Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Emulazione di una Colonnina di Ricarica per Auto Elettrica tramite Arduino

Relatore:
Chiar.mo Prof.
Luciano Bononi

Correlatori:
Dott. Luca Bedogni
Dott. Marco Di Felice

Presentata da:
Antonio Carbonara

Sessione I
Anno Accademico 2013/2014

Sono convinto che l'informatica abbia molto in comune con la fisica.
Entrambe si occupano di come funziona il mondo a un livello abbastanza fondamentale.
La differenza, naturalmente, è che mentre in fisica devi capire come è fatto il mondo, in informatica sei tu a crearlo.
Dentro i confini del computer, sei tu il creatore.
Controlli (almeno potenzialmente) tutto ciò che vi succede.
Se sei abbastanza bravo, puoi essere un dio. Su piccola scala.
(Linus Torvalds)

Sommario

Internet of Energy for Electric Mobility è un progetto di ricerca europeo il cui scopo consiste nello sviluppo di infrastrutture di comunicazione, siano esse sia hardware che software, volte alla facilitazione, supporto e miglioramento di tutte quelle operazioni legate al processo di ricarica di auto elettriche.
A tale progetto vi ha aderito anche l'Università di Bologna ed è stato oggetto di studio di Federico Montori e Simone Rondelli. Il primo ha dato il là allo sviluppo del progetto realizzandovi, in una fase embrionale, una piattaforma legata alla gestione di un servizio cittadino (bolognese) per la gestione di ricariche elettriche, un'applicazione mobile in grado di interagire con tale servizio ed un simulatore per la piattaforma.
In un lavoro durato oltre un anno, Simone Rondelli ha ripreso il progetto di Federico Montori riscrivendone le componenti in maniera tale da migliorarne le funzionalità ed aggiungerne anche di nuove; in particolare ha realizzato in maniera efficiente un'applicazione mobile la quale si occupa di gestire la prenotazione di colonnine elettriche di ricarica e di monitorare lo stato attuale di un'auto (peso, livello batteria, ecc...).
Nel marzo del 2014 è cominciato il mio contributo nel contesto di Internet of Energy di cui ne ho ereditato tutta l'architettura derivante dai due sviluppi precedenti.
Il mio compito è stato quello di realizzare (cioè emulare) una colonnina di ricarica auto elettrica, tramite la piattaforma elettronica Arduino, la quale al suo primo avvio informa il database semantico del sistema (SIB) della sua presenza in maniera tale che il simulatore sia in grado di poter far ricaricare un'auto anche a questa nuova colonnina.
Di conseguenza ho fatto in modo di instaurare (tramite socket) una comunicazione tra il simulatore e la colonnina così che il simulatore informi la colonnina che è stata raggiunta da un'auto e, viceversa, la colonnina informi il simulatore sullo stato di ricarica dell'auto in modo che quest'ultima possa ripartire al termine della ricarica.

Ho anche realizzato un'applicazione mobile in grado di comunicare con la colonnina, il cui scopo è quello di ottenere un codice di ricarica che poi l'utente deve digitare per autenticarsi presso di essa.
Realizzando tale tipo di contributo si è data dunque la possibilità di integrare una componente "reale" con componenti simulate quali le auto del simulatore di Internet of Energy e si sono poste le basi per estensioni future, le quali permettano di integrare anche più componenti che si registrano nel sistema e danno dunque la possibilità di essere utilizzate dalle auto elettriche.

Indice

Capitolo 1

Introduzione

1.1 Il problema della Mobilità Elettrica

Ultimamente il problema della mobilità elettrica sta riscuotendo molto successo e viene posto al centro di molte discussioni riguardanti i servizi che una città deve garantire a sostegno dei propri cittadini.
Infatti, grazie allo sforzo di molti Stati volto al miglioramento dell'efficienza energetica per il sistema di trasporti al fine di ridurre le emissioni di anidride carbonica, il problema della mobilità elettrica (Electric Mobility - EM) ha guadagnato investimenti significativi da parte dei governi e dei maggiori produttori di auto (basti pensare a Smart, Renault, BMW, Fiat...).

Tuttavia, nonostante l'ottimismo iniziale rivolto verso l'EM, recenti analisi di mercato hanno dimostrato che l'introduzione di veicoli elettrici (EV) sarà abbastanza limitata nei prossimi anni finchè non saranno sviluppati servizi adeguati che facilitino e supportino la gestione degli EV da parte dei loro utilizzatori. Tali analisi sono state confermate da un' indagine condotta dall'U.S. National Energy Technology Lab, secondo la quale circa il 70% delle persone non comprerà un EV a causa di dubbi riguardanti la disponibilità di colonnine di ricarica collocate presso i luoghi di loro interesse ed i lunghi tempi di ricarica.

È per tale motivo che molte iniziative e progetti Europei sono stati attivati. Però al tempo stesso, data la grande quantità di stakeholder coinvolti nel problema (e.g. gli autisti di auto elettriche, i produttori di colonnine di ricarica, ecc...), risulta abbastanza complicato proporre servizi che garantiscano la massima interoperabilità tra i vari attori e l'eterogeneità delle tecnologie e dispositivi coinvolti.

Negli ultimi anni sono stati proposti molti sistemi di gestione dell'EM, i quali forniscono un'interfaccia Web-based in modo tale da essere sempre raggiungibili dai propri utenti. Tra le varie iniziative è bene citare i progetti *Car Station* ([6]) e *ENELDrive* ([9]) che offrono un servizio di Web mapping per le stazioni di ricarica. Inoltre, data la crescente popolarità e pervasività dei dispositivi mobili, sono state proposte delle applicazioni mobili (sia in ambiente Android che IoS) al fine di aiutare gli automobilisti nel trovare le stazioni di ricarica più vicine alla posizione attuale e di dar loro la possibilità di prenotare degli slot. Tra queste applicazioni vi sono *ChargePoint App* della Coulomb Technologies ([7]), *Blink Mobile Application* di ECOItality ([8]), ma anche l'applicazione sviluppata da Simone Rondelli.

A tutto ciò si lega il progetto europeo Internet of Energy (IoE) for Electric Mobility (descritto nella sezione 1.2) il cui scopo è di tentare di risolvere i problemi succitati tramite lo sviluppo di componenti hardware, software e middleware che siano in grado di garantire interoperabilità tra la moltitudine di stakeholder all'interno di uno scenario di tipo *smart-grid* (l'insieme delle stazioni di ricarica).

Scopo di questa tesi è quello di inserirsi all'interno di tale scenario fornendo la possibilità di inserire dinamicamente colonnine emulate nella smart-grid così da studiarne il comportamento avvicinandoci il più possibile alla realtà, con l' obiettivo di garantire in futuro un servizio cittadino che possa essere più efficiente.

1.2 Internet of Energy (IoE)

Internet of Energy ([3]) è un progetto di ricerca europeo nato ad opera del framework ARTEMIS e racchiude al suo interno il contributo di circa 40 partner europei.
Esso può essere definito come un'infrastruttura dinamica di rete integrata basata sui protocolli di comunicazione standard i quali interconnettono la rete energetica con Internet permettendo così che punti di energia siano distribuiti quando e dove se ne ha bisogno.

IoE si sta occupando di sviluppare componenti hardware, software e middleware che garantiscano connettività sicura ed interoperabilità tramite la connessione di Internet con la smart-grid ([4]) al fine di creare un'infrastruttura di mobilità elettrica. Tale progetto si propone di sviluppare un sistema con architettura di alta efficienza, sistemi di reti intelligenti e innovative riguardanti requisiti di compatibilità, di networking, sicurezza, robustezza, diagnostica, manutenzione ed auto-organizzazione.

IoE inoltre propone soluzioni innovative per interfacciare la rete con la power grid ([10]), con applicazioni per la EM, aiutando nella creazione di una rete di trasporti sostenibile, efficiente, pulita e sicura.

Infine IoE si occupa di supportare lo sviluppo di future stazioni di ricarica tramite la comunicazione di dati al fine di spostare il flusso di elettricità efficientemente ed in maniera fidata utilizzando la smart-grid per facilitare e velocizzare le comunicazioni tra i vari nodi e domini d'energia.

1.3 Contributo al progetto IoE

L'Università di Bologna col contributo di ARCES ha aderito al progetto al fine di valutare l'impatto della smart-grid all'interno del contesto urbano bolognese. Ciò è giustificato dal fatto che la città è sempre più attiva nell'interesse verso le forme di energia alternativa come può essere ad esem-

pio l'installazione di impianti fotovoltaici.
Lo sviluppo del progetto ([5]) ha comportato la realizzazione di un'architettura software volta a garantire l'interazione tra gli utenti (possessori di un veicolo elettrico) e la smart-grid, tale tipo di interazione è garantita da un componente, che vedremo essere molto importante in IoE, denominato City Service. In particolare è stata realizzata un'applicazione mobile tramite la quale qualsiasi utente in possessore di un'auto elettrica e di uno smartphone possa comunicare col City Service per effettuare una richiesta di ricarica in base alle proprie esigenze.
Inoltre l'applicazione fornisce anche la possibilità di monitorare lo stato attuale del proprio veicolo come ad esempio lo stato di carica, il peso ed altri parametri attuali in base ai quali l'utente possa comportarsi di conseguenza decidendo se prenotare o meno la ricarica presso una delle apposite colonnine disposte lungo la città. Tuttavia, data l'impraticabilità nell'effettuare un'analisi quotidiana sui veicoli reali (i quali non sono ancora del tutto dotati di tali tecnologie), è stata anche sviluppata una piattaforma di simulazione che simula, appunto, veicoli elettrici che si muovono su di una mappa rappresentante Bologna e si ricaricano presso le colonnine disposte lungo la mappa quando la carica sta per esaurirsi, il tutto per valutare l'impatto dell'utilizzo dei veicoli elettrici nel traffico bolognese.
All'interno di tale contesto si aggiunge il mio contributo volto all'emulazione di una colonnina di ricarica di auto elettrica al fine di realizzare un'analisi più approfondita la quale potrà servirsi non solo di componenti simulate, ma anche fisiche le quali imitano i comportamenti dei componenti reali in gioco nel contesto di IoE. L'emulazione è stata realizzata grazie alla combinazione di componenti elettriche le quali si poggiano su una board elettrica il cui nome è Arduino Uno, la quale instaura una comunicazione con la piattaforma di simulazione di IoE al fine di far ricaricare i veicoli simulati presso il componente esterno.
Inoltre ho realizzato anche un'applicazione mobile volta ad introdurre i concetti di *codice di prenotazione* e *autenticazione presso la colonnina* all'interno del mondo IoE. Difatti tale applicazione comunica col componente emulato chiedendo un codice che poi l'utente digiterà, tramite l'ausilio di un keypad,

sul componente stesso. Quest'ultimo se accetterà il codice, avvierà il processo di ricarica di un'auto elettrica reagendo fornendo degli output visivi (come la creazione di una barra di ricarica sul suo schermo o l'accensione/spegnimento di led).

1.4 Arduino

Come detto l'emulazione della colonnina di ricarica avviene tramite la piattaforma elettronica Arduino e dunque risulta opportuno fare una piccola introduzione a tale piccolo ma potente componente di sviluppo.
Arduino ([1]) è una piattaforma elettronica di prototipazione open-source i cui componenti elettronici sono made in Italy rivolta ad artisti, progettisti, hobbisti e chiunque abbia interesse nella creazione di ambienti interattivi. Arduino è in grado di interagire con l'ambiente circostante reagendo ad input provenienti da alcuni sensori attivando i componenti ad esso legati come ad esempio le luci led. Il microcontrollore presente sulla sua board è programmato facendo uso di un linguaggio simil C/C++ utilizzato all'interno di un IDE liberamente scaricabile.
Una board di tipo Arduino può essere costruita a mano acquistando i vari singoli componenti oppure acquistando piattaforme già preassemblate dallo store di *arduino.cc* i cui schemi sono liberamente consultabili e possono anche essere adattati a seconda delle proprie esigenze.

Capitolo 2

Architettura

In questo capitolo saranno discusse quelle che sono state le scelte architetturali principali del progetto IoE e quindi saranno descritte le componenti con le quali mi sono dovuto interfacciare per integrare la colonnina di ricarica emulata all'interno di tutto l'ambiente IoE.
Il perno sul quale poggia tutta l'architettura è una piattaforma il cui nome è **Smart-M3**, all'interno della quale risiede un database detto **SIB** in cui tutta l'informazione è rappresentata come un insieme di **triple RDF** corrispondenti ad un grafo diretto i cui nodi ed archi sono univocamente identificati da Uniform Resource Identifier (URI). La caratteristica principale della piattaforma è quella di essere completamente interoperabile, quindi permette a vari agenti dell'architettura, siano essi fisici (esterni) o logici (simulati) di interagire tra loro tramite interfacce. Gli agenti che influenzano l'ambiente Smart-M3 si interfacciano al fine di richiedere informazioni alla SIB. Il loro nome è **Knowledge Processor** e consistono in agenti software indipendenti l'un dall'altro che estraggono (o anche inseriscono o eliminano) informazioni dalla SIB tramite API che implementano un protocollo di comunicazione tramite socket TCP, che prende il nome di **SSAP**. Andiamo a descrivere uno ad uno tutte le componenti architetturali qui nominate.

2.1 Smart-M3

Al fine di gestire lo scenario dell'EM, caratterizzato da una grande varietà dei domini, di piattaforme e stakeholder, IoE ha scelto di utilizzare la piattaforma per l'interoperabilità Smart-M3 come cuore della propria architettura.
Smart-M3 ([16]) è un progetto open-source, sviluppato da ARTEMIS nell'ambito del progetto SOFIA, il cui scopo è quello di fornire un'infrastruttura di condivisione di informazioni tra entità software e dispositivi combinando l'idea dei sistemi di rete, distribuiti e del Semantic Web. Come fine ultimo esso si pone l'obiettivo di abilitare ambienti intelligenti e collegare il mondo reale con quello virtuale.
L'idea chiave di Smart-M3 è legata al fatto che i dispositivi e le entità software possono pubblicare le loro informazioni per altri dispositivi ed entità software facendo uso di database semantici condivisi detti Semantic Information Broker (SIB), tramite l'utilizzo di triple RDF o query SPARQL che interrogano modelli ontologici.
Un'altra idea che sta alla base di Smart-M3 riguarda il fatto che tale tipo di architettura è multi-vendor, multi-device e multi-piattaforma e le parti software sono open-source, infatti sono scaricabili da sourceforge, indipendenti ma interoperabili tra loro. Ciò significa che ciascun dispositivo dell'architettura potrebbe essere composto da parti che sono considerate come partner individuali per l'interazione con un altro dispositivo ed ogni pezzo di software che si occupa di un compito di Smart-M3 deve essere abile ad agire propriamente nei vincoli definiti dall'ambiente, ma anche abile nel sfruttare i vantaggi offerti dall'ambiente.
Questi pezzi di software sono detti M3-agent, risiedono all'interno dei vari dispositivi e sono in grado di comunicare con la SIB all'interno della quale le informazioni sono immagazzinate tramite un grafo RDF.

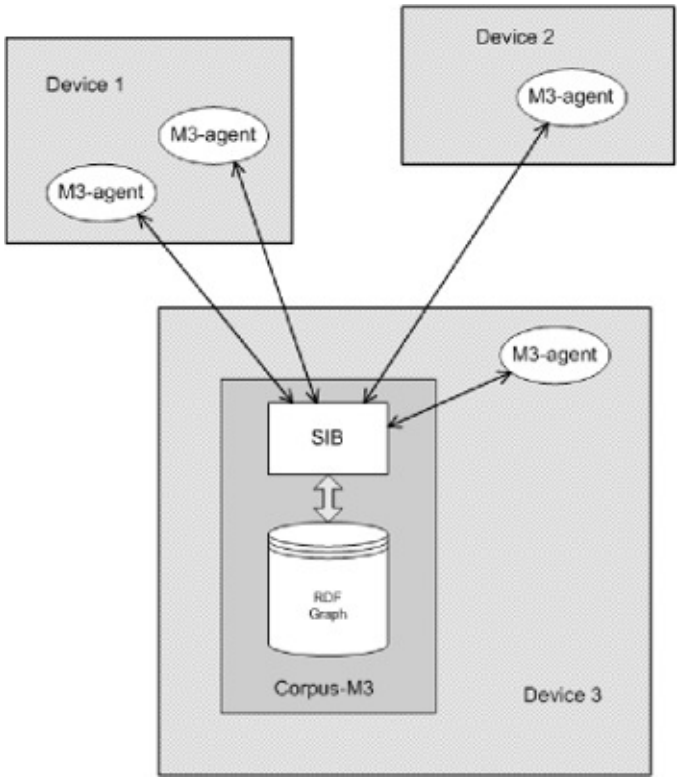

Figura 2.1: Decomposizione Sistema Smart-M3

2.1.1 Il Semantic Web e le triple RDF

Come si è ben intuito dalla sezione precedente, IoE utilizza le tecnologie legate al Semantic Web.
Il Semantic Web ([15]) è un progetto di enormi pretese, ideato da Tim Berners-Lee, al fine di trasformare il World Wide Web in un ambiente in cui i documenti pubblicati sono associati ad informazioni e dati che ne specificano il contesto semantico in un formato che sia di tipo *machine-interpretable* e dunque adatto all'elaborazione automatica.
Il formato principale nel quale le informazioni sono rappresentate è il formato RDF (Resource Description Framework) composto da triple indicanti: un *soggetto*, un *predicato* ed un *complemento oggetto*.

Un esempio di triple RDF utilizzate all'interno di Internet of Energy è il seguente:

```
<http://www.m3.com/2012/05/m3/ioe-ontology.owl#ba21d8dd>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.m3.com/2012/05/m3/ioe-ontology.owl#GridConnectionPoint> .

<http://www.m3.com/2012/05/m3/ioe-ontology.owl#ba21d8dd>
<http://www.m3.com/2012/05/m3/ioe-ontology.owl#hasName>
"Arduino GCP" .
```

Quelli listati sono due statement RDF che possono essere interpretati nelle seguenti due frasi:

"*L'entità* ***ba21d8dd*** (soggetto) *é di tipo* (predicato) *Grid Connection Point (GCP - ovvero la colonnina di ricarica)* (complemento oggetto)"

e

"*L'entità* ***ba21d8dd*** (soggetto) *si chiama* (predicato) ***Arduino GCP*** (complemento oggetto)" .

2.1.2 SPARQL

SPARQL (acronimo ricorsivo che sta per SPARQL Protocol And RDF Query Language) è un linguaggio di interrogazione con sintassi simil-SQL utile nell'interrogazione di grafi RDF le cui query hanno come risultato tutte le triple RDF del grafo che soddisfano i requisiti indicati nella clausola WHERE. Tale linguaggio viene utilizzato dagli M3-agent di IoE per interrogare la SIB e di seguito ne viene mostrato un esempio:

```
PREFIX ioe: <http://www.m3.com/2012/05/m3/ioe-ontology.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?gcp ?gcpName ?gps ?lat ?lon
WHERE {
        ?gcp rdf:type ioe:GridConnectionPoint .
        ?gcp ioe:hasGPSData ?gps .
        ?gps ioe:hasGPSLatitude ?lat .
        ?gps ioe:hasGPSLongitude ?lon .
        ?gcp ioe:hasName ?gcpName .
}
```

Lo scopo di tale interrogazione è quello di richiedere tutte le informazioni delle coloninne di ricarica relative al loro nome e coordinate gps quali latitudine e longitudine in cui la variabile ?gcp è di tipo *Grid Connection Point*, ha dati GPS relativi alla variabile ?gps e nome relativo alla variabile ?gcpName. Quindi la variabile ?gps ha latitudine relativa alla variabile ?lat e longitudine relativa alla variabile ?lon.

2.2 Semantic Information Broker

Il Semantic Information Broker (altrimenti detto SIB) è il componente che sta alla base dell'architettura Smart-M3 il quale si occupa di ospitare il grafo RDF per la gestione e condivisione di informazioni disponibili ai vari agenti dell'architettura.

Gli M3-agent, citati in precedenza, che si occupano di comunicare con la SIB sono detti Knowledge Processor (KP). La comunicazione avviene tramite le Knowledge Processor Interface (KPI) che implementano lo Smart Space Access Protocol (SSAP) in un linguaggio come C, C#, Java, Javascript, Python, PHP (C nelle KPI utilizzate in questo progetto) nascondendo i dettagli dei messaggi XML che vengono scambiati con la SIB tramite socket TCP. Tramite un meccanismo di sottoiscrizioni (subscribe-notify) i KP si registrano presso la SIB in maniera tale che le applicazioni dell'architettura possano reagire ai cambiamenti di informazione che avvengono dinamicamente.
La SIB utilizzata da IoE ed in particolare dall'Università di Bologna è detta RedSib ([11]) che estende l'architettura precedente chiamata Piglet SIB. Quindi è utile descrivere prima l'architettura originale:

- Il Piglet SIB si compone di due processi demoni ed un insieme di librerie (come ad esempio Piglet RDF Store) che si occupano dello storage delle informazioni RDF.

- I processi demoni vengono visti dal Sistema Operativo come processi indipendenti che si scambiano contenuti tramite i D-Bus.
  A tal fine molto utile è il componente *sib-tcp* il quale permette di scambiare informazioni con l'esterno (quindi con i KP) tramite pacchetti tcp.

- Il *sib-daemon* è implementato come un'applicazione multi-thread ove lo scheduler è il thread principale. Ogni volta che un'operazione di richiesta (che può essere di tipo insert, remove, update, query, subscribe o unsubscribe) giunge dal D-Bus, allora viene allocato un nuovo thread.

- Le sottoiscrizioni sono implementate come thread di query che si aggiornano costantemente nella coda delle query finchè non viene eseguita la corrispondente operazione di unsubscribe.

Di seguito vengono descritte le estensioni effettuate con l'implementazione di RedSib:

- Con la componente RedLand-UNIBO sono stati riscritti i supporti per SPARQL e per la sintassi RDF/XML.
- Nel *sib-daemon* delle sottoiscrizioni relative ai pattern delle triple ne viene fatta una copia cache nella RAM al fine di evitare le query a disco quando viene effettuata un'operazione di insert o remove.
- È stato implementato un nuovo meccanismo delle sottoiscrizioni basato sul concetto dello *state-buffer* tramite l'aggiunta di un nuovo metodo detto *Persistent query emulation*.
- È stato aggiunto un meccanismo di **Garbage collector** per le sottoiscrizioni sia in *sib-tcp* che in *RedSib-daemon* che fa unsubscribe automatico delle sottoiscrizioni non più utilizzate.
- Il protocollo SSAP è stato esteso con query SPARQL di richiesta e risposta ed operazioni di insert, remove e update RDF/XML le quali incrementano la conoscenza semantica del database.

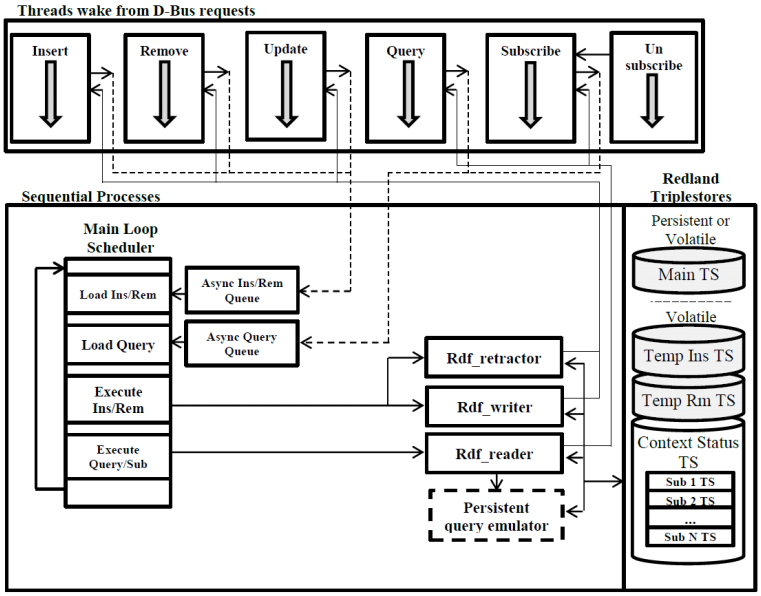

Figura 2.2: Architettura RedSib

2.2.1 City SIB

Il SIB cittadino (o *City SIB*) rappresenta un'istanziazione della SIB descritta poco fa. Al suo interno vengono immagazzinate tutte le informazioni utili allo scenario di Mobilità Elettrica ed è sfruttato dai *city service* per lo scambio di dati con gli M3-agent.
I City Service sono dei programmi con visibilità globale in grado di aggregare dati grezzi e fornire funzionalità complesse ad alti livelli di astrazione. Essi dunque hanno bisogno di essere visibili e per ottenere tale visibilità all'interno dell'intero contesto cittadino, sono implementati tramite i KP che estraggono dati dal City SIB.
Nel City SIB vengono mantenute le informazioni relative alle coloninne di ricarica (i GCP), agli utenti ed ai veicoli.
La sua inizializzazione viene effettuata dal City Service che carica le informazioni dei GCP (tranne quelle relative alla colonnina emulata da Arduino che verrà caricata dinamicamente) da un file XML e le inserisce nella SIB.

2.2.2 Dash SIB

Il *Dash SIB* è un SIB contenente anch'esso tutte le informazioni relative ai veicoli ai quali però vi aggiunge le informazioni relative allo stato della batteria di ciascun veicolo.
Lo scopo della sua ideazione riguarda il fatto che in futuro esso possa essere montato su dei veicoli elettrici reali fungendo come una specie di meccanismo di *caching* tra un veicolo elettrico ed il City SIB condividendo in tempo reale informazioni relative al veicolo. Ovviamente nonostante esso non sia disposto ancora realmente sui veicoli, le funzionalità di un Dash SIB sono testabili tramite i veicoli simulati all'interno del simulatore di Internet of Energy.
Le informazioni di un Dash SIB possono essere controllate tramite smartphone il quale si occupa di colmare il gap tra un EV ed il City SIB al fine, magari, di poter effettuare la richiesta di ricarica migliore in base alla situazione attuale della batteria di un EV.

2.3 Il protocollo SSAP

Smart Space Access Protocol (SSAP) è il protocollo utilizzato in IoE per la comunicazione tra le KPI, dunque i KP, e la SIB. La comunicazione avviene tramite l'apertura di sessioni tra i KP e la SIB, i quali fanno richiesta di JOIN. I messaggi che vengono scambiati sono di tipo XML.
Di seguito le operazioni effettuate nel protocollo di comunicazione:

- **Join** - Operazione tramite la quale un KP richiede l'apertura di una sessione per poter avere l'autorizzazione ad estrarre informazioni dal database semantico.
- **Leave** - Operazione tramite la quale un KP chiude la sessione ad esso associata e quindi viene deregistrato dalla ricezione di notifiche dalla SIB.
- **Insert** - Operazione consistente nell'inserimento di triple RDF nella SIB col fine di aumentare la conoscenza semantica.
- **Remove** - Operazione opposta alla precedente consistente quindi nell'eliminazione di triple RDF dal database semantico.
- **Update** - Operazione consistente nell'aggiornamento del grafo RDF della SIB.
- **Query** - Operazione tramite la quale un KP effettua delle query sia di tipo SPARQL o tramite l'invio di triple RDF, presso la SIB al fine di ottenere o anche modificare informazioni semantiche (questa è l'operazione principale utilizzata nella comunicazione tra l'Arduino GCP e la SIB).
- **Subscribe** - Operazione tramite la quale un KP si sottoscrive presso la SIB al fine di ricevere notifiche durante la sua computazione.
- **Unsubscribe** - Operazione tramite la quale un KP elimina la corrispondente sottoiscrizione.

Esempio di un Messaggio XML scambiato tramite SSAP:

```
<SSAP_message>
  <transaction_type>JOIN</transaction_type>
  <message_type>REQUEST</message_type>
  <transaction_id>1</transaction_id>
  <node_id>ba21d8dd</node_id>
  <space_id>X</space_id>
</SSAP_message>
```

2.4 L'Ontologia di IoE

Utilizzando le tecnologie legate al Semantic Web, il progetto IoE non può non utilizzare il potere rappresentazionale e concettuale dell'ontologia.
Un'ontologia ([12]) infatti è una rappresentazione formale, condivisa ed esplicita (ovvero non ambigua) di una concettualizzazione di un dominio di interesse.
Lo scopo dell'ontologia è volto alla rappresentazione della conoscenza così da descrivere come diversi schemi sono combinati in una struttura dati che contiene le entità rilevanti del modello da rappresentare e le relazioni che descrivono come le entità sono collegate tra di loro.
L'ontologia sulla quale si poggia la conoscenza di IoE è forse quella più utilizzata nel Semantic Web: OWL (Ontology Web Language [13]).
OWL è un linguaggio che può essere visto come una specie di "grande vocabolario" che rappresenta in modo esplicito la semantica delle entità di cui se ne vuole estrarre la conoscenza; esso dunque ha lo scopo di descrivere basi di conoscenze e di effettuare delle deduzioni su di esse così da poter portare a compimento ricerche più complesse eliminando le ambiguità. Nel nostro caso specifico OWL è utilizzata in combinazione con RDF al fine di ricavare informazioni dalla SIB. Nella prossima sezione si andranno ad analizzare i concetti principali dell'ontologia del progetto. Al fine di agevolare la leggibilità delle entità si utilizzerà il prefisso ioe: come abbrevazione dell'uri dell'ontologia <http://www.m3.com/2012/05/m3/ioe-ontology.owl>.

2.4.1 Classi Principali di IoE

Una classe in OWL rappresenta un'entità dello schema che si analizza ed ha il fine di descrivere un concetto. Come avviene nell'ambito dei linguaggi *object-oriented* esistono anche le sottoclassi, le quali ereditano tutte le caratteristiche dalla superclasse e ne possono aggiungere delle altre.
In OWL tutte le classi sono sottoclassi di owl:Thing che rappresenta il concetto di "qualsiasi cosa". In IoE la maggior parte delle classi sono sottoclassi di owl:Thing, ma per essere più precisi si è voluto tenere il concetto di dato distinto dalle altre classi, che sono entità fisiche, tramite la classe ioe:Data che è anch'essa sottoclasse di owl:Thing, ma è superclasse dei vari dati quali possono essere i dati relativi alla batteria di un'auto, alla posizione di un GCP e così via. Ecco le classi principali di IoE:

- ioe:Person - entità che rappresenta il concetto di persona utile nel caso dell'autenticazione di una persona al servizio cittadino ed anche per conoscere le prenotazioni di ricarica associate a quella determinata persona.

- ioe:Vehicle - entità che rappresenta il concetto di veicolo elettrico al quale ovviamente sono associati tutti i dati relativi alla batteria.

- ioe:GridConnectionPoint - entità che rappresenta il concetto di colonnina di ricarica (GCP). Questa è l'entità più utilizzata in questo progetto e le informazioni ad essa associate riguardano i bocchettoni di ricarica (EVSE), associati tramite il predicato ioe:hasEVSE, il nome del GCP (ioe:hasName) ed i dati relativi alla posizione del GCP (ioe:hasGPSData).

- ioe:EVSE - entità che rappresenta il concetto di bocchettone di ricarica chiamato in inglese *Electric Vehicle Supply Equipment*.
  Un EVSE dispone di un profilo di ricarica (ioe:hasChargeProfile) che descrive le caratteristiche principali dell'EVSE, può disporre di più connettori (ioe:hasConnector) ed inoltre ad esso è associata una lista di prenotazioni (ioe:hasReservationList).

- ioe:ChargeProfile - entità utile a descrivere i dati associati ad un EVSE.

Tali dati sono: il voltaggio (ioe:hasVoltage), la potenza (ioe:hasPower), l'intensità di entrata ed uscita (ioe:hasCurrentData) ed il prezzo per unità di ricarica (ioe:hasPrice).

- ioe:Connector - entità che rappresenta il concetto di connettore. Un EVSE difatti può disporre di uno o più connettori al fine di garantire compatibilità internazionale. Il valore del connettore viene indicato con un letterale (e.g. it per quello italiano).

- ioe:UnitOfMeasure - entità che rappresenta il concetto di unità di misura ed è legata ai vari dati dell'EVSE o della batteria (e.g. ioe:Ampere, ioe:Volt, ioe:Watt, ecc.).

- ioe:ChargeRequest - entità che rappresenta il concetto di richiesta di ricarica. Una richiesta di ricarica viene creata quando un utente richiede una prenotazione e contiene le informazioni per descrivere quest'ultima, quali il tempo di ricarica preferito (ioe:hasPreferredTime), l'utente associato (ioe:hasRequestingUser) e così via.

- ioe:ChargeResponse - entità volta ad indicare la risposta fornita dal City Service in seguito ad una richiesta di ricarica. Ad esso viene ovviamente associata la relativa richiesta (ioe:hasRelatedRequest) ed anche delle opzioni (ioe:hasChargeOption).

- ioe:ChargeOption - entità che indica le opzioni relative ad una *Charge Request* contenente l'EVSE (ioe:optionHasEVSE) e le coordinate del GCP (ioe:hasGCPPosition) al quale è stata effettuata richiesta di ricarica ed infine il prezzo della ricarica (ioe:hasTotalPrice).

- ioe:Reservation - entità che rappresenta il concetto di prenotazione. Tale classe viene istanziata se la richiesta di ricarica è andata a buon fine e dunque il sistema ha dato l'ok per riservare una prenotazione. Dunque una volta istanziata, viene associata ad un utente (ioe:ReservationHasUser) e ad un veicolo (ioe:reservedByVehicle) e dunque indica che un determinato EVSE sarà occupato per un certo lasso di

tempo che è associato alla *Reservation* da ioe:hasStartingTimeMillisec e ioe:hasEndingTimeMillisec.

- ioe:ReservationList - entità indicante una lista di prenotazioni associate ad un determinato EVSE.

- ioe:ReservationRetire - entità utile ad esprimere il fatto che un utente vuole ritirare una richiesta di ricarica. Dunque a tale entità viene associato l'utente che ha effettuato il ritiro (ioe:retiredByUser), mentre essa è associata alla *Reservation* tramite il predicato ioe:retiredReservation.

- ioe:Recharge - entità che viene creata quando un veicolo finisce di ricaricarsi al GCP. Questa classe è associata ad una prenotazione ed ha lo scopo di mantenere traccia dell'attività di un utente.

- ioe:Currency - entità indicante la valuta di un prezzo, quindi può essere espressa in euro, dollari, ecc.

2.4.2 Le sottoclassi di ioe:Data

In questa sezione saranno descritte quelle che sono le sottoclassi di ioe:Data, ovvero l'entità utile a descrivere i dati utilizzati dalla batteria, dai GCP ed EVSE all'interno della SIB. Ciascuna sottoclasse è costituita da un'unità di misura ed un valore, ecco la loro descrizione:

- ioe:BatteryData - entità indicante i dati relativi alla batteria di un GCP quali la carica, il voltaggio, la potenza e l'intensità.

- ioe:ChargeData - entità indicante la quantità di carica misurata in KiloWattOra (kWh).

- ioe:VoltageData - entità indicante la tensione di carica elettrica misurata in Volt (ioe:Volt - V).

- ioe:PowerData - entità indicante la potenza elettrica misurata in KiloWatt (ioe:kiloWatt - kW).

- ioe:CurrentData - entità indicante sia l'intensità di entrata (ioe:hasMaxCurrentDensityIn) che di uscita (ioe:hasMaxCurrentDensityOut) misurata in Ampere (ioe:Ampere - A).

- ioe:GPSData - entità indicante le coordinate geografiche di un GCP ovviamente espresse in latitudine (ioe:hasGPSLatitude) e longitudine (ioe:hasGPSLongitude).

- ioe:PriceData - entità rappresentante le informazioni dei prezzi caratterizzata da una valuta (ioe:hasCurrency) e da una scadenza (ioe:hasValidityTo/From).

- ioe:TimeIntervalData - entità rappresentante un intervallo di tempo utile nella prenotazione degli EVSE.

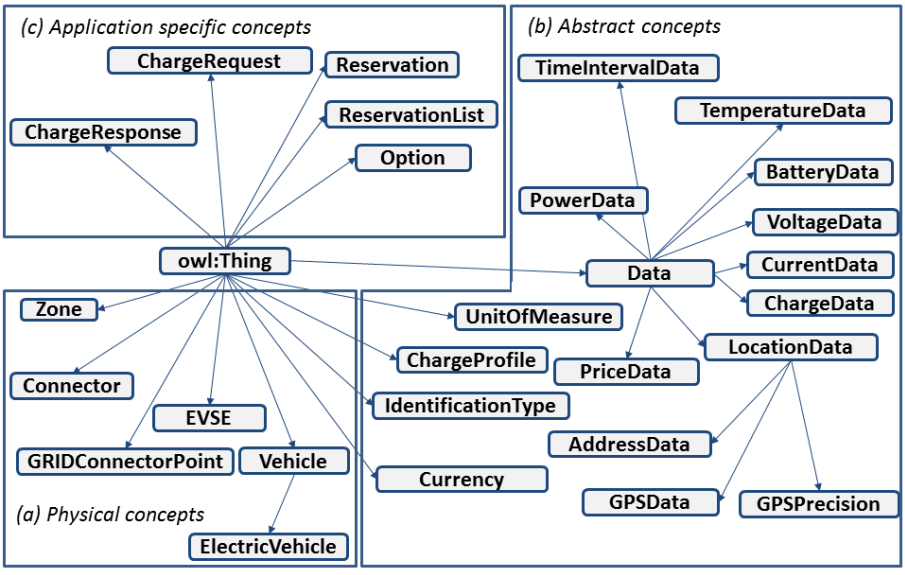

Figura 2.3: Ontologia IoE

2.5 Un esempio di City Service: Prenotazione di una Ricarica Elettrica

Come si può ben intuire il City Service (CS) è la componente principale dell'architettura di IoE in quanto rappresenta il fulcro che gestisce e modella i dati che poi vengono forniti ai cittadini che ne richiedono fruizione.
In questa sezione sarà descritto un esempio di applicazione fornita dal City Service al fine di comprendere le funzionalità del CS e come esso interagisce con le altre componenti architetturali quali la SIB, i KP ed anche con i cittadini. Tale servizio riguarda la prenotazione di una ricarica elettrica tramite un'applicazione mobile (nel caso si voglia disporre di più informazioni a riguardo, risulta utile fare riferimento alla tesi di Simone Rondelli [14]).

Analizziamo quindi questo caso d'uso: in tal caso il CS fa riferimento ad un sistema di prenotazione remoto che quindi consente la prenotazione di slot di ricarica elettrici tramite un'applicazione mobile che è in esecuzione sullo smartphone di un guidatore di un EV. Il sistema inoltre può anche idealmente provvedere a calcolare un *route-planning*, per la selezione dell'EVSE più adatto, in base alle preferenze del guidatore (ma questa è una funzionalità che può essere certamente estesa e migliorata).
Dunque cosa succede? Un KP di un EVSE pubblica, tramite l'ausilio di una KPI, dei dati nel City SIB che sono rilevanti per l'applicazione: lo stato della ricarica, la locazione, il tipo di ricarica, il prezzo e la lista delle prenotazioni attuale. L'applicazione smartphone dell'utente raccoglie i dati del veicolo direttamente dall'EV (tramite la Dash SIB) e quindi notifica l'utente quando la carica della batteria raggiunge una certa soglia.
Dunque l'utente, dopo aver ricevuto la notifica, può decidere di effettuare una richiesta di ricarica al CS inviando informazioni addizionali quali la posizione attuale, lo stato attuale della batteria e le preferenze di ricarica.
A questo punto il CS analizza la richiesta dell'utente, calcola quelle che possono essere le opzioni migliori all'interno di tutto il contesto cittadino e quindi fornisce tutte le possibili alternative calcolate.
Una volta che l'utente ha ricevuto la lista degli EVSE, allora può decidere

qual è quello più adatto alle proprie esigenze ed attende per la conferma finale. Questa conferma finale è opportuna in quanto volta ad evitare problemi di sincronizzazioni dovuti a richieste concorrenti da parte dei vari utenti che operano all'interno del contesto cittadino.

Capitolo 3

Emulazione GCP

Siamo giunti al capitolo in cui verrà spiegato quello che è il fulcro del mio elaborato: **l'emulazione di una colonnina di ricarica**.
Il capitolo consterà di due parti: nella prima verrà descritta la struttura hardware della colonnina, denominata *Arduino GCP*, in cui sarà spiegata la specializzazione di ogni componente hardware ai fini di questo progetto e di come tali componenti sono collegate tra di loro. Nella seconda parte, invece, verrà descritta l'implementazione software e quindi le funzionalità che permettono alle componenti di Arduino GCP di reagire in base ad alcuni casi e di comunicare con la City SIB ed il simulatore di IoE.

3.1 Architettura Arduino GCP

Prima di descrivere quello che è l'applicativo che si occupa di implementare la logica che effettua la parte di emulazione, risulta opportuno mostrare quali sono le componenti hardware che compongono il mio GCP e quindi descriverle.
Ecco l'elenco delle componenti:

1. **Arduino UNO**
2. **Ethernet Shield**
3. **Luci LED** - una verde ed una rossa

4. **Schermo LCD** con 6 pulsanti: up, down, right, left, select, reset
5. **Breadboard**

3.1.1 Arduino UNO

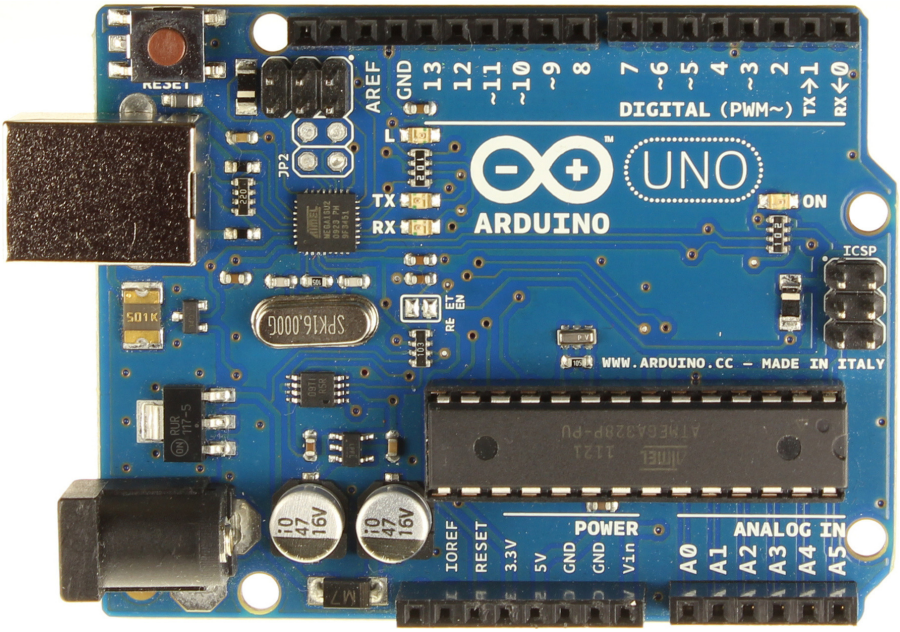

Figura 3.1: Arduino Uno

La piattaforma elettronica Arduino Uno è la board principale sulla quale si poggiano le altre componenti elettroniche che eseguono determinati compiti per l'emulazione di un GCP.
Esso è basato sullo schema ATmega328 e possiede 14 pin digitali e 6 analogici, una porta per la connessione USB, un jack per la connessione ad una presa elettrica ed un bottone di reset.
Arduino lavora ad una tensione elettrica pari a 5V e l'accensione è determinata dal semplice collegamento tra un cavo USB, inserito nell'apposita porta, ed un computer, oppure tramite il collegamento tra un alimentatore ed una presa elettrica. Alternativamente si può decidere di utilizzare supporti di potenza esterni, ma in tal caso si deve tenere presente che la tensione elettrica deve essere compresa tra 7V e 12V in quanto un superamento di questa

soglia potrebbe danneggiare seriamente la board elettrica. Per la potenza elettrica, i pin utilizzati sono quelli contrassegnati da:

- **VIN** - pin al quale si possono collegare supporti di potenza esterni.
- **5V** - pin che lavora alla tensione elettrica tipica della board.
- **3.3V** - pin utile per la compatibilità con altri dispositivi (ad esempio la board Arduino Due lavora a tale tensione) con una corrente massima di 50mA.
- **GND** (ground) - pin per la messa a terra.

Per quanto riguarda la memoria, Arduino Uno possiede una memoria flash di 32 KB (di cui 0.5 utilizzati dal bootloader), una memoria RAM (detta SRAM) di 2 KB ed un'ulteriore unità di memoria detta EEPROM di 1 KB, utile a mantenere alcuni valori quando la board è spenta.
Come detto in precedenza, Arduino Uno è composto da 6 pin analogici (contrassegnati da A0 a A5) e 14 digitali ai quali possono essere collegati dei fili per trasmettere corrente ad altri circuiti elettrici, dunque con le opportune precauzioni si possono estendere le funzionalità della board. Però è anche opportuno considerare di non collegare nulla ai pin digitali 0 e 1 in quanto utili rispettivamente per la ricezione (RX) ed invio (TX) di dati tramite una porta seriale in grado di comunicare con un computer sul quale è anche possibile visualizzare i risultati della comunicazione tramite il terminale dell'Arduino IDE.

3.1.2 Ethernet Shield

L'Arduino Ethernet Shield è una board che permette di collegare la board dell'Arduino ad Internet. Essa è basata sullo schema Wiznet W5100 in grado di fornire stack di rete sia di tipo TCP che UDP ed è in grado di supportare fino a 4 connessioni socket aperte simultaneamente.
La Ethernet Shield si compone di una porta alla quale collegare un cavo ethernet per la trasmissione di pacchetti di rete, uno slot in cui è possibile inserire una card SD nel caso si vogliano immagazzinare dati per realizzare

un mini-server ed un bottone di reset che resetta la scheda ethernet, ma anche la board dell'Arduino.
Al fine di sfruttare le capacità della scheda ethernet, essa può essere inserita "a pila" su Arduino Uno in quanto riporta, quasi fedelmente, gli stessi pin di quest'ultimo. I pin di Arduino Uno utilizzati dalla Ethernet Shield e che non possono essere utilizzati per I/O generale sono: 10, 11, 12, 13.
All'interno di questo progetto la Ethernet Shield viene utilizzata per l'invio e ricezione di pacchetti sia TCP che UDP per lo scambio di dati tramite un cavo ethernet collegato con la shield ad un laptop sul quale agisce l'ambiente di simulazione di IoE ed è attiva la SIB.

Librerie per la programmazione della Ethernet Shield: <Ethernet.h>, <EthernetUdp.h>

3.1.3 Schermo LCD

La LCD Keypad Shield ([2]) è una board dotata di un piccolo schermo LCD ed un keypad con 6 pulsanti.
Essa è molto utile nella creazione di interfacce *user-friendly* che possano permettere, ad esempio, la navigazione all'interno di menu, dunque di scegliere opzioni che permettano all'Arduino di comportarsi secondo una funzionalità scelta da un utente ed implementata da un programmatore della board.
Lo schermo ha una dimensione di 16x2 (ovvero 2 righe e 16 colonne) con sfondo blu e vi è la possibilità di scriverci caratteri ed anche di spostare al suo interno un cursore che è anche possibile visualizzare tramite l'effetto *blink*. I pulsanti di cui si compone il keypad sono: i quattro tasti direzionali **up**, **down**, **right** e **left** per spostare il cursore all'interno dello schermo, un pulsante detto **select** per implementare magari la funzione di conferma della selezione, un pulsante di **reset**.
Anche la LCD Keypad Shield può essere inserita al di sopra della board dell'Arduino ed i pin occupati sono: 4, 5, 6, 7, 8, 9 e vi è anche il pin analogico A0 dedicato a tutti i pulsanti di interazione del keypad tranne quello di reset.

Lo scopo di questo componente all'interno del progetto consiste nel fornire informazioni visuali all'utente, infatti descrivendo in breve quello che è il comportamento della fase di comunicazione tra il simulatore di IoE ed il GCP emulato, questo è quanto succede: al momento dell'avvio della board, lo schermo lcd mostra un messaggio di benvenuto ("*Hello!*"); quando un'auto elettrica del simulatore raggiunge l'Arduino GCP, sullo schermo viene mostrato il messaggio "*Attached*" e dopo un breve lasso di tempo il messaggio viene sostituito da "*Charging...*", presente sulla prima riga dello schermo, che indicherà l'avvio del ciclo di ricarica. Nel frattempo, sulla seconda riga dello schermo verrà via via creata una progress bar seguita dalla percentuale di ricarica. Quando la percentuale di ricarica avrà raggiunto il 100% e l'Arduino avrà ricevuto un messaggio che lo informa che la carica è terminata, sullo schermo sarà visualizzato il messaggio "*Charging Finished*" che dopo un secondo sarà sostituito dal messaggio iniziale "*Hello!*".

Libreria per la programmazione della LCD Keypad Shield:
<LiquidCrystal.h>.

3.1.4 Breadboard

Una breadboard (detta anche basetta sperimentale) è uno strumento che in ambito elettronico è utilizzato per la creazione di prototipi di circuiti elettrici.
Essa è caratterizzata da fori all'interno dei quali si possono inserire delle componenti da collegare a dei fili o a degli *stepper* per il passaggio di corrente elettrica da un lato all'altro dei circuiti creati.
La breadboard è una componente essenziale per l'assemblaggio dell'Arduino GCP poichè collega tra di loro le board dell'Arduino, della scheda ethernet e dello schermo LCD ed anche due led utili ad indicare lo stato del GCP. Dunque in tale sezione è opportuno descrivere come è stato assemblato l'Arduino GCP.

Arduino Board ⟷ Ethernet Shield

La Ethernet Shield è semplicamente collegata alla board principale, l'Arduino Uno, "a pila".
Quindi, come detto in precedenza, i pin della board dell'Arduino impegnati saranno: 10, 11, 12, 13.

Ethernet Shield ⟷ Schermo LCD

Dato che per questioni di spazio e dimensioni non è stato possibile inserire "a pila" la LCD Keypad Shield al di sopra della Ethernet Shield, in quanto la board dello schermo lcd veniva a contatto con la porta ethernet della scheda ethernet, allora si è scelto di inserire la LCD Keypad Shield sulla breadboard e riportare i suoi pin sull'Arduino tramite dei cavi. Un altro motivo che ha spinto verso questa soluzione è stato il fatto che la board dello schermo lcd, oltre ai pin 4, 5, 6, 7, 8, 9, utilizza anche il pin 10 per la retroilluminazione. Quindi questo pin è in contrasto con quello della scheda ethernet alla quale (a quest'ultima) si è scelto di dare priorità essendo esso un pin essenziale per la Ethernet Shield e trascurabile per la LCD Keypad Shield. I pin sottostanti della board dell'lcd sono stati inseriti direttamente nei fori della breadboard, mentre tramite l'ausilio di cavi elettrici (di tipo maschio-femmina), sono stati collegati tutti gli altri pin dello schermo LCD ai corrispondenti pin della Ethernet Shield (che poi riportano quelli di Arduino Uno).

Ethernet Shield ⟷ Luci LED

Al fine di emulare lo stato di un GCP durante la fase di ricarica di un'auto sono state utilizzate due luci led di due colori distinti, una verde ed un'altra rossa. Affinchè queste luci possano funzionare e quindi possano accendersi, il loro ramo positivo (detto anodo) deve essere collegato ad una resistenza che poi sarà collegata con un flusso di corrente positivo, mentre il ramo negativo (detto catodo) deve essere collegato alla messa a terra.
Per fare ciò i due led sono stati inseriti, uno di fianco all'altro, nei fori della breadboard; successivamente, si è collegato con un cavo di tipo maschio-maschio il pin GND con un punto della breadboard (che si trova indicativa-

mente sulla linea col simbolo - disegnata sulla breadboard).
A questo punto vengono collegati i catodi dei led tramite l'ausilio di stepper, mentre gli anodi vengono collegati a delle piccole resistenze e poi con l'utilizzo di cavi di tipo maschio-maschio i led vengono collegati agli unici due pin digitali della Ethernet Shield rimasti inutilizzati: 2 per il LED di colore rosso, 3 per quello verde. Una volta che questo circuito è pronto, in breve questo è quanto avviene durante l'emulazione: finchè nessuna auto simulata raggiunge l'Arduino GCP le luci rimangono spente. Non appena un'auto è riuscita a collegarsi viene acceso il LED verde che rimane fisso, mentre durante la fase di ricarica il LED rosso lampeggia. Quando la ricarica sarà terminata, per un breve lasso di tempo i due LED rimarranno entrambi fissi e dopo un secondo si spegneranno.

3.2 Implementazione

In questa sezione verranno discussi i dettagli implementativi riguardanti la logica che hanno permesso la realizzazione dell'emulazione di una colonnina di ricarica per auto elettriche.
Lo sviluppo del codice di Arduino Uno è stato realizzato tramite l'ambiente di sviluppo Arduino IDE nella sua versione 1.0.5, all'interno del quale vengono realizzati i progetti Arduino chiamati *sketch* il cui linguaggio sfrutta la potenza espressiva del C, del C++, ma si serve anche di alcune librerie Java. Gli sketch sono molto semplici da realizzare e sono caratterizzati da due funzioni principali di tipo `void`:

1. `setup()` - funzione invocata una volta sola all'avvio di Arduino Uno (oppure al suo reset) all'interno della quale vengono inequivocabilmente inseriti tutti i parametri per l'inizializzazione delle variabili e strutture principali che verranno utilizzate successivamente nello sketch.

2. `loop()` - funzione che può essere vista come un `while(1)`, ovvero una funzione che va a ciclare continuamente finchè la board è accesa, allo scopo di interrogare le componenti hardware di cui si compone la piattaforma elettrica ideata.

3.2.1 Fase di setup

Secondo quanto appena descritto, si può ben intuire che nella funzione setup() sono state inizializzate tutte le variabili che fanno riferimento alle componenti essenziali dell'Arduino GCP.
In particolare: viene inizializzata la comunicazione seriale con una trasmissione pari a 9600 bit di dati per secondo (Serial.begin(9600)), viene inizializzato lo schermo LCD con lcd.begin(16, 2) dove 16 è il numero di colonne sullo schermo e due il numero delle righe, vengono inizializzati i due led con i comandi pinMode(GREENLED, OUTPUT), pinMode(REDLED, OUTPUT) dove GREENLED e REDLED sono due costanti rispettivamente di valore 2 e 3 indicanti il numero dei pin digitali ai quali sono collegati. Inoltre viene inizializzata la scheda ethernet con Ethernet.begin(mac, ip) in cui ip è una variabile indicante l'indirizzo IP dell'Arduino Uno e mac indica il suo indirizzo MAC di valore 0x90, 0xA2, 0xDA, 0x03, 0x00, 0x5B, valore che generalmente viene riportato in una targhetta al di sotto della scheda. Poi viene stampata la stringa "*Hello!*" sullo schermo LCD, viene impostato il cursore dello schermo alla posizione corrispondente alla riga 1 e colonna 0 e tramite il comando lcd.createChar(0, p1) viene inizializzato un char particolare che in realtà è un vettore di 8 byte (byte p1[8] = {0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F}), utile a rappresentare sullo schermo un rettangolino che andrà poi a comporre una *progress bar* durante la fase di ricarica di un'auto.

3.2.2 Comunicazione con la City SIB

Come da titolo in questa sessione sarà discussa la parte di comunicazione tra l'Arduino GCP e la City SIB, volta a comunicare a quest'ultima le caratteristiche del GCP così che il simulatore di IoE dia la possibilità alle auto simulate di ricaricarsi anche presso l'Arduino GCP.
La City SIB, come visto nella sezione dedicata al Semantic Information Broker, utilizza un componente dell'architettura chiamato *sib-tcp* il quale resta in ascolto sulle porte 10020, 10021, 10022 in attesa di ricevere richieste dirette al database semantico. Dunque è compito dell'Arduino GCP instaurare una connessione TCP con la City SIB.

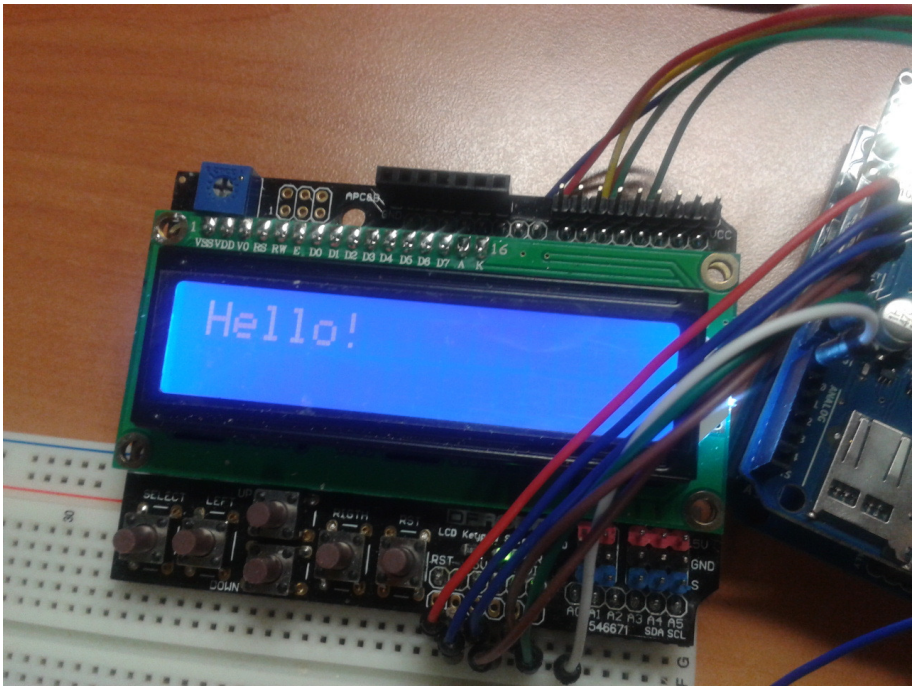

Figura 3.2: Arduino GCP

Infatti al suo avvio, ovvero nella fase di setup ed in particolare successivamente alle inizializzazioni spiegate in precedenza, il GCP invoca una funzione denominata sendSIB() i cui parametri sono un messaggio da inviare alla SIB e la porta della SIB (si è scelto di comunicare con la porta 10020), la quale si occupa di instaurare la connessione con la SIB tramite il metodo connect() della libreria <Ethernet.h> e quindi di inviare il messaggio tramite il metodo print(). Questi ultimi due metodi vengono invocati su di una variabile (client) appartenente alla classe EthernetClient specializzata, come si può ben intuire, nella comunicazione con il corrispettivo server (in questo caso la SIB). Il metodo connect() prende in input l'indirizzo ip del server e la porta sulla quale connettersi (10020); esso restituisce la variabile true se la connessione è andata a buon fine.
Le informazioni relative all'Arduino GCP che vengono inviate tramite print(), che prende in input una stringa rappresentante il messaggio da inviare, consistono in messaggi XML appartenenti al protocollo SSAP aventi la seguente struttura:

```
<SSAP_message>
  <transaction_type>QUERY</transaction_type>
  <message_type>REQUEST</message_type>
  <transaction_id>
    <!-- numero transazione -->
  </transaction_id>
  <node_id>ARDUINO_KP</node_id>
  <space_id>X</space_id>
  <parameter name = "type">sparql</parameter>
  <parameter name = "query">
    <!-- Query -->
  </parameter>
</SSAP_message>
```

Le query che vengono inviate all'avvio di Arduino GCP sono di tipo INSERT DATA e, cosa molto importante, le stringhe che le rappresentano sono di un tipo di dato particolare PROGMEM const char facente parte della libreria <avr/pgmspace.h>. Si è dovuto utilizzare tale tipo di dato in quanto i messaggi XML risultavano troppo grandi e, data la limitata capacità di memoria di Arduino Uno, le stringhe troncate; invece con questa libreria viene riservato dello spazio per le stringhe nella memoria Flash anzichè nella SRAM.
Di seguito saranno elencate le informazioni principali dell'Arduino GCP rappresentate dalle seguenti costanti:

- insertArduinoGCPName: query di insert che informa la SIB che vi è una nuova entità con id ba21d8dd-c8ff-4bb1-ac82-c86c57a9c36a di tipo ioe:GridConnectionPoint e nome *Arduino GCP*.

- insertGPSData: query di insert volta ad aggiungere nella SIB i dati dell'Arduino GCP relativi alle sue coordinate GPS, cioè latitudine e longitudine.

- insertEVSEInfo: query di insert che informa la SIB di quale sia l'EVSE legato ad Arduino GCP.

- insertChargeProfileInfo: query di insert che informa la SIB di quale sia l' ioe:ChargeProfile relativo all'EVSE dell'Arduino GCP.
- insertCurrentData1, insertCurrentData2: query di insert che associa gli ioe:CurrentData, quali l'intesità di corrente di entrata e di uscita, all' ioe:ChargeProfile del corrispondente EVSE.
- insertPowerData: query di insert che associa i dati relativi alla potenza dell'EVSE all' ioe:ChargeProfile.
- insertPriceData1, insertPriceData2: query di insert che associa i dati relativi al prezzo di ricarica all' ioe:ChargeProfile.
- insertVoltageData: query di insert che associa i dati relativi alla tensione di carica elettrica all' ioe:ChargeProfile.
- insertConnectorInfo: query di insert che associa i dati del tipo di connettore al corrispondente EVSE.

Per ognuna di queste stringhe che viene inviata, viene analizzata la risposta tramite una funzione denominata receiveSiBPkt() (che si mette in attesa di ricevere pacchetti di tipo TCP) si controlla il messaggio di risposta della SIB che ci si aspetta debba essere di questo tipo:

```
<SSAP_message>
  <message_type>CONFIRM</message_type>
  <transaction_type>QUERY</transaction_type>
  <transaction_id>1</transaction_id>
  <space_id>X</space_id>
  <node_id>ARDUINO_KP</node_id>
  <parameter name="status">m3:Success</parameter>
  <parameter name="results">
    <sparql xmlns="http://www.w3.org/2005/sparql-results#">
      <head></head>
      <boolean>true</boolean>
    </sparql>
```

```
  </parameter>
</SSAP_message>
```

Quindi se per tutti i messaggi le query saranno andate a buon fine (status m3:Success) e ciascuna risposta conterrà <boolean>true<boolean> nei propri results, allora si potrà proseguire col ciclo principale dell'Arduino GCP (loop()). Data la grande dimensione delle stringhe su elencate, concludiamo la sezione fornendo solamente un esempio di INSERT query rappresentante specificatamente il valore di <parameter name = "query"> della costante insertArduinoGCPName (i caratteri di escape < e > hanno lo scopo di rimuovere le ambiguità affinchè non siano confusi dal protocollo SSAP con < e > dei messaggi XML):

```
PREFIX ioe:&lt;http://www.m3.com/2012/05/m3/ioe-ontology.owl#&gt;
PREFIX rdf:&lt;http://www.w3.org/1999/02/22-rdf-syntax-ns#&gt;

INSERT DATA {
  ioe:ba21d8dd-c8ff-4bb1-ac82-c86c57a9c36a
  rdf:type ioe:GridConnectionPoint .
  ioe:ba21d8dd-c8ff-4bb1-ac82-c86c57a9c36a
  ioe:hasName \"Arduino GCP\" .
}
```

3.2.3 Comunicazione con il simulatore di IoE

Il cuore della computazione di Arduino GCP è rappresentato dalla funzione loop() che viene ripetuta ciclicamente e all'interno della quale la piattaforma elettronica rimane in ascolto sulla porta 8888, dopo aver instaurato una connessione di tipo UDP in setup() in particolare quando è terminato l'invio delle query alla City SIB, al fine di ricevere informazioni dal simulatore di IoE. L'istanza della connessione UDP è rappresentata dalla variabile udp della classe EthernetUdp che è instaurata in questo modo: udp.begin(localPort) (dove localPort è un intero pari appunto a 8888).
Allo scopo di verificare se vi sono dati in arrivo, viene invocato il metodo

parsePacket() che restituisce la lunghezza del pacchetto ricevuto. Se il valore ricevuto è maggiore di 0, tramite udp.read(buffer, packetSize) si proseguirà alla lettura dei dati in cui buffer è un vettore di char nel quale saranno memorizzati i dati ricevuti e packetSize è la dimensione del buffer pari a quanto restituito da parsePacket(). I possibili valori di buffer inviati dal simulatore sono rappresentati dalle stringhe *ATTACHED*, *REQUEST*, *FINISHED* oppure da un numero di tipo double denominato capacity:

- ATTACHED: viene inviata dal simulatore nel momento in cui un EV simulato ha raggiunto l'Arduino GCP ed ha occupato il suo EVSE. In questo caso Arduino GCP reagisce accendendo il led verde (digitalWrite(GREENLED, HIGH)), stampando sullo schermo LCD la scritta "*Attached*" ed inviando (sempre tramite il metodo print(), come avviene per le connessioni di tipo TCP), una stringa (*CAPACITY*), al fine di ottenere dal simulatore la capacità della batteria dell'auto che si vuole ricaricare presso Arduino GCP.

- REQUEST: viene inviata dal simulatore mentre un EV si sta ricaricando presso l'EVSE di Arduino GCP al fine di conoscere la percentuale di carica attuale della batteria dell'EV (rappresentata da una variabile double chiamata socPerc) in modo tale che, se l'EV ha raggiunto il valore massimo di carica (soglia impostata ad un valore pari a 1.0) possa lasciare l'EVSE e ripartire. Dunque in questo caso, compito dell'Arduino è di inviare al simulatore la socPerc attuale.

- FINISHED: viene inviata dal simulatore quando un'auto sta per lasciare l'EVSE di Arduino GCP. In questo caso la piattaforma elettronica si occupa di resettare il valore di socPerc, scrive sullo schermo LCD la scritta "*Charging Finished*" e, dopo un intervallo di tempo di 1s, spegne sia il led verde che rosso (digitalWrite(GREENLED, LOW), digitalWrite(REDLED, LOW)) e ripresenta sullo schermo il messaggio iniziale (che era "*Hello!*").

- capacity: valore che viene ricevuto dal simulatore dopo che Arduino GCP ne ha fatto richiesta nel caso *ATTACHED* e che sarà molto utile

durante la fase di ricarica per aggiornare la percentuale di ricarica corrente. Ovviamente in questo caso il valore viene memorizzato in una variabile globale chiamata appunto capacity e si può cominciare a calcore il valore effettivo di socPerc.

Tra i casi appena descritti ce n'è uno che manca e che deve essere analizzato: **la fase di ricarica di un EV**.
In loop(), dopo aver controllato se vi sono pacchetti in arrivo (reagendo secondo quanto descritto sopra), si verifica il valore di socPerc e se tale valore è compreso tra i valori 0.01 e 1.00, si calcolerà il nuovo valore di socPerc secondo la formula seguente:

$$socPerc = \frac{socKwh + \frac{power * timeSlice}{3600}}{capacity} \tag{3.1}$$

In cui socKwh è pari al vecchio valore della socPerc moltiplicato per la capacità della batteria dell'EV che si sta ricaricando, power è la potenza di ricarica di Arduino GCP e timeSlice è un intervallo di tempo pari a 1s.
Oltre al calcolo di socPerc, in questa fase l'Arduino GCP scrive sullo schermo la scritta "*Charging...*", accende e spegne il led rosso (con un intervallo di 100ms) per creare un effetto *blink* rappresentante il fatto che l'auto è ancora in carica, ma viene anche creata una barra di ricarica sullo schermo LCD tramite un metodo da me chiamato createProgressBar(). Tale metodo prende in input il valore attuale di socPerc ed in base a quest'ultimo stampa sulla seconda riga dello schermo una barra (che ricordiamo è l'array di 8 byte p1 introdotto nella sezione 3.2.1) per ogni 10% del valore della variabile e stampa anche la percentuale accanto alla barra di ricarica.

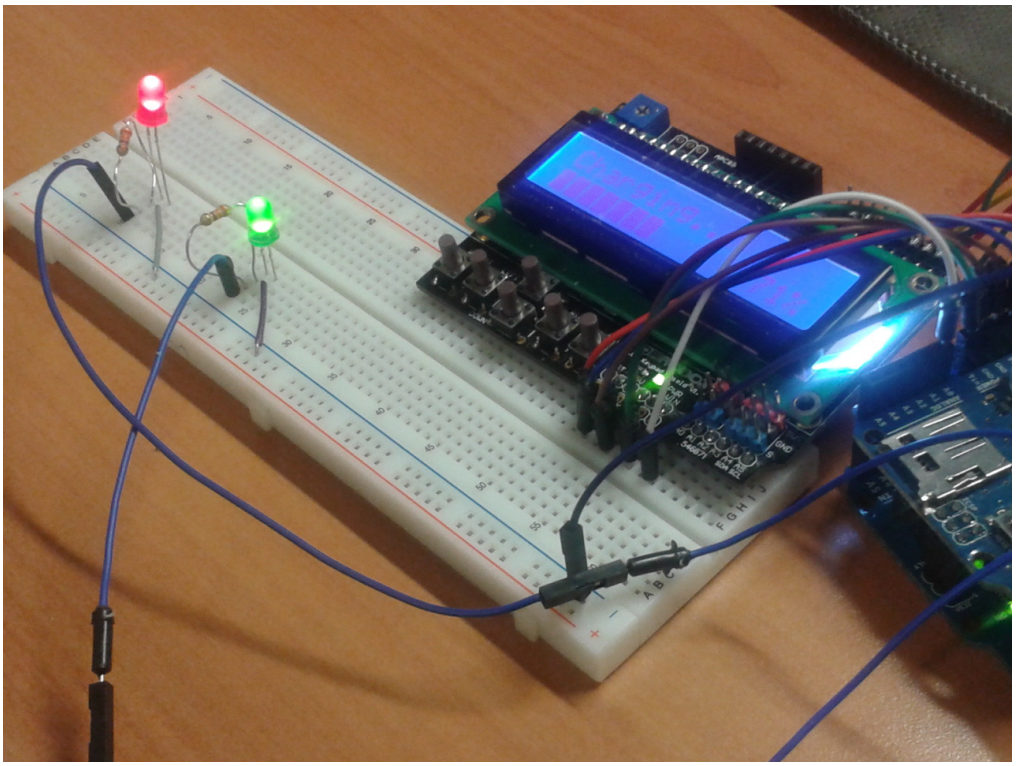

Figura 3.3: Arduino GCP - fase di ricarica

Capitolo 4

La piattaforma di Simulazione

In questo capitolo sarà analizzata la piattaforma di simulazione di IoE la quale rappresenta uno strumento molto potente in quanto all'interno di questo progetto di ambito europeo permette di studiare l'impatto dell'analisi dei GCP sull'EM in maniera tale che anche le città che hanno aderito al progetto possano migliorare i servizi per queste nuove tecnologie veicolari che vanno sempre più in crescendo. Qui si analizzeranno, in un primo momento, quelle che sono le varie parti che costituiscono il simulatore di IoE e successivamente saranno descritti principalmente le classi e i moduli che sono stati utili allo sviluppo del mio progetto e come ho dovuto modificarli per rendere possibile la comunicazione con Arduino GCP.

4.1 Architettura del simulatore

La piattaforma di simulazione di IoE si compone di più applicativi, come viene mostrato nella *Figura 4.1*, che sono **SUMO** e **OMNeT++** connessi tra di loro tramite il framework **Veins**.
SUMO si occupa di modellare, cioè simulare, la mobilità all'interno del contesto di un traffico urbano, mentre OMNeT++ implementa tutte le operazioni legate agli EVSE ed agli EV compresi i meccanismi di richiesta e risposta tra queste entità.
Inoltre come si nota dalla figura, ad OMNeT++ sono connessi un modello di

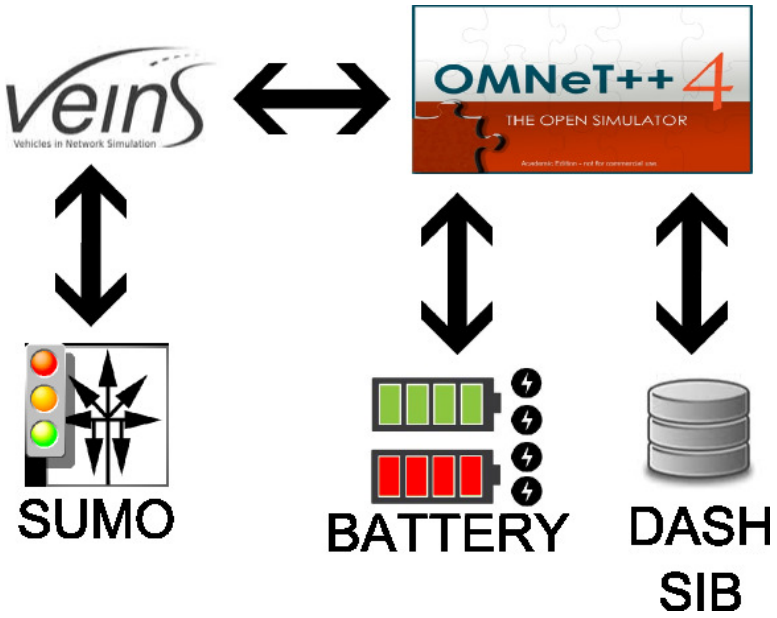

Figura 4.1: Architettura Simulatore IoE

batteria realistico (fornito tramite una libreria da SIEMENS) che gestisce gli eventi di scarica di un EV e di ricarica di un EVSE, e la Dash SIB all'interno della quale ciascun EV pubblica le informazioni relative alla posizione e stato di carica attuali in modo tale che un utente possa interfacciarsi col proprio smartphone. La presenza della Dash SIB all'interno dell'architettura ha l'unico scopo di permettere la simulazione del sistema di prenotazione dei vari EVSE tramite smartphone ed in futuro, quando magari i veicoli reali disporranno a bordo di meccanismi per la comunicazione dello stato attuale al servizio cittadino, il suo uso potrebbe non essere più necessario.
Procediamo con una descrizione un po' più dettagliata delle componenti della piattaforma appena citati.

4.1.1 SUMO

SUMO (Simulator of Urban MObility) è un simulatore di mobilità veicolare sviluppato in C++ e supportato principalmente dall'Institute of Transportation Systems at the German Aerospace Center.
Visto il suo scopo, SUMO ha ben presente i concetti di veicoli, semafori

e ostacoli (ad esempio palazzi). Tutte queste entità vengono create grazie all'ausilio di file XML, soprattutto per quanto riguarda le auto, infatti ciascun veicolo ha un proprio itinerario che gli viene associato al momento della sua creazione e che viene scelto casualmente da un file XML in cui sono descritti tutti i possibili nodi di una rete stradale.
SUMO permette di avviare la simulazione sia in modalità grafica che visuale. Nella modalità grafica viene mostrata un'interfaccia grafica (*sumo gui*) che consente tramite dei tasti di interagire col simulatore e che visualizza la mappa della città, che si è scelto di studiare, all'interno della quale è possibile vedere i veicoli che si spostano, si fermano ai semafori e si ricaricano presso i GCP.

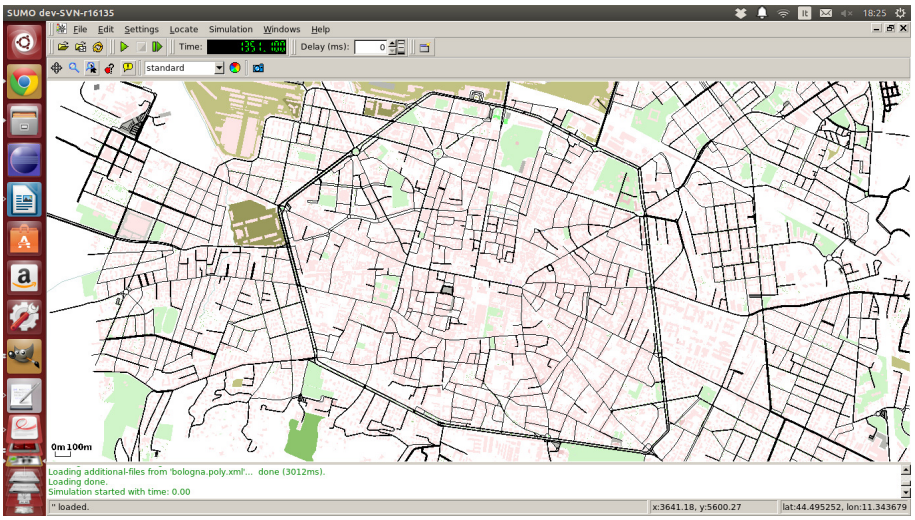

Figura 4.2: Esempio Sumo GUI

Nella modalità visuale vengono mostrati su di un terminale i risultati della simulazione, ma anche i vari messaggi di debug ed errore. Dunque tale modalità è molto più utile per effettuare il debug delle applicazioni.
Al fine di migliorare il servizio cittadino, l'Università di Bologna per effettuare le proprie analisi tramite SUMO ha considerato uno scenario su larga scala caricando nell'emulatore la mappa di Bologna servendosi del servizio libero di mappe *Open Street Map*. Inoltre l'analisi si è basata su dati reali,

riguardanti la densità di traffico, raccolti con ausilio di strumenti posizionati lungo varie zone della città, chiamati *induction loop*.
Infine è opportuno specificare che SUMO si serve di un modulo per comunicare con *Veins* chiamato **TraCI** (Traffic Controller Interface), il quale rende possibile interagire con la simulazione tramite un protocollo di tipo TCP/IP.

4.1.2 Veins

Veins è un framework volto alla simulazione di reti veicolari al di sotto del livello del quale vi è un altro framework che si chiama MiXiM (che a sua volta è al di sopra di OMNeT++) il quale implementa modelli per reti wireless.

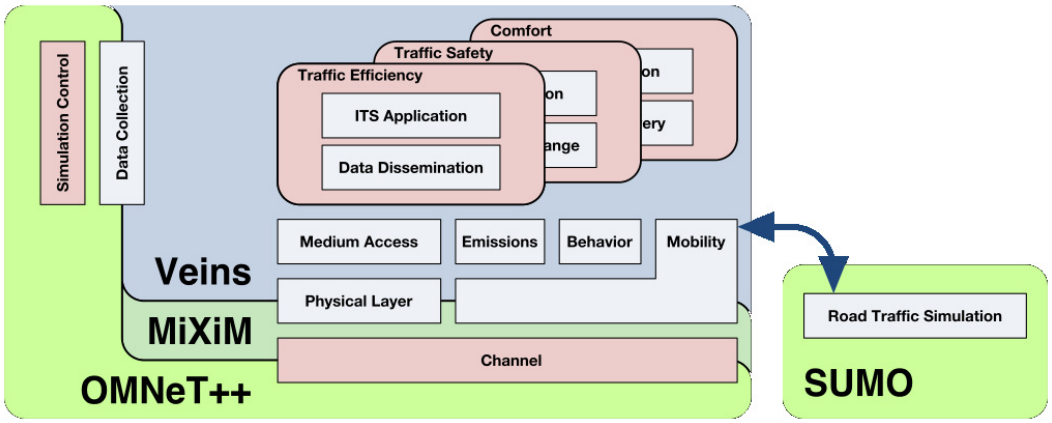

Figura 4.3: Stack di Veins

Veins si occupa di gestire la comunicazione tra OMNeT++ e SUMO tramite il modulo TraCI, in particolare ogni volta che in SUMO viene creato un veicolo, tale framework crea il corrispondente modulo in OMNeT++.
Veins inoltre è il responsabile della sincronizzazione tra OMNeT++ e SUMO che avviene in questo modo: durante l'esecuzione avanzano dei passi temporali, per ogni passo temporale il framework invia a SUMO un insieme di comandi che sono contenuti in un buffer, i quali causano l'avanzamento del

corrispondente passo temporale nella simulazione del traffico stradale.
Al termine di tale passo, SUMO invia a Veins una serie di comandi contenenti lo stato e la posizione dei veicoli istanziati che vengono elaborati dal framework e, successivamente, Veins aggiunge i corrispettivi nodi per ogni nuovo veicolo introdotto nella simulazione e rimuove i nodi relativi ai veicoli giunti a destinazione. Tuttavia i nodi possono essere anche rimossi se i veicoli non sono giunti ancora a destinazione, ma è terminata la loro batteria.

4.1.3 OMNeT++

OMNeT++ è un ambiente open source (basato su Eclipse) definito anche come un simulatore ad eventi discreti generalmente utilizzato per la simulazione di reti di comunicazione. La sua caratteristica principale risiede nella modularità il che consente di estendere le funzionalità di OMNeT++ per la simulazione in altri ambiti. Infatti per IoE è stato introdotto il supporto per le reti veicolari estendendo l'ambiente con i modelli degli EVSE (inclusa la gestione della coda dei veicoli in attesa sull'EVSE) e degli EV integrandovi anche il modello di ricarica e scarica della batteria fornito da SIEMENS. Infine è stata modellata la comunicazione di rete tra gli EV e la SIB per le richieste di ricarica.
Le simulazioni in OMNeT++ vengono modellate tramite l'utilizzo di componenti riusabili e tra loro combinabili, detti moduli.
La comunicazione tra questi moduli avviene per via del meccanismo di *message passing* i cui messaggi possono contenere strutture dati arbitrarie.
I moduli, i loro parametri e collegamenti sono definiti in un linguaggio di alto livello di nome NED in file con estensione, appunto, .ned, la loro implementazione invece consiste in una classe scritta in C++ che è il linguaggio principale dell'IDE.
Essendo OMNeT++ un simulatore ad eventi discreti, i moduli non agiscono se non viene schedulato un evento, il che avviene tramite una funzione: scheduleAt() che vedremo in seguito. Per la schedulazione di un evento ad ogni modulo viene inviato un messaggio di un tipo di dato particolare di OMNeT++ che prende il nome di cMessage (che nei vari moduli viene an-

che esteso da messaggi specifici), il quale viene catturato dai metodi handleMessage() e handleSelfMessage() (nel caso di un messaggio auto-inviato dal modulo per "mantenersi in vita"). I moduli inoltre dispongono di altri due metodi: initialize() e finish() che rispettivamente inizializzano e terminano il modulo.

4.2 Moduli di OMNeT

In questa sezione verranno analizzati i moduli e i metodi più importanti di OMNeT++ e che sono stati utili per la comunicazione con Arduino GCP. Secondo la descrizione di OMNeT++ fatta in precedenza, si dovrebbe intuire che i moduli vengono eseguiti in concorrenza e sono sincronizzati (o meglio schedulati) tramite la funzione scheduleAt() precedentemente nominata. Tale funzione prende in input un intervallo di tempo espresso in secondi che generalmente viene ricavato dalla funzione simTime(), la quale restituisce il tempo corrente di simulazione, a cui eventualmente si aggiungono i secondi dopo i quali il modulo debba essere richiamato (e.g. simTime() + 10 indica che il modulo sarà richiamato dopo 10s). Inoltre a scheduleAt() viene passato un secondo parametro

4.2.1 City Service

Il modulo CityService è il modulo principale della simulazione e si occupa di simulare la smart-grid.
Nella fase di inizializzazione (initialize()) esso si occupa di istanziare nel simulatore tutte le colonnine di ricarica (eccetto Arduino GCP che viene creata dinamicamente quando la piattaforma di simulazione comunica con la piattaforma elettronica) e per fare ciò si serve del metodo init() appartenente alla classe GcpController, principale responsabile di tutte quelle funzionalità che riguardano la gestione dei GCP (e.g. trovare il GCP più vicino alla posizione attuale di un EV, occupare l'EVSE del corrispondente GCP, ecc.). Tale metodo si occupa di caricare un file XML (nel nostro caso *gcp.bologna.2power.xml*) il quale contiene tutte le informazioni relative ai

GCP di Bologna. Per ogni colonnina di ricarica presente in questo file, viene istanziato un oggetto appartenente alla classe GCP di cui tutti gli attributi della classe corrispondono alle informazioni estratte dal file XML.
Le informazioni relative alla classe GCP sono rappresentate dalle seguenti variabili:

- name: stringa indicante il nome del GCP.
- sumoRoadId: stringa indicante l'id della strada sulla quale si trova il GCP e che è conosciuto da SUMO.
- latitude: variabile double indicante la latitudine del GCP.
- longitudine: variabile double indicante la longitudine del GCP.
- evseList: variabile di tipo vector<EVSE*> rappresentante la lista degli EVSE del GCP.

Dopo aver inizializzato in maniera opportuna tutti gli attributi di un GCP, esso viene inserito in una lista denominata gcpList e di tipo vector<GCP*>, la quale verrà analizzata ogni volta che un EV cerca un GCP vicino alla propria posizione attuale.
Il modulo CityService, inoltre, si occupa di memorizzare alcuni valori utili in fase di analisi che tengono conto del numero dei veicoli in carica (chargingVehicles), del numero dei veicoli eliminati a causa dell'esaurimento della batteria (vaporizedVehicles), del numero dei veicoli che hanno terminato normalmente la simulazione, raggiungendo la propria destinazione (leavingVehicles), e del numero di veicoli elettrici (electricalVehicles) in quanto dato che SUMO propone di simulare una situazione di traffico reale, in realtà vengono simulati anche veicoli a combustibile fossile di cui però non ci interessa molto.

4.2.2 Car Logic

Come suggerisce il suo stesso nome, il modulo CarLogic si occupa di implementare la logica legata al movimento delle auto ed alle sue fasi di ricarica.

Nel metodo initialize(), CarLogic interroga il City Service per conoscere qual è il tipo di auto attuale: se essa è di tipo combustibile fossile allora viene colorata di rosso nella *sumo gui* e viene fatta proseguire lungo la mappa senza che siano impostate ulteriori opzioni; se l'auto è di tipo elettrico, allora viene colorata di verde nella *sumo gui* e vengono inizializzate le seguenti variabili:

- 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: coefficiente resistenza d'attrito delle gomme sull'asfalto.
- cDrag: coefficiente della levigatezza del veicolo.
- across: sezione frontale del veicolo espressa in m^2.
- rhoAir: coefficiente di resistenza dell'aria.
- weight: peso dell'auto.
- threshold: soglia della batteria al di sotto della quale l'auto dovrebbe ricaricarsi.
- minRequestedEnergyKwh: quantità di energia minima in una prenotazione di ricarica.
- writeCarStatusOnSib: variabile che se impostata a *true* indica i scrivere le informazioni di stato del veicoli sul Dash SIB consentendo il monitoraggio dello stato dei veicoli dall'applicazione mobile.

Dopo tutte queste inizializzazioni CarLogic chiede a CityService se è attivo il servizio di prenotazione, se lo è allora viene instaziato un oggetto appartentente alla classe SibController, classe responsabile delle interrogazioni RDF o SPARQL della SIB che vengono effettuate tramite il richiamo delle funzioni della libreria *kpi_low.h* che implementano lo scambio di messaggi con la SIB

tramite il protocollo SSAP. Dopo di ciò viene impostato lo stato del veicolo attuale come DRIVING che è una costante facente parte di un'enumerazione denominata CarState ed il modulo viene schedulato tramite un automessaggio chiamato CarMessage (che estende cMessage) in cui sono contenute le informazioni relative allo stato attuale di un'auto. Il CarMessage, dunque viene catturato dalla funzione handleSelfMessage() la quale esegue uno switch-case su msg->getCarState() (dove msg è l'instanza del CarMessage) i cui possibili valori possono essere:

- DRIVING: costante che indica che l'auto è in movimento - in questo caso viene richiamata la funzione handleDriving() in cui, tramite un'istanza del modulo Battery, si verifica se lo stato della batteria (socPerc - variabile che dunque ha lo stesso nome di quella presente nello sketch di Arduino) è minore di threshold e se cio avviene:

  A. se è attiva la reservation, viene fatta richiesta di prenotazione ricarica con il metodo insertChargeRequest() del SibController il quale inserisce le triple RDF corrispondenti alla richiesta, nella SIB. Dopo questa azione, il veicolo si mette in attesa di ricevere una risposta dal sistema (costante WAITING_RESPONSE).

  B. se la reservation non è attiva, viene invocato il metodo findRandomNearestGCP() di GcpController che prende in input le coordinate correnti dell'auto ed un numero e si occupa di cercare il GCP più vicino entro il *range* specificato dal secondo parametro del metodo. Dopo aver trovato il GCP, lo stato del veicolo viene posto come GO_TO_RECHARGE.

  Se invece lo stato della batteria risulterà ancora ottimale, verrà invocata la funzione goToRandomRoad() che si occuperà di muovere l'auto verso un'altra strada.

- WAITING_RESPONSE: costante che indica che l'EV è in attesa di ricevere una risposta dal sistema - in questo caso viene richiamata la funzione handleWaitingResponse() all'interno della quale viene invocato il metodo checkResponseAndConfirmOption() di SibController. Tale metodo

esegue una query SPARQL al fine di ottenere delle opzioni di ricarica dal servizio cittadino; se la query è andata a buon fine, viene scelta una delle opzioni fornite dal City Service e lo stato dell'EV viene posto come WAITING_CONFIRM, altrimenti se non sono state ricevute opzioni valide, lo stato dell'auto viene reimpostato a DRIVING.

- WAITING_CONFIRM: costante che indica che l'EV è in attesa di ricevere conferma di prenotazione della colonnina, dopo aver scelto le opportune opzioni - in questo caso viene invocata la funzione handleWaitingConfirm() al cui interno viene richiamato il metodo checkConfirmBySystem() del SibController che si occupa di associare le opzioni alla reservation e di verificare la risposta del sistema. Se la risposta del sistema è negativa, come nel caso precedente lo stato dell'auto viene impostato a DRIVING, altrimenti nel CarMessage vengono inserite tutte le informazioni riguardanti il GCP e l'EVSE prenotati ed il tempo di inizio e fine richiesti dalla ricarica. Successivamente viene calcolato un *delay* con la sottrazione tra il tempo di inizio della reservation e getRelativeTime() che è una funzione che restituisce il tempo di esecuzione (non quello simulato); tale *delay* indica il tempo mancante alla ricarica e se è minore di una certa costante (impostata a 1000), lo stato dell'auto viene posto come GO_TO_RECHARGE, altrimenti lo stato dell'auto viene impostato come PARKING.

- GO_TO_RECHARGE: costante che indica che l'EV sta per andare a ricaricarsi presso un EVSE - in questo caso viene invocata la funzione handleGoToRecharge() in cui tramite un'istanza del modulo TraCI si recupera il *road Id* sul quale si sta muovendo l'auto e lo si confronta col sumoRoadId del GCP prenotato contenuto nel CarMessage. Se i due id risultano essere uguali, si va a verificare la velocità del veicolo e se essa è minore ad un valore pari a 1.0, l'auto viene fatta rallentare (commandSlowDown() del modulo TraCI), altrimenti:

  A. se la reservation è attiva, si verifica se l'EVSE è occupato. Se lo è lo stato dell'auto viene impostato a WAITING_EVSE, altrimenti

viene impostato come CHARGING.

B. se la reservation non è attiva anche in questo caso si verifica se l'EVSE è occupato. Però, dato che non vi è prenotazione, se il bocchettone risulta occupato viene invocato il metodo findRandomNearestGCPExcluding() di GcpController. Tale metodo prende in input le coordinate attuali del veicolo, ed il GCP appena visitato ed un numero e si occupa di cercare il GCP più vicino entro il *range* indicato dal terzo parametro, escludendo il GCP visitato. Dopo aver trovato il GCP, lo stato del veicolo viene posto come GO_TO_RECHARGE.
Se invece l'EVSE risulta libero, allora lo stato dell'auto viene posto come CHARGING.

- CHARGING: costante che indica che l'auto si sta ricaricando presso un EVSE - in questo caso viene richiamata la funzione handleCharging() nella quale si invoca il metodo isCharging() di Battery per verificare se la batteria del veicolo ha cominciato il suo ciclo di ricarica. Se non l'ha fatto, allora col metodo setChargingOnEVSE() di Battery lo stato della batteria verrà impostato su di una costante denominata CHARGE (che analizzeremo nella prossima sezione) e l'auto corrente verrà associata all'EVSE presso la quale si sta ricaricando.
  Inoltre in handleCharging() si verifica periodicamente se la batteria è piena, quindi se è stata ricaricata del tutto allora viene invocato il metodo setDischarging() di Battery, che imposta lo stato della batteria a DISCHARGE (vedi prossima sezione), il veicolo corrente viene rimosso dall'EVSE, viene richiamato un altro veicolo che eventualmente era in attesa dell'EVSE e l'auto corrente viene reimpostata sullo stato DRIVING.

- PARKING: costante che indica che l'auto deve essere parcheggiata - in questo caso viene chiamata la funzione handleParking() la quale si occupa semplicemente di rallentare l'auto se la sua velocità e maggiore di 1.0, altrimenti la ferma col comando di TraCI commandStopNode() e

imposta a true l'informazione relativa all'auto parcheggiata nel CarMessage.

- WAITING_EVSE: costante che indica che l'auto viene messa in attesa finchè l'EVSE da essa prenotato non è stato liberato - in questo caso viene chiamata la funzione handleWaitingEVSE() la quale aggiunge il veicolo nella coda dei veicoli in attesa facente parte della classe EVSE e imposta a true lâinformazione relativa allâauto parcheggiata nel CarMessage in quanto prima di invocare handleWaitingEVSE(), l'auto è stata parcheggiata.

4.2.3 Battery

Il modulo Battery è un modulo molto importante ai fini del mio progetto in quanto si occupa di implementare la logica relativa al ciclo di carica/scarica della batteria di un EV relativamente agli stati CHARGE e DISCHARGE dell'enumerate BatteryState che saranno analizzati fra un po'. Nel suo metodo initialize(), Battery (come avveniva per CarLogic) verifica se l'auto attuale è di tipo elettrico e se la condizione è verificata allora inizializza i seguenti parametri:

- manufacturer: azienda produttrice della batteria (SIEMENS in quanto come visto dall'architettura del simulatore, essa rappresenta il partner che ha fornito il modello della batteria).
- engineEfficense: efficenza gestione energetica del motore che varia tra 0 e 1.
- socPerc: stato attuale della batteria.
- capacity: capacità della batteria espressa in Kwh.
- voltage: tensione di carica elettrica misurata in V.
- maxVoltage: tensione massima della batteria.
- stateOfHealth: stato di vita della batteria che varia tra 0 e 1.

- maxCurrentIn: intensità massima di corrente di entrata espressa in A.
- maxCurrentOut: intensità massima di corrente di uscita espressa in A.
- nominalTemperature: temperatura massima di lavoro del motore.
- recordVector: struttura dati particolare nella quale vengono memorizzati i valori della batteria attuali.
- maxSoc: stato di ricarica massimo oltre il quale la batteria viene considerata carica. Il suo valore è 1.0.
- oldSpeed, oldTime, oldPos: rappresentano i vecchi valori di simulazione di velocità, del tempo e della posizione del veicolo in quanto, non dimenticando che Battery è un modulo che viene continuamente schedulato, tali valori saranno utili in particolare per il calcolo del livello di batteria attuale nel ciclo di scarica di un EV.
- cRoll, cDrag, rhoAir, across, carWeight: valori ricavati dal modulo CarLogic e che riguardano l'auto corrente, anch'essi utili per il calcolo del nuovo stato di carica della batteria.
- power: potenza della batteria.

In seguito alle suddette inizializzazioni, il BatteryState viene impostato a DISCHARGE e Battery si autoinvia un messaggio chiamato msgRefresh di tipo cMessage che viene catturato da handleSelfMessage().
Dunque gli stati di questo modulo sono solo DISCHARGE e CHARGE ai quali corrisponde l'invocazione di handleDischarge() e handleCharge():

1. handleDischarge: funzione che si occupa di tutti quei calcoli relativi al consumo energetico e del livello di batteria corrente di un EV, facendo uso delle variabili precedentemente inizializzate e tenendo anche in considerazione l'angolo di inclinazione della strata ricavato grazie ad un metodo di utilità denominato getInclinationRad(). In particolare vengono calcolati:

- forceIn: la forza di inerzia data dalla moltiplicazione tra il peso dell'auto e la sua accelarazione.

$$forceIn = carWeight * accel \tag{4.1}$$

dove accel è l'accelerazione attuale dell'auto ricavata da:

$$accel = \frac{currSpeed - oldSpeed}{\Delta t} \tag{4.2}$$

in cui currSpeed é la velocità attuale dell'auto e viene ricavata dal metodo getSpeed() di TraCI, mentre Δt è un timeSlice dato dalla sottrazione tra il tempo attuale di simulazione (simTime()) e oldTime.

- forceSlope: forza dovuta alla pendenza della strada così determinata:

$$forceSlope = carWeight * g * sin(\alpha) \tag{4.3}$$

in cui g è l'accelerazione di gravità sulla terra (pari a 9.81 m/s^2) e α è l'angolo di inclinazione della strada.

- forceRoll: forza che determinata la resistenza all'attrito delle gomme sull'asfalto, così calcolata:

$$forceRoll = carWeight * g * cos(\alpha) * cRoll \tag{4.4}$$

- forceAir: forza dovuta alla resistenza aereodinamica dell'auto, così calcolata:

$$forceAir = 0.5 * rhoAir * cDrag * across * avgSpeed^2 \tag{4.5}$$

dove avgSpeed è la media tra la velocità corrente e quella precedente:

$$avgSpeed = 0.5 * (oldSpeed + currSpeed) \tag{4.6}$$

Date queste forze è possibile calcolare il valore del lavoro totale eserci-

tato dall'auto:

$$totalWork = (F_{IN} + F_S + F_R + F_{Air}) * \frac{1}{efficense} * \Delta t * avgSpeed \quad (4.7)$$

dove:

- F_{IN} = forceIn
- F_S = forceSlope
- F_R = forceRoll
- F_{Air} = forceAir
- *efficense* = engineEfficense

Una volta calcolato il lavoro totale, sarà possibile calcolare l'energia fino a qui consumata (energyConsumption) tramite la divisione tra totalWork e 3600000, il che permetterà finalmente il calcolo dello stato di carica attuale:

$$socKwh = \frac{energyConsumption}{\Delta t} * 3600000 \quad (4.8)$$

la quale sarà poi convertita in percentuale (socPerc).

2. handleCharge: funzione che si occupa di gestire il processo di ricarica di un EV: viene controllato il valore di socPerc, se tale valore è minore di maxSoc (che ricordiamo essere pari a 1.0) allora viene preso il valore della potenza dell'EVSE sul quale si sta caricando il veicolo corrente e viene calcolato il nuovo valore di socPerc tramite la Formula 3.1 vista nel capitolo dell'Emulazione GCP, che quindi sarà bypassata quando un veicolo si ricaricherà presso Arduino GCP in quanto il calcolo è gestito da quest'ultimo (ma questo lo vedremo dopo).

4.2.4 Modifiche apportate

Quelli descritti fino ad ora sono i moduli di OMNeT++ nello stato originario in cui li ho ereditati prima di realizzare la mia emulazione.
In questa sezione sarà dunque mio compito descrivere quali moduli e come li

ho modificati per far sì che il City Service "si accorga" anche della presenza di Arduino GCP e renda possibile la comunicazione tra quest'ultimo ed un EV.

Car Logic

Innanzitutto è bene dire quali sono i metodi che ho introdotto in Car Logic e quali sono le loro funzionalità:

- int createUDPSocket(): metodo che si occupa di creare l'istanza di un socket per la comunicazione di tipo UDP secondo le modalità standard del linguaggio C. Ovvero inizialmente viene creato un socket con la funzione socket() alla quale si specifica che si vuole creare un *socket datagram* (SOCK_DGRAM) di tipo UDP (IPPROTO_UDP) appartenente alla famiglia AF_INET (famiglia di socket generalmente utilizzata in ambiente Linux), poi con la funzione setsockopt() si cerca di evitare l'errore tipico dei socket *Address Already Used* specificando il FLAG SO_REUSEADDR. Successivamente viene inizializzata una struttura dati di tipo struct sockaddr_in denominata host_address della quale si impostano la famiglia (host_address.sin_family), l'indirizzo IP (host_address.sin_addr.s_addr) e la porta (host_address.sin_port), per poi collegare tale struttura dati al socket per mezzo del metodo bind() al fine di far rimanere in ascolto il *socket* sulla porta dell'*host* impostato. Se tutti questi passaggi avranno avuto successo, l'intero che viene restituito dal metodo è l'identificativo del socket chiamato *socket descriptor* che viene restituito da socket() e memorizzato in una variabile intera denominata sd.

- void sendUDP(char* buffer): metodo che si occupa di inviare un pacchetto UDP contenente il messaggio passato in input al metodo, sempre secondo le modalità standard del C. In particolare in questa funzione viene inizializzata la struttura struct sock_addr_in denominata target_host_address di cui vengono impostati la famiglia, la porta e l'indirizzo IP (che corrispondono all'indirizzo IP ed alla porta 8888

dell'Arduino). Successivamente, tramite la funzione sendto(), viene inviata la stringa buffer presa in input, al target_host_address e quindi viene chiuso il socket (close(sd)).

- char* receiveUDP(): metodo che si occupa di ricevere un pacchetto UDP secondo le modalità standard del C. In particolare viene invocata la funzione recvfrom() che riempie un array di caratteri denominato buffer ed inizializza anche una struttura struct sock_addr_in chiamata receive_host_address che può essere analizzata per ottenere informazioni sul mittente. Prima del termine del metodo, viene ritornato il buffer contenente gli eventuali dati ricevuti.

- returnNumberOfGcp(): metodo che si occupa semplicemente di ritornare la grandezza della lista dei GCP contenuta nel GcpController e che viene recuperata col metodo getAllGcp() dell'istanza del GcpController, utile a CarLogic al fine di confrontarlo con il numero dei GCP effettivamente presenti nella SIB per sapere se ve ne sono di nuovi.

Dopo aver descritto le nuove funzionalità dei nuovi metodi vediamo quelle che sono le modifiche effettive al modulo CarLogic. Dato che il metodo initialize() viene richiamato per ogni auto, dopo le varie inizializzazioni viene fatta un'istanza del SibController, dunque rispetto al CarLogic originale tale istanza viene creata anche quando non vi è reservation in quanto nel SibController ho implementato dei metodi per effettuare delle query SPARQL al fine di comunicare con la SIB. Di tale istanza viene richiamato il metodo countGcp() che restituisce il numero dei GCP presenti nella SIB (assegnato ad una variabile di nome newNumberOfGcp) e che sarà confrontato col numero restituito da returnNumberOfGcp() (assegnato alla variabile numberOfGcp). Per ogni auto inoltre viene fatta istanza della lista dei GCP del GcpController tramite il metodo getAllGcp(). Viene dunque confrontato newNumberOfGcp con numberOfGcp e se Arduino GCP avrà inserito le sue informazioni nella SIB, allora tale numero risulterà maggiore. Ciò comporterà l'istanziazione di un nuovo GCP che viene effettuata da un nuovo metodo del SibController chiamato getNewGcp() e quindi l'aggiunta nella lista dei GCP dell'auto corrente ed

infine numberOfGcp viene posto uguale a newNumberOfGcp. Un'altra modifica è stata apportata al metodo handleCharging() nel quale, dopo aver impostato lo stato della batteria su CHARGE, si compara il *road Id* preso da TraCI col sumoRoadId di Arduino GCP (che ho impostato pari a 35449271). Se la condizione è verificata allora vuol dire che l'auto si vuole ricaricare presso Arduino GCP e dunque viene inviata la stringa "*Attached*" alla piattaforma elettronica, tramite sendUDP() e l'auto si mette in attesa di ricevere una richiesta con receiveUDP(). La stringa che ci si aspetta da parte di Arduino GCP è "*Capacity*" e quando viene ricevuta, CarLogic risponde inviando la capacità della batteria dell'auto corrente.
Sempre in handleCharging(), quando la batteria è piena e l'auto sta per ripartire, viene nuovamente verificato se il *road Id* di TraCI è uguale a quello di Arduino GCP e se ciò si verifica viene inviata la stringa "*Finished*" all'Arduino che si comporterà di conseguenza.

GcpController

A questa classe è stata apportata una piccola modifica, ma essenziale: ai metodi findRandomNearestGCP() e findRandomNearestGCPExcluding() è stato aggiunto il parametro vector<GCP*>gcpList che è molto importante in quanto ciascun auto possiede la propria gcpList che è differente da quella del GcpController e dato che l'auto aggiorna la propria lista, in seguito alla query getNewGcp(), se non vi fosse questo ulteriore parametro, essa non avrebbe mai modo di dirigersi verso Arduino GCP anche se l'auto si troverà nei pressi del componente emulato.

SibController

La maggior parte dei metodi del SibController originario sono specializzati nelle query sia RDF che SPARQL riguardanti le richieste di prenotazione. Io ho esteso tale classe aggiungendovi i seguenti tre metodi che non richiedono che la reservation sia attiva:

- int countGcp(): metodo che si occupa di contare il numero di GCP presenti nella City SIB. La query effettuata è rappresentata dalla seguente

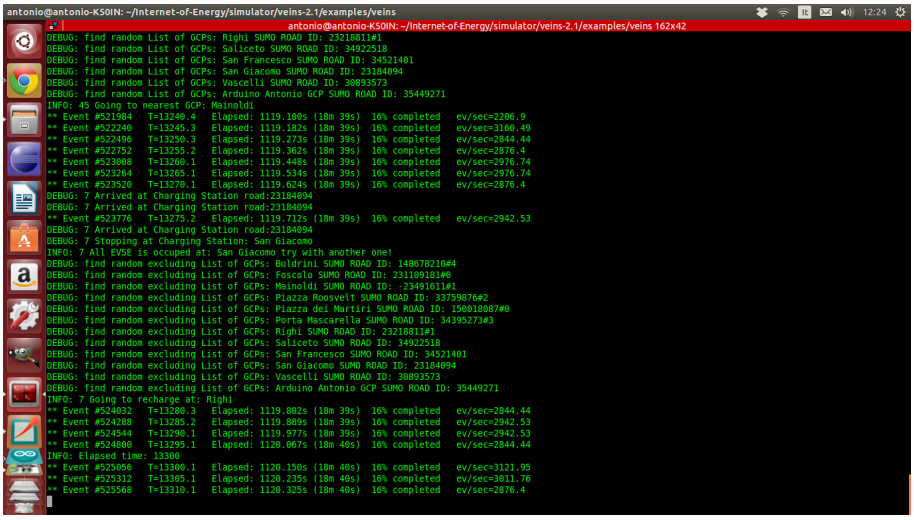

Figura 4.4: Ricerca GCP più vicino - con Arduino GCP

stringa:

```
ostringstream query;
query << "SELECT ?gcp WHERE {"
"?gcp <" + RDF_TYPE + "> <" + IOE_URI
 + "GridConnectionPoint>.}";
```

la quale viene passata alla funzione sib_sparql_query(), presente nella classe SibUtil, che prende in input anche una struttura dati chiamata ss_info_t indicante il tipo di SIB da interrogare (citySibInfo in questo caso), ed un'altra struttura dati di tipo ss_nodes_t (la cui variabile si chiama sparql_result) ed invoca, a sua volta, la funzione ss_sparql_query() che richiama la corrispondente funzione in *kpi_low.h* che implementa SSAP. Se la query è andata a buon fine, sib_sparql_query() riempie sparql_result che può anche essere visto come un puntatore alla testa di una lista di ss_nodes_t e dunque scandendo tale lista tramite il campo next, si potrà incrementare un contatore finchè sparql_result->next non sarà uguale a NULL. Infine tale contatore sarà restituito dalla funzione.

- GCP* getNewGcp(int index, CityService* cityService): metodo che si occupa di istanziare il GCP corrispondente ad Arduino GCP. Prende in

input un indice che indica fin dove deve essere scandita la lista dei GCP (fino a newNumberOfGcp) e l'istanza del CityService che viene associato al GCP. Le query effettuate in questo metodo sono tre: una uguale a quella di countGcp() utile ad ottenere la lista dei GCP presenti nella SIB, la quale viene scandita finchè non si raggiunge l'indice desiderato e quindi può essere eseguita la seconda query. Questa seconda query è volta ad ottenere il nome, la latitude, la longitudine ed il nome dell'EVSE del GCP il cui uri è stato ricavato dalla query precedente (ovviamente nel nostro caso sarà l'uri di Arduino GCP). Dunque può essere istanziato il nuovo GCP: new GCP(gcpName, sumoRoadId, lat, lng) dove sumoRoadId è stato posto uguale a 35449271. Dato che un GCP dispone anche di una lista di EVSE, la terza query è volta a recuperare dalla SIB le informazioni riferite all'unico EVSE di Arduino GCP consistenti nelle variabili: evseName, maxCurrentOut, maxCurrentIn, power e voltage. Una volta ottenute, si può proseguire con la sua istanziazione e associazione all'Arduino GCP:

```
string evseName = sparql_new_gcp->binding;
EVSE* evse = new EVSE(gcp, evseName);
double maxCurrOut = atof(sparql_evse->binding);
double maxCurrIn = atof(sparql_evse->next->binding);
double power = atof(sparql_evse->next->next->binding);
double voltage = atof(sparql_evse->next->next->next->binding);
evse->maxCurrentDensityOut = maxCurrOut;
evse->maxCurrentDensityIn = maxCurrIn;
evse->power = power;
evse->voltage = voltage;
gcp->evseList.push_back(evse);
```

da notare il campo binding della struttura ss_nodes_t: questo campo è un array di 100 caratteri contenente il valore effettivo dei risultati.

- void printResult(string query): questo è un metodo di utilità da me implementato al fine di effettuare la query passata in input, per stamparne

i risultati in fase di DEBUG.

Battery

L'unica *feature* modificata all'interno di questo modulo è quella che riguarda il ciclo di ricarica all'interno di handleCharge(): dopo aver verificato che socPerc sia minore di maxSoc e la condizione viene soddisfatta, si controlla se un'auto si sta caricando presso l'Arduino GCP (if(traci->getRoadId() == "35449271")), se così non è l'aggiornamento di socPerc con la Formula 3.1 viene effettuato da questo modulo, altrimenti viene delegato ad Arduino GCP. La richiesta di ricarica presso Arduino GCP viene inoltrata tramite sendUDP() del modulo CarLogic al quale si passa la stringa "*Request*". Dopo aver fatto ciò l'auto si mette in attesa di ricevere il valore di socPerc dall'Arduino e quando tale valore sarà uguale a 1.0, allora l'auto potrà ripartire.

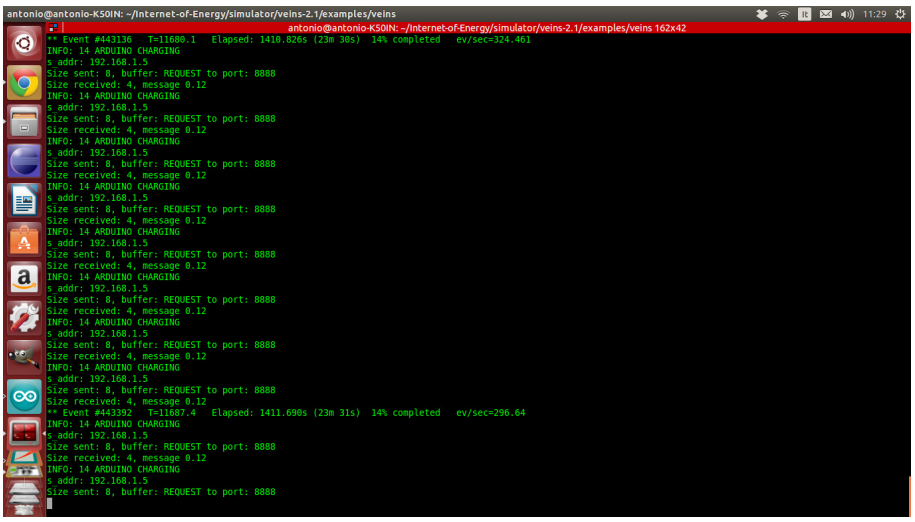

Figura 4.5: Richiesta stato batteria - comunicazione con Arduino GCP

Capitolo 5

Un esempio di Applicazione Mobile

Oltre alla parte di comunicazione tra la colonnina di ricarica emulata e la piattaforma di simulazione, il mio progetto ha previsto anche l'implementazione di una funzionalità molto aperta a possibili estensioni future e che, se proseguita, comporterà la modifica del modello ontologico di IoE.
Tale ulteriore funzionalità riguarda la comunicazione tra Arduino GCP ed uno smartphone il quale si connette (sempre tramite l'ausilio di socket di tipo UDP) alla colonnina emulata e richiede un codice di prenotazione. Dopo aver ottenuto il codice, l'utente possessore dello smartphone si recherà con l'auto da ricaricare presso la colonnina e digiterà il codice ottenuto e quest'ultimo, se verificato, darà la possibilità di ricaricare il proprio EV. In questo capitolo descriveremo questa ulteriore funzionalità.

5.1 Applicazione Android

L'applicazione smartphone di questo progetto è stata realizzata per il sistema operativo mobile **Android** con l'ausilio dell'IDE **Eclipse** nel quale è stato installato l'**Android SDK**. Nonostante il mondo Android sia ormai famosissimo e conosciutissimo, risulta comunque utile introdurre quelle sue componenti e caratteristiche risultanti utili per lo sviluppo della mia appli-

cazione.

5.1.1 Activity

La componente principale di un'applicazione Android è sicuramente l'**Activity** che realizza l'interfaccia per l'applicazione e rappresenta verosimilmente una schermata.
Suo compito è quello di descrivere tale schermata e di gestire le azioni dell'utente che vi interagisce, tramite delle componenti che vedremo nella prossima sezione denominate **View**.
Un'applicazione può essere composta da più Activity o comunque un utente potrebbe aprire più Activity appartenenti ad applicazioni diverse; in entrambi i casi il sistema operativo gestisce l'avvicendamento di tali componenti secondo una struttura a *stack*: quando viene richiamata una nuova Activity, viene salvato lo stato di quella corrente, la nuova viene posta in cima allo stack al di sopra dell'altra. Nel momento in cui l'Activity che si sta utilizzando viene chiusa, essa viene rimossa dallo stack, viene ripristinato lo stato dell'Activity precedente che tornerà in cima allo stack. Quanto descritto è un semplice esempio di avvicendamento tra Activity, ma in realtà esse sono caratterizzate da un ciclo di vita molto più complesso in cui interviene il sistema operativo che gestisce la creazione, l'avvio, la pausa, il ripristino, ... di ciascuna secondo lo schema descritto nella figura seguente.

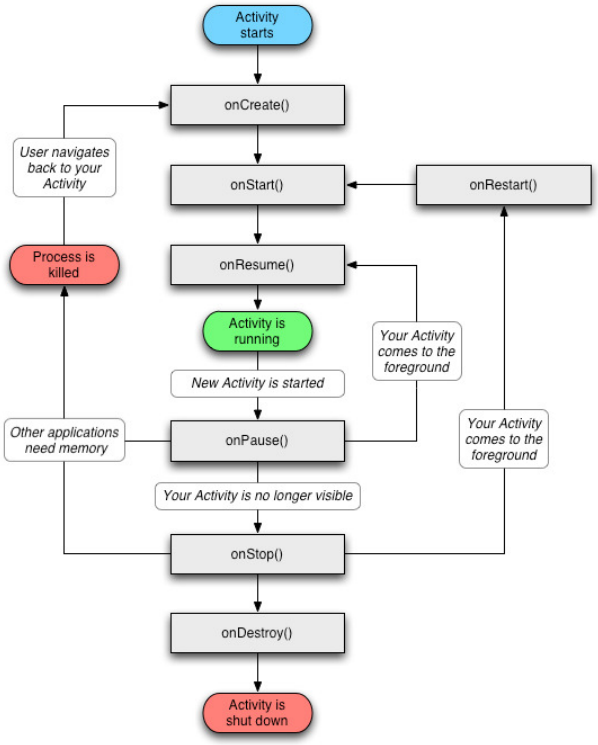

Figura 5.1: Ciclo di vita delle Activity

5.1.2 View

Parti essenziali senza le quali non si potebbero avere applicazioni mobili anche "belle da vedere" sono le View.
Una View è la componente grafica che si occupa di presentare ed arrichire un'applicazione gestendo anche quelle che sono le interazioni con l'utente tramite degli *handler* che ne catturano gli eventi.
Una View è descritta da un file XML ove vengono inseriti tutti i componenti che la realizzano quali possono essere dei Button, dei Layout, delle TextBox, ecc. Di fatti la prima operazione che si compie nel metodo onCreate() (che si occupa della creazione) di un'Activity è setContentView(R.layout.activity_main_id) dove activity_main_id è un intero che identifica il file XML della View principale.

5.1.3 Modalità di Sviluppo

Il linguaggio di programmazione utilizzato in Android SDK è Java del quale il gruppo di sviluppo di Android ne ha riscritto tutte le librerie reimplementandone la *Java Virtual Machine* in maniera più efficiente e chiamandola **Dalvik Virtual Machine**. Per sviluppare un'applicazione Android esistono due modalità di sviluppo:

- **Procedurale**: in questo tipo di approccio l'intera applicazione viene scritta utilizzando completamente e solo codice servendosi dei metodi delle librerie grafiche per implementare le View.

- **Dichiarativo**: in questo tipo di approccio si utilizza in combinazione codice Java con XML. Infatti le risorse di un'applicazione Android consistono in file XML residenti all'interno della cartella *res* dell'applicazione che ne determinano la loro struttura e permettono di realizzare componenti grafiche (come ad es. layout, immagini, colori, menu, ecc.) anche tramite l'ausilio di *tool* che permettono di spostare ed inserire gli elementi tramite *drag & drop*.
  Nonostante la mia applicazione sia di piccole dimensioni, ho comunque scelto di utilizzare tale tipo di approccio in quanto più chiaro e soprattutto permette di separare la parte grafica di un'applicazione dalla sua logica.

5.2 Implementazione

In questa sezione saranno descritti i dettagli implementativi dell'applicazione Android e dello sketch Arduino specializzato nell'emulazione del GCP e che implementa anche i comandi relativi alla digitazione del codice sulla colonnina elettrica. Verrà prima descritta la parte di implementazione riguardante la comunicazione tra l'applicazione mobile e Arduino GCP e successivamente la comunicazione inversa.

5.2.1 Comunicazione con Arduino GCP

L'implementazione dell'applicazione mobile è molto semplice ed è costituita da un'unica View il cui layout mostra un semplice button con la scritta *Send Reservation Request*. L'applicazione si compone di una sola classe che è la MainActivity. Nel suo metodo onCreate() viene settato il layout principale con setContentView(R.layout.activity_main), in cui activity_main identifica il file activity_main.xml e vengono inizializzate due View che sono di tipo TextView e raffigurano quindi delle stringhe sullo schermo che però inizialmente sono nascoste:

```
reservationText = (TextView) findViewById(R.id.reservationText);
reservationCode = (TextView) findViewById(R.id.reservationCode);
```

La prima TextView è posizionata appena al di sotto del button (chiamato sendButton) e mostra la scritta: " *Your Reservation code is:*", mentre la seconda TextView è inizialmente vuota e servirà per contenere il codice fornito da Arduino GCP ed entrambe le TextView verranno mostrate a schermo non appena l'applicazione riceverà la risposta della piattaforma elettronica.
Da notare come sono state inizializzate entrambe le componenti: metodo principe è findViewById() al quale viene passato come parametro l'id che identifica la risorsa nel file XML del layout ed in base all'id tale metodo cerca appunto la risorsa.
Per quanto riguarda la parte di comunicazione è stato implementato il metodo public void sendReservationRequest(View button) che viene associato al sendButton nel file activity_main.xml all'interno del quale vengono sfruttate le potenzialità della libreria Java java.net la quale implementa tutti i meccanismi relativi ai socket.
Inizialmente vengono impostate la porta tramite la quale inviare dati all'Arduino (sempre la 8888) e la porta di ricezione dell'applicazione, viene inoltre impostato l'indirizzo IP dell'Arduino tramite una struttura dati particolare denominata InetAddress. L'istanziazione di un socket UDP è effettuata semplicemente eseguendo l'istanza della classe DatagramSocket (DatagramSocket socket = new DatagramSocket()). Mentre l'invio del messaggio viene descritto dal seguente listato:

```
String messageString = "Reservation Request";
int messageLength = messageString.length();
byte[] message = messageString.getBytes();
DatagramPacket packet = new DatagramPacket(message,
    messageLength, targetAddress, port);
socket.send(packet);
```

In cui si nota che il messaggio da inviare ad Arduino GCP è la stringa "*Reservation Request*" dalla quale si ricavano la lunghezza e i byte che saranno utili per l'inizializzazione del pacchetto da inviare insieme alla porta sulla quale Arduino è in attesa.
Dopo l'invio l'applicazione resta in attesa sulla sua porta denominata serverPort e che ho impostato a 6000 e per rendere possibile la ricezione si deve: istanziare un nuovo DatagramPacket tramite l'ausilio di un array di byte (receivedMessage), si deve istanziare un nuovo socket in cui questa volta il metodo di inizializzazione prende in input la serverPort e quindi poi si esegue la receive() che prende in input il pacchetto denominato receivedPacket. Una volta ricevuta la risposta, l'applicazione ricava il testo dal pacchetto con String text = new String(receivedMessage, 0, receivedPacket.getLength()) e successivamente rende visibili reservationText e reservationCode impostando il testo di quest'ultima con quanto ricavato dalla risposta del pacchetto UDP.

5.2.2 Comunicazione con l'Applicazione Mobile

Per quanto riguarda la comunicazione tra l'applicazione mobile e Arduino GCP, la struttura della piattaforma elettronica è la stessa descritta nel Capitolo 3 (con l'unica differenza che in questo caso verranno anche utilizzati i tasti della Keypad Shield), mentre lo sketch è implementato in maniera differente e le sue costanti e variabili globali principali sono:

- btnRIGHT(0), btnUP(1), btnDOWN(2), btnLEFT(3), btnSELECT(4), btnNONE(5) - sono le costanti che identificano i bottoni principale del Keypad Shield e vi è anche la costante che identifica il fatto che non è stato premuto alcun bottone.

- GREENLED(3), REDLED(2) - come per lo sketch precendente, tali costanti identificano i red verde e rosso.
- read_key - intero che sarà posto uguale ad analogRead(0) (che legge la porta A0 della Keypad Shield, dove in pratica sono collegati i tasti) per essere confrontata con dei determinati valori corrispondenti ai tasti premuti.
- leftright - variabile globale intera utile quando si deve muovere il cursore sullo schermo.
- random_code - intero nel quale sarà contenuto un codice random di 4 cifre calcolato dall'Arduino.
- typedCode - array di 4 interi contenenti i numeri digitati attualmente sullo schermo.
- p1, p2, p3, p4, p5 - array di 8 byte i quali rappresentano rispetivamente 1/5, 1/4, 1/3, 1/2 di barra ed una barra intera e torneranno molto utili per la creazione di una *progress bar*.
- mac - indirizzo MAC dell'Arduino.
- ip - indirizzo IP dell'Arduino.
- port - porta sulla quale l'Arduino è in ascolto.
- udp - istanza di EthernetUdp.

Dopo aver definito le costanti e le variabili globali utilizzati nello sketch, passiamo alla spiegazione delle funzioni setup(), loop() e delle funzioni di utilità implementate.

Setup

In questa fase viene inizializzato lo schermo LCD sulla prima riga del quale viene scritto "*Write the code*" e sulla seconda "*0000*" e con lcd.blink() il cursore dello schermo viene fatto lampeggiare. Inoltre vengono inizializzati i led, i char relativi a p1, p2, p3, p4, p5 e le comunicazioni seriale e UDP.

Loop - Fase di Ricezione di Richiesta di Codice

Ogni volta che viene avviata la funzione loop() di Arduino, si verifica se vi sono pacchetti UDP in entrata tramite una funzione denominata getReservationRequest(). Tale funzione, dopo aver verficato se vi sono dati in entrata (via udp.parsePacket()), legge il pacchetto in entrata e controlla se la stringa del buffer è uguale a "*Reservation Request*". Se la condizione è soddisfatta, allora viene creato un codice random di 4 cifre (random(1000, 10000) - viene generato un numero che può variare da 1000 a 9999) il quale viene convertito in un array di char che poi viene inviato tramite print() all'applicazione mobile.

Loop - Riconoscimento tasti premuti

Dopo aver verificato se vi sono pacchetti in entrata, ciclicamente si verifica il valore della variabile read_key che viene assegnato tramite la funzione readbutton() la quale ritorna una delle costanti sopra definite, i valori dei tasti sono i seguenti:

- read_key <50: btnRIGHT.
- read_key <250: btnUP.
- read_key <450: btnDOWN.
- read_key <650: btnLEFT.
- read_key <850: btnSELECT.
- read_key >1000: btnNONE.

Dopo aver ottenuto il valore del tasto da read_button(), viene effettuato uno switch-case ed i possibili casi sono:

- btnRIGHT: sposta il cursore con lcd.setCursor(moveRight(), 1) dove moveRight() si occupa di incrementare il valore di leftright e verificare che non superi la quarta cella della seconda riga dello schermo LCD.

- btnUP: invoca increaseCode() il quale si occupa di incrementare il valore di typedCode[leftright] e verificare che questo valore non superi 9, quindi viene stampato sullo schermo il nuovo valore in corrispondenza della cella il cui indice è leftright.

- btnDOWN: analogo a btnUP, ma in questo caso viene invocata decreaseCode() che decrementa il valore di typedCode[leftright] e verifica che non scenda sotto lo 0.

- btnLEFT: analogo a btnRIGHT con la differenza che viene invocata moveLeft() che decrementa il valore di leftright e verifica che non scenda sotto lo 0.

- btnSELECT: caso in cui è stato digitato il bottone di conferma per effettuare la verifica del codice - in questo caso viene in un primo momento invocata la funzione verifyCode() la quale concatena tutti i valori contenuti in typedCode ottenendo così un numero di 4 cifre che viene confrontato con random_code ed in base al risultato del confronto viene restituito un booleano. Se l'esito sarnègativo verrà stampata sullo schermo la scritta "*Permission Denied*" e dopo 1s sarà ripristinata la situazione attuale, se invece l'esito sarà positivo allora sarà stampata sullo schermo la scritta "*Authenticated*" e sarà acceso il led verde. Dopo 1s sarà stampata la scritta "*Charging...*" e verrà avviata la funzione createProgressBar(). Dato che in realtà in questo tipo di implementazione non vi è la comunicazione tra Arduino GCP ed il simulatore di IoE, allora ho scelto di effettuare un ciclo di ricarica "fasullo" in cui viene eseguito un ciclo while() finchè il valore di un certo contatore (count) non diventa maggiore di 16 (il numero delle celle dello schermo LCD): all'interno del ciclo viene fatto lampeggiare il led rosso e viene eseguito un ciclo for(), il cui limite superiore è 5, nel quale vengono man mano scritti i caratteri da p1 a p5 nella cella corrente dando l'impressione che la barra si stia componendo volta per volta. Dunque una volta che sulla cella corrente sarà stato scritto il carattere p5, count sarà incrementato di 1 e dunque verrà ripetuto lo stesso procedimento per le celle succes-

sive. Una volta che createProgressBar() sarà terminata, sullo schermo viene stampata la scritta "*Charging Completed*", dopo 1s entrambi i led saranno spenti ed infine sarà ripristinata la situazione iniziale.

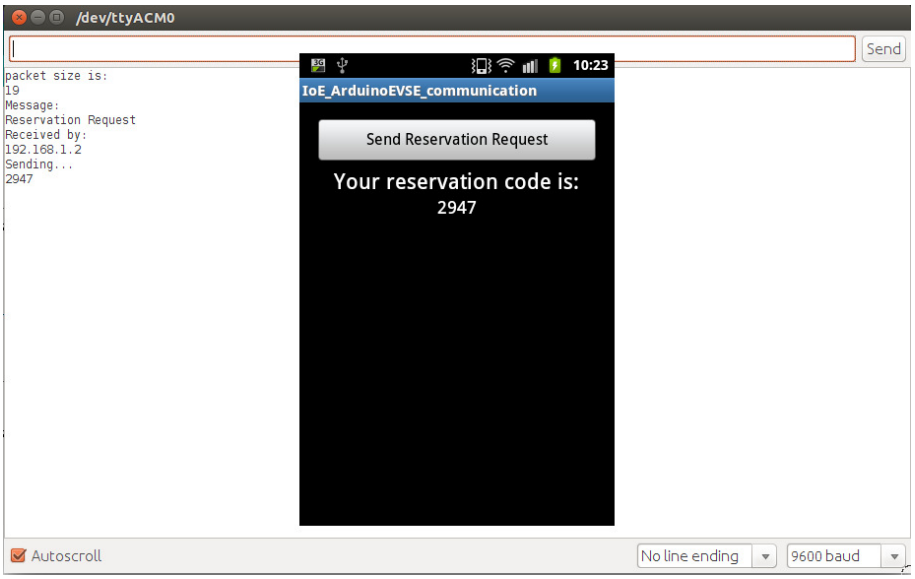

Figura 5.2: Cominunicazione smartphone - Arduino GCP

Con questa sezione si conclude tutta la parte relativa alla descrizione della realizzazione sia hardware che software di una colonnina di ricarica ed alla sua comunicazione con le componenti del mondo di IoE.

Capitolo 6

Conclusioni

Concludendo il mio elaborato posso ritenere che l'emulazione di colonnine di ricarica possa risultare molto utile nel contesto di Internet of Energy in quanto, in futuro, potrà comportare l'aggiunta di nuove funzionalità nel City Service, il quale sarà in grado di riscontrare dinamicamente (e non tramite un file XML statico) l'aggiunta di nuovi GCP all'interno della smart-grid fornendo così il supporto alla prenotazione verso di essi agli utenti. Tutto ciò farà sì che si possa ottenere un'analisi più approfondita dell'impatto della smart-grid in un contesto di traffico urbano visto l'utilizzo di componenti fisiche che emulano il comportamento dei GCP rispondendo in tempi "reali" e non a livello di simulazione. Inoltre, grazie allo sviluppo e le future estensioni dell'applicazione mobile realizzata, potrà essere introdotta un'ulteriore nuova funzionalità nel City Service consistente nella comunicazione di un codice di ricarica agli automobilisti quando le operazioni di prenotazione di un GCP saranno andate a buon fine e tutto ciò comporterà anche delle modifiche alla struttura fisica dei vari GCP. Vista la possibile introduzione di future funzionalità legate a tale progetto, ritengo utile elencare le possibili estensioni:

- possibilità di effettuare la reservation anche presso Arduino GCP (o altre componenti che emulano un GCP) - in realtà questa *feature* non è complicata da implementare, ma non è stata possibile testarla a causa di limiti architetturali della mia macchina.

- deregistrazione del GCP emulato sia dalla SIB che rimozione della sua istanza da OMNeT++.
- miglioramento dell'applicazione mobile la quale anzichè comunicare direttamente con Arduino GCP, comunica con la piattaforma di simulazione fornendo l'intervallo di tempo entro il quale l'utente voglia ricaricare l'auto. A questo punto il simulatore, tramite query alla SIB si informerà sullo stato attuale del GCP emulato, cioè verificherà se è libero o occupato nell'intervallo di tempo richiesto dell'utente. Dunque se risulterà libero OMNeT si occuperà di richiedere un codice di prenotazione al GCP emulato e dopo averlo ricevuto, lo comunicherà all'applicazione dell'utente. Infine l'utente si recherà presso il compnente digitando il codice e Arduino GCP si comporterà secondo le funzionalità già definite dal mio sketch.

Capitolo 7

Ringraziamenti

Desidero innanzitutto ringraziare i miei genitori in quanto in questi anni mi hanno sostenuto sia economicamente che moralmente dandomi la possibilità di conseguire un risultato tanto agognato e mio fratello per il suo desiderio di essere presente alla mia laurea.
Ringrazio il professor Luciano Bononi per avermi fornito la possibilità di fare parte di questo progetto ed i dott. Luca Bedogni, Marco Di Felice e Angelo Trotta perchè sono stati sempre disponibili e mi hanno sempre aiutato quando ho posto loro dei dubbi durante la realizzazione del mio progetto.
Ringrazio i miei amici e compagni di studio Gianluca, Emanuele e Lara e la mia amica Antonella con i quali ho trascorso la maggior parte delle mie giornate e si sono dovuti subire le mie ansie e preoccupazioni man mano che il traguardo si avvicinava.
Un grazie va anche al mio conquilino e amico Danilo col quale ho condiviso le mie esperienze allo studentato in questi (quasi quattro) anni e a tutti i miei parenti.
Grazie a tutti coloro che hanno fatto degli sforzi per essere presenti il giorno della mia laurea, ma anche a chi per causa di forze maggiori non è potuto venire.
Infine, ma non per ordine di importanza, grazie anche a Claudio e Luca per la fiducia riposta in me, i quali hanno prenotato i biglietti del treno per esseri presenti a Maggio nonostante non ci fosse la certezza che sarei riuscito a

laurearmi (anche se Luca non potrà più essere per ordini superiori) e grazie anche a Giuseppe per essersi riuscito ad organizzare ed essere venuto qui dalla Puglia.

Riferimenti Bibliografici

[1] Arduino: *http://www.arduino.cc*

[2] Arduino LCD KeyPad Shield: *http://www.dfrobot.com/wiki/index.php?title=Arduino_LCD_KeyPad_Shield_(SKU:_DFR0009)*

[3] Internet of Energy (IoE) - ARTEMIS European Project.
Project Website: *http://www.artemis-ioe.eu*

[4] Bedogni, Achtzehn, Petrova, Mahonen, Smart Meters with TV Gray Spaces Connectivity: A Feasibility Study for Two Reference Network Topologies - *Singapore, 02.07.2014*

[5] Luca Bedogni, L.Bononi, M. Di Felice, A. D'Elia, R. Mock, F. Montori, F Morandi, L. Roffia, S. Rondelli, T.S. Cinotti, F. Vergari.
An Interoperable Architecture for Mobile Smart Services over the Internet of Energy. *In: World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a. IEEE. 2013, pp. 1â6*

[6] CarStation project. Carstations - find your charge. *http://www.carstations.com*

[7] Coulomb Technologies, Inc. The Charge Point Application. *https://www.chargepoint.net.au*

[8] Inc. ECOItality - The Blink Mobile Application.
Project Website: *http://www.ecoitality.com*

[9] The EnelDrive Application - Project Website: *http://www.eneldrive.it*

[10] Fang X., Misra S., Xue G., Yang D. (2012). Smart Grid - The New and Improved Power Grid: A Survey.
IEEE Communications Surveys Tutorials, 14(4), 944â980. doi:10.1109/SURV.2011.101911.00087

[11] F. Morandi, L. Roffia, A. D'Elia, F. Vergari, T.S. Cinotti. RedSib: a Smart-M3 semantic information broker implementation. *Proc of FRUCT, Oulu, Finland, 2012*

[12] Ontology: *http://en.wikipedia.org/wiki/Ontology_(information_science)*

[13] OWL: *http://en.wikipedia.org/wiki/Web_Ontology_Language*

[14] S. Rondelli. Un framework di analisi e di servizi innovativi per la mobilità veicolare elettrica, 2014.
URL: *http://http://amslaurea.unibo.it/6750/*

[15] Semantic Web: *http://en.wikipedia.org/wiki/Semantic_Web*

[16] Smart-M3: *http://en.wikipedia.org/wiki/Smart-M3*

Lista delle Figure