

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

IoE Internet of Energy

Tesi di Laurea in Laboratorio di Applicazioni Mobili

Relatore:
Chiar.mo Prof.
LUCIANO BONONI

Presentata da:
SIMONE RONDELLI

Sessione (che cazzo ne so?)
Anno Accademico

Sommario

Abstract L^AT_EX.

Indice

| | | |
|----------|--|-----------|
| 1 | Introduzione | 2 |
| 1.1 | Smart Cities | 2 |
| 1.2 | Electrical Mobility | 2 |
| 1.3 | Smart-M3 | 4 |
| 1.4 | Lavori Correlati | 6 |
| 1.5 | Un po di storia | 6 |
| 2 | Servizio Cittadino | 7 |
| 2.1 | Architettura | 7 |
| 2.1.1 | La libreria IoE | 7 |
| 2.2 | La comunicazione con il City Service | 11 |
| 2.2.1 | Protocollo di Richiesta di Prenotazione | 11 |
| 2.3 | Il Protocollo di Rimozione di una Prenotazione di Ricarica | 12 |
| 2.4 | Implementazione | 13 |
| 2.4.1 | Pool Di Oggetti | 14 |
| 2.4.2 | Pool Di Thread | 15 |
| 2.4.3 | Il funzionamento del City Service | 16 |
| 2.5 | Testing | 20 |
| 3 | Applicazione Mobile | 21 |
| 3.1 | Architettura | 22 |
| 3.1.1 | Android | 22 |
| 3.1.2 | Google Play Service | 22 |
| 3.1.3 | Bluetooth | 22 |

| | | |
|----------|--|-----------|
| 3.1.4 | UniboGeoTools | 22 |
| 3.2 | Modalità di esecuzione | 22 |
| 3.2.1 | Simulazione | 22 |
| 3.2.2 | Con Blue Me | 24 |
| 3.2.3 | Senza Blue Me | 24 |
| 3.3 | Richiesta di prenotazione | 24 |
| 3.4 | Activities | 24 |
| 4 | Piattaforma di Simulazione | 25 |
| 4.1 | Architettura | 25 |
| 4.1.1 | SUMO | 25 |
| 4.1.2 | OMNeT++ | 27 |
| 4.1.3 | Veins | 28 |
| 4.2 | L'ambiente di simulazione | 29 |
| 4.2.1 | Generazione file di Configurazione | 29 |
| 4.2.2 | Download Scenario | 29 |
| 4.2.3 | Generazione XML di SUMO | 30 |
| 4.2.4 | Il funzionamento di Veins | 32 |
| 4.3 | I Moduli | 33 |
| 4.3.1 | CarLogic | 34 |
| 4.3.2 | Battery | 35 |
| 4.3.3 | CityService | 35 |
| 4.4 | parametri no scrittura sb | 35 |
| A | Installazione Ambiente | 36 |
| A.1 | Installazione | 36 |
| A.1.1 | Installazioni preliminari | 36 |
| A.1.2 | OMNeT++ | 37 |
| A.1.3 | SUMO | 37 |
| A.1.4 | SMART-M3 | 38 |
| A.1.5 | KPI_Low | 38 |
| A.1.6 | Importare il progetto in OMNeT++ | 39 |

Capitolo 1

Introduzione

1.1 Smart Cities

Città 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 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'efficienza 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, il quale è stato fondato dall'Unione Europea e comprende 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 Smart-M3

M3 è un'architettura middleware per consentire l'interoperabilità delle informazioni in maniera cross-domain, multi-vendor, multi-device, multi-piattaforma. Smart-M3 è la sua prima implementazione Open Source, proposta da SOFIA, un Progetto Europeo

(2009-11), appartenente al framework ARTEMIS. 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.

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 (KPs). L'accesso alla SIB da parte dei KP avviene attraverso lo Smart Space Access Protocol (SSAP), esso consiste in messaggi XML scambiati attraverso socket TCP/IP. Vengono fornite API che implementano il protocollo SSAP in diversi linguaggi.

L'architettura Smart-M3 permette:

- **Interoperabilità dell'informazione:** L'interoperabilità è resa possibile da un modello di dati condiviso che si basa su tecnologie tipiche del Semantic Web.
- **Notifiche al variare dei dati:** Grazie a un meccanismo di sottoscrizione è possibile ricevere notifiche al variare dei dati specificati.

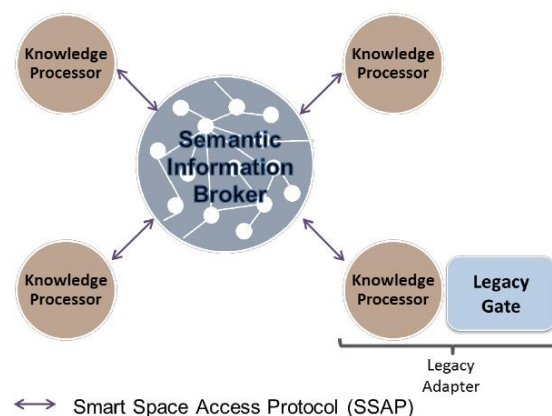


Figura 1.1: Architettura Smart-M3

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 questa opportunità: ovvero rendere più accattivante un applicazione mobile fatta 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*, il quale mi ha introdotto al progetto, sono riuscito ad avere un ambiente più o meno funzionante, la simulazione mi crashava dopo un secondo ma tanto bastava per introdurre un veicolo e fare i test con la nuova applicazione mobile. Successivamente, capendo la vastità del progetto che stava dietro a tale applicazione, decisi di farlo diventare progetto di laurea in quanto era evidente che c'era ancora molto lavoro da fare e la mia innata capacità per complicarmi la vita ha giocato un ruolo fondamentale in tutto questo.

??

Capitolo 2

Servizio Cittadino

Il servizio cittadino (*City Service*) è il cuore dell'architettura software e, al fine di supportare le interazioni tra gli EV e la Smart Grid. Lo scambio di informazioni avviene tramite un SIB cittadino e la struttura dei messaggi è definita all'interno dell'Ontologia.

2.1 Architettura

Il principio di base con cui ho progettato il *City Service* è la modularità e riusabilità. Ho quindi creato una libreria, **ioe_lib**, condivisa tra il servizio cittadino e l'applicazione mobile. Essa fornisce i servizi di base di accesso al *SIB*, nonché un approccio *Object Oriented* ai dati in essa contenuti. Ho infatti implementato, per ogni classe presente nell'ontologia, una corrispondente classe Entity Java e un **Controller** che incapsula la logica di lettura scrittura e aggiornamento dei dati nella *SIB*. Questa scelta progettuale si rifà ai principi di Ingegneria del Software *High Cohesion* e *Low Coupling* [1]

2.1.1 La libreria IoE

Questa libreria implementa la logica applicativa ovvero il nucleo operativo del nostro strato di servizi. Viene infatti sfruttata sia dal *City Service* che dall'applicazione mobile. Viene inoltre utilizzato da un visualizzatore di ricariche, nato da un progetto parallelo al mio, del quale sviluppo si è occupata una ragazza di Ingegneria.

Connessione al SIB

La connessione al *SIB* viene eseguita da una libreria Java (*JavaKPI*), sviluppata da ARCES, che implementa il protocollo SSAP. Ho creato un wrapper di questa libreria che ne semplifica l'utilizzo e aggiunge alcune funzionalità. Essa si trova nel package `it.unibo.ioe.sib`.

La libreria *JavaKPI* rappresenta le triple RDF come vettore di Stringhe di 5 elementi (`Vector<String>`): soggetto, predicato, oggetto, tipo soggetto, tipo oggetto. Per semplificare la gestione delle triple RDF, che sono l'entità di base su cui si basa il SIB, ho creato una classe `RdfTriple` che possiede gli attributi `subject`, `predicate`, `object`, `subjectType`, `objectType` e i corrispettivi `getter` e `setter`.

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` svolgendo le operazioni di parsing anche sui messaggi di risposta. Al posto dei vettori di stringhe ho utilizzato istanze della classe `RdfTriple`. Mentre dove venivano usate liste di triple (es: inserimento multiplo di triple, risultati di query SPARQL) ho sostituito con liste di oggetti di tipo `RdfTriple` (`List<RdfTriple>`). La classe che si occupa di convertire i tipi di dato usati dalla libreria *JavaKPI* ai tipi usati dal wrapper da me creato è `RdfParser`.

La classe `KpConnector` necessita di indirizzo, porta e nome del SIB a cui ci si vuole connettere, questo perché il suo utilizzo deve essere il più generico possibile. Nel nostro caso però le connessioni avvengono sempre verso gli stessi due SIB (CITY e DASH), ho quindi creato una classe factory (`KpFactory`) che, conoscendo i parametri di connessione necessari, crea due istanze di `KpConnector`, una per il CITY SIB e una per il DASH SIB, e restituisce sempre quello evitando quindi la creazione di istanze di oggetti inutili. Essendo la libreria *JavaKPI* non thread-safe viene comunque data la possibilità di creare istanze in modo veloce nel caso si lavori in ambienti multi-thread. Questo aspetto verrà approfondito più avanti dove vedremo come, tramite la tecnica dei pool di oggetti,

possiamo risparmiare il tempo necessario a creare nuove istanze.

Entities

Ogni classe dell'ontologia è stata mappata con una rispettiva classe Java (Entity) nel package `it.unibo.ioe.entity`. Per questa scelta architetturale mi sono ispirato all'ORM (Object Relational Mapping) che è una 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 Il mapping che ho creato è molto semplice e tiene conto solo delle proprietà strettamente necessarie nello strato di servizi e traslascia 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.). Mentre le proprietà che come oggetto hanno un'altra classe sono rappresentate come attributo che come tipo ha l'Entity che corrisponde alla classe.

Serializzazione Alcune Entity sono state opportunamente annotate al fine di poter essere serializzate in XML tramite la tecnologia **JAXB**. Questo è risultato necessario nel caso dei *GCP* che vengono caricati da un file XML, nella cartella del *City Service*, il quale si occupa poi di inserirli nel SIB. Ogni classe inoltre implementa l'interfaccia `java.io.Serializable` al fine di permettere 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 la proprietà `codevcard:hasName`.

Questa situazione viene tradotta nell'Entity:

```

1 @XmlElement(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

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 permette la comunicazione con il SIB.

Lettura Le operazioni di lettura vengono eseguite tramite una query **SPARQL** che preleva dal SIB le informazioni necessarie che poi vengono inserite 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 al fine di 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 utilizza la libreria da un dispositivo mobile connesso a internet tramite rete cellulare (es: EDGE, GPRS, HSDPA ecc...).

Aggiornamento Le operazioni di aggiornamento ricavano i dati aggiornati tramite una query **SPARQL** che 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. Mentre le operazioni di rimozione all'esterno (es. applicazione mobile), per motivi di sicurezza, sono eseguite tramite richiesta al servizio cittadino il quale si occupa di effettuare la rimozione vera e propria.

2.2 La comunicazione con il City Service

In questa sezione analizzeremo in dettaglio come avviene la comunicazione da e verso il servizio cittadino. Per ogni operazione esiste un protocollo basato su scambi di messaggi la quale struttura è definita attraverso 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 vengono scritti sul *SIB* cittadino il quale manda una notifica al *KP* che era sottoscritto a quella particolare modifica. Questo rende il protocollo di comunicazione asincrono.

2.2.1 Protocollo di Richiesta di Prenotazione

Qui verranno descritti tutti i passaggi necessari al completamento del protocollo di Richiesta di Prenotazione, in particolare quali messaggi vengono scambiati. Il fine della Prenotazione è avere la certezza che quando andremo a caricarci troveremo l'EVSE libero. Come già detto in precedenza, i tempi di ricarica per i veicoli elettrici possono essere molto lunghi. È quindi necessario dare all'utente la sicurezza che potrà ricaricare il suo veicolo senza dover rimanere a piedi.

- 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 viene inserita la richiesta, arriva una notifica che ne contiene l'URI dal quale si possono ricavare

tutti 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 notificato quando il *City Service* inserisce la risposta. Le opzioni di ricarica vengono analizzate dall'utente il quale sceglie quella che più si addice 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 è iscritto all'inserimento di triple che come predicato hanno `ioe:confirmByUser` e quindi verifica se l'opzione selezionata è ancora disponibile in tal caso inserisce una tripla siffatta: `[ioe:chargeOptURI ioe:confirmBySystem true]`.
- 5 Acknowledgment da parte dell'utente:** L'utente riceve la notifica della conferma da parte di *City Service*. Se l'opzione è confermata allora 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:** Quando il *City Service* riceve 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.

2.3 Il Protocollo di Rimozione di una Prenotazione di Ricarica

Una volta completata la procedura di prenotazione l'EVSE è diventa inagibile nell'orario richiesto dall'utente. Nel caso in cui un utente voglia ritirare la prenotazione deve

mandare una richiesta al *City Service*. 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.

Il Protocollo si basa su cambio di messaggi proprio come nel caso precedente.

1. **Richiesta ritiro Prenotazione:** L'utente inserisce nel SIB cittadino un'istanza della classe `ioe:ReservationRetire`
2. **Ritiro della Prenotazione:** Il *City Service*, che 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 come unico modo per sapere dell'avvenuta cancellazione deve sottoscrivere ai cambiamenti dell'URI della Prenotazione che sta cancellando.

2.4 Implementazione

In questa sezione discuteremo i dettagli implementativi del *City Service*, dalle tecnologie usate alle scelte Architetture. Principalmente il servizio cittadino deve essere altamente performante in quanto, una volta a regime, dovrebbe poter soddisfare le richieste di centinaia di utenti se non migliaia. Contemporaneamente bisogna preoccuparsi del fatto che l'alto parallelismo non intacchi l'integrità dei dati residenti sul SIB cittadino evitando, ad esempio, che due persone che fanno una prenotazione nello stesso momento, nello stesso EVSE, alla stessa ora non riescano entrambe a completare la procedura di prenotazione.

Il servizio è scritto interamente in Java il che lo rende multi piattaforma, facile da debuggare, e soprattutto permette di usare utilità messe a disposizione del linguaggio per quanto riguarda il multithreading. Inoltre permette di accedere alla moltitudine di librerie scritte per questo linguaggio per i più disparati propositi. 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.

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

2.4.1 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** che sono forniti dalla libreria **JavaKPI**. Come è stato detto nella sezione 2.4 il *City Service* deve supportare connessioni multiple simultanee, ognuna delle quali dialoga con il SIB. Dal momento che la libreria **JavaKPI** non è thread-safe e nemmeno il wrapper che ho creato io lo è, è necessario 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**, che comunque è un oggetto pesante, ho deciso di usare la tecnica dei pool di oggetti. La tecnica consiste nel creare un numero sufficienti di oggetti all'inizio dell'applicazione e quando si necessita di usarne uno lo si chiede al pool il quale lo fornisce al tempo di una chiamata a metodo. Una volta terminato di utilizzare l'oggetto lo si restituisce al pool il quale poi potrà a sua volta cederlo ad un altro richiedente. In questo modo si evita che ogni volta che viene fatta una richiesta al *City Service* ci sia il delay necessario a istanziare una connessione con il SIB.

C'è da dire che vista le ottimizzazioni delle moderne *Java Virtual Machine* e del *Garbage Collector* per quanto riguarda gli oggetti con breve durata questa tecnica può rischiare di abbassare le performance anziché aumentarle [3]. Rimane comunque vantaggiosa nel caso di oggetti la quale creazione può essere abbastanza onerosa come per le connessioni ai database o alla rete.

Il cuore di questo sistema è la classe **ObjectPool<T>** ([4]) 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 mettendoci dentro 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:

```
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 2.2: Implementazione di `ObjectPool`

Come si può vedere è molto semplice creare un pool per un determinato tipo di dato. Una volta creato il pool, per interagire con esso, si usano i seguenti metodi:

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

2.4.2 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 ([2]).

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 2.3: Creazione Pool di Thread

2.4.3 Il funzionamento del City Service

Come visto nella sezione 2.2 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

All'avvio il *City Service* compie innumerevoli compiti:

- **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 thread 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 2.2:
 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 2.2, 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 quando 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 all'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 2.4 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 scaglie 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 corrispondenti 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 ChargeRequestHandler(subscribeResult));
6 } else if (confirmByUserSubId.equals(subscriptionID)) {
7     confirmByUserExecutr.execute(new ConfirmByUserHandler(subscribeResult));
8 } else if (ackByUserSubId.equals(subscriptionID)) {
9     ackByUserExecutor.execute(new AckByUserHandler(subscribeResult));
10 } else if (retireReservationSubId.equals(subscriptionID)) {
11     retireReservationExecutor.execute(new RetireReservationHandler(subscribeResult));
12 } else {
13     logger.error("Unexpected subscription id: " + subscriptionID);
14 }
15
16 xmlToolsPool.returnObject(xmlTools);
```

Listato 2.4: Corpo di RequestDispatcher

2.5 Testing

bla bla bla

ecc ecc ecc

spacca bella ciao

Capitolo 3

Applicazione Mobile

In questo capitolo verrà presentato un esempio di applicazione mobile in grado di connettersi al *City Service* e di eseguire le operazioni di prenotazione e ritiro delle ricariche. Il suo scopo è permettere all'utente di eseguire operazioni di interazioni con la smart-city al fine di ridurre le problematiche derivanti dall'utilizzo di veicoli elettrici. In particolare permette di eseguire richieste e cancellazioni di prenotazioni di ricariche attraverso i protocolli visti nella sezione 2.2.

Inizialmente si limitava solo a queste operazioni ovvero prenotazione e cancellazioni di ricariche e inoltre funzionava solo in presenza del simulatore (4) in quanto prendeva il possesso di un veicolo gestito dal simulatore stesso. 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.

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.

3.1 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 bella è più facile da programmare. La libreria di base per interfacciarsi con il SIB è quella esposta nella sezione 2.1.1.

3.1.1 Android

3.1.2 Google Play Service

3.1.3 Bluetooth

3.1.4 UniboGeoTools

3.2 Modalità di esecuzione

L'applicazione al fine di adattarsi ai diversi scenari possibili offre molteplici modalità di esecuzione, questo per adattarsi alle specifiche richieste per le dimostrazioni del Remote Monitoring. La scelta della modalità di esecuzione avviene nella schermata iniziale come si può vedere in figura 3.2

3.2.1 Simulazione

Questa modalità permette di prendere il controllo di un veicolo contenuto nel simulatore, il quale deve essere avviato con appositi parametri che causino la scrittura dei parametri relativi ai veicoli sul *Dash SIB*. Questo implica che una volta che i abbiamo premuto sul pulsante *Connect* (Fig. 3.2), dobbiamo scegliere l'utente Luciano Bononi in quanto è l'utente di default usato dalle macchine del simulatore (Fig. 3.1a). Una volta selezionato l'utente ci troveremo nel menu principale con solo due opzioni. Questo perché le altre opzioni sono subordinate alla scelta di un veicolo. Una volta selezionato il veicolo vedremo esattamente i parametri che esso ha nel simulatore (posizione, carica, potenza ecc.). La posizione segnata nella mappa dell'app sarà la stessa segnata su SUMO.

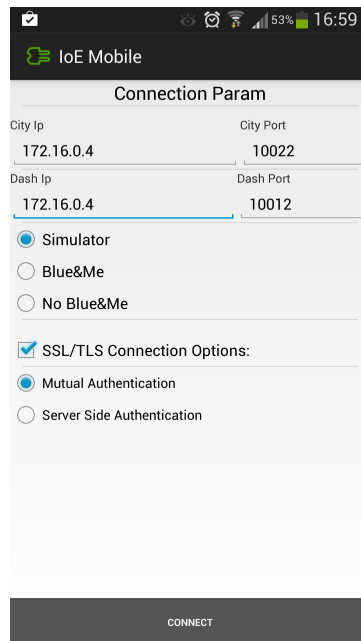
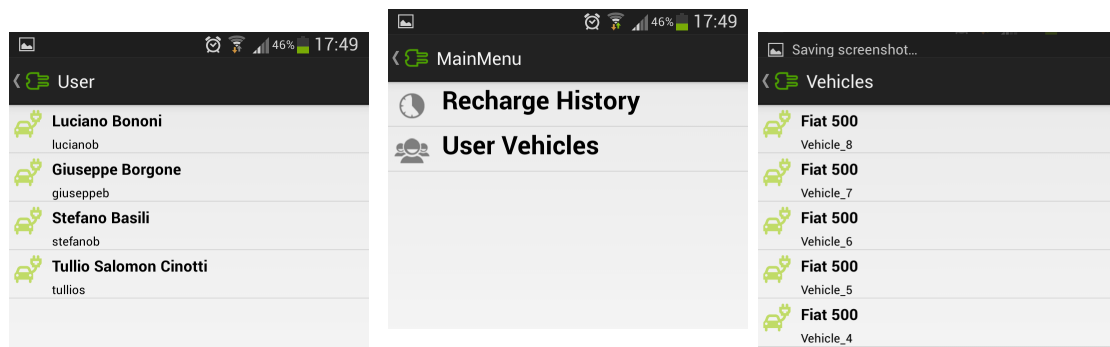


Figura 3.2: Schermata Principale con i parametri di connessione ai SIB e la scelta di modalità di esecuzione.



(a) Selezione Utente.

(b) Menu principale senza
nessun veicolo selezionato

(c) Selezione veicolo

Figura 3.1: Schermate Applicazione Mobile

3.2.2 Con Blue Me

3.2.3 Senza Blue Me

3.3 Richiesta di prenotazione

3.4 Activities

Capitolo 4

Piattaforma di Simulazione

4.1 Architettura

Al fine di poter simulare gli innumerevoli aspetti legati all'Electrical Mobility sono state usate diversi simulatori/tecnologie in simbiosi. In questa sezione verranno introdotte con una breve descrizione. Nelle sezioni successive verranno analizzate dettagliatamente mettendone alla luce gli aspetti utili al nostro fine.

4.1.1 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 Dijkstra 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.

4.1.2 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 utilizzarlo 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 riusabili 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.

4.1.3 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, reercurso, colore, velocità, accelerazione, parcheggio ecc..ti 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 controllarlo sotto ogni aspetto (ercurso, colore, velocità, accelerazione, parcheggio ecc..).

Il nostro 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 ne-

cessari 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

4.2 L'ambiente di simulazione

In questa sezione verranno descritti in dettaglio i componenti del simulatore, da noi creati, ovvero i moduli di OMNeT++, gli script per generare i file di configurazione di SUMO e gli script per lanciare le simulazioni batch.

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.

4.2.1 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.

4.2.2 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 ufficiale:

<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.

4.2.3 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" speed="13.889"/>
7   <type id="highway.primary" priority="11" numLanes="2" speed="13.889"/>
8   <type id="highway.primary_link" priority="8" numLanes="1" speed="13.889"/>
9   <type id="highway.secondary" priority="10" numLanes="2" speed="13.889"/>
10   ....
11 </types>
```

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

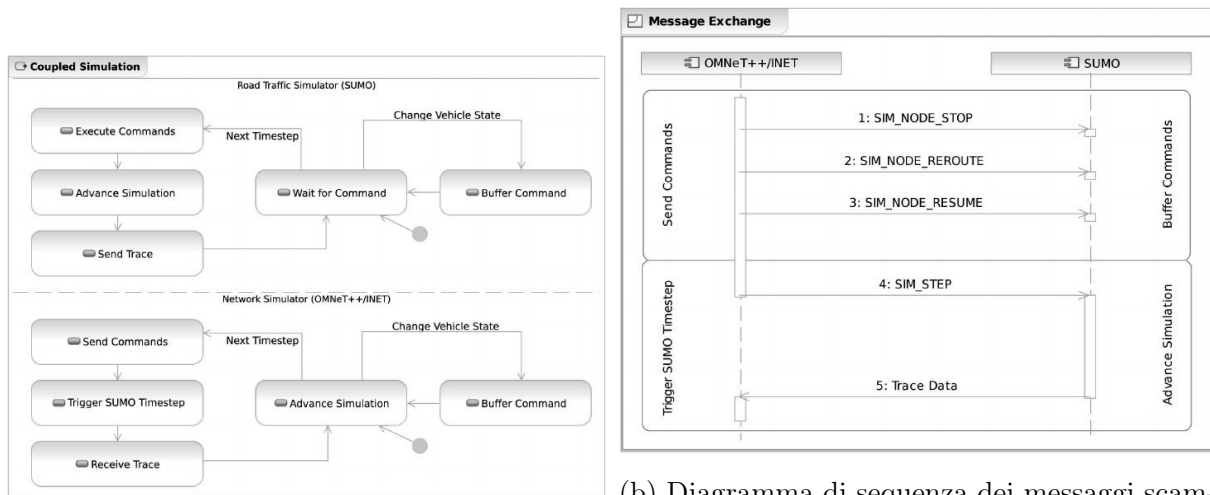
- **--type-files**: Specifica il file che contiene i vincoli e i limiti, quello citato sopra.
- **--ramps.guess**: Prova a capire dove sono le rampe e ad eseguirne l'importazione
- **--remove-edges.by-vclass**: Siccome Open Street Map include un'infinità di informazioni del tutto inutili al nostro fine (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.

4.2.4 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. 4.1a. Per garantire l'esecuzione sincrona a intervalli definiti Veins inserisce in un buffer tutti i comandi da inviare a SUMO (Fig. 4.1b). 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. (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 4.1: Architettura Veins

4.3 I Moduli

Car

(Fig: 4.2).

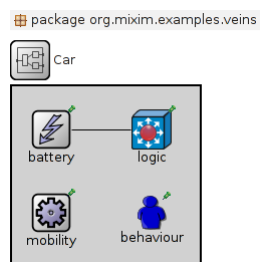


Figura 4.2: Modulo Car

4.3.1 CarLogic

CarLogic è il modulo che implementa la logica di ogni veicolo gestito dal simulatore attraverso un automa a stati finiti.

userName

Nome completo dell'utente che possiede il veicolo

userId

Username dl"utente che possiede il veicolo

manufacturer

Fabbricante

model

Modello

cRoll

Attrito di qualcosa

cDrag

Attrito di qualcos altro

across

Attrito di qualcos altro ancora

rhoAir

Attrito dell'aria

weight

Peso del veicolo

threshold

Soglia sotto la quale il veicolo richiede la ricarica . . .

4.3.2 Battery

4.3.3 CityService

4.4 parametri no scrittura sb

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 essendone 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 libgdallh 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 `~/.bashrc`:

```
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 fornisce 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 Kiljander, 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/Ubuntu, 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 proponendocene uno predefinito, in tal caso noi scegliamo 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 progetti 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 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

UniboGeoTools

Libreria java sviluppata per motivi strani

Riferimenti bibliografici

Manuali cartacei

- [1] Luca Cabibbo Craig Larman. *Applicare UML e i pattern: analisi e progettazione orientata agli oggetti*. Pearson Italia S.p.a, 2005.
- [2] Luca Vetti Tagliati. *Java quality programming. I migliori consigli per scrivere codice di qualità*. Tecniche Nuove, 2008.

Siti Web consultati

- [3] Péter Török. *Is object pooling a deprecated technique?* 2011. URL: <http://programmers.stackexchange.com/questions/115163/is-object-pooling-a-deprecated-technique>.
- [4] Oleg Varaksin. *Simple and lightweight pool implementation*. 2013. URL: <http://www.javacodegeeks.com/2013/08/simple-and-lightweight-pool-implementation.html>.