Sezione 1.4.3) e Real-time RideSharing (Sezione 1.1), poniamo l'attenzione sulle nuove funzionalità che fornisce il sistema *FCRR*:

Pooling last-minute (o Instant-pooling). Le caratteristiche dinamiche e "finanziarie" dei sistemi *FCRR* permettono di semplificare la realizzazione del *pooling last-minute* (LMP). Nel caso di LMP le caratteristiche ereditate dal sistema Realt-time RideSharing sono abilitanti dal punto di vista del matching domanda-offerta poiché un carpooler proprietario può accettare anche durante il viaggio (in real-time) una nuova richiesta di prenotazione da parte di un carpooler passeggero. Inoltre la posizione del nuovo carpooler è visualizzata sulla mappa del carpooler proprietario semplificando ulteriormente l'operazione di *picking-up* (prelievo del carpooler passeggero). Grazie alle caratteristiche ereditate dal sistema *CCF*, non è necessario ricalcolare la quota pro-capite del costo di viaggio poiché è un costo fisso che dipende soltanto dal numero dei chilometri percorsi. Il sistema *FCRR* terrà traccia dei chilometri e quindi dell'ammontare di cc che automaticamente preleverà dal conto del nuovo carpoo-

ler passeggero a fine viaggio. Infine, grazie al meccanismo di prenotazione del sistema *CCF*, il proprietario ottiene **istantaneamente** l'anticipo di una quota dei cc e accetta la richiesta con la garanzia di accredito dell'anticipo. Questo eviterebbe al carpooler proprietario di dover cambiare inutilmente il percorso di viaggio in caso di richieste di prenotazione errate o malevoli.

Informazioni real-time su imprevisti. Un eventuale imprevisto può essere notificato in tempo reale dal carpooler proprietario ai carpooler in attesa mediante i contatti che i carpooler condividono nel sistema. Qualunque altro tipo di comunicazione da parte del carpooler proprietario può essere inviata agli altri carpooler tramite gli stessi contatti.

Trasferimento elettronico di cc. Come anticipato nel caso del LMP, il sistema *FCRR* eredita la caratteristica dei rimborsi elettronici tramite credito chilometrico abilitando e velocizzandone l'esecuzione. I cc, nel caso di sistemi *CCF*, corrispondono ad una somma virtuale (ma monetizzabile) di credito.

Check-in e check-out. Grazie alla funzionalità di geolocalizzazione ereditata dai sistemi Real-time RideSharing, il *FCRR* realizza meccanismi di *check-in* e *check-out*. Il check-in è l'azione che compie un carpooler passeggero nell'atto di entrare in vettura al momento in cui viene prelevato (*picked-up*) dal carpooler proprietario. É il proprietario stesso a confermare l'avvenuto check-in al sistema. Dall'istante di check-in in poi, il sistema è a conoscenza della partecipazione al viaggio del determinato passeggero. Contestualmente all'operazione di check-in il passeggero comunica al guidatore un codice fornitogli dal sistema al momento della prenotazione. Il codice è utile all'identificazione del passeggero ed a dare conferma al sistema che il passeggero ha accettato ufficialmente di partecipare al viaggio, per cui alla fine dello stesso (istante di *check-out*) il trasferimento dei cc sull'account del carpooler proprietario corrisponderà al totale. L'operazione di check-out (uscita dalla vettura) è effettuata in automatico dal sistema su tutti i passeggeri quando il carpoo-

ler proprietario termina tecnicamente il viaggio notificandolo al sistema attraverso il dispositivo mobile.

Pooling a tappe. In un sistema *FCRR* implementa un meccanismo che permette ai carpooler passeggeri di intraprendere viaggi da un luogo di partenza A ad un luogo di destinazione B che implichino la necessità di effettuare il pooling a tappe, ovvero ottenendo l'intera tratta desiderata dalla somma di tratte più brevi in sequenza e collegate da un luogo di cambio della vettura di pooling. In questi casi il sistema, dotato di una certa "intelligenza" assiste al meglio il carpooler passeggero grazie ai suoi suggerimenti. In una visione estremamente ottimista, un sistema *FCRR* con un elevato numero di carpooler attivi in una determinata area riesce ad offrire la garanzie offerte dai trasporti pubblici, con il vantaggio di poter accedere a luoghi purtroppo non raggiungibili attraverso i trasporti pubblici.

Capitolo 2

Requisiti ed analisi del *FCRRS*

Questo capitolo descrive dettagliatamente i requisiti di un software che supporti un sistema di Carpooling di tipo ***F**ixed-**C**ost **R**eal-time **R**idesharing* definito precedentemente (Sezione 1.4.4). Successivamente ne fa un'analisi con l'obiettivo di mettere in luce gli aspetti critici e significativi del problema, ponendo quindi le basi per la progettazione contenuta nei capitoli successivi. In particolare, verranno descritti dettagliatamente i requisiti del software, analizzati attraverso i casi d'uso, fornite le descrizioni del modello del dominio del sistema e dei limiti del software ed infine dell'architettura logica. In questa parte del documento vi è un uso frequente di termini specifici ampiamente trattati nei capitoli precedenti, pertanto è consigliato acquisirne una certa familiarità con una attenta lettura a ritroso partendo dalla Sezione 1.4.4.

2.1 Requisiti

Il software di supporto al sistema di Carpooling di tipo *FCRR*, che chiameremo ***F**ixed-**C**ost **R**eal-time **R**idesharing **S**ystem* (*FCRRS*) deve rispondere allo stesso tempo ai requisiti *finanziari* dei sistemi *CCF* e ai requisiti *dinamici* dei sistemi Real-time RideSharing. Inoltre, per completezza, deve realizzare i requisiti dell'aspetto *social* relativi ai sistemi di carpooling in generale.

Accesso al sistema (*Access*). L'accesso alle funzionalità del sistema deve essere garantito sia ai dispositivi fissi (es. *computer desktop*) oltre che ai dispositivi mobile (es. *smartphone*). Il carpooler proprietario della vettura, che d'ora in poi chiameremo *driver*, è obbligato (soltanto a viaggio in corso) ad effettuare l'accesso al sistema da supporto mobile. Le funzionalità caratterizzate da una forte componente dinamica devono essere disponibili e usufruibili, per quanto sia possibile, oltre che dai dispositivi mobile anche dai dispositivi fissi. L'accesso al sistema è consentito previa registrazione e autenticazione del carpooler tramite la coppia e-mail e password.

Gestione account (*AccountManagement*). Il software deve tenere traccia e consentire la modifica delle informazioni (private e non) di ogni **carpooler** mediante un apposito account utente. Oltre ai dati personali dei carpooler (utili per l'aspetto social), un account tiene traccia dell'ammontare dei cc acquisiti e ne permette la modifica tramite operazioni di acquisizione e cessione. L'**acquisizione** dei cc da parte di un carpooler deve poter avvenire in due modalità:

a) Acquisto dal sistema tramite pagamento elettronico[1] al costo di riferimento per l'acquisto (CRA o *buy-cost*) (vedi par. 1.4.2).

b) Acquisizione in seguito ad un trasferimento dal conto di un altro carpooler. Il trasferimento può essere eseguito su iniziativa del sistema (ad esempio contestualmente ad una azione di *check-out*[2]) o su iniziativa del carpooler che desidera effettuare una donazione.

La **cessione** (o vendita) al sistema dei cc acquisiti da parte di un carpooler avviene con la l'apposita operazione di cessione. L'operazione prevede che il sistema accrediti, a fronte della cessione di un quantitativo di cc scelto dal carpooler, il corrispettivo valore in denaro calcolato sul valore di riferimento per la cessione dei cc (CRC o *sell-cost*) (vedi par. 1.4.2).

[1] Con carta di credito o qualsiasi altra modalità di pagamento on-line
[2] Vedi Sezione 1.4.4

Offerta di pooling (*TripOffering*). Il *driver* deve poter sottoporre al sistema un'offerta di pooling che ha come oggetto un viaggio, che d'ora in poi chiameremo **trip**, con la quale specifica: la tratta che intende percorrere; indicativamente data ed ora di inizio del viaggio; data ed ora di arrivo prevista; numero di posti disponibili all'interno della vettura; denominazione della vettura che intende usare. Altre informazioni di carattere sociale sul *driver* sono rese pubbliche dal sistema nella descrizione finale dell'offerta di pooling. In particolare per l'aspetto social, le informazioni sul driver devono (se il driver lo consente) comprendere un riferimento ad un suo profilo su un social network tramite il quale i rider potranno verificare se hanno dei conoscenti in comune, acquisendo in tal caso maggiore fiducia nei confronti del driver. Lo stesso discorso vale ovviamente per i rider poiché quest'ultima è una informazione che riguarda il profilo del carpooler.

Ricerca del pooling (*TripSearching*). Il carpooler che farà da passeggero, che chiameremo *rider*, deve poter effettuare una ricerca tra i *trip* disponibili specificando: il punto geografico da cui vorrebbe essere prelevato (*check-in point*), il punto geografico che ha come destinazione (*check-out point*); eventuale dimensione del suo bagaglio ed indicativamente data ed ora di inizio del viaggio. I parametri da impostare nella ricerca devono consentire una certa flessibilità. In particolare, il check-in point point ed il check-up point, devono poter essere impostati con una tolleranza in chilometri e la data del viaggio con una tolleranza in termini di giorni e settimane. Il software in seguito ad una operazione di filtraggio (*matching*), presenta una lista di *trip* dalla quale il rider ne selezionerà uno di suo interesse. Inoltre, quando necessario, deve supportare il rider nella realizzazione di un pooling a tappe. Il sistema software deve suggerire in modo appropriato quali trip prenotare in sequenza per raggiungere una determinata destinazione.

Prenotazione del posto (*Reservation*). Il software deve realizzare un meccanismo di prenotazione dei posti all'interno della vettura di un determinato *trip* che consenta al rider di garantirsi il

passaggio ed al driver di scegliere i rider che usufruiranno della sua condivisione. In particolare, una volta visualizzato il *trip* di suo interesse, il rider deve poter eseguire una richiesta di prenotazione in riferimento ad esso che ha come prodotto finale una **reservation**, ovvero un posto riservato all'interno della vettura non appena il driver accetta la sua richiesta. Il sistema richiede di assicurare in anticipo la somma di cc per il pagamento del *trip* prima ancora di inoltrare la prenotazione. In questo caso, il sistema si assicura che il rider abbia cc sufficienti per prenotare e dopo aver prelevato la somma di cc dall'account del rider e riposto la stessa a garanzia della prenotazione invia tecnicamente la richiesta di prenotazione al driver. La richiesta di una prenotazione viene notificata al driver attraverso una **notification**, ovvero una notifica a video che lo metta al corrente dell'avvenuta richiesta. Il rider può altresì cancellare una prenotazione effettuata precedentemente, ma nel caso non lo facesse entro 24h prima della partenza del viaggio perderebbe parte della quota cc riservata per il viaggio (il 30%) che va a risarcire il driver. La cancellazione di una prenotazione (accettata o non), viene notificata al driver attraverso una *notification*. A fronte di una richiesta di prenotazione pervenutagli, il driver può decidere di accettarla o di rifiutarla. Nel caso la risposta sia affermativa, il sistema aggiornerà i dati del *trip* diminuendo di una unità il numero dei posti disponibili in vettura, genererà e fornirà al rider un codice valido per il check-in ed invierà una *notification* di accettazione al rider. In caso di rifiuto della richiesta di prenotazione, il rider riceverà una *notification* di rifiuto e la quota di cc posta a garanzia del pagamento del viaggio sarà ritrasferita sull'account del rider. Infine, come modalità di prenotazione avanzata, il sistema deve permettere al rider di effettuare più richieste di prenotazione per la stessa tratta utilizzando la medesima quantità di cc. Al momento dell'accettazione di una di esse, il sistema provvederà a cancellare automaticamente le altre richieste di prenotazione pendenti.

Monitoraggio real-time (*TripMonitoring*) Il software deve consentire ai carpooler il monitoraggio real-time di un *trip* in corso.

Alla partenza, il driver avvia la sessione di viaggio (*trip session*) attraverso il comando *start*. Lo stesso driver, al termine del viaggio, conclude la *trip session* attraverso il comando *stop*.

A *trip session* avviata, i carpooler visualizzano su una mappa digitale la loro posizione corrente. Il driver raggiunge fisicamente la posizione esatta dei suoi rider per prelevarli uno alla volta (picking-up). Quando il driver è in prossimità della posizione del rider, preleva il passeggero contestualmente ad una operazione di check-in sul sistema software. A tal proposito il rider fornirà il suo codice di check-in (*check-in-code*) al driver che concluderà l'operazione di check-in con un pulsante di conferma. Nel caso in cui il driver non trovasse il rider nel punto stabilito, può procedere ad una operazione di risarcimento nei confronti del rider. Per fare ciò gli sarà sufficiente posizionarsi nel punto in cui avrebbe dovuto effettuare il prelievo del rider e azionare il pulsante *amends* per vedersi accreditare dal sistema la quota ccdi risarcimento. Nel caso in cui il driver giunga con ingente ritardo al punto di pick-up non potrà usufruire di un eventuale rimborso. Il rider provvisto del dispositivo mobile potrà durante la sessione di viaggio attiva monitorare senza poter eseguire alcuna speficifica operazione.

Al termine del viaggio, in prossimità del punto geografico di arrivo, attraverso il pulsante *stop*, il driver determinerà la fine della sessione di viaggio ed il sistema effettuerà automaticamente il check-out di tutti i rider ed accrediterà i loro ccriservati al pagamento del viaggio sull'account del driver. Nel caso in cui il viaggio termini prima del previsto, i rider possono cancellare la reservation a sessione di viaggio attiva recuperando il 70% dei cc garantiti per la reservation.

Durante il viaggio, il sistema deve supportare il pooling dinamico (vedi Sezione 1.4.4). In particolare, deve informare in tempo reale il driver richieste di Instant-Pooling con lo stesso meccanismo di prenotazione illustrato precedentemente.

Avvisi istantanei (*Notification*). Il software implementa una funzionalità mediante la quale i carpooler possano essere automaticamente avvisati istantaneamente sullo stato delle loro *reserva-*

tion. In particolare sui cambi di stato di una loro prenotazione o di una loro offerta di pooling. I messaggi di notifica per il driver saranno:

- *hai una nuova richiesta di prenotazione per il viaggio <from> <to> <date-dep> <nome-driver>*
- *la prenotazione di <nome-rider> per il viaggio <from> <to> <date-dep> è stata cancellata*

I messaggi di notifica per il rider saranno:

- *la tua richiesta di prenotazione per il viaggio <from> <to> <date-dep> <nome-driver> è stata accettata*
- *la tua richiesta di prenotazione per il viaggio <from> <to> <date-dep> <nome-driver> è stata rifiutata dal driver per l'eccessiva distanza del punto di pick-up dal percorso*
- *la tua richiesta di prenotazione per il viaggio <from> <to> <date-dep> <nome-driver> è stata rifiutata dal driver per motivi non specificati*

2.1.1 Glossario

Termine	Descrizione
Driver	Carpooler che mette a disposizione il mezzo da condividere durante il viaggio. Normalmente ne è il guidatore, ma potrebbe anche non esserlo.
Rider	Carpooler passeggero che usufruisce della condivisione del mezzo da parte di un driver.
cc	Il cc è l'acronimo di *credito chilometrico*, non è un valore frazionabile, pertanto un ammontare di cc è sempre rappresentato da un numero intero positivo.

Tabella 2.1: Glossario

2.1.2 Casi d'uso

Dall'analisi dei requisiti, emergono con chiarezza le funzionalità che sono messe a disposizione degli utenti del sistema. Riassumiamo graficamente quali sono considerando il sistema stesso come una "black-box" della quale si conosce soltanto l'utilizzo che l'utente può farne in base al proprio ruolo. Di seguito riportiamo i casi d'uso degli attori *driver* (fig. 2.1), *rider* (fig. 2.2), *admin* (fig. 2.3). Anche se non menzionato esplicitamente nei requisiti, includiamo tra gli utilizzatori del sistema un amministratore che possa modulare opportunamente i parametri sui costi.

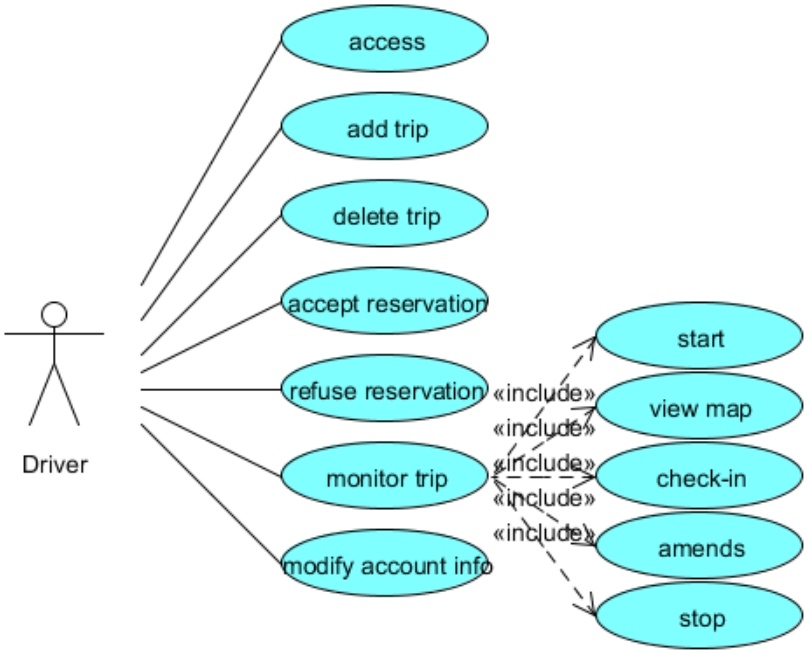

Figura 2.1: Casi d'uso del driver

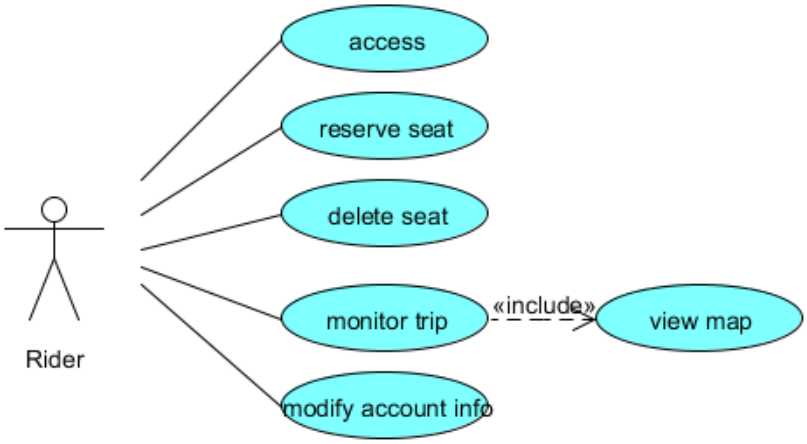

Figura 2.2: Casi d'uso del rider

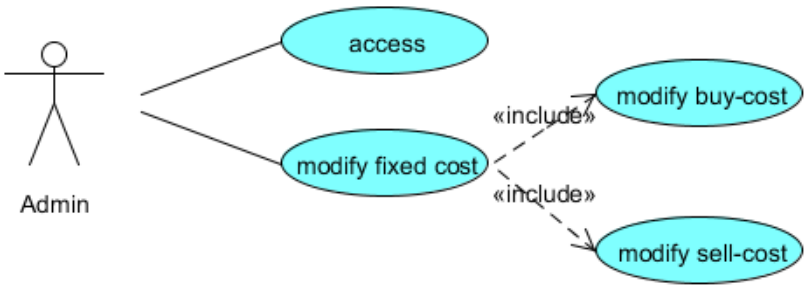

Figura 2.3: Casi d'uso dell'amministratore

2.1.3 Attori esterni al sistema

É utile, in questa fase di analisi dei requisiti, avere una visione globale del sistema software in grado di esplicitare eventuali dipendenze da altri sistemi o servizi esterni. Individuiamo quindi, a questo livello, i sistemi esterni dai quali dipende il nostro sistema *FCRRS* (**F**ixed-**C**ost **R**eal-time **R**idesharing **S**ystem).

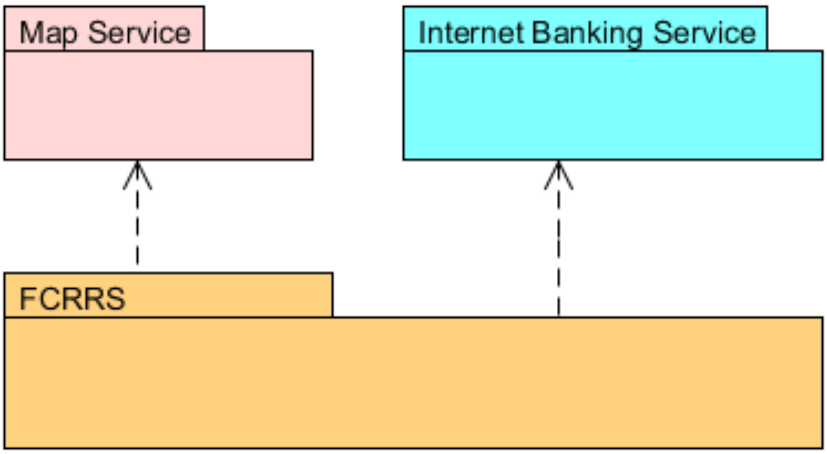

Figura 2.4: Attori esterni al sistema

Il sistema *FCRRS* necessita di due servizi esterni entrambi accessibili attraverso la rete internet:

Map Service che fornisca le mappe digitali e realizzi il calcolo delle distanze tra due punti geografici nonché il calcolo dei percorsi stradali, la ricerca delle coordinate geografiche data la denominazione del luogo (*geocoding*) e la ricerca inversa (*reverse-geocoding*).

Internet Banking Service che realizzi le transazioni bancarie utili all'acquisto ed alla cessione dei cc.

2.2 Analisi del problema

Avendo una chiara trattazione dei requisiti del sistema grazie all'analisi appena conclusa, possiamo passare ad analizzare il problema nella sua totalità considerando in modo dettagliato quali sono i concetti che costituiscono il suo modello del dominio e che rappresentano il prodotto finale del software e dei quali il sistema creerà e manipolerà le istanze. Successivamente saranno esplicitati fin da subito, i vincoli del software di cui questa analisi intende tener conto al fine di restringere il campo alle funzionalità essenziali (*core*) da prendere in esame durante la progettazione del primo prototipo. Successivamente, verrà sviluppata una prima analisi dei task e dei relativi subtask che racchiudono le funzionalità cardine descritte nei requisiti. Infine verrà presentata l'architettura logica globale del sistema, articolata internamente nei sottosistemi FCRR Service, FCRR Desktop Web Application e FCRR Mobile Application.

2.2.1 Modello del dominio

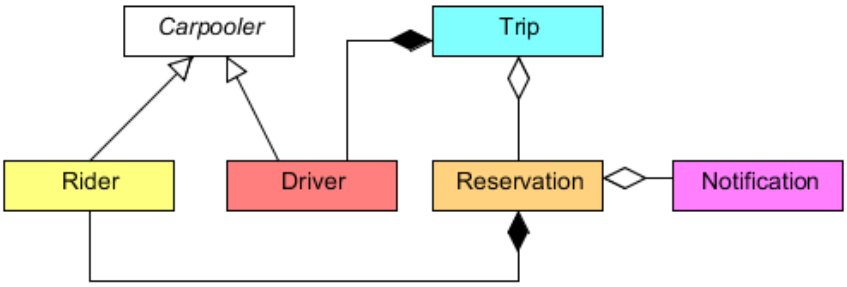

Figura 2.5: Class diagram dominio

Dall'analisi dei requisiti estrapoliamo i concetti che a nostro avviso costituiranno le entità chiave manipolate dal sistema software. In particolare, i termini evidenziati con il colore blu nel testo descrittivo dei requisiti *Carpooler*, *Driver*, *Rider*, *Trip*, *Reservation*, *Notification*

(Sezione 2.1) rappresentano i prodotti finali dell'attività degli utenti attraverso il sistema, pertanto avranno fondamentale importanza nel comprendere come il sistema andrà ad agire nel manipolarli. Inoltre, tali concetti, avranno una loro proiezione digitale all'interno del sottosistema che si occuperà dello storing dei dati. Pertanto una corretta individuazione e definizione delle entità avrà come duplice risultato la modellazione del dominio e una prima struttura implicita della base di dati. La figura 2.5 illustra graficamente attraverso un *class diagram* le relazioni presenti tra i concetti del dominio senza specificarne i valori di cardinalità e rimandando in seguito la definizione degli attributi in forma tabellare. Riteniamo prematuro definire le eventuali operazioni sui concetti rimandandole ad un eventuale passo di analisi successivo. Definiamo quindi i concetti in forma tabellare, elencandone sinteticamente gli attributi e specificando ove opportuno un valore di default dell'attributo con un valore numerico 0.

Concetto	Attributi
Carpooler	
	id identificativo
	name nome del carpooler
	surname cognome del carpooler
	genre sesso del carpooler (0) uomo (1) donna
	geopoint coordinate posizione geografica default
	cc ammontare di cc in suo possesso
	smoke abitudine al fumo (0) non fumatore (1) fumatore (2) indifferente
	phone numero di cellulare
	email indirizzo email
	pw password di accesso
	fb-account url account facebook
	feedback valore di feedback (scala 1 : 10 in 5 stelle)
	status stato dell'account (0) attivo (1) sospeso (dal sistema) (−1) cancellato (dall'utente)

Tabella 2.2: Carpooler

Concetto	**Attributi**
Trip	**id** identificativo **id-driver** identificativo del driver **from** coordinate del luogo di partenza **to** coordinate del luogo di arrivo **vehicle** denominazione del mezzo **total-seats** posti totali messi a disposizione dal driver **taken-seats** posti prenotati **date-dep** data ed ora della partenza **date-arr** data ed ora di arrivo stimate **status** stato del viaggio (−1) cancellato (dal driver) (0) programmato (1) in corso (2) terminato

Tabella 2.3: Trip

Concetto	Attributi
Reservation	
	id identificativo
	id-trip identificativo del viaggio
	id-rider identificativo del rider
	rider-status stato del rider (0) check-in non effettuato (1) checked-in (dopo il pick-up) (2) checked-out (a fine viaggio)
	check-in-code codice di check-in del rider
	pick-up-geopoint coordinate punto di pick-up
	km-covered km totali stimati dal sistema
	cc-reserved n cc a garanzia di pagamento
	luggage-size dimensione del bagaglio del rider (0) non presente (1) piccolo (2) medio (3) grande
	status stato della prenotazione (−1) cancellata dal rider (0) in attesa di risposta (1) accettata (2) rifiutata per motivi non specificati (3) rifiutata per lontananza del punto di pick-up (4) rimborsata al driver del 30%

Tabella 2.4: Reservation

Concetto	**Attributi**
Notification	**id** identificativo **id-res** identificativo della reservation **msg** messaggio testuale di notifica (0) "hai una nuova richiesta di prenotazione per il viaggio <from> <to> <date-dep> <nome-driver>" (1) "la richiesta di prenotazione di <nome-rider> per il viaggio <from> <to> <date-dep> è stata cancellata" (2) "la tua richiesta di prenotazione per il viaggio <from> <to> <date-dep> <nome-driver> è stata accettata" (3) "la tua richiesta di prenotazione per il viaggio <from> <to> <date-dep> <nome-driver> è stata rifiutata dal driver per l'eccessiva distanza del punto di pick-up dal percorso" (4) "la tua richiesta di prenotazione per il viaggio <from> <to> <date-dep> <nome-driver> è stata rifiutata dal driver per motivi non specificati" **date** data e ora di invio della notifica **status** stato della notifica (−1) cancellata (0) non letta (1) letta

Tabella 2.5: Notification

Concetto	Attributi
Fixedcost	
	buy-cost prezzo di acquisto in € del singolo cc
	sell-cost prezzo di vendita in € del singolo cc

Tabella 2.6: Fixed cost

2.2.2 Vincoli del sistema e requisiti aggiuntivi

Nel sistema FCRR esistono alcuni vincoli impliciti che richiedono la definizione di requisiti aggiuntivi per il software. Tali requisiti evidenziano il comportamento del sistema per alcuni scenari particolari. É opportuno esplicitare eventuali scenari non ancora contemplati e definire il comportamento del sistema dove possibile. Inoltre, per ottenere un insieme di requisiti ben definito per il sistema di base, vengono omesse funzionalità considerate avanzate, ma che verranno opportunamente realizzate in estensioni future del software.

+**TripSearch** : per consentire al driver di poter sfruttare il più possibile i posti disponibili bisogna privilegiare le prenotazioni dei rider che intendono percorrere la tratta intera. A tal fine, un viaggio non ancora avviato non sarà incluso nel ricerche in cui è specificato un pick-up point molto distante dal luogo di partenza del viaggio.

+**TripPath** : il sistema dà per scontato che il driver durante il viaggio segua approssimativamente il percorso minimo suggerito dal sistema di tracking. Questo incentiva il driver a percorrere la tratta seguendo il percorso più breve possibile, poiché diversamente potrebbe vedersi rimborsare una quantità di cc minore da quella attesa.

+**Check-out** : i riders che terminano il viaggio abbandonando la vettura prima del raggiungimento della destinazione finale pagano comunque l'intera somma di cc garantiti per il determinato viaggio.

-Pooling a tappe : si ritiene necessario, in questa progettazione, escludere dai requisiti la funzionalità avanzata del pooling a tappe. Poiché non è considerata come funzionalità core, non sarà inclusa nel progetto.

2.2.3 Analisi dei Task e dei Subtask

Passiamo ad analizzare sommariamente l'insieme delle attività principali (***task***) e di quelle secondarie che derivano da ognuna di esse (***subtask***) in modo da avere un quadro completo dei copiti che il software dovrà svolgere. É opportuno discriminare fin da subito i subtask che sono attivati in modo esclusivo dal profilo *driver* o dal profilo *rider* piuttosto che da quello *admin*. Adottiamo una convenzione per differenziare i subtask nel nome usando il formato `X<NomeSubtask>` in cui `X` assumerà come valore "`D`", "`R`" oppure "`A`". Per la creazione dei dati (*data creating*) verrà usata la parola chiave *create*, per il recupero dei dati (*data retrieving*) verrà usata la parola chiave *retrieve*e per l'aggiornamento dei dati (*data updating*) verrà usata la parola chiave *update*.

- **AccessManagement** (***AccessMng***) è il task responsabile della gestione dell'accesso degli utenti al sistema.

  Login è il subtask che gestisce l'accesso al sistema tramite le credenziali di un utente registrato. Prevede l'esecuzione sequenziale delle operazioni di recupero delle credenziali valide (*retrieve*), confronto delle credenziali valide con le credenziali ricevute dall'utente, memorizzazione delle informazioni dell'utente (*create*), visualizzazione all'utente dell'avvenuto accesso.

  Registration è il subtask che gestisce la registrazione di un utente presso il sistema. Prevede l'esecuzione sequenziale delle operazioni di ricezione delle informazioni specificate dall'utente, recupero di un eventuale utente registrato con le stesse informazioni (*retrieve*), quindi del controllo che l'utente non sia già registrato al sistema, memorizzazione delle nuove informazioni (*create*), invio di una email all'indirizzo specificato dall'utente, visualizzazione dell'avvenuta registrazione.

  PasswordRecovery è il subtask che gestisce il recupero delle credenziali di accesso al sistema di un utente. Prevede l'esecuzione sequenziale delle operazioni di ricezione delle informazioni specificate dall'utente, con-

trollo che l'utente sia già registrato al sistema (dopo averne recuperato le informazioni valide *retrieve*), invio di una email all'indirizzo specificato dall'utente contenente le credenziali valide, visualizzazione all'utente dell'avvenuto invio della mail.

Logout è il subtask che gestisce l'uscita di un utente dal sistema. Prevede la cancellazione delle informazioni create a runtime riguardanti l'utente che utilizza il sistema e la visualizzazione all'utente dell'avvenuta uscita dal sistema.

- **AccountManagement** (***AccountMng***) è il task responsabile della gestione della manipolazione delle informazioni contenute nell'account di un utente.

  CCManaging è il subtask che gestisce l'ammontare di crediti chilometrici di un utente. Prevede tre ulteriori subtask, uno per l'acquisto dei crediti (**CCBuying**), uno per la cessione dei crediti (**CCSelling**), uno per il trasferimento dei crediti (**CCTransfering**).

  PersInfoManaging è il subtask che gestisce la manipolazione delle informazioni personali di un utente, contenute nel relativo account. Prevede l'esecuzione sequenziale delle operazioni di ricezione delle informazioni specificate dall'utente, controllo di validità delle stesse, memorizzazione delle informazioni (*update*), visualizzazione dell'avvenuta modifica all'utente.

  FixedCostManaging è il subtask che gestisce la manipolazione delle variabili dei costi fissi usati dal sistema da parte dell'amministratore del sistema. Prevede l'esecuzione sequenziale delle operazioni di ricezione delle informazioni specificate dall'amministratore, controllo di validità delle stesse, memorizzazione delle nuove informazioni (*update*), visualizzazione all'utente dell'avvenuta modifica.

- **Notification** è il task responsabile della gestione delle notifiche ricevute ed inviate dagli utenti.

NotificationReading è il subtask che gestisce la lettura e l'eliminazione delle notifiche ricevute da parte di un utente. Prevede l'esecuzione sequenziale delle operazioni di recupero della lista di notifiche indirizzate all'utente (*retrieve*), visualizzazione della lista mediante l'apposita interfaccia, ricezione della selezione di un elemento della lista, visualizzazione delle info riguardanti la notifica mediante l'apposita interfaccia, eventuale cancellazione della notifica (*update*).

NotificationChecking è il subtask che gestisce la ricezione in tempo-reale delle notifiche riguardanti un trip a cui partecipa un utente. Prevede l'esecuzione **ciclica** delle operazioni di recupero della lista di notifiche non lette indirizzate all'utente (*retrieve*), controllo della dimensione della lista, visualizzazione all'utente di un avviso di notifica ricevuta se la lista recuperata non è vuota.

- **Reservation** è il task responsabile della gestione delle prenotazioni di un trip.

  RReservationRequesting è il subtask che gestisce la richiesta di una prenotazione da parte di un rider per un determinato trip. Prevede l'esecuzione sequenziale delle operazioni di creazione di una nuova reservation per il determinato trip (*create*), creazione della relativa notifica per il driver del trip (*create*), visualizzazione dell'avvenuta richiesta all'utente.

  DReservationResponding è il subtask che gestisce la manipolazione di una richiesta di prenotazione da parte del driver. In particolare un driver può accettare o rifiutare una richiesta di prenotazione per un suo trip. Prevede l'esecuzione sequenziale delle operazioni di aggiornamento della reservation con cambiamento di stato (*update*), creazione della relativa notifica per il rider (*create*), visualizzazione dell'avvenuta operazione all'utente.

- **TripManagement** (***TripMng***) è il task responsabile della gestione dei trip, in particolare l'offerta e la ricerca dei trip esistenti.

  RoleSelecting è il subtask che gestisce l'accesso dell'utente alle funzionalità disponibili per il suo profilo di driver o per il suo profilo di rider. Prevede la sola visualizzazione dell'interfaccia per la gestione del profilo driver o rider.

  DTripOffering è il subtask che gestisce la creazione di una offerta di trip da parte di un driver. Prevede l'esecuzione sequenziale delle operazioni di creazione di una nuovo trip (*create*), visualizzazione dell'avvenuta creazione tramite la lista aggiornata di tutti i trip creati dal driver.

  RTripSearching è il subtask che gestisce la ricerca di un trip da parte di un rider specificando una tratta ed una data. Prevede l'esecuzione sequenziale delle operazioni ricezione delle informazioni per il filtraggio, recupero della lista dei trip corrispondenti alle specifiche (*retrieve*), visualizzazione all'utente della contenente della lista dei trip con le determinate caratteristiche volute.

  DTripListManaging è il subtask che gestisce la visualizzazione della lista dei trip creati dall'utente (driver). Prevede l'esecuzione sequenziale delle operazioni di recupero della lista dei trip (*retrieve*), visualizzazione a video della lista contenente i trip creati dal driver.

  RResListManaging è il subtask che gestisce la visualizzazione della lista delle reservation dell'utente (rider). Prevede l'esecuzione sequenziale delle operazioni di recupero della lista delle reservation (*retrieve*), visualizzazione a video della lista contenente le reservation riguardanti il rider.

- **TripMonitoring** (***TripMnt***) è il task responsabile della gestione e del monitoraggio di un trip in corso.

DTripWatching è il subtask che gestisce il monitoraggio di un trip in corso da parte del driver, in particolare visualizza a video in tempo reale le posizioni dei rider e la propria posizione sulla mappa digitale. Prevede l'esecuzione **ciclica** delle operazioni di lettura delle coordinate gps correnti del dispositivo mobile, memorizzazione delle stesse con aggiornamento della posizione del driver (*update*), recupero delle posizioni dei rider coinvolti nel trip (*retrieve*), visualizzazione a video della mappa digitale contenente le posizioni aggiornate.

DCheckingIn è il subtask che gestisce, durante un trip in corso, il check-in di un rider da parte del driver. Prevede l'esecuzione sequenziale delle operazioni di ricezione delle informazioni sul rider (compreso il codice per il check-in), recupero (*retrieve*) e controllo che la posizione geografica del driver sia sufficientemente vicina alla posizione geografica del rider, recupero del codice di check-in valido (*retrieve*), controllo di validità del codice di check-in comunicato dal rider, aggiornamento dello stato del rider (per la reservation) (*update*), visualizzazione a video dell'avvenuto check-in.

DAmending è il subtask che gestisce l'operazione di amends (rimborso della prenotazione di un rider) da parte di un driver e nei confronti di un rider per il trip in corso. Prevede l'esecuzione sequenziale delle operazioni di recupero (*retrieve*) e controllo che la posizione geografica del driver sia sufficientemente vicina a quella prevista per il pick-up del rider e non sia troppo in ritardo rispetto all'ora prevista per la partenza, aggiornamento dello stato del rider (per la reservation) e il trasferimento della prevista per il rimborso (*update*), visualizzazione dell'avvenuta operazione amends.

RTripWatching è il subtask che gestisce il monitoraggio di un trip in corso da parte del rider. In particolare visualizza a video in tempo reale le posizioni del driver e dei rider sulla mappa digitale. Prevede l'esecuzione **ciclica** delle operazioni di recupero delle posizioni dei ri-

der coinvolti nel trip (*retrieve*), visualizzazione a video della mappa digitale contenente le posizioni aggiornate.

2.3 Architettura logica

Sottosistemi individuati

Il sistema *FCRRS* nel suo complesso sarà composto da tre sottosistemi. Un sottosistema che, vista l'analisi dei task, dovrà realizzare le funzionalità per lo storing dei dati; un sottosistema che realizzerà le funzionalità necessarie per l'applicazione che accede dal dispositivo mobile; un sottosistema che realizzerà le funzionalità necessarie per l'applicazione che accede da un computer desktop o laptop. Il sottosistema per lo storing, chiamato *Service*, sarà un servizio reso disponibile agli altri due sottosistemi attraverso la rete internet. Pertanto, i due sottosistemi ***F****CRR* ***M****obile* ***A****pplication* (*FMA*) e ***F****CRR* ***D****esktop* ***W****eb* ***A****pplication* (*FDWA*) avranno una dipendenza diretta dal sottosistema Service. In conseguenza di questo, il sottosistema Service rappresenta un punto critico del sistema generale, per questo richiederà alcuni requisiti non funzionali che mirano ad esplicitare determinate caratteristiche di affidabilità necessarie per il sottosistema Service.

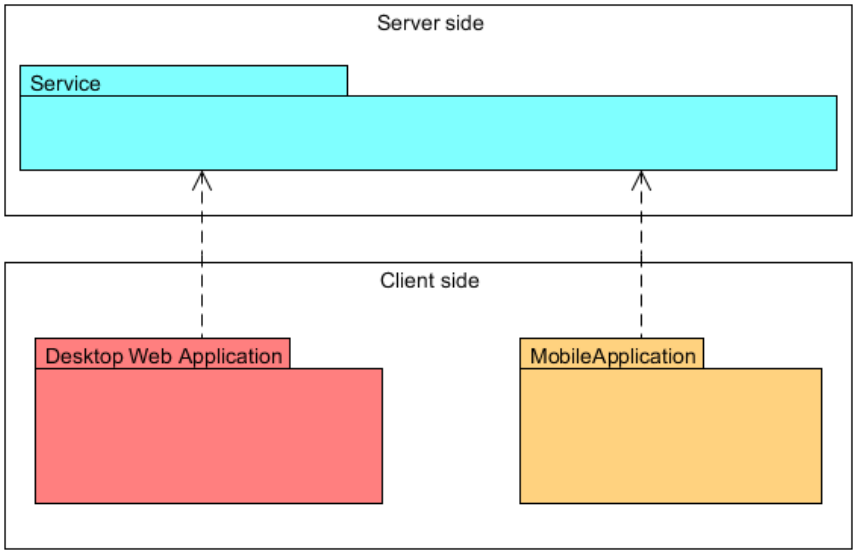

Figura 2.6: Architettura logica del sistema (Layer 1)

2.3.1 FCRR Service (*FS*)

***F**CRR **S**ervice* (*FS*) è il sottosistema che nell'architettura generale risiede lato server e che andrà ad offrire ai client l'insieme delle funzioni tipiche di un *persistent storage system*, ovvero di un sistema che permette di immagazzinare le informazioni in una memoria non volatile (*non-volatile storage*). Le sue funzioni principali operazioni per la manipolazione dei dati sono: *create* per la creazione, *retrieve* per la lettura, *update* per la modifica e *destroy* per la cancellazione. Le funzionalità offerte da *FS* corrispondono alle prime tre operazioni sui dati: *data creating*, *data retrieving*, *data updating*. Trascuriamo in questa trattazione la funzionalità di *data destroying* poiché nel sistema *FCRR* non è prevista esplicitamente la cancellazione dei dati ma è ottenuta implicitamente attraverso una operazione di update che va a modificare lo "status" del particolare dato. *FS* avrà quindi una sua interfaccia d'uso basata sull'insieme delle tre operazioni **create**, **retrieve**, **update**, associate alle entità del dominio: *Carpooler*, *Notification*, *Reservation*, *Trip*, *Fixed-cost*.

Requisiti non funzionali

Scalabilità. Il sottosistema *FS* può essere soggetto ad un rapido aumento del numero dei client che usufruiscono in contemporanea dei servizi offerti, pertanto è fondamentale che l'aumento del numero delle richieste non vada a compromettere la stabilità del servizio stesso o ad imporre una modifica al software del servizio. Il carico aggiuntivo di richieste deve poter essere fronteggiato con la sola aggiunta di risorse hardware.

Affidabilità Il sottosistema *FS* deve essere affidabile e poter mantenere i propri dati anche in caso di guasti dovuti alla fornitura di energia, usura dell'hardware, attacchi informatici. A tal proposito *FS* deve realizzare un meccanismo di replicazione dei dati che possa evitarne la perdita anche parziale.

Prestazioni La natura del sistema *FCRRS* impone al sottosistema *FS* un certo livello di performance al di sotto del quale il funzionamento dei client sarebbe compromesso. In particolare, il servizio, indipendentemente dal numero di richieste contemporanee,

deve poter mantenere comunque un tempo di risposta inferiore alla decina di secondi poiché è prevista una frequenza massima di aggiornamento della mappa digitale (task **TripMonitoring**) di poco superiore ai 10 secondi. Con tempi di risposta superiori, l'utente percepirebbe il sistema come bloccato non avendo un riscontro esplicito dell'elaborazione in corso.

Sicurezza La comunicazione tra i client e *FS*, oltre ad essere affidabile, deve garantire un certo grado di sicurezza pur mantenendo i requisiti prestazionali descritti sopra. Notevole importanza assume l'autenticazione dei client presso il servizio. Un meccanismo di autenticazione tra client e servizio escluderebbe la possibilità che un client non legittimo possa effettuare delle richieste modificando in modo improprio i dati immagazzinati lato server.

Requisiti funzionali

É opportuno definire con chiarezza quali sono i requisiti funzionali del sottosistema *FS* poiché costituiscono il punto di partenza per la futura progettazione del sistema stesso che risiede sul server e della modalità di interazione con l'applicazione mobile lato client. Come anticipato precedentemente, le funzioni principali del sottosistema *FS* sono:

- *data creating*
- *data retrieving*
- *data updating*

Di seguito riportiamo in maniera astratta l'insieme delle richieste che *FS* deve soddisfare mappando le stesse sui task che ne fanno uso. La tabella 2.7 fa uso della seguente simbologia:

- `inputParams`: lista di attributi con associato il relativo valore utili alla creazione di una nuova istanza del relativo concetto
- `<Concetto>List`: lista di istanze di tipo `<Concetto>` restituite dall'operazione

- `selectParams`: lista di attributi con associato il relativo valore utili alla selezione di un insieme di istanze esistenti
- `updateParams`: lista di attributi con associato il relativo valore utili all'aggiornamento di una istanza esistente
- `<Concetto>.<attr>`: valore dell'attributo `<attr>` associato all'istanza del relativo `<Concetto>`

Operation	**Input,*Output***	**Task**
create **Carpooler**	`inputParams`, *`Carpooler.id`*	AccessMng
retrieve **Carpooler**	`selectParams`, *`CarpoolerList`*	AccessMng, AccountMng, Reservation, TripMnt
update **Carpooler**	`selectParams` `updateParams`, -	AccountMng, Reservation, TripMnt
create **Trip**	`inputParams`, *`Trip.id`*	TripMng
retrieve **Trip**	`selectParams`, *`TripList`*	TripMng, TripMnt
update **Trip**	`selectParams` `updateParams`, -	TripMnt
create **Reservation**	`inputParams`, *`Reservation.id`*	Reservation
retrieve **Reservation**	`selectParams`, *`ReservationList`*	TripMng, TripMnt
update **Reservation**	`selectParams` `updateParams`, -	Reservation, TripMnt
create **Notification**	`inputParams`, *`Notification.id`*	Reservation
retrieve **Notification**	`selectParams`, *`NotificationList`*	Notification
update **Notification**	`selectParams` `updateParams`, -	Notification
retrieve **Fixedcost**	`selectParams`, *`FixedcostList`*	AccountMng
update **Fixedcost**	`selectParams` `updateParams`, -	AccountMng

Tabella 2.7: Requisiti funzionali del FCRR Service

2.3.2 FCRR Desktop Web Application (*FDWA*)

L'accesso al sistema *FCRR* da parte dei dispositivi desktop avviene attraverso la *FCRR Desktop Web Application* (*FDWA*). Si accede alla *FDWA* da un dispositivo desktop con accesso ad internet e su cui è installato un comune web browser. Come evidenziato nei requisiti, gli utenti che accedono da client desktop devono poter avere accesso alle stesse funzionalità dei client mobile, eccezion fatta per quelle funzionalità per cui è necessario che il dispositivo sia dotato del sensore GPS per rilevare della posizione corrente. In particolare, sotto il profilo driver, l'accesso al sistema dalla *FDWA* non consente l'avvio di un trip precedentemente programmato e quindi la possibilità di monitorare il trip. Il software *FCRRS*, consente al driver, dopo l'ora del termine prevista per il trip programmato, di chiudere il trip. A fronte della chiusura del trip, il software completa il trasferimento di cc dai rider al driver secondo le informazioni definite nelle reservation del trip stesso. Sotto il profilo rider, il software consente entro l'ora prevista per la fine del trip, la cancellazione della prenotazione, come da requisiti generali. In conclusione, un driver che accede al sistema soltanto dalla *FDWA* non ha la possibilità di monitorare il trip attraverso la mappa (***TripMonitoring***) e richiedere un eventuale risarcimento (*amends*). Pertanto i requisiti della *FDWA* sono un sottoinsieme dei requisiti generali descritti nella Sezione 2.1.

2.3.3 FCRR Mobile Application (*FMA*)

L'accesso al sistema *FCRRS*da parte dei dispositivi mobile avviene attraverso un'apposita applicazione, la *FCRR Mobile Application* (*FMA*). Essendo installata su dispositivi mobile (*smartphone*) dotati del sensore di posizione GPS (*Global Positioning System*), la *FMA* implementa in modo completo tutti i requisiti del client per il sistema *FCRRS*, includendo quindi il **TripMonitoring** e la funzionalità per il risarcimento istantaneo *amends*. Inoltre consente ai rider di affettuare ricerche di instant pooling e di cancellare una prenotazione per forfait del relativo driver. Focalizzeremo d'ora in poi l'attenzione sui sottosistemi *FS* (lato server) e FCRR Mobile Application (lato client) per la progettazione del prototipo del sistema *FCRRS*.

Capitolo 3

Progettazione del FCRRS

Questo capitolo descrive la progettazione del sistema ***F**ixed-**C**ost **R**eal-time **R**idesharing* ponendo il focus sul sottosistema *FMA* riguardante l'applicazione mobile. Basandosi sulle informazioni prodotte dall'analisi sviluppata nel capitolo precedente viene illustrata l'architettura generale (Sezione 3.1) per poi passare ad una breve trattazione del ***F**CRR **S**ervice* ispirato ai principi REST per giungere alla progettazione della FCRR Mobile Application. In quest'ultima parte verranno preventivamente motivate le ragioni dell'approccio agent-oriented, descritto il modello A&A adottato e la tecnologia mobile JaCa-Android, dopodiché verrà presentata l'architettura del sistema multi-agente della FCRR Mobile Application.

3.1 Architettura generale

Nei requisiti di accessibilità al sistema è esplicitamente richiesta la possibilità di accedere alla piattaforma attravero dispositivi sia mobile che desktop per mezzo della rete internet. Il sistema nel suo complesso è un sistema distribuito con architettura ***client-server*** in cui i *client*, rappresentati dai dispositivi desktop e mobile, quindi dai sottosistemi *FDWA* e *FMA*, hanno accesso ed usufruiscono di un servizio comune, rappresentato dal *FCRR REST Service* (lato *server*), che realizza una specifica funzione all'interno del sistema software *FCRRS*. L'architettura generale del *FCRRS* è articolata nel modo illustrato in figura 3.1.

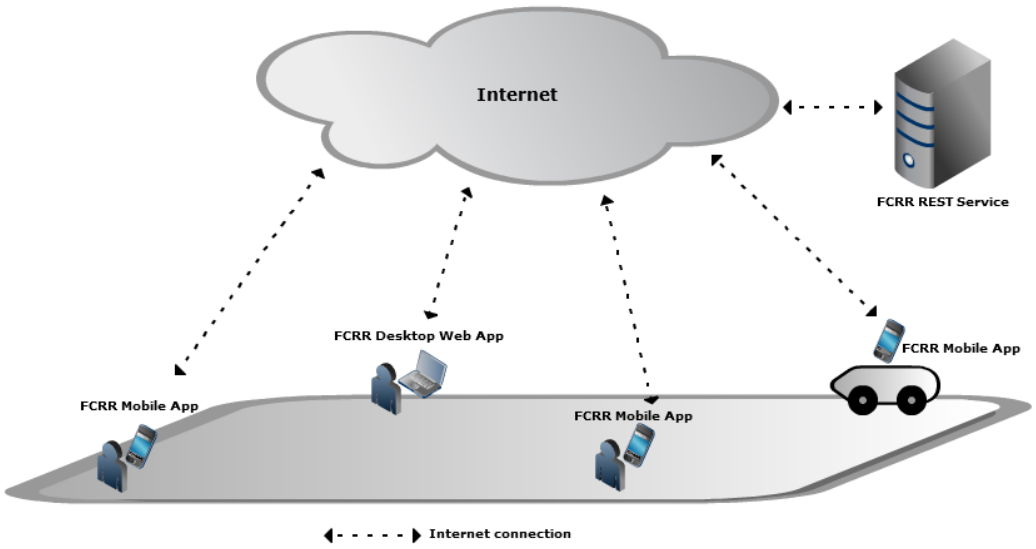

Figura 3.1: Architettura generale del *FCRRS*

3.2 FCRR REST Service

Visti ed analizzati i requisiti funzionali e non funzionali del servizio (contenuti nella Sezione 2.3.1), la progettazione dello stesso implica delle scelte in un primo momento architetturali ed in un secondo momento tecnologiche, mirate a soddisfarne non solo i requisiti esplicitati, ma a tenere conto dei tempi necessari per lo sviluppo, quindi della reale fattibilità. Inoltre, un aspetto importante riguarda la contestualizzazione del servizio all'interno del sistema *FCRRS*. Pertanto, è necessario tenere in considerazione che il servizio stesso, in virtù dell'architettura di rete, risiederà su un nodo server e sarà interrogato sia da una applicazione mobile (*FMA*) che da una desktop (*FDWA*) con le conseguenze che ciò implica. Una analisi delle varie opzioni porta a considerare come scelta privilegiata, nel nostro caso specifico, la realizzazione di un Web Service leggero ispirato ai principi *REST*[7], una soluzione molto diffusa in ambito mobile. Le ragioni della scelta verranno esplicitate nella Sezione 3.2.2, quindi dopo aver dato una descrizione generale di cos'è un servizio *REST* (Sezione 3.2.1).

3.2.1 REpresentational State Transfer (*REST*)

Il ***RE****presentational* ***S****tate* ***T****ransfer*[7] (*REST*) è una particolare di ***W****eb* ***S****ervice* (*WS*), talvolta definito anche come "web service leggero" (*Lightweight Web Service*) e realizzato seguendo un insieme di principi (o linee guida) che portano a delle precise scelte di carattere implementativo.

Sviluppare un servizio *REST* significa seguire un preciso insieme di principi technology-independent, fatta eccezione per il protocollo utilizzato per la comunicazione: **HTTP** (*Hyper Text Transfer Protocol*). I principi basilari[5] di un *REST* sono:

- Architettura client/server;
- Comunicazioni stateless;
- Ogni componente del sistema è identificato da un *URL* (*Uniform Resource Locator*);
- Tutti i componenti del sistema comunicano attraverso interfacce con metodi ben definiti;
- L'Architettura è a layer e i dati possono essere memorizzati (cached) in ogni layer.

Un sistema conforme ai cinque principi *REST* è detto *RESTful*. L'adozione del protocollo *HTTP*, che prevede per sua natura l'impiego di una architettura client-server, l'uso degli URL e dei metodi GET e POST (ma talvolta anche dei restanti verbi HTTP), la caratteristica stateless della comunicazione (evitando l'uso di cookie e sessioni), fanno del REST la soluzione più semplice per la realizzazione di un servizio web che sia scalabile. Dal punto di vista della sicurezza, il servizio *REST* può contare sull'affidabilità di una comunicazione sicura basata sul protocollo *HTTPS* (***H****yper****T****ext* ***T****ransfer* ***P****rotocol over* ***S****ecure Socket Layer*) e grazie alla sua architettura stratificata (derivante dall'ultimo principio) sull'impiego del tutto "indolore" delle tecnologie di rete esistenti ed applicate comunemente nel mondo web come *firewall* e *load-balancers* (per la distribuzione del carico delle richieste di un servizio su più server).

3.2.2 Motivazioni della scelta

La semplità d'uso di un servizio REST è la caratteristica principale per cui questa tipologia di Web Services sta avendo successo in ambito mobile, ma la scelta di questo tipo di servizio è guidata principalmente dalla corrispondenza tra i requisiti richiesti e le proprietà di cui gode. I requisiti funzionali del servizio descritti nella tabella 2.7 portano a considerare il sottosistema *FS* come un servizio la cui funzione principale è quella di offrire ai client che lo interrogano la possibilità di recuperare e manipolare una certa quantità di dati organizzati secondo una struttura che fa riferimento ai concetti che compongono il modello del dominio trattato nella Sezione 2.2.1. La stessa interfaccia astratta in tabella 2.7 evidenzia come per i client del sistema le uniche informazioni utili al fine di poter usufruire correttamente del servizio debbano interessare esclusivamente il nome e la struttura, quindi le proprietà, dei concetti definiti (*Carpooler*, *Driver*, *Rider*, *Trip*, *Reservation*, *Notification*). Questo requisito funzionale si sposa perfettamente con il principio **stateless** dei servizi *REST* in quanto ogni singola richiesta dei client nel nostro caso ha storia a sé e non dipende dalle precedenti richieste trattandosi di semplice creazione ed aggiornamento di dati. Inoltre, è perfettamente in linea con il principio secondo il quale "**i componenti del sistema comunicano attraverso interfacce con metodi ben definiti**", il che è esattamente quanto richiesto dai requisiti. Di fatti il servizio REST risiederà sul nodo server e sarà interrogato da un insieme di client che sanno perfettamente come interfacciarsi con esso. L'architettura **client/server** del servizio REST aderisce perfettamente all'architettura di rete illustrata in figura 3.1. Dal punto di vista di un client, la richiesta dell'esecuzione di una singola operazione del servizio, sarà creata indicando attraverso una *request HTTP* di tipo *GET* (per la lettura) o *POST* (per la creazione o l'aggiornamento), un *URL* identificativo del concetto oggetto dell'operazione, una struttura dati (di solito JSON o XML) contenente i valori degli attributi necessari all'esecuzione dell'operazione.

In riferimento ai requisiti non funzionali richiesti come **scalabilità** e **prestazioni**, il servizio REST può contare sulle soluzioni adottate per la maggior parte dei classici Web Services, inoltre dal punto di vista della **sicurezza** gode del privilegio di poter affidarsi in modo

del tutto naturale a protocolli come l'*HTTPS* ed alle tecnologie di rete esistenti nel mondo del web come i *firewall* ed a *load-balancers* in tema di **affidabilità**, dando quindi la possibilità di aumentare e diminuire in maniera dinamica la capacità computazionale richiesta mantenendo invariata l'interfaccia del servizio.

3.3 FCRR Mobile Application

L'analisi del problema (Sezione 2.2.3) presenta l'insieme delle attività che dovranno essere svolte dal sistema software esplicitandole in termini di task e subtask. Il passaggio dai task alla progettazione delle parti del sistema che lo realizzano non è in genere una operazione semplice. Un sistema distribuito, presenta spesso un livello di complessità e di dinamicità non semplice da modellare con le astrazioni fornite dal paradigma ad oggetti nonostante quest'ultimo non sia di basso livello. Tra i task che definiscono le funzionalità del sistema ad alto livello ed il paradigma ad oggetti esiste un gap concettuale che può essere agevolmente colmato dall'adozione di un paradigma di alto livello come quello agent-oriented. Pertanto si ritiene opportuno proseguire con la progettazione del *FCRRS* adottando una metodologia di sviluppo che possa rendere agevole il passaggio da una analisi task-oriented alla progettazione vera e propria del sistema *FMA*. La scelta della metodologia di sviluppo di alto livello ricade sul paradigma ad agenti, in particolare sul modello A&A[11, 14, 15] e sulla tecnologia Ja-Ca. Essendo il sistema *FMA* costituito da una "mobile application" è necessario che la tecnologia Ja-Ca sia disponibile per almeno una delle piattaforme mobile di ultima generazione. Ciò è reso possibile dal middleware JaCa-Android[16] che permette l'uso della tecnologia Ja-Ca sulla piattaforma Android. É opportuno quindi illustrare sinteticamente la tecnologia JaCa-Android ed il modello ad agenti su cui si basa prima di proseguire nella progettazione (Sezione 3.3.1).

3.3.1 Il modello A&A e JaCa-Android

JaCa-Android[16] è una piattaforma agent-oriented che fornisce un alto livello di astrazione per poter progettare, implementare ed eseguire applicazioni mobile "smart" su Android. JaCa-Android adotta per l'astra-

zione di agente il modello BDI(*Belief-Desire-Intention*)[13] ovvero la nozione "strong" di agente tipica del campo dell'intelligenza artificiale. Una applicazione in JaCa-Android va in esecuzione sull'infrastruttura Ja-Ca arricchita da un livello che permette l'accesso alle risorse del framework di Android. Essendo sviluppata in Java, la stessa piattaforma Ja-Ca viene eseguita come una applicazione di Android.

Modello A&A di Ja-Ca

Il modello concettuale su cui è basato Ja-Ca è A&A[11, 14, 15] (Agents and Artifacts). A&A prevede un insieme dinamico di entità attive e dinamiche, gli agenti (*agents*) che interagiscono all'interno di un ambiente comune (*environment*). L'agente, astrazione principale nel modello A&A, è una entità autonoma che incapsula un suo flusso di controllo, ha una sua business logic ed ha un comportamento sia reattivo che pro-attivo. L'ambiente è composto da uno o più *workspaces* ognuno dei quali contiene delle entità passive chiamate artefatti (*artifacts*). Un artefatto incapsula ed offre agli agenti tutte le funzionalità e i servizi necessari e per questo è visto dagli agenti come una risorsa da usare, creare, condividere, osservare, durante le loro attività. I *workspace* definiscono la topologia dell'applicazione ed ognuno di essi può essere situato su un nodo diverso (processo) e per questo possono quindi essere distribuiti su una rete.

Integrazione su Android

Gran parte della piattaforma JaCa-Android (figura 3.2) è costituita da un insieme di artefatti che incapsulano le principali caratteristiche fornite dal sistema Android. Il workspace condiviso che permette agli agenti di sfruttare gli artefatti di cui sopra è chiamato `JaCa-Services`. Il workspace `JaCa-Services` viene lanciato allo startup di Android ed è condiviso tra tutte le applicazioni JaCa-Android.

Gli artefatti all'interno di `JaCa-Services` sono istanziati secondo il pattern *singleton*, poiché ne è necessaria una sola istanza da convidere eventualmente tra più applicazioni. Da qui, un elenco degli artefatti principali contenuti in `JaCa-Services`:

- `SMSService`

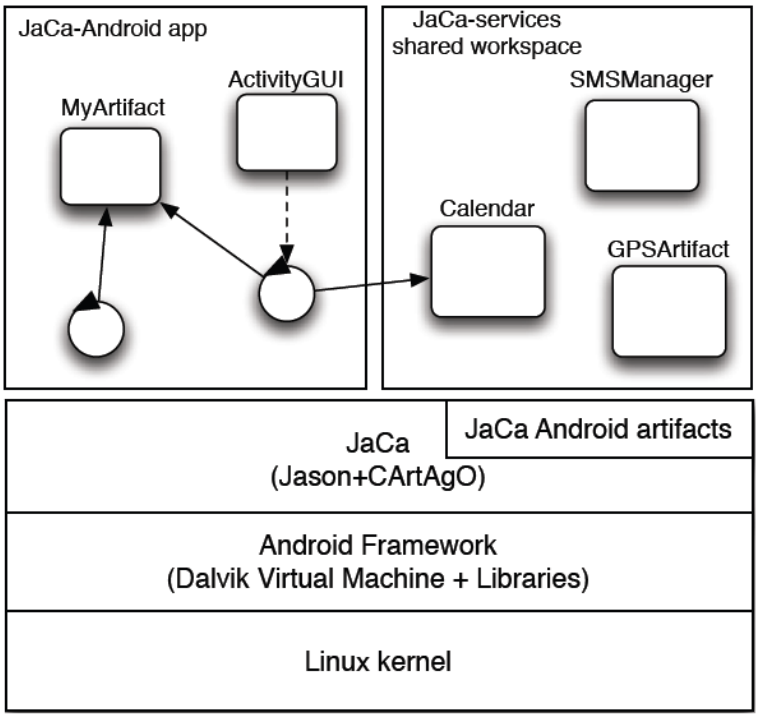

Figura 3.2: Piattaforma JaCa-Android

- `MailService`
- `GPSService`
- `AccelerometerSensorService`
- `GyroscopeSensorService`
- `CallServices`
- `ConnectivityService`
- `CalendarService` (per usare Google Calendar)
- `PhoneSettingsService` (per la suoneria e la vibrazione)

Per completezza e compatibilità con Android, `JaCa-Services` include anche i seguenti artefatti:

- `BroadcastReceiverArtifact`
- `GUIArtifact`
- `ContentProviderArtifact`
- `ServiceArtifact`

E' bene notare che, grazie a JaCa-Android, qualunque applicazione Ja-Ca può essere messa direttamente in esecuzione su un sistema operativo Android.

Framework Ja-Ca: Jason + CArtAgO

Ja-Ca è il modello di programmazione di JaCa-Android ed esso è il risultato dell'integrazione di due tecnologie differenti con modelli di programmazione differenti: Jason e CArtAgO.

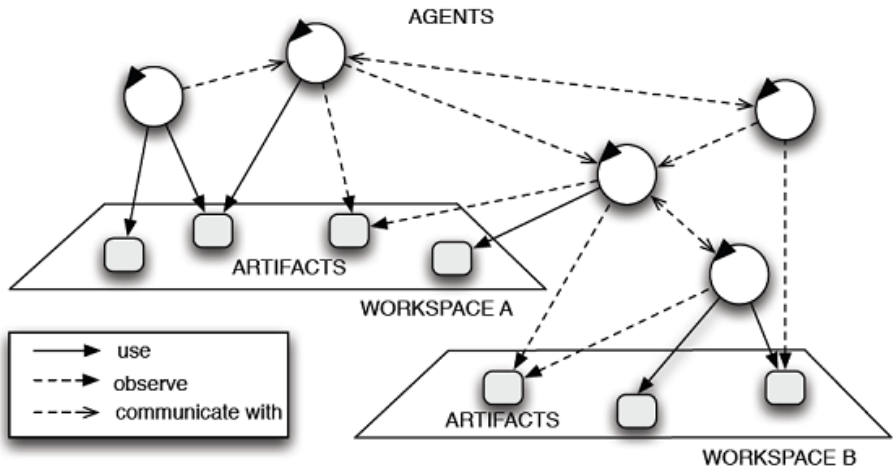

Figura 3.3: Applicazione nel modello di programmazione Ja-Ca

Jason[3, 4] è la tecnologia che permette di implementare e mettere in esecuzione gli agenti. Il modello computazionale e l'architettura adottati in Jason fanno riferimento al modello BDI di agente e al linguaggio AgentSpeak[12]. Un agente in Jason è visto come un "sistema di pianificazione reattivo", ovvero come una entità con un comportamento pro-attivo. Esso è in continua esecuzione reagendo agli stimoli

esterni (chiamati *events*) ed attuando opportunamente i diversi piani definiti dal progettista. Un piano è composto da una lista di azioni che l'agente esegue per raggiungere il proprio obiettivo, o "stato desiderato", definito come *goal*. I costrutti del linguaggio si dividono in tre macrocategorie attraverso i cui valori iniziali un agente viene definito:

- ⋆ **Beliefs**. Rappresentano la conoscenza dell'agente (sia esterna che interna) e vengono immagazzinati nella *belief base* dell'agente stesso. Vengono rappresentati tramite letterali (simili ai predicati Prolog) e possono essere riferiti allo stato interno dell'agente o allo stato osservabile degli artefatti che osserva l'agente stesso. I Beliefs possono essere aggiunti e rimossi a runtime e grazie all'uso di regole Prolog (consentite da Jason) l'agente può inferirne di nuovi semplificando notevolmente la risoluzione di determinati task.

- ⋆ **Goals**. Rappresentano il target assegnato all'agente, il quale utilizza i piani (*plans*) come azione concreta per raggiungere l'obiettivo. Essi sono introdotti dopo l'insieme dei beliefs ed in Jason vengono rappresentati tramite formule atomiche Prolog precedute da un punto escalamativo ('!'). Esiste la possibilità di sottoporre goals a runtime inviando all'agente un messaggio del tipo *achieve-goal message*.

- ⋆ **Plans**. L'insieme dei plans all'interno di un'agente costituisce le azioni che l'agente può compiere per raggiungere un goal ed in Jason vengono rappresentate attraverso regole del tipo: `Event : Context <- Body`. Per `Event` si intende l'evento che nel caso si verifichi andrà a far "scattare" il piano. `Event` può essere di diversi tipi a seconda dei casi, ovvero può trattarsi di un nuovo goal o di una percezione dall'ambiente. `Context` è l'espressione booleana che definisce le condizioni sotto le quali il piano può essere eseguito. Il `Body` descrive l'elenco delle azioni (interne o esterne) da compiere per eseguire il piano.

CArtAgO[14] è il framework per implementare gli ambienti (*environments*) e gli artefatti (*artifacts*) al loro interno. Il modello degli artefatti in CArtAgO (figura 3.4) segue il meta-modello A&A, quindi un artefatto è un'entità passiva che incapsula delle funzionalità

che possono essere usate dagli agenti attraverso una interfaccia d'uso composta da operazioni (*operations*) e proprietà osservabili (*observable properties*).

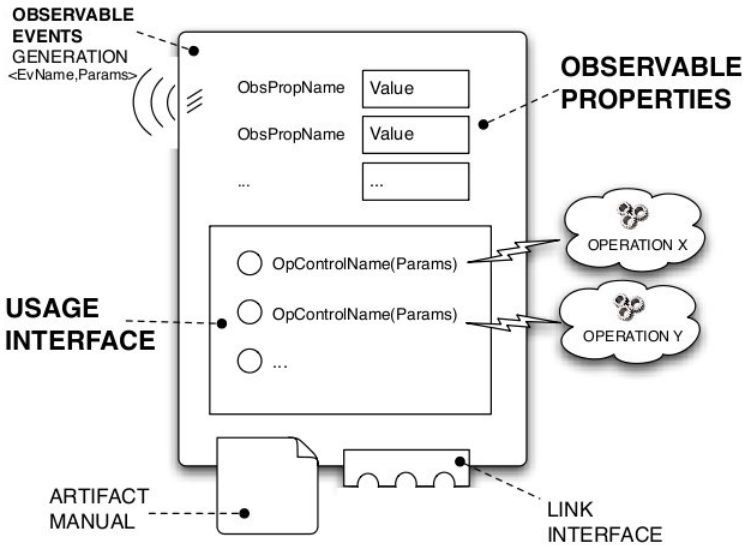

Figura 3.4: Artefatto CArtAgO (modello A&A)

Le operazioni sono delle azioni che gli agenti possono compiere tramite l'artefatto per "modificare" una risorsa/artefatto presente nell'ambiente in cui risiede l'artefatto stesso.

Le proprietà osservabili sono l'insieme delle informazioni che definiscono lo stato dell'artefatto e che sono visibili all'esterno e percepite dagli agenti. Operativamente, definire un artefatto in generale in CArtAgO, significa estendere la classe base `Artifact`. Per implementare un artefatto specifico, ad esempio quello relativo ad una GUI è sufficiente estendere la classe `GUIArtifact` che estende di suo la classe `Artifact`. Dopo aver implementato il metodo `init` che definisce lo stato iniziale dell'artefatto, possiamo implementare le operazioni 'esterne' ed 'interne' usando come preambolo del metodo le annotazioni Java `@OPERATION` e `@INTERNAL_OPERATION`.

Il flusso di esecuzione delle operazioni all'interno dell'artefatto è indipendente dal flusso di esecuzione interno all'agente che lo usa, tuttavia il piano in esecuzione nell'agente può sospendersi autonomamente

in attesa della conclusione dell'operazione. In ogni caso, l'agente può evitare di sospendere completamente il suo flusso di controllo continuando nella percezione degli eventi esterni e nel compiere altre azioni rimanendo reattivo. L'esecuzione di una operazione in un artefatto è mutualmente esclusiva, così da permetterne l'uso concorrente a più agenti. Le operazioni sono realizzate come transazioni, ovvero se una operazione non va a buon fine, viene effettuato il roll back per riportare lo stato dell'artefatto nelle condizioni iniziali.

Le proprietà osservabili sono definite da tuple etichettate attraverso la primitiva `defineObsProperty('Etichetta', 'Valore')`, recuperate attraverso `getObsProperty('Etichetta')` e modificate attraverso `obj.updateObsProperty('NuovoValore')` dove `obj` è l'oggetto di tipo `ObsProperty` restituito dalla primitiva `getObsProperty`. Un artefatto ha la possibilità di notificare un evento agli agenti che lo osservano usando la primitiva `signal` nella quale specifica il tipo di evento e i parametri. Una proprietà aggiuntiva degli artefatti è la possibilità di linkarli insieme in modo da permettere ad un agente di eseguire operazioni a catena interagendo soltando con un solo artefatto. Tecnicamente, nell'artefatto non vi è differenza tra l'essere usato da un agente o da un altro artefatto, ma resta comunque dinamicamente valida l'invariante per cui è sempre un agente a dare il via ad esecuzioni di questo tipo.

In CArtAgO i workspace definiscono la struttura dell'applicazione. Essi possono anche essere distribuiti su differenti nodi computazionali Ja-Ca. Di norma, ogni applicazione CArtAgO inizializza un workspace di default, ma lo stesso può essere creato a runtime da un agente attraverso l'azione `createWorkspace`. Un agente può entrare od uscire da un workspace attraverso le azioni `joinWorkspace` e `quitWorkspace`.

Vantaggi di JaCa-Android

- Il paradigma ad agenti adottato permette di **ridurre drasticamente il gap concettuale** che esiste tra l'analisi-progettazione e l'implementazione di sistemi software più o meno complessi. La progettazione di un sistema multi-agente si articola in tre fasi: definizione della topologia attraverso i workspaces; definizione dell'insieme di agenti che operano nei workspaces; defini-

zione dell'insieme degli artefatti presenti all'interno dei workspace. L'implementazione dello stesso sistema è realizzata utilizzando l'insieme di classi così come sono state definite nella progettazione e realizzandole utilizzando Jason e CArtAgO.

- Il paradigma ad agenti adottato permette di mappare all'interno degli agenti tutti i task e le attività interni al sistema, consentendo di strutturare l'architettura nel migliore dei modi poiché è possibile scegliere di caso in caso sia una **esecuzione concorrente** (task distribuiti tra gli agenti) che una **esecuzione interleaved** (task all'interno di un solo agente).

- Le **interazioni asincrone** possono essere gestite semplicemente specificando il comportamento reattivo degli agenti a fronte della percezione di un dato evento.

- Ja-Ca permette di realizzare nelle applicazioni contemporaneamente comportamenti **reattivi e pro-attivi** (all'interno degli agenti) in modo tale da evitare l'uso di callback nel caso in cui si vuole informare un componente software di un determinato evento. Così facendo, si evita l'inversione del flusso di controllo e la proliferazione di spaghetti-code asincrono.

- L'abilità degli agenti di adattare il proprio comportamento in base alle informazioni del contesto in cui si trovano, permette di realizzare facilmente applicazioni **context-aware** e **context-sensitive**.

- A differenza delle altre piattaforme, JaCa-Android non prende in considerazione i dispositivi mobile di vecchia generazione (*low-capacity devices*) o middleware ad-hoc come la JavaME, ma concentra l'interesse sui dispositivi di nuova generazione (*smartphones*) con particolare interesse ad Android come sistema operativo. Ciò non significa aver trascurato del tutto il problema delle risorse (memoria e capacità di calcolo), anzi, i risultati dei test sulle performance di reattività, sostenibilità computazionale e consumo della memoria dimostrano che le differenze di **performance tra una applicazione Android ed una** JaCa-Android **sono trascurabili**. Si può quindi asserire che il poco interesse è

giustificato dal fatto che il problema del vincolo sta scomparendo del tutto con la nuova generazione di dispositivi.

3.3.2 Progettazione agent-oriented

Focalizziamo la progettazione sulla applicazione mobile definendo dal punto di vista dell'applicazione quali sono gli elementi significativi del sistema, inquadrandoli poi nell'architettura generale. In particolare si vuole giungere ad avere un quadro completo dell'architettura che includa le entità in gioco individuandole dalle specifiche di alto livello dei task descritte nella Sezione 2.2.3.

I prossimi passi in direzione dell'architettura prevedono il mapping dei task nelle parti attive del sistema (*agenti*), l'individuazione ed il mapping delle risorse (*resources*) e delle loro modalità di accesso mediante le parti passive del sistema (*artefatti*), la definizione delle interazioni (*interactions*) che intercorrono tra agenti ed artefatti appartenenti al sistema.

Mapping Agents-Task

Riportiamo l'insieme dei task che il sistema dovrà svolgere dandone una descrizione ad alto livello per poi effettuare un mapping degli stessi nel numero di agenti opportuno e spiegandone brevemente le ragioni che hanno guidato il mapping.

- **AccessManagement** (***AccessMng***) è il task che regola l'accesso degli utenti al sistema. Il sistema prevede che un utente debba inizialmente registrarsi (*Registration*) per ottenere le credenziali di accesso. L'accesso al sistema *Login* avviene specificando le credenziali, mentre l'uscita dal sistema (*Logout*) non richiede particolari informazioni. Un utente già registrato può recuperare le credenziali con una procedura di *PasswordRecovery*.

- **AccountManagement** (***AccountMng***) è il task responsabile della gestione della manipolazione delle informazioni contenute nell'account di un utente. In particolare permette di manipolare l'ammontare di crediti chilometrici di un utente (*CCManaging*),

manipolare le informazioni personali di un utente (*PersInfoManaging*), manipolare (solo per l'amministratore) i costi fissi usati nel sistema (*FixedCostManaging*).

- **Notification** è il task responsabile della gestione delle notifiche inviate dal sistema agli utenti. Permette all'utente di leggere le notifice ricevute (*NotificationReading*) e di essere avvisato in tempo-reale dal sistema dell'avvenuta ricezione di una notifica (*NotificationChecking*).

- **Reservation** è il task responsabile della gestione delle prenotazioni di un trip. Permette al rider di richiedere la prenotazione di un trip (*RReservationRequesting*) ed al driver di accettare o rifiutare una richiesta di prenotazione per un suo trip (*DReservationResponding*).

- **TripManagement** (***TripMng***) è il task responsabile della gestione dei trip. Permette all'utente, dal profilo driver, di gestire i trip di cui è appunto il driver, in particolare di offrire un nuovo trip (*DTripOffering*) e visualizzare l'insieme dei trip creati (*DTripListManaging*). Permette all'utente, dal profilo rider, di gestire i trip in cui è rider, in particolare di cercare un nuovo trip a cui partecipare (*RTripSearching*), visualizzare l'insieme dei trip ai quali partecipa come rider tramite una prenotazione (*RResListManaging*).

- **TripMonitoring** (***TripMnt***) è il task responsabile della gestione e del monitoraggio di un trip in corso. Permette al driver di avviare e monitorare un trip da lui creato precedentemente visualizzandone l'evoluzione su una mappa digitale (*DTripWatching*). Permette al rider di monitorare un trip in corso a cui partecipa visualizzandone l'evoluzione su una mappa digitale (*RTripWatching*). A trip in corso permette al driver di effettuare presso il sistema il check-in di un rider (ingresso sul mezzo) (*DCheckingIn*) ed eventualmente di richiedere il rimborso per l'assenza del rider nel punto previsto per il prelievo (*DAmending*).

Una lettura attenta dei task permette di individuare due particolari task del sistema che implicano delle attività a lungo termine. I due task sono **Notification** e **TripMonitoring** (TripMnt) ed essi sono attivi allo stesso tempo quando l'utente (che sia driver o rider) intende monitorare un viaggio in corso al quale partecipa, pertanto possiamo considerarli come due attività a lungo termine in parallelo. I task **AccessManagement** (AccessMng), **AccountManagement** (AccountMng), **Reservation** e **TripManagement** (TripMng) sono attivi uno alla volta, per questo possono essere considerati come un insieme di attività a breve termine con esecuzione sequenziale.

Ritengo opportuno mappare i due task con esecuzione parallela a lungo termine in agenti diversi, in particolare il task **Notification** sarà mappato nell'agente `Notifier Agent`, invece il task **TripMonitoring** e l'insieme rimanente dei task saranno mappati in un unico agente, il `Carpooler Agent`. Di seguito il mapping completo dei task nei due agenti:

`Notifier Agent`

- **Notification**

`Carpooler Agent`

- **AccessManagement** (AccessMng)
- **AccountManagement** (AccountMng)
- **Reservation**
- **TripManagement** (TripMng)
- **TripMonitoring**

Individuazione delle risorse

Per svolgere i loro task, ognuno dei due agenti `Notifier Agent` e `Carpooler Agent` necessiterà di usare un insieme di risorse. Considerando il mapping dei task descritto precedentemente, individuiamo l'insieme delle risorse di cui l'agente avrà bisogno per lo svolgimento dell'insieme dei task a lui assegnati. Per semplificare la classificazione

delle risorse, operiamo dove possibile per tipologia, ad esempio raccogliamo l'insieme delle risorse che permettono l'interazione con l'utente in un'unica tipologia di risorsa.

Il **Notifier Agent**, per svolgere il task **Notification** avrà bisogno di accedere al *REST Service* per il recupero e la modifica delle notification, di interagire con l'utente per l'invio degli avvisi e la visualizzazione delle notification tramite una apposita *UserGUI* e di accedere alle *informazioni riguardanti l'utente corrente* che utilizza il sistema.

Il **Carpooler Agent**, per svolgere il **TripMonitoring** task avrà bisogno di accedere alle informazioni sulla localizzazione del dispositivo fornite dal *sensore GPS*, di accedere al *REST Service* per creare, recuperare e modificare le informazioni sui trip, le reservation, il driver e i rider (carpooler), di interagire con l'utente per la ricezione dei comandi e la visualizzazione delle informazioni tramite le apposite *UserGUI*, infine di accedere alle *informazioni riguardanti l'utente corrente* che utilizza il sistema. Per i rimanenti task **AccessManagement**, **AccountManagement**, **Reservation** e **TripManagement** valgono le risorse individuate precedentemente, escluso l'accesso alle informazioni sulla localizzazione.

Di seguito riportiamo l'elenco delle risorse individuate:

Sensore GPS , il sensore GPS che fornirà i dati sulla localizzazione geografica del dispositivo mobile;

REST Service , il servizio che fornirà l'accesso ai dati su carpooler, trip, reservation, notification (istanze del dominio descritte nella Sezione 2.2.1) e ne consentirà creazione, recupero e la modifica nelle modalità descritte dall'interfaccia in tabella 2.7;

UserGUI , la risorsa rappresentata dall'insieme delle interfacce utente, le quali gestiscono le interazioni con l'utente (comandi inviati dall'utente al sistema) e permettono la visualizzazione delle informazioni all'utente;

UserSession , la risorsa che terrà traccia delle *informazioni riguardanti l'utente corrente* che utilizza il sistema (dopo avervi fatto accesso) e delle quali informazioni ne consentirà la modifica (da parte del sistema).

Mapping Risorse-Artefatti

Individuate le risorse, possiamo procedere all'incapsulamento delle stesse all'interno di opportuni artefatti rendendole quindi accessibili all'esterno attraverso *operazioni* e/o *proprietà osservabili*. Teniamo in considerazione il vincolo che, un agente può avere diverse modalità di interazione con un artefatto, mentre un artefatto può accedere direttamente ad un altro artefatto soltanto attraverso un link precedentemente costruito ad-hoc da un agente. Pertanto, nei casi in cui si ritiene opportuno che un artefatto acceda direttamente ad un altro, si avrà l'accortezza di consentire il recupero delle informazioni incapsulate dal secondo artefatto mediante un'operazione di tipo `link` rendendo quindi l'artefatto *linkable*. L'insieme degli artefatti definiti è composto dai seguenti elementi:

`GPSManager` è l'artefatto che incapsula la risorsa *GPSSensor* e che quindi fornirà i dati sulla localizzazione geografica del dispositivo mobile. L'artefatto interagirà soltanto con il `Carpooler Agent`, per cui possiamo esporre l'informazione riguardante la posizione gps attraverso le proprietà osservabili `Latitude` e `Longitude`.

Nome	`GPSManager`
Prop. osservabili	`Latitude`, `Longitude`
Operazioni	-
Eventi generati	-

Tabella 3.1: GPSManager

`FCRRServiceArtifact` è l'artefatto che espone direttamente all'interno del sistema le operazioni fornite dall'interfaccia del REST Service, ovvero le operazioni `create`, `retrieve` e `update` relative a `Carpooler`, `Trip`, `Reservation`, `Notification` e `Fidex-cost`. L'artefatto FCRRServiceArtifact per poter sfruttare le operazioni dell'interfaccia del REST Service usa tramite link un artefatto che gli offre la possibilità di sottoporre richieste HTTP al servizio e di recuperarne le response. Grazie alla presenza dell'artefatto

FCRRServiceArtifact all'interno del sistema, è data la possibilità all'agente di accedere direttamente alle operazioni del REST Service e di vedere quindi il servizio come se fosse locale.

Nome	`FCRRServiceArtifact`
Prop. osservabili	-
Operazioni	• `createCarpooler(inputParams, OFP Carpooler.id)` • `retrieveCarpooler(selectParams, OFP CarpoolerList)` • `updateCarpooler(selectParams, updateParams)` • `...`
Eventi generati	-

Tabella 3.2: FCRRServiceArtifact

`UserSessionArtifact` è l'artefatto che incapsula le informazioni (in particolare l'identificativo) relative all'utente che in quel momento utilizza il sistema (dopo avervi fatto accesso) e delle quali informazioni ne consente la modifica. É necessario che queste informazioni siano incapsulate in un artefatto poiché devono essere accessibili contemporaneamente ai due agenti. La sua interfaccia avrà come operazioni `getUserID(OFP userID)` e `setUserID(userID)` per il recupero e la modifica delle informazioni sull'utente. Poiché le informazioni sull'utente devono essere accessibili ad altri artefatti, allora l'artefatto `UserSessionArtifact` sarà *linkable* e le sue operazioni saranno `link`.

`GUIArtifact` è l'artefatto che incapsula la risorsa rappresentata dall'insieme delle interfacce utente. L'artefatto genererà un evento a

Nome	`UserSessionArtifact`
Prop. osservabili	-
Operazioni	• `getUserID(OFP userID)` • `setUserID(userID)`
Eventi generati	-

Tabella 3.3: UserSessionArtifact

fronte di un comando inviato dall'utente. L'artefatto espone una interfaccia la cui operazione principale è `updateView()`. L'operazione `updateView()` avrà in ingresso le informazioni da visualizzare all'utente. Inoltre, un artefatto di tipo `UserGUIArtifact` avrà la necessità di recuperare le informazioni riguardanti l'utente che ha effettuato il login, pertanto sarà linkato anche all'artefatto `UserSessionArtifact` e potrà quindi triggerare il link `getUserID(OFP userID)`.

É opportuno specializzare l'artefatto `GUIArtifact` definito precedentemente definendone per ogni caso specifico l'insieme dei comandi (provenienti dall'utente) e quindi l'insieme degli eventi generati dall'artefatto. Ogni interazione utente-sistema ha origine attraverso una interfaccia grafica per l'utente (**G**raphical **U**ser **I**nterface), quindi è necessario conoscere gli elementi delle varie interfacce grafiche tramite i quali l'utente invierà i comandi al sistema.

Gli attori che utilizzano il software si dividono in *driver*, *rider* e *admin*. Nelle funzionalità comuni, i driver ed i rider visualizzeranno ed useranno le medesime interfacce utente, quindi interagiranno con il sistema attraverso gli stessi `UserGUIArtifact`. In alcuni casi però, il sistema dovrà sottoporre ai due profili di utilizzo delle interfacce differenti quindi delle specializzazioni diverse dell'artefatto `UserGUIArtifact`. Per disambiguazione, nominiamo convenzionalmente gli artefatti secondo il formato `X<Nome>GUI`. In particolare, `X` sarà la lettera "`D`" per le interfacce visualizzate dal profilo *driver*, la lettera "`R`" per quelle visualizzate dal profilo *rider* e la lettera "`A`" per

quelle visualizzate dal profilo *admin*. Tutte le interfacce in comune avranno il formato `<Nome>GUI`, quindi una sola lettera maiuscola iniziale. Ogni interfaccia è costituita da una serie di elementi grafici, i quali possono essere interattivi o meno. Per gli elementi grafici interattivi è evidenziato il comando che l'utente può sottoporre al sistema tramite il carattere *corsivo*. Le operazioni dell'artefatto sono specificate con il formato `<nomeOperazione>(<input>, ...)` e gli eventi generati con il formato `<nomeEvento>(<param>, ...)`.

Nome	`AccessGUI`
Elementi grafici	• form email e password • pulsante *login* • pulsante *lost password* • pulsante *registration*
Operazione	`updateView()`
Eventi generati	`login(email, pw)`, `lost-password(email)`, `registration`

Tabella 3.4: AccessGUI

Nome	`RegistrationGUI`
Elementi grafici	• form info di registrazione • pulsante *save*
Operazione	`updateView()`
Eventi generati	`save(info)`

Tabella 3.5: RegistrationGUI

Nome	`MainGUI`
Elementi grafici	• pulsante *account manage* • pulsante *notification list* • pulsante *driver* • pulsante *rider* • pulsante *logout*
Operazione	`updateView()`
Eventi generati	`account-manage`, `notification-list`, `driver`, `rider`, `logout`

Tabella 3.6: MainGUI

Nome	`AccountPanelGUI`
Elementi grafici	• pulsante *mod personal info* • pulsante *manage cc*
Operazione	`updateView()`
Eventi generati	`mod-personal-info`, `manage-cc`

Tabella 3.7: AccountPanelGUI

Nome	`ManageCCGUI`
Elementi grafici	• info ammontare cc e fixed-cost • input cc • pulsante *buy* • pulsante *sell* • pulsante *transfer*
Operazione	`updateView(carpooler)`
Eventi generati	`buy(cc)`, `sell(cc)`, `transfer`

Tabella 3.8: ManageCCGUI

Nome	`TransferCCGUI`
Elementi grafici	• info ammontare `cc` da trasferire • input email destinatario • pulsante *transfer* • pulsante *cancel*
Operazione	`updateView(carpooler)`
Eventi generati	`transfer(cc, email)`, `cancel`

Tabella 3.9: TransferCCGUI

Nome	`ManagePersInfoGUI`
Elementi grafici	• tabella info personali utente • pulsante *modify* • pulsante *save*
Operazione	`updateView(carpooler)`
Eventi generati	`modify`, `save(info)`

Tabella 3.10: ManagePersInfoGUI

Nome	`ACostManageGUI`
Elementi grafici	• info fixed-cost • pulsante *modify buy-cost* • pulsante *modify sell-cost*
Operazione	`updateView(fixed-cost)`
Eventi generati	`modify-buy-cost(cost)`, `modify-sell-cost(cost)`

Tabella 3.11: ACostManageGUI

Nome	`NotificationListGUI`
Elementi grafici	• lista delle notifiche ricevute • *selezione* notifica
Operazione	`updateView(notificationList)`
Eventi generati	`selezione-ntf(notificationID)`

Tabella 3.12: NotificationListGUI

Nome	`NotificationInfoGUI`
Elementi grafici	• info notifica • pulsante *see reservation* • pulsante *delete*
Operazione	`updateView(notification)`
Eventi generati	`see-reservation(reservationID)`, `delete(notificationID)`

Tabella 3.13: NotificationInfoGUI

Nome	`RTripInfoGUI`
Elementi grafici	• informazioni trip • menu opzione *reserve* • menu opzione *info driver* • menu opzione *monitor trip*
Operazione	`updateView(trip)`
Eventi generati	`reserve(tripID)`, `info-driver(driverID)`, `monitor-trip(tripID)`

Tabella 3.14: RTripInfoGUI

Nome	`DTripInfoGUI`
Elementi grafici	• informazioni trip • lista reservation del trip • *selezione* reservation • pulsante *delete* • pulsante *start*
Operazione	`updateView(trip, reservationList)`
Eventi generati	`selezione-drsv(reservationID)`, `delete(tripID)`, `start(tripID)`

Tabella 3.15: DTripInfoGUI

Nome	`DResInfoGUI`
Elementi grafici	• informazioni reservation • info rider • menu opzione *accept* • menu opezione *refuse*
Operazione	`updateView(reservation, rider)`
Eventi generati	`accept(reservationID)`, `refuse(reservationID)`

Tabella 3.16: DResInfoGUI

Nome	`RResListGUI`
Elementi grafici	• lista reservation personali • *selezione* reservation
Operazione	`updateView(reservationList)`
Eventi generati	`selezione-rrsv(reservationID)`

Tabella 3.17: RResListGUI

Nome	`RResInfoGUI`
Elementi grafici	• informazioni reservation • info driver • menu opzione *cancel*
Operazione	`updateView(reservation, driver)`
Eventi generati	`cancel(reservationID)`

Tabella 3.18: RResInfoGUI

Nome	`DTripManageGUI`
Elementi grafici	• pulsante *add trip* • pulsante *trip list*
Operazione	`updateView()`
Eventi generati	`add-trip`, `trip-list`

Tabella 3.19: DTripManageGUI

Nome	`RTripManageGUI`
Elementi grafici	• pulsante *search trip* • pulsante *reservation list*
Operazione	`updateView()`
Eventi generati	`search-trip`, `reservation-list`

Tabella 3.20: RTripManageGUI

Nome	`DTripListGUI`
Elementi grafici	• lista trip • *selezione* trip
Operazione	`updateView(tripList)`
Eventi generati	`selezione-dtrp(tripID)`

Tabella 3.21: DTripListGUI

Nome	`RTripListGUI`
Elementi grafici	• lista trip • *selezione* trip
Operazione	`updateView(tripList)`
Eventi generati	`selezione-rtrp(tripID)`

Tabella 3.22: RTripListGUI

Nome	`DAddTripGUI`
Elementi grafici	• form dati richiesti • pulsante *add*
Operazione	`updateView()`
Eventi generati	`add(tripData)`

Tabella 3.23: DAddTripGUI

Nome	`RSearchTripGUI`
Elementi grafici	• form dati richiesti • pulsante *find*
Operazione	`updateView()`
Eventi generati	`find(tripFilters)`

Tabella 3.24: RSearchTripGUI

Nome	`DTripMapGUI`
Elementi grafici	• mappa digitale • marker relativi ai carpooler • *selezione* marker • menu opzione *stop*
Operazione	`updateView(trip, driver, resList)`
Eventi generati	`selezione-drsv(reservationID)`, `stop(tripID)`

Tabella 3.25: DTripMapGUI

Nome	`DCheckInGUI`
Elementi grafici	• info nome rider • input check-in-code • pulsante *amends* • pulsante *check-in*
Operazione	`updateView(reservation)`
Eventi generati	`amends(reservationID)`, `check-in(check-in-code, resID)`

Tabella 3.26: DCheckInGUI

Nome	`RTripMapGUI`
Elementi grafici	• mappa digitale • marker rider e driver
Operazione	`updateView(trip, driver, resList)`
Eventi generati	-

Tabella 3.27: RTripMapGUI

3.3.3 Architettura della FCRR Mobile Application

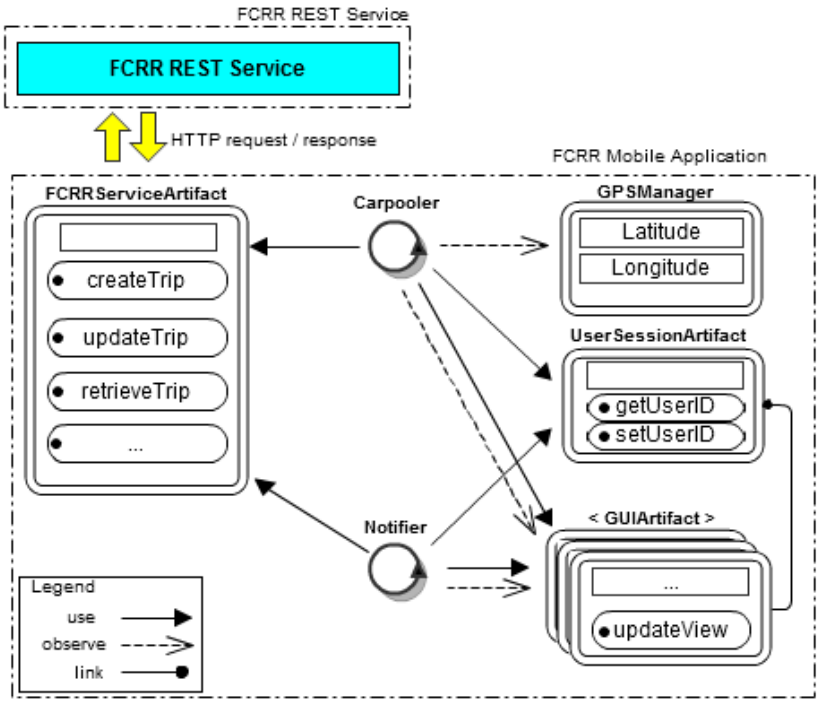

Figura 3.5: Architettura della *FMA*

Avendo definito l'insieme degli artefatti con cui interagisce la parte attiva del sistema *FMA*, in questo caso gli agenti `Notifier Agent` e `Carpooler Agent`, possiamo delineare l'architettura generale del sistema multi-agente esplicitando graficamente le tipologie di interazione che intercorrono tra le entità del sistema. Nel costruire la struttura del sistema, si ritiene opportuna l'adozione del pattern architetturale *MVC* (***M**odel **V**iew **C**ontroller*). Il pattern prevede la separazione del sistema in tre sottosistemi che interpretano tre ruoli diversi:

Model il sottosistema che fornisce l'accesso ai dati dell'applicazione, nel nostro caso rappresentata dagli artefatti `UserSessionArtifact`, `FCRRServiceArtifact`, `GPSManager`;

View il sottosistema che visualizza i dati contenuti nel model e si occupa dell'interazione con l'utente, nel nostro caso rappresentata dagli artefatti `GUIArtifact`;

Controller il sottosistema che reagisce ai comandi dell'utente e li attua modificando lo stato di Model e View, nel nostro caso rappresentato dagli agenti `Notifier` e `Carpooler`.

É bene notare come nella *FMA* il `FCRRServiceArtifact` rappresenta l'applicazione del pattern strutturale *Proxy* nella sua accezione più generale. L'artefatto fornisce lato client (*FMA*) la rappresentazione dell'interfaccia del REST Service, la quale in realtà appartiene ad un servizio che risiede lato server.

Interazioni

Con riferimento all'architettura di figura 3.5 esaminiamo le interazioni che si instaurano tra agenti ed artefatti.

- L'artefatto `GPSManager` sarà **creato** ed **osservato** dal `Carpooler Agent` con l'obiettivo di recuperare, in qualsiasi momento sia necessario, le informazioni sulla localizzazione gps del dispositivo mobile fornite dalle apposite proprietà osservabili `Latitude` e `Longitude`.

- L'artefatto `UserSessionArtifact` sarà **creato** (al momento opportuno) dal `Carpooler Agent` ed **usato** dagli agenti `Notifier` e `Carpooler` per recuperare e modificare tramite le operazioni `getUserID(OFP userID)` e `setUserID(userID)` le informazioni dell'utente corrente che utilizza il software.

- L'artefatto `GUIArtifact` sarà **creato**, **osservato** ed **usato** nelle sue diverse specializzazioni (a seconda dei task) dagli agenti `Notifier` e `Carpooler`. L'interazione tra l'agente e l'artefatto segue un preciso protocollo basato sulla generazione di un evento. L'artefatto `GUIArtifact` genera un determinato evento ogni qualvolta l'utente invia un comando al sistema tramite l'interfaccia, l'agente che osserva l'artefatto `GUIArtifact` percepisce l'evento generato e reagisce con un piano reattivo.

- L'artefatto `FCRRServiceArtifact` sarà **creato** dall'agente `Carpooler` ed **usato** dagli agenti `Notifier` e `Carpooler` triggerando le operazioni dell'interfaccia del REST Service (ad esempio `createCarpooler(inputParams, OFP Carpooler.id)`, `retrieveCarpooler(selectParams, OFP CarpoolerList)`, `updateCarpooler(selectParams, updateParams)`, ...).

Capitolo 4

Implementazione

In questo capitolo viene riportato un esempio di implementazione dell'applicazione mobile appartenente al sistema ***F**ixed-**C**ost **R**eal-time **R**idesharing* e progettata nel capitolo precedente. In particolare viene studiato il comportamento degli agenti `Notifier` e `Carpooler` introdotti nella progettazione descrivendo in maniera separata ogni macro-attività da essi svolta. Inoltre, per ogni task viene riportato un frammento del codice Jason che evidenzia i goal, eventuali sub-goal ed i piani reattivi più significativi. Infine viene riportato un frammento di codice per ogni artefatto definito ed un test-simulazione del monitoring di un viaggio in *FCRRS*.

4.1 Notifier Agent

L'agente ha in carico il task **Notification**, all'interno del quale distinguiamo due subtask indipendenti, **NotificationReading** (la lettura delle notification) e **NotificationChecking** (checking di nuove notification). Quest'ultimo subtask rappresenta una attività a lungo termine che l'agente esegue per l'intero ciclo di vita. I goal Jason possono essere considerati a tutti gli effetti come task, quindi i due termini task e goal all'interno della trattazione vengono considerati equivalenti.

4.1.1 Inizializzazione

Il goal iniziale `!start` prevede il raggiungimento di due sub-goal, il sub-goal di inizializzazione `!init` ed il subgoal `!notification_checking` che dà il via all'attività periodica di rilevazione delle notification. In fase di inizializzazione l'agente ha la necessità di recuperare e tenere traccia (attraverso un belief) dell'informazione sull'identificativo dell'utente corrente che utilizza il sistema. Per ottenere l'informazione si pone in attesa della creazione dell'artefatto `UserSessionArtifact` sul quale triggerare l'operazione `getUserID(OFP UserID)`, per poi aggiungere il belief `userID(UserID)` nella sua base di conoscenza. Al raggiungimento del goal di inizializzazione `!init` l'agente prosegue con il raggiungimento del goal `!notification_checking`.

4.1.2 Comportamento

Prendiamo in considerazione l'unico task mappato nel `Notifier Agent`, il **Notification**. Studiamo il comportamento dell'agente nei suoi subtask **NotificationChecking** e **NotificationReading** esplicitandone i piani con cui reagisce agli eventi generati dagli artefatti GUI, i goal ed eventuali sub-goal. Forniamo inoltre una pseudo-codifica del codice Jason dell'agente in modo da comprenderne con maggiore dettaglio il comportamento.

Notification Task.

Questo task si suddivide in due attività indipendenti e diverse tra loro. Il subtask *NotificationChecking* è una attività a lungo termine che viene svolta dall'agente `Notifier` al termine della sua fase di inizializzazione. Quest'ultimo subtask gestisce le notifiche inviate dal sistema all'utente avvisando l'utente in tempo reale dell'avvenuta ricezione di una notifica. Il sutask *NotificationReading* gestisce invece la lettura delle notifiche ricevute, un'attività a breve termine avviata per iniziativa dell'utente.

NotificationChecking.

- In seguito all'inizializzazione, l'agente ha in esecuzione il piano per il raggiungimento del goal `!notification_checking`. Con questo piano, l'agente ricava l'identificativo dell'utente, recupera la lista delle eventuali nuove notifiche relative all'utente usando l'artefatto `FCRRServiceArtifact` e aggiunge un nuovo goal `checkEmpty(NotificationList)`. Se il nuovo goal ha successo (la lista non è vuota), l'agente notifica all'utente la presenza delle notification (`send_advice(List)`) per poi aggiungere nuovamente il goal `!notification_checking`. Nel caso che il goal fallisca, il piano di riparazione fa ripartire il ciclo aggiungendo direttamente un nuovo goal `!notification_checking`.

NotificationReading. Questo subtask si compone di tre semplici piani reattivi triggerati da tre diverse percezioni.

- L'agente percepisce l'evento `notification_list` generato dall'artefatto `MainGUI`, usa l'artefatto `FCRRServiceArtifact` per recuperare la lista di notification dell'utente, usa l'artefatto `NotificationListGUI` per visualizzare la lista all'utente.
- L'agente percepisce l'evento `selezione_ntf(NotificationID)` generato dall'artefatto `NotificationListGUI`, usa l'artefatto `FCRRServiceArtifact` per recuperare i dettagli sulla notification e modificarne lo stato, usa l'artefatto `NotificationInfoGUI` per visualizzare i dettagli della notification all'utente.
- L'agente percepisce l'evento `delete(NotificationID)` generato dall'artefatto `NotificationInfoGUI`, usa l'artefatto `FCRRServiceArtifact` per modificare lo stato della notification.

```
!start.

+!start
  <-  !init;
      !notification_checking.
```

```
+!init
  <-  focusWhenAvailable("usersessionartifact");
      getUserID(UserID)[UserSessionArtifact];
      +userID(UserID);
      focusWhenAvailable("fcrrserviceartifact").

// NotificationChecking subtask
+!notification_checking: true
  <-  ?userID(UserID);
      retrieveNotification(UserID, 'unread', NotificationList)[FCRRServiceArtifact];
      !checkEmpty(NotificationList).

+!checkEmpty(NotificationList): notEmpty(NotificationList)
  <-  send_advice(List)
      !notification_checking.

-!checkEmpty(NotificationList): true
  <-  !notification_checking.

//NotificationReading
+notification-list: true
  <-  ?userID(UserID);
      retrieveNotification(UserID, NotificationList)[FCRRServiceArtifact];
      makeArtifact('NotificationListGUI');
      updateView(NotificationList)[NotificationListGUI].

+selezione_ntf(NotificationID): true
  <-  retrieveNotification(NotificationID, NotificationInfo)[FCRRServiceArtifact];
      updateNotification(NotificationID, 'read')[FCRRServiceArtifact];
      makeArtifact('NotificationInfoGUI');
      updateView(NotificationInfo)[NotificationInfoGUI].

+delete(NotificationID): true
  <-  retrieveNotification(NotificationID, NotificationInfo)[FCRRServiceArtifact];
      updateNotification(NotificationID, 'deleted')[FCRRServiceArtifact];
      makeArtifact('NotificationInfoGUI');
      updateView(NotificationInfo)[NotificationInfoGUI].
```

Tabella 4.1: Frammento pseudo-codice Jason dell'agente **Notifier** (***Notification Task***).

4.2 Carpooler Agent

L'agente ha in carico un insieme di task tra i quali possiamo distinguere task complessi e con esecuzione a lungo termine, come ad esempio il **TripMonitoring** (monitoraggio del viaggio), e task più semplici e caratterizzati da attività con esecuzione a breve termine.

4.2.1 Inizializzazione

L'agente non necessita di particolari belief iniziali e la sua fase di inizializzazione consiste nella creazione degli artefatti utili allo svolgimento delle attività. In particolare crea gli artefatti `FCRRServiceArtifact` e `GPSManager` e l'artefatto `AccessGUI` per permettere l'ingresso di un utente nel sistema.

4.2.2 Comportamento

Consideriamo uno alla volta i task mappati nel `Carpooler Agent` e studiamo il comportamento reattivo dell'agente esplicitandone i piani con cui reagisce agli eventi generati dagli artefatti GUI ed eventuali sub-goal. Dei task più significativi verrà fornita una pseudo-codifica del codice dell'agente Jason che andrà ad illustrarne con maggiore dettaglio il comportamento. I goal Jason possono essere considerati a tutti gli effetti come task, quindi i due termini task e goal all'interno della trattazione vengono considerati equivalenti.

TripMonitoring Task.

Essendo il task responsabile della gestione e del monitoraggio di un trip in corso, il **TripMonitoring** è uno dei task più significativi dal punto di vista del comportamento dinamico del `Carpooler Agent`. É opportuno separare l'intero task nei subtask *DTripWatching* (avvio del viaggio e monitoraggio da parte del driver), *DCheckingIn* (check-in di un rider a viaggio in corso), *DAmending* (risarcimento al driver a causa del forfait di un rider) *RTripWatching* (monitoraggio di un viaggio in corso da parte di un rider).

DTripWatching.

- L'agente percepisce l'evento `start(tripID)` generato dall'artefatto `DTripInfoGUI` che triggera l'omonimo piano reattivo. All'interno del piano memorizza nei belief l'informazione sullo stato del viaggio `trip_status('incorso')`, usa l'artefatto `FCRRServiceArtifact` per aggiornare lo stato del trip, memorizza nei belief l'identificativo del trip `tripID(TripID)` ed aggiunge il goal `!monitor(TripID)` con il quale dà il via al monitoraggio del viaggio che consiste nell'attività ciclica a lungo termine di recupero e visualizzazione delle posizioni di driver (tramite `GPSManager`) e rider (tramite `FCRRServiceArtifact`) sulla mappa digitale (`DTripMapGUI`).
- L'agente percepisce l'evento `stop(TripID)` generato dall'artefatto `DTripMapGUI` che triggera l'omonimo piano reattivo. All'interno del piano rimuove dai belief l'informazione sullo stato del viaggio `-trip_status('incorso')`, usa l'artefatto `FCRRServiceArtifact` per ricavare la lista delle reservation, aggiornare lo stato del trip e lo stato delle reservation azzerando le quote di `cc` riservate per il viaggio. Infine, accredita la somma dei `cc` sull'account del driver.

RTripWatching.

- L'agente percepisce l'evento `monitor_trip(TripID)` generato dall'artefatto `RTripInfoGUI` che triggera l'omonimo piano reattivo. Il piano aggiunge il gol `!monitoring(TripID)`. Il gol dà il via all'attività ciclica di monitoraggio che consiste nel recuperare lo stato reale del viaggio tramite `FCRRServiceArtifact`, controllare che il viaggio sia in corso, recuperare le posizioni dei riders e del driver per poi visualizzarle sulla mappa digitale. Il ciclo dell'attività riprende quando viene riaggiunto il medesimo goal (`!monitor_trip(TripID)`) alla fine del piano.

DCheckingIn.

- L'agente percepisce l'evento `check_in(Code,reservationID)` generato dall'artefatto `DCheckInGUI` che triggera il relativo piano. Il piano consiste nel raggiungimento dei goal `!check_proximity(ReservationID)` e `!check_in_code(Code,ReservationID)`. Il primo goal è raggiunto se il driver è abbastanza vicino al punto di picking-up, il secondo goal se il codice di check-in comunicato dal rider fa match con quello valido generato dal sistema al momento della prenotazione. Al raggiungimento dei due goal viene aggiornato il belief sulla stato del trip `-+trip_status('incorso')` e riprende l'attività di monitoring con l'aggiunta del goal `!monitor(TripID)`.

DAmending.

- L'agente percepisce l'evento `amends(reservationID)` generato dall'artefatto `DCheckInGUI` che fa scattare l'omonimo piano. Il piano prevede l'aggiornamento del belief `-+trip_status('paused')` ed il raggiungimento dei goal `!check_proximity(ReservationID)` e `!amending(ReservationID)`. Il primo goal è raggiunto se il driver si trova in prossimità del punto di pick-up, il secondo goal è raggiunto con il risarcimento del driver dovuto al forfait del rider. Con il raggiungimento dei due goal il belief sullo stato del viaggio viene aggiornato `-+trip_status('incorso')` e con l'aggiunta del goal `!monitor(TripID)` riprende il monitoraggio del viaggio.

```
// DTripWatching subtask
+start(TripID):
  <-  +trip_status('incorso');
      updateTrip('status','incorso')[FCRRServiceArtifact];
      +tripID(TripID);
      !monitor(TripID).

+!monitor(TripID): trip_status('incorso')
  <-  ?latitude(Lat)[GPSManager];
      ?longitude(Lng)[GPSManager];
      ?userID(UserID);
      updateCarpooler(UserID, Lat, Lng)[FCRRServiceArtifact];
      retrieveReservation(TripID, LatLngRiders)[FCRRServiceArtifact];
      retrieveCarpooler(UserID, LatDriver, LngDriver)[FCRRServiceArtifact];
```

```
      updateView(LatDriver, LngDriver, LatLngRiders)[DTripMapGUI];
      .wait(5000);
      !monitor(TripID).

+stop(TripID): trip_status('incorso')
  <-  -trip_status('incorso');
    retrieveReservation(TripID, ReservationList)[FCRRServiceArtifact];
    updateTrip(TripID, 'terminated')[FCRRServiceArtifact];
    updateReservation(ReservationList, status='checked-out', SumCC)[FCRRServiceArtifact];
    retrieveTrip(TripID, DriverID)[FCRRServiceArtifact];
    updateCarpooler(DriverID, CC=SumCC)[FCRRServiceArtifact].

// RTripWatching subtask
+monitor_trip(TripID):
  <-  !monitoring(TripID).

+!monitoring(TripID):
  <-  retrieveTrip(TripID, Status)[FCRRServiceArtifact];
      checkMatch(Status, 'incorso');
      retrieveReservation(TripID, LatLngRiders)[FCRRServiceArtifact];
      retrieveTrip(TripID, DriverID)[FCRRServiceArtifact];
      retrieveCarpooler(DriverID, LatDriver, LngDriver)[FCRRServiceArtifact];
      updateView(LatDriver, LngDriver, LatLngRiders)[DTripMapGUI];
      .wait(5000);
      !monitoring(TripID).

-!monitoring(TripID):
  <-  println('il viaggio non è in corso').


// DCheckingIn subtask
+check_in(Code, ReservationID): trip_status('incorso')
  <-  -+trip_status('paused');
      !check_proximity(ReservationID);
      !check_in_code(Code, ReservationID).

+!check_proximity(ReservationID): trip_status('paused')
  <-  retrieveReservation(ReservationID, LatRider, LngRider)[FCRRServiceArtifact];
      ?latitude(Lat)[GPSManager];
      ?longitude(Lng)[GPSManager];
      checkProximity(Lat, Lng, LatRider, LngRider).

+!check_in_code(Code, ReservationID): trip_status('paused')
  <-  ?tripID(TripID);
      retrieveReservation(ReservationID, ValidCode)[FCRRServiceArtifact];
      checkMatch(Code, ValidCode);
      updateReservation(ReservationID, 'checked-in')[FCRRServiceArtifact];
      -+trip_status('incorso');
      !monitor(TripID).

-!check_proximity(ReservationID): trip_status('paused')
  <-  println('posizione driver lontada dal pickup point');
      -+trip_status('incorso');
      ?tripID(TripID);
      !monitor(TripID);
```

```
-!check_in_code(Code, ReservationID): trip_status('paused')
  <-  println('codice check-in rider errato');
      -+trip_status('incorso');
      ?tripID(TripID);
      !monitor(TripID);

// DAmending subtask
+amends(reservationID): trip_status('incorso')
  <-  -+trip_status('paused');
      !check_proximity(ReservationID);
      !amending(ReservationID).

+!amending(ReservationID): trip_status('paused')
  <-  ?tripID(TripID);
      retrieveReservation(ReservationID, TripID, CCreserved)[FCRRServiceArtifact];
      updateReservation(ReservationID, 'amends', CCreserved=0)[FCRRServiceArtifact];
      retrieveCarpooler(TripID, DriverID)[FCRRServiceArtifact];
      updateCarpooler(DriverID, CC+CCreserved)[FCRRServiceArtifact];
      -+trip_status('incorso');
      !monitor(TripID).
```

Tabella 4.2: Frammento pseudo-codice Jason dell'agente **Carpooler** (***TripMonitoring Task***).

AccessManagement Task.

AccessManagement è il task che regola l'accesso degli utenti al sistema. Comprende quattro subtask indipendenti tra loro e caratterizzati a loro volta da semplici piani reattivi. I subtask si dividono in *Registration* (registrazione dell'utente al sistema per ottenere le credenziali), *Login* (accesso dell'utente al sistema per mezzo delle credenziali), *PasswordRecovery* (recupero delle credenziali dell'utente), *Logout* (uscita dell'utente dal sistema). Riportiamo in seguito le parti più significative di pseudo-codice Jason dell'agente senza entrare nei dettagli dei sub-goal di livello inferiore e della gestione del fallimento dei piani associati.

Registration.

- L'agente percepisce l'evento `registration` generato dall'artefatto `AccessGUI`, usa l'artefatto `RegistrationGUI`

per visualizzare il pannello per la registrazione di un utente nel sistema.

- L'agente percepisce l'evento `save(Info)` generato dall'artefatto `RegistrationGUI`, controlla con il raggiungimento del goal `!checkNotExist(Info)` che l'utente non sià già iscritto al sistema, usa l'artefatto `FCRRServiceArtifact` per creare il nuovo utente, usa l'artefatto `AccessGUI` per visualizzare il pannello di ingresso al sistema.

Login.

- L'agente percepisce l'evento `login(Email, Pw)` generato dall'artefatto `AccessGUI`, usa l'artefatto `FCRRServiceArtifact` per recuperare le credenziali valide ed altre informazioni dell'utente, controlla con il raggiungimento del goal `!check(Pw,ValidPw)` che ci sia corrispondenza tra le credenziali specificate e quelle valide, usa l'artefatto `UserSessionArtifact` per memorizzare le informazioni dell'utente appena entrato nel sistema, usa l'artefatto `MainGUI` per visualizzare il pannello principale del sistema.

PasswordRecovery.

- L'agente percepisce l'evento `lost_password(Email)` generato dall'artefatto `AccessGUI`, controlla con il raggiungimento del goal `!checkSubscribed(Email)` che l'utente sia iscritto al sistema, usa l'artefatto `FCRRServiceArtifact` per recuperare le credenziali dell'utente, invia tramite email le credenziali all'utente con il raggiungimento del goal `!sendEmail(Email,ValidPw)`, usa l'artefatto `AccessGUI` per visualizzare il pannello di ingresso al sistema.

Logout.

- L'agente percepisce l'evento `logout` generato dall'artefatto `MainGUI`, usa l'artefatto `UserSessionArtifact` per modificare le informazioni dell'utente utilizzatore del sistema,

usa l'artefatto `AccessGUI` per visualizzare il pannello di ingresso al sistema.

```
//Registration
+registration: true
  <-  makeArtifact('RegistrationGUI');
      updateView()[RegistrationGUI].

+save(Info): true
  <-  !checkNotExist(Info);
      createCarpooler(Info)[FCRRServiceArtifact];
      makeArtifact('AccessGUI');
      updateView()[AccessGUI].

//Login
+login(Email, Pw): true
  <-  retrieveCarpooler(Email, ValidPw, UserID)[FCRRServiceArtifact];
      !check(Pw, ValidPw);
      makeArtifact('UserSessionArtifact');
      setUserID(UserID)[UserSessionArtifact];
      makeArtifact('MainGUI');
      updateView()[MainGUI].

//PasswordRecovery
+lost_password(Email): true
  <-  !checkSubscribed(Email);
      retrieveCarpooler(Email, ValidPw)[FCRRServiceArtifact];
      !sendEmail(Email, ValidPw);
      updateView()[MainGUI].

//Logout
+logout: true
  <-  setUserID(ResetValue)[UserSessionArtifact];
      makeArtifact('AccessGUI');
      updateView()[AccessGUI].
```

Tabella 4.3: Frammento pseudo-codice Jason dell'agente **Carpooler** (***AccessManagement Task***).

AccountManagement Task.

Il task gestisce la manipolazione delle informazioni contenute nell'account dell'utente. Comprende quattro subtask indipendenti tra loro e caratterizzati a loro volta da semplici piani reattivi. Tra i subtask abbiamo *ProfileManaging* (accesso al pannello per le manipolazioni dei dati utente), *CCManaging* (gestione dell'ammontare dei crediti chilometrici dell'utente con i relativi subtask indipendenti per la vendita,

l'acquisto ed il trasferimento dei crediti), *PersInfoManaging* (gestione dei dati personali dell'utente) e *FixedCostManaging* (gestione dei costi fissi del sistema a carico dell'amministratore). Riportiamo in seguito le parti più significative di pseudo-codice Jason dell'agente senza entrare nei dettagli dei sub-goal di livello inferiore e della gestione del fallimento dei piani associati.

ProfileManaging.

- L'agente percepisce l'evento `account_manage` generato dall'artefatto `MainGUI`, usa l'artefatto `AccountPanelGUI` per visualizzare il pannello per la gestione dell'account.

CCManaging.

- L'agente percepisce l'evento `manage_cc` generato dall'artefatto `AccountPanelGUI`, usa l'artefatto `FCRRServiceArtifact` per recuperare l'ammontare corrente dei `cc` del suo account, usa l'artefatto `ManageCCGUI` per visualizzare l'ammontare dei `cc`.
- L'agente percepisce l'evento `transfer` generato dall'artefatto `ManageCCGUI`, usa l'artefatto `TransferCCGUI` per visualizzare la GUI per il trasferimento dei `cc`.

  CCBuying subtask. L'agente percepisce l'evento `buy(Cc)` generato dall'artefatto `ManageCCGUI`, aggiunge il goal `!buy_cc(UserID,Cc)` il cui piano usa l'artefatto `FCRRServiceArtifact` per accreditare i `cc` nel suo account (contestualmente alla transazione bancaria), usa l'artefatto `FCRRServiceArtifact` per recuperare l'informazione aggiornata sull'ammontare, usa l'artefatto `ManageCCGUI` per visualizzare l'ammontare dei `cc` aggiornato.

  CCSelling subtask. L'agente percepisce l'evento `sell(Cc)` generato dall'artefatto `ManageCCGUI`, aggiunge il goal `!sell_cc(UserID,Cc)` il cui piano usa l'artefatto `FCRRServiceArtifact` per diminuire i

`cc` nel suo account (contestualmente alla transazione bancaria), usa l'artefatto `FCRRServiceArtifact` per recuperare l'informazione aggiornata sull'ammontare, usa l'artefatto `ManageCCGUI` per visualizzare l'ammontare dei `cc` aggiornato.

CCTransfering subtask. L'agente percepisce l'evento `transfer(Cc,Email)` generato dall'artefatto `TransferCCGUI`, aggiunge il goal `!checkSubscribed(Email)` con il raggiungimento del quale si assicura che esista l'utente destinatario, ricava le informazioni sull'utente destinatario usando l'artefatto `FCRRServiceArtifact` aggiunge il goal `!transfer_cc(UserID,DestUserID,Cc)` il cui piano usa l'artefatto `FCRRServiceArtifact` per effettuare la transazione, usa l'artefatto `FCRRServiceArtifact` per recuperare l'informazione aggiornata sull'ammontare, usa l'artefatto `ManageCCGUI` per visualizzare l'ammontare dei `cc` aggiornato.

PersInfoManaging.

- L'agente percepisce l'evento `mod_personal_info` generato dall'artefatto `AccountPanelGUI`, usa l'artefatto `FCRRServiceArtifact` per recuperare le informazioni personali, usa l'artefatto `ManagePersInfoGUI` per visualizzare le informazioni personali.
- L'agente percepisce l'evento `modify` generato dall'artefatto `ManagePersInfoGUI`, usa l'artefatto `ManagePersInfoGUI` per rendere modificabili le informazioni personali.
- L'agente percepisce l'evento `save(Info)` generato dall'artefatto `ManagePersInfoGUI`, usa l'artefatto `FCRRServiceArtifact` per aggiornare le informazioni personali e recuperarle nuovamente, usa l'artefatto `ManagePersInfoGUI` per visualizzare le informazioni personali aggiornate.

FixedCostManaging.

- L'agente percepisce l'evento `modify_buy_cost(Cost)` generato dall'artefatto `ACostManageGUI`, usa l'artefatto `FCRRServiceArtifact` per aggiornare il nuovo valore del costo di acquisto e recuperare i valori aggiornati del costo di acquisto, usa l'artefatto `ACostManageGUI` per visualizzare i nuovi valori dei costi.
- L'agente percepisce l'evento `modify_sell_cost(Cost)` generato dall'artefatto `ACostManageGUI`, usa l'artefatto `FCRRServiceArtifact` per aggiornare il nuovo valore del costo di vendita e recuperare i valori aggiornati dei costi di vendita, usa l'artefatto `ACostManageGUI` per visualizzare i nuovi valori dei costi.

```
//CCManaging
+manage_cc: true
  <-  ?userID(UserID);
      retrieveCarpooler(UserID, InfoCC)[FCRRServiceArtifact];
      makeArtifact('ManageCCGUI');
      updateView(InfoCC)[ManageCCGUI].

+transfer: true
  <-  makeArtifact('TransferCCGUI');
      updateView()[TransferCCGUI].

// CCBuying
+buy(Cc): true
  <-  ?userID(UserID);
      !buy_cc(UserID, Cc);
      retrieveCarpooler(UserID, InfoCC)[FCRRServiceArtifact];
      makeArtifact('ManageCCGUI');
      updateView(InfoCC)[ManageCCGUI].

// CCSelling
+sell(Cc): true
  <-  ?userID(UserID);
      !sell_cc(UserID, Cc);
      retrieveCarpooler(UserID, InfoCC)[FCRRServiceArtifact];
      makeArtifact('ManageCCGUI');
      updateView(InfoCC)[ManageCCGUI].

// CCTransfering
+transfer(Cc,Email): true
  <-  ?userID(UserID);
      !checkSubscribed(Email);
      retrieveCarpooler(Email,DestUserID)[FCRRServiceArtifact];
      !transfer_cc(UserID,DestUserID,Cc);
      retrieveCarpooler(UserID, InfoCC)[FCRRServiceArtifact];
      makeArtifact('ManageCCGUI');
      updateView(InfoCC)[ManageCCGUI].
```

```
//PersInfoManaging
+mod_personal_info: true
  <-  ?userID(UserID);
      retrieveCarpooler(UserID, InfoPers)[FCRRServiceArtifact];
      makeArtifact('ManagePersInfoGUI');
      updateView(InfoPers)[ManagePersInfoGUI].

+save(Info): true
  <-  ?userID(UserID);
      updateCarpooler(UserID, InfoPers)[FCRRServiceArtifact];
      retrieveCarpooler(UserID, InfoPersUpdated)[FCRRServiceArtifact];
      updateView(InfoPersUpdated)[ManagePersInfoGUI].

//FixedCostManaging
+modify_buy_cost(Cost): true
  <-  updateFixedCost(Cost)[FCRRServiceArtifact];
      retrieveFixedCost(CostUpdated)[FCRRServiceArtifact];
      updateView(CostUpdated)[ACostManageGUI].

+modify_sell_cost(Cost): true
  <-  updateFixedCost(Cost)[FCRRServiceArtifact];
      retrieveFixedCost(CostUpdated)[FCRRServiceArtifact];
      updateView(CostUpdated)[ACostManageGUI].
```

Tabella 4.4: Frammento pseudo-codice Jason dell'agente **Carpooler** (***Account-Management Task***).

Reservation Task.

Il task gestisce le prenotazioni dei trip, in particolare le richieste da parte dei rider e le risposte da parte dei driver. Comprende due subtask indipendenti tra loro e caratterizzati a loro volta da semplici piani reattivi. Tra i subtask abbiamo *RReservationRequesting* (invio della richiesta di prenotazione da parte di un rider) e *DReservationResponding* (la risposta del driver in merito alla richiesta di prenotazione ricevuta). Riportiamo in seguito le parti più significative di pseudo-codice Jason dell'agente senza entrare nei dettagli degli eventuali sub-goal di livello inferiore e della gestione del fallimento dei piani associati.

RReservationRequesting.

- L'agente percepisce l'evento `reserve(TripID)` generato dall'artefatto `RTripInfoGUI` che triggera il relativo piano. All'interno del piano reattivo l'agente usa l'artefatto `FCRRServiceArtifact` per creare il la nuova reservation, la nuova notification per il driver e recuperare la lista aggiornata delle reservation, usa l'artefatto `RResListGUI` per visualizzare la lista delle reservation.

DReservationResponding.

- L'agente percepisce l'evento `accept(reservationID)` generato dall'artefatto `DResInfoGUI` che fa scattare il relativo piano. All'interno del piano, l'agente usa l'artefatto `FCRRServiceArtifact` per aggiornare lo stato della reservation, creare la nuova notification per il rider e recuperare le informazioni aggiornate sul trip, usa l'artefatto `DTripInfoGUI` per visualizzare le informazioni aggiornate sul trip.
- L'agente percepisce l'evento `refuse(reservationID)` generato dall'artefatto `DResInfoGUI` che fa scattare il relativo piano. All'interno del piano, l'agente usa l'artefatto `FCRRServiceArtifact` per aggiornare lo stato della reservation, creare la nuova notification per il rider e recuperare le informazioni aggiornate sul trip, usa l'artefatto `DTripInfoGUI` per visualizzare le informazioni aggiornate sul trip.

```
//RReservationRequesting
+reserve(TripID): true
  <-  ?userID(UserID);
      createReservation(TripID, UserID)[FCRRServiceArtifact];
      retrieveCarpooler(TripID, DriverID)[FCRRServiceArtifact];
      createNotification(TripID, DriverID)[FCRRServiceArtifact];
      retrieveReservation(UserID, ResList)[FCRRServiceArtifact];
      makeArtifact('RResListGUI');
      updateView(ResList)[RResListGUI].

//DReservationResponding
+accept(reservationID): true
  <-  updateReservation(reservationID, 'accepted')[FCRRServiceArtifact];
      createNotification(reservationID)[FCRRServiceArtifact];
      retrieveReservation(reservationID, TripID)[FCRRServiceArtifact];
      retrieveTrip(TripID, TripInfo)[FCRRServiceArtifact];
```

```
      makeArtifact('DTripInfoGUI');
      updateView(TripInfo)[DTripInfoGUI].

+refuse(reservationID): true
  <-  updateReservation(reservationID, 'refused')[FCRRServiceArtifact];
      createNotification(reservationID)[FCRRServiceArtifact];
      retrieveReservation(reservationID, TripID)[FCRRServiceArtifact];
      retrieveTrip(TripID, TripInfo)[FCRRServiceArtifact];
      makeArtifact('DTripInfoGUI');
      updateView(TripInfo)[DTripInfoGUI].
```

Tabella 4.5: Frammento pseudo-codice Jason dell'agente **Carpooler** (***Reservation Task***).

TripManagement Task.

Il task è responsabile della gestione dei trip. Comprende quattro task indipendenti tra loro e caratterizzati a loro volta da semplici piani reattivi. Tra i subtask abbiamo *DTripOffering* (l'offerta di un nuovo trip da parte del driver), *DTripListManaging* (la visualizzazione della lista dei trip creati dal driver), *RTripSearching* (la ricerca di un trip da parte del rider), *RResListManaging* (la visualizzazione dei trip a cui partecipa per mezzo di una prenotazione). Riportiamo in seguito le parti più significative di pseudo-codice Jason dell'agente senza entrare nei dettagli degli eventuali sub-goal di livello inferiore e della gestione del fallimento dei piani associati.

RoleSelecting.

- L'agente percepisce l'evento `driver` generato dall'artefatto `MainGUI`. Il piano associato usa l'artefatto `DTripManageGUI` per visualizzare il pannello per le operazioni dal profilo driver.
- L'agente percepisce l'evento `rider` generato dall'artefatto `MainGUI`. Il piano associato usa l'artefatto `RTripManageGUI` per visualizzare il pannello per le operazioni dal profilo rider.

DTripOffering.

- L'agente percepisce l'evento `add_trip` generato dall'artefatto `DTripManageGUI` e reagisce usando l'artefatto `DAddTripGUI` per visualizzare il pannello per la creazione di un nuovo trip.
- L'agente percepisce l'evento `add(TripData)`generato dall'artefatto `DAddTripGUI` e reagisce con l'apposito piano. Il piano prevede il raggiungimento del goal `!check_date(UserID,TripData)` il cui piano (omesso nella pseudo-codifica) ha successo se la data e l'ora indicate dal driver non sono già coperte da un altro viaggio organizzato dallo stesso driver. Raggiunto il goal, l'agente crea il nuovo viaggio e recupera la lista dei trip creati usando l'artefatto `FCRRServiceArtifact`. Infine visualizza la lista dei trip usando l'artefatto `DTripListGUI`.

RTripSearching.

- L'agente percepisce l'evento `search_trip` generato dall'artefatto `RTripManageGUI`, usa l'artefatto `RSearchTripGUI` per visualizzare il pannello per la ricerca di un trip.
- L'agente percepisce l'evento `find(TripFilters)` generato dall'artefatto `RSearchTripGUI` e fa scattare il relativo piano. Il piano prevede l'uso dell'artefatto `FCRRServiceArtifact` per recuperare la lista filtrata dei trip, l'uso dell'artefatto `RTripListGUI` per visualizzare la lista.
- L'agente percepisce l'evento `selezione_rtrp(TripID}` generato dall'artefatto `RTripListGUI` e ne viene avviato il piano reattivo associato. Il piano prevede l'uso dell'artefatto `FCRRServiceArtifact` per il recuperaro delle informazioni sul trip, l'uso dell'artefatto `RTripInfoGUI` per visualizzare le informazioni del trip.

DTripListManaging.

- L'agente percepisce l'evento `trip_list` generato dall'artefatto `DTripManageGUI`, usa l'artefatto `UserSessionArtifact` per recuperare l'identificativo dell'utente, usa l'artefatto `FCRRServiceArtifact` per recuperare la lista filtrata dei trip creati dall'utente, usa l'artefatto `DTripListGUI` per visualizzare la lista.
- L'agente percepisce l'evento `selezione_dtrp(TripID)` generato dall'artefatto `DTripListGUI`, usa l'artefatto `FCRRServiceArtifact` per recuperare le informazioni sul trip, usa l'artefatto `DTripInfoGUI` per visualizzare i dettagli sul trip.

RResListManaging.

- L'agente percepisce l'evento `reservation_list` generato dall'artefatto `RTripManageGUI`, usa l'artefatto `UserSessionArtifact` per recuperare l'identificativo dell'utente, usa l'artefatto `FCRRServiceArtifact` per recuperare la lista delle reservation relative all'utente, usa l'artefatto `RResListGUI` per visualizzare la lista delle reservation.
- L'agente percepisce l'evento `selezione_rres(ReservationID)` generato dall'artefatto `RResListGUI`, usa l'artefatto `FCRRServiceArtifact` per recuperare le informazioni sulla reservation, usa l'artefatto `RResInfoGUI` per visualizzare i dettagli sulla reservation.

```
//RoleSelecting
+driver: true
  <-  makeArtifact('DTripManageGUI');
      updateView()[DTripManageGUI].

+rider: true
  <-  makeArtifact('RTripManageGUI');
      updateView()[RTripManageGUI].

//DTripOffering
+add_trip: true
  <-  makeArtifact('DAddTripGUI');
      updateView()[DAddTripGUI].
```

```
+add(TripData): true
  <-  ?userID(UserID);
      !check_date(UserID,TripData);
      createTrip(UserID, TripData)[FCRRServiceArtifact];
      retrieveTrip(UserID,TripList)[FCRRServiceArtifact];
      makeArtifact('DTripListGUI');
      updateView(TripList)[DTripListGUI].

//RTripSearching
+search_trip: true
  <-  makeArtifact('RSearchTripGUI');
      updateView()[RSearchTripGUI].

+find(TripFilters): true
  <-  retrieveTrip(TripFilters,TripList)[FCRRServiceArtifact];
      makeArtifact('RTripListGUI');
      updateView(TripList)[RTripListGUI].

+selezione_rtrp(TripID}: true
  <-  retrieveTrip(TripID,TripInfo)[FCRRServiceArtifact];
      makeArtifact('RTripInfoGUI');
      updateView(TripInfo)[RTripInfoGUI].

//RResListManaging
+reservation_list: true
  <-  ?userID(UserID);
      retrieveReservation(UserID,ResList)[FCRRServiceArtifact];
      makeArtifact('RResListGUI');
      updateView()[RResListGUI].

+selezione_rres(ReservationID): true
  <-  retrieveReservation(ReservationID,ResInfo)[FCRRServiceArtifact];
      makeArtifact('RResInfoGUI');
      updateView(ResInfo)[RResInfoGUI].
```

Tabella 4.6: Frammento pseudo-codice Jason dell'agente **Carpooler** (***TripManagement Task***).

4.3 Artefatti

Riportiamo i frammenti di codice più significativi degli artefatti definiti nella progettazione della FCRR Mobile Application (Sezione 3.3.2) con cui interagiscono gli agenti `Notifier` e `Carpooler`. Verranno illustrati nell'ordine `FCRRServiceArtifact`, `GPSManager`, `UserSessionArtifact` e `DTripInfoGUI` che è stato scelto come esempio di implementazione di un artefatto della tipologia `GUIArtifact`. In particolare `DTripInfoGUI` rappresenta l'interfaccia che visualizza la lista dei viaggi creati dal driver.

FCRRServiceArtifact

Ricordiamo che `FCRRServiceArtifact` è l'artefatto che espone direttamente all'interno del sistema FCRR Mobile Application le operazioni fornite dall'interfaccia del REST Service, ovvero le operazioni `create`, `retrieve` e `update` relative a `Carpooler`, `Trip`, `Reservation`, `Notification` e `Fidex-cost`. In questo esempio di codifica sono presenti le operazioni `retrieveCarpooler()`, `retrieveTrip()`, `retrieveReservation()`.

```
package jaca.android.fcrr;

import java.util.ArrayList;
import java.util.HashMap;

import org.json.JSONException;
import org.json.JSONObject;

import cartago.ARTIFACT_INFO;
import cartago.INTERNAL_OPERATION;
import cartago.OPERATION;
import cartago.OUTPORT;
import cartago.OpFeedbackParam;
import cartago.OperationException;

import jaca.android.dev.JaCaArtifact;
import jaca.android.fcrr.util.Url;
import jaca.android.fcrr.util.JsonParam;
import jaca.android.fcrr.util.HttpRequestLinkedOperations;

@ARTIFACT_INFO(
  outports = {
    @OUTPORT(name = "out-1")
      }
) public class FCRRServiceArtifact extends JaCaArtifact {
```

```
  ...

  protected void init(){}


  /**
  * Operazione di recupero del Carpooler
  */
  @OPERATION void retrieveCarpooler( int userID,
  OpFeedbackParam<ArrayList<HashMap<String, String>>> outInfoCarpooler){
    // costruisco l'oggetto json che andrà in input
    JSONObject params = new JSONObject();
    JSONObject select = new JSONObject();
    try {
      select.put(JsonParam.Carpooler.id, userID);
      params.put(JsonParam.selectParams, select);
    } catch (JSONException e) {
      e.printStackTrace();
    }
// istanzio la "lista" delle info carpooler
OpFeedbackParam<ArrayList<HashMap<String,String>>> infoCarpooler =
  new OpFeedbackParam<ArrayList<HashMap<String,String>>>();
try {
  execLinkedOp( "out-1",
      HttpRequestLinkedOperations.getHttpReqLinked,
      Url.getCarpooler,
      params,
      infoCarpooler);
  //log("printing response: "+infoCarpooler.get());
} catch (Exception ex){
  ex.printStackTrace();
}
outInfoCarpooler.set(infoCarpooler.get());
  }



  /**
  * Operazione di recupero del Trip
  */
@OPERATION void retrieveTrip( JSONObject joTripID,
  OpFeedbackParam<ArrayList<HashMap<String, String>>> outInfoTrip){
// costruisco l'oggetto json che andrà in input
JSONObject params = new JSONObject();
try {
  params.put(JsonParam.selectParams, joTripID);
    } catch (JSONException e) {
      e.printStackTrace();
    }
    // istanzio la "lista" delle info sul viaggio
    OpFeedbackParam<ArrayList<HashMap<String,String>>> infoTrip =
      new OpFeedbackParam<ArrayList<HashMap<String,String>>>();
    try {
      execLinkedOp( "out-1",
        HttpRequestLinkedOperations.getHttpReqLinked,
        Url.getTrip,
```

```
        params,
        infoTrip);
      //log("printing response: "+infoTrip.get());
    } catch (Exception ex){
      ex.printStackTrace();
    }
    outInfoTrip.set(infoTrip.get());
  }


  /**
  * Operazione di recupero della Reservation
  */
@OPERATION void retrieveReservation(JSONObject joTripID,
  OpFeedbackParam<ArrayList<HashMap<String, String>>> outRiders){

    // costruisco l'oggetto json che andrà in input
    JSONObject params = new JSONObject();
    try {
      params.put(JsonParam.selectParams, joTripID);
    } catch (JSONException e) {
      e.printStackTrace();
    }
    // istanzio la lista dei riders
    OpFeedbackParam<ArrayList<HashMap<String,String>>> riders =
      new OpFeedbackParam<ArrayList<HashMap<String,String>>>();
    try {
      execLinkedOp( "out-1",
        HttpRequestLinkedOperations.getHttpReqLinked,
        Url.getReservation,
        params,
        riders);
      //log("printing response: "+riders.get());
    } catch (Exception ex){
      ex.printStackTrace();
    }
    outRiders.set(riders.get());
  }

  /**
  * Operazione ...
  */
  @OPERATION void ...
}
```

Tabella 4.7: Frammento codice dell'artefatto **FCRRServiceArtifact**.

GPSManager

Ricordiamo che `GPSManager` è l'artefatto sarà usato dall'agente `Carpooler` per il recupero dei dati sulla localizzazione geogra-

fica del dispositivo mobile rese disponibili attraverso le proprietà osservabili `Latitude` e `Longitude`. L'implementazione completa dell'artefatto è disponibile nella libreria di JaCa-Android (`jaca.android.tools.GPSManager`).

```
package jaca.android.tools;

import jaca.android.dev.LocationManagerArtifact;
...

/**
 * Artifact used for managing the GPS of the device
 */
public class GPSManager extends LocationManagerArtifact {

  ...

  /**
  * Observable property containing the current latitude
  */
  public static final String LATITUDE = "latitude";

  /**
  * Observable property containing the current longitude
  */
  public static final String LONGITUDE = "longitude";

  /**
  * Observable property containing the current altitude
  */
  public static final String ALTITUDE = "altitude";

  /**
  * Observable property containing the current accurancy
  */
  public static final String ACCURANCY = "accurancy";

  /**
  * Observable property containing the current bearing
  */
  public static final String BEARING = "bearing";

  /**
  * Observable property containing the current speed
  */
  public static final String SPEED = "speed";

  /**
  * Observable property containing the current time
  */
  public static final String TIME = "time";

  /**
  * Artifact default name
  */
```

```
public static final String ARTIFACT_DEF_NAME =  "gps-manager";

/**
* Default init
*/
public void init() {
  //Log.v("gpsmanager", "ON_INIT");
  init(0,0);
}

/**
* Init with specific parameters
* @param minTime
* @param minDistance
*/
public void init(int minTime, int minDistance) {
  super.init(minTime, minDistance);
  linkOnLocationChangedEventToOp(LocationManager.GPS_PROVIDER, OP_ON_LOCATION_CHANGE);
  linkOnProviderEnabledEventToOp(LocationManager.GPS_PROVIDER, OP_ON_PROVIDER_ENABLED);
  linkOnProviderDisabledEventToOp(LocationManager.GPS_PROVIDER, OP_ON_PROVIDER_DISABLED);

  Location location = getLocationManager().getLastKnownLocation(LocationManager.GPS_PROVIDER);
  if (location!=null) {
    defineObsProperty(LATITUDE, location.getLatitude());
    defineObsProperty(LONGITUDE, location.getLongitude());
    defineObsProperty(ALTITUDE, location.getAltitude());
    defineObsProperty(ACCURANCY, location.getAccuracy());
    defineObsProperty(BEARING, location.getBearing());
    defineObsProperty(SPEED, location.getSpeed());
    defineObsProperty(TIME, location.getTime());
  } else {
    defineObsProperty(LATITUDE, 0);
    defineObsProperty(LONGITUDE, 0);
    defineObsProperty(ALTITUDE, 0);
    defineObsProperty(ACCURANCY, 0);
    defineObsProperty(BEARING, 0);
    defineObsProperty(SPEED, 0);
    defineObsProperty(TIME, 0);
  }
}

protected void dispose() {
  super.dispose();
  removeLocationUpdates(LocationManager.GPS_PROVIDER);
}

@INTERNAL_OPERATION void onLocationChange(Location arg0) {
  //Log.v("gpsmanager", "locationchangefound");
  signal(ON_LOCATION_CHANGE, arg0.getProvider());
  getObsProperty(LATITUDE).updateValue(arg0.getLatitude());
  getObsProperty(LATITUDE).updateValue(arg0.getLatitude());
  getObsProperty(LONGITUDE).updateValue(arg0.getLongitude());
  getObsProperty(ALTITUDE).updateValue(arg0.getAltitude());
  getObsProperty(ACCURANCY).updateValue(arg0.getAccuracy());
  getObsProperty(BEARING).updateValue(arg0.getBearing());
  getObsProperty(SPEED).updateValue(arg0.getSpeed());
```

```
    getObsProperty(TIME).updateValue(arg0.getTime());
    //Log.v("gpsmanager", "locationchangeserved");
  }

  @INTERNAL_OPERATION void onProviderEnabled(String provider) {
    //Log.v("gpsmanager", "onProviderEnabled");
    signal(ON_PROVIDER_ENABLED,provider);
  }

  @INTERNAL_OPERATION void onProviderDisabled(String provider) {
    signal(ON_PROVIDER_DISABLED,provider);
  }
}
```

Tabella 4.8: Frammento codice dell'artefatto **GPSManager**.

UserSessionArtifact

Ricordiamo che `UserSessionArtifact` è l'artefatto che fornisce agli agenti ad ad altri artefatti le informazioni (in particolare l'identificativo) relative all'utente che in quel momento utilizza il sistema (dopo avervi fatto accesso) e delle quali informazioni ne consente la modifica. Il recupero e la modifica dell'identificativo utente avviene attraverso i *link* `getUserID(OFP userID)` e `setUserID(userID)`.

```
package jaca.android.fcrr;

import jaca.android.dev.JaCaArtifact;
import cartago.LINK;
import cartago.OpFeedbackParam;

public class UserSessionArtifact extends JaCaArtifact {

  private Integer userID;

  ...

  protected void init(){
    this.userID = 0;
    ...
  }

  @LINK void setUserID(Integer userID){
    this.userID = userID;
  }

  @LINK void getUserID(OpFeedbackParam<Integer> userID){
```

```
    userID.set(this.userID);
  }

  ...
}
```

Tabella 4.9: Frammento codice dell'artefatto **UserSessionArtifact**.

DTripInfoGUI (GUIArtifact)

Come anticipato precedentemente, `DTripInfoGUI` è l'artefatto che rappresenta la GUI per visualizzare al driver l'elenco dei viaggi da lui creati. Da notare come l'invio dei comandi di pressione dei pulsanti da parte dell'utente venga gestito da un'operazione interna dell'artefatto `onClick(View v)` linkata al pulsante (in questo caso *start*) attraverso `linkOnClickEventToOp(startTripButton, ‘‘onClick’’)`. L'operazione interna genera un evento (in questo caso un segnale) `signal(‘‘start’’, joTripID)` percepito poi dall'opportuno agente. É opportuno osservare come l'aggiornamento della lista venga triggerato attraverso l'operazione `updateView(tripID, tripInfo,riders)` caratteristica dei GUIArtifact che ha in ingresso i dati da visualizzare all'utente.

```
package jaca.android.fcrr;

import java.util.ArrayList;
import java.util.HashMap;
...


public class DTripInfoGUI extends ListActivityArtifact {

  private DTripInfoActivity activityRef;
  private View startTripButton;
  private View deleteTripButton;
  private JSONObject joTripID;

  @Override
  protected void init(IJaCaActivity activity, Bundle savedInstanceState) {
    super.init(activity, savedInstanceState);

    // ricavo l'attività associata alle risorse
    activityRef = (DTripInfoActivity)activity;
```

```
    // definisco una proprietà osservabile (visualizzazione GUI)
    defineObsProperty("gui_state", "not_displayed");

    startTripButton = activityRef.findViewById(R.id.startTrip);
    deleteTripButton = activityRef.findViewById(R.id.deleteTrip);

    linkOnClickEventToOp(startTripButton, "onClick");
    linkOnClickEventToOp(deleteTripButton, "onClick");
  }

  /**
  * Operazione di aggiornamento dell'interfaccia
  */
  @OPERATION void updateView( JSONObject tripID,
      ArrayList<HashMap<String, String>> tripInfo,
      ArrayList<HashMap<String, String>> riders){
    // salvo l'identificativo del trip
    joTripID = tripID;
    // riempio la lista con il metodo dell'activity
    activityRef.fillView(riders, tripInfo);
  }

  @INTERNAL_OPERATION void onClick(View v) {
    switch (v.getId()) {
      case R.id.startTrip:
        signal("start", joTripID);
        break;
      case R.id.deleteTrip:
        signal("delete", joTripID);
      default:
        break;
    }
  }

  @INTERNAL_OPERATION void onStart(){
    ObsProperty obsProp = getObsProperty("gui_state");
      obsProp.updateValue("displayed");
  }

  @INTERNAL_OPERATION void onStop(){
    ObsProperty obsProp = getObsProperty("gui_state");
    obsProp.updateValue("not_displayed");
  }
}
```

Tabella 4.10: Frammento codice dell'artefatto **DTripInfoGUI (GUIArtifact)**.

4.4 Test-simulazione di un viaggio in *FCRRS* (Driver TripMonitoring)

Riportiamo ora un test di simulazione di viaggio monitorato attraverso la *FCRR Mobile Application* (*FMA*) installata sul dispositivo mobile del driver. Focalizziamo l'attenzione sulla fase di monitoring, ovvero il momento in cui il driver avvia il viaggio ed ipotizziamo che lo stesso driver abbia precedentemente accettato tre prenotazioni (*Reservation*) per il suddetto viaggio.

Diamo per scontato che l'utente, successivamente all'operazione di login abbia scelto di operare sotto il profilo driver e che stia visualizzando l'apposito menù operativo per la gestione dei trip lato driver (`DTripManageGUI`). Procediamo passo per passo seguendo le azioni del driver e visualizzandone l'effetto sul touchscreen del dispositivo:

1. Per effetto del pulsante *trip list* il driver visualizza l'elenco dei trip da lui stesso programmati (visibile in figura 4.1). Ogni elemento della lista espone il set di attributi fondamentali che caratterizzano il viaggio: **from/to** (luogo di partenza/destinazione), **date** (data ed ora di partenza), **taken seats** (il numero di posti occupati rispetto al numero iniziale di posti disponibili).

2. L'utente sceglie di visualizzare ulteriori dettagli circa il primo viaggio della lista selezionandolo mediante il touchscreen. L'effetto di questa azione è la visualizzazione delle informazioni sul viaggio attraverso la `DTripInfoGUI` in figura 4.2. Oltre alle informazioni sul viaggio, in questa videata appare anche l'elenco dei rider o meglio delle *Reservation* accettate dal driver per quel determinato viaggio. In questo caso le prenotazioni sono tre e riportano i nomi dei carpooler *Marco Rossi*, *Angelo Verdi*, *Luca Bianchi*.

3. A questo punto il driver decide di far iniziare il viaggio per mezzo del pulsante *start*, dando così il via al monitoraggio. L'effetto dello *start* è la visualizzazione della mappa digitale (DTripMapGUI) che evidenzia le posizioni geografiche dei rider attraverso i marker colorati raffiguranti un omino stilizzato e la corrente

posizione geografica del driver mediante un marker rosso raffigurante una automobile stilizzata. Da questo momento in poi la mappa continuerà ad aggiornarsi periodicamente riportando gli spostamenti del driver in tempo reale e facilitando l'operazione di picking-up dei rider. É bene precisare che gli stessi rider dotati della FCRR Mobile Application potranno allo stesso tempo visualizzare la mappa dal loro dispositivo seguendo così in tempo reale gli spostamenti dell'automobile.

4. Giunti a destinazione (figura 4.4), il driver decreterà la fine del viaggio mediante un comando di *stop* disponibile dal menu (non visualizzato nello screenshot) che realizzerà in modo trasparente il check-out dei passeggeri e di conseguenza l'accredito della quota di ccriservati da ognuno di essi per il viaggio.

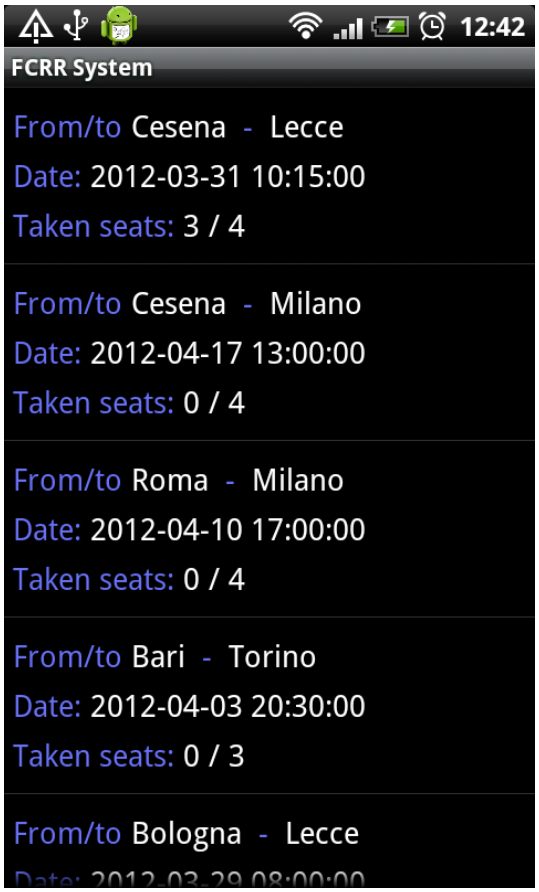

Figura 4.1: Screenshot dell'elenco dei viaggi programmati dal driver (`DTripListGUI` *FMA*)

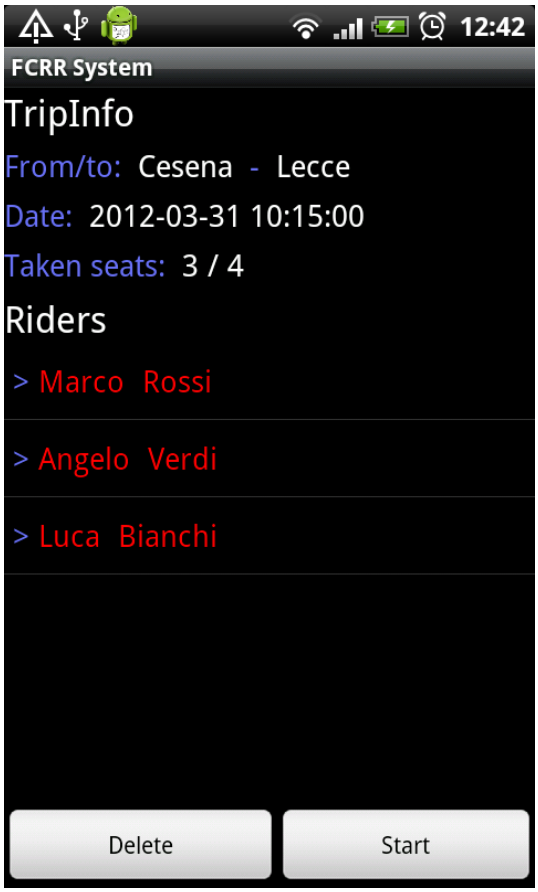

Figura 4.2: Screenshot delle informazioni sul viaggio programmato (`DTripInfoGUI` *FMA*)

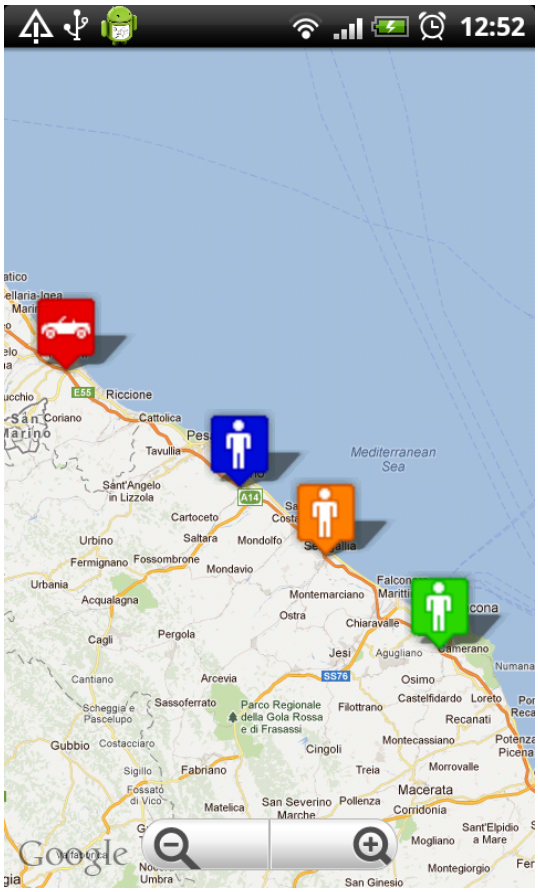

Figura 4.3: Screenshot della mappa iniziale di monitoraggio (`DTripMapGUI` *FMA*)

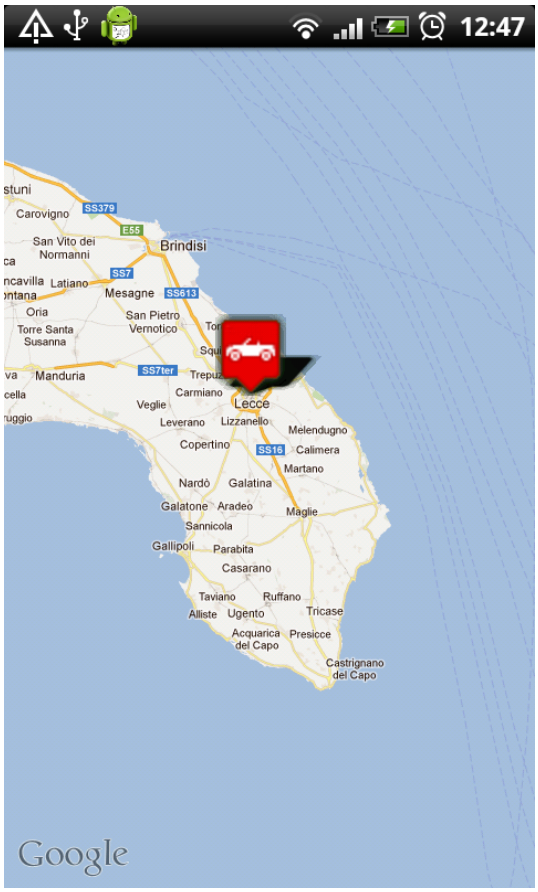

Figura 4.4: Screenshot della mappa finale di monitoraggio (`DTripMapGUI` *FMA*)

Note: per realizzare la variazione delle coordinate di geolocalizzazione del sensore GPS durante il test-simulazione di viaggio, l'artefatto `GPSManager` è stato sostituito da un artefatto "mock" (`GPSManager` simulato), chiamato `GpsSimulatorArtifact` ed usato dall'agente `Simulator` mediante triggering periodico dell'operazione `move()` che ha come effetto la variazione delle coordinate `Latitude` e `Longitude` (proprietà osservabili dell'artefatto `GpsSimulatorArtifact`). Per completezza riportiamo i frammenti significativi del codice dell'artefatto `GpsSimulatorArtifact` (tabella 4.11) e dell'agente `Jason Simulator` (tabella 4.12).

```
package jaca.android.fcrr.simulator;

import java.util.HashMap;

import cartago.INTERNAL_OPERATION;
import cartago.OPERATION;
import cartago.ObsProperty;
import jaca.android.dev.LocationManagerArtifact;
import jaca.android.fcrr.util.MapPoint;

public class GpsSimulatorArtifact extends LocationManagerArtifact {

  public static final String OP_ON_LOCATION_CHANGE = "onLocationChange";
  public static final String ON_LOCATION_CHANGE = "onLocationChange";
  public static final String LATITUDE = "latitude";
  public static final String LONGITUDE = "longitude";
  public static final String SENSOR_NAME = "gps-sensor";

  private Integer currentMapPoint;
  private HashMap<Integer, MapPoint> pathHashMap;


  protected void init(){
    //log(" Init passed.");
    defineObsProperty(LATITUDE, 0);
    defineObsProperty(LONGITUDE, 0);
    //linkOnLocationChangedEventToOp(SENSOR_NAME, OP_ON_LOCATION_CHANGE);

// inizializzazione hasmap path
currentMapPoint = 0;
pathHashMap = new HashMap<Integer, MapPoint>();
initPath();
}

  @INTERNAL_OPERATION void onLocationChange() {
    // aggiornamento valori
    getObsProperty(LATITUDE).updateValue(pathHashMap.get(currentMapPoint).getDoubleLat());
    getObsProperty(LONGITUDE).updateValue(pathHashMap.get(currentMapPoint).getDoubleLng());
    signal(ON_LOCATION_CHANGE, SENSOR_NAME);
  }
```

```
  @OPERATION void move(){
    currentMapPoint++;
    if (currentMapPoint<31) {
      ObsProperty lat = getObsProperty(LATITUDE);
      ObsProperty lng = getObsProperty(LONGITUDE);
        lat.updateValue(pathHashMap.get(currentMapPoint).getDoubleLat());
        lng.updateValue(pathHashMap.get(currentMapPoint).getDoubleLng());
        signal(ON_LOCATION_CHANGE, SENSOR_NAME);
    }
  }

  @INTERNAL_OPERATION void initPath() {
    // inserisco i punti del percorso Cesena, via Genova -> Lecce, via Trinchese
    /**
    * PARTENZA
    */
    // cesena, via genova
    pathHashMap.put(0, new MapPoint("44.151897,12.240154"));

    ...

    /**
    * ARRIVO
    */
    // Lecce, via trinchese
    pathHashMap.put(22, new MapPoint("40.353461,18.174026"));
  }
}
```

Tabella 4.11: Frammento codice dell'artefatto **GpsSimulatorArtifact**.

```
!start.

+!start
  <-  !init;
      !do_job.

+!init
  <-  println("[fcrr-sim:] Running.");
      // Console
      lookupArtifact("console", LocalConsoleID);
      +art_id(console, LocalConsoleID);
      println("[fcrr-sim:] Init done!.").

+!do_job
  <-  println("[fcrr-sim:] Waiting for tripGUI...")[artifact_id(LocalConsoleID)];
      focusWhenAvailable("dtripinfogui");
      println("[fcrr-sim:] *dtripinfogui* focussed.")[artifact_id(LocalConsoleID)];
      println("[fcrr-sim:] Waiting for start trip...")[artifact_id(LocalConsoleID)].

+start(TripID) : art_id(console,LocalConsoleID)
  <-  println("[fcrr-sim:] Start trip button pressed")[artifact_id(LocalConsoleID)];
```

```
      focusWhenAvailable("gps-simulator");
      lookupArtifact("gps-simulator", GpsSimID);
      +tripstatus("incorso");
      !go.

+stop(TripID) : art_id(console,LocalConsoleID)
  <-  println("[fcrr-sim:] Stop trip button pressed")[artifact_id(LocalConsoleID)];
      -tripstatus("incorso").

+!go : tripstatus("incorso")
  <-  move;
      println("[fcrr-sim:] New gps position created")[artifact_id(LocalConsoleID)];
      .wait(5000);
      !go.
```

Tabella 4.12: Frammento codice dell'agente Jason **Simulator**.

Capitolo 5

Conclusioni

Le finalità di questa tesi possono essere riassunte in due punti principali. Il primo punto riguarda la definizione di un modello innovativo di Ridesharing (condivisione dei viaggi mediante l'uso di mezzi di trasporto privati) basato su un sistema finanziario a crediti chilometrici e denominato Fixed-Cost Real-time Ridesharing, il secondo punto concerne l'ingegnerizzazione prototipale di una piattaforma software di supporto al suddetto modello.

La trattazione segue un iter la cui la fase iniziale sviluppa la presentazione del modello *FCRR* evidenziando come le caratteristiche introdotte possano rafforzare i punti deboli emersi dallo studio dei modelli e delle piattaforme di Ridesharing esistenti ad oggi. La seconda fase si sviluppa nella definizione e nell'analisi dell'insieme dei requisiti e quindi delle funzionalità che la piattaforma software di supporto deve realizzare terminando con una analisi del problema che fornisce le conoscenze utili a delineare il quadro generale del sistema ed i suoi vincoli. La terza fase del lavoro sviluppa la progettazione del sistema motivando le opportune scelte progettuali e fornendo una prima implementazione delle parti più significative del software.

I risultati ottenuti da questo lavoro di tesi dimostrano come le attuali tecnologie mobile, grazie alle funzionalità di geolocalizzazione ed all'accesso alla rete internet, possano favorire lo sviluppo di sistemi come il *FCRR* utili alla diffusione di realtà importanti per la collettività quali il Ridesharing. Realtà che ad oggi costituiscono una delle armi strategiche nell'organizzare e sviluppare una mobilità sostenibile

affrontando i problemi in ambito economico ed ecologico-ambientale legati all'uso sconsiderato dei mezzi di trasporto privati. Inoltre, un risultato sorprendente è il dato che emerge dalla scelta di approcciare l'ingegnerizzazione di tali sistemi ad un alto livello di astrazione come nel caso del paradigma agent-oriented. Tale approccio permette di passare direttamente da una analisi task-oriented delle funzionalità ad una progettazione agent-oriented del sistema strutturandolo in termini di entità attive (agenti) e passive (artefatti) ed interazioni che intercorrono tra di esse.

Questo lavoro di tesi apre a diverse possibilità di eventuali sviluppi futuri:

- Il completamento dell'implementazione del sistema *FCRR*, includendo la FCRR Desktop Web Application;
- L'estensione della piattaforma includendo funzionalità avanzate come il pooling a tappe (*multiple drivers per trip*);
- Lo sviluppo della FCRR Mobile Application per le restanti piattaforme operative in ambito mobile;
- L'integrazione della piattaforma nei sistemi di navigazione per auto;
- Pianificazione strategica e business plan per il lancio sul mercato del software *FCRRS*.

Bibliografia

[1] A. Amey, J. Attanucci, D. Veneziano, and J. Ferreira. *Real-Time Ridesharing: Exploring the Opportunities and Challenges of Designing a Technology-based Rideshare Trial for the MIT Community*. PhD thesis, Massachusetts Institute of Technology, Dept. of Civil and Environmental Engineering; and, (MCP)–Massachusetts Institute of Technology, Dept. of Urban Studies and Planning, 2010.

[2] Avego. Real-time ridesharing platform. `http://www.avego.com/`, 2012.

[3] R. Bordini and J. Hübner. Bdi agent programming in agentspeak using jason. *Computational logic in multi-agent systems*, pages 143–164, 2006.

[4] R. Bordini, J. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. Wiley-Interscience, 2008.

[5] E. Bruno. Soa, web services, and restful systems-representational state transfer, or rest for short, is a less restrictive form of soa than web services. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, pages 32–37, 2007.

[6] N. Chan and S. Shaheen. Ridesharing in north america: Past, present, and future. 2011.

[7] R. Fielding and R. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[8] S. Hartwig and M. Buchmann. Empty seats traveling: next-generation ridesharing and its potential to mitigate traffic-and emission problems in the 21st century. 2006.

[9] iCarpool. Real-time ridesharing platform. `http://www.icarpool.com`, 2012.

[10] Massachusetts Institute of Technology. Real-time rideshare research page. `http://ridesharechoices.scripts.mit.edu/home/`, 2010.

[11] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.

[12] A. Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. *Agents Breaking Away*, pages 42–55, 1996.

[13] A. Rao, M. Georgeff, et al. Bdi agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*, pages 312–319. San Francisco, 1995.

[14] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming in cartago. *Multi-Agent Programming:*, pages 259–288, 2009.

[15] A. Ricci, M. Viroli, and A. Omicini. The a&a programming model and technology for developing agent environments in mas. *Programming multi-agent systems*, pages 89–106, 2008.

[16] A. Santi, M. Guidi, and A. Ricci. Jaca-android: an agent-based platform for building smart mobile applications. *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, pages 95–114, 2011.

[17] Wikipedia. Car pooling page. `http://en.wikipedia.org/wiki/Car_pooling`, 2012.

[18] Wikipedia. Real-time ridesharing. `http://en.wikipedia.org/wiki/Real-time_ridesharing`, 2012.

[19] Wikipedia. **H**igh-**O**ccupancy **V**ehicle lane. `http://en.wikipedia.org/wiki/High-occupancy_vehicle`, 2012.

[20] Zimride. Ridesharing platform. `http://public.zimride.com/`, 2012.