

ALMA MATER STUDIORUM · UNIVERSITÀ DI
BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea Triennale in Informatica

**Un Framework di analisi
e di servizi innovativi
per la mobilità veicolare elettrica**

Tesi di Laurea in Laboratorio di Applicazioni Mobili

Relatore:
Chiar.mo Prof.
Luciano Bononi

Presentata da:
Simone Rondelli

Correlatori:
Chiar.mo Prof. Tullio Salmon
Cinotti
Dott. Marco Di Felice
Dott. Luca Bedogni

Sessione III
Anno Accademico 2012/2013

Sommario

Abstract L^AT_EX.

Abbiamo la testa rotonda per pensare in tutte le direzioni.

Francis Picabia - (1922)

Indice

1	Introduzione	5
1.1	Smart Cities	5
1.2	Electrical Mobility	5
1.3	Internet of Energy	8
1.3.1	Artemis	8
1.3.2	Far progredire il mondo dell'energia	8
1.3.3	Mobilità/software	8
1.4	Lavori Correlati	8
1.5	Un po di storia	8
2	Architettura	9
2.1	Smart-M3	9
2.1.1	Semantic Information Broker	10
2.1.2	I Knowledge Processor	11
2.1.3	Le triple RDF	12
2.1.4	Ontologie	13
2.1.5	Sottoscrizioni	13
2.1.6	SPARQL	13
2.1.7	Il protocollo SSAP	14
2.2	Il Modello Ontologico	15
2.2.1	Introduzione	15
2.2.2	Classi di IoE	16
2.2.3	Sottoclassi di owl:Thing	16
2.2.4	Sottoclassi di ioe:Data	19
2.2.5	Modifiche apportate all'ontologia	20

2.3	I Semantic Information Broker	25
2.3.1	City SIB	26
2.3.2	Dash SIB	26
2.4	La libreria IoE	27
2.4.1	Connessione al SIB	27
2.4.2	Entities	29
2.4.3	Controller	30
3	Servizio Cittadino	33
3.1	Architettura	33
3.2	La comunicazione con il City Service	33
3.2.1	Protocollo di richiesta di prenotazione	34
3.3	Il Protocollo di rimozione di una prenotazione	35
3.4	Implementazione	36
3.4.1	Pool Di Oggetti	36
3.4.2	Pool Di Thread	38
3.4.3	Il funzionamento del City Service	39
3.5	Testing	43
3.5.1	Test Protocolli	44
3.5.2	Valutazione Performance	45
4	Applicazione Mobile	47
4.1	Architettura	47
4.1.1	Android	48
4.1.2	Blue&Me	50
4.1.3	Profilo Altimetrico e Contributo Energetico	51
4.2	Modalità di esecuzione	51
4.2.1	Simulazione	52
4.2.2	Con Blue&Me	52
4.2.3	Senza Blue&Me	52
4.3	Funzionalità	54
4.3.1	Il menu principale	54
4.3.2	Storia delle ricariche effettuate	54
4.3.3	Monitoraggio Parametri Batteria	54

4.3.4	Effettuare una richiesta di prenotazione	55
4.3.5	Visualizzazione e Ritiro delle Prenotazioni	58
4.3.6	Profilo Altimetrico e Contributo Energetico	58
4.3.7	Mappa	60
4.4	Notifica batteria Scarica	62
4.5	Implementazione	63
4.5.1	Implementazione	63
4.5.2	Operazioni Asincrone	63
4.5.3	Activities	63
4.5.4	Servizi	63
5	Piattaforma di Simulazione	65
5.1	Architettura	65
5.1.1	SUMO	65
5.1.2	OMNeT++	67
5.1.3	Veins	69
5.2	Modellazione della Simulazione	70
5.2.1	Ciclo di Vita dei moduli	72
5.2.2	CityService	73
5.2.3	CarLogic	75
5.2.4	Battery	79
5.2.5	DriverBeahviour	79
5.3	Implementazione	79
5.3.1	Logging	80
5.3.2	SibController	80
5.3.3	GcpController	80
5.3.4	Utility	80
5.4	L'ambiente di simulazione	80
5.4.1	Generazione file di Configurazione	81
5.4.2	Download Scenario	81
5.4.3	Generazione XML di SUMO	82
A	Installazione Ambiente	85
A.1	Installazione	85

A.1.1	Installazioni preliminari	85
A.1.2	OMNeT++	86
A.1.3	SUMO	86
A.1.4	SMART-M3	87
A.1.5	KPI_Low	87
A.1.6	Importare il progetto in OMNeT++	88
B	Vista al Centro Ricerche Fiat	91
C	UniboGeoTools	93
D	OntologyLoader	95

Capitolo 1

Introduzione

1.1 Smart Cities

Citta intelligenti

1.2 Electrical Mobility

Al giorno d'oggi l'Electrical Mobility (EM) è considerata una degli elementi chiave per ridurre l'inquinamento e al contempo liberarsi dalla dipendenza dai combustibili fossili. Questo sta portando a ingenti investimenti da parte di governi e delle industrie automobilistiche.

Nel breve periodo il mercato legato all'EM è destinato a crescere rapidamente come conseguenza dell'incremento della varietà di Veicoli Elettrici (EVs) introdotti dalle Case Automobilistiche. Secondo recenti studi infatti il numero di EVs venduti nel periodo tra il 2010 e il 2012 è aumentato del 200%. Nonostante il crescente interesse nei confronti dell'EM, recenti analisi di mercato dimostrano che i benefici ad essa legati saranno tangibili soltanto nel lungo termine questo è confermato da una ricerca condotta dal U.S. National Energy Technology secondo cui il 70% delle persone non comprerà un EV a causa dell'incertezza sulla disponibilità delle stazioni di ricarica. A questo si vanno ad aggiungere le ben note problematiche riguardanti la capacità, la durata delle batterie e i tempi di ricarica estremamente lunghi (nell'ordine delle decine di minuti).

Da un lato la durata dei tempi di ricarica, la limitata capacità delle batterie e la disposizione degli Electric Vehicle Supply Element influisce direttamente sull'esperienza di guida di ogni autista e può avere un impatto decisivo sulla penetrazione di mercato dei veicoli elettrici. D'altra parte diversi studi hanno dimostrato che l'impatto sulla rete energetica causato dalla ricarica simultanea di molti Veicoli Elettrici può avere ripercussioni negative e si è quindi delineata la necessità di coordinare le attività tra EVs ed EVSEs.

Molti progetti Europei sono stati avviati con lo scopo di limitare queste problematiche. Allo stesso tempo bisogna considerare che un uno scenario realistico di EM ci sono diverse parti interessate (es: autisti, case automobilistiche, produttori di energia) coinvolte nella gestione dell'EM. La ricerca si è mobilitata in direzione dell'Information and Communication Technology (ICT) per fornire servizi di supporto all'EM e permettere alle parti interessate di cooperare in modo intelligente. Sebbene siano state sviluppate diverse applicazioni su scenari in piccola scala, si è ancora lontani dall'ottenere l'interoperabilità tra gli attori in gioco i quali utilizzano diverse tecnologie e dispositivi.

Dato l'elevato costo che avrebbero i test su larga scala, la simulazione costituisce lo strumento più adatto per testare l'efficenza delle soluzioni ICT prima che vengano realmente sviluppate. Al giorno d'oggi sono stati sviluppati alcuni simulatori veicolari che permettono un controllo molto fine a livello di veicolo e similarmante altrettanti modelli di batteria sono stati create al fine di riprodurre in modo realistico le dinamiche di carica e scarica della batteria. Tuttavia nessuno di questo strumenti è adatto al fine di studiare le dinamiche assai complesse che si presentano nello scenario dell'EM, come l'impatto degli EV sulla rete elettrica cittadina oppure l'effettiva utilità dell'utilizzo di sistemi di prenotazione delle ricariche.

Il progetto Internet of Energy (IoE) for Electrical Mobility, fondato dall'Unione Europea e comprendente 40 partner da 10 nazioni Europee, mira a colmare queste lacuna, sviluppando hardware, software e sistemi middleware che forniranno un infrastruttura di comunicazione interpolabile tra le parti in gioco all'interno della (? dire due parole a riguardo smart grid).

Lo scopo di questa tesi, frutto del lavoro congiunto tra UNIBO e ARCES, seguito della tesi di laurea di Federico Montori, è fornire contributi su tre

diversi fronti a questo progetto.

Inanzitutto abbiamo sviluppato un architettura software con lo scopo di fornire servizi per l'interazione tra gli EVs ed EM attraverso lo smartphone. Il servizio centrale è il City Service (CS) il quale si prende a carico le richieste di ricarica, che arrivano dagli smartphone, fornendo la lista degli EVSEs disponibili che più si adattano alle esigenze dell'utente. Il modello di dati usato dal servizio si basa su un'ontologia che rappresenta tutte le informazioni relative alla smart-grid. Le informazioni sono condivise attraverso un repository semantico chiamato Semantic Information Broker (SIB), esso garantisce, grazie all'ontologia, un'interazione uniforme tra i vari componenti del sistema.

In secondo luogo abbiamo creato un'applicazione mobile che permette all'utente di monitorare i parametri della batteria del veicolo e di prenotare slot di tempo presso gli EVSE grazie all'interazione con la SIB cittadina. Il servizio di prenotazione dà la possibilità di scegliere in base a vari parametri come il prezzo, la distanza, il contributo energetico necessario a raggiungere l'EVSE e il tempo totale di ricarica.

Infine abbiamo creato una piattaforma di simulazione integrata che permette di valutare su larga scala l'impatto della EM. Diversamente ad altri tool già presenti il nostro framework permette di studiare il comportamento degli EV, con relativo modello di carica e scarica della batteria, insieme all'interazione di essi con la smart grid attraverso gli EVSE. A questo proposito sono stati usati diversi tool tra i quali SUMO, un "simulatore di traffico microscopico", OMNET++, un simulatore a eventi discreti, e infine per far comunicare i due simulatori viene usata l'interfaccia TRACI, messa a disposizione da SUMO, la quale comunica con OMNET++ attraverso Veins. Grazie a SUMO riusciamo a modellare l'ambiente urbano, nel nostro caso Bologna e Torino, includendo dati topografici e altimetrici realistici. OMNET++ invece è stato utilizzato per implementare i modelli dell'EV, compresa la batteria e il comportamento dell'autista.

1.3 Internet of Energy

1.3.1 Artemis

1.3.2 Far progredire il mondo dell'energia

1.3.3 Mobilità/software

1.4 Lavori Correlati

1.5 Un po di storia

DA LASCIARE??? a me *MI* piace

L'applicazione mobile è stato il cavallo di troia con il quale ho avuto il privilegio di partecipare a questo progetto. Era il 19/07/2012 quando ho inviato al Prof *Luciano Bononi* la richiesta di progetto per il corso di *Laboratorio di Applicazioni Mobili*. Non potevo immaginare che quella email mi avrebbe portato ad un lavoro durato oltre un anno e che dura tutt'ora. Al colloquio per l'assegnazione del progetto mi venne presentato l'opportunità di rendere più accattivante l'applicazione mobile creata da Federico Montori per il suo progetto di laurea. Questo perché, visto il poco tempo che egli aveva potuto dedicarci, era ancora in fase embrionale. Così, in seguito a qualche meeting, con i vari componenti del progetto e notevoli dosi di pazienza da parte di *Federico Montori*, che mi ha introdotto al progetto, sono riuscito a disporre un ambiente più o meno funzionante. La simulazione *crashava* dopo un secondo ma tanto bastava per introdurre un veicolo ed eseguire i test con la nuova applicazione mobile. Successivamente, capendo la vastità del progetto che stava dietro all'applicazione, decisi di farlo diventare progetto di laurea poiché era evidente che c'era ancora molto lavoro da fare e la mia innata capacità di complicarmi la vita ha giocato un ruolo fondamentale in tutto questo.

Capitolo 2

Architettura

In questo capitolo verranno descritte le scelte architetturali e implementative che stanno alla base del contributo di questa tesi al progetto IoE.

Lo scenario legato alla mobilità elettrica veicolare è caratterizzato dalla presenza di diversi domini applicativi, piattaforme e parti interessate i quali necessitano di comunicare in modo unificato e trasparente. A tal fine è stato utilizzato il progetto Smart-M3 ([9]), cuore della nostra architettura. Appoggiandosi sulle tecnologie tipiche del *Semantic Web*, Smart-M3 assicura l'interoperabilità tra gli attori in gioco.

In particolare vedremo come possono coesistere elementi reali ed elementi simulati e come il passaggio dall'uno all'altro sia assolutamente trasparente a tutti i componenti del sistema grazie all'uso di tecnologie ontology-based.

2.1 Smart-M3

Prima di parlare dei componenti strettamente legati a questo progetto è doveroso fare un introduzione alla tecnologia che fa da collante tra di essi, ovvero Smart-M3. Capire come funziona Smart-M3 e quali sono i suoi principi è fondamentale per comprendere a fondo il seguito di questo documento.

M3 è un architettura middleware che realizza l'interoperabilità delle informazioni in maniera cross-domain, multi-vendor, multi-device, multi-piattaforma ([3]). Smart-M3 è la sua prima implementazione Open Source, proposta da SOFIA, un Progetto Europeo (2009-11) appartenente al framework ARTE-

MIS. La piattaforma implementa il disaccoppiamento tra produttori e consumatori di informazione. In questa architettura tutti gli attori (sensori, dispositivi, servizi, attuatori ecc..) cooperano attraverso un database RDF che è lo standard deciso dal World Wide Web Consortium per la descrizione di informazioni e concetti. L'interoperabilità è resa possibile da un modello di dati condiviso che si basa su tecnologie tipiche del Semantic Web.

Il Semantic Web è un framework sviluppato dal World Wide Web Consortium per consentire la condivisione e il riutilizzo dei dati attraverso applicazioni, aziende e comunità eterogenee.

La figura 2.1 mostra il funzionamento dell'architettura M3. Il “legacy gate” è un interfaccia con il mondo esterno e possono coesisterne molteplici istanze in un architettura M3.

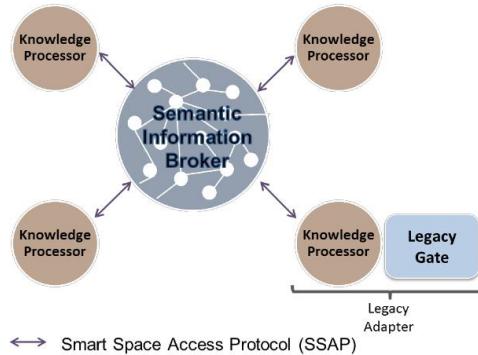


Figura 2.1: Architettura Smart-M3

2.1.1 Semantic Information Broker

Il *Semantic Information Broker* (SIB) è l'entità responsabile della conservazione e della gestione delle informazioni condivise nell'architettura M3. Gli agenti Software che si scambiano le informazioni vengono chiamati *Knowledge Processors* (KP). L'accesso alla SIB da parte dei KP avviene attraverso lo *Smart Space Access Protocol* (SSAP) basato su messaggi XML scambiati attraverso socket TCP/IP. Vengono fornite API che implementano il protocollo SSAP in diversi linguaggi.

Il SIB è un architettura a 5 livelli ([6]) come mostrato in figura 2.2a:

1. **Transport:** Gestisce una o più comunicazioni di rete a livello di trasporto, permettendo al SIB di comunicare con diverse reti e architetture. Il livello di trasporto è collegato a quello sottostante tramite il DBus, rendendo possibile l'aggiunta e la rimozione di connettori a runtime.
2. **Operation Handling:** Gestisce le diverse operazioni del protocollo SSAP ognuna delle quali viene eseguita in un thread dedicato. Malgrado l'uso intensivo di thread possa degradare le performance, è stata ritenuta determinante la chiarezza di codice che ne consegue.
3. **Graph Operations:** Gestisce le operazioni di inserimento, rimozione e query dal database RDF come richiesto dal livello 2. Viene eseguito all'interno di un singolo thread che schedula ed esegue le richieste provenienti dai thread che gestiscono le operazioni SSAP la cui comunicazione avviene tramite code asincrone.
4. **Triple Operations:** Gestisce le operazioni SPARQL, WQL e le query basate su pattern-matching di triple RDF. Attualmente è implementato tramite Piglet, un database RDF che si appoggia ad SQL lite per la persistenza delle informazioni. Lo strato può essere tranquillamente cambiato a patto che si scriva il codice necessario ad interfacciare le operazioni a livello di grafo (3) con l'interfaccia fornita dal nuovo store RDF.
5. **Persistent storage:** Assicura la persistenza dei dati.

2.1.2 I Knowledge Processor

I Knowledge Processor (KP) sono le parti attive dell'architettura Smart-M3. Un KP interagisce con il SIB non direttamente tramite il protocollo SSAP ma tramite le Knowledge Processor Interface (KPI) ovvero le librerie che lo implementano. Queste possono trovarsi a qualunque livello di astrazione ed essere scritte in qualunque linguaggio. Le funzioni messe a disposizione dal KPI in genere sono speculari alle operazioni del protocollo SSAP.

I KP sono le entità che forniscono, modificano e richiedono le informazioni le informazioni contenute nello smart-space. L'architettura dei KP è mostrata in figura 2.2b.

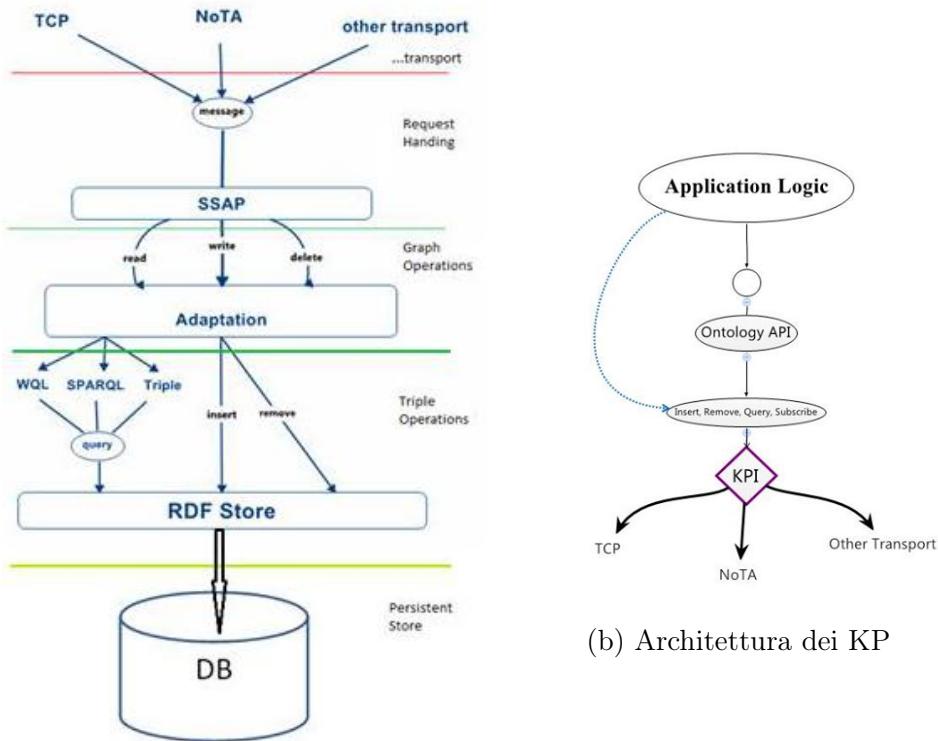


Figura 2.2: Architetture SIB e KP

2.1.3 Le triple RDF

Nell'architettura Smart-M3 le informazioni sono rappresentate in formato RDF (Resource Description Framework). In RDF le informazioni sono rappresentate in forma di triplete *soggetto, predicato, oggetto*. Le triple vengono memorizzate nel SIB e formano un grafo etichettato diretto che non necessariamente è un grafo connesso.

2.1.4 Ontologie

Mentre RDF fornisce il modello di dati standard per la rappresentazione delle informazioni, l'uso di un linguaggio ontologico è indispensabile per assegnare una semantica all'informazione. Linguaggi ontologici come RDFS e OWL forniscono un vocabolario comune. L'uso di una ontologia comune consente a tutti gli attori (uomini e macchine) di capire reciprocamente la semantica delle informazioni e di cooperare in simbiosi attraverso il SIB. Smart-M3 è agnostico rispetto all'ontologia e quindi consente agli sviluppatori di scegliere il modo migliore di modellare le informazioni per soddisfare le esigenze funzionali del dominio applicativo indirizzato.

2.1.5 Sottoscrizioni

Un aspetto fondamentale di questa tecnologia è il meccanismo delle sottoscrizioni grazie al quale è possibile ricevere notifiche al variare di set di triple. Le sottoscrizioni sono determinanti nella nostra architettura perché, come vedremo più avanti (Sez. 3.2), sono alla base dei protocolli di scambio dati tra i componenti del sistema.

2.1.6 SPARQL

SPARQL (pronuncia sparkle, acronimo ricorsivo di SPARQL Protocol and RDF Query Language) è il linguaggio standard de facto per interrogare dataset RDF. Come si può dedurre dal nome stesso, SPARQL non è semplicemente un linguaggio di interrogazione di dati RDF ma definisce anche il protocollo applicativo utilizzato per comunicare con le sorgenti RDF (si tratta di un binding su HTTP).

Così come SQL riflette nella rappresentazione della query il modello relazionale sottostante, allo stesso modo SPARQL basa la rappresentazione della query sul concetto di tripla e di grafo. Il meccanismo alla base della rappresentazione di una query e della ricerca della sua risposta è il graph matching. La query rappresenta un pattern di un grafo (RDF) e la risposta alla query sono tutte le triple (sotto-grafo) che *fanno match* con il pattern.

2.1.7 Il protocollo SSAP

L'SSAP (*Smart Space Access Protocol*) è il protocollo con cui si comunica con il SIB. Essendo un protocollo session-based, i KP che vogliono comunicare con lo smart-space dovranno prima aderirvi con un'operazione di Join prevista dall'SSAP. Il KP fornisce le sue credenziali nel messaggio di Join, il SIB esamina le credenziali e decide se accettare il KP o meno. Dopo l'operazione di Join, il KP può eseguire le altre operazioni. L'SSAP è il punto di integrazione principale dell'architettura Smart-M3. Le implementazioni di SIB e KP devono implementare tutte le operazioni del protocollo SSAP per garantire l'interoperabilità.

Le operazioni supportate dal protocollo SSAP sono:

- **JOIN:** Associa il KP allo smart-space solo se le credenziali vengono ritenute valide. Determina l'inizio della sessione.
- **LEAVE:** Determina il termine dell'associazione con lo smart-space e quindi la fine della sessione. Da questo momento in poi non potranno essere eseguite altre operazioni di associazione allo smart-space.
- **INSERT:** Operazione atomica di inserzione di un Grafo, formato da triple RDF, nel SIB.
- **REMOVE:** Operazione atomica di rimozione di un Grafo, formato da triple RDF, nel SIB.
- **UPDATE:** Operazione atomica di aggiornamento di un Grafo, formato da triple RDF, nel SIB. In realtà si tratta di una combinazione di DELETE e INSERT eseguita in modo atomico con precedenza dell'operazione di DELETE.
- **QUERY:** Richiesta di informazioni contenute nel SIB attraverso una delle modalità supportate.
- **SUBSCRIBE:** Sottoscrizione a un set di triple contenute nel SIB. Il KP riceve una notifica quando avviene un cambiamento su una di queste triple.
- **UNSUBSCRIBE:** Cancella una sottoscrizione.

2.2 Il Modello Ontologico

In questa sezione cercherò di illustrare come sono stati modellati i dati attraverso una ontologia. Ho ereditato dalla progetto di Tesi di *Federico Montori* ([8]). Io ho contribuito espandendolo per adattarlo ai nuovi requisiti funzionali sorti durante lo sviluppo del progetto. Verranno quindi mostrati gli aspetti dell'ontologia necessari per comprendere il resto della trattazione e verranno approfondite le modifiche da me apportate.

2.2.1 Introduzione

L'ontologia è definibile come una rappresentazione formale ed esplicita di una concettualizzazione condivisa di un dominio di interesse.

L'ontologia presenta le seguenti proprietà:

- **Rappresentazione Formale:** come tale usa un linguaggio logico processabile da elaboratori.
- **Esplicita:** cioè non ambigua e tale da chiarire ogni assunzione fatta.
- **Concettuale:** è una concettualizzazione cioè una vista astratta e semplificata del dominio di interesse.
- **Condivisa:** determinata dal consenso di una pluralità il più ampia possibile di soggetti.

Lo scopo delle ontologie è quindi descrivere delle basi di conoscenze, effettuare delle deduzioni su di esse e integrarle tra le varie applicazioni. Per descrivere le ontologie viene utilizzato il linguaggio OWL (*Ontology Web Language*), estensione di RDF. È un linguaggio di markup per rappresentare esplicitamente significato e semantica di termini con vocabolari e relazioni tra gli stessi.

I linguaggi della famiglia OWL sono in grado di creare *classi*, *proprietà*, *istanze* e le relative *operazioni*.

2.2.2 Classi di IoE

Una classe è una collezione di oggetti che corrisponde alla descrizione logica di un concetto. Da una classe si possono creare un numero arbitrario di istanze mentre ad un'istanza possono corrispondere una, nessuna o molteplici classi.

Una classe può essere sottoclasse di un'altra classe, ereditando le caratteristiche della super-classe. Tutte le classi sono sottoclassi di **owl:Thing**, rappresentazione concettuale di “cosa”.

Nel modello di dati utilizzato in questo progetto si è cercato di tenere disaccoppiato il concetto di dato dalle altre entità fisiche. Ne consegue che tutte le entità fisiche sono sottoclassi dirette di **owl:Thing**, mentre le classi destinate a rappresentare i dati sono sottoclassi di **ioe:Data** che a sua volta è sottoclasse di **owl:Thing**.

Nel resto di questo documento userò il prefisso **ioe:** come abbreviazione di *http://www.m3.com/2012/05/m3/ioe-ontology.owl#* il namespace scelto per l'ontologia. In generale userò questo prefisso per distinguere le classi dell'ontologia dalle classi Java che come vedremo nella sezione 2.4 hanno lo stesso nome essendo mapping diretto di quest'ultime.

2.2.3 Sottoclassi di owl:Thing

Come già accennato tutte le entità fisiche del nostro modello ontologico sono sottoclassi dirette di **owl:Thing**. Quella mostrata di seguito è una lista delle Classi usate in questo progetto omettendo quelle attualmente irrilevanti o inutilizzate.

ioe:Person: rappresenta il concetto di persona. Ad ogni persona possono essere associati diversi veicoli (**ioe:Vehicle**), diverse richieste di ricarica (**ioe:Reservation**) nonché la storia delle ricariche effettuate (**ioe:Recharge**). Il concetto di persona viene usato ai fini dell'autenticazione e in un futuro potrà essere determinante ai fini della fatturazione che il provider energetico eseguirà a fronte delle ricariche.

ioe:Vehicle: rappresenta il concetto di Veicolo Elettrico. I veicoli NON elettrici sono infatti irrilevanti al fine di questa trattazione. Ad ogni vei-

colo sono ovviamente associati i dati della batteria (**ioe:BatteryData**) che verranno trattati nella sezione relativa alle sottoclassi di **ioe:Data** (2.2.4);

ioe:Zone: rappresenta l'area di copertura del *City Service*. È definita da un rettangolo individuato dagli angoli nord-ovest e sud-est.

ioe:GridConnectionPoint: il *Grid Connection Pointer* (**ioe:GCP**) è la stazione di ricarica. Contiene almeno un EVSE, rappresentazione della colonnina dove avviene la ricarica effettiva. Il rapporto tra un GCP e gli EVSE è lo stesso che intercorre tra una stazione di rifornimento e le pompe di benzina.

ioe:EVSE: Il *Electrical Vehicle Supply Equipment* è il punto in cui il veicolo si connette alla rete elettrica. Una volta connesso può sia ricaricare la sua batteria che cedere energia alla Grid. Un EVSE ha diversi connettori (**ioe:Connector**) per adattarsi ai vari tipi di presa in dotazione ai veicoli elettrici. Inoltre ogni EVSE dispone di una lista di prenotazioni associate.

ioe:ChargeProfile: è l'insieme dei parametri che caratterizzano il profilo energetico di un EVSE in un determinato istante. I parametri attualmente sono: potenza, orario di validità del profilo stesso e prezzo per unità di energia (in genere 1 kWh). Può essere attivo un solo **ioe:ChargeProfile** alla volta, variabile optionalmente in base a fasce orarie analogamente a quanto avviene per l'energia elettrica casalinga.

ioe:Connector: è il connettore di ricarica ovvero il tramite tra l'EVSE e l'EV. Ogni EVSE può avere diversi connettori per la massima compatibilità col maggior numero di veicoli possibile. Malgrado negli USA si stia cercando di introdurre uno standard, esistono ormai diversi tipi di connettori.

ioe:ChargeRequest: Richiesta di ricarica. Viene istanziata quando un utente deve creare una prenotazione. Al suo interno sono contenuti tutti i parametri necessari a descriverla. Fa parte del protocollo di

richiesta di prenotazione discusso nella Sez. 3.2. Un approfondimento sulla sua struttura è trattato nella Sez. 2.2.5.

ioe:ChargeResponse: è la risposta fornita dal servizio cittadino a seguito della richiesta di prenotazione. Al suo interno contiene un riferimento alla richiesta (**ioe:ChargeRequest**) da cui è stata generata e una lista di opzioni di ricarica conformi alla richiesta dell'utente (**ioe:ChargeOption**).

ioe:ChargeOption: Fa parte della risposta(**ioe:ChargeResponse**) che il servizio cittadino fa all'utente in seguito a una richiesta di prenotazione (**ioe:ChargeRequest**). Contiene i parametri di ricarica quali EVSE, orario e prezzo.

ioe:Currency: rappresenta la divisa monetaria relativa a un prezzo. Alcune sue istanze sono state inserite direttamente nell'ontologia (**ioe:Euro**, **ioe:Dollar** ecc..);

ioe:Reservation: viene creata un'istanza di questa classe se il protocollo di richiesta di prenotazione va a buon fine. Indica che l'EVSE a cui è associata è impegnato per un certo lasso di tempo.

ioe:ReservationList: lista di prenotazioni associate ad un EVSE. Ogni EVSE può avere un'unica lista di prenotazioni associata.

ioe:ReservationRetire: classe che denota la volontà dell'utente di ritirare una prenotazione.

ioe:Recharge: quando un veicolo, in seguito a una prenotazione, termina di ricaricarsi, viene inserita questa entità ad esso associata. Sancisce l'avvenuta ricarica è può essere utile per tener traccia dell'attività dell'utente nonché per fare statistiche.

ioe:UnityOfMeasure: rappresenta l'unità di misura dei dati del progetto. Deve esserne associata una ad ogni sottoclasse di **ioe:Data**. Attualmente sono *hardcoded* all'interno dell'ontologia (**ioe:Watt**, **ioe:Volt** ecc..)

ioe:Data: rappresenta il concetto di dato misurabile e ogni sua sottoclassse è caratterizzata da un valore e da un unità di misura.

2.2.4 Sottoclassi di ioe:Data

La sezione descrive la lista delle tipologie di dati usate nel progetto. Sono tutte sottoclassi di **ioe:Data** caratterizzate da un valore e da unità di misura **ioe:UnityOfMeasure**. Le unità di misura associate ai dati mostrati nel seguito sono quelle utilizzate nell'ambito di questo progetto ma nulla vieta di cambiarle.

ioe:BatteryData: raggruppa i dati relativi allo stato della batteria di un veicolo (**ioe:ChargeData**, **ioe:VoltageData**, **ioe:PowerData**, **ioe:CurrentData**, **ioe:TempertureData**)

ioe:ChargeData: rappresenta la quantità di carica misurata in Kilowattora (kWh).

ioe:VoltageData: rappresenta la tensione elettrica misurata in Volt (V).

ioe:PowerData: rappresenta la potenza elettrica misurata in Kilowatt (kW).

ioe:CurrentData: rappresenta l'intensità di corrente misurata in Ampere (A). Indica sia la corrente in uscita che quella in entrata, come ad esempio i cicli di carica e scarica della batteria.

ioe:TempertureData: rappresenta la temperatura misurata in Gradi Celsius (°C). Attualmente non è considerata nei nostri modelli.

ioe:LocationData: rappresenta i dati geografici dell'entità a cui si riferisce (es: posizione del veicolo). In realtà nel progetto è sostituita dall'uso diretto della sua sottoclassse **ioe:GPSData**.

ioe:GPSData: rappresenta le coordinate GPS ovvero latitudine e longitudine misurate in Gradi Angolari.

ioe:SpatialRangeData: Rappresenta uno spazio geografico determinato da un punto GPS e da un raggio intorno ad esso. Si misura in Metri (m).

ioe:PriceData: Rappresenta le informazioni relative a un prezzo qui è associata una divisa monetaria (sezione 2.2.3)

ioe:TimeIntervalData: Rappresenta un intervallo che originariamente era compreso tra due date nella forma gg/mm/aaaa mentre attualmente si misura in Millisecondi (ms).

2.2.5 Modifiche apportate all'ontologia

La versione dell'ontologia su cui ho iniziato a lavorare era la 1.5.4 alla quale sono seguiti 10 successivi rilasci fino all'attuale release 1.6.2. Le modifiche più importanti hanno riguardato il supporto di un nuovo protocollo di prenotazione delle ricariche oltre a operazioni varie operazioni di refactoring e correzione di incongruenze.

Il concetto di Utente

Inizialmente il concetto di utente non era previsto nell'ontologia in quanto non trovava nessuna applicazione pratica. L'entità che interagiva con la Grid era il veicolo e non l'utente. Questo approccio evidenzia i suoi limiti nel caso in cui un utente possieda più veicoli e voglia monitorarne contemporaneamente le ricariche effettuate o le prenotazioni pendenti. Inoltre anche dal punto di vista dei fornitori di corrente elettrica può essere utile una visione a livello di utente per semplificare le operazioni di fatturazione. Successivamente è stata introdotta la necessità di autenticare gli utenti per poter cifrare le comunicazioni con il SIB.

È stata quindi introdotta la classe **ioe:Person**. Seppur esistano già delle ontologie con classi che rappresentative di questo concetto, si è ritenuto più semplice crearne uno nostro. Sviluppi futuri potrebbero legare questo concetto ad uno già esistente per rendere più semplice l'interoperabilità tra sistemi diversi. La classe presenta le seguenti proprietà:

- **ioe:hasName:** Nome e Cognome dell’utente in formato stringa. Attualmente è irrilevante avere una separazione dei due.
- **ioe:hasUserIdentifer:** codice che identifica univocamente l’utente.
- **ioe:hasVehicle:** proprietà associa a un utente uno o più veicoli.

Come si può vedere attualmente l’utente è modellato in modo molto primitivo. Sono state infatti incluse solo le proprietà strettamente necessarie al nostro ambito di interesse.

Il concetto di Veicolo

Nel vecchio modello ontologico il concetto di veicolo, in quanto non essenziale, non veniva particolarmente enfatizzato. Nella ultime versioni dell’ontologia ho aggiunto al veicolo alcune proprietà per soddisfare all’esigenza di distinguere le diverse possibili provenienze dei veicoli. Infatti i veicoli vengono inseriti nel SIB dal simulatore e possono anche essere reali come nel caso del Fiat Daily provato al CRF (App. B).

Il veicolo è attualmente definito dalle seguenti proprietà:

- **ioe:hasManufacturer:** Casa automobilistica che produce il veicolo.
- **ioe:hasModel:** modello del veicolo.
- **ioe:hasGPSData:** la proprietà punta ad un’istanza di **ioe:GPSData** che a sua volta contiene le informazioni di latitudine e longitudine.
- **ioe:hasBatteryData:** la proprietà punta ad un’istanza di **ioe:BatteryData** contenente i dati della batteria.
- **ioe:hasIdentificationData:** identificativo del veicolo attualmente non utilizzato a favore della più semplice proprietà **ioe:hasName**.

Il concetto di Ricarica

Nella prime versioni dell'ontologia non esisteva nulla che indicasse il concetto di “ricarica avvenuta”, molto utile per le statistiche lato utente (es: quanto si è speso in un mese per ricaricare il veicolo) e lato Grid (es: quanta energia è stata erogata e quali sono stati gli introiti). Questo perché i tempi previsti dalla prenotazione possono differire sensibilmente da quelli reali: basti pensare a una persona che arriva in ritardo a ricaricare il veicolo o che lascia la colonnina in anticipo.

È stata quindi introdotta la classe **ioe:Recharge**. Si noti che le proprietà di seguito elencate sono state decise di comune accordo con un altro partner del progetto IoE, la spagnola AICIA, allo scopo di eseguire una demo congiunta in cui dimostrare l'interoperabilità tra la nostra piattaforma e quella sviluppata da loro.

- **ioe:hasDate**: data e ora in cui è avvenuta la ricarica.
- **ioe:hasUser**: utente che ha effettuato la ricarica, da cui si evidenzia la necessità di inserire la classe **ioe:Person** (2.2.5).
- **ioe:hasRechargeTime**: il tempo necessario ad effettuare la ricarica.
- **ioe:hasConsumption**: la quantità di corrente impiegata per effettuare la ricarica.

Il Vecchio Protocollo di Prenotazione

Il protocollo di prenotazione iniziale era in stato embrionale e serviva a scopo esemplificativo per dimostrare la fattibilità del progetto. I parametri previsti per la richiesta di ricarica, ovvero le proprietà della classe **ioe:ChargeRequest** erano:

- **ioe:hasPreferredTime**: la data in cui l'utente desidera effettuare la ricarica.
- **ioe:hasPosition**: la posizione da cui si sta eseguendo la richiesta.
- **ioe:hasRequestedEnergy**: la quantità di carica richiesta.

- **ioe:hasRequestingVehicle**: il veicolo che richiede la ricarica.
- **ioe:hasChargeResponse**: la risposta fornita dal sistema.

In seguito ad alcune problematiche verificatesi durante le simulazioni è stato necessario aggiornare il protocollo, all'epoca ancora in fase embrionale. Al di là della mancanza del concetto di utente, l'assenza di range nei campi **ioe:hasPreferredTime** e **ioe:hasPosition** poteva portare ad un'eccessiva ridondanza delle risposte con conseguenti insostenibilità nei tempi e nelle entità del traffico di dati particolarmente in caso di utilizzo di smartphone agganciati alla rete tramite connettività mobile.

Il nuovo Protocollo di prenotazione

In seguito ai problemi messi in luce nel paragrafo precedente (2.2.5) si è deciso di sviluppare un nuovo protocollo di prenotazione che prendesse in considerazione i nuovi requisiti funzionali.

La classe è **ioe:ChargeRequest** è stata quindi ridefinita con le seguenti proprietà:

- **ioe:hasRequestingUser**: utente che richiede la ricarica. Malgrado si possa risalire all'utente tramite il veicolo è stato comunque inserita la proprietà al fine di semplificare le query SPARQL.
- **ioe:hasSpatialRange**: area nella quale si vuole eseguire la ricarica. Il punto centrale di quest'area non corrisponde necessariamente con la posizione dell'utente. Come vedremo nella sezione relativa all'applicazione mobile (4) l'area di prenotazione può essere scelta arbitrariamente sulla mappa.
- **ioe:hasTimeInterval**: il range temporale all'interno del quale si è disposti ad eseguire la ricarica. Può anche essere molto superiore al tempo necessario per la ricarica nel caso in cui non ci siano particolari requisiti di tempo.
- **ioe:hasRequestingVehicle**: il veicolo per cui si richiede la ricarica. Non differisce rispetto al vecchio protocollo.

- **ioe:hasRequestedEnergy**: la quantità di carica richiesta. Non differisce rispetto al vecchio protocollo.

Per quanto migliorato il protocollo è ancora incompleto ed in quanto tale è ancora suscettibile di ulteriori significative migliorie. Ora ad es. non è possibile specificare la volontà dell'utente di essere più flessibile riguardo alla quantità di carica richiesta e nemmeno, nel caso sia richiesto dalla Grid, se fosse disposto a cedere parte della sua carica.

Oltre alla richiesta è stata modificata anche la risposta. Alla classe **ioe:ChargeResponse** è stata aggiunta la proprietà **ioe:hasRelatedRequest** per semplificare le query SPARQL. Anche le opzioni di ricarica contenute nella risposta sono state modificate non solo per farle aderire ai nuovi requisiti funzionali ma anche per semplificare le query SPARQL.

Di seguito verranno esposte le proprietà della classe **ioe:ChargeOption**. Dalla vecchia definizione è stata rimossa la proprietà **ioe:hasChargeProfile**. Le proprietà ereditate dalla vecchia ontologia verranno opportunamente segnalate.

- **ioe:optionHasEVSE**: EVSE presso il quale avverrà la ricarica (ereditata).
- **ioe:hasTimeInterval**: è un'istanza della classe **ioe:TimeIntervalData**. Specifica il tempo necessario a ricaricarsi calcolato sulla base di energia richiesta e di potenza della colonnina. La proprietà era presente anche nella vecchia definizione con i tempi indicati con le date in formato gg/mm/aaaa sostituita con gli attuali millisecondi (passati dalla mezzanotte del 1 Gennaio 1970 UTC) con le proprietà **ioe:hasFromTimeMillisec** e **ioe:hasToTimeMillisec**.
- **ioe:hasRequestingVehicle**: il veicolo per cui è stata effettuata la richiesta. (ereditata)
- **ioe:hasRequestingUser**: l'utente che ha effettuato la richiesta.

- **ioe:hasGridConnectionPoint**: il GCP presso cui avverrà la ricarica. Malgrado vi si possa accedere tramite l'EVSE è stato inserito per semplificare le query SPARQL.
- **ioe:hasTotalPrice**: prezzo totale della ricarica.
- **ioe:hasGcpPosition**: posizione del GCP. Anche questa proprietà è stata aggiunta per semplificare le query SPARQL.

Rimozione informazioni Hardcoded

Fino alla versione 1.5.11 dell'ontologia le informazioni relative ai GCP erano hardcoded nell'ontologia, scelta che all'inizio del progetto è stata dettata del poco tempo. Successivamente questo approccio si è rivelato fortemente limitante poiché la modifica dell'ontologia è un'operazione abbastanza tediosa e soprattutto poco flessibile.

Per di risolvere questo problema ho deciso di rimuovere questi dati dall'ontologia e inserirli all'interno di un file XML che viene caricato dal *City Service* (Sez. 3.4.3) e dal simulatore (Sez. 5) in fase di inizializzazione.

Grazie a questo approccio è diventato relativamente semplice impostare uno scenario del tutto diverso da quello previsto inizialmente dal progetto.

2.3 I Semantic Information Broker

I SIB sono alla base dell'infrastruttura semantica che caratterizza il progetto. I componenti del sistema (dal servizio cittadino, allo smartphone, all'EVSE ecc..) comunicano e rendono permanenti le informazioni grazie ad essi. L'architettura rimane sostanzialmente invariata da quella presentata da Federico Montori ([8]) nel suo progetto di tesi. Verrà comunque esposta al fine di comprendere il resto della trattazione e verranno evidenziate le variazioni apportate alla soluzione iniziale.

L'architettura proposta prevede 2 SIB: il *City SIB* e il *Dash SIB*.

2.3.1 City SIB

Il *City SIB* è il SIB cittadino. Al suo interno vengono immagazzinate tutte le informazioni utili a caratterizzare uno scenario di mobilità elettrica veicolare. Si usa inoltre come interfaccia di scambio dati tra il *City Service* e gli agenti esterni. È infatti nel *emphCity SIB* che vengono scritti i messaggi che compongono i protocolli i quali vengono cancellati una volta terminati. L'area di copertura di tale SIB e di conseguenza del *City Service* è definita dalla classe dell'ontologia **ioe:Zone** anche se attualmente non viene utilizzata poiché gli scenari proposti prendono in considerazione una città alla volta.

Per di poter rispondere alle richieste di prenotazione degli utenti il *City Service* contiene le informazioni relative a tutte le colonnine della zona che ricopre. Possiede inoltre le informazioni relative agli utenti e ai veicoli di competenza. Non contiene le informazioni relative alla batteria, contenute nel *Dash SIB* (2.3.2). Nell'architettura proposta nel progetto precedente gli utenti non erano previsti (2.2.5) per cui le informazioni relative ai veicoli si trovavano esclusivamente nella *Dash SIB*.

L'inizializzazione del *City SIB*, contrariamente a quanto avveniva in precedenza, viene eseguita dal servizio cittadino che inoltre il servizio cittadino si occupa di caricare le informazioni dei GCP da un file XML e di inserirle nel SIB (Sez. 3.4.3).

2.3.2 Dash SIB

Il *Dash SIB* è un SIB che dovrebbe essere integrato a bordo di ogni veicolo. Il suo scopo è tenere costantemente traccia dei parametri che caratterizzano posizione, stato della batteria e tutti gli altri parametri variabili che caratterizzano un veicolo. Per collegare questi dati a un veicolo la tripla che descrive un'istanza di **ioe:Vehicle** viene ripetuta anche su questo SIB.

Inizialmente si pensava che questo SIB sarebbe stato eliminato in un contesto reale in quanto i dati in esso contenuti sarebbero stati letti direttamente dal veicolo. Si è invece deciso di mantenerlo con la funzione di interfaccia comune per l'interrogazione dei dati. Ovvero le applicazioni che utilizzano i dati del veicolo, come ad esempio l'applicazione mobile, dovrebbero esse-

re indifferenti alla sorgente dei dati (es: Veicolo Reale, Veicolo Simulato). Compito del programmatore è implementare un adattatore che scrive i dati in provenienti dalle varie fonti sul *Dash SIB*. Come vedremo più avanti (Cap. 4 e 5) grazie a questa tecnica si può controllare dallo smartphone un veicolo presente nel simulatore tenendone monitorati tutti i parametri. I veicoli del simulatore scrivono tutti sullo stesso *Dash SIB* in quanto sarebbe computazionalmente proibitivo e del tutto inutile fare altrimenti.

Poiché attualmente non esistono veicoli dotati di un SIB, in fase di test su veicolo reale (App. B) è stato necessario dotarsi di un computer a bordo che contenesse il *Dash SIB* dal quale l'applicazione mobile prelevava i dati.

2.4 La libreria IoE

Implementa la logica applicativa ovvero il nucleo operativo del nostro strato di servizi. È la base per qualunque applicazione che intenda interfacciarsi con il sistema in modo semplice ed efficace. Viene infatti sfruttata sia dal *City Service* che dall'applicazione mobile, da me sviluppati. Viene inoltre utilizzata da un visualizzatore di ricariche, nato da un progetto parallelo a questo, del cui sviluppo si è occupata una studentessa di Ingegneria.

2.4.1 Connessione al SIB

La connessione al *SIB* è implementata da una libreria Java (JavaKPI), sviluppata da ARCES, che fa uso del protocollo SSAP. Ho creato un wrapper di questa libreria che ne semplifica l'utilizzo e aggiunge alcune funzionalità: si trova nel package `it.unibo.ioe.sib`.

RdfTriple

La libreria JavaKPI rappresenta le triple RDF come vettore di Stringhe di 5 elementi (`Vector<String>`): soggetto, predicato, oggetto, tipo soggetto, tipo oggetto (dove con tipo si intende o un URI, ovvero un'altra istanza di classe dell'ontologia, oppure un letterale ovvero un dato diretto). Per semplificare la gestione delle triple RDF, che sono l'entità di base del SIB, ho

creato la classe `RdfTriple` che possiede gli attributi `subject`, `predicate`, `object`, `subjectType`, `objectType` e i corrispettivi `getter` e `setter`.

KpConnector

La maggior parte delle operazioni che si possono eseguire con la libreria JavaKPI richiedono due passaggi: l'invio del comando e il parsing della risposta. Questo perché ogni operazione di interazione con il SIB avviene tramite messaggi XML conformi al protocollo SSAP. La libreria genera automaticamente il messaggio da inviare alla SIB ma lascia al programmatore l'onere di effettuare il parsing della risposta. Ho quindi mappato tutte le operazioni di interazione con la SIB all'interno della classe `KpConnector`.

La classe `KpConnector` svolge le operazioni di parsing anche sui messaggi di risposta. Anzichè i vettori di stringhe ho utilizzato istanze della classe `RdfTriple`. Inoltre ho sostituito le liste di triple (es: inserimento multiplo di triple, risultati di query SPARQL implementati con `Vector<Vector<String>>`) con liste di oggetti di tipo `RdfTriple` (`List<RdfTriple>`). `RdfParser` è la classe che converte i tipi di dato usati dalla libreria JavaKPI nei tipi usati dal wrapper che ho implementato.

KpFactory

La classe `KpConnector` necessita di indirizzo, porta e nome del SIB a cui ci si vuole connettere ai fini dell'utilizzo più generico possibile. Nel nostro caso però le connessioni avvengono sempre verso gli stessi due SIB (*City* e *Dash*), quindi ho creato una classe factory (`KpFactory`) che, noti i parametri di connessione necessari, crea due istanze di `KpConnector`, una per il *City SIB* e una per il *Dash SIB*. Quando è necessario connettersi ai SIB il factory restituisce gli oggetti precedentemente creati evitando quindi la creazione di istanze di oggetti inutili. Non essendo la libreria JavaKPI *thread-safe* viene comunque data la possibilità di creare nuove istanze nel caso si lavori in ambienti multi-thread. Questo aspetto verrà approfondito più avanti dove vedremo come, tramite la tecnica dei pool di oggetti, si può risparmiare il tempo necessario a creare nuove istanze.

2.4.2 Entities

Ogni classe dell'ontologia è stata mappata con una rispettiva classe Java (Entity) nel package `it.unibo.ioe.entity`. Per questa scelta architettonica mi sono ispirato all'ORM (Object Relational Mapping), tecnica di programmazione che favorisce l'integrazione di sistemi software aderenti al paradigma della programmazione orientata agli oggetti con sistemi RDBMS (Relational Database Management System).

Mapping

Ho realizzato un mapping molto semplice che tiene conto delle sole proprietà necessarie nello strato di servizi tralasciando alcuni dettagli, come l'unità di misura, che attualmente, malgrado siano previsti nell'ontologia, vengono dati per scontati a livello applicativo. Le proprietà delle classi che hanno come oggetto un letterale sono state mappate con tipi primitivi Java (`int`, `double`, `String` ecc..). Le proprietà che invece come oggetto hanno un'altra classe sono rappresentate come attributo avente come tipo l'Entity che corrisponde alla classe.

Serializzazione

Alcune Entity sono state opportunamente annotate per poterle serializzate in XML tramite la tecnologia **JAXB**. Questo è risultato necessario nel caso dei GCP che vengono caricati da un file XML dal *City Service* che poi li inserisce nel SIB. Ogni classe inoltre implementa l'interfaccia `java.io.Serializable` per consentire il passaggio delle Entity tra le varie **Activity** dell'applicazione mobile.

Esempio La proprietà dell'ontologia `ioe:hasEVSE` ha come dominio `ioe:GridConnectionPoint` e come codominio `ioe:EVSE`. Inoltre tutte le classi dell'ontologia hanno al proprietà `ioe:hasName`.

Questo si traduce nell'Entity mostrato nel Lst. 2.1

```

1  @XmlRootElement(name = "GCP")
2  @XmlAccessorType(XmlAccessType.FIELD)
3  public class GCP implements Serializable {
4      @XmlTransient
5      private String URI;
6      private String gcpName;
7      @XmlElement(name = "EVSE")
8      private List<EVSE> evseList;
9      /* other properties*/
10     /* getter & setter*/
11 }
```

Listato 2.1: Entity di esempio

2.4.3 Controller

I **Controller** sono le classi delegate ad eseguire le operazioni **CRUD** (create, read, delete, update) con il SIB e si trovano nel package `it.unibo.ioe.controller`. Ne esiste uno per ogni Entity. Ogni **Controller** possiede un’istanza di **KpConnector** che realizza la comunicazione con il SIB.

Lettura Le operazioni di lettura si eseguono con una query SPARQL che preleva dal SIB le informazioni necessarie per poi inserirle in una nuova istanza di Entity la quale viene restituita all’utente.

Scrittura Le operazioni di scrittura ricavano una lista di triple RDF a partire da un’istanza di Entity. La lista di triple viene poi convertita in una *SPARQL insert* per ridurre i dati inviati al SIB. L’operazione di conversione è eseguita da una funzione della classe **SibUtil**. Questa tecnica si rivela particolarmente utile quando si usa la libreria da un dispositivo mobile connesso a internet tramite rete cellulare (es: EDGE, GPRS, HSDPA ecc...). Si noti che tutte le Entity possiedono un campo **URI** che viene valorizzato nell’operazione di inserzione con l’URI assegnato all’istanza che si sta per scrivere sul SIB.

Aggiornamento Le operazioni di aggiornamento ricavano i dati aggiornati tramite una query SPARQL e andranno a sostituire quelli obsoleti all’interno di un’istanza di Entity

Rimozione Le operazioni di rimozione lato *City Service* sono eseguite direttamente tramite *SPARQL delete*. Le operazioni di rimozione all'esterno (es. applicazione mobile) per motivi di sicurezza sono invece eseguite tramite richiesta al servizio cittadino il quale si occupa di effettuare la rimozione vera e propria.

Capitolo 3

Servizio Cittadino

Il servizio cittadino (*City Service* o CS) è il cuore dell’architettura software e supporta le interazioni tra gli EV e la Smart Grid. Lo scambio di informazioni avviene tramite il *City SIB* con la struttura dei messaggi definita all’interno dell’Ontologia.

3.1 Architettura

Il CS era già stato implementato da Federico Montori nel progetto precedente. Ho deciso di riscriverlo puntando ad una maggiore modularità e riusabilità. Ho quindi creato libreria di cui ho già trattato in Sez. 2.4 condivisa tra il servizio cittadino e l’applicazione mobile. La libreria fornisce i servizi di base per l’accesso al SIB con un approccio *Object Oriented* ai dati in essa contenuti. Questa scelta progettuale si rifà ai principi di Ingegneria del Software *High Cohesion* e *Low Coupling* [2]

3.2 La comunicazione con il City Service

In questa sezione si analizzano in dettaglio le modalità di comunicazione da e verso il servizio cittadino. Per ogni operazione esiste un protocollo basato su scambio di messaggi la cui struttura è definita mediante classi dell’ontologia. Attualmente le uniche operazioni supportate sono la richiesta di prenotazione e la richiesta di ritiro di prenotazione.

Lo scambio dei messaggi è implementato tramite il meccanismo delle Subscription messo a disposizione dal SIB. Questo comporta che i messaggi si scrivano sul SIB cittadino che manda una notifica al KP sottoscritto a quella particolare modifica. Ne consegue un protocollo di comunicazione asincrono.

3.2.1 Protocollo di richiesta di prenotazione

Descriverò i passaggi necessari al completamento del protocollo di richiesta di prenotazione con particolare attenzione ai messaggi vengono scambiati. Il fine della prenotazione è la certezza di trovare l'EVSE libero quando andremo a caricarci. Come già detto in precedenza, i tempi di ricarica per i veicoli elettrici possono essere molto lunghi. È quindi necessario dare all'utente la certezza che potrà ricaricare il suo veicolo senza il rischio di terminare la carica della batteria.

- 1 Richiesta di Prenotazione:** quando l'utente necessita di fare una ricarica inserisce una richiesta nel SIB. La richiesta è descritta dalla classe dell'ontologia **ioe:ChargeRequest**.
- 2 Risposta da parte del City Service:** il servizio cittadino è sottoscritto all'inserimento di nuove istanze di **ioe:ChargeRequest**. Quindi quando si inserisce la richiesta, il CS riceve una notifica che ne contiene l'URI dal quale può ricavare i parametri che la compongono. A questo punto viene creata una lista di opzioni di ricarica conformi alla richiesta dell'utente compatibilmente con la disponibilità degli EVSE. Le opzioni di ricarica sono classi di tipo **ioe:ChargeOption** e vengono inserite dentro a una classe di tipo **ioe:ChargeResponse**.
- 3 Conferma da parte dell'utente:** L'utente che è sottoscritto all'inserimento di nuove istanze della classe **ioe:ChargeResponse** viene avvisato dal SIB quando il CS inserisce la risposta. Le opzioni di ricarica vengono analizzate dall'utente che sceglierà quella più consona alle sue esigenze. La scelta viene notificata al sistema tramite inserimento di una tripla così formata: **[ioe:chargeOptURI ioe:confirmByUser true]** presupponendo che **ioe:chargeOptURI** sia un'istanza di **ioe:ChargeOption**.

- 4 **Conferma da parte del City Service:** Il servizio cittadino, iscritto all'inserimento di triple aventi come predicato `ioe:confirmByUser`, verifica se l'opzione selezionata è ancora disponibile e in tal caso inserisce la tripla: `[ioe:chargeOptURI ioe:confirmBySystem true]`.
- 5 **Acknowledgment da parte dell'utente:** L'utente riceve la notifica della conferma da parte di CS. Se l'opzione è confermata il CS invia una tripla di Acknowledgment `[ioe:userURI ioe:ackByUser true]`. Altrimenti può provare con un'altra opzione e il protocollo riprende dal punto 3.
- 6 **Creazione Prenotazione:** Il CS, ricevuto l'acknowledgment dall'utente, "blocca" l'EVSE nella finestra di tempo richiesta creando un'istanza della classe `ioe:Reservation`. Inoltre cancella dal SIB tutte le triple necessarie allo svolgimento del protocollo che, una volta terminato, diventano inutili.

3.3 Il Protocollo di rimozione di una prenotazione

Una volta completata la procedura di prenotazione l'EVSE diventa inagibile nella finestra di tempo richiesta dall'utente. Per ritirare la prenotazione l'utente deve inviare una richiesta al CS. Attualmente il servizio cittadino rimuove semplicemente dal SIB i dati relativi alla prenotazione rendendo nuovamente disponibile la ricarica. In futuro il servizio potrà stabilire, in base a regole dettate dai gestori della rete elettrica, se accettare o meno la richiesta ed eventualmente accreditare una penale all'utente.

Come nel caso precedente il protocollo si basa su cambio di messaggi.

1. **Richiesta ritiro Prenotazione:** l'utente inserisce nel SIB cittadino un'istanza della classe `ioe:ReservationRetire`
2. **Ritiro della Prenotazione:** il CS, ovviamente era sottoscritto alla creazione di nuove istanze di `ioe:ReservationRetire`, provvede a rimuovere dal SIB le triple relative alla Prenotazione.

3. **Notifica avvenuta cancellazione:** attualmente l'utente per accertarsi dell'avvenuta cancellazione deve sottoscriversi ai cambiamenti relativi all'URI della prenotazione che sta cancellando.

3.4 Implementazione

In questa sezione discuteremo i dettagli implementativi del *City Service*, dalle tecnologie usate alle scelte architetturali. Principalmente il servizio cittadino deve essere altamente performante in quanto, una vota a regime, deve essere in grado di soddisfare le richieste di centinaia se non migliaia di utenti. Contemporaneamente bisogna controllare che l'elevato parallelismo di thread gestiti dal CS non intacchi l'integrità dei dati residenti sul SIB cittadino per evitare ad esempio che due persone che prenotano nello stesso momento e nello stesso EVSE riescano entrambe a completare la procedura di prenotazione.

Il servizio è scritto interamente in Java. Questo lo rende multi piattaforma, facile da *debuggare* e soprattutto consente l'uso di utilità estremamente versatili riguardo al multi-threading rese disponibili dal linguaggio. Inoltre permette di accedere alla molitudine di librerie scritte per le più disparate necessità. Tra queste troviamo *log4j*, un robusto quanto versatile sistema di logging che permette di tenere costantemente sotto controllo l'esecuzione del servizio con vari gradi di granularità del log.

Per rendere il servizio più performante possibile si sono adottate tecniche di programmazione quali: pool di oggetti, pool di thread, e caching delle risorse.

3.4.1 Pool Di Oggetti

Per effettuare le connessioni al SIB cittadino è necessario istanziare oggetti di tipo **KpConnector**. Inoltre per effettuare il parsing delle risposte alle sottoscrizioni sono necessari oggetti di tipo **SSAP_XMLTools** forniti dalla libreria JavaKPI. Come anticipato nella Sez. 3.4 il CS deve supportare connessioni multiple simultanee, ognuna delle quali dialogante con il SIB. Dal momento che ne la libreria JavaKPI ne wrapper da me creato sono thread-safe, è ne-

cessario istanziare un oggetto **KpConnector** per ogni connessione insieme a uno di tipo **SSAP_XMLTools** per *parsare* i risultati delle sottoscrizioni.

Per evitare che all'avvio di ogni connessione venisse creato un oggetto **KpConnector**, operazione assai onerosa, ho optato per la tecnica dei pool di oggetti. Questa consiste nel creare un numero sufficiente di istanziare oggetti all'inizio dell'applicazione e quando ne serve uno lo si chiede al pool che lo fornisce nel tempo di una *chiamata a metodo*. Terminato l'uso dell'oggetto lo si restituisce al pool che lo renderà disponibile ad un altro richiedente. Così si evita il delay necessario a instanziare una connessione con il SIB ogni volta che si esegue una richiesta al CS.

Aggiungo che, viste le ottimizzazioni delle moderne *Java Virtual Machine* e dei *Garbage Collector* per quanto riguarda gli oggetti con breve durata, questa tecnica può rischiare di abbassare le performance anziché aumentarle [11]. Resta comunque vantaggiosa nel caso di oggetti la cui creazione potrebbe risultare abbastanza onerosa come nel caso delle connessioni ai database o alla rete.

Il cuore di questo sistema è la classe **ObjectPool<T>** ([12]) che troviamo nel package **it.unibo.cityservice.pool** che rappresenta un pool di oggetti di tipo **T**. Questa classe contiene un metodo astratto, **createObject()**, che va implementato nelle sottoclassi con la logica di creazione dell'oggetto di cui vogliamo creare il pool.

Nel mio caso ho creato **XmlToolsPool** e **CitySibPool** e a titolo esemplificativo mostrerò l'implementazione del primo nel Lst. 3.1:

```

1  public class XmlToolsPool extends ObjectPool<SSAP_XMLTools>{
2
3      public XmlToolsPool(final int minIdle) {
4          super(minIdle);
5      }
6
7      @Override
8      protected SSAP_XMLTools createObject() {
9          return new SSAP_XMLTools();
10     }
11 }
```

Listato 3.1: Implementazione di ObjectPool

Evidenzio la semplicità di creazione di un pool per un determinato tipo di dato. Creato il pool, per interagire con esso, si usano i seguenti metodi:

- `public T borrowObject()`: preleva un oggetto dal pool.
- `public void returnObject(T object)`: restituisce un oggetto al pool.

3.4.2 Pool Di Thread

Adesso che abbiamo visto come è che cos'è il meccanismo dei pool degli oggetti e perchè è stato utilizzato vediamo invece di capire cosa sono i pool di thread e quali problematiche vanno a risolvere. La creazione di thread può creare problemi in termini di performance (in quanto la creazione e distruzione di questo oggetti è abbastanza onerosa), di controllare il numero del numero dei thread creati e infine di scalabilità ([10]).

Pertanto è necessario ricorrere alle classi del package `java.util.concurrent` che implementano l'interfaccia `Executor`. In questo caso sono state poi utilizzate istanze dell'interfaccia `ExecutorService` che mette a disposizione metodi volti a controllare il ciclo di vita del pool stesso.

L'inizializzazione degli `ExecutorService` avviene tramite l'invocazione di un metodo statico della classe `Executors` che specifica la dimensione del pool che si vuole creare. Quando invece si vuole assegnare un compito ad uno dei thread del pool si usa il metodo `execute` che prende in ingresso un'istanza dell'interfaccia `Runnable`.

```
1 ExecutorService pool = Executors.newFixedThreadPool(30);
2 pool.execute(new Runnable() {
3     @Override
4     public void run() {
5         System.out.println("hello world");
6     }
7});
```

Listato 3.2: Creazione Pool di Thread

3.4.3 Il funzionamento del City Service

Come visto nella sezione 3.2 il servizio cittadino deve gestire un gran numero di messaggi a ogni tipologia dei quali corrisponde una sottoscrizione al SIB cittadino. Inoltre per poter rispondere alle richieste degli utenti deve anche avere le informazioni relative a tutti gli EVSE.

Inizializzazione

Il CITY SERVICE all'avvio compie innumerevoli compiti. Contrariamente a quanto avveniva in precedenza, dove i GCP erano codificati nell'ontologia e l'ontologia stessa veniva caricata da un programma esterno, ho deciso di delegare le inizializzazioni al CITY SERVICE col fine di semplificarne lo sviluppo. Infatti in precedenza ad ogni riavvio del servizio era necessario uccidere manualmente il SIB e riavviarlo per avere una situazione di partenza pulita. Il caricamento dell'ontologia avviene tramite una libreria da me sviluppata, *OntologyLoader*, (App. D) che a sua volta si appoggia alle librerie Apache Jena ([5]). In ambito di produzione ovviamente il servizio cittadino non eliminerebbe i dati e nemmeno ricaricherebbe quelli già presenti.

Di seguito una lista dettagliata delle operazioni eseguite dal CITY SERVICE in fase di inizializzazione:

- **Lettura file di configurazione:** Cerca il file di configurazione `cityservice.properties` dal quale carica le informazioni del SIB cittadino, il nome dell'ontologia, il nome del file contenente i GCP.
- **Scrittura Ontologia:** L'ontologia viene scritta nel SIB cittadino
- **Caricamento Informazioni GCP:** Le informazioni di tutte le stazioni di ricarica presenti in città vengono caricate da un file xml. Le stesse informazioni vengono inserite nel SIB al fine di poterle condividere con le altre entità del sistema.
- **Creazione Pool:** Vengono creati i pool di thred e di oggetti.
- **Sottoscrizioni:** Ci si sottoscrive alle informazioni su cui si vuole rimanere aggiornate. Sostanzialmente ci si assicura che arrivino le notifiche per i messaggi descritti nella sezione 3.2:

1. Creazione di nuove istanze di **ioe:ChargeRequest**
2. Inserimento di triple contenenti come predicato **ioe:confirmByUser**
3. Inserimento di triple contenenti come predicato **ioe:ackByUser**
4. Creazione di nuove istanze di **ioe:ReservationRetire**

Gestione delle richieste

Quando ci si sottoscrive a qualcosa nel SIB oltre a definire a cosa ci vogliamo sottoscrivere dobbiamo anche definire un *handler* che verrà eseguito quando avverrà il cambiamento a cui siamo interessati. Come visto sopra, nella sezione 3.2, di sottoscrizioni ne vengono fatte 4, per ognuna di esse viene definito lo stesso *handler* il quale lancia un thread istanza della classe **RequestDispatcher**. Esso si occupa semplicemente di gestire la richiesta e di eseguire un altro thread, sempre contenuto in un pool, che la soddisfi.

Sessioni Per ottenere un ulteriore incremento di performance è stato introdotto il concetto di sessione. La sessione inizia al momento in cui il *City Service* riceve la richiesta e finisce quando riceve l'acknowledgment. Il fine della sessione è mantenere una cache dei dati che vengono scambiati al fine di risparmiare query SPARQL, che sono assai onerose in termini di performance, e di dare un limite temporale alle sessioni stesse. La classe che si occupa di gestire le sessioni è **SessionManager** mentre la sessione è rappresentata dalla classe **session**.

All'interno della sessione vengono salvate le seguenti informazioni:

- **chargeRequest**: Un'istanza dell'entity **ChargeRequest** ricevuta dall'utente.
- **chargeResponse**: Un'istanza dell'entity **ChargeResponse** inviata all'utente.
- **reservation**: Un'istanza dell'entity **Reservation** creata in seguito alla conferma dell'utente.
- **startTime**: Il momento in cui inizia la sessione.

- **endTime**: Il momento in cui finisce la sessione che corrisponde all'arrivo dell'acknowledgment dell'utente.

La classe **SessionManager** possiede un timer che a tempo prefissato lancia un thread che controlla le sessioni attive. Se una delle sessioni è attiva da molto tempo senza essere stata chiusa allora significa che probabilmente c'è stato un problema e quindi vengono rimossi dal SIB tutti i dati relativi alla sessione compresa la prenotazione.

Prenotazioni La gestione delle prenotazioni, una volta che sono state create e confermate dall'utente, è delegata alla classe **ReservationManager**. A suo interno vengono salvate in una cache le istanze di **Reservation** create. Quando arriva una nuova richiesta di prenotazione la verifica della disponibilità viene fatta su questa cache anziché sul SIB. Questo sempre al fine di ridurre gli accessi al database e il successivo parsing delle risposte. Siccome l'uso di questa classe è altamente parallelo, ovvero possono accedervi molti thread contemporaneamente, viene utilizzata una mappa thread-safe messa a disposizione da java **ConcurrentHashMap**. Inoltre le operazioni di verifica di disponibilità delle prenotazioni vengono sono atomiche a livello di EVSE grazie a un lock per ognuno di essi.

Thread Ci sono cinque classi diverse che implementano l'interfaccia **Runnable** ognuna delle quali ha il compito di gestire un determinato aspetto dei protocolli di richiesta. Ovviamente c'è un pool di esecuzione per ognuna di esse.

- **RequestDispatcher**: È il thread che si occupa di smistare le richieste agli altri esecutori, viene eseguito ogni volta che arriva una notifica da una sottoscrizione. La decisione avviene in base all'id della sottoscrizione che viene assegnato in fase di inizializzazione ed immagazzinato all'interno di variabili globali. Il codice di questo del corpo di questo thread è mostrato nel listato 3.3 dove si nota chiaramente l'utilizzo del pool di oggetti e dei pool di thread nonché del **logger**. Questo approccio viene usato anche per gli altri thread con l'aggiunta del reperimento dei **KpConnector** dal relativo pool.

- **ChargeRequestHandler:** Viene eseguito quando un utente inserisce un'istanza di `ioe:ChargeRequest` nel SIB. Dalla sottoscrizione ne ricava l'URI e con l'apposito controller `ChargeRequestController` la trasforma in un Entity java istanza della classe `ChargeRequest`. I passaggi che dalla richiesta elaborano una risposta sono i seguenti:
 1. Controllo di coerenza sulla richiesta. Se la richiesta non è valida allora viene inviata una risposta vuota. Altrimenti si procede con il resto delle operazioni.
 2. Viene creata un'istanza di `Session` gestita dalla classe `SessionManager`.
 3. Scelta dei *GCP* che si trovano nell'area scelta dall'utente tramite la libreria `UniboGeoTools`.
 4. Vengono ciclati tutti gli *EVSE* appartenenti ai *GCP* selezionati.
 5. Per ogni *EVSE* vengono ricavati gli slot di tempo compatibili con la fascia oraria e la quantità di carica richieste dall'utente.
 6. Viene generata la risposta istanza dell'Entity `ChargeResponse`. Al fine di minimizzare lo scambio di dati attraverso la rete soprattutto per non penalizzare i dispositivi mobili, le richieste vengono filtrate. Vengono inviate le opzioni di ricarica provenienti dai 5 *GCP* più vicini e ne vengono scelte al massimo 2 per ogni *EVSE*
 7. La risposta viene trasformata inserita nel SIB tramite la classe `ChargeResponseController`.
- **ConfirmByUserHandler:** Questo thread viene invocato quando l'utente scaglia un'opzione di ricarica e semplicemente controlla che sia ancora disponibile. In tal caso crea la prenotazione in modo che nessun altro possa usare la colonnina nell'orario richiesto. Da notare che l'istanza di `ChargeOption` viene presa dalla cache contenuta nella sessione anziché tramite query SPARQL e che l'istanza di `Reservation` viene salvata nel gestore di prenotazioni `ReservationManager`.
- **AckByUserHandler:** Viene invocato quando l'utente conferma l'opzione di ricarica. A questo punto il protocollo può considerarsi terminato e quindi vengono eliminate tutte le informazioni ad esso relative dal SIB

e la sessione associata. L'unica informazione che rimane è un'istanza di `ioe:Reservation` nel SIB.

- `RetireReservationHandler`: Si occupa semplicemente di eliminare le istanze di `ioe:Reservation` dal SIB e le corrispettive informazioni nella cache contenuta in `ReservationManager`.

```

1 SSAP_XMLTools xmlTools = xmlToolsPool.borrowObject();
2 String subscriptionID = xmlTools.getSubscriptionID(subscribeResult);
3
4 if (chargeRequestSubId.equals(subscriptionID)) {
5     chargeRequestExecutor.execute(new
6         ChargeRequestHandler(subscribeResult));
7 } else if (confirmByUserSubId.equals(subscriptionID)) {
8     confirmByUserExecutr.execute(new
9         ConfirmByUserHandler(subscribeResult));
10 } else if (ackByUserSubId.equals(subscriptionID)) {
11     ackByUserExecutor.execute(new AckByUserHandler(subscribeResult));
12 } else if (retireReservationSubId.equals(subscriptionID)) {
13     retireReservationExecutor.execute(new
14         RetireReservationHandler(subscribeResult));
15 } else {
16     logger.error("Unexpected subscription id: " + subscriptionID);
17 }
18
19 xmlToolsPool.returnObject(xmlTools);

```

Listato 3.3: Corpo di RequestDispatcher

3.5 Testing

Aspetto fondamentale che ha caratterizzato lo sviluppo del *City Service* è stata l'integrazione con test di unità che ne hanno assicurato la continuità di funzionamento durante le fasi di modifica e sviluppo.

Il framework utilizzato per eseguire il testing è `junit4`, un'ottima libreria Java che tramite il meccanismo delle annotazioni permette di scrivere ed eseguire i test in maniera semplice ed efficace.

I test sono stati determinanti non solo per assicurare la stabilità del codice ma anche per testare le performance del sistema, soprattutto del SIB.

3.5.1 Test Protocolli

Per testare i protocolli ho implementato un thread che svolgesse tutte le operazioni necessarie per completare una richiesta di prenotazione ed il ritiro della stessa. L'incapsulamento della logica del protocollo di richiesta all'interno di un thread permette di fatto l'esecuzione multipla di istanze di quest'ultimo simulando quindi l'interazione di molteplici utenti. La classe delegata a svolgere questo compito è `ioe:ChargeProtocolTest` nel packacge `it.unibo.ioe.cityservice.chargeprotocol` situato nella cartella di test. All'interno di questa classe si trova una inner-class, `ReservationProtocol` che implementa `Runnable` rendendo quindi possibile la sua esecuzione all'interno di un thread separato.

Per simulare le attese dell'utente è stato usato il meccanismo dei lock. Ogni fase del protocollo ha un suo lock che viene acquisito subito dopo l'invio messaggio (List. 3.4) e viene rilasciato al momento dell'esecuzione l'handler associato alla sottoscrizione della risposta (List. 3.5).

```

1 chargeRequestController.insertChargeRequest(request);
2 synchronized (chargeResponseLock) {
3     try {
4         chargeResponseLock.wait();
5     } catch (InterruptedException ex) {
6         logger.error(ex.getMessage(), ex);
7     }
8 }
```

Listato 3.4: Inserimento della `ChargeRequest` e attesa della risposta

```

1 String subscriptionID = xmlTools.getSubscriptionID(xml);
2 if (chargeResponseSubId.equals(subscriptionID)) {
3     [...]
4     chargeResponsesUri = subscriptionResult.get(0);
5     synchronized (chargeResponseLock) {
6         chargeResponseLock.notify();
7     }
8 }
```

Listato 3.5: Handler associato al messaggio di risposta

L'opzione di ricarica viene scelta casualmente tra quelle fornite, nel caso in cui venisse confermata dal sistema il thread preleva la corrispondente istanza di **Reservation** creata e controlla che i parametri in essa contenuti siano coerenti con quelli dell'opzione scelta.

Tutti i dati scambiati dal protocollo vengono testati se uno di essi dovesse fallire causerebbe la terminazione del protocollo stesso. A tal proposito **junit** mette a disposizione una serie di funzioni che permettono di fare asserzioni di ogni tipo al fine di validare i dati. Queste funzioni iniziano con il prefisso **assert**, un esempio si può vedere nel listato 3.6.

```
1 response =
2     chargeResponseController.findChargeResponse(chargeResponsesUri);
3 assertEquals(request.getUri(), response.getChargeRequestUri());
```

Listato 3.6: Risposta ricavata a partire dall'uri, test del risultato

3.5.2 Valutazione Performance

L'esecuzione di molteplici istanze del protocollo di richiesta è stato determinante al fine di testare le performance del sistema in quanto ha permesso di capire quante richieste contemporanee potessero essere servite e quali fossero i colli di bottiglia. È stato infatti scoperto che arrivati a un certo numero di connessioni simultanee il SIB allunga i tempi di risposta fino a far scattare il **socket-timeout** della libreria JavaKPI, in quanto, come indicato nella Sez. 2.1.1, il SIB comunica con l'esterno tramite protocollo *TCP/IP*. Questa scoperta ha portato a trovare un bug che affliggeva il SIB stesso, il quale, una volta interrotta brutalmente la connessione con il socket di JavaKPI, dava seri problemi di memory-leak come si può ben vedere in Fig. 3.1. L'immagine l'ho presa dal mio computer e l'ho inviata agli sviluppatori di Smart-M3-B per dimostrare il problema. Si nota chiaramente il momento precedente all'uccisione del SIB in cui erano allocati circa 8GB di Ram e 3Gb di Swap. Il problema è stato risolto e la versione attuale (0.901) ne è esente.

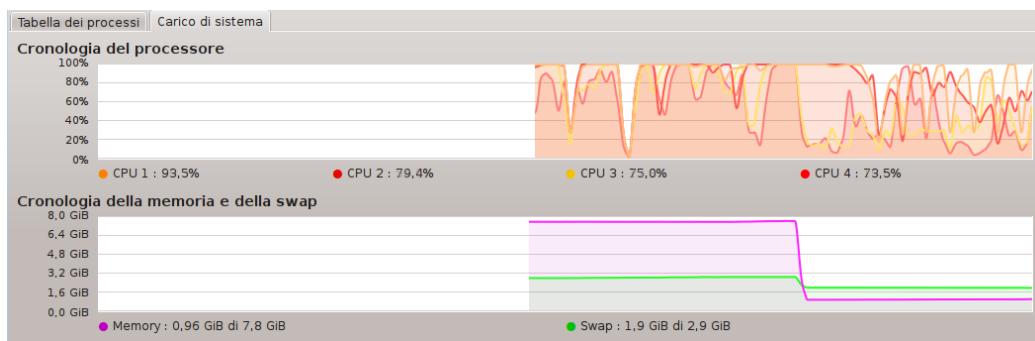


Figura 3.1: Memory Leak del SIB che avveniva quando la connessione viene interrotta da un socket-timeout

La scoperta di questo problema ha portato quindi alla necessità di ottimizzare le performance, è stato quindi deciso di limitare il numero di risposte fornite dal servizio cittadino (Par. 3.4.3) che causa un ingente traffico di dati da e verso il SIB. Altra ottimizzazione che decisa in seguito a queste scoperte è stata quella di trasformare le operazioni di INSERT di triple in query SPARQL, in quanto, grazie all’uso dei prefissi al posto degli URL interi, si riesce ad ottimizzare notevolmente il volume di dati scambiato, il che porta a notevoli vantaggi durante l’uso nell’applicazione mobile.

Capitolo 4

Applicazione Mobile

In questo capitolo verrà presentato un esempio di applicazione mobile in grado di connettersi al *City Service* e di eseguire le operazione di prenotazione e ritiro delle ricariche. Il suo scopo è permettere all’utente di interagire con la smart-city al fine di ridurre le problematiche derivanti dall’utilizzo di veicoli elettrici. La comunicazione avviene mediante i protocolli visti nella Sez. 3.2.

Inizialmente era possibile eseguire solo operazioni di prenotazione e cancellazione di ricariche e funzionava unicamente in presenza del simulatore in quanto si prendeva il possesso di un veicolo simulato. Il funzionamento è stato poi ampliato con la possibilità di connettersi tramite Bluetooth a un veicolo reale, opportunità concessa dal *Centro Ricerche Fiat (CRF)*, oppure di connettersi in assenza di veicoli per permettere all’utente di effettuare una ricarica comodamente seduto a casa.

A questo si è aggiunta la possibilità di analizzare il profilo altimetrico che separa il dispositivo mobile da un determinato EVSE con lo scopo di fare previsioni più accurate sui consumi necessari a raggiungerlo.

La piattaforma di sviluppo scelta è *Android* vista la sua grandissima diffusione e versatilità.

4.1 Architettura

La piattaforma scelta per lo sviluppo è Android dalla versione *4.0.3* in su. Questo perché vanta maggiori performance e un interfaccia utente più

gradevole è facile da programmare. La libreria di base per interfacciarsi con il SIB è quella esposta nella sezione 2.4.

La comunicazione con il servizio cittadino avviene tramite scambio di messaggi con il *City SIB*, mentre le informazioni relative al veicolo, in particolar modo se quest'ultimo è simulato, arrivano dal *Dash SIB*. È possibile collegare l'applicazione ad un veicolo reale che sia provvisto della tecnologia Blue&Me di Fiat. I dati del profilo altimetrico sono ottenuti tramite una libreria, chiamata *UniboGeoTools*, che ho sviluppato appositamente per l'occasione.

4.1.1 Android

Malgrado Android sia ampiamente conosciuto penso sia necessario spendere qualche parola al fine di introdurre i concetti che stanno alla base della programmazione di applicazioni su questo sistema operativo per poter capire approfonditamente il resto della trattazione.

Android è un sistema operativo basato Linux-based, Open Source, orientato all'utilizzo su dispositivi mobili anche se negli ultimi anni sta prendendo sempre più piede all'interno di smart-tv, dispositivi embedded, mini computer ecc..

La programmazione di applicazioni avviene attraverso una versione ad-hoc del linguaggio Java che, seppur venga eseguita su una virtual machine diversa dalla JVM (Dalvik), ne mantiene quasi tutte le caratteristiche e la libreria di base. Naturalmente oltre alla libreria standard vengono fornite le API che permettono l'interfacciamento con le funzionalità di Android.

Activity

Le Activity sono uno degli elementi centrali della programmazione di applicazioni Android ([7]). In genere un Activity rappresenta una singola schermata della nostra applicazione. Le applicazioni possono definire una o più Activity per trattare diverse fasi del software, e generalmente ognuna di esse corrisponde ad un'azione specifica che può essere eseguita dall'utente.

Ci può essere una sola Activity attiva in un determinato istante, quelle che invece non sono attive possono essere terminate in qualunque momento

dal sistema operativo al fine di recuperare memoria, questo comporta che il programmatore debba prevedere per ogni Activity il codice necessario a salvarne lo stato per permetterne il ripristino nel caso sia necessario. Questo comporta, come si può vedere in figura 4.1a, che le Activity di Android abbiano un ciclo di vita abbastanza complesso.

Service

Un Service è un processo che gira in background (un concetto molto simile al deamon in ambiente Unix) e può essere avviato e comandato da Activity o altri Service. La classe Service viene utilizzata per creare componenti software che possono svolgere attività in modo “invisibile”, senza interfaccia utente.

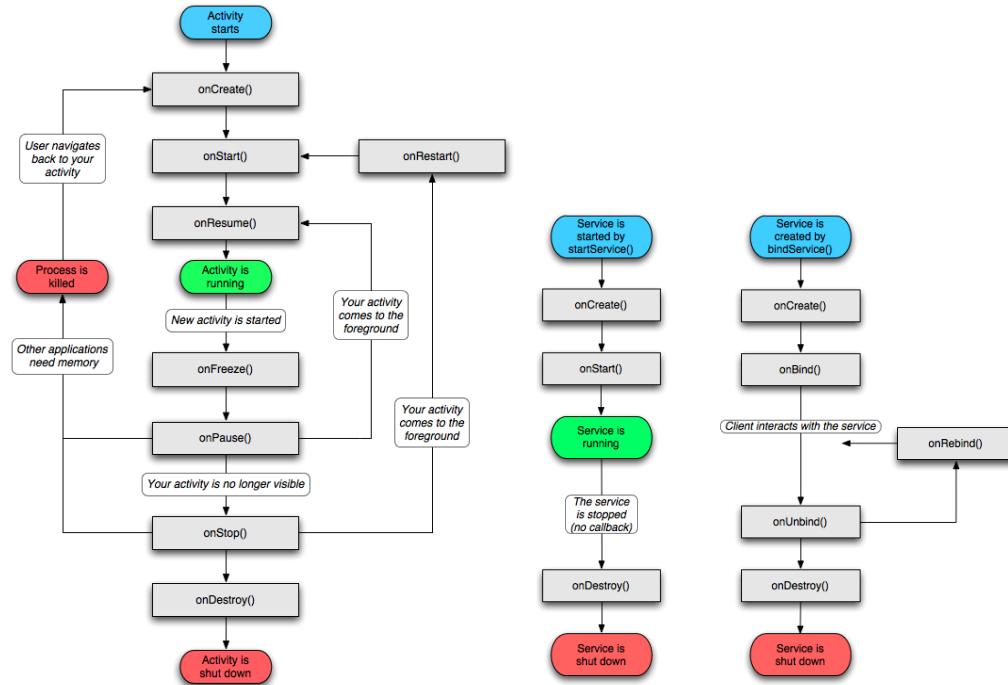
Un Service può trovarsi in due stati([4]):

- **Started:** Un servizio si trova in questo stato quando viene invocato il metodo startService(), il servizio gira in background per un tempo indefinito o finché il componente che lo ha invocato non viene distrutto (Fig. 4.1b sinistra).
- **Bounded:** Un servizio si trova in questo stato quando si invoca il metodo bindService(). I servizi di questo tipo offrono un’interfaccia per la comunicazione client-server, in questo modo le componenti che invocano il servizio possono interagire con esso. In questo caso il servizio è attivo solo finché le componenti sono associate con esso (Fig. 4.1b destra).

Un Servizio avviato ha una priorità più alta rispetto ad Activity in stato di inattività, in questo modo vi è minore probabilità per un Service di essere terminato dal gestore delle risorse di runtime. L’unica ragione per cui Android potrebbe fermare un Service prematuramente è per fornire risorse addizionali al componente software in primo piano (normalmente una Activity).

I servizi vengono avviati nel thread principale del processo, ciò significa che se eseguissero operazioni bloccanti o ad alto consumo di risorse potrebbero portare al blocco dell’intero processo e far generare ad Android un errore

del tipo: “Application Not Responding”, per evitare questo è bene far gestire il servizio in un thread separato.



(a) Ciclo di vita di un Activity Android (b) Ciclo di vita di un Service Android

Figura 4.1: Ciclo di vita Activity e Service Android

4.1.2 Blue&Me

Il Blue&Me è il risultato di un accordo tra Fiat Auto e Microsoft con l’obiettivo di progettare sistemi telematici innovativi per l’automotive, il sistema è stato presentato nel 2006 ([1]). Basato sulla piattaforma Windows Embedded Automotive (unione del sistema operativo Windows CE con il middleware Microsoft Auto) è sviluppato da Magneti Marelli (azienda del gruppo FIAT) in collaborazione con Microsoft ([13]).

Blue&Me è un sistema viva voce con tecnologia Bluetooth, riconoscimento vocale, lettore multimediale con presa USB e comandi al volante. Fiat Auto e Microsoft, con il supporto di Magneti Marelli, offrono una piattaforma adattabile alla maggior parte dei telefoni cellulari e lettori musicali. Il

sistema si interfaccia al cellulare mediante la tecnologia Bluetooth e permette al guidatore di rispondere al telefono lasciando il telefono in tasca.

La funzionalità del Blue&Me più importante ai nostri scopi è la possibilità di monitorare i parametri del veicolo. Nel nostro caso abbiamo lavorato con un prototipo di Fiat Daily elettrico opportunamente modificato per trasmettere i dati della batteria tramite tecnologia Bluetooth. La connessione è stata possibile grazie a delle librerie fornite dal CRF con tanto di applicazione di esempio.

L'interazione con il Blue&Me è di tipo push ovvero ogniqualvolta che la centralina della macchina si accorge che un dato è variato (secondo una determinata soglia) allora lo "scrive" sull'interfaccia Bluetooth. Il che implica che se dall'altra parte ci deve essere un'applicazione che dedica un processo alla sola lettura delle informazioni che arrivano dal Blue&Me. Fiat, nella libreria fornita, mette a disposizione un Service Andorid adatto allo scopo.

4.1.3 Profilo Altimetrico e Contributo Energetico

Grazie allo sviluppo di un'apposita libreria, UniboGeoTools (App. C) l'applicazione è in grado di prelevare i dati relativi al profilo altimetrico che separa l'utente da una determinata destinazione e quindi fare uno studio sul contributo energetico necessario a vincere il dislivello. Le informazioni riguardo al consumo energetico sono approssimative ma danno comunque un'indicazione valida all'utente il quale se vede che l'energia necessaria a vincere il dislivello è maggiore di quella contenuta nella batteria allora saprà per certo che in quel determinato punto non ci potrà arrivare.

La libreria UniboGeoTools possiede anche funzioni utili a trovare il percorso, in strada, tra due punti. Queste si sono rivelate particolarmente utili al fine di mostrare i percorsi sulla mappa e la distanza precisa che separa l'utente, ad esempio, da una colonnina di ricarica.

4.2 Modalità di esecuzione

L'applicazione al fine di adattarsi ai diversi scenari possibili offre molteplici modalità di esecuzione, questo per adattarsi a tutti gli scenari possibili.

La scelta della modalità di esecuzione avviene nella schermata iniziale come si può vedere in figura 4.2a

4.2.1 Simulazione

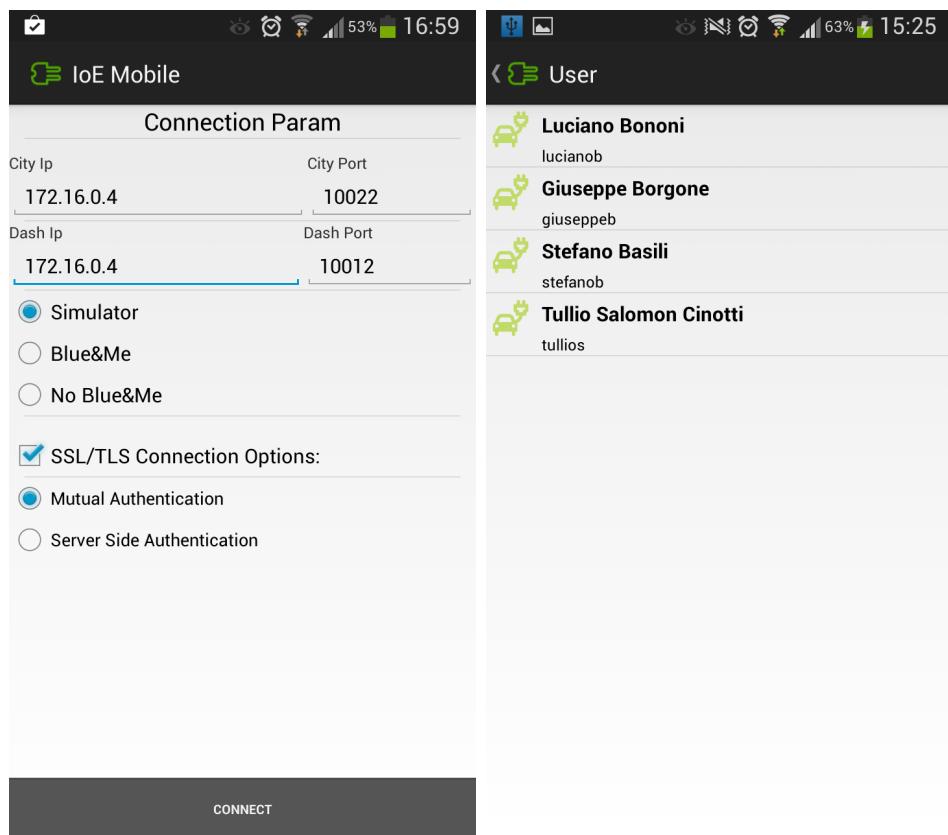
Questa modalità permette di prendere il controllo di un veicolo contenuto nel simulatore, il quale deve essere avviato con un apposito parametro che causa la scrittura dei dati relativi ai veicoli sul *Dash SIB*. Questo implica che una volta premuto sul pulsante *Connect* (Fig. 4.2a), bisogna scegliere Luciano Bononi in quanto è l'utente di default usato dalle macchine del simulatore (Fig. 4.2b). Da qui in poi l'applicazione non ha molte differenze rispetto alla modalità di esecuzione con Blue&Me se non che le azioni intraprese avranno ripercussioni sul simulatore e non sul mondo reale.

4.2.2 Con Blue&Me

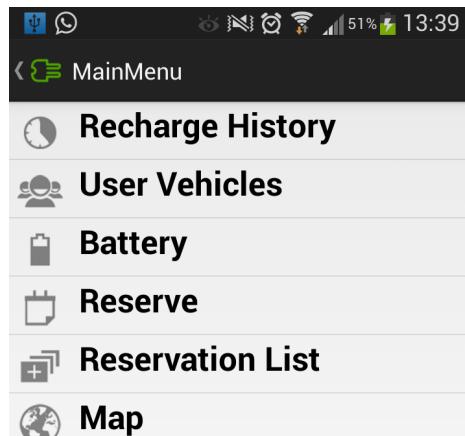
Questa modalità viene usata in un contesto reale e necessita la presenza di un veicolo che possegga la tecnologia Blue&Me di Fiat. Le informazioni relative alla batteria vengono prelevate tramite Bluetooth scritte nel *Dash SIB* per i motivi di interoperabilità spiegati nella Sez. 2.3.2 ovvero dare la possibilità a chiunque di interfacciarsi con l'applicazione mobile a patto di scrivere un adattatore che scriva i dati sulla *Dash SIB*. Le informazioni relative al GPS, siccome non fornite dal veicolo, vengono prelevati dal GPS (o altre fonti) fornito dallo smartphone.

4.2.3 Senza Blue&Me

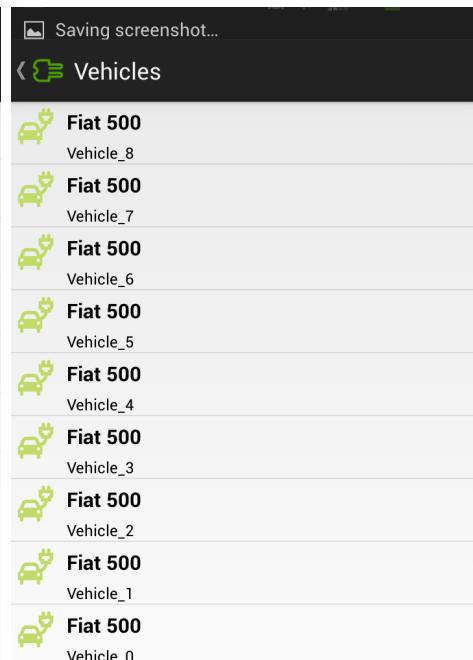
Questa modalità d'uso è rivolta principalmente a chi vuole svolgere le attività di interazione con il *City Service* senza essere a bordo del proprio veicolo. Questo comporta l'assenza della DASH SIB, infatti quando viene scelta viene disabilitato l'inserimento dei parametri di quest'ultima. Diviene quindi impossibile monitorare i parametri del veicolo e di conseguenza parte del menu principale viene disabilitata. Anche se la *Dash SIB* fosse situata sul cellulare anziché sul veicolo a poco servirebbe in quanto il veicolo sarebbe comunque fuori portata o comunque spento. Si possono comunque svolgere



(a) Schermata Principale



(b) Selezione Utente



(c) Menu Principale

(d) Selezione Veicolo

le operazioni di prenotazione e ritiro delle ricariche e tenere monitorato lo stato di quelle già effettuate nonché guardare la mappa con le colonnine.

4.3 Funzionalità

In questa sezione eseguirò un analisi dettagliata delle funzionalità dell'applicazione. La descrizione cercherà di essere il più possibile funzionalità-centrica e non activity-centrica anche se spesso le due cose coincidono.

4.3.1 Il menu principale

La prima schermata dell'applicazione è quella che permette di scegliere i parametri di connessione ai SIB (Fig. 4.2a) e la modalità di esecuzione. Una volta premuto il tasto *connect* ci troveremo a scegliere l'utente (Fig. 4.2b) a questo punto ci troveremo davanti il menu principale (Fig. 4.2c).

Il menu apparirà con solo due opzioni: *Recharge History* e *Select Vehicle*. Questo perché tutte le altre opzioni sono subordinate al veicolo che si sta utilizzando e quindi non vengono mostrate finché non se ne sceglie uno attraverso l'apposito menu (Fig. 4.2d).

4.3.2 Storia delle ricariche effettuate

In questa schermata (Fig. 4.3c) possiamo tener monitorata la storia delle ricariche effettuate. Questa parte è stata introdotta anche per dimostrare l'interoperabilità del nostro sistema con uno SIB-based sviluppato dalla spagnola AICIA.

4.3.3 Monitoraggio Parametri Batteria

Dal menu principale si può accedere al monitoraggio dei parametri della batteria attraverso l'opzione *Battery*. Il monitoraggio consente di vedere sia i parametri variabili (Fig. 4.3a) che quelli nominali (Fig. 4.3b). I dati relativi all'intensità di corrente e al voltaggio sono visibili solo se si è connessi a un veicolo reale tramite Blue&Me in quanto il nostro modello di simulazione non li implementa. Mentre i dati nominali sono disponibili in modalità di

simulazione ma non in quella reale in quanto queste informazioni non vengono fornite dalla centralina del veicolo.

4.3.4 Effettuare una richiesta di prenotazione

Per accedere a questa funzionalità bisogna scegliere dal menu principale l'opzione *Reserve*. Ci troveremo davanti alla schermata che ci permette di impostare i parametri necessari a creare una richiesta di prenotazione. Come si può vedere in figura 4.4a il servizio mette a disposizione svariate opzioni per personalizzare il processo di prenotazione. Essenzialmente in questa schermata viene data una veste grafica al protocollo di richiesta descritto nella sezione 3.2.

Creazione della richiesta

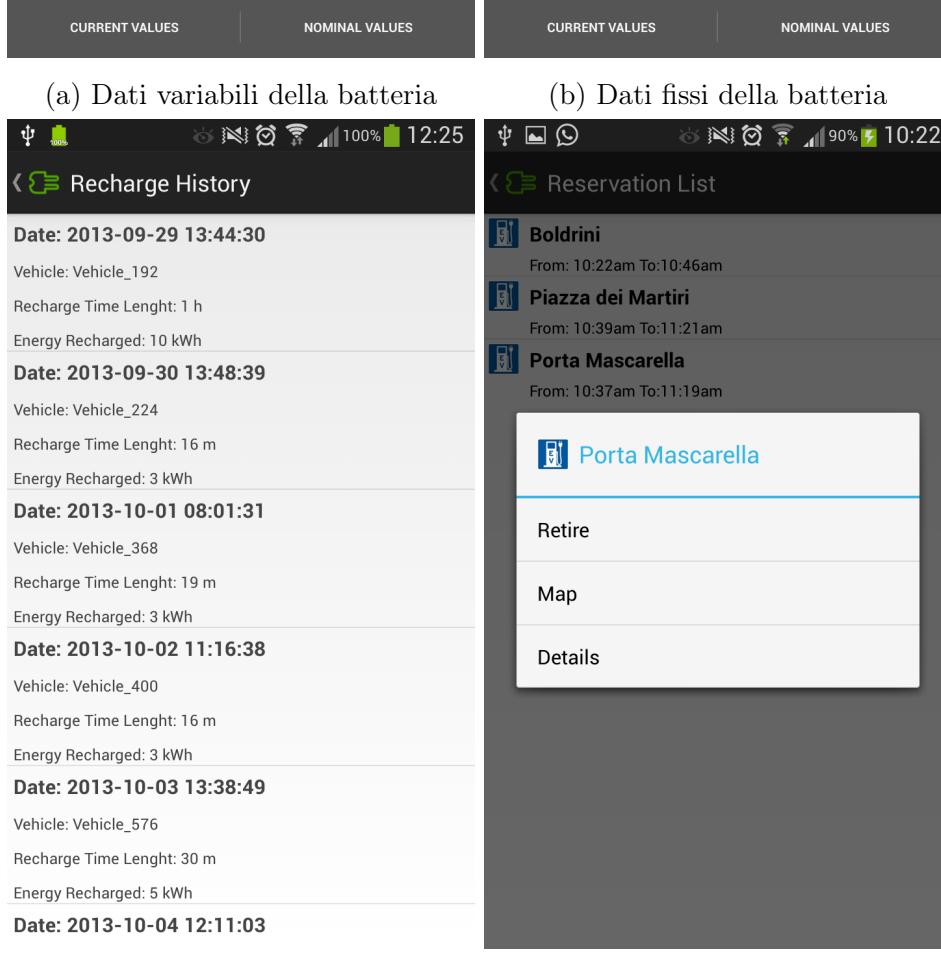
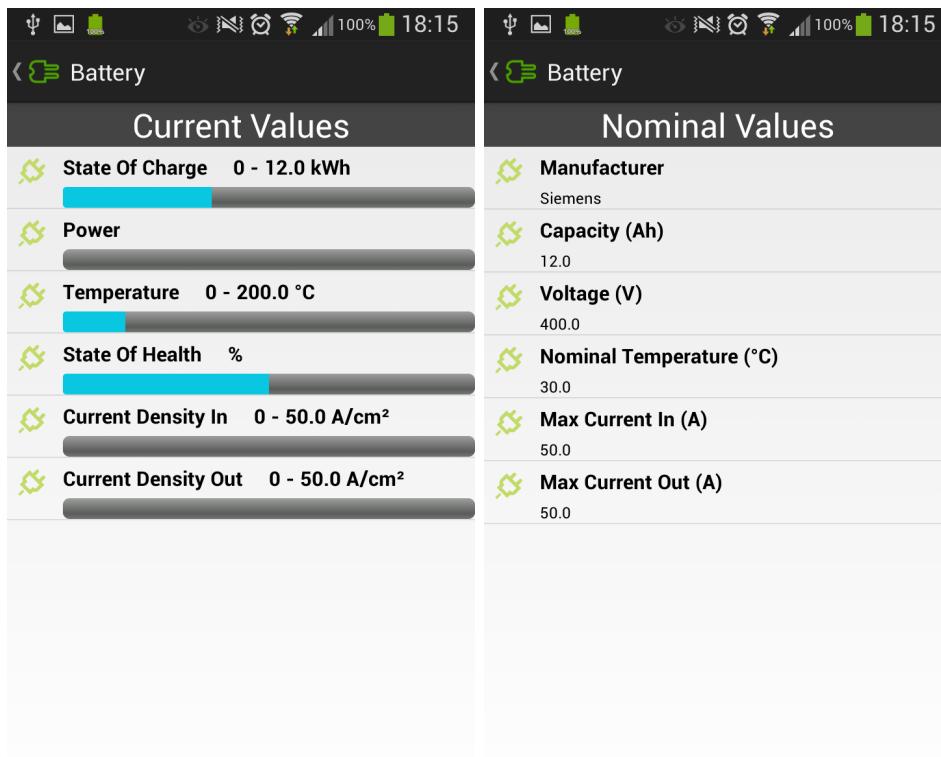
La scelta dell'area dentro la quale cercare le colonnine assume di default la posizione del veicolo come punto centrale, nel caso essa non sia disponibile (modalità *Senza Blue&Me*, Sez. 4.2.3) allora l'utente è costretto a sceglierne dalla mappa. Per accedere alla funzione di scelta del punto si clicca sul pulsante *Map* il quale mostra apre la mappa centrata sulla città in cui ci troviamo e tramite un tocco sullo schermo permette di scegliere il punto di interesse (Fig. 4.4b).

Una volta scelto centrata la nostra area di interesse si procede scegliendo il raggio di ricerca. Di default è impostato a 6 e al massimo si può impostare a 15, di più non avrebbe senso visto che in linea di massima si sposta il punto di interesse.

La barra che permette di scegliere la quantità di energia ha come dimensione massima la capacità della batteria. Nell'immagine in Fig. 4.4a abbiamo una batteria con capacità 40kWh e una quantità di energia residua di 12kWh, questo si capisce dal colore rosso del numero a destra della barra che indica che stiamo tentando di acquistare 0kWh. Spostando la barra verso destra andremo a indicare fino a che punto volgiamo ricaricare la batteria, il numero a destra a questo punto diventa nero. Non si può procedere finché non si acquista almeno 1kWh. Nel caso in cui la capacità e la quantità di carica non siano disponibili allora viene data la possibilità di scegliere qua-

4.3 Funzionalità

4. Applicazione Mobile



lunque quantità di energia, l'utente viene però messo in guardia del fatto che potrebbe acquistare più energia di quella che può contenere la batteria del veicolo.

L'ultimo parametro da impostare è l'intervallo di tempo in cui siamo disposti a ricaricarci. Di default viene proposto un lasso temporale di 3 ore. È possibile variarlo premendo sui pulsanti nel quale è scritta la data i quali mostreranno un calendario e un selettori di orario. Ovviamente sono stati messi dei controlli che impediscono di mettere orari incoerenti come ora di inizio successiva a quella di fine.

A questo punto possiamo inviare la richiesta premendo l'apposito pulsante in alto a destra.

Scelta della risposta

In seguito all'invio della richiesta viene mostrato all'utente un messaggio che invita ad attendere la risposta. Nel caso la richiesta sia fallita allora l'utente viene allertato e rimandato nella schermata di prenotazione con l'invito di cambiare i parametri.

In caso di successo viene mostrata una schermata con tutte le opzioni di ricarica restituite dal servizio cittadino. Come si può vedere in Fig. 4.4c vengono fornite diverse informazioni per ognuna di esse:

- Nome del GCP presso cui avverrà la ricarica.
- Distanza reale dal GCP, calcolata grazie alla libreria UniboGeoTools.
- Orario e prezzo della ricarica.
- Energia stimata per raggiungere la colonnina.
- Dislivello in salita e in discesa.

Le informazioni sul profilo altimetrico ed il contributo energetico vengono mostrate in una schermata a parte accessibile tramite un apposito menu visualizzabile tenendo premuta a lungo un'opzione di ricarica (Fig. 4.4d), questo aspetto verrà approfondito in una sezione a parte (4.3.6).

Viene data inoltre la possibilità di eseguire operazioni di ordinamento delle varie ricariche in base ai parametri sopra elencati ovvero: distanza, prezzo, orario, contributo energetico ecc.. Infine volendo si possono guardare le opzioni di ricarica sulla mappa. Premendo su una di esse viene disegnato il percorso necessario ad arrivarci con la particolarità che i tratti in salita sono colorati di rosso, quelli in discesa di verde e infine quelli in pianura di grigio. Le caratteristiche della mappa sono comunque approfondite nella Sez. 4.3.7.

Scegliendo l'opzione di ricarica viene eseguita la parte restante del protocollo, e nel caso in cui sia ancora valida allora verranno mandati alla schermata che presenta le prenotazioni attive per l'utente (Fig. 4.3d).

4.3.5 Visualizzazione e Ritiro delle Prenotazioni

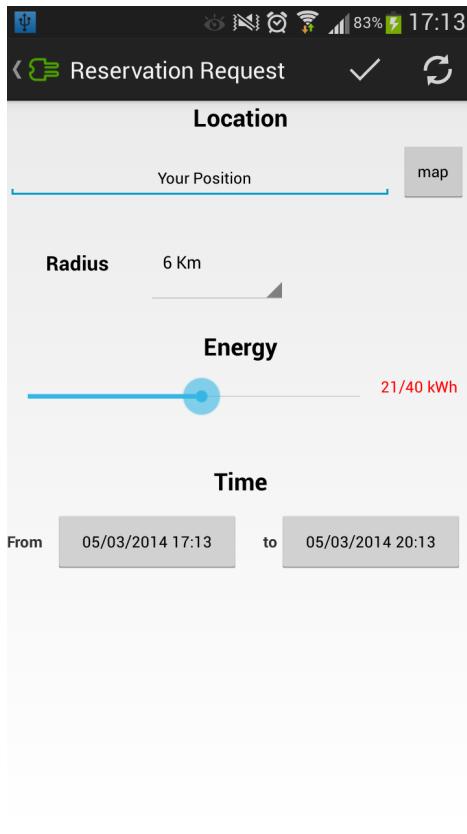
In questa schermata vengono visualizzate le prenotazioni pendenti per l'utente (Fig. 4.3d). Vi si può accedere tramite il menu principale *Reservation List* oppure ci si viene portati direttamente se viene confermata dal sistema l'opzione di ricarica scelta durante il processo di prenotazione.

Selezionando una delle prenotazioni apparirà un menu che permette di mostrarla sulla mappa oppure di ritirarla.

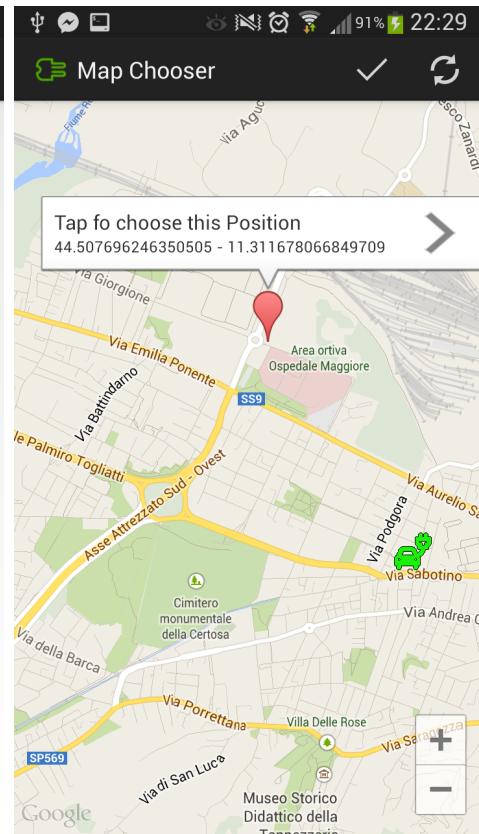
4.3.6 Profilo Altimetrico e Contributo Energetico

Il profilo altimetrico è una delle feature più innovative introdotte nell'applicazione. Può rivelarsi di fondamentale importanza nelle scelte dell'utente il quale, grazie ad essa, potrà scegliere il percorso migliore al fine di raggiungere la sua destinazione. Come vedremo nella sezione relativa al modello della batteria (5.2.4), i veicoli elettrici hanno la possibilità di recuperare energia grazie alla frenata rigenerativa e al recupero in discesa. Questo significa che scegliere una strada con molti tratti in discesa può portare un recupero di energia non indifferente.

La libreria UniboGeoTools, usata a questo scopo, permette inoltre di fare delle stime sull'energia necessaria a percorrere un determinato tratto di strada. Queste informazioni sono mostrate nella schermata *Elevation Profile*:



(a) Inserimento Prenotazione



(b) Scelta di un punto nella mappa

Charge Response		SORT	MAP
Foscolo (4.3 Km)			
From: 6:47am To: 7:26am Price: 0.65€			
0.67 (kWh) Uphill: 28 (m) Downhill: -16 (m)			
Foscolo (4.3 Km)			
From: 7:26am To: 8:05am Price: 0.65€			
0.67 (kWh) Uphill: 28 (m) Downhill: -16 (m)			
Piazza dei Martiri (4.1 Km)			
From: 6:55am To: 7:34am Price: 4.55€			
0.47 (kWh) Uphill: 10 (m) Downhill: -14 (m)			
Piazza dei Martiri (4.1 Km)			
From: 7:34am To: 8:13am Price: 4.55€			
0.47 (kWh) Uphill: 10 (m) Downhill: -14 (m)			
Piazza Roosevelt (4.3 Km)			
From: 6:36am To: 7:15am Price: 0.65€			
0.73 (kWh) Uphill: 26 (m) Downhill: -9 (m)			
Piazza Roosevelt (4.3 Km)			
From: 7:15am To: 7:54am Price: 0.65€			
0.73 (kWh) Uphill: 26 (m) Downhill: -9 (m)			
Vascelli (4.1 Km)			
From: 6:34am To: 7:13am Price: 0.65€			
0.71 (kWh) Uphill: 25 (m) Downhill: -7 (m)			
Vascelli (4.1 Km)			
From: 7:13am To: 7:52am Price: 0.65€			
0.71 (kWh) Uphill: 25 (m) Downhill: -7 (m)			
San Francesco (4.6 Km)			
From: 6:31am To: 7:10am Price: 4.55€			

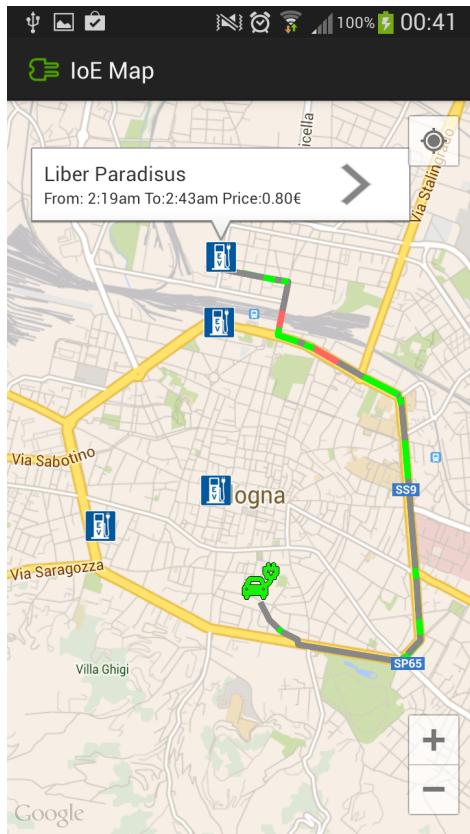
(c) Visualizzazione opzioni di ricarica

Charge Response		SORT	MAP
Foscolo (29.3 Km)			
From: 12:55am To: 1:04am Price: 0.15€			
3.13 (kWh) Uphill: 506 (m) Downhill: -550 (m)			
Foscolo (29.3 Km)			
From: 1:04am To: 1:13am Price: 0.15€			
3.13 (kWh) Uphill: 506 (m) Downhill: -550 (m)			
Righi (33.1 Km)			
Boldrini			
Reserve			
Map			
Show Elevation Profile To Dest.			
From: 2:21am To: 2:30am Price: 0.15€			
3.09 (kWh) Uphill: 499 (m) Downhill: -555 (m)			
Liber Paradisus (35.9 Km)			
From: 11:36pm To: 11:45pm Price: 0.30€			
3.69 (kWh) Uphill: 499 (m) Downhill: -537 (m)			
Liber Paradisus (35.9 Km)			
From: 11:45pm To: 11:54pm Price: 0.30€			
3.69 (kWh) Uphill: 499 (m) Downhill: -537 (m)			
Piazza Roosevelt (30.1 Km)			
From: 12:28am To: 12:37am Price: 0.15€			

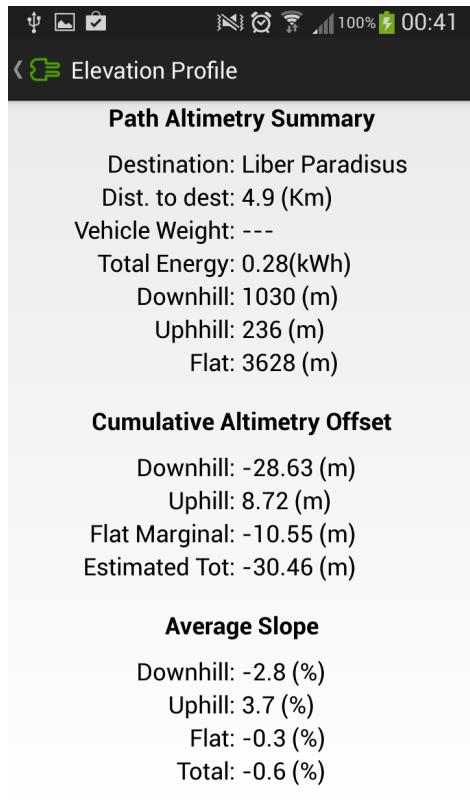
(d) Menu opzioni di ricarica

- **Path Altimetry Summary:** Informazioni riassuntive sul viaggio che stiamo per intraprendere. Come si può vedere in Fig. 4.5b, al di là delle informazioni di destinazione, distanza e peso, viene fornita una stima sull'energia totale che impiegherà il veicolo a raggiungere la destinazione, insieme alle informazioni su quanta discesa, salita e pianura compongo il percorso.
- **Cumulative Altimetry Offset:** In questo riquadro vengono fornite informazioni sul dislivello che caratterizza il percorso (Fig 4.5b).
- **Average Slope:** Fornisce informazioni sulla pendenza media che caratterizza il percorso. Importante in quanto, come detto prima, una pendenza molto elevata può portare ad elevati consumi oppure a un ricavo di energia (Fig 4.5b).
- **Max Slope:** Fornisce informazioni di massima sulla pendenza del percorso, ovvero la pendenza massimo che troveremo in salita e in discesa (Fig 4.5c).
- **Elevation Dependent Energy Ref:** Informazioni sul contributo energetico impiegato per vincere il dislivello. A tal scopo viene utilizzata l'energia potenziale gravitazionale (Fig 4.5c).
- **Energy Profile for Path Length:** Energia impiegata per percorrere il tragitto. Mentre nel caso precedente veniva preso in considerazione unicamente il dislivello qui viene considerata anche la distanza che ci separa dalla destinazione (Fig 4.5c).
- **Altitude Graph:** Grafico che mostra il profilo altimetrico che ci separa dalla destinazione. I tratti in discesa sono in verde mentre quelli in salita in rosso (Fig 4.5d).

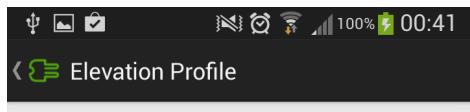
Ulteriori informazioni sul profilo altimetrico vengono date sulla mappa aspetto approfondito nella Sez. 4.3.7.



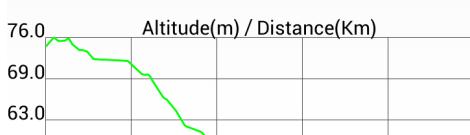
(a) Mappa con Profilo Altimetrico



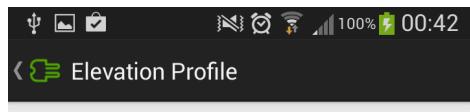
(b) Scelta di un punto nella mappa

**Energy Profile for Path Length**

Downhill:	0.13(kWh)
Uphill:	0.03(kWh)
Flat:	0.45(kWh)
Total:	0.61(kWh)

Altitude Graph

(c) Visualizzazione opzioni di ricarica



(d) Menu opzioni di ricarica

4.3.7 Mappa

La mappa, quando mostrata, viene centrata sulla città di riferimento. Su di essa si può vedere il veicolo che si muove, sia che esso sia simulato sia che sia reale. Premendo il pulsante di localizzazione in alto a destra viene effettuato lo zoom sul veicolo. La mappa può essere mostrata in diverse occasioni:

- **Menu principale:** Dal menu principale si seleziona *Map* e si può accedere alla mappa la quale mostrerà tutti i GCP della città e da qui potremo effettuare una prenotazione cliccando direttamente sul GCP interessato (Fig. 4.6d).
- **Scelta delle Opzioni Ricarica:** Quando il *City Service* ci restituisce le opzioni di carica disponibili possiamo visualizzarle sulla mappa. Se sceglieremo di visualizzare la mappa a partire da un'opzione di ricarica allora verrà anche disegnato il percorso che ci separa dalla colonnina, altrimenti verranno visualizzate tutte le colonnine e per visualizzare il percorso dovremmo selezionare quella interessata (Fig. 4.5a).
- **Prenotazioni:** Dalla schermata delle prenotazioni pendenti possiamo visualizzare sulla mappa e volendo effettuare l'operazione di ritiro semplicemente selezionando la colonnina associata.

Come mostrato in Fig. 4.5a viene disegnato il percorso che ci separa dalla destinazione evidenziando i tratti in discesa, rosso, in salita, verde e pianura grigio.

Premendo su una delle colonnine visualizzate nella mappa viene aperto un popup che mostra informazioni sommarie e una freccia che permette di accedere a un menu con altre opzioni (Fig. 4.6a). Da questo menu si può eseguire una prenotazione, vedere i dettagli sul profilo altimetrico oppure aprire il navigatore messo a disposizione dal Sistema Operativo con partenza il punto dove ci troviamo e destinazione il punto della mappa selezionato (Fig. 4.6c).

4.4 Notifica batteria Scarica

Un funzionalità molto importante è quella delle notifiche che avvisano l’utente quando la batteria sta per scaricarsi (Fig. 4.6b). Quando la batteria scende sotto una certa soglia, che attualmente è impostata al 30%, viene lanciata una notifica all’utente che apparirà nell’area di notifica presente nei Sistemi Operativi Android, la notifica viene lanciata nuovamente a intervalli di 5% di batteria. Premendo sulla notifica Si apre direttamente la schermata destinata alle prenotazioni.

4.5 Implementazione

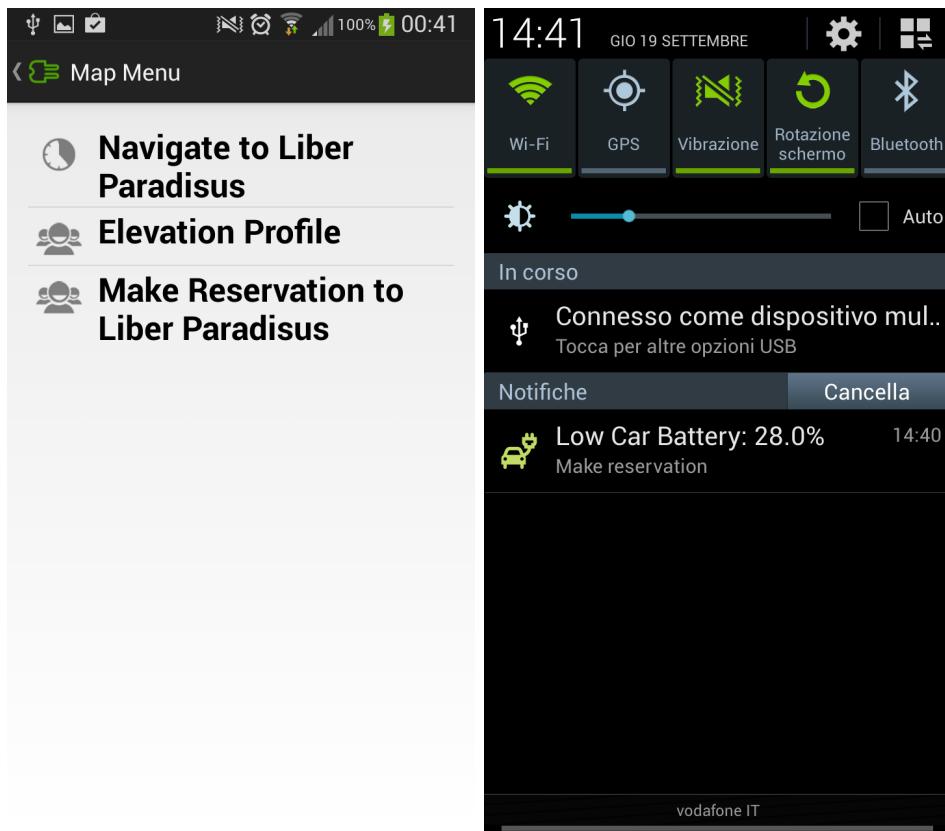
4.5.1 Implementazione

4.5.2 Operazioni Asincrone

Tutte le operazioni che prevedono l’utilizzo della rete, quindi potenzialmente lunghe, sono eseguite all’interno di un **AsyncTask** ovvero una classe messa a disposizione dalla libreria di base di Android che permette di eseguire operazioni asincrone e contemporaneamente aggiornare l’interfaccia grafica. Questo da un lato permette di avere un applicazione fluida in quanto l’interfaccia non rimane bloccata in attesa dei risultati e dall’altro evita il verificarsi di errori dovuti al fatto che Android non permette di modificare l’interfaccia da un thread diverso da quello destinato al disegno di quest’ultima.

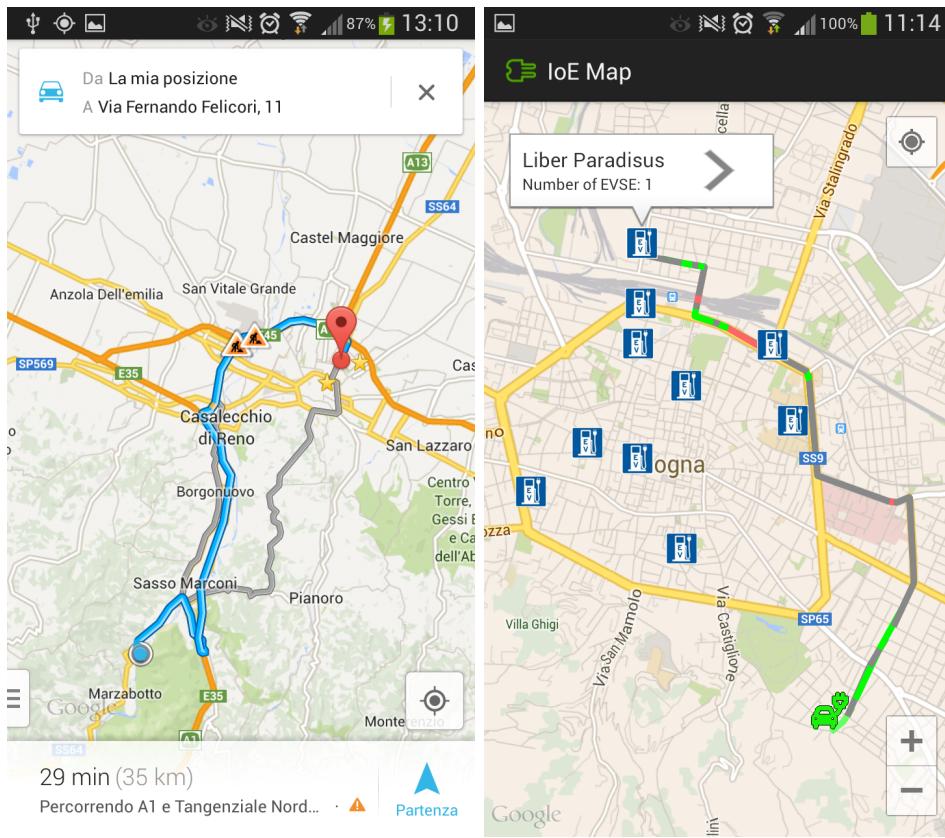
La maggior parte delle operazioni effettuate tramite rete, quindi nel nostro caso scambio di dati con i SIB, reperisce liste di elementi (veicoli, utenti, prenotazioni ecc..) che vengono mostrate all’interno di un apposita tipologia di Activity di Android le **ListActivity**.

Al fine di semplificare la programmazione delle Activity che contengono liste i quali elementi sono reperiti tramite la rete, ho creato la classe **ListLoaderTask<T>** che ne facilita il caricamento asincrono.



(a) Menu Mappa

(b) Notifica batteria scarica



(c) Navigatore Satellitare

(d) Visione di tutti i GCP della città

4.5.3 Activities

4.5.4 Servizi

Capitolo 5

Piattaforma di Simulazione

La piattaforma di simulazione è uno strumento di fondamentale importanza al fine di validare l'infrastruttura software proposta. Risulta inoltre essere un valido strumento per valutare l'impatto dell'introduzione della mobilità elettrica veicolare all'interno di un determinato contesto, grazie ad esso si può prevedere quanti veicoli sarà in grado di supportare la grid, quante colonnine saranno necessarie e che potenza dovranno avere. Risulta quindi uno strumento fondamentale sia sotto il punto di vista dell'amministrazione pubblica/cittadina, che può prevedere un piano urbanistico sostenibile, sia dal punto di vista dei gestori della rete elettrica, i quali potranno valutare la richiesta energetica di tale scenario ed eventualmente prevedere investimenti in quella direzione.

5.1 Architettura

Al fine di poter simulare gli innumerevoli aspetti legati all'Electrical Mobility sono stati usati diversi simulatori/tecniche in symbiosi. In questa sezione verranno introdotte brevemente al fine di introdurre al resto della trattazione.

5.1.1 SUMO

SUMO (Simulator of Urban Mobility) è un simulatore Open Source e multi-piattaforma di traffico urbano progettato per simulare reti stradali di

grandi dimensioni. Sviluppato in C++ è supportato principalmente dall’Institute of Transportation Systems at the German Aerospace Center. La simulazione è di tipo microscopico ovvero ogni veicolo è modellato in modo esplicito, ha un proprio itinerario e si muove individualmente attraverso la rete. Ogni aspetto relativo alla simulazione viene configurato attraverso file XML i quali descrivono la rete stradale, i parametri ed i percorsi di ogni singolo veicolo, ed eventualmente altri aspetti legati alla simulazione come i flussi di traffico oppure la descrizione degli edifici.

SUMO permette di avviare la simulazione in due modalità:

- **Visuale:** La modalità visuale permette di avere un riscontro visuale l’andamento della simulazione tramite un interfaccia che mostra la mappa della rete/città con vista dall’alto. Vengono mostrati tutti i veicoli ed è possibile accedere a tutti i parametri della simulazione. Vengono mostrati inoltre i semafori agli incroci, la segnaletica delle strade e, nel caso siano stati caricati, gli edifici della città. Tutto questo ovviamente impatta notevolmente sulle performance ma, al di là del gradevole effetto visivo, è utile per vedere come evolve la simulazione. Nel nostro caso, ad esempio, è servito per assicurarsi che i veicoli si fermassero alle colonnine, oppure per valutare la quantità di traffico generata in seguito all’inserimento di un determinato numero di veicoli. Molto utile è stato anche in fase di Demo per mostrare il funzionamento del nostro simulatore.
- **Testuale:** Con la modalità testuale vengono stampati nel terminale i messaggi di Warning ed Error nel terminale e se richiesto anche qualche messaggio di debug in più che indica gli step di avanzamento della simulazione. Dopo aver constatato che la simulazione si comporta come ci si aspetta tramite la modalità visuale si passa allora a questa modalità che ha performance assai maggiori. È quindi particolarmente indicata per le simulazioni di lunga durata.

Tools

I file XML che descrivono le simulazioni possono diventare molto complessi qualora si decida di simulare scenari realistici (come Bologna). SUMO

mette a disposizione innumerevoli tool automatici per la generazione dei file di configurazione. In questa tesi prenderemo in esame solo quelli che ci sono stati utili:

- **netconvert**: Genera file con estensione .net.xml della dove viene mappata la rete stradale. La generazione avviene in modo pseudo-casuale, tramite la definizione dei nodi e degli archi che definiscono il grafo della rete stradale oppure, come nel nostro caso, attraverso la conversione da formati esterni(OpenStreetMap, VISUM, VISSIM, OpenDRIVE, MATsim ecc..)
- **polyconvert**: Genera file con estensione .poly.xml dove sono contenute le informazioni relative agli edifici, zone di verde, fiumi laghi ecc.. Anch'esse vengono importate dai file delle mappe in altri formati.
- **duarouter**: Genera file con estensione .rou.xml che descrivono per ogni veicolo il suo percorso, compresi tutti i suoi step intermedi. La generazione dei percorsi avviene applicando un algoritmo di cammino su grafi a scelta tra Dijkistra o A*. I punti di partenza e arrivo vengono generati casualmente da uno script in python messo a disposizione tra i tool di sumo (randomTrips.py).

TRaCI

TraCI (Traffic Controller Interface) è un modulo messo a disposizione da SUMO che permette di interagire con la simulazione in tempo reale tramite un protocollo Client/Server basato su TCP/IP. All'avvio della simulazione SUMO si mette in ascolto su una porta in attesa di messaggi, qualunque linguaggio che supporti il protocollo TCP/IP può dunque modificare lo stato della simulazione oppure ricevere notifiche sul cambiamento di variabili alle quali ci si può sottoscrivere. È proprio TraCI che farà da ponte tra SUMO e l'altro simulatore usato all'interno della nostra piattaforma.

5.1.2 OMNeT++

OMNeT++ è un ambiente OpenSource di simulazione a eventi discreti. È principalmente usato per la simulazione di reti di comunicazione, ma grazie

alla sua architettura modulare ed estremamente flessibile è possibile utilizzarla negli ambiti più disparati come la simulazione di sistemi informatici complessi, architetture hardware o, come nel nostro caso, per supporto alla simulazione veicolare.

Le simulazioni vengono modellate tramite l'impiego di componenti riutilizzabili chiamati *moduli* i quali possono essere combinati tra loro come dei blocchi LEGO.

I moduli possono essere connessi tra di loro attraverso i *gates* e combinati insieme per formare dei moduli composti (compound modules). La comunicazione tra moduli normalmente avviene tramite message passing e i messaggi possono contenere strutture dati arbitrarie (a parte informazioni predefinite tipo i timestamp). Questi messaggi possono viaggiare attraverso percorsi predefiniti dai gates e dalle connections oppure essere inviati direttamente alla loro destinazione, quest'ultima scelta è molto utile nel caso delle comunicazioni wireless.

I moduli, i relativi parametri e i collegamenti fra loro, vengono definiti tramite un linguaggio di alto livello (NED) in appositi file con estensione .ned, mentre la logica viene implementata in una corrispondente classe C++.

OMNeT++ viene distribuito con un IDE basato su Eclipse grazie al quale possono essere eseguite molte operazioni in modo visuale, come ad esempio la creazione e aggregazione di moduli.

Anche OMNeT++ mette a disposizione due modalità di esecuzione della simulazione una visuale (*Tkenv*) e una testuale (*Cmdenv*). La modalità visuale permette di vedere i moduli con i relativi messaggi che vengono scambiati, viene usata in fase di debug o in fase di Demo. La modalità testuale, ovviamente più performante e adatta alle simulazioni batch, mostra solo i messaggi di debug della simulazione insieme allo standard output dei moduli. Per i nostri scopi abbiamo usato solo la modalità testuale.

Un grande punto di forza di OMNeT++ sono gli strumenti messi a disposizione per l'analisi dei dati generati dalle simulazioni, che permettono di applicare, in tempo reale, trasformazioni e aggregazioni tra i set di dati e, in fine, visualizzare i risultati con varie tipologie di grafici: a barre, a linee, istogrammi e molti altri.

Ci sono due tipi di dato che si possono registrare in OMNeT++ i vettori

e gli scalari, ne caso dei vettori si hanno i dati sul piano cartesiano, con il tempo come ascissa e il dato come ordinata, mentre nel caso degli scalari viene registrato un solamente un dato.

5.1.3 Veins

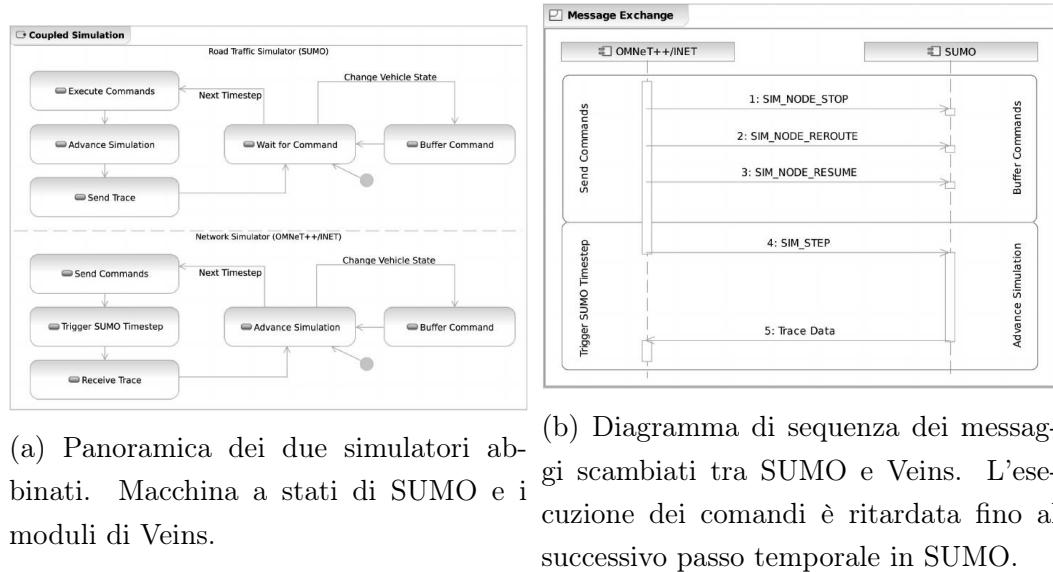
Veins è un framework OpenSource per la simulazione di reti veicolari IVC (Inter-Vehicular Communication). Utilizza OMNeT++ e SUMO in simbiosi. Si appoggia su MiXiM, un framework per OMNeT++, che implementa modelli per reti wireless fisse e mobili (reti di sensori wireless, reti ad hoc, reti veicolari ecc.). La comunicazione con SUMO avviene tramite TRaCI. Ogni volta che nella simulazione in SUMO viene aggiunto un veicolo Veins crea dinamicamente un corrispondente modulo OMNeT++ che permette di controlarlo sotto ogni aspetto (percorso, colore, velocità, accelerazione, parcheggio ecc.).

Il simulatore consiste in un modulo di Veins, il quale è stato opportunamente modificato al fine di avere un ambiente che contiene solo i componenti strettamente necessari allo scopo in quanto le performance sono determinanti al fine di poter avere dei risultati in tempi utili. Infatti sono stati rimossi da Veins i moduli necessari alla comunicazione wireless (nic80211 e ARP), il modulo per la gestione degli ostacoli (obstacles) che era utilizzato per la gestione dello shadowing delle reti wireless.

Il funzionamento di Veins

Veins è il ponte tra OMNeT++ e SUMO e la comunicazione tra i due avviene tramite TraCI. In realtà in mezzo ai due simulatori si trova uno script python, *sumo-launchd.py*, che sta in ascolto sulla prima porta libera che trova, in attesa che venga avviato Veins. Quando Veins viene avviato si connette a questo script il quale lancia SUMO, a questo punto inizia la sincronizzazione tra i due simulatori che avviene tramite staffetta come mostrato in Fig. 5.1a. Per garantire l'esecuzione sincrona a intervalli definiti Veins inserisce in un buffer tutti i comandi da inviare a SUMO (Fig. 5.1b). Ad ogni passo temporale, i comandi contenuti nel buffer vengono inviati. Ciò innesca l'avanzamento del corrispondente passo temporale nella simulazione

del traffico stradale. Al termine dello step temporale di simulazione del traffico stradale, SUMO invia una serie di comandi con lo stato e la posizione di tutti i veicoli istanziati in risposta a Veins. Dopo l'elaborazione di tutti i comandi ricevuti Veins aggiunge i corrispettivi nodi per ogni nuovo veicolo introdotto nella simulazione e rimuove invece i nodi relativi ai veicoli che sono giunti a destinazione. A questo punto la simulazione può avanzare al prossimo step temporale.



(a) Panoramica dei due simulatori abbinati. Macchina a stati di SUMO e i moduli di Veins.

(b) Diagramma di sequenza dei messaggi scambiati tra SUMO e Veins. L'esecuzione dei comandi è ritardata fino al successivo passo temporale in SUMO.

Figura 5.1: Architettura Veins

5.2 Modellazione della Simulazione

L'intera simulazione viene incapsulata all'interno di una Network. Il Network è lo scenario da simulare, all'interno di esso si definiscono i moduli che compongono la simulazione. Come mostrato in Fig 5.2b sono sostanzialmente 4 i moduli che compongono la nostra simulazione:

- **world**: È un modulo di tipo **BaseWorldUtility**, fornito da MiXiM, che rappresenta l'area che circoscrive lo scenario simulato. Come configurazione richiede di definire la grandezza dello scenario in metri. La mappa usata in SUMO non può essere più grande delle dimensioni definite in questo modulo.

- **manager:** È un modulo di tipo `TraCIScenarioManagerLaunchd`, fornito da Veins, che mette in comunicazione OMNeT++ con SUMO. Tutte i messaggi inviati a TraCI passano da questo modulo (l'implementazione vera e propria della comunicazione con TraCI avviene nel modulo padre `TraCIScenarioManager`). Questo modulo è fondamentale in quanto è quello che crea un modulo OMNeT++ per ogni veicolo di SUMO.
- **cityService:** Rappresenta la grid, infatti contiene i GCP e gli EVSE. Ricopre anche la funzione di raccoglitore statistiche globali sulle colonnine e i veicoli (Sez: 5.2.2).
- **connectionManager:** Questo modulo si occupa della comunicazione tra moduli ma è inutilizzato. Non l'ho rimosso per questioni di compatibilità con Veins.

Ogni volta che SUMO crea un veicolo Veins si occupa di creare il corrispondente modulo in OMNeT++, il modulo in questione è Car (Fig. 5.2a). Car in realtà è un modulo composto ovvero un contenitore, privo di implementazione, che contiene altri moduli. Di default, conterrebbe tutti i componenti relativi alle comunicazioni wireless in quanto sarebbe il target di Veins, ma io li ho rimossi in quanto non utili, per ora, nel nostro scenario.

Al fine di rendere la simulazione il più possibile attinente alla realtà risulta necessario implementare i modelli di carica e scarica dei veicoli e i modelli comportamentali degli utenti nonché l'implementazione dei moduli di comunicazione con il servizio cittadino. Per svolgere questi compiti è stato necessario arricchire la definizione di Car con 3 nuovi moduli:

- **TraciMobility:** È un modulo fornito da Veins che mette in comunicazione il veicolo con OMNeT++ tramite TRaci.
- **CarLogic:** È il modulo principale, in esso è implementata la logica del veicolo.
- **Battery:** Implementa il modello di carica e scarica del veicolo.
- **DriverBehaviour:** In questo modulo sono implementati i comportamenti che l'utente assume dinanzi a determinate scelte.

Ogni modulo contiene dei parametri che permettono di cambiarne il comportamento, grazie a OMNeT++ diventa semplice lanciare molteplici simulazioni con diversi set di dati. Particolarmente interessante è la possibilità di eseguire la simulazione senza prenotazione per poter confrontare le differenze di occupazione delle colonnine rispetto allo scenario con prenotazione.

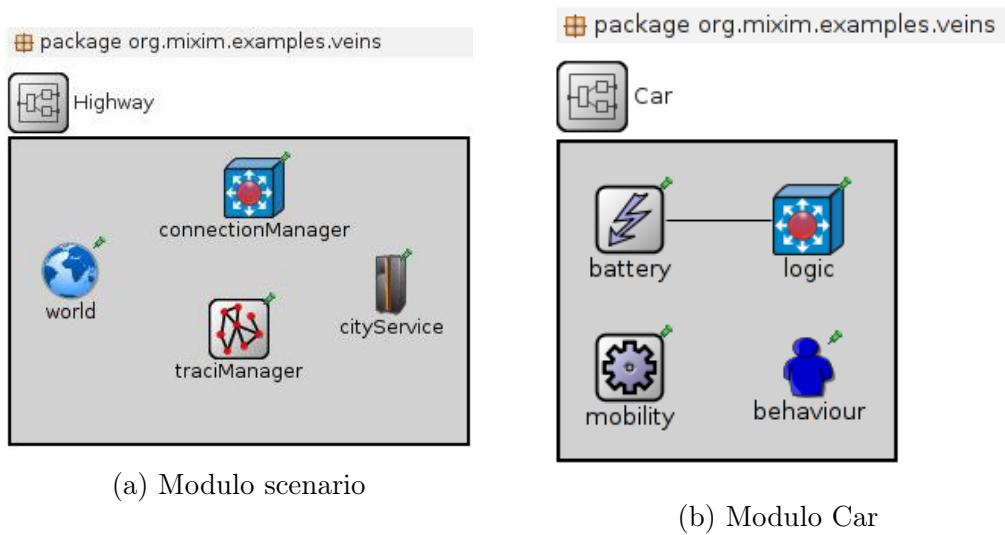


Figura 5.2: Moduli OMNeT++

5.2.1 Ciclo di Vita dei moduli

Essendo OMNeT++ un simulatore a eventi discreti i moduli, in esso implementati, non fanno nulla finché non viene schedulato un evento. Gli eventi vengono schedulati attraverso messaggi inviati dai moduli stessi. Sostanzialmente si tratta di decidere quando, a chi e cosa mandare. Il quando è il tempo di simulazione e quindi deve essere nel futuro o al massimo nel momento attuale di simulazione, a chi è il modulo destinatario e cosa è il messaggio da mandare. OMNeT++ fornisce un messaggio di base, `cSimpleMessage`, ma, come vedremo più avanti, è possibile definirsi messaggi propri più complessi.

All'interno di questa simulazione non avviene molto scambio di messaggi tra i diversi moduli, ragion per cui risulta necessario auto-inviarsi i messaggi al fine di mantenere vivo il modulo.

Quando Veins crea un modulo il corrispondente veicolo in SUMO viene aggiornato ogni 0.1 secondi, che è lo step temporale di default. In OMNeT++ invece siamo noi a decidere ogni quanto aggiornare un modulo attraverso il meccanismo degli auto-messaggi. Schedulando i messaggi in modo intelligente si può guadagnare in performance in quanto ci si può inviare il messaggio solo quando è realmente necessario.

Quando un modulo Car viene creato viene chiamata la funzione `initialize()`, della quale il programmatore può eseguire l'override, nella quale dopo le opportune inizializzazioni, è necessario auto-schedularsi un messaggio. I messaggi vengono ricevuti dalla funzione `handleMessage()` alla quale viene passato un riferimento al messaggio stesso. Da dentro questa funzione si può implementare la logica del modulo. Il modulo continua a vivere finché, il corrispondente veicolo in SUMO, non giunge a destinazione momento in cui Veins, attraverso la classe `TraCIScenarioManager`, si accorge che il veicolo non è più nella simulazione e quindi lo elimina anche da OMNeT++ il che causa una chiamata alla funzione di terminazione `finish()`.

5.2.2 CityService

Il modulo CityService simula la grid, al suo avvio carica le informazioni relative ai GCP presenti nello scenario simulato da un file XML. Il file utilizzato è lo stesso del servizio cittadino (Sez. 3.4.3). Le colonnine caricate vengono trasformate in oggetti C++ al fine di poter essere usate dai veicoli virtuali.

Questo modulo, attraverso il settaggio di parametri esterni, provvede ad abilitare/disabilitare le prenotazioni, e inoltre decide il tasso di penetrazione di veicoli elettrici nello scenario.

Quelli mostrati di seguito sono i parametri del modulo:

- **gcpList**: Percorso del file XML che contiene la definizione dei GCP. Deve essere lo stesso utilizzato dal servizio cittadino reale.
- **electricalVehicleFreq**: Questo parametro stabilisce la frequenza con cui viene immesso un veicolo elettrico nella simulazione. È un numero che può variare da 0 a 100 e, più precisamente, indica con quale

probabilità il veicolo inserito nella simulazione è elettrico. È necessario ricordare che il numero e la frequenza con cui i veicoli vengono immessi nella simulazione è determinato da SUMO attraverso i suoi file di configurazione.

- **maxElectricalVeh:** Impone un limite superiore al numero di veicoli che possono essere presenti nella simulazione in un determinato istante. Questo significa che se è stato raggiunto il massimo numero di veicoli, e uno di questi lascia la simulazione per un qualunque motivo, allora verrà rimpiazzato da uno nuovo (sempre ammesso che SUMO generi altri veicoli e che la statistica sia favorevole). Se viene impostato a -1 allora non ci sarà nessun limite al numero di veicoli.
- **reservationEnabled:** Abilita/Disabilita il protocollo di prenotazione per i veicoli. Il fine di ottimizzare le performance lo scambio di dati con il SIB, necessario per le prenotazioni, viene completamente disabilitato quando quest'ultime non sono attive.

Altra importante funzione svolta da questo modulo è la raccolta di statistiche globali sui veicoli elettrici. I dati raccolti sono tutti in formato vettoriale:

- **chargingVehicles:** In questo vettore viene salvato lo stato di occupazione degli EVSE della città, ogni volta che un veicolo si va a ricaricare aggiunge un'unità al vettore e quando finisce la ricarica l'unità viene rimossa. Questo comporta che come ordinata avremo al massimo il numero totale di EVSE presenti nello scenario.
- **electricalVehicles:** Numero di veicoli elettrici presenti nella simulazione, semplicemente ogni volta che viene aggiunto un veicolo elettrico alla simulazione viene incrementata di un'unità il vettore.
- **vaporizedVehicles:** I veicoli vaporizzati sono quei veicoli che hanno terminato la batteria e quindi vengono letteralmente vaporizzati. Il termine vaporizzati deriva dall'analogo comando di TRaci che permette di rimuovere un veicolo dalla simulazione.

- **leavingVehicles**: Questo veicolo tiene traccia dei veicoli che riescono a lasciare normalmente la simulazione ovvero arrivano a destinazione, evento che ne causa la rimozione da parte di SUMO. In realtà uno degli obiettivi raggiunti è stato proprio quello di dirottare i veicoli che arrivano a destinazione verso una strada casuale. Non è sempre possibile intercettare l'arrivo del veicolo in una determinata strada, questo comporta che qualcuno di essi sfugga e venga rimosso dalla simulazione.

5.2.3 CarLogic

Il comportamento del veicolo è definito dal modulo CarLogic. Questo modulo implementa tutta la logica relativa alla guida del veicolo. In esso sono contenute le informazioni che ne descrivono la tipologia, l'appartenenza, e alcuni comportamenti di base. I comportamenti più complessi sono delegati al modulo DriverBehaviour.

- **userName**: Nome dell'utente che possiede il veicolo
- **userId**: Identificativo dell'utente che possiede il veicolo
- **manufacturer**: Casa produttrice del veicolo
- **model**: Modello del veicolo
- **cRoll**: Resistenza attrito gomme su asfalto
- **cDrag**:
- **across**: Sezione frontale del veicolo
- **rhoAir**: Resistenza dell'aria
- **weight**: Peso del veicolo
- **threshold**: Soglia sotto la quale il veicolo si considera scarico e quindi diviene necessario fare una ricarica. Varia da 0 a 1.

- **minRequestedEnergyKwh:** Quantità minima di energia richiedibile in una richiesta di prenotazione. Questo valore serve nei casi in cui una richiesta non venga accettata dal *City Service*, in tal caso viene diminuita gradualmente la quantità di energia richiesta fino ad arrivare a questa soglia.
- **writeCarStatusOnSib:** Dice se scrivere le informazioni di stato dei veicoli sul *Dash SIB*. Questo permette di vedere lo stato dei veicoli dall'applicazione mobile. Essendo la scrittura sul SIB un'operazione abbastanza onerosa è meglio tenere disattivata questa opzione a meno che non si sia in fase di demo.

Inizializzazione

In fase di inizializzazione la prima cosa che viene fatta è prendere un riferimento a tutti i moduli necessari, tra i quali il *CityService*, che deciderà se il veicolo sarà elettrico o meno. Nel caso in cui sia elettrico allora si procede a reperire tutti i parametri e, se è abilitata la prenotazione, vengono scritte le informazioni del veicolo sul *Dash SIB*. Viene inoltre mandato un comando a SUMO colora il veicolo di verde, funzionalità molto utile sia in fase di debug che in fase di demo.

Nel caso in cui il veicolo non sia elettrico allora vengono eliminati i relativi moduli da OMNeT++ ma il veicolo rimane in SUMO. Questa funzionalità non era prevista da Veins e quindi è stato necessario modificarlo opportunamente per introdurla. L'eliminazione avviene indirettamente, dal momento che un modulo di Veins non può eliminare se stesso, mandando un messaggio al *CityService* con la richiesta di eliminazione.

Un problema di SUMO, almeno per quelli che sono i nostri obiettivi, è che un veicolo una volta giunto alla sua destinazione viene eliminato dalla simulazione. Questo comportamento influisce negativamente sulla simulazione in quanto a noi interessa simulare un periodo di vita dei veicoli lungo abbastanza da poterne studiare diversi cicli di carica e scarica. Per sopperire a questo problema vengono ottenuti, tramite TraCI, gli ID di tutte le strade che compongono il percorso del veicolo e viene preso l'ultimo in modo da poter intercettare l'arrivo del a quest'ultimo e quindi dirottarlo verso un'altra

destinazione casuale. Le altre destinazioni vengono scelte da una lista che viene riempita con gli ID delle strade di destinazione, prese dai veicoli stessi, più lunghe di 50 metri. Il controllo sulla lunghezza serve in quanto è più probabile intercettare il momento in cui il veicolo arriva a destinazione.

A questo punto come ultima operazione viene istanziato un messaggio di tipo **CarMessage**, appositamente creato per mantenere lo stato del veicolo attraverso le varie transazioni di stato. I campi del messaggio vengono inizializzati e il messaggio viene schedulato al tempo attuale di simulazione. Come si vede nel List. 5.1 il messaggio viene creato, viene impostato lo stato del veicolo (Sez. 5.2.3), e infine viene schedulato tramite la funzione **scheduleAt()** al tempo attuale di simulazione che viene fornito dalla funzione **simTime()**. Per schedulare il messaggio dopo 25 secondi sarebbe stato necessario usare **simTime() + 25**.

```

1 carMessage = new CarMessage("CarMessage");
2 carMessage->setCarState(CarState::DRIVING);
3 [...]
4 scheduleAt(simTime(), carMessage);

```

Listato 5.1: Autoschedulazione Messaggio

Gli stati del veicolo

Lo stato del veicolo è definito da un automa a stati finiti come mostrato in Fig. 5.3. Le transazioni tra gli stati avvengono tramite scambio di messaggi nei quali è definito lo stato successivo. I messaggi arrivano alla funzione **handleMessage** la quale controlla se il messaggio arriva dall'esterno oppure se è auto-inviato (**msg->isSelfMessage()**). In quest'ultimo caso allora il messaggio viene inoltrato a una funzione chiamata **handleSelfMessage()**. All'interno di quest'ultima funzione avviene la scelta di quale hanlder eseguire in base allo stato definito nel messaggio (Lst. lst:self-msg).

```

1 void CarLogic::handleSelfMessage(cMessage *msg) {
2     CarMessage* carMsg = check_and_cast<CarMessage *>(msg);
3
4     switch (carMsg->getCarState()) {
5         case CarState::DRIVING:
6             handleDriving(carMsg);
7             break;
8             [...]
9         case CarState::CHARGING:
10            handleCharging(carMsg);
11            break;
12        default:
13            error("Unknown Car State!");
14            break;
15    }
16 }
```

Listato 5.2: Funzione di scelta dello stato

Ogni stato del veicolo ha una sua funzione handler che ne determina il comportamento. Alla funzione viene passato un riferimento al messaggio che contiene informazioni sullo stato del veicolo. L'handler prima di finire rischedula il messaggio con un nuovo stato, o con lo stesso in alcuni casi. La Fig. 5.3 mostra l'automa a stati finiti che descrive il veicolo.

I possibili stati del veicolo sono definiti nell'enumerazione **CarState**. Sarà quindi presente, ad esempio, la funzione **handleDriving** associata allo stato **CarState::DRIVING**, e così via per tutti gli stati del veicolo.

DRIVING: Quando il veicolo si trova in questo stato significa che si sta dirigendo verso la sua destinazione. Ogni volta che viene eseguito fa un controllo sullo stato di carica della batteria e se questa è inferiore alla soglia stabilita dal parametro **threshold** allora il veicolo si considera scarico e quindi riuslta necessario dirigesi a una colonnina. A questo punto si presentano due casistiche:

- Con Prenotazione: se la simulazione è stata eseguita con le prenotazioni attive allora il veicolo deve eseguire il protocollo di prenotazione. Vengono quindi create le triple necessarie a istanziare una richiesta di prenotazione e inserite nel SIB. Se la richiesta va

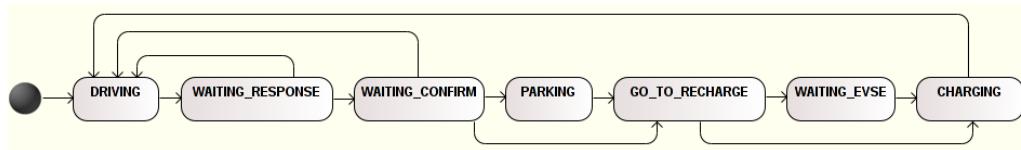


Figura 5.3: Automa a stati finiti che descrive il veicolo, tutti gli stati possono essere finali.

a buon fine allora si imposta come stato successivo **WAITING_RESPONSE**, altrimenti viene rischedulato questo e reiterata la richiesta.

- Senza Prenotazione: Se la simulazione è stata eseguita senza prenotazione allora viene scelto casualmente un GCP tra i 3 più vicini e il veicolo dirige direttamente verso quello.

WAITING_RESPONSE: rappresenta lo stato di attesa della risposta da parte del CS. Siccome

WAITING_CONFIRM:

PARKING:

GO_TO_RECHARGE:

WAITING_EVSE:

CHARGING:

5.2.4 Battery

5.2.5 Driver Behaviour

5.3 Implementazione

???? Esisteva già una versione del simulatore ma era a puro titolo dimostrativo e di demo e soffriva del fatto che era stato sviluppato da diverse

persone (me compreso) in tempi molto brevi e con deadline che corrispondendo a demo internazionali si era costretti a rispettare. Questo ha portato ad avere un codice farraginoso e pieno di memory leak. Basti pensare che la prima versione del simulatore allocava RAM esponenzialmente e già dopo 2000 secondi si poteva arrivare ad avere un'occupazione di 4GB.

La prima cosa che ho fatto, quando ho capito che la situazione stava diventando ingestibile è stato profilare e rifattorizzare il codice. La profilazione è avvenuta tramite il tool Valgrind grazie al quale sono riuscito ad ottenere un consumo di memoria lineare, infatti, dove prima venivano occupati 4GB, sono riuscito a raggiungere il traguardo dei 100MB. La rifattorizzazione del codice invece è stata più complessa in quanto diverse mani hanno messo mano con diversi stili di programmazione.

5.3.1 Logging

5.3.2 SibController

5.3.3 GcpController

GCP e EVSE

5.3.4 Utility

5.4 L'ambiente di simulazione

In questa sezione verranno descritti in dettaglio i componenti necessari a creare un ambiente di simulazione funzionante.

La logica del simulatore è implementata attraverso moduli di OMNeT++. Grazie ad essi sono implementati, i modelli di consumo dei veicoli elettrici, i comportamenti degli autisti e la rete di distribuzione elettrica cittadina. L'unico aspetto non implementato è la guida dei veicoli in quanto è gestita da SUMO.

I file di configurazione di SUMO sono generati da script che in base ai parametri specificati possono variare l'intensità del traffico.

5.4.1 Generazione file di Configurazione

Dopo aver scaricato compilato ed installato tutti i componenti è necessario generare i file di configurazione riguardanti lo scenario che si vuole simulare.

5.4.2 Download Scenario

A questo punto è necessario scegliere quale scenario si vuole simulare. Lo scenario di Bologna è già disponibile nella cartella `simulator/veins-2.1/examples/veins/bologna` siccome è quello di nostro interesse.

Nel caso in cui si sia interessati ad uno scenario diverso da quello di Bologna il modo più semplice per ottenere la mappa desiderata è andare all'indirizzo <http://www.openstreetmap.org/export> e scaricarsi l'area interessata. La dimensione delle mappe scaricabili è limitata onde evitare la saturazione della banda del server. Per sopperire a questa mancanza SUMO mette a disposizione un tool situato in `<SUMO_HOME>/tools/import/osm/osmGet.py` che permette di scaricare mappe di dimensione arbitraria. Per l'utilizzo di questo tool rimando alla documentazione dello script oppure alla pagine ufficiole:

<http://sumo-sim.org/userdoc/Networks/Import/OpenStreetMapDownload.html>.

Profilo Altimetrico

Da notare che le mappe di Open Street Map non contengono le informazioni relative al profilo altimetrico. È quindi necessario arricchire la mappa scaricata con tali informazioni. Il programma utilizzato a questo scopo è Osmosis, presente nella cartella `osmosis` del progetto. In particolare ho usato `osmosis-srtm-plugin_1.1.0` che permette, attraverso l'interrogazione di file SRTM (scaricabili da http://dds.cr.usgs.gov/srtm/version2_1/SRTM3/), di inserire i dati del profilo altimetrico nelle mappe di Open Street Map.

Di seguito viene mostrato l'utilizzo del Osmosis e del relativo plugin considerando `$SRTM_HOME` la cartelle che contiene i file SRTM e `$CITY_NAME` il nome della città. Quindi avendo, ad esempio, `bologna.osm`, ovvero la

mappa della città di Bologna senza dati riguardanti il profilo altimetrico, in output avremo **bologna_srtm.osm**, ovvero la stessa mappa con i dati estratti dai file SRTM.

```
1 osmosis -plugin org.srtmplugin.osm.osmosis.SrtmPlugin_loader --read-xml
  "\$CITY_NAME".osm --write-srtm locDir="\$SRTM_HOME" locOnly=true
  repExisting=false --write-xml "\$CITY_NAME"_srtm.osm
```

Da tenere in considerazione il fatto che il comando mostrato è incluso nello script di generazione automatica da me creato al fine di velocizzare la configurazione dello scenario.

5.4.3 Generazione XML di SUMO

SUMO necessita di file di configurazione in XML che descrivono la rete stradale, i poligoni dei palazzi e i percorsi di ogni singolo veicolo. Siccome ognuno di questi file, per essere generato, richiede un apposito comando il quale a sua volta richiede vari parametri, ho creato uno script che data la mappa di una città in formato Open Street Map esegue tutte le operazioni necessarie.

Verranno comunque analizzati tutti i comandi singolarmente in modo da avere una panoramica sulle scelte implementative.

La rete Stradale (.net.xml)

Il file della rete stradale viene generato attraverso il tool **netconvert** direttamente dalla mappa di Open Street Map. Oltre al file **.osm** è necessario anche un file di supporto che istruisca SUMO sui vincoli e i limiti di velocità delle strade importate. Noi ne utilizziamo uno creato ad hoc per il traffico tedesco.

Qui sotto ne riporto un frammento a puro titolo esemplificativo, il file intero si trova in **simulator/veins-2.1/examples/veins/bologna/osm-urban-de.typ.xml**

```
1 <types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
2   <type id="highway.motorway" priority="13" numLanes="2" speed="41.667"
3       oneway="true" disallow="bicycle_pedestrian"/>
4   <type id="highway.motorway_link" priority="8" numLanes="1"
      speed="13.889"/>
```

```

5   <type id="highway.trunk" priority="12" numLanes="2" speed="13.889"/>
6   <type id="highway.trunk_link" priority="8" numLanes="1"
7     speed="13.889"/>
8   <type id="highway.primary" priority="11" numLanes="2" speed="13.889"/>
9   <type id="highway.primary_link" priority="8" numLanes="1"
10    speed="13.889"/>
11   <type id="highway.secondary" priority="10" numLanes="2"
12     speed="13.889"/>
13   ....
14 </types>
```

Di seguito passiamo ad un analisi dettagliata di tutti i parametri passati a **netconvert**:

- **--type-files**: Specifica il file che contiene i vincoli e i limiti, quello citato sopra.
- **--ramps.guess**: Prova a capire dove sono le rampe e ad eseguirne l'importazione
- **--remove-edges.by-vclass**: Siccome Open Street Map include un infinità informazioni del tutto inutili al nostro fine (ferrovie, piste ciclabili, aree pedonali ecc..) con questo parametro si indicano le classi da non importare (**bicycle**,**pedestrian**...)
- **--geometry.remove**:
- **--remove-edges.isolated**:
- **--tls.join**:
- **--osm-files**:
- **--output.street-names**:
- **--output.original-names**:
- **--output-file**:

Siccome Open Street Map include molte informazioni che sono del tutto inutili al nostro fine (ferrovie, piste ciclabili, aree pedonali ecc..) bisogna istruire **netconvert** affinché le escluda dall'importazione.

Appendice A

Installazione Ambiente

A.1 Installazione

Per far interagire tutti gli elementi necessari alla simulazione è necessario installare numerosi framework e librerie. In questa sezione verrà data una guida il più esaustiva possibile per installare e configurare un ambiente funzionante. Verranno inoltre forniti i link specifici per l'installazione di ogni componente qualora insorgano delle problematiche.

Il procedimento di installazione è testato e funzionante su Debian 7 Wheezy (con versioni precedenti potrebbero esserci problemi con le versioni delle librerie) e Ubuntu dalla versione *12.10* alla *13.10*. È stato anche possibile completare l'installazione su MacOSX ma non essendomene occupato personalmente non posso assicurare nulla al riguardo.

A.1.1 Installazioni preliminari

Questi sono i pacchetti che vanno installati su Debian 7 al fine di installare tutti i componenti successivi. Non è sicuro che siano gli unici necessari. È probabile che lo stesso comando vada bene anche per Ubuntu.

```
1 sudo apt-get install bison flex build-essential zlib1g-dev tk8.4-dev  
      blt-dev libxml2-dev libpcap0.8-dev autoconf automake libtool  
      libxerces-c2-dev libproj-dev libproj0 libfox-1.6-dev libgdal1h  
      libboost-dev
```

A.1.2 OMNeT++

AL momento di scrivere questo documento la versione usata per il progetto è la 4.4 ma in generale le versioni dalla 4.2 in su dovrebbero andare bene. Questo è il link per la versione 4.4 http://www.omnetpp.org/omnetpp/cat_view/17-downloads/1-omnet-releases. Dopo aver scaricato il tar.gz lo si estragga e si proceda con l'installazione:

```
1 ./configure
2 make
3 bin/omnetpp
```

Durante l'installazione verrà detto di inserire alcune variabili d'ambiente nel file .bashrc non dimenticarsi di eseguire queste direttive.

In Ubuntu 13.10 si può assistere a un bug che determina la sparizione dei menu di OMNeT++, per risolverlo è necessario impostare la seguente variabile d'ambiente nel file `~/.basrc`:

```
1 export UBUNTU_MENUPROXY=0
```

per maggiori informazioni guardare questa discussione su StackOverflow <http://stackoverflow.com/questions/19452390/eclipse-menus-dont-show-up-after-upgrading-to-ubuntu-13-10>

A.1.3 SUMO

Seppur SUMO sia disponibile tra i pacchetti di Debian/Ubuntu è necessario comunque scaricare i sorgenti tramite SVN di una versione successiva alla 15340 e compilarli. Questo perchè la versione attualmente disponibile tramite il gestore di pacchetti, ovvero la 0.19.0, non supporta l'importazione nelle mappe (i file .net.xml) dei dati del profilo altimetrico, fondamentali per avere un modello di consumo energetico del veicolo realistico.

Quindi i comandi necessari, presupponendo di avere Subversion installato, sono:

```
1 svn co https://sumo.svn.sourceforge.net/svnroot/sumo/trunk/sumo
2 make -f Makefile.cvs
3 ./configure
4 make
5 sudo make install
```

Per una trattazione più completa dell'installazione rimando il sito ufficiale
http://sourceforge.net/apps/mediawiki/sumo/index.php?title=Installing_Linux_Build

A.1.4 SMART-M3

La tecnologia Smart-M3 forisce la SIB, ovvero il database semantico usato per lo scambio di informazioni tra i vari componenti del sistema. Noi utilizzeremo nello specifico la RedSIB sviluppata da ARCES e basata su un progetto di Nokia (Nokia C Smart M3). La versione supportata dal nostro ambiente è la 0.9 ma anche le successive dovrebbero andare bene. Il link per il download è questo: http://sourceforge.net/projects/smart-m3/files/Smart-M3-RedSIB_0.9/. Una volta estratto il tar.gz al suo interno troveremo sia i sorgenti che i pacchetti per Debian. Nel caso si intenda compilare i sorgenti rimando alle istruzioni contenute all'interno del pacchetto. Qui ci limiteremo a installare i deb attraverso gli script forniti:

```
1 sudo ./install.sh      #per architetture x86
2 sudo ./install_x64.sh #per architetture amd64
```

All'interno del pacchetto viene data la possibilità di utilizzare Virtuoso come database RDF ma, seppur probabilmente sia più performante, non lo utilizzeremo in quanto è una feature introdotta recentemente e quindi non abbastanza testata.

A.1.5 KPI_Low

La libreria KPI_Low è un API scritta in C che, attraverso il protocollo SSAP, permette di interfacciarsi alla SIB. È stata scritta da Jussi Kijander, un ricercatore del VTT Technical Research Centre of Finland, e successivamente modificata da Federico Montori di UNIBO per aggiungervi il supporto alle query SPARQL. Io l'ho modificata al fine di rimuovere dei Memory Leak trovati grazie al tool Valgrind. In quanto la versione della libreria non è quella originale è necessario usare la nostra versione che si trova nella cartella **kpi_low_mod** nella root del progetto. Le KPI_Low necessitano della libreria SCEW per il parsing XML, la quale non si trova

nei repository di Debian/Ubutnu, è quindi necessario scaricarla dal seguente indirizzo <http://nongnu.askapache.com/scew/scew-1.1.3.tar.gz> e compilarla. Una volta scaricata estrarla e spostarsi nella cartella estratta:

```
1 ./configure
2 make
3 sudo make install
```

Adesso possiamo procedere con l'installazione delle KPI_Low, spostarsi dunque nella cartella **kpi_low_mod**:

```
1 ./autogen.sh
2 ./configure
3 make
4 sudo make install
```

per istruzioni più dettagliate guardare il documento **kpi_low_mod/KPI_Low.pdf**

A.1.6 Importare il progetto in OMNeT++

Adesso che abbiamo predisposto l'ambiente possiamo procedere con l'importazione in OMNeT++ del simulatore e con la compilazione. Apriamo OMNeT++, se è il primo avvio ci chiederà che Workspace usare proponendoci uno predefinito, in tal caso noi sceglieremo la cartella **simulator** all'interno della root del progetto. Probabilmente verrà chiesto anche se si vuole abilitare il supporto ai framework MiXiM e INET e se si vogliono importare i porgetti di esempio, in entrambi i casi diciamo di no. Nel caso in cui il workspace fosse già impostato allora andiamo su **File -> Switch Workspace -> Other...** e selezioniamo la cartella **simulator** nella root del progetto proprio come sopra. Se a seguito della selezione del workspace **simulator** la scheda dei progetti rimane vuota allora andiamo su **File -> Import... -> General/Existing Project into Workspace -> Next** e come root directory scegliamo **simulator**, dovremmo vedere il progetto **veins-2.1** nel riquadro **Projects**, lo selezioniamo e clicchiamo su **Finish**.

A questo punto non rimane che compilare il progetto. La compilazione può avvenire in due modalità:

- **gcc-debug:** Compila includendo le informazioni di debug rendendo possibile l'utilizzo di **gdb** per analizzare il funzionamento del programma. OMNeT++ mette a disposizione un front-end visuale per **gdb** che permette di inserire breakpoint nel sorgente ed eseguire l'avanzamento step a step. Inoltre permette di visualizzare il contenuto delle variabili durante l'esecuzione semplicemente semplicemente spostando il cursore sulla variabile interessata nel riquadro dei sorgenti. Queste funzionalità sono da prendere seriamente in considerazione qualora, a seguito di modifiche, la simulazione dovesse fallire.
- **gcc-release:** Compila non includendo le informazioni di debug e applicando le ottimizzazioni previste dal compilatore **gcc** con il flag **-O2**. Ovviamente questa configurazione è più performante della precedente e andrebbe usata quando, una volta ritenuto stabile il codice, si vogliono eseguire simulazioni batch.

Il cambio di modalità di compilazione si può effettuare tramite: **Tasto DX su veins-2.1 -> Build Configurations -> Set Active -> gcc-debug/gcc-release.**
I file che fanno parte del simulatore si trovano sotto la directory **simulator/veins-2.1/examples**.

Appendice B

Vista al Centro Ricerche Fiat

B. Vista al Centro Ricerche Fiat

Appendice C

UniboGeoTools

Libreria java sviluppata per motivi strani

C. UniboGeoTools

Appendice D

OntologyLoader

Riferimenti bibliografici

Manuali cartacei

- [1] Tarik Al-Ani. «Android In-Vehicle Infotainment System (AIVI)». Tesi di dott. University of Otago, 2012.
- [2] Luca Cabibbo Craig Larman. *Applicare UML e i pattern: analisi e progettazione orientata agli oggetti*. Pearson Italia S.p.a, 2005.
- [3] Alfredo D'Elia et al. *A semantic event processing engine supporting information level interoperability in ambient intelligence*. online. 2013. URL: <http://amsacta.unibo.it/3877/>.
- [6] Jukka Honkola et al. «Smart-M3 Information Sharing Platform». In: *Trans. of the IEEE Symposium on Computers and Communications (ISCC)* (2010).
- [8] Federico Montori. *Design and evaluation of an experimental platform about Internet of Energy for Electrical Vehicles*. online. 2012. URL: <http://amslaurea.unibo.it/3900/>.
- [9] E. Ovaska e A .Toninelli T.S. Cinotti. «The Design Principles and Practices of Interoperable Smart Spaces». In: *Advanced Design Approaches to Emerging Software Systems* (2011).
- [10] Luca Vetti Tagliati. *Java quality programming. I migliori consigli per scrivere codice di qualità*. Tecniche Nuove, 2008.

Siti Web consultati

- [4] Emanuele Dinaro. *Android – Services*. 2012. URL: <http://emanueledinardo.com/wp/android-services/>.

D. OntologyLoader

- [5] Apache Foundation. *Apache Jena*. 2011. URL: <https://jena.apache.org/>.
- [7] Marco Lecce. *Guida Android*. 2011. URL: <http://www.html.it/guide/guida-android/>.
- [11] Péter Török. *Is object pooling a deprecated technique?* 2011. URL: <http://programmers.stackexchange.com/questions/115163/is-object-pooling-a-deprecated-technique>.
- [12] Oleg Varaksin. *Simple and lightweight pool implementation*. 2013. URL: <http://www.javacodegeeks.com/2013/08/simple-and-lightweight-pool-implementation.html>.
- [13] Wikipedia. *Blue&Me*. 2010. URL: <http://it.wikipedia.org/wiki/Blue%5C&%7B%7DMe>.