



UNIVERSITÀ DEGLI STUDI DI MESSINA
FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

**SENSORISTICA E BACKEND PER IL
MONITORAGGIO AMBIENTALE IN AMBIENTE FI-
WARE**

Tesi di Laurea di:
Gioacchino BOMBACI

Relatore:
Prof. M. SCARPA

Correlatore:
Ing. F. LONGO

Ai miei genitori.

Indice

Introduzione	1
1 Piattaforma Fi-ware e le sue componenti	7
1.1 Introduzione	7
1.2 Nascita ed obiettivi	7
1.3 Architettura	9
1.4 Il Filab	14
1.5 I Generic Enablers	18
1.6 Il ContextBroker GE	19
1.6.1 Orion e l’interfaccia NGSI	23
1.6.2 Federazione tra istanze Orion	32
1.7 Il BigData GE	39
1.7.1 Servizio di elaborazione	39
1.7.2 Servizio di memorizzazione	41
1.7.3 Architettura	42
1.7.4 Cosmos	43
2 Hardware utilizzato: sistemi embedded e sensori	48
2.1 Introduzione	48
2.2 Raspberry Pi	50
2.2.1 Raspbian: installazione e configurazione	51
2.2.2 Caratteristiche hardware del modello B+	53
2.2.3 GPIO e la libreria python RPi.GPIO	54
2.3 Linino ONE	59
2.2.1 Linino OS: installazione e configurazione	61
2.2.2 Caratteristiche hardware	62
2.2.3 GPIO ed il linguaggio Arduino	63
2.4 ShinyeiPPD42NS	68
2.4.1 Il particolato	68
2.4.2 Principio di funzionamento dei sensori di	

Indice

particolato	70
2.4.3 Shinyei	71
2.5 OBD II bluetooth adapter	77
2.5.1 ELM327	79
2.5.2 L’adattatore OBD II	82
3 Descrizione del progetto ed implementazione	85
3.1 Introduzione	85
3.2 Panoramica generale del progetto	86
3.2.1 Livello di sensoristica	90
3.2.2 Livello di comunicazione e memorizzazione	94
3.2.3 Livello di interfaccia utente	97
3.3 Struttura dati	101
3.4 Livello sensoristica: collegamento dei sensori	104
3.4.1 Interfacciamento dei sensori con la Raspberry	104
3.4.2 Interfacciamento dei sensori con la Linino	108
3.5 Livello sensoristica: implementazione software	111
3.5.1 Shinyei.py	112
3.5.2 Shinyei.ino	115
3.5.3 Obd.py	118
3.5.4 Script NGSI per la connessione ad Orion	120
3.5.5 Send_measures.py	126
3.5.6 Server_rasp.py	129
3.5.7 Taxi_emulator.py	132
3.6 Livello di comunicazione e memorizzazione	133
3.6.1 La nostra VM sul Filab	134
4 Studio delle prestazioni	137
4.1 Introduzione	137
4.2 Simulazione in Python	138
4.2.1 Flusso del processo e parametri di simulazione	139
4.3 Configurazione singolo Orion: scenario e casi di studio	143
4.4 Federazione e studio della scalabilità	146
4.5 Confronti: Un Orion vs Federazione	149
4.5.1 Query	149
4.5.2 Update	151

4.6 Conclusioni e sviluppi futuri	153
A Codice Python per lo studio delle prestazioni del nostro sistema	156
A.1 Taxi_emul_toBenchOrionFederation.py	156
A.2 Taxi_emulator.py	165
B Script Matlab	168
B.1 diff_singolo_orion_5_10_20_30_utenti_all_request.m	168
B.2 diff_query_update.m	169
B.3 diff_singolo_doppio_orion_query_mean_totale.m . . .	170
Bibliografia e Sitografia	173

Introduzione

Negli ultimi anni, il settore informatico e di rimpetto anche il mondo reale è stato colpito da quello che si può definire il fenomeno del CLOUD. Tutti al giorno d'oggi, informatici e non, hanno almeno una volta sentito parlare di CLOUD.

Anche se non esiste una vera e propria definizione ufficiale, per “CLOUD Computing” in ambito informatico, si intende prettamente un insieme di tecnologie hardware e software (distribuiti e virtualizzati in rete) che consentono di offrire agli utenti servizi di storage ed elaborazione dei dati.

Più in generale, può essere visto come una forma di fornitura di servizi legati all’IT (Information Technology) che si avvale di nuove tendenze evolutive tra cui l’uso della virtualizzazione. In realtà, secondo la definizione data, potremmo tranquillamente affermare che il CLOUD esiste da molto più tempo. Pensiamo ad esempio alle e-mail web-based. Una volta le e-mail potevano essere inviate e ricevute grazie all’utilizzo di un programma (client di posta) in esecuzione su nostro PC privato. Oggi, invece, siamo tutti abituati all’idea che le nostre e-mail vengono memorizzate e processate su server che si trovano in una qualsiasi parte del mondo, e al quale possiamo facilmente accedere tramite un Web browser ovunque ci troviamo. Nuovi esempi di CLOUD Computing sono invece, la preparazione di documenti, il salvataggio dei dati sulla rete e la virtualizzazione di una macchina remota.

Nel mondo dell’IT si parla generalmente di tre diversi tipi di CLOUD Computing, con cui vengono forniti diversi servizi all’utenza finale, e sono:

- **Infrastructure as a Service (IaaS):** rappresenta l’utilizzo di risorse hardware (fisiche o virtualizzate), messe a disposizione da

Introduzione

qualche provider sulla rete, come un servizio. Le risorse vengono istanziate “on demand” in base alle richieste ricevute dagli utenti. Questo tipo di servizio ingloba al suo interno un altro tipo servizio CLOUD, il Daas (Data as a Service). Quest’ultimo rappresenta l’utilizzo di risorse hardware di memorizzazione, messe a disposizione da qualche provider sulla rete, per consentire l’archiviazione di dati personali degli utenti su server remoti. Google Drive e Dropbox sono esempi di servizi di questo tipo, ormai comunemente utilizzati.

- **Software as a Service (SaaS):** rappresenta l’utilizzo di un applicazione come un servizio, in quanto in esecuzione su un sistema remoto. Esempi di questo tipo di CLOUD sono le già menzionate e-mail web-based e Google Documents.
- **Platform as a Service (PaaS):** rappresenta l’utilizzo di un intera piattaforma software, piuttosto che un programma singolo, come un servizio. Una piattaforma software può quindi includere al suo interno diversi altri tipi di servizi. Google App Engine è un esempio di questo tipo.

I tipi di CLOUD Computing elencati e qualsiasi altro tipo di servizio CLOUD hanno tutti una matrice comune, ossia quella di essere servizi forniti da qualcuno (provider di servizi) e gestiti a nome degli utenti utilizzatori. Uno dei suoi principi base è quello di consentire all’utente l’uso del servizio in quanto tale, sollevandolo da ogni tipo di preoccupazione relativa all’acquisto, all’affidabilità e alla sicurezza di quest’ultimo. Ad esempio, utilizzando Google Documents, non ci dovremo preoccupare di problemi relativi all’acquisto della licenza, all’aggiornamento del software, al backup dei file creati, ecc... Il provider del servizio (Google) fa tutto questo per noi, lasciandoci la possibilità di concentrarci solo ed esclusivamente sul nostro lavoro da svolgere, quindi all’uso del servizio stesso.

Introduzione

Altra caratteristica comune di tutti i servizi CLOUD è che sono tutti disponibili on-demand, e possono essere acquistati così come viene acquistato un servizio telefonico o l'accesso alla rete internet.

Questo fenomeno CLOUD, la cui diffusione è stata indubbiamente incentivata dalla considerevole diminuzione dei costi d'abbonamento per l'accesso alla rete, e da un progressivo aumento delle velocità delle connessioni, è tutt'oggi in continua espansione. Dopo i primi grandi provider di questo servizio (Google, Amazon, Yahoo...) ne sono nati e continuano a nascerne tantissimi altri. Tutti i grandi colossi informatici hanno ormai la propria piattaforma CLOUD tramite la quale offrono servizi di storage ed elaborazione dati ad utenti di vario tipo. Gli utenti possono essere semplici clienti privati che sfruttano generalmente servizi CLOUD gratuiti offerti dai vari gestori, o delle imprese che, valutando i costi di installazione e mantenimento di un proprio sistema informatico aziendale, decidono di "affittare" risorse hardware o software da uno di questi provider.

Da questa seppur breve introduzione al CLOUD Computing possiamo estrapolare facilmente quelli che possono rappresentare in qualche modo i vantaggi e gli svantaggi legati all'uso di questa tecnologia. I vantaggi sono sicuramente caratterizzati dal fatto che un privato ed in particolar modo un'azienda, può mettere su il proprio sistema informatico più o meno complesso, abbattendo completamente i costi d'investimento iniziali ed i costi di mantenimento, eliminando le problematiche relative all'aggiornamento dei software utilizzati, all'affidabilità e alla sicurezza del sistema ed infine la possibilità di poter aggiungere o togliere servizi extra in base alle proprie necessità e all'andamento del proprio business. Gli svantaggi possono invece essere in qualche modo ricercati analizzando in maniera trasversale gli stessi punti a favore elencati precedentemente. Ad esempio, i costi d'investimento iniziali necessari all'installazione di un sistema informatico complesso, potrebbero risultare una spesa di gran lunga inferiore rispetto ai costi d'affitto

Introduzione

delle risorse nel lungo termine. O anche, il vantaggio di poter acquistare servizi da un fornitore, potrebbe risultare una decisione limitativa, in quanto si ha la possibilità di usufruire dei soli prodotti che il provider è in grado di fornire. Inoltre, l'uso di un servizio comporta la nascita di un rapporto di dipendenza dell'utente dal proprio fornitore di servizio, in quanto soggetto alle politiche e alle dinamiche interne al provider stesso. Per chiarire meglio il concetto potremmo paragonare il CLOUD Computing all'affitto di un appartamento completamente attrezzato, piuttosto che una casa di propria scelta completamente da arredare. Si riscontrano enormi vantaggi in termini di praticità, ma altrettante restrizioni su come si può viverci dentro e su quello che è possibile o no modificare.

E' chiaro comunque come questo fenomeno stia investendo corposamente anche il settore marketing, motivo principale per cui il CLOUD rappresenta in questo momento la principale tecnologia di interesse sia in ambito commerciale sia in ambito di ricerca. Da questo fermento nasce l'idea ed il progetto Fi-Ware.

Fi-ware è un progetto il cui obiettivo è quello di realizzare una piattaforma CLOUD pubblica per l'Europa, aperta ed innovativa che consenta ai semplici utenti, imprese e grandi organizzazioni di poter creare, fornire o usufruire di servizi digitali versatili e di alta qualità. Lo scopo finale è quello di promuovere l'innovazione e l'imprenditorialità, e ciò viene garantito tramite la fornitura di API aperte che sviluppatori, aziende e chiunque possono sfruttare per inventarsi e creare un proprio servizio, e che possono successivamente rivendere attraverso il MarketPlace della piattaforma stessa. La piattaforma dovrà essere in grado di soddisfare le esigenze sia dei clienti finali, sia delle imprese. Dovrà essere in grado di gestire al meglio la sempre più enorme quantità di dati e di informazioni e renderle fruibili agli utenti in modo semplice. Dovrà fornire ai clienti un facile accesso all'infrastruttura, affinchè possano utilizzare efficacemente le proprie applicazioni d'interesse e

Introduzione

migliorare il loro quotidiano. Dovrà chiaramente risultare un sistema affidabile, garantendo i requisiti di sicurezza necessari e rispettando la privacy degli utenti. L'idea è quella che tale piattaforma possa contribuire a sfruttare la rete internet come un programma utile per la società, e che possa rendere l'ambiente circostante un posto migliore e più sicuro in cui vivere. Il progetto nasce anche con l'obiettivo di contrastare i grandi colossi del settore e realizzare quindi una piattaforma innovativa per l'Europa che possa frenare parzialmente i confini di mercato dei suddetti Google, Amazon, ecc...

Per dare un'idea dell'importanza, e delle massicce aspettative che si hanno da questo progetto, basterebbe ricordare gli 80 milioni di euro di fondi stanziati dalla comunità europea ed elencare qualcuna delle aziende partner del progetto, aziende del calibro di: IBM, Telefonica, Intel, Siemens e via dicendo. Il progetto mira chiaramente ad espandersi e a diventare una realtà nel più breve tempo possibile, motivo per cui è stato predisposto un programma di accelerazione attraverso cui, gruppi di lavoro, startup, imprenditori web e piccole e medie imprese possono presentare le loro idee di progetto partecipando a delle Open Call. Tali idee di progetto, oltre ad essere idee innovative, devono ovviamente rientrare all'interno della logica del progetto Fi-ware, affinché possano essere selezionate dall'acceleratore di appartenenza e ricevere quindi i fondi prestabiliti per la realizzazione. Il nostro lavoro di tesi nasce quindi da un triplice interesse. Un interesse da parte dell'Università degli studi di Messina in quanto è recentemente diventato un nodo per questa piattaforma, un interesse da parte dell'azienda Arkimede s.r.l. presso cui abbiamo svolto lo stage pre-laurea e grazie alla quale abbiamo ricevuto la strumentazione ed il supporto tecnico necessario per poter partecipare alle Open Call del progetto, ed infine un interesse da parte nostro, in quanto abbiamo avuto la possibilità di poter realizzare un lavoro che ci ha permesso di confrontarci con il mondo reale ed acquisire esperienza sull'uso di nuove tecnologie. Il lavoro di tesi che ci è stato assegnato è stato, quindi, quello di

Introduzione

documentarci sullo stato dell'arte della piattaforma, capire la sua struttura, le sue componenti, studiare le API e i servizi messi a disposizione per poterci interagire, e sviluppare infine un'applicazione innovativa, scalabile e che fosse logicamente Fi-Ware compliant.

I capitoli successivi sono stati così organizzati: nel Capitolo 1 abbiamo riportato una presentazione più dettagliata della piattaforma e delle sue componenti, soffermandoci principalmente sulle componenti utilizzate nel progetto. Nel Capitolo 2 abbiamo illustrato caratteristiche e proprietà degli strumenti hardware utilizzati per la realizzazione del progetto. Nel Capitolo 3 abbiamo, invece, spiegato dettagliatamente tutta la fase di implementazione del progetto, a partire dall'interconnessione e l'interfacciamento tra i moduli hardware utilizzati, fino ad arrivare all'implementazione software. Infine, nel Capitolo 4, abbiamo riportato lo studio delle prestazioni relativo al motore della nostra applicazione e le eventuali soluzioni da adottare nel caso si volessero ottenere prestazioni migliori.

Capitolo 1

Piattaforma Fi-ware e le sue componenti

1.1 Introduzione

Nella sezione introduttiva abbiamo già accennato in termini molto generali a cosa sia il progetto Fi-ware e quelli che sono i suoi principali obiettivi. In questo capitolo descriveremo più nel dettaglio questi aspetti, cercando di capire meglio la filosofia adottata nello sviluppo del progetto, chi sono le parti interessate e i principali settori coinvolti. Andremo poi a raccontare un pò le principali caratteristiche della piattaforma, facendo una panoramica sulla sua architettura e sulle sue componenti, soffermandoci in particolar modo su quelle utilizzate all'interno del nostro lavoro.

1.2 Nascita ed obiettivi

Sono molteplici i fattori scatenanti che hanno portato alla nascita del progetto FiWare. Uno dei fattori principali, come detto anche nella parte introduttiva, è sicuramente la sempre maggior diffusione del Cloud Computing

Capitolo 1: Piattaforma Fi-ware e le sue componenti

e di piattaforme di distribuzione di servizi on demand in formula gratuita o a pagamento. Un ruolo fondamentale, è senza dubbio stato svolto dall'espansione sempre più massiccia dell'IoT (Internet delle Cose) e dalla sua progressiva maturità. Attraverso l'IoT, si riesce a connettere tra di loro un numero sempre maggiore di dispositivi in grado di generare moli di dati sempre più grandi. Tramite il processamento di questi dati, si riescono a produrre e far circolare un numero sempre maggiore di informazioni. Quest'espansione dell'IoT è stata favorita dalla diffusione di nuove tecnologie di rete, sia wireless che wired, che hanno contribuito ad avere connessioni di rete più stabili e veloci e che hanno fatto da apripista allo sviluppo di applicazioni "real time" anche in campo mobile. Tutti questi fenomeni, Fiware compreso, possiamo collocarli tra le componenti che daranno forma all'internet del futuro.

Obiettivo di alto livello del progetto Fiware è, difatti, la realizzazione della piattaforma dell'internet del futuro. Tale piattaforma dovrà migliorare la competitività globale dell'economia europea e dovrà portare benefici in ambito economico, politico e sociale. Il progetto si presuppone di realizzare delle interfacce standard e promuove lo sviluppo di applicazioni *agili*¹ ed innovative che possano migliorare la qualità dei servizi offerti in tutti i settori di mercato quali: i trasporti, l'energia e l'ambiente, la salute, il manifatturiero, l'agroalimentare e via discorrendo. Le parti interessate sono quindi diverse, si va dal semplice utente, che mira a sfruttare la piattaforma per usufruire di servizi per le proprie necessità in modo semplice e sicuro, alle piccole e medie imprese nonché grandi organizzazioni che puntano a sfruttare la piattaforma per modificare od organizzare i propri modelli di business per restare o accedere in maniera competitiva all'interno del mercato globale. Fiware si

¹ Con il termine agile ci si riferisce ad una metodologia di progettazione del software basata sui principi del "Manifesto Agile" del 2001. Il principio basilare di questa metodologia è quello di produrre software di qualità in tempi brevi. L'intera piattaforma è stata progettata seguendo i principi di questo Manifesto.

Capitolo 1: Piattaforma Fi-ware e le sue componenti

propone inoltre come piattaforma aperta, per favorire l'interoperabilità e l'affidabilità delle sue componenti, nonché la loro riusabilità [1]. Attraverso uno studio delle tecnologie e dei modelli di business sui vari settori di mercato, Fiware si propone come ulteriore obiettivo, quello di prevedere e progettare dei meccanismi di estensione che consentano di sostenere tutti quei settori che non sono ancora stati abbracciati dal progetto. Il fine ultimo è quello di validare l'approccio Fiware attraverso l'implementazione di applicazioni e servizi nei diversi settori precedentemente elencati, per promuovere la fiducia per gli investimenti su larga scala.

1.3 Architettura

Abbiamo già detto che Fiware nasce come piattaforma aperta, costituita quindi, da componenti che soddisfano la caratteristica della riusabilità. Queste componenti su cui Fiware si basa vengono chiamate **Generic Enablers** (GE), e sono fondamentalmente dei moduli software che offrono funzioni riutilizzabili e che sono quanto più generici possibile, nel senso che devono essere in grado di poter essere condivisi e utilizzati in una molteplicità di settori differenti. Proprio questa caratteristica della generalità distingue i Generic Enablers dagli Specific Enablers. Questi ultimi sono dei moduli software che possono essere condivisi ed utilizzati da più applicazioni, ma che sono sfruttabili all'interno di un limitato numero di settori. Gli obiettivi primari del progetto sono, infatti, stati quelli di identificare e definire i principali GE per la piattaforma e sviluppare delle implementazioni di riferimento (GEi). Per ognuno dei GE fin ora definiti sono state fissate una serie di specifiche che consentono a chiunque di poter realizzare una propria personale implementazione. Uno sviluppatore di applicazioni per la piattaforma dovrà quindi prendere visione di tali specifiche per poter interfacciare correttamente

Capitolo 1: Piattaforma Fi-ware e le sue componenti

la propria applicazione con i GE d’interesse. Uno sviluppatore di un GE dovrà attenersi scrupolosamente alle specifiche stabilitate, affinché più implementazioni di uno stesso GE siano sostituibili ed interscambiabili tra di loro.

L’architettura di riferimento della piattaforma Fiware è strutturata e suddivisa in un certo numero di capitoli tecnici. All’interno di ogni capitolo vengono descritti ed illustrati quelli che sono i GE di riferimento del capitolo stesso [2].

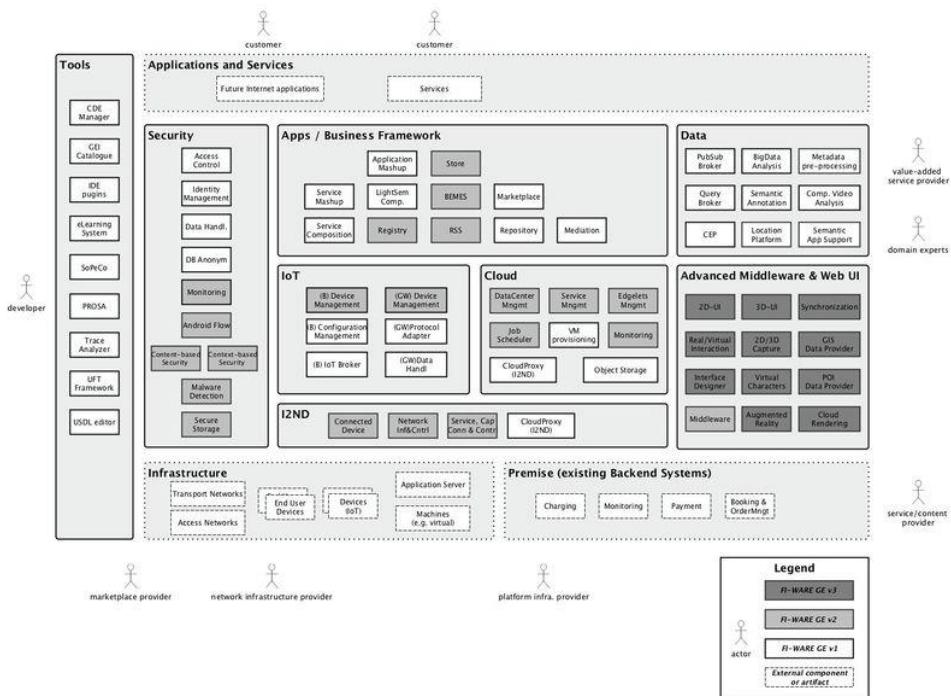


Figura 1.1: Architettura di riferimento della piattaforma Fiware

I capitoli tecnici in cui è stata suddivisa l’architettura sono:

- **Cloud Hosting:** questo capitolo comprende tutti quei GE che fungono da base per la progettazione ed il funzionamento dell’infrastruttura Cloud, che consente lo sviluppo e la gestione delle applicazioni e dei servizi. In base alle esigenze di ogni applicazione si

Capitolo 1: Piattaforma Fi-ware e le sue componenti

riescono a fornire capacità di hosting e di astrazione delle risorse differenti.

- **Data/Context Management:** questo capitolo comprende tutti i GE che facilitano lo sviluppo di applicazioni che richiedono la raccolta di dati, il loro processamento e la loro pubblicazione. Tali GE sono chiaramente in grado di gestire flussi di dati real time e manipolare grosse quantità di dati da cui estrarre nuove informazioni e conoscenze.

- **Internet of Things (IoT) Services Enablement:** questo capitolo racchiude tutti i GE che consentono l’interfacciamento e la coesione di differenti dispositivi, quali sensori e attuatori, alla piattaforma. In questo capitolo si parla di risorsa IoT intendendo un qualsiasi elemento software che consente di eseguire il rilevamento di una misura per un sensore o l’attuazione di un movimento per un attuatore. I GE di tale capitolo sono stati divisi in due categorie differenti, quella dei Gateway e quella dei Backend. I GE che appartengono alla categoria dei Gateway forniscono oltre alle normali funzionalità di inter-networking, quelle di traduzione dei protocolli tra i vari dispositivi, mentre i GE di Backend semplicemente consentono di effettuare la gestione dei dispositivi.

- **Applications/Services Ecosystem and Delivery Framework:** fanno parte di questo capitolo tutti i GE che contribuiscono alla formazione di quell’ecosistema di applicazioni e servizi di cui abbiamo parlato nella sezione di introduzione. Tali GE sono quelli che supportano l’intero ciclo di vita di un servizio dalla sua nascita alla sua vendita e distribuzione.

- **Security:** fanno parte di questo capitolo tutti i GE che mirano a rendere sicura la piattaforma, proteggendola da minacce e attacchi di vario tipo.

Capitolo 1: Piattaforma Fi-ware e le sue componenti

- **Interface to Networks and Devices (I2ND):** per far avanzare l’Internet delle Cose e far sì che i dispositivi possano acquisire intelligenza, grazie ad uno scambio continuo di dati tra loro, è necessaria la presenza di un’infrastruttura di rete che sia in grado di rendere accessibili ed utilizzabili tali dispositivi. Fanno parte di questo capitolo tutti i GE che offrono funzionalità per questo scopo.
- **Advanced Middleware and Web-based User Interface:** appartengono a questo capitolo i GE che contribuiscono a migliorare l’efficienza e la sicurezza delle comunicazioni tra applicazioni distribuite e che mirano a migliorare su tutti i livelli (prestazioni, grafica, e aggiunta di nuove funzionalità) l’esperienza web dell’utente, pur mantenendo la retrocompatibilità con i servizi web tradizionali.

I GE che fanno capo all’architettura di riferimento di un determinato capitolo possono poi essere implementati ed istanziati su una reale architettura, andando a costituire una “**Fiware Instance**”. In generale per istanza Fiware si intende un’architettura che ingloba al suo interno un insieme di GE collegati tra loro ed ad altri eventuali prodotti complementari propri del “**Fiware Instance Provider**”. Tali prodotti complementari consentono a quest’ultimo di differenziare le proprie offerte ed i propri modelli di business. Per “Fiware Instance Provider” si intende semplicemente l’azienda, l’organizzazione o l’impresa che implementa e gestisce la/e “Fiware Instance”, e che definisce lo scenario, il dominio e le modalità d’uso dell’istanza

Capitolo 1: Piattaforma Fi-ware e le sue componenti

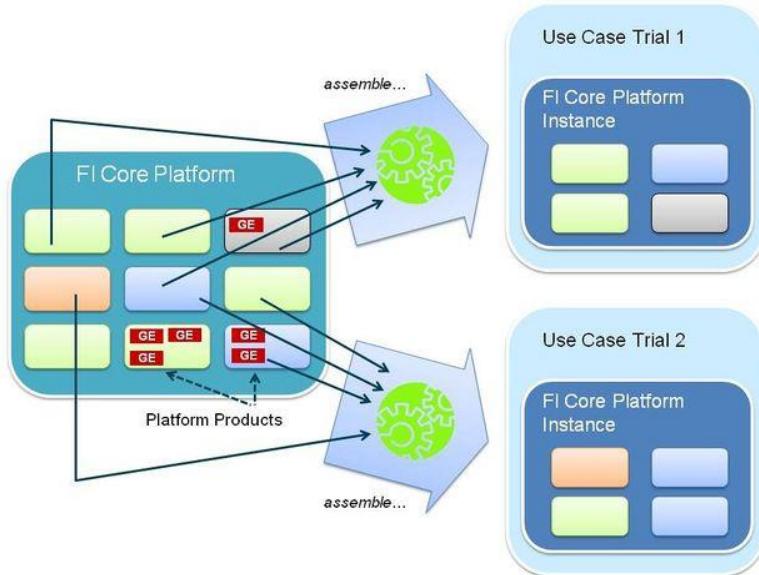


Figura 1.2: Due esempi di istanze Fiware

Il Fiware instance provider rappresenta uno dei ruoli che una azienda o ente generico possono svolgere all'interno dell'architettura della piattaforma. Gli altri ruoli sono:

- **Application Developer:** ruolo attribuito a chiunque (persona o azienda) sviluppi applicazioni che siano conformi e che interagiscano attivamente con la piattaforma.
- **Enabler Developer:** ruolo attribuito a chiunque (persona o azienda) sviluppi componenti software o sistemi più complessi che favoriscano lo sviluppo ed il funzionamento delle applicazioni sulla piattaforma. Architetturalmente parlando, gli Enablers e le applicazioni sono la stessa cosa, in quanto entrambi forniscono servizi. Ciò che cambia è però l'utenza a cui tali servizi sono rivolti, nel caso delle applicazioni l'utenza è rappresentata dagli utenti finali, nel caso degli Enablers l'utenza è rappresentata dalle applicazioni stesse.

Capitolo 1: Piattaforma Fi-ware e le sue componenti

- **Service Provider:** ruolo attribuito generalmente ad una qualsiasi azienda, organizzazione o ente che consente la distribuzione e la gestione di applicazioni o di Enablers per la piattaforma. Un ente che svolge il ruolo di “Service Provider” non esclude il fatto che possa svolgere anche il ruolo di “Application Developer” o “Enabler Devloper”.
- **Service Hosting Provider:** questo ruolo è coperto da qualsiasi azienda, ente o organizzazione che fornisce e gestisce l’infrastruttura sulla quale vengono ospitati Enablers ed applicazioni. Spesso può capitare che gli attori che svolgono questo ruolo, forniscano anche servizi Cloud per le applicazioni e gli Enablers ospitati, andando a coprire così anche il ruolo di “Service Provider”.
- **Service Aggregators:** svolgono questo ruolo le organizzazioni e gli enti che aggregano tra loro un insieme di servizi messi a disposizione da diversi “Service Provider”, per creare delle nuove offerte di servizi indirizzati però, ad una più ristretta cerchia di utenti.

1.4 Il FILAB

Un esempio di istanza Fiware che funge da punto di incontro tra i vari attori in gioco sulla piattaforma è il Fi-Lab. Questa nasce come una sorta di istanza di testbed, contenente tutte le implementazioni di riferimento dei principali GE, e a cui gli sviluppatori di applicazioni possono accedere liberamente per testare e sperimentare le funzionalità della piattaforma. Il compito del Fi-Lab è quindi quello di coinvolgere la collettività all’uso delle tecnologie Fiware, e stimolare ed incentivare utenti finali e fornitori di applicazioni alla costruzione di una comunità. Attraverso il portale del Fi-Lab all’indirizzo <https://cloud.lab.fiware.org/> si può quindi interagire con i

Capitolo 1: Piattaforma Fi-ware e le sue componenti

principali GEi dell’architettura Fiware, ma solo a seguito della registrazione e creazione di un account.

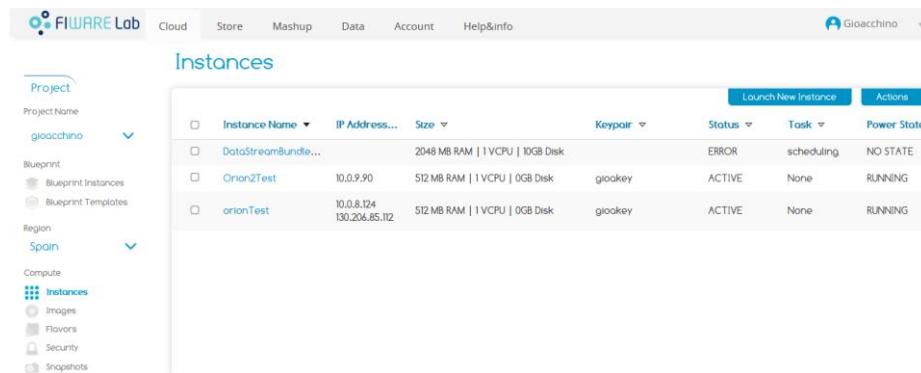


Figura 1.3: Screenshot del Cloud portal di Fiware

Dalla figura soprastante possiamo notare un barra di menù superiore, che ci consente di selezionare i diversi campi di interazione con la piattaforma. Entrando nella sezione *Cloud* potremo sfruttare le funzionalità di Cloud Hosting della piattaforma, basata su OpenStack². Tramite la sezione *Cloud* è possibile quindi deployare e gestire una o più macchine virtuali. Una macchina virtuale può essere creata cliccando su *Images*, dove è possibile trovare un elenco di immagini di sistema con già preinstallate le implementazioni di riferimento dei principali GE. È anche possibile creare un proprio template personalizzato, inserendo il SO desiderato, i software da installare (mongoDB, apache, ecc..) e gli eventuali GEi che si vogliono utilizzare, attraverso la sezione *Blueprint*. Dalla sezione *Region* è inoltre possibile selezionare la regione, su cui creare le proprie istanze. Dalla sezione *Security* è possibile creare una propria chiave per accedere in ssh alle proprie macchine virtuali, è possibile aprire le porte desiderate sulla propria VM e richiedere un’indirizzo

² OpenStack è un software open source rilasciato sotto licenza Apache, che consente di creare e controllare infrastrutture cloud pubbliche e private.

Capitolo 1: Piattaforma Fi-ware e le sue componenti

ip pubblico da assegnare ad una delle proprie istanze. Dalla sezione *Instances* è possibile visualizzare tutte le VM istanziate.

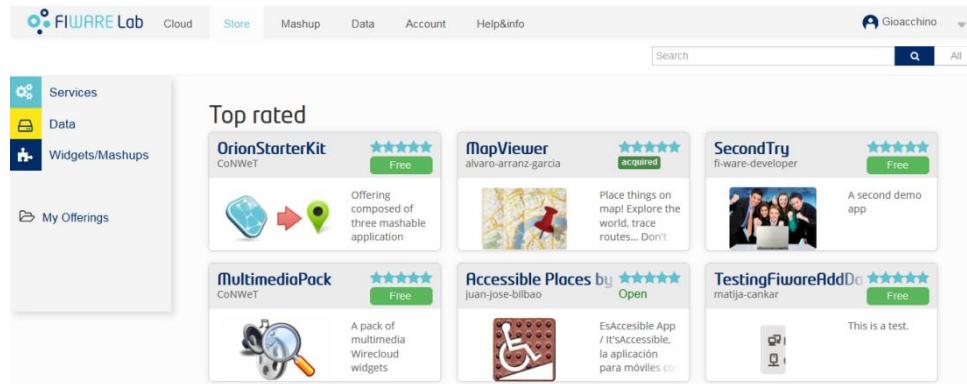


Figura 1.4: Screenshot dello Store portal di Fiware

Dallo *Store portal* è possibile pubblicare e gestire le proprie *Offerings*, ossia i propri widget, mashup, dati o servizi creati. Questo portale rappresenta quindi il Front-end utente per poter interagire con lo Store GE. È chiaramente possibile acquistare le Offerings pubblicate da altre organizzazioni. Le Offerings si dividono in Services, Data e Widget/Mashups.

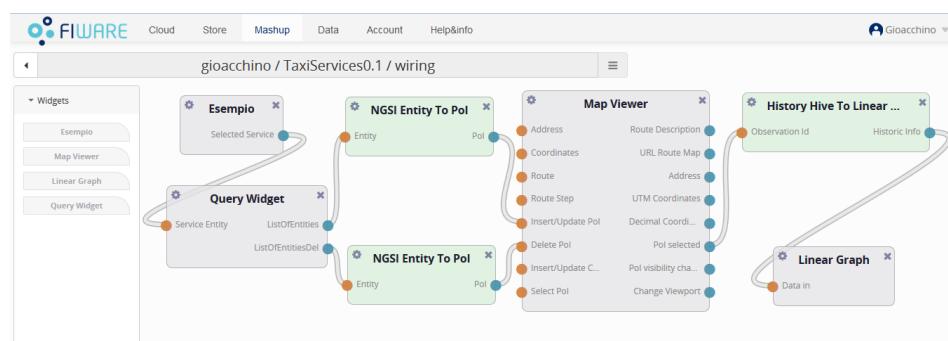


Figura 1.5: Screenshot del MashUp portal del Filab

Nel portale MashUp del filab è possibile realizzare appunto dei mashup tra widget che possono essere stati sviluppati da se o che possono essere stati

Capitolo 1: Piattaforma Fi-ware e le sue componenti

scaricati da uno store. Un Widget è una componente web che generalmente funge da interfaccia grafica per servizi di backend. Queste componenti web hanno la caratteristica che possono essere collegate tra di loro per formare nuove web application, che vengono appunto chiamate mashup. È possibile quindi, anche per utenti non esperti di programmazione, creare la propria applicazione web in maniera molto semplice.

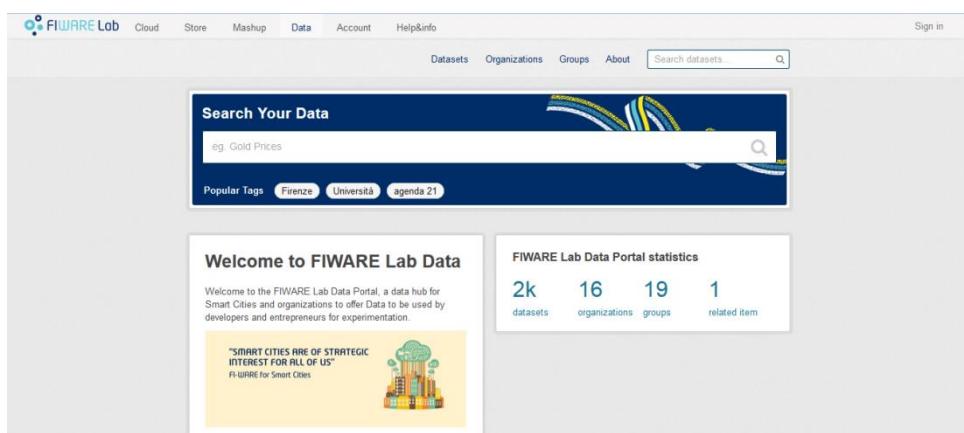


Figura 1.6: Screenshot del Data portal del filab

Infine abbiamo il *Data portal*, attraverso il quale è possibile pubblicare o ricercare dati in formato *OpenData*³, ossia dati che sono liberamente accessibili a tutti e che sono privi di brevetti. Questo portale rappresenta il front-end, tramite il quale gli utenti possono interagire con il CKAN GE. CKAN è un software di pubblicazione e gestione di contenuti, nel caso specifico i contenuti sono i dati che organizzazioni, aziende o pubbliche amministrazioni decidono di mettere in condivisione. CKAN GE ingloba praticamente le funzionalità di questo tool, esponendo la sua interfaccia grafica sul Data portal del Filab.

³ Con il termine *OpenData* si indica una modalità di rilascio di insiemi di dati, che rispetta la proprietà del paradigma “open”, ossia mancanza di brevetti e libero accesso da parte degli utenti.

Capitolo 1: Piattaforma Fi-ware e le sue componenti

1.5 I Generic Enablers

Abbiamo già introdotto e spiegato sommariamente cosa siano i Generic Enablers nel paragrafo sull’architettura della piattaforma. In questo paragrafo entreremo nel dettaglio dei GE utilizzati all’interno del nostro lavoro di tesi, spiegandone le funzionalità, i campi di applicazione e le modalità di interazione con gli altri GE.

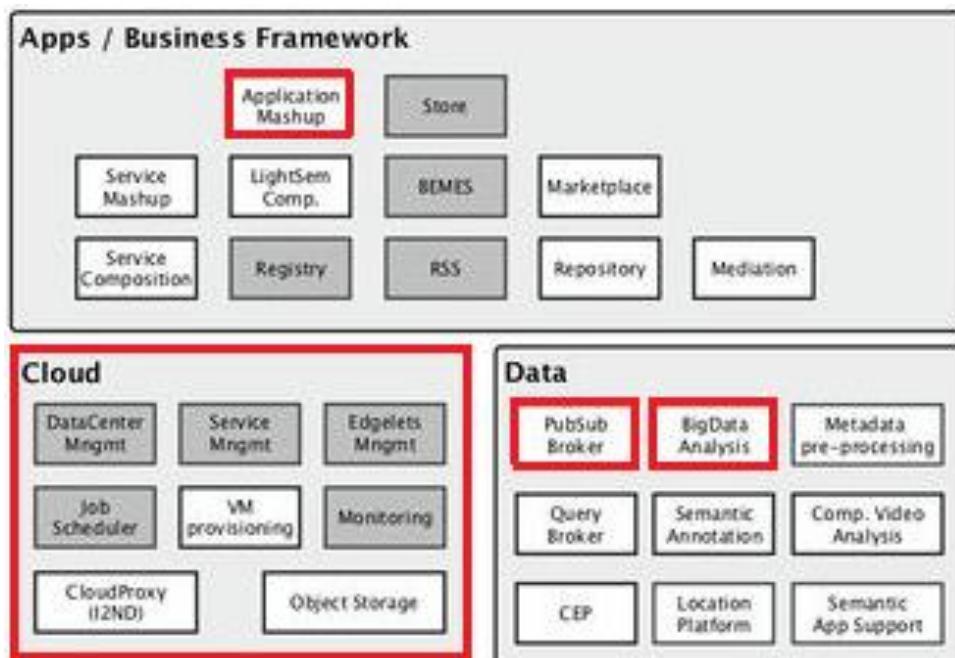


Figura 1.7: Componenti Fi-ware utilizzate nel nostro progetto

Nella figura soprastante abbiamo riportato la parte dell’architettura della piattaforma con cui abbiamo dovuto interagire per la realizzazione del nostro progetto. I blocchi evidenziati in rosso rappresentano le parti utilizzate. Abbiamo evidenziato l’intero capitolo del Cloud in quanto indirettamente, attraverso il Fi-Lab abbiamo utilizzato tutti quei GE che consentono di sfruttare il servizio di IaaS (Infrastructure as a Service), e che ci hanno consentito di istanziare tutte le macchine virtuali di cui necessitavamo per lo svolgimento del

Capitolo 1: Piattaforma Fi-ware e le sue componenti

nostro lavoro. Abbiamo quindi studiato ed utilizzato tre GE relativi al capitolo del Data Context Management, che sono:

- **ContextBroker**
- **BigData**
- **CKAN**

Ed un GE relativo al capitolo delle Applications/Services che è:

- **Application Mashup**

Nel prosieguo del capitolo illustreremo dettagliatamente le funzionalità del ContextBroker GE ed del BigData GE, accennando soltanto alle funzionalità degli altri due GE in quanto verranno meglio approfondite dal mio collega Antonio Caristia nel suo elaborato “Backend e interfaccia utente per il monitoraggio ambientale in ambiente Fiware”.

1.6 Il ContextBroker GE

Il ContextBroker GE è senza dubbio il GE più importante dell'intera architettura Fiware, in quanto come si può intuire dal nome stesso, funge da broker, quindi da centro di smistamento delle informazioni sulla piattaforma. L'importanza di questo modulo è motivata dalla continua crescita dell'Internet delle Cose e quindi da un progressivo aumento della mole di dati generati dai milioni e milioni di dispositivi collegati in rete. In questo scenario ci si rende conto della rilevanza che può assumere un modulo come il ContextBroker, che funga da ponte tra coloro che generano e pubblicano dati, anche detti **ContextProducers** (Produttori di contesto) e coloro che, invece, sono interessati ed elaborano i dati pubblicati per le proprie necessità, anche detti

Capitolo 1: Piattaforma Fi-ware e le sue componenti

ContextConsumers (Consumatori di Contesto). Il cambiamento dei dati di contesto sul ContextBroker rappresenta un evento. Sia applicazioni che altri GE sulla piattaforma possono svolgere entrambi i ruoli di produttori e consumatori di contesto. Il principio basilare su cui si fonda questo GE è che le due parti interessate, ossia produttori e consumatori, siano completamente disaccoppiati nella logica di funzionamento. Ciò vuol dire che i produttori di contesto potranno produrre e pubblicare dati senza doversi interessare di chi poi andrà ad utilizzare tali dati, di contro i consumatori di contesto, in maniera del tutto analoga, si dovranno preoccupare soltanto di consumare ed elaborare i dati relativi agli eventi di interesse, senza preoccuparsi di conoscere chi abbia generato o pubblicato i dati relativi a tali eventi. L'osservanza di questo principio è in parte garantita dal supporto di una doppia modalità di comunicazione (da/verso il Broker), che sono la modalità “*push*” e la modalità “*pull*”. Un produttore di contesti può, quindi, inviare sempre nuovi dati, quando disponibili, al Broker (modalità push). Il Broker di contro può richiedere delle informazioni di contesto al produttore, se questo prevede la possibilità di essere interrogato (modalità pull). Il ruolo del ContextProducer si può quindi suddividere in due sottoruoli all'interno dell'architettura del ContextBroker GE [4], che sono:

- **Il Context Provider:** un ContextProducer che consente di essere interrogato direttamente dal ContextBroker o da un ContextConsumer (funzionamento in pull mode), che richiede il recupero di determinate informazioni di contesto. Ogni ContextProvider deve registrare la sua disponibilità e le sue capacità presso un ContextBroker.
- **Il Context Source:** un ContextProducer che invia le informazioni di contesto al ContextBroker secondo una propria logica e

Capitolo 1: Piattaforma Fi-ware e le sue componenti

non espone interfacce per l’interrogazione (funzionamento il push mode).

Un consumatore di contesti, allo stesso modo, può estrarre informazioni dal Broker (modalità pull), o ricevere dal Broker le informazioni d’interesse quando disponibili (modalità push).

Gli altri concetti base necessari a comprendere a pieno l’architettura del ContextBroker GE sono i concetti di:

- **Entities**
- **Attributes**
- **Attribute Domains**
- **Context Elements**

Le **Entities** sono le rappresentazioni virtuali di un qualsiasi oggetto fisico, persona o gruppo di persone del mondo reale. Ogni cambiamento di dati di contesto è sempre riferito ad un’entità. Ogni entità si compone di due parti che sono:

- **Type**: identifica la categoria di appartenenza dell’entità (es. essere umano, dispositivo mobile, gruppo di entità ecc...)
- **Id**: identifica una determinata entità tra l’insieme di entità appartenenti alla stessa categoria.

Gli **Attributes** sono le informazioni che si hanno a disposizione relativamente ad una determinata entità. Gli attributi si compongono di:

- **Type**: identifica il tipo dell’attributo (es. temperatura)
- **Name**: identifica l’attributo specifico (es. temperatura dell’ambiente)
- **Value**: il valore di temperatura rilevato

Capitolo 1: Piattaforma Fi-ware e le sue componenti

Ogni attributo può avere associato dei *metadata*. I *metadata* rappresentano delle informazioni aggiuntive sull’attributo e anche questi sono costituiti dalla tripla <type, name, value>. Si possono ad esempio usare i *metadata* per specificare l’unità di misura del valore riportato nel campo *value* dell’attributo, o per riportare il tempo di rilevamento della misura.

Un **Attributes Domain** rappresenta un insieme di attributi che sono legati tra di loro da una qualche relazione logica. Ad esempio gli attributi “temperatura” e “pressione” potrebbero far parte del dominio “meteo”. Il ContextBroker GE offre la possibilità di eseguire le operazioni (query, update, ecc..) atomiche sui domini di attributo, in modo da garantire la consistenza sui dati degli attributi appartenenti al dominio.

Infine, per **Context Elements** si intende la struttura dati usata per lo scambio di informazioni relative ad un’entità.

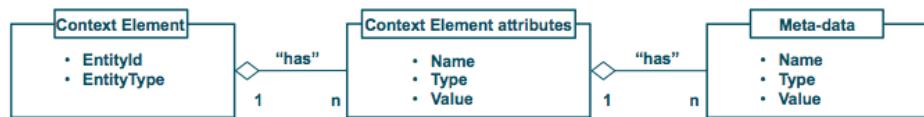


Figura 1.8: Diagramma della struttura del Context Element

Un Context Element contiene quindi:

- Un EntityId ed un EntityType per identificare in maniera univoca l’entità a cui i dati si riferiscono.
- Una sequenza di una o più triple che identificano gli attributi dell’entità.
 - Opzionalmente un nome di dominio per gli attributi dell’entità
 - Opzionalmente una sequenza di triple che rappresentano i *metadata* relazionati ai valori degli attributi

1.6.1 Orion e l’interfaccia NGSI

L’implementazione di riferimento del ContextBroker GE è rappresentata dall’Orion Context Broker. Così come definito nelle specifiche del ContextBroker GE, quest’implementazione utilizza delle interfacce standard per consentire la comunicazione ed il recupero delle informazioni da parte dei Producers e dei Consumers. L’interfaccia standard utilizzata a questo scopo in Fiware è l’interfaccia RESTful⁴ NGSI, basata sulle specifiche OMA NGSI⁵. OMA NGSI(Next Generation Service Interface) si divide in due sottointerfacce che sono **NGSI-9** ed **NGSI-10**. Entrambe le interfacce sono di tipo RESTful e basate sul protocollo HTTP. La differenza tra le due interfacce consiste nel fatto che, l’interfaccia NGSI-9 raggruppa un insieme di operazioni che consentono lo scambio di informazioni relative alla disponibilità delle informazioni di contesto [8]. Queste operazioni sono:

- registerContext
- discoverContextAvailability
- subscribeContextAvailability / unsubscribeContextAvailability / updateContextAvailabilitySubscription
- notifyContextAvailability

L’interfaccia NGSI-10 raggruppa, invece, un insieme di operazioni il cui scopo è quello di consentire lo scambio di informazioni di contesto. Di seguito

⁴ Il termine *RESTful* rappresenta un tipo di architettura software costituita da regole ben precise che devono essere rispettate per favorire la creazione di servizi web scalabili.

⁵ OMA NGSI sta per Open Mobile Alliance (OMA) Next Generation Services Interface (NGSI) Specification, e sono delle specifiche per lo scambio di informazioni di contesto. Tali specifiche sono consultabili all’indirizzo http://www.openmobilealliance.org/Technical/release_program/docs/NGSI/V1_0-20101207-C/OMA-TS-NGSI_Context_Management-V1_0-20100803-C.pdf

Capitolo 1: Piattaforma Fi-ware e le sue componenti

illustreremo quelle che sono le principali operazioni dell’interfaccia NGSI-10, in quanto quella utilizzata all’interno del nostro lavoro.

I tre principali gruppi di operazioni che una componente che espone l’interfaccia NGSI-10 è in grado di gestire sono:

- Richieste di informazioni di contesto
- Aggiornamenti delle informazioni di contesto
- Sottoscrizione per specifici aggiornamenti di informazioni di contesto

Possiamo dividere le operazioni di questa interfaccia in due categorie, le **Convenience Operations** e le **Standard Operations** [9].

Capitolo 1: Piattaforma Fi-ware e le sue componenti

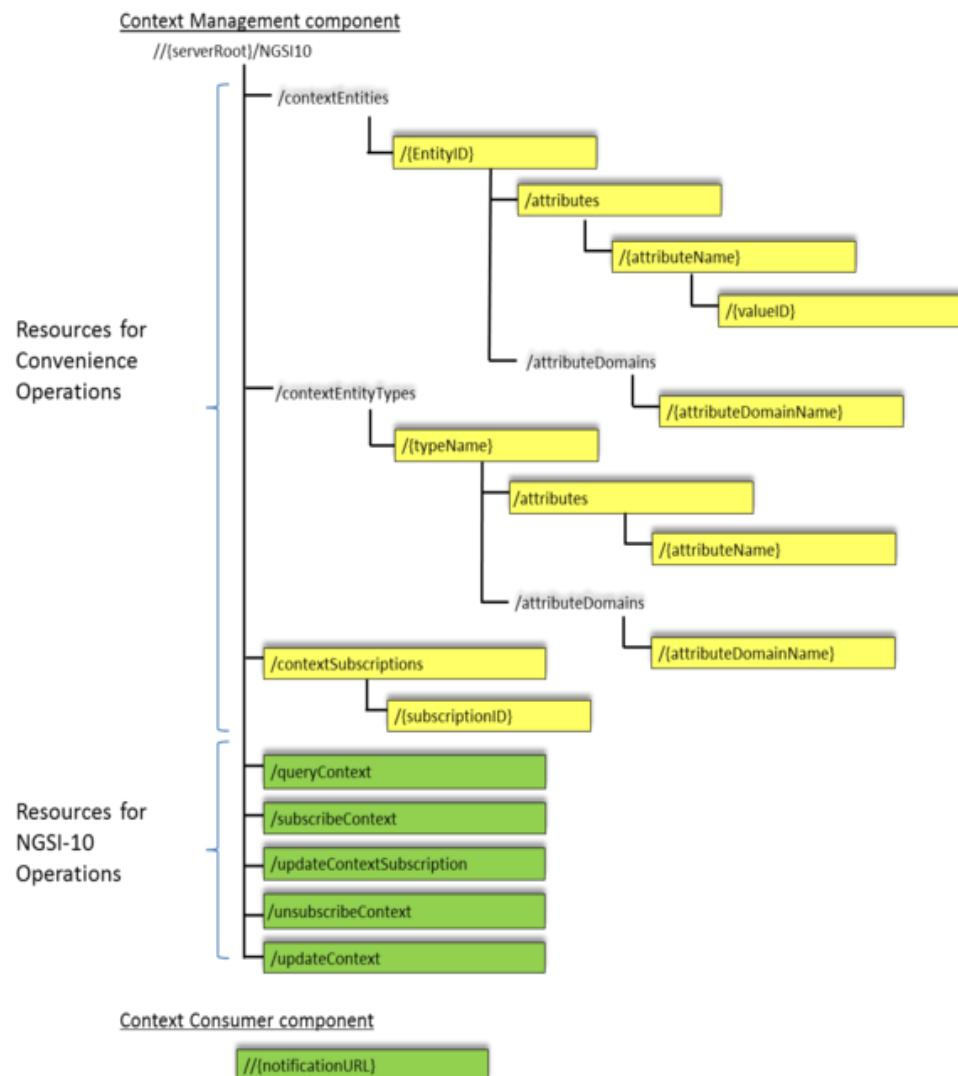


Figura 1.9: Albero delle risorse REST dell'interfaccia NGSI-10

Sia le Convenience che le Standard Operations si basano sul protocollo HTTP, la differenza sta nel fatto che le Convenience rispettano maggiormente il paradigma RESTful (utilizzano richieste POST, GET, PUT e DELETE) e facilitano l'uso dell'implementazione NGSI, mentre le Standard utilizzano solo il metodo POST del protocollo HTTP e rappresentano una pura traduzione

Capitolo 1: Piattaforma Fi-ware e le sue componenti

delle operazioni specificate in OMA NGSI. Quest’ultime sono le operazioni utilizzate nel nostro lavoro, andiamo quindi ad illustrare più approfonditamente.

QueryContext

La queryContext è l’operazione utilizzata per richiedere informazioni di contesto. L’url usata per inoltrare la richiesta sarà della forma

> http://server_ip:server_port/ngsi10/queryContext

Le informazioni da recuperare vengono inserite all’interno del corpo della richiesta, dentro un’istanza di tipo “*queryContextRequest*”, mentre il corpo della risposta sarà un’istanza di tipo “*queryContextResponse*”. Di seguito riportiamo un esempio di ciò che dovrà contenere il corpo di una POST Request per ottenere le informazioni relative all’entità di tipo “Room” e id “Room1” [6].

```
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
<entityIdList>
<entityId type="Room" isPattern="false">
<id>Room1</id>
</entityId>
</entityIdList>
<attributeList/>
</queryContextRequest>
```

Di seguito riportiamo invece un esempio del corpo della risposta ricevuto a seguito della precedente richiesta di query.

```
<queryContextResponse>
<contextResponseList>
```

Capitolo 1: Piattaforma Fi-ware e le sue componenti

```
<contextElementResponse>
<contextElement>
<entityId type="Room" isPattern="false">
<id>Room1</id>
</entityId>
<contextAttributeList>
<contextAttribute>
<name>temperature</name>
<type>float</type>
<contextValue>23</contextValue>
</contextAttribute>
<contextAttribute>
<name>pressure</name>
<type>integer</type>
<contextValue>720</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
<statusCode>
<code>200</code>
<reasonPhrase>OK</reasonPhrase>
</statusCode>
</contextElementResponse>
</contextResponseList>
</queryContextResponse>
```

È possibile notare come vengano riportati all'interno della risposta tutte le informazioni relative all'entità richiesta.

UpdateContext

L'updateContext è l'operazione utilizzata per aggiornare le informazioni relative ad un contesto. . L'url usata per inoltrare la richiesta sarà della forma:

> *http://server_ip:server_port/ngsi10/updateContext*

Capitolo 1: Piattaforma Fi-ware e le sue componenti

Le informazioni da aggiornare vengono inserite all'interno del corpo della richiesta, dentro un'istanza di tipo “*updateContextRequest*”, mentre il corpo della risposta sarà un'istanza di tipo “*updateContextResponse*”. Di seguito riportiamo un esempio di ciò che dovrà contenere il corpo di una POST Request per aggiornare le informazioni relative all'entità di tipo “Room” e id “Room1” [6].

```
<updateContextRequest>
<contextElementList>
<contextElement>
<entityId type="Room" isPattern="false">
<id>Room1</id>
</entityId>
<contextAttributeList>
<contextAttribute>
<name>temperature</name>
<type>float</type>
<contextValue>26.5</contextValue>
</contextAttribute>
<contextAttribute>
<name>pressure</name>
<type>integer</type>
<contextValue>763</contextValue>
</contextAttribute>
</contextAttributeList>
</contextElement>
</contextElementList>
<updateAction>UPDATE</updateAction>
</updateContextRequest>
```

È necessario sottolineare che l'operazione di *updateContext*, oltre ad essere sfruttata per l'aggiornamento dei valori degli attributi di un entità, viene anche usata per cancellare o creare una nuova entità. La struttura dell'xml all'interno del corpo della richiesta resta la stessa, l'unica cosa che va cambiata è il valore del tag *<updateAction>*. Questo tag andrà valorizzato ad APPEND nel caso si voglia creare una nuova entità, andrà valorizzato a DELETE nel

Capitolo 1: Piattaforma Fi-ware e le sue componenti

caso si voglia cancellare un'entità già esistente. Riportiamo di seguito un esempio di interazione tra un Consumer, un Producer e un ContextBroker nel caso di update e query context requests.

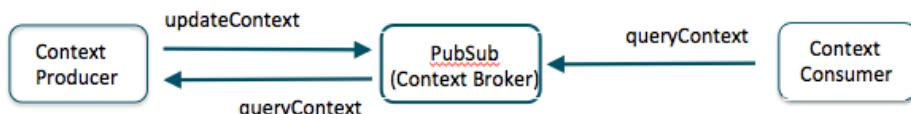


Figura 1.10: Esempi di interazione (update e queryContext)

- Un ContextConsumer può quindi richiedere informazioni di contesto ad un ContextBroker tramite una queryContext.
- Un ContextProducer può pubblicare informazioni di contesto (in maniera del tutto asincrona) tramite un updateContext su un ContextBroker, il quale manterrà la persistenza dei dati.
- Il ContextProducer (se è un ContextProvider) può implementare il metodo queryContext e rispondere ad alcune richieste di informazioni di contesto.

SubscribeContext

La subscribeContext è l'operazione utilizzata per sottoscrivere le informazioni di contesto. Viene utilizzata quando si vuole informare una determinata entità (che può essere un'applicazione, un servizio web, ecc...) del cambiamento di un'informazione di contesto. La subscribeContext viene quindi utilizzata per consentire ad un Consumer di ricevere informazioni su una determinata entità in maniera asincrona. L'url usata per inoltrare la richiesta sarà della forma

Capitolo 1: Piattaforma Fi-ware e le sue componenti

> `http://server_ip:server_port/ngsi10/subscribeContext`

Un tipico caso d'uso della subscribeContext request è il seguente:

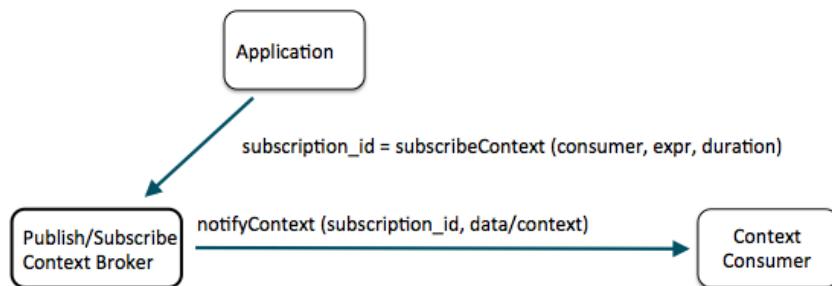


Figura 1.11: Esempio di interazione (subscription e notifyContext)

Un applicazione richiama la funzione di `subscribeContext` su un `ContextBroker`, specificando l'indirizzo del `Consumer` a cui dovranno essere notificati gli aggiornamenti, e la durata della sottoscrizione. Il `ContextBroker` ad ogni cambiamento dell'entità sottoscritta invierà al `Consumer` i dati di contesto attraverso una chiamata `NotifyContext`. Sono state definite due tipologie differenti di sottoscrizioni:

- **ONTIMEINTERVAL**: il `ContextBroker` invia i dati di contesto dell'entità sottoscritta al `Consumer`, in maniera periodica, rispettando l'intervallo di tempo fissato in fase di sottoscrizione
- **ONCHANGE**: il `ContextBroker` invia i dati di contesto dell'entità sottoscritta al `Consumer`, ogni volta che cambia almeno uno dei valori degli attributi sottoscritti.

L'entità da sottoscrivere viene inserita all'interno del corpo della richiesta, dentro un'istanza di tipo “`subscribeContextRequest`”, mentre il corpo della

Capitolo 1: Piattaforma Fi-ware e le sue componenti

risposta sarà un’istanza di tipo “*subscribeContextResponse*”. Di seguito riportiamo un esempio di ciò che dovrà contenere il corpo di una POST Request per sottoscrivere le informazioni relative all’entità di tipo “Room” e id “Room1” nel caso di tipologia ONTIMEINTERVAL [6].

```
<?xml version="1.0"?>
<subscribeContextRequest>
<entityIdList>
<entityId type="Room" isPattern="false">
<id>Room1</id>
</entityId>
</entityIdList>
<attributeList>
<attribute>temperature</attribute>
</attributeList>
<reference>http://localhost:1028/accumulate</reference>
<duration>P1M</duration>
<notifyConditions>
<notifyCondition>
<type>ONTIMEINTERVAL</type>
<condValueList>
<condValue>PT10S</condValue>
</condValueList>
</notifyCondition>
</notifyConditions>
</subscribeContextRequest>
```

Il tag **<attributeList>** conterrà la lista di attributi che si vuole che vengano notificati al Consumer. Il tag **<reference>** conterrà l’indirizzo del Consumer, il tag **<duration>** conterrà la durata della sottoscrizione nel formato ISO8601⁶, infine il tag **<condValue>** conterrà l’intervallo di tempo ogni quanto dovranno essere notificate al Consumer le informazioni di contesto sottoscritte. Di seguito riportiamo, invece, un esempio di ciò che dovrà contenere il corpo di

⁶ ISO8601 è uno standard internazionale per la rappresentazione di date ed orari

Capitolo 1: Piattaforma Fi-ware e le sue componenti

una POST Request per sottoscrivere le informazioni relative all’entità di tipo “Room” e id “Room1” nel caso di tipologia ONCHANGE [6].

```
<subscribeContextRequest>
<entityIdList>
<entityId type="Room" isPattern="false">
<id>Room1</id>
</entityId>
</entityIdList>
<attributeList>
<attribute>temperature</attribute>
</attributeList>
<reference>http://localhost:1028/accumulate</reference>
<duration>P1M</duration>
<notifyConditions>
<notifyCondition>
<type>ONCHANGE</type>
<condValueList>
<condValue>pressure</condValue>
</condValueList>
</notifyCondition>
</notifyConditions>
<throttling>PT5S</throttling>
</subscribeContextRequest>
```

I tag `<attributeList>`, `<reference>` e `<duration>` hanno la stessa funzione definita nel caso di sottoscrizione ONTIMEINTERVAL. Il tag `<condValue>`, invece, conterrà tutti gli attributi che, in caso di cambiamento, genereranno una notifyContext verso il Consumer.

1.6.2 Federazione tra istanze Orion

Una importante e comoda caratteristica del ContextBroker GE, che favorisce la scalabilità dei sistemi, è rappresentata dalla possibilità di poter federare tra loro più contextBroker in modo che il carico di lavoro possa essere

Capitolo 1: Piattaforma Fi-ware e le sue componenti

suddiviso e spalmato su tutti i partecipanti alla federazione. L’idea che sta dietro al concetto di federazione è quella chiaramente, di creare una fitta rete di contextBroker GEi, federati tra loro, in modo tale che si possa andare a formare un’architettura a livelli.

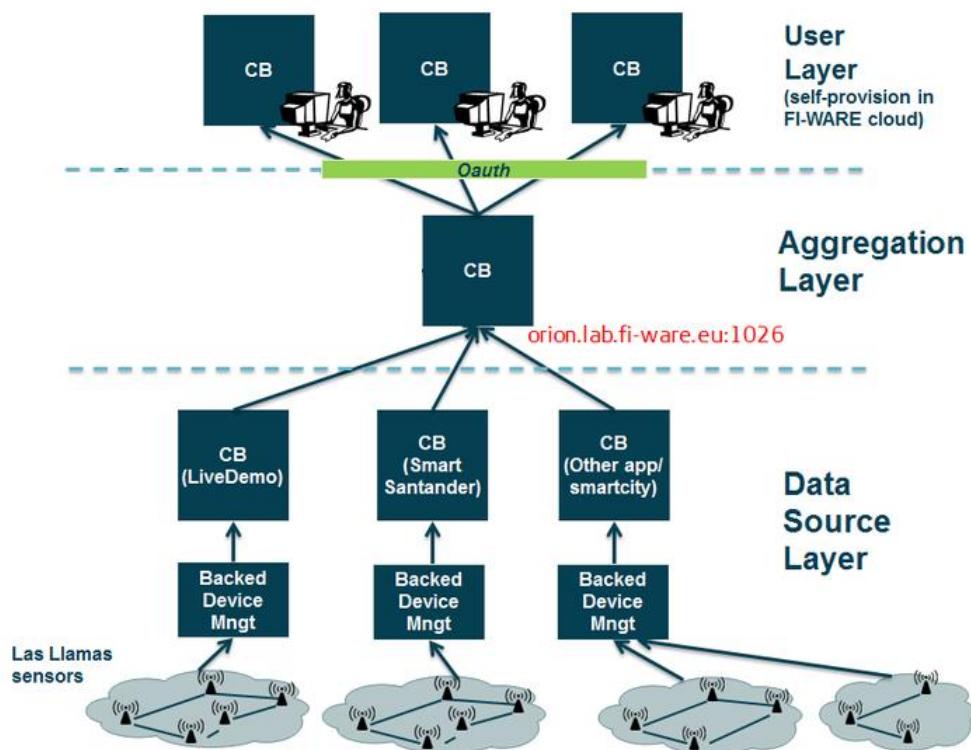


Figura 1.12: Esempio di federazione tra più contextBroker

Possiamo trovare il livello del “Data Source”, dove i ContextBroker collezionano dati direttamente da reti di sensori per conto di una o più applicazioni specifiche, un livello di “Aggregation” dove i ContextBroker svolgono la funzione di “switch”, una sorta di ponte tra il livello dei dati ed il livello utente, ed infine il livello “User”. Le tipologie di federazione che è possibile realizzare con i ContextBroker sono fondamentalmente due:

Capitolo 1: Piattaforma Fi-ware e le sue componenti

- **Push federation:** questo tipo di federazione prevede che un’istanza di ContextBroker GEi (es. un’istanza di Orion) si possa federare con altre istanze tramite delle semplici sottoscrizioni (*subscriptiononContext*). Ciò che succede è che le *notifyContext* inviate alle varie istanze federate, consentiranno a tali istanze di eseguire il processamento dei dati inviati, ottenendo così che tutti le istanze dei CB conterranno gli stessi dati aggiornati.
- **Pull federation:** questo tipo di federazione, invece, sfrutta l’operazione di *resisterContext* dell’interfaccia NGSI-9. La differenza sostanziale con l’approccio “push” è che, in questo caso, il ContextBroker federato agisce come un proxy, inoltrando operazioni di *queryContext* o *updateContext* ad altre istanze, senza memorizzare nulla sul proprio database locale.

Push Federation

Riportiamo un esempio di push federation per meglio capire quali sono i messaggi che devono essere scambiati tra i CB federati. Ammettiamo di lanciare tre istanze di Orion sulla stessa macchina e di metterle in ascolto su tre porte differenti (1030, 1031, 1032) [6].

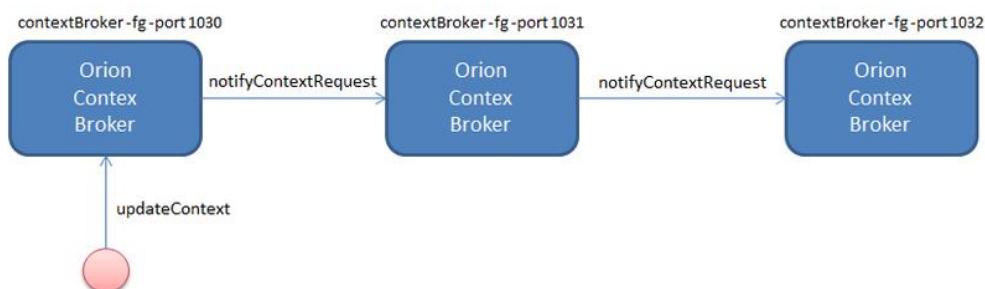


Figura 1.13: Un esempio di push federation

Capitolo 1: Piattaforma Fi-ware e le sue componenti

La sequenza di operazioni che devono essere eseguite per federare le tre istanze sono:

1. Inviare una subscribeContext all'istanza in ascolto sulla porta 1030 con <reference> l'indirizzo dell'istanza in ascolto sulla porta 1031.

```
(curl localhost:1030/v1/subscribeContext -s -S --  
header 'Content-Type: application/xml' -d @- | xmllint --  
format -) <<EOF  
    <?xml version="1.0"?>  
    <subscribeContextRequest>  
        <entityIdList>  
            <entityId type="Room" isPattern="false">  
                <id>Room1</id>  
            </entityId>  
        </entityIdList>  
  
        <reference>http://localhost:1031/v1/notifyContext</refere  
nce>  
        <duration>P1M</duration>  
        <notifyConditions>  
            <notifyCondition>  
                <type>ONCHANGE</type>  
                <condValueList>  
                    <condValue>temperature</condValue>  
                </condValueList>  
            </notifyCondition>  
        </notifyConditions>  
        <throttling>PT5S</throttling>  
    </subscribeContextRequest>  
EOF
```

2. Inviare una subscribeContext all'istanza in ascolto sulla porta 1031 con <reference> l'indirizzo dell'istanza in ascolto sulla porta 1032.

```
(curl localhost:1031/v1/subscribeContext -s -S --  
header 'Content-Type: application/xml' -d @- | xmllint --  
format -) <<EOF  
    <?xml version="1.0"?>  
    <subscribeContextRequest>  
        <entityIdList>  
            <entityId type="Room" isPattern="false">  
                <id>Room1</id>  
            </entityId>
```

Capitolo 1: Piattaforma Fi-ware e le sue componenti

```
</entityIdList>

<reference>http://localhost:1032/v1/notifyContext</reference>
    <duration>P1M</duration>
    <notifyConditions>
        <notifyCondition>
            <type>ONCHANGE</type>
            <condValueList>
                <condValue>temperature</condValue>
            </condValueList>
        </notifyCondition>
    </notifyConditions>
    <throttling>PT5S</throttling>
</subscribeContextRequest>
EOF
```

Creando a questo punto l’entità di tipo “Room” ed id “Room1” sull’istanza del CB in ascolto sulla porta 1030, attraverso una updateContext, avremo che quest’entità verrà creata e resterà aggiornata ad ogni cambiamento su tutte e tre le istanze. Interrogando uno qualsiasi dei CB federati otterremo le informazioni aggiornate relative all’entità in questione.

Pull federation

Riportiamo un esempio di pull federation per meglio capire quali sono i messaggi che devono essere scambiati tra i CB federati. Un semplice scenario potrebbe essere quello in cui un’applicazione registra presso un’istanza di ContextBroker, il ContextProvider che conterrà le informazioni relative ad un contesto.

Capitolo 1: Piattaforma Fi-ware e le sue componenti

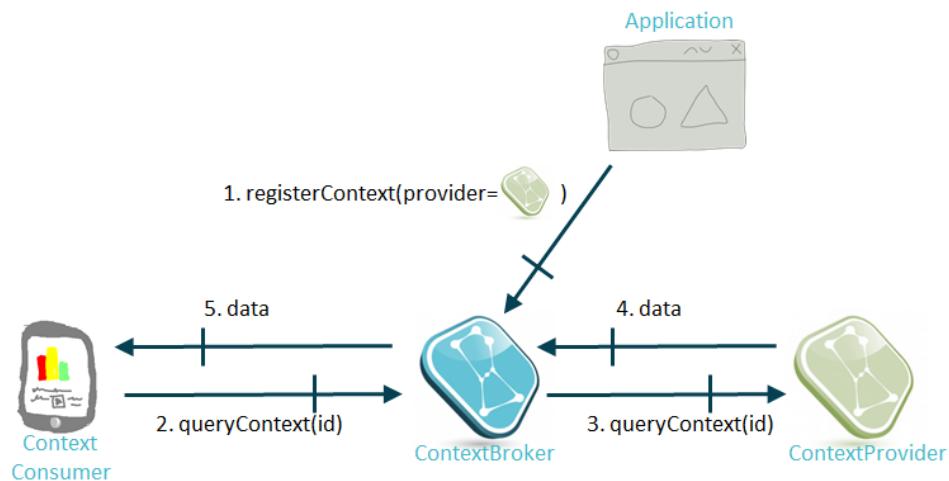


Figura 1.14: Esempio di Pull federation

Ammettiamo che il ContextProvider espone l’interfaccia NGSI-10 all’indirizzo <http://sensor48.mycity.com/ngsi10>, e che un applicazione vuole comunicare all’istanza dell’Orion Context Broker, che le informazioni relative all’attributo “temperature” dell’entità “Street4”, le potrà trovare presso un determinato ContextProvider. Il messaggio che l’applicazione dovrà inviare sarà [6]:

```
(curl localhost:1026/v1/registry/registerContext -s -S --  
header 'Content-Type: application/xml' -d @- | xmllint --format  
- ) <<EOF  
<?xml version="1.0"?>  
<registerContextRequest>  
  <contextRegistrationList>  
    <contextRegistration>  
      <entityIdList>  
        <entityId type="Street" isPattern="false">  
          <id>Street4</id>  
        </entityId>  
      </entityIdList>  
      <contextRegistrationAttributeList>  
        <contextRegistrationAttribute>  
          <name>temperature</name>  
          <type>float</type>  
          <isDomain>false</isDomain>  
        </contextRegistrationAttribute>  
      </contextRegistrationAttributeList>  
    </contextRegistration>  
  </contextRegistrationList>  
</registerContextRequest>
```

Capitolo 1: Piattaforma Fi-ware e le sue componenti

```
</contextRegistrationAttributeList>

<providingApplication>http://sensor48.mycity.com/ngsi10</provid
ingApplication>
    </contextRegistration>
    </contextRegistrationList>
    <duration>P1M</duration>
</registerContextRequest>
EOF
```

È possibile notare che l'operazione di *registerContext* ha un tag `<providingApplication>` all'interno del quale verrà inserito l'indirizzo del ContextProvider che conterrà le informazioni relative all'entità richiesta. A questo punto la registrazione è completata. Ogni qualvolta verrà inviata una *queryContext* sull'entità “Street4” all'istanza del ContextBroker, questo inoltrerà la query al ContextProvider opportuno, il quale restituirà i dati richiesti al Broker, che a sua volta li inoltrerà alla componente che ne aveva fatto richiesta, all'interno di un `<contextElementResponse>`.

```
<?xml version="1.0"?>
<contextElementResponse>
    <contextElement>
        <entityId type="Street" isPattern="false">
            <id>Street4</id>
        </entityId>
        <contextAttributeList>
            <contextAttribute>
                <name>temperature</name>
                <type>float</type>
                <contextValue>16</contextValue>
            </contextAttribute>
        </contextAttributeList>
    </contextElement>
    <statusCode>
        <code>200</code>
        <details>Redirected to context provider
http://sensor48.mycity.com/ngsi10</details>
        <reasonPhrase>OK</reasonPhrase>
    </statusCode>
</contextElementResponse>
```

Capitolo 1: Piattaforma Fi-ware e le sue componenti

L'unica differenza del corpo della risposta rispetto ad una normale risposta ad un'operazione di *queryContext* è che all'interno del tag <details> viene inserito l'indirizzo del contextProvider che ha risolto la query.

1.7 BigData GE

Questo GE nasce dalla necessità di dover elaborare e memorizzare le enormi quantità di dati che vengono generate dal mondo dell'IoT. I due principali servizi offerti da questo GE sono quindi il servizio di elaborazione e storage dei dati su un sistema distribuito. Entrambi i servizi consentono la creazione e configurazione di un cluster di macchine tramite riga di comando o tramite l'uso di REST API. L'uso di un servizio non implica necessariamente l'uso dell'altro. Inoltre, il GE consente l'utilizzo di un insieme di tecnologie quali, Hadoop, Hive, Oozie ecc... che agevolano la fornitura di tali servizi.

1.7.1 Servizio di elaborazione

L'efficienza della capacità di calcolo del BigData GE è garantita dall'utilizzo del paradigma MapReduce. Questo paradigma consente di eseguire elaborazioni su grosse quantità di dati memorizzate su sistemi distribuiti, esulando l'utente da eventuali problemi di parallelizzazione e scalabilità.

Capitolo 1: Piattaforma Fi-ware e le sue componenti

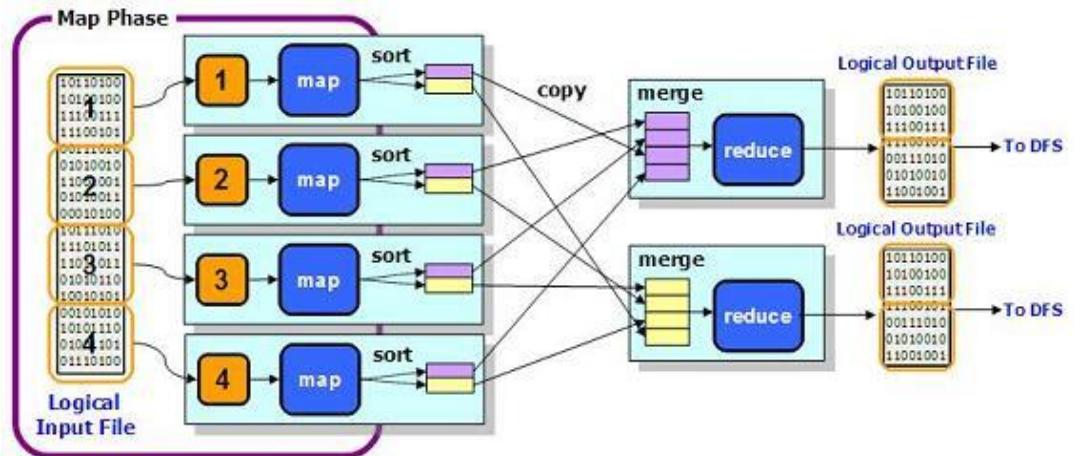


Figura 1.15: architettura del paradigma MapReduce

Tale paradigma è stato presentato da Google nel 2004 e si ispira, come si può intuire dal nome, alle note funzioni *map* e *reduce* della programmazione funzionale. Il problema principale dell'utilizzo di questo paradigma è che l'interfaccia utente è abbastanza complessa e quindi utilizzabile da utenti quali programmati, che possono conoscere ed interagire con le API esposte nei vari linguaggi di programmazione. Difatti, l'efficacia del servizio di elaborazione è garantito dall'uso di tecnologie, solitamente SQL-like, che consentono di astrarre la complessità delle API MapReduce tramite delle interfacce più semplici. Questi strumenti, quali ad esempio *Hive*⁷, consentono per esempio di tradurre una query eseguita da un utente su un determinato dataset in un MapReduce Job.

⁷ *Hive* è un software Apache che facilita l'esecuzione di query e la gestione di grandi quantità di dati memorizzati su sistemi distribuiti. Fornisce, inoltre, un meccanismo per creare delle strutture a partire dai dati e di interrogarli successivamente attraverso un comodo linguaggio di query SQL-like chiamato *HiveQL*.

1.7.2 Servizio di memorizzazione

Grazie al servizio di memorizzazione è possibile archiviare su un sistema distribuito enormi quantità di dati in maniera efficace. Un sistema di archiviazione distribuito è costituito da una rete di computer in cui una stessa informazione è memorizzata su più nodi. Ciò garantisce la recuperabilità dei dati nel caso di guasto di qualche nodo e contemporaneamente consente un tempo d'accesso più veloce ai dati in caso di necessità. I dati che si possono memorizzare sono di due tipi: *batch* e *streaming*. I dati batch sono quelli che necessitano di essere memorizzati prima di subire un'elaborazione, i dati streaming necessitano invece di essere elaborati in real time. In entrambi i casi, comunque, i dati devono seguire un ben preciso ciclo di vita che si divide in quattro parti:

- **Data Injection Phase:** sarebbe la fase di iniezione dei dati, durante la quale questi possono subire operazioni di normalizzazione, filtraggio, compressione e criptaggio.
- **Data Ingestion Phase:** durante la quale è necessario garantire che lo spazio per l'archiviazione del corrente volume dei dati sia disponibile.
- **Data Processing Phase:** durante la quale vengono riservate le capacità di calcolo opportune per la corretta elaborazione dei dati. Questa fase di elaborazione può a sua volta suddividersi in ulteriori sottofasi.
- **Data Result Phase:** rappresenta la fase finale, in cui si ottengono i risultati delle elaborazioni e i nuovi dati. Tali dati possono a loro volta essere interrogati o rappresentare dati d'ingresso per nuove elaborazioni.

1.7.3 Architettura

L’architettura del BigData GE espande fondamentalmente quella che è l’architettura di Apache Hadoop. Hadoop è un framework che consente l’elaborazione distribuita e l’accesso ad enormi quantità di dati attraverso l’uso di cluster di computer. Il punto di forza di Hadoop è senza dubbio la sua scalabilità e funzionalità [5]. Le tre componenti principali di Hadoop sono:

- Hadoop Distributed File System (HDFS): Il file system distribuito di Hadoop che fornisce un accesso ai dati molto performante.
- Hadoop Common: Il package contenente gli script e i file jar necessari per avviare Hadoop e tramite i quali si fornisce l’accesso ai vari file system supportati (HDFS, Amazon S3, FTP, ecc...).
- Hadoop MapReduce: Il motore che consente l’elaborazione parallela di enormi quantità di dati.

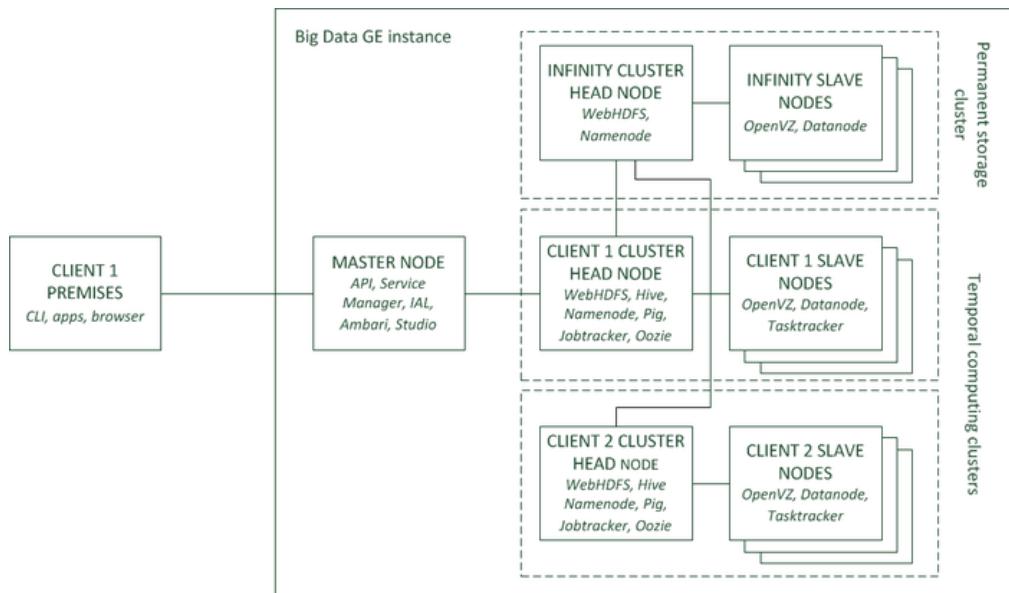


Figura 1.16: Architettura del BigData GE

L’architettura del BigData GE è basata su un nodo Master che si occupa della gestione dei vari software usati dal GE, agendo quindi come punto d’ingresso per tutte le applicazioni ed in generale, per tutte le utenze che richiedono la creazione di un cluster per l’elaborazione o per lo storage. Tutti i cluster creati sono costituiti da più macchine. Una di queste, chiamata *Head Node*, si occupa della gestione della memoria e delle elaborazioni dei dati all’interno del cluster, ma non memorizza né elabora nessun dato. Questa è la macchina a cui ogni utente deve accedere per poter eseguire una qualsiasi operazione sul proprio cluster privato. Gli altri nodi del cluster, anche detti *Slave Nodes*, sono quelli dove effettivamente vengono memorizzati i dati e vengono fatti girare i *MapReduce Jobs*. Siccome il ciclo di vita di un cluster privato HDFS è transitorio, i dati che necessitano di essere memorizzati in modo persistente vengono archiviati su uno speciale cluster, chiamato Infinity.

1.7.4 Cosmos

Cosmos rappresenta l’implementazione di riferimento del Big Data GE, il cui scopo è quello di consentire il deployment di cluster di computer privati (basati su ecosistema Hadoop), per la memorizzazione e l’elaborazione di enormi quantità di dati [7]. Tramite l’uso di questo GEi possiamo quindi:

- Eseguire operazioni di I/O su Infinity.
- Creare, usare e cancellare facilmente cluster di elaborazione privati.
- Analizzare i dati memorizzati attraverso una varietà di strumenti, che vanno dal semplice sistema di interrogazione come *Hive*

Capitolo 1: Piattaforma Fi-ware e le sue componenti

per la restituzione dei dati grezzi memorizzati, all'uso di sistemi di gestione dei MapReduce Job quali *Oozie*, per consentire elaborazioni più complesse sui dati.

- Gestire la piattaforma come un servizio tramite l'uso delle API o delle CLI fornite da Cosmos.

API e CLI

Sono diverse le API e le CLI tramite cui è possibile interagire con il BigData GE. Possiamo logicamente dividere le operazioni d'interazione con il GE in quattro livelli.

What	Locally (ssh'ing into the Head Node)	Remotely (connecting your app)
Clusters operation	Cosmos CLI	REST API
I/O operation	'hadoop fs' command	REST API (WebHDFS, HttpFS, Infinity protocol)
Querying tools (basic analysis)	Hive CLI	JDBC, Thrift*
MapReduce (advanced analysis)	'hadoop jar' command	Oozie REST API

Figura 1.17: Livelli di interazione con il BigData GE

Il primo livello è quello delle **Cluster Operations**, tramite le quali è possibile eseguire le operazioni di gestione e amministrazione del cluster (cancellazione, creazione, richiesta di dettagli, aggiunta di un nuovo utente, richiesta dei servizi attivi, ecc...). Tali operazioni possono essere eseguite

Capitolo 1: Piattaforma Fi-ware e le sue componenti

localmente attraverso le *Cosmos CLI* e da remoto (direttamente dalla nostra applicazione) tramite delle *API REST* di amministrazione.

Il secondo livello è quello delle **I/O Operations**, tramite le quali è possibile andare ad eseguire operazioni direttamente sui file memorizzati sull'HDFS del cluster. Tali operazioni possono essere gestite localmente attraverso gli *Hadoop filesystem commands*.

1. Hadoop filesystem commands

- Hadoop general command
 \$ hadoop
- Hadoop file system subcommand
 \$ hadoop fs
- Hadoop file system options
 \$ hadoop fs -ls
 \$ hadoop fs -mkdir <hdfs-dir>
 \$ hadoop fs -rmr <hdfs-file>
 \$ hadoop fs -cat <hdfs-file>
 \$ hadoop fs -put <local-file> <hdfs-dir>
 \$ hadoop fs -get <hdfs-file> <local-dir>

Figura 1.: Comandi filesystem Hadoop

Oppure è possibile eseguire queste operazioni sull'HDFS da remoto tramite delle API REST quali *WebHDFS* e *HttpFS*. Attraverso queste API è possibile eseguire delle operazioni di gestione dei dati nel cluster come ad esempio:

- Caricamento/scaricamento dati .
- Creazione/rinomina/lettura/scrittura di file e cartelle.
- Settaggio di permessi/proprietari per file e cartelle.

Capitolo 1: Piattaforma Fi-ware e le sue componenti

Queste API vengono chiaramente utilizzate quando si deve accedere alle risorse dell'HDFS da un entità esterna. Il vantaggio nel loro utilizzo sta nel fatto che sono delle API REST e quindi lo sviluppatore non è legato a nessun linguaggio di programmazione in particolare, ma necessita soltanto di una libreria o di un API per client http disponibile per il linguaggio di programmazione che vuole utilizzare. WebHDFS sono le API native di Hadoop, che supportano una completa interfaccia per HDFS e il cui servizio resta in ascolto sulla porta 50070/TCP dell'Head Node. HttpFS è un ulteriore implementazione delle WebHDFS API in ascolto sulla porta 14000/TCP dell'Head Node. Questa implementazione si comporta come un gateway tra il client HTTP e le WebHDFS, consentendo di ridurre il numero di indirizzi pubblici da assegnare ai vari Head Node e Slave Nodes del cluster.

Il terzo livello è quello delle **Querying Tools**, e racchiude tutte le operazioni che consento di recuperare i dati memorizzati sul cluster tramite l'uso di semplici tool. Per recuperare i dati memorizzati sul proprio cluster Cosmos viene utilizzato **Hive**, un sistema di deposito dati per Hadoop che ne semplifica il recupero. Hive carica tutti i dati in tabelle SQL-like, fornendo anche un linguaggio, chiamato *HiveQL* (molto simile a SQL), per querare facilmente tali dati. Tranne che per pochi tipi di query, ogni volta che viene eseguita una query, viene lanciato una *HiveMapReduce Job* predefinito, in base al tipo di operazione richiesta sui dati (select,filter, join, group). E' possibile sfruttare le potenzialità di Hive in locale (se si ha accesso all'Head Node), attraverso le *CLI Hive*, digitando *hive* sulla shell dell'Head Node. Da remoto, invece, è possibile fare delle queryHive implementando il proprio Hive client personalizzato in un qualche linguaggio di programmazione. Esistono diverse librerie (*JDBC*, *Thrift*) per svariati linguaggi di programmazione, che consentono facilmente di creare un *client Hive*. Per le operazioni da remoto un server Hive è costantemente attivo sul nodo master alla porta 1000/TCP.

Capitolo 1: Piattaforma Fi-ware e le sue componenti

Il quarto ed ultimo livello è quello del **MapReduce**, che comprende tutte le operazioni che consentono di eseguire analisi ed elaborazioni complesse sui dati archiviati. Gli HadoopMapReduce Jobs sono scritti in Java e impacchettati come file jar e possono essere eseguiti localmente digitando dalla shell dell'Head Node il comando:

```
hadoop jar <jar_file><main_class><existing_input_folder><non_existing_output_fold  
er>
```

La cartella di uscita verrà creata automaticamente alla fine dell'esecuzione del job e conterrà i risultati dell'esecuzione. Hadoop/Cosmos è già distribuito con una serie di applicazione MapReduce d'esempio (hadoop-examples.jar), che possono essere visualizzate tramite un opportuno comando:

```
$ hadoop jar /usr/lib/hadoop-0.20/hadoop-examples.jar
```

Oltre alle applicazioni MapReduce già disponibili, è possibile crearsi la propria applicazione seguendo una linea guida che consiste nella realizzazione di tre componenti:

- Un driver, per la definizione degli ingressi, delle uscite e dei formati dei dati.
- Un insieme di Mapper.
- Un insieme di Reducer.

Da remoto è invece possibile eseguire un'applicazione MapReduce sfruttando le *Oozie REST API*.

Capitolo 2

Hardware utilizzato: sistemi embedded e sensori

2.1 Introduzione

Come già anticipato nella sezione introduttiva, il nostro lavoro di tesi è consistito nello studio della piattaforma Fiware e nello sviluppo di un'applicazione per questa piattaforma. In maniera molto approssimativa possiamo dire che, l'applicazione che abbiamo realizzato è fondamentalmente un sistema di monitoraggio ambientale per la misurazione di temperatura, pressione e pm10, sia in ambienti indoor che outdoor. Le misurazioni effettuate vengono inviate ed archiviate sulla piattaforma Fiware, dove possono essere elaborate e rese fruibili agli utenti della piattaforma stessa. Per una descrizione più approfondita del sistema e per i dettagli tecnici dell'implementazione rimandiamo il lettore al Capitolo 3. Il sistema di monitoraggio implementato (inteso come dispositivi fisici che lo costituiscono), possiamo suddividerlo sia fisicamente che logicamente in due livelli: il livello di sensoristica ed il livello

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

di gateway. Il livello di sensoristica è chiaramente costituito dai dispositivi utilizzati per il rilevamento delle misurazioni, mentre il livello di gateway è costituito dalle boards (sistemi embedded) usate come ponte tra i sensori e la piattaforma Fiware.

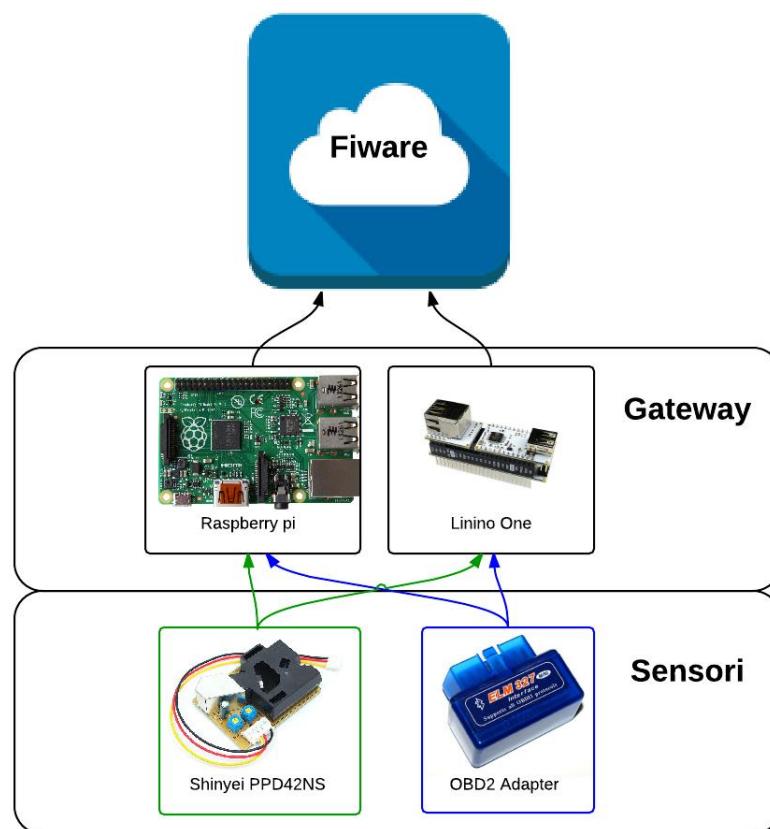


Figura 2.1: Livelli del sistema di monitoraggio

I dispositivi del livello di sensoristica adoperati in fase di realizzazione sono due:

1. Uno ShinyeiPPD42NS: sensore per la rilevazione di pm10

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

2. Un adattatore bluetooth OBD2: dispositivo che consente l’interfacciamento con la centralina dell’auto e da cui estraiamo i valori di temperatura e pressione.

Anche le boards del livello di gateway impiegate in fase di realizzazione sono due:

1. Raspberry pi B+
2. Linino One

In questo capitolo ci soffermeremo sulla descrizione e sui dettagli tecnici di queste componenti hardware, rimandando al Capitolo 3 la trattazione sull’interconnessione tra sensori e gateway e sull’implementazione software.

2.2 Raspberry Pi

La Raspberry Pi è un computer implementato interamente su una singola scheda elettronica, sviluppato dalla Raspberry Pi Foundation. Nasce principalmente dall’idea di stimolare e favorire l’insegnamento dell’informatica nelle scuole e si affaccia per la prima volta sul mercato nel febbraio 2012. In realtà, questo dispositivo è in grado di fare tutto ciò che ci si aspetta da un computer desktop come la navigazione internet, la riproduzione video, l’elaborazione dei testi e via dicendo, con in più però, la possibilità di poter interagire con il mondo esterno attraverso un interfaccia GPIO. Negli anni sono state apportate diverse modifiche hardware al modulo originario, portando alla commercializzazione di svariati modelli di Raspberry, tutti comunque basati sullo stesso SoC (System on Chip). L’ultimo modello della scheda, nonché quello da noi utilizzato in fase di progetto, è il modello B+ la cui revisione risale a luglio 2014.

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori



Figura 2.2: Raspberry pi modello B+

Ciò che occorre per il corretto utilizzo della scheda è:

- Una SD card sulla quale verrà caricata l'immagine del sistema operativo desiderato.
- Un cavo HDMI per la connessione della scheda ad un monitor o ad una TV.
- Tastiera e mouse per utilizzare la scheda come un vero e proprio desktop.
- Un cavetto micro USB a 5V per l'alimentazione.

2.2.1 Raspbian: installazione e configurazione

L'installazione del sistema operativo può avvenire in tre modi:

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

- Acquistando direttamente una SD card con NOOBS⁸ (New Out Of the Box Software) preinstallato dallo Swag Store Raspberry.
- Scaricare NOOBS dalla sezione download del sito ufficiale, copiare incollare il contenuto dell'archivio zip all'interno della SD card e dopo averla inserita nell'apposito slot, alimentare la scheda e procedere con l'installazione guidata.
- Scaricare direttamente dalla sezione download del sito ufficiale l'immagine del sistema operativo desiderata, caricare l'immagine del sistema all'interno della SD card tramite un tool per la scrittura di file immagine, inserire la SD nell'apposito slot e procedere con l'installazione.

Sul sito ufficiale sono disponibili diverse immagini di sistema basate su diverse distribuzioni Linux e non solo. Il sistema operativo raccomandato è però Raspbian, basato su Debian Wheezy e ottimizzato per l'hardware della Raspberry. È possibile modificare le configurazioni di base attraverso il tool *raspi-config*. Questo tool si apre automaticamente in fase di installazione del sistema, oppure è possibile aprirlo in un secondo momento tramite il comando:

```
sudo raspi-config
```

Si aprirà un'interfaccia semigrafica dalla quale sarà possibile espandere il file system, cambiare la password di un utente, impostare la lingua ed il layout della tastiera, overclockare la CPU della Raspberry e via dicendo. Le modifiche fatte con il tool di configurazione possono andare a modificare il file *config.txt*, che in qualche modo rappresenta il BIOS della Raspberry e che, infatti, viene letto dalla GPU prima che la CPU e il sistema operativo vengano

⁸ NOOBS è un semplice gestore di installazione del sistema operativo, tramite il quale è possibile scegliere in fase di installazione la distribuzione del sistema desiderata.

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

inizializzati. Il formato di questo file è molto semplice e riporta ad ogni riga uno statement del tipo *proprietà=valore*. I valori della proprietà possono essere di tipo numerico o di tipo stringa. Riportiamo di seguito un esempio di questo file [12].

```
# Force the monitor to HDMI mode so that sound will be sent over HDMI
cable
hdmi_drive=2
# Set monitor mode to DMT
hdmi_group=2
# Set monitor resolution to 1024x768 XGA 60Hz (HDMI_DMT_XGA_60)
hdmi_mode=16
# Make display smaller to stop text spilling off the screen
overscan_left=20
overscan_right=12
overscan_top=10
overscan_bottom=10
```

Questo file di configurazione può essere raggiunto a sistema operativo avviato tramite il percorso */boot/config.txt* e può essere modificato solo se si acquisiscono i permessi di root. Raspbian viene distribuito con una serie di software già preinstallati quali Scratch, Python, WordPress, XBMC e tanti altri, che consentono di sfruttare da subito le potenzialità della scheda.

2.2.2 Caratteristiche hardware del modello B+

Le principali migliorie apportate a tale modello rispetto ai precedenti riguardano l'aumento del numero di porte USB disponibili, l'aumento del numero di pin ingresso/uscita, una riprogettazione del circuito di alimentazione che ha portato ad una riduzione dell'assorbimento di potenza della scheda, la sostituzione dello slot per SD card con uno slot per microSD ed infine l'eliminazione del connettore *composite video* la cui funzionalità è stata

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

spostata sul jack audio/video da 3.5 mm. Di seguito una tabella con le caratteristiche hardware principali:

Caratteristiche	Model B+
BRCM2835 SoC	Yes
Standard SoC Speed	700Mhz
RAM	512MB
Storage	Micro SD
Ethernet 10/100	Yes
HDMI output port	Yes
Composite video output	On 3.5mm jack
Number of USB2.0 ports	4
Expansion header	40
Number of available GPIO	26
3.5mm audio jack	Audio/Video
Number of camera interface ports (CSI-2)	1
Number of LCD display interface ports (DSI)	1
Power (bare, approx, 5v)	650mA, 3W
Size	85 x 56 x 17mm

Il SoC usato dal Raspberry Project è il BCM2835 che comprende un processore ARM1176JZF-S a 700 MHz, una GPU VideoCore IV e 512MB di memoria RAM. La quantità di corrente necessaria per alimentare la Raspberry dipenderà da quante periferiche andremo ad utilizzare contemporaneamente. Ogni periferica assorbirà un certo quantitativo di corrente, per cui si dovrà fare attenzione e calcolare la quantità di corrente necessaria al corretto funzionamento, se disponiamo di un alimentatore limitato.

2.2.3 GPIO e la libreria python RPi.GPIO

La componente hardware che però ha contribuito principalmente alla diffusione di questa scheda è senza dubbio rappresentata dall'interfaccia GPIO

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

(Generale Purpose Input/Output pins). Questa può essere vista come una vera e propria interfaccia fisica tra la scheda ed il mondo esterno.

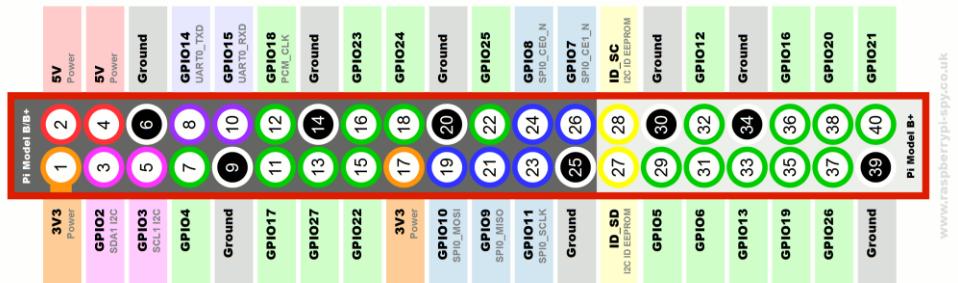


Figura 2.3: Schema grafico della GPIO della raspberry b+

Tutti i pin sono di tipo digitale e si possono dividere in 3 tipi:

- I pin di alimentazione che possono essere a 3,3Volt (1 e 17) o a 5Volt (2 e 4).
- I pin di massa (6, 9, 14, 20, 25, 30, 34 e 39).
- I pin di ingresso/uscita (pin GPIO).

I pin di ingresso/uscita sono i pin a cui è possibile collegare l'uscita dati di un sensore o l'ingresso di controllo accensione/spegnimento di un LED o qualsiasi altro dispositivo che sia in grado di ricevere o inviare segnali digitali. Non essendo equipaggiata di pin analogici, come le schede Arduino, per poter collegare un dispositivo analogico è necessario un convertitore Analogico/Digitale (ADC). Essendo la Raspberry dotata sia del modulo ethernet sia di quello wifi, è ovviamente possibile controllare i dispositivi collegati ai pin da remoto e inviare i dati generati sulla rete. Quando un pin GPIO viene usato come pin di output, ad esempio per accendere e spegnere un led, ciò che si deve fare è calcolare opportunamente il valore della resistenza

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

necessaria per proteggere il dispositivo da controllare, in quanto il ruolo dello switch e della batteria da 3.3V vengono sostituiti dalla GPIO della Raspberry.

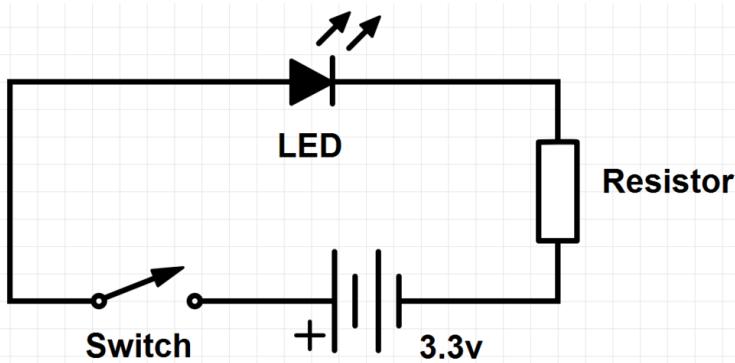


Figura 2.4: Semplice circuito per il controllo di un led

Quando, invece, si utilizza un pin GPIO come input, possono essere utilizzate delle resistenze interne alla Raspberry, chiamate di pull up e pull down, che consentono di fissare ad un valore (alto o basso) il livello logico d'ingresso del dispositivo collegato, in modo che la Raspberry possa capire quando inizia ad essere ricevuto un segnale di input. Chiaramente, si deve fare attenzione alle correnti e alle tensioni che si vanno ad applicare su un pin quando viene collegato a qualche dispositivo. Ad esempio si deve fare sempre attenzione che la tensione applicata ad ogni pin non superi i 3.3Volt, un voltaggio superiore potrebbe rendere inutilizzabile la scheda in maniera definitiva. Dalla figura 2.3 si può vedere inoltre che alcuni pin GPIO possono avere una doppia funzione. Ad esempio alcuni pin supportano il protocollo di comunicazione seriale a due vie I²C utilizzato tra circuiti integrati, altri pin supportano un tipo di protocollo bus chiamato SPI (Serial Peripheral Interface) per trasferimenti dati ad alta velocità, altri ancora consentono la generazione di segnali PWM (Pulse Width Modulation) utili per il controllo di alcuni servomotori, altri consentono il collegamento con il ricevitore-trasmettitore

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

UART per la conversione di bit da un formato parallelo ad uno seriale asincrono.

RPi.GPIO

L’interfaccia GPIO può essere gestita attraverso l’uso di linguaggi di programmazione quali Scratch o Python. La libreria python che consente la programmazione della GPIO è la RPi.GPIO. Facciamo un resoconto di quelle che sono le funzioni principali della libreria. Le funzioni di base sono:

- **GPIO.setup(canale, GPIO.IN/GPIO.OUT):** per impostare un pin GPIO come pin di input o come pin di output. Nel caso di pin di input è possibile impostare il valore delle resistenze di pull up e pull down tramite il passaggio di un terzo parametro (pull_up_down=GPIO.PUD_UP/PUD_DOWN).
- **GPIO.input(canale):** per leggere il valore da un pin di input.
- **GPIO.output(canale, stato):** per impostare lo stato del pin ad un livello logico alto o basso. Lo stato può essere 0/GPIO.LOW/False oppure 1/GPIO.HIGH/True.
- **GPIO.cleanup(canale):** per ripulire le impostazioni settate su un pin.

La libreria è fornita di funzioni che consentono di generare degli interrupt al processore o di rilevare un fronte di salita o di discesa di un segnale su un pin GPIO. Queste funzioni sono:

- **GPIO.wait_for_edge(canale, evento):** interrompe l’esecuzione del programma finché non si verifica un determinato evento sul canale.
- **GPIO.add_event_detect(canale, evento):** consente di rilevare il verificarsi di un determinato evento senza stoppare l’esecuzione del

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

programma. Si potrà poi controllare se l'evento si è verificato in un altro punto della procedura, richiamando la funzione `GPIO.event_detected(canale)`.

- **`GPIO.add_event_detect(canale, evento, callback=my_callback)`**: consente di rilevare il verificarsi di un determinato evento, e lanciare un thread parallelo al thread principale che consente l'esecuzione della callback⁹ al verificarsi dell'evento.
- **`GPIO.add_event_callback(canale, my_callback)`**: può essere utilizzata quando si vogliono associare più callback ad un evento su un canale. Le callback verranno in questo caso eseguite sequenzialmente, in quanto viene istanziato un solo thread per l'esecuzione delle callback.
- **`GPIO.remove_event_detect(canale)`**: rimuove la rilevazione degli eventi su un canale.

I canali sono chiaramente rappresentati dai pin GPIO, gli eventi possono invece essere di 3 tipi:

- **`GPIO.RISING`**: fronte di salita di un segnale
- **`GPIO.FALLING`**: fronte di discesa di un segnale
- **`GPIO.BOTH`**: fronte di salita o discesa di un segnale

All'inizio di ogni programma è generalmente buona norma specificare il tipo di numerazione da utilizzare per l'identificazione dei pin tramite la seguente funzione:

- **`GPIO.setmode(GPIO.BOARD/GPIO.BCM)`**

⁹ Con il termine callback si intende un blocco di codice o una funzione che viene generalmente passata come parametro ad un'altra funzione.

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

La tipologia BOARD numera i pin secondo la loro disposizione fisica, mentre la tipologia BCM si rifà alla numerazione dei canali usata dal SoC Broadcom2835.

2.3 Linino ONE

La Linino One è una scheda elettronica basata su architettura MIPS prodotta dalla compagnia Dog Hunter LLC nel 2014. La scheda fondamentalmente, è stata realizzata seguendo le specifiche hardware della Arduino Yun, inglobando al suo interno un microprocessore Atheros AR9331, un microcontrollore ATmega32u4 ed il modulo Wi-Fi. L'obiettivo è stato quello di creare una scheda con le stesse caratteristiche e proprietà della scheda Arduino, ma di dimensioni ridotte. Ciò è stato possibile eliminando sia la presa ethernet sia la presa USB e mantenendo soltanto la GPIO per l'interazione con il mondo esterno. L'aspirazione della compagnia è stata quella di creare una scheda che potesse essere facilmente utilizzata nell'ambito dei sistemi di controllo automatici e per la gestione sensoristica in ambito IoT [13].



Figura 2.5: Linino One

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

Ciò che occorre per il corretto utilizzo della scheda è:

- Un cavo micro USB per l'alimentazione
- Un terminale dotato di modulo Wi-Fi e un client ssh, o di un programma quale *minicom* per il controllo remoto della scheda.

Nel caso di necessità di una presa ethernet o di una presa USB o di entrambe, la scheda può essere estesa tramite delle shield apposite che vanno installate sui pin GPIO. Le shield sono:

- dogRJ45: connettore ethernet 10/100 Mbps conforme al protocollo IEEE 802.3.
- dogUSB: connettore USB conforme alle specifiche USB Hub 2.0 e retrocompatibile con le specifiche 1.1. Ingloba uno slot per microSD e supporta operazioni di lettura/scrittura in memoria esterna.

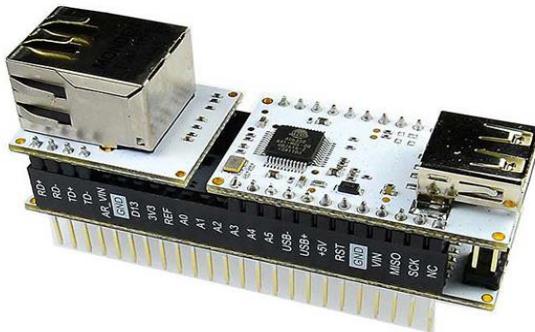


Figura 2.6: Linino One con le shield installate

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

Come si può vedere dalla figura 2.6 lo svantaggio di utilizzare le shield è che si vanno ad occupare quasi tutti i pin della GPIO.

2.3.1 Linino OS: installazione e configurazione

La scheda viene data in dotazione con il sistema operativo Linino OS, un sistema basato sulla distribuzione Linux OpenWRT. Linino OS integra inoltre il framework LininoIO, il quale consente di integrare le proprietà del microcontrollore all'interno dell'ambiente del microprocessore. Grazie a questo framework è possibile quindi interagire con la GPIO della scheda direttamente dal sistema operativo.

La prima cosa da fare una volta avviata la scheda ed aver guadagnato l'accesso come super utente, è aggiornare il sistema all'ultima versione disponibile attraverso i seguenti comandi:

```
$ cd /tmp  
$ wget  
http://download.linino.org/linino_distro/master/latest/openwrt-  
ar71xx-generic-linino-one-16M-squashfs-sysupgrade.bin  
$ sysupgrade -v openwrt-ar71xx-generic-linino-one-16M-  
squashfs-sysupgrade.bin
```

È possibile modificare i settaggi di base del sistema attraverso i files di configurazione uci (unified configuration interfaces), modificabili o tramite riga di comando al percorso */etc/config/* o tramite la Web GUI, raggiungibile digitando l'indirizzo ip della Linino sulla barra degli indirizzi del browser. I files di configurazione del firmware Linino sono dodici e sono:

- arduino
- dhcp

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

- dropbear
- firewall
- fstab
- luci
- network
- system
- ubootenv
- ucitrack
- uhttpd
- wireless

I settaggi su ognuno di questi file sono memorizzati come coppie “chiave=valore” e consentono di, configurare il demone ssh, modificare i settaggi per le interfacce di rete, configurare le regole del firewall e molto altro ancora. Linino OS viene distribuito con il pacchetto python già preinstallato.

2.3.2 Caratteristiche hardware

La scheda Linino One, così come la Arduino Yun, monta due processori. Un microcontrollore *ATmega32u4* tramite il quale si riproduce l’ambiente Arduino ed un processore *Atheros AR9331* tramite il quale si riesce a riprodurre un ambiente Linux. Come detto è inoltre dotato di un modulo Wi-Fi, di una GPIO con 20 pin ingresso/uscita, un oscillatore al cristallo a 16Mhz, e tre buttoni di reset (uno per il microcontrollore, uno per il microprocessore, uno per il modulo Wi-Fi). Di seguito riportiamo le caratteristiche principali del microcontrollore:

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

AVR Arduino Microcontroller	
Microcontroller	ATmega32u4
Clock Speed	16 MHz
SRAM	2.5 KB
EEPROM	1 KB
Flash Memory	32 KB
Input and Operating Voltage	5V
Digital I/O Pins	20
PWM Channels	7
Analog Input Channels	12

Le caratteristiche principali del microprocessore sono invece:

Linux Microprocessor	
Processor	Atheros AR9331
Architecture	MIPS 400MHz
RAM	64 MB DDR2
Flash Memory	16 MB
Operating Voltage	3.3V
Ethernet	IEEE 802.3
WiFi	IEEE 802.11b/g/n
USB Type-A	2.0 Host
Card Reader	MicroSD
PWM Channels	7
Analog Input Channels	12

La scheda come abbiamo detto deve essere alimentata con un cavo a micro USB a 5 Volts.

2.3.3 GPIO ed il linguaggio Arduino

Anche nel caso della Linino One, la componente più importante della scheda risulta essere l'interfaccia GPIO, alla quale è possibile collegare dispositivi quali sensori e attuatori sia analogici, che digitali. La GPIO è costituita da 20 pin digitali che possono essere usati sia come input, sia come

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

output; operano tutti a 5 Volts; possono fornire e ricevere un valore di corrente massimo di 40mA e sono tutti dotati di una resistenza interna di pull-up che può variare da 20 a 50 KOhms. Alcuni pin, così come nella GPIO della Raspberry Pi, possono svolgere una duplice funzione:

- (RX) – (TX): Ricevono e trasmettono dati in seriale attraverso l'ATmega32u4.
- (SDA) – (SCL): Supportano il protocollo di comunicazione I²C
- (PWM): consentono di fornire in uscita segnali PWM a 8 bit.
- Altri pin supportano il protocollo seriale SPI per trasferimenti dati ad alta velocità. Questi pin sono connessi anche ai pin del microprocessore, per cui la comunicazione tra microcontrollore e microprocessore può avvenire anche attraverso il protocollo SPI.

L'interazione con la GPIO può avvenire quindi in due modi:

1. Tramite il protocollo MCUIO, che consente al microprocessore di comunicare con il microcontrollore e vedere le periferiche di quest'ultimo come delle periferiche Linux standard. Tale protocollo consente quindi al processore Atheros di vedere la GPIO del microprocessore ATmega32u4 come se fosse, direttamente ad esso collegata. La numerazione di questa GPIO “virtualizzata” segue quella definita nel file *yun-gpios.cfg*, solo che l'identificazione del pin avviene sommando il numero 100 a quello che è il suo numero identificativo originale. Ad esempio se volessimo impostare il pin GPIO15 come pin digitale d'uscita dall'ambiente Linux, dovremmo usare le seguenti istruzioni:

```
echo 115 > /sys/class/gpio/export  
echo out > /sys/class/gpio/D13/direction  
echo 1 > /sys/class/gpio/D13/value
```

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

```
echo 0 > /sys/class/gpio/D13/value
```

È possibile consultare la documentazione completa sul *repository linino*¹⁰ su github.

2. Tramite il linguaggio di programmazione Arduino e le sue librerie.

L'editing di uno sketch Arduino viene eseguito tramite l'Arduino IDE. Questo mette a disposizione tutta una serie di funzionalità che consentono di: controllare gli errori nel codice, compilare il codice sviluppato e caricarlo sul microcontrollore, aprire un monitor seriale per ricevere/inviare dati alla board, importare nuove librerie da un file zip, e tante altre ancora [14].

Il linguaggio Arduino può essere diviso in tre sezioni principali che sono: le *strutture*, le *variabili* e le *funzioni*. Di seguito faremo una breve panoramica di quelle che sono le principali funzioni del linguaggio Arduino che consentono l'interazione con la GPIO:

- Funzioni Digital I/O:
 - **pinMode(pin,mode)**: consente di configurare un pin in modo che si comporti come pin di input o come pin di output.
 - **digitalWrite(pin,value)**: consente di settare il voltaggio del pin digitale ad un valore (High o Low). Nel caso il pin era stato settato come pin d'ingresso, il valore scritto (High o Low) abilita o disabilita il pull_up resistor.
 - **digitalRead(pin)**: consente di leggere il valore (High o Low) da uno specifico pin digitale.

¹⁰ Il repository linino è consultabile all'indirizzo https://github.com/linino/kernel_3.3.8/blob/master/Documentation/gpio.txt

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

- Funzioni Analog I/O:
 - **analogRead(pin)**: consente di leggere il valore da uno specifico pin analogico. Il valore di tensione analogico d'ingresso, viene convertito, tramite un convertitore analogico digitale a 10 bit, in un numero compreso tra 0 e 1023.
 - **analogWrite(pin,value)**: consente di scrivere un valore analogico (da 0 a 1023) su un pin.
- Funzioni Advanced I/O:
 - **pulseIn(pin,value)**: consente di calcolare la durata di un impulso (High o Low) in microsecondi. Ad esempio se il valore del parametro value è High, la funzione attenderà che il segnale d'uscita raggiunga il livello logico alto, farà partire un timer, e lo stopperà quando il segnale tornerà a livello logico basso.
- Funzioni Time:
 - **millis()**: ritorna il numero di millisecondi trascorsi da quando è stato caricato il programma corrente sulla scheda.
 - **delay(ms)**: mette in pausa l'esecuzione del programma per il numero di millisecondi specificati come parametro.

Oltre alle funzioni base del linguaggio, l'ambiente Arduino può essere esteso attraverso l'uso di librerie, che forniscono funzionalità aggiuntive per favorire le interazione con l'hardware, la manipolazione di dati ecc... Un certo numero di librerie vengono installate direttamente tramite l'IDE Arduino, ma è possibile anche scaricarne di nuove o crearsi la propria. Le librerie standard più importanti sono:

- Serial: per la comunicazione seriale tra la board Arduino ed un altro computer o dispositivo.
- EEPROM: per leggere e scrivere dalla memoria permanente del microprocessore.

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

- Ethernet: per la connessione a internet usando l'ethernet shield.
- SPI: per comunicare con dispositivi che usano il bus SPI.
- WiFi: per la connessione ad internet usando il modulo WiFi.

Per la board Arduino Yun, e quindi anche per la Linino One, è disponibile la libreria **Bridge**. Questa libreria semplifica la comunicazione tra il microprocessore Atheros AR9331 ed il microcontrollore ATmega32u4.

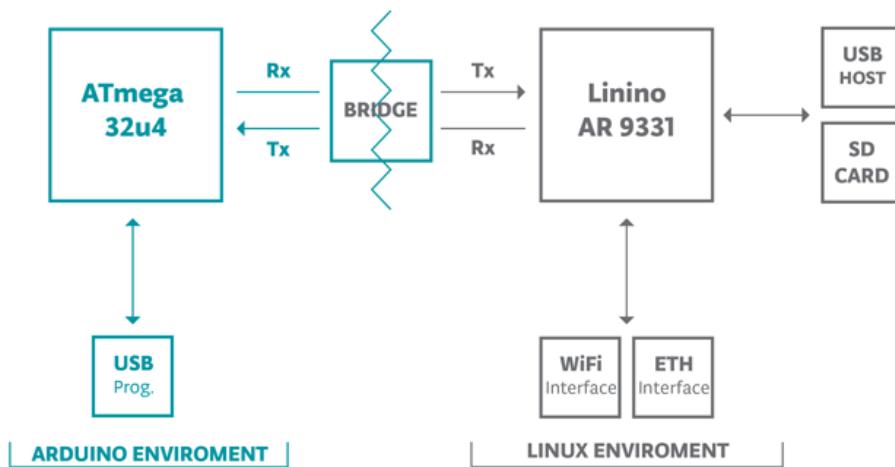


Figura 2.7: Schema ad alto livello della comunicazione tra i due processori

I comandi Bridge inviati dal microcontrollore sono interpretati tramite python sul microprocessore. La libreria Bridge è costituita da diverse classi, ognuna delle quali offre delle funzioni di interfacciamento. Troviamo ad esempio: la classe *Process* per l'esecuzione e gestione di processi sul processore Linux, la classe *Console* per la comunicazione con il monitor seriale dell'IDE attraverso una shell, la classe *HttpClient* per la creazione di un client HTTP sull'ambiente Linux, e tante altre. Un'altra importante e utile classe della libreria è la *FileIO* che fa da interfaccia per il file system dell'ambiente

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

Linux, consentendo la lettura e scrittura di file sulla microSD. Tra le funzioni di questa classe troviamo:

- **FileSystem.begin():** inizializza la SD card e la classe FileIO.
- **FileSystem.open(filename,mode):** apre un file sulla SD card.
- **FileSystem.exist(filename):** controlla se un file o una directory esistono sulla SD card.
- **File:** un oggetto file da manipolare tramite le funzioni della classe FileIO
 - **File.write(data):** scrive dati sul file.
 - **File.read():** legge un byte dal file.

2.4 Shinyei PPD42NS

Prima di trattare quelle che sono le caratteristiche tecniche ed il principio di funzionamento di questo sensore, facciamo una breve introduzione su quella che è la grandezza fisica che è in grado di misurare, ossia il particolato.

2.4.1 Il particolato

Molti studi sono stati fatti relativamente agli effetti delle polveri sottili sulla salute umana.

E' certo, ormai, che l'eccessiva presenza di polveri nell'aria può causare o contribuire all'insorgere di malattie respiratorie e cardiovascolari. Da anni i governi hanno deciso di regolamentare e definire dei limiti ben precisi sulle concentrazioni consentite delle polveri sottili nell'aria. Sono stati quindi definiti limiti per tutte le classi di dimensione delle particelle e dei range di tolleranza consentiti.

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

Le due più nocive classi di particelle individuate sono: PM10 e PM2.5. Rientrano in queste due categorie le particelle che hanno una massa totale di diametro inferiore rispettivamente a 10 micron e 2.5 micron. La figura in basso mostra per le classi PM2.5 e PM10 i valori stabiliti rispettivamente negli stati uniti dal WHO (World Health Organization) e dall'Environmental Protection Agency (EPA), mentre nel vecchio continente dalla Commissione Europea (EU). Se i valori definiti dal WHO sono delle semplici raccomandazioni, tutti gli altri valori devono essere considerati come soglie che non è possibile superare [19].

TABLE I
**MAXIMUM PERMISSIBLE VALUES FOR PARTICULATE MATTER AS
 SUGGESTED BY THE WHO [4], THE EU [1], AND EPA (USA) [5].**

	Class	Maximum permitted	Tolerated exceedings
WHO	PM _{2.5}	10 $\frac{\mu g}{m^3}$ (annual mean)	–
		25 $\frac{\mu g}{m^3}$ (24-hour mean)	–
	PM ₁₀	20 $\frac{\mu g}{m^3}$ (annual mean)	–
		50 $\frac{\mu g}{m^3}$ (24-hour mean)	–
EU	PM ₁₀	40 $\frac{\mu g}{m^3}$ (annual mean)	
		50 $\frac{\mu g}{m^3}$ (24-hour mean)	max. 35 days per year
USA	PM _{2.5}	15 $\frac{\mu g}{m^3}$ (annual mean)	
		35 $\frac{\mu g}{m^3}$ (24-hour mean)	
	PM ₁₀	150 $\frac{\mu g}{m^3}$ (24-hour mean)	max. 1 day in 3 years

Figura 2.8: Valori di particolato massimi ammissibili

Ciò dovrebbe far riflettere sul bisogno e sull'importanza che i sistemi di monitoraggio delle polveri sottili dovrebbero avere. Purtroppo, la sensoristica

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

necessaria ad effettuare le rilevazioni è molto costosa, ciò comporta il fatto che nell'ottica di limitare i costi, spesso viene usata una sola stazione di misurazione per aree urbane molto grandi e ciò incide negativamente sulla precisione delle misurazioni. Avremmo bisogno invece di sistemi di monitoraggio con una granularità molto più fine nella rilevazione. Tale sensibilità di rilevamento può essere raggiunta attraverso sistemi di monitoraggio distribuiti, in cui più sensori sono installati in modo omogeneo su tutta l'area da monitorare e collaborano tra di loro correlando i loro dati per migliorare la precisione della misurazione. Abbiamo già detto come i sensori per la rilevazione delle polveri sottili sono molto costosi, per cui lo scenario descritto in precedenza ci obbliga ad un cambio di paradigma. Invece di utilizzare un solo sensore tanto costoso quanto preciso è preferibile adottare una soluzione in cui tanti sensori, poco precisi e dai costi contenuti collaborano tra di loro fornendo dati capillari sulla presenza delle polveri in tutta l'area sotto controllo. E' questo l'approccio che abbiamo scelto

per implementare il nostro sistema di monitoraggio ambientale.

2.4.2 Principio di funzionamento dei sensori di particolato

Tutti i sensori a basso costo per la rilevazione delle polveri sottili sono di tipo ottico e quindi si basano sul medesimo principio di funzionamento. Un fascio di luce viene emesso da un led in una camera di misura. Quando la polvere è presente all'interno della camera di misurazione, la luce emessa dal led viene rifratta dalle particelle di polvere. Un recettore di luce (fotodiodo) ha il compito di rilevare la luce rifratta dalle particelle, producendo una tensione proporzionale alla concentrazione rilevata. Quasi tutti i sensori utilizzano in aggiunta una resistenza, che, al passaggio della corrente, si riscalda per effetto Joule e causa il sorgere di una corrente ascensionale (vedi figura in basso) che obbliga le particelle a passare lungo la direzione del fascio di luce, facilitando

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

così il rilevamento di tutte le particelle.

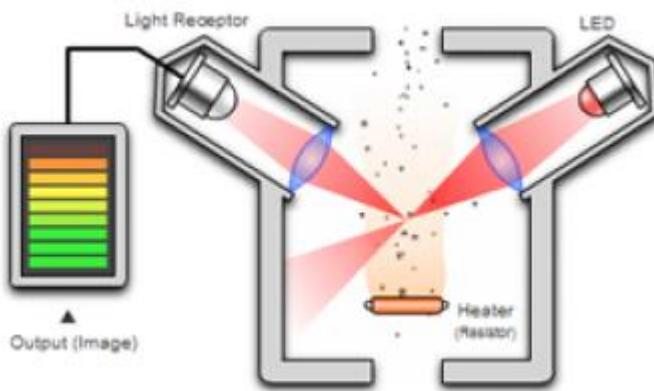


Figura 2.9: Principio di funzionamento dei sensori ottici di particolato

L’uso di tale resistenza causa alcuni inconvenienti: in primo luogo, a causa del fatto che è necessaria una corrente per riscaldare il resistore, il consumo energetico è generalmente superiore. In secondo luogo, il tempo di risposta è maggiore, in quanto si deve aspettare un certo tempo (solitamente trenta secondi) necessario a riscaldare la resistenza. Inoltre il riscaldamento impone un orientamento rigoroso del sensore durante il funzionamento rendendo più difficoltosa l’installazione del dispositivo. Infine il sensore non può essere direttamente ventilato, in quanto ciò inciderebbe sul riscaldamento della resistenza.

2.4.3 Shinyei

Lo Shinyei PPD42NS è un sensore ottico a basso costo per la misurazione del particolato (PM). In particolar modo, il sensore è in grado di rilevare la presenza di particelle di particolato che hanno una dimensione maggiore o uguale ad 1 micron. Come detto, lo Shiney è un sensore ottico e segue il principio di funzionamento che abbiamo presentato nel paragrafo precedente,

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

inoltre fa uso della resistenza di riscaldamento di cui abbiamo parlato sempre nel paragrafo precedente, ed è quindi soggetto a tutte le limitazione sopra menzionate. Vediamo adesso di studiare in dettaglio le varie parti che compongono il sensore.



Figura 2.10: Shinyei

La parte metallizzata che si vede nella figura soprastante è in pratica il canale attraverso il quale l'aria circostante passa dentro il sensore, tale canale va a finire all'interno dell'area di rilevazione, ossia l'area interna alla copertura in plastica nera, visibile sempre dalla figura sopra [17].

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

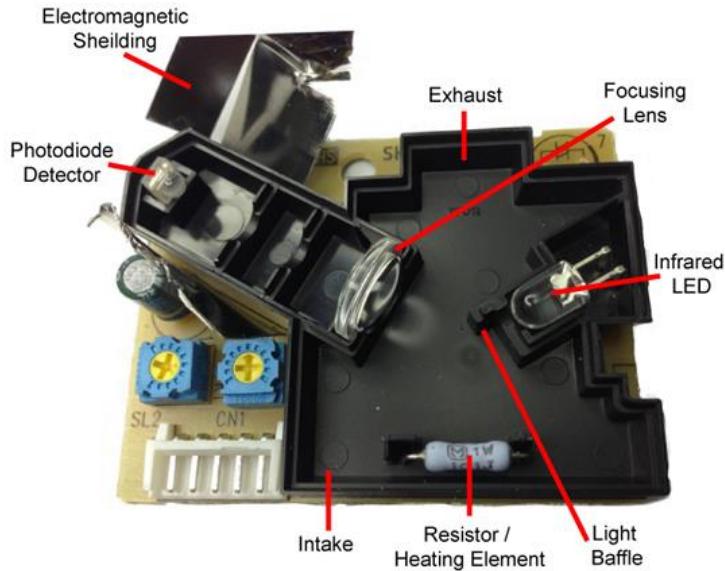


Figura 2.11: Componenti interni dello Shinyei

Internamente all'area di rilevazione è possibile vedere nella figura 2.11 tutte le componenti necessarie per l'individuazione delle particelle di particolato nell'aria raccolta. Le principali componenti del sensore sono :

- Un LED infrarosso necessario per l'immissione di luce all'interno dell'area di rilevazione.
- Un resistore di 100 Ohm necessario per generare la corrente ascensionale di cui abbiamo parlato nel paragrafo sul principio di funzionamento.
- Una lente necessaria per la raccolta dei raggi riflessi dalle particelle di particolato.
- Un fotodiodo necessario per la trasduzione dei segnali ottici rilevati in segnali elettrici digitali.

Come è possibile vedere sempre dalla figura 2.11, il sensore è dotato di 5 pin di connessione (numerati partendo dal pin più a destra):

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

- Il pin 1 che è il pin di massa
- Il pin 3 (input) che è il pin di alimentazione del sensore a 5Volts
- Il pin 4 (output) che è il pin dal quale viene fornito il segnale digitale di uscita del sensore relativo alla rilevazione di particelle di particolato con dimensione minima pari a 1 micron. Un livello logico basso del segnale d'uscita sta a significare che è stata rilevata la presenza di una particella di particolato. Un livello logico alto, viceversa, sta a significare l'assenza di particelle di particolato.
- Il pin 5 che è un pin di input attraverso il quale sarebbe possibile variare la soglia rappresentante la dimensione minima delle particelle rilevabili dal fotodiodo
- Il pin 2 che è il pin dal quale prelevare il segnale di uscita quando viene fornito un valore di soglia attraverso il pin 5.

I pin da noi utilizzati, nonché quelli usati abitualmente sono i pin 1 3 e 4. Tramite questo sensore ottico, quindi, si riesce a misurare il grado di concentrazione di particelle di particolato

all'interno dell'area di rilevazione, in un determinato intervallo di tempo. La concentrazione viene quantificata dal rapporto tra *pcs* (numero di pezzi, cioè di particelle rilevate) su *liter*. Liter è un'unità di misura volumetrica definita come: il volume occupato da un Kg di acqua alla temperatura di 4 gradi e pressione atmosferica standard pari a 760 millimetri. Meno rigorosamente possiamo dire che 1 liter equivale ad un decimetro cubico. L'uso dei *pcs* per quantificare la concentrazione ha lo svantaggio di fornire delle misure di natura qualitativa, non è possibile infatti trasformare i dati ad unità di misure quantitative come micro grammi su metro cubo.

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

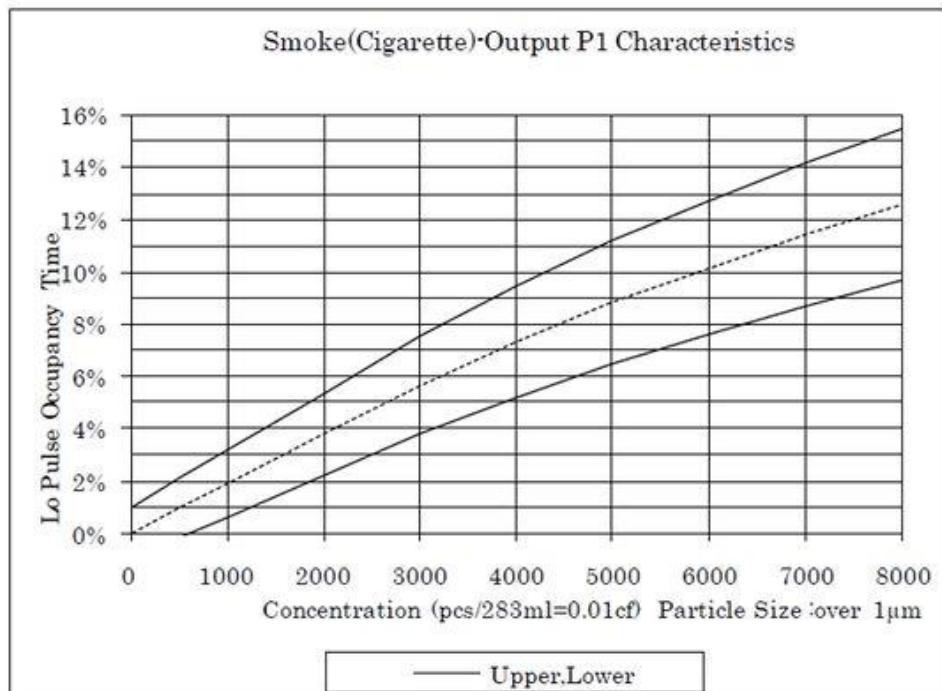


Figura 2.12: Curva caratteristica dello Shinyei per il calcolo della concentrazione

Per calcolare la concentrazione viene fornito, sul datasheet del sensore, un grafico di una curva rappresentante il legame tra la concentrazione (in ascissa) appunto e la LPO (LowPulseOccupancy in ordinata) [18]. La LPO può facilmente essere calcolata e rappresenta la percentuale di tempo in cui il segnale d'uscita del sensore si trova a livello logico basso (ossia rileva la presenza di particelle). L'intervallo di tempo che viene consigliato utilizzare per la rilevazione di una misurazione di concentrazione è di 30 secondi. Come detto nel paragrafo precedente questo vincolo è necessario per permettere alla resistenza di riscaldarsi adeguatamente.

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

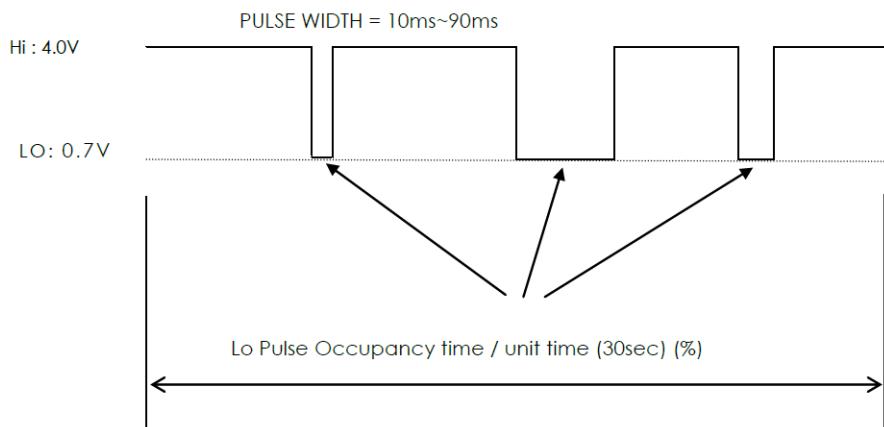


Figura 2.13: Low Pulse Occupancy

Seguendo quindi la curva in figura 2.13, una volta calcolata la LPO relativa ad un determinato intervallo di tempo, sarà facile individuare quella che sarà la rispettiva concentrazione di particelle nello stesso intervallo. Di seguito riportiamo una tabella relativa alle specifiche tecniche del sensore.

Model PPD42NS Model	PPD42NS
Detectable particle size	approx. 1µm (minimum.)
Detectable range of concentration	0~28,000 pcs/liter (0~8,000pcs/0.01 CF=283ml)
Supply Voltage	DC5V +/- 10% (CN1:Pin1=GND, Pin3=+5V) Ripple Voltage within 30mV
Operating Temperature Range	0~45°C
Operating Humidity Range	95%rh or less (without dew condensation)
Power consumption	90mA
Storage temperature	-30~60°C
Time for stabilization	1 minute after power turned on
Dimensions	59(W) × 45(H) × 22(D) [mm]
Weight	24g(approx.)
Output Method	Negative Logic, Digital output, Hi : over 4.0V(Rev.2) Lo : under 0.7V (As Input impedance : 200kΩ) OP-Amp

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

Detectable particle size	output, Pull-up resistor : 10kΩ approx. 1µm (minimum.)
Detectable range of concentration	0~28,000 pcs/liter (0~8,000pcs/0.01 CF=283ml)
Supply Voltage	DC5V +/- 10% (CN1:Pin1=GND, Pin3=+5V) Ripple Voltage within 30mV
Operating Temperature Range	0~45°C

2.5 OBD2 bluetooth adapter

Il termine OBD sta per On-Board Diagnostic ed è stato coniato nel 1980 per identificare il sistema che consente ad un veicolo di eseguire un auto diagnosi e di riportare eventuali segnalazioni di malfunzionamenti dei suoi sottosistemi. I primi computer di bordo che implementavano questo sistema, consentivano l'auto diagnosi grazie all'uso di una routine di diagnostica, che controllava ciclicamente tutte le componenti, verificandone il corretto funzionamento e accendendo una spia nel caso di guasto di qualche componente [20]. La prima versione dello standard OBD, detto OBD I, nasce in California nel 1991, obbligando tutte le case automobilistiche a montare questo sistema sui loro nuovi veicoli, affinché potessero monitorare tutte quelle componenti il cui guasto avrebbe potuto influenzare la quantità di emissioni inquinanti. Di fatto, OBD I non è mai stato uno standard, in quanto ogni produttore si poteva progettare il proprio connettore diagnostico di collegamento, poteva deciderne la posizione, definire i propri codici di diagnostica (DTC) e le proprie procedure per la lettura dei dati dal veicolo. Fu così che nel 1996 sempre in California si decise di creare uno standard, che prese poi il nome di OBD II, che consentì di rendere più o meno uniformi i sistemi di diagnostica montati su tutte le autovetture, rendendo il sistema più facile da utilizzare per l'utenza tecnica. Lo standard OBD II specifica il tipo di

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

connettore diagnostico, la sua piedinatura, la sua posizione sull'autovettura, i protocolli di segnalazione elettrica ed il formato dei dati.

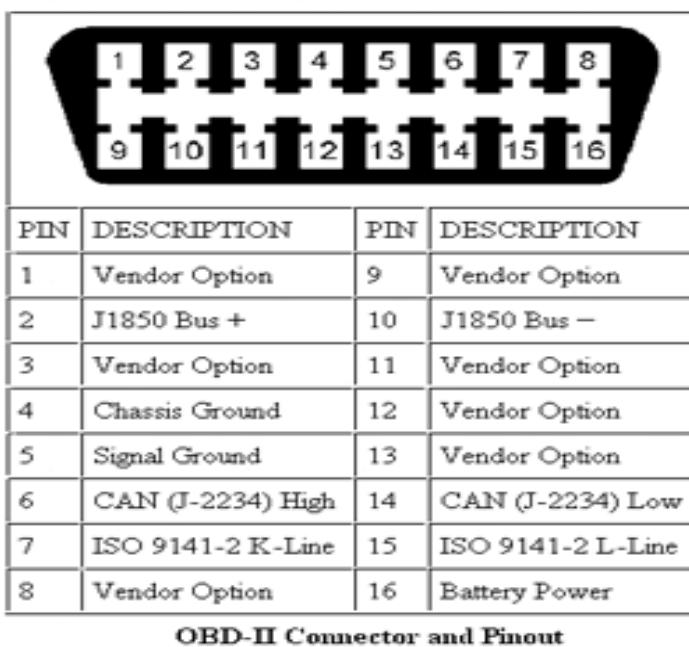


Figura 2.14: Pinout del connettore OBD II

Il perno 16 del connettore consente di alimentare, tramite la batteria dell'autovettura, il dispositivo di scansione collegato al connettore. Lo standard definisce anche una lista di quelli che sono i parametri della vettura da tenere sotto controllo e associa un codice ad ognuno di questi. Definisce, infine, anche un elenco estendibile di DTC (Diagnostic Trouble Code). Le normative OBD II arrivano ufficialmente in Europa nel 2001 per le auto con motore benzina e nel 2004 per le auto con motore diesel, sotto il nome di EOBD (European On Board Diagnostic). Quest'opera di standardizzazione ha consentito di poter interrogare i computer di bordo di tutti i veicoli attraverso l'uso di un unico dispositivo.

2.5.1 L'ELM327

L'interfaccia OBD II supporta diversi protocolli di comunicazione per il trasferimento dei dati, purtroppo però, nessuno di questi protocolli può essere usato direttamente tramite un pc o un qualsiasi altro dispositivo “intelligente”. L'ELM327 è un microcontrollore prodotto dalla ELM Electronics, e progettato appositamente per fungere da ponte tra la porta OBD del veicolo e l'interfaccia seriale standard RS232. Inoltre è in grado di rilevare automaticamente e interpretare nove differenti protocolli OBD, di supportare comunicazioni ad alta velocità, e dispone di una modalità di funzionamento a bassa potenza [21].

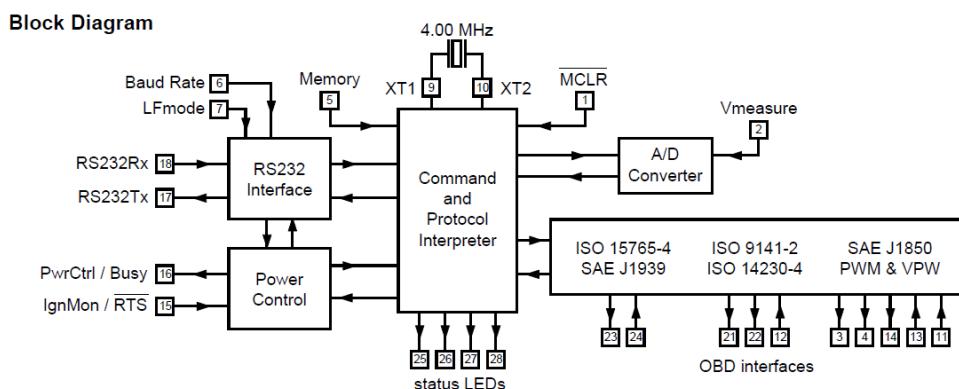


Figura 2.15: Diagramma a blocchi dell'ELM327

Come è possibile vedere dalla figura sopra, l'ELM327 può essere diviso in cinque macro blocchi che sono:

- L'interfaccia RS232: per il collegamento ad un PC.
- L'unità per il controllo della potenza: che consente il funzionamento in modalità “low power”.
- L'interprete dei comandi e dei protocolli: che consente il riconoscimento automatico del protocollo con cui è stato inviato un

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

messaggio, del tipo di messaggio e della decodifica del punto di destinazione.

- Un convertitore A/D.
- L'interfaccia OBD per inviare i comandi alla centralina del veicolo.

I comandi possono essere inviati all'ELM327 utilizzando una semplicissima console quale HyperTerminal o ZTerm. Tali comandi devono finire tutti con il carattere di a capo e possono essere fondamentalmente di due tipi:

- Comandi AT: sono comandi destinati direttamente all'ELM327 per un uso interno. Attraverso questi comandi è possibile configurare e personalizzare i parametri di funzionamento del microprocessore per modificarne il comportamento.
- Comandi OBD: sono comandi destinati alla centralina dell'auto. Attraverso questi comandi è possibile interrogare la centralina per ottenere dati in tempo reale, codici diagnostici pendenti ecc..

Comandi AT

I comandi AT sono riconoscibili dall'interprete ELM dal fatto che il codice del messaggio deve sempre essere preceduto dai caratteri AT. Utilizzando questi comandi è possibile abilitare o disabilitare il comando di echo (E0, E1), impostare il timeout della Baud Rate (BRT hh), impostare la modalità a bassa potenza (LP), ottenere una descrizione del protocollo corrente (DP), settare il protocollo da utilizzare (SP h), riportare tutti i parametri ai valori di default (D), resettare tutto (Z) ecc...

Comandi OBD

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

I comandi che non sono destinati per un uso interno dell'ELM327 sono comandi destinati al veicolo. Tali comandi possono contenere solo i codici ASCII relativi alle cifre usate per la notazione decimale (da 0 a 9 e da A a F). Generalmente, quando l'interprete ELM riconosce un comando OBD, lo riformatta aggiungendo 3 header byte e un byte per la checksum per formare il messaggio OBD da inviare al veicolo. La maggior parte dei comandi OBD sono costituiti da uno o due bytes, ma ci possono essere comandi che arrivano ad avere una lunghezza di sette bytes. Lo standard richiede che ogni comando OBD rispetti un determinato formato, che deve essere del tipo:

Mode (1 byte)	PID (1 o più byte)
----------------------	---------------------------

Il **Mode** rappresenta il primo byte inviato e descrive il tipo di dato che è stato richiesto. I bytes che seguono il Mode byte rappresentano il “parametro di identificazione” (**PID**) ed identificano la reale informazione richiesta. Lo standard SAE J1979 definisce dieci mode:

Mode (hex)	Description
01	Show current data
02	Show freeze frame data
03	Show stored Diagnostic Trouble Codes
04	Clear Diagnostic Trouble Codes and stored values
05	Test results, oxygen sensor monitoring
06	Test results, other component/system monitoring
07	Show pending Diagnostic Trouble Codes
08	Special Control Operation
09	Request vehicle information
0A	Request Permanent Trouble Codes

Utilizzando il *mode 01* come primo byte del comando seguito da un determinato PID, è possibile interrogare la centralina per conoscere i dati generati da una certa sottocomponente del veicolo in quel determinato istante.

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

Si può richiedere ad esempio, la pressione del carburante, i giri del motore, la velocità del veicolo, la temperatura del liquido di raffreddamento del motore, la pressione atmosferica, la temperatura esterna, la temperatura dell'olio, e tante altre informazioni. Non tutti i veicoli supportano tutti i codici PID definiti dallo standard, per cui il PID 00 è generalmente usato per interrogare l'ELM relativamente a quelli che sono i PID supportati dal veicolo. Riportiamo di seguito qualche codice PID usabile con il mode 01.

Mode 01	
PID (hex)	Description
03	Fuel system status
05	Engine coolant temperature
0A	Fuel pressure
0C	Engine RPM
0D	Vehicle speed
33	Barometric pressure
46	Ambient air temperature
51	Fuel type
5C	Engine oil temperature

Una volta che l'ELM ha opportunamente formattato ed inviato il comando alla centralina dell'auto, resta in ascolto sul bus per una risposta. La risposta visualizzata sulla console è anch'essa codificata in esadecimale ed in genere contiene, un primo byte che indica, seguendo una determinata codifica (mode + 40), qual'era il mode della richiesta, un secondo byte che riporta il PID richiesto, infine i restanti byte rappresentano la codifica della risposta.

2.5.2 L'adattatore OBD II

L'adattatore OBD utilizzato nel nostro progetto è un dispositivo dotato appunto di uno spinotto OBD II, un interprete OBD che è un ELM32x

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

compatibile con (320,322,323,327), ed un’interfaccia bluetooth per l’interfacciamento con un pc, un laptop od uno smartphone [22].



Figura 2.16: Adattatore bluetooth OBD II

Supporta tutti i protocolli OBD II:

- SAE J1850 PWM(41.6Kbaud)
- SAE J1850 VPW(10.4Kbaud)
- ISO9141-2(5 baud init,10.4Kbaud)
- ISO14230-4 KWP(5 baud init,10.4 Kbaud)
- ISO14230-4 KWP(fast init,10.4 Kbaud)
- ISO15765-4 CAN(11bit ID,500 Kbaud)
- ISO15765-4 CAN(29bit ID,500 Kbaud)
- ISO15765-4 CAN(11bit ID,250 Kbaud)
- ISO15765-4 CAN(29bit ID,250 Kbaud)
- SAE J1939 CAN(29bit ID,250*Kbaud)

Capitolo 2: Hardware utilizzato: sistemi embedded e sensori

- USER1 CAN(11*bit ID,125*Kbaud)
- USER2 CAN(11*bit ID,50*kbaud)

Funziona con tutti i tipi di veicoli OBD II (USA) e EOBD (Europa). È inoltre compatibile con diversi software tra cui:

- Scantool_net113win
- EasyOBD-II V2.2
- OBD2Spy
- ScanMaster-ELM
- E tanti altri

Esistono diverse librerie per svariati linguaggi di programmazione che mettono a disposizione metodi, classi e funzioni per l'invio di comandi OBD e l'interpretazione della risposta. Quella usata da noi per l'interazione con l'adattatore è stata la libreria python *obd0.1.0*, di cui parleremo meglio nel Capitolo 3.

Capitolo 3

Descrizione del progetto ed implementazione

3.1 Introduzione

Finalmente, dopo aver introdotto la piattaforma Fiware ed i suoi principi di funzionamento, e aver discusso quelle che sono le principali peculiarità e qualità delle componenti hardware utilizzate, iniziamo a discutere del progetto realizzato. Riporteremo inizialmente una panoramica generale, spiegando quelli che sono stati gli obiettivi del lavoro ed illustrando ad alto livello i meccanismi di interazione tra le varie componenti che prendono parte al sistema. Andremo poi ad analizzare più nel dettaglio la parte del lavoro relativa al sistema di monitoraggio di basso livello, discutendo l’interfacciamento tra i sensori e le boards a disposizione (Raspberry e Linino), sia a livello hardware che al livello software. Andremo infine ad approfondire una parte della logica di comunicazione, che consente l’interfacciamento tra il sistema di monitoraggio di basso livello e la piattaforma Fiware. Altre parti relative alla

Capitolo 3: Descrizione del progetto ed implementazione

logica di comunicazione e la parte relativa all’interfaccia utente del sistema verrà trattata solo nella panoramica generale, in quanto verrà esaminata nel dettaglio dal collega Antonio Caristia nella sua tesi “Backend ed interfaccia utente per il monitoraggio ambientale in ambiente Fiware”.

3.2 Panoramica generale del progetto

L’obiettivo del nostro lavoro di tesi, come già accennato nei capitoli precedenti, è stato quello di realizzare un sistema di monitoraggio ambientale attraverso l’uso di sensori a basso costo e sistemi embedded con cui poterli interfacciare, e facendo uso ovviamente delle tecnologie hardware e software messe a disposizione dalla piattaforma Fiware. L’idea nasce all’interno dell’azienda Arkimede s.r.l. presso la quale ci troviamo da un paio di mesi nel ruolo di stagisti, e con la quale abbiamo intrapreso l’avventura di presentare un progetto valido per le open call di Fiware. Arkimede s.r.l. gestisce già il sistema informatico dell’unica cooperativa di taxi presente sul territorio messinese che è Radio Taxi Jolly e la cui flotta supera le cinquanta unità. Su ogni taxi è già installato un tablet tramite il quale ogni tassista può, attraverso un’applicazione sviluppata da Arkimede s.r.l., comunicare agevolmente con il centralino e con ognuno degli altri tassisti appartenenti alla flotta, controllare quali taxi sono impegnati in una corsa e quali sono in attesa di clienti in una determinata area di sosta, comunicare facilmente con i clienti per eventuali segnalazioni di ritardo o di arrivo in zona e diverse altre operazioni che mirano a migliorare il servizio offerto ed aumentare allo stesso tempo i profitti della cooperativa. Partendo da una base di questo tipo, ossia la disposizione di una flotta di auto informatizzata che si sposta quotidianamente sul territorio cittadino, si è pensato di poter equipaggiare le auto con dei sensori a basso

Capitolo 3: Descrizione del progetto ed implementazione

costo (Shinyei PPD42NS) per la rilevazione delle concentrazioni di particolato, misurazioni che potrebbero chiaramente risultare di grande interesse per tenere sotto controllo il livello di inquinamento cittadino. La scelta di adottare sensori di questo tipo scaturisce da aspetti sia economici che pratici. Gli aspetti pratici sono legati alle dimensioni ridotte di questi sensori, che favorirebbero una più facile installazione su una vettura. Gli aspetti economici sono invece legati al fatto che, ogni vettura della flotta dovrebbe essere dotata di un sensore, motivo per cui la scelta è ricaduta su sensori economici. Logicamente, la bontà delle misurazioni di pm10 eseguite con lo Shinyei, non potranno mai essere paragonate alle misurazioni eseguite con strumenti che costano due o tre ordini di grandezza in più, ma la letteratura ci insegna che la scarsa precisione di una misurazione può essere compensata dalla quantità di misurazioni effettuate e da processi di post-elaborazione sui dati, tramite i quali è possibile fare in modo che i risultati delle elaborazioni si avvicinino in maniera significativa a quelli che sono i valori misurati con sensori ad alta precisione [15]. Ad ogni modo, lo studio della bontà delle misurazioni non è stato oggetto del nostro lavoro, il cui fine è stato prettamente quello di implementare un sistema di monitoraggio che prescinde dai tipi di sensori utilizzati. Oltre alla misurazione di pm10 si è pensato di monitorare anche la temperatura e la pressione atmosferica, sfruttando i sensori di cui le auto moderne sono già equipaggiate. Si è pensato di prelevare questi dati, utilizzando un adattatore bluetooth OBDII, tramite il quale riusciamo ad interrogare direttamente la centralina delle auto. Anche in questo caso la scelta può essere discussa sia sotto un aspetto economico, in quanto, un dispositivo di questo tipo ha un valore di mercato di una decina di euro, sia sotto un aspetto pratico, in quanto, facile da installare. Inoltre a fronte di una minima spesa, riusciamo ad usufruire di sensori per la misurazione di temperatura e pressione con un grado di precisione senz'altro maggiore rispetto ai sensori di cui avremmo potuto disporre spendendo la stessa cifra. Anche le boards utilizzate per l'interfacciamento dei sensori sono state scelte seguendo

Capitolo 3: Descrizione del progetto ed implementazione

la stessa logica. Difatti, entrambe le boards sono di dimensioni abbastanza ridotte in modo da poter favorire una facile installazione sull'autovettura, sono abbastanza economiche ed offrono un ambiente Linux attraverso il quale è possibile, sia gestire i sensori ad esse collegati, sia sviluppare software usando i linguaggi di programmazione più comuni.

Il sistema realizzato garantisce la caratteristica della scalabilità. Ciò grazie al fatto di aver utilizzato tecnologie quali HDFS, per la memorizzazione dei dati su un ambienti distribuiti, sia grazie alla possibilità di poter federare tra di loro più istanze del ContextBroker GE in caso di crescita del bacino di utenza del sistema. Inoltre, grazie all'uso dell'interfaccia NGSI e di diversi servizi basati sul paradigma RESTful, il sistema si è reso facilmente interfacciabile con applicazioni e servizi di terze parti.

Per fornire una visione generale del progetto, potremmo virtualmente dividerlo su tre livelli:

Capitolo 3: Descrizione del progetto ed implementazione

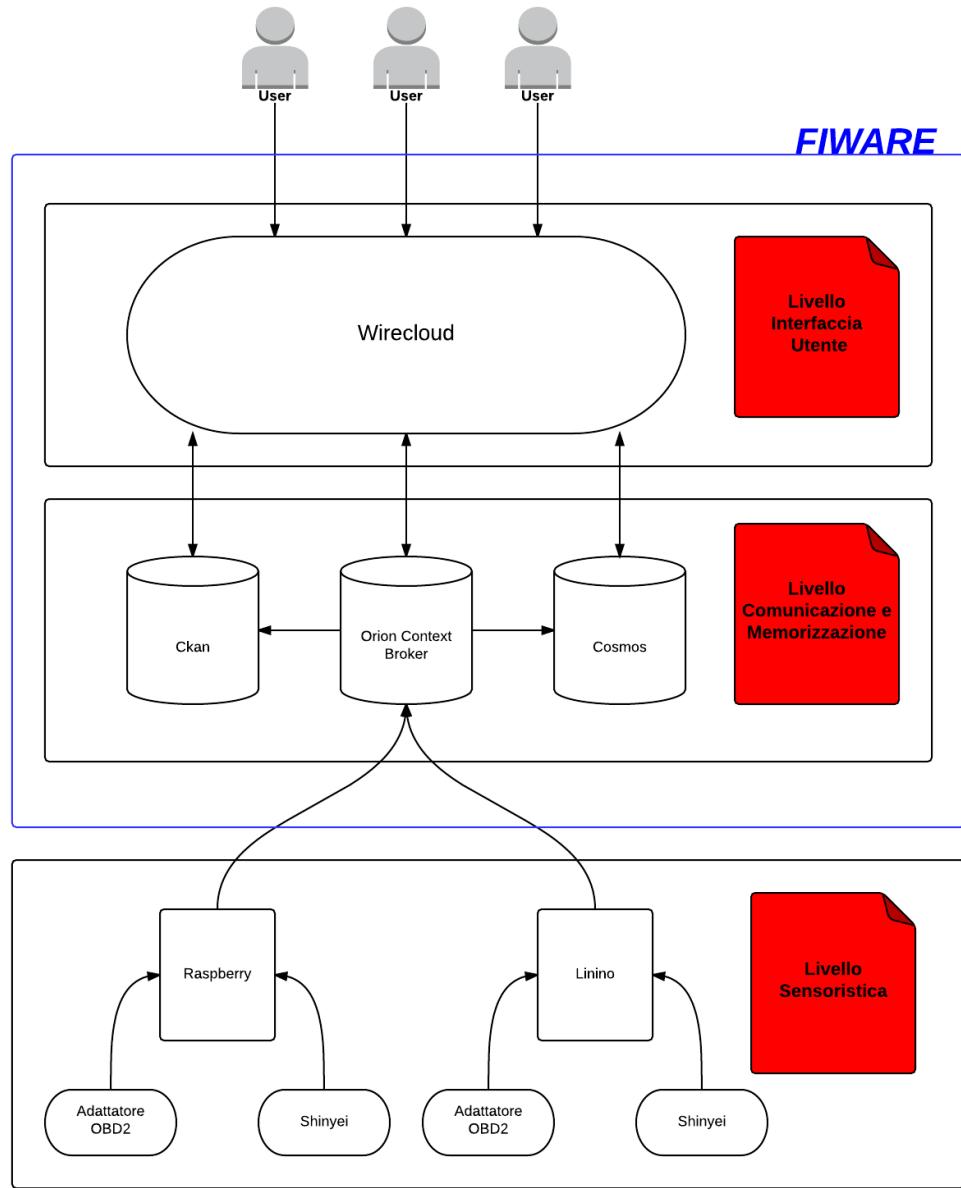


Figura 3.1: schema a livelli generale del progetto

1. Il livello di sensoristica: adoperando una vista ad alto livello potremmo dire che questo rappresenta il livello fisico del nostro sistema. A questo livello partecipano tutti i dispositivi di cui abbiamo discusso nel capitolo 2. Attraverso lo Shinyei e l'adattatore OBDII

Capitolo 3: Descrizione del progetto ed implementazione

rileviamo i dati di temperatura, pressione e pm10 (livello sensori), e attraverso la Raspberry pi o la Linino One (livello gateway) archiviamo i dati raccolti localmente e li inviamo alla logica di comunicazione.

2. Il livello di comunicazione e memorizzazione: il livello di comunicazione e memorizzazione è stato implementato sfruttando gli strumenti della piattaforma Fiware. Partecipano a questo livello una nostra istanza dell'Orion Context Broker che funge da centro di smistamento e memorizzazione temporanea dei dati, il nostro cluster Cosmos sul quale vengono memorizzati i dati in maniera permanente, ed il dataset in formato open data che carichiamo sul portale di CKAN.

3. Il livello di interfaccia utente: anche il livello di interfaccia utente è stato implementato attraverso l'uso di uno strumento messo a disposizione dalla piattaforma che è Wirecloud. Wirecloud rappresenta l'implementazione di riferimento dell'Application Mashup GE. Attraverso questo strumento abbiamo realizzato il nostro mashup, ossia un insieme di widget collegati tra loro che consentono agli utenti di visualizzare e graficare le informazioni d'interesse.

Andiamo a questo punto ad analizzare più dettagliatamente le varie componenti hardware e software che prendono parte ad ognuno dei tre livelli, in modo da dare una visione più esaustiva della logica di funzionamento dell'intero sistema. Iniziamo la descrizione dal livello più basso, ossia quello di sensoristica.

3.2.1 Livello di sensoristica

Abbiamo già detto precedentemente quali sono le componenti hardware che compongono questo livello, introduciamo ora le componenti software sviluppate per la lettura dei dati dai sensori e l'invio di tali dati al livello

Capitolo 3: Descrizione del progetto ed implementazione

superiore. I moduli software realizzati sono fondamentalmente degli script python che vengono fatti girare all'interno delle boards (Raspberry e Linino). Grazie al fatto che python è un linguaggio di programmazione che fa uso di un interprete dei comandi, e che quindi favorisce la portabilità del codice su diverse piattaforme hardware, siamo riusciti a riusare la maggior parte degli script realizzati su entrambe le board, fatta eccezione per quelli che interagiscono con le GPIO. Elenchiamo di seguito gli script realizzati con delle brevi descrizioni relative alla loro mansione:

Per la Raspberry:

- **Shinyei.py:** script python che interagisce con la GPIO della raspberry per la lettura dei valori di pm10 rilevati dal sensore ShinyeiPPD42NS.

Per la Linino:

- **Shinyei.ino:** sketch arduino che interagisce con la GPIO della Linino per la lettura dei valori di pm10 rilevati dal sensore ShinyeiPPD42NS

Script comuni ad entrambe le boards:

- **Obd.py:** script che consente l'interfacciamento con l'adattatore OBDII per la lettura dei dati di temperatura e pressione dalla centralina dell'auto.
- **Taxi_emulator.py:** script che consente di emulare il movimento di un taxi, simulando i tempi di sosta, i tempi di spostamento ed il cambio delle coordinate GPS.
- **Send_measures.py:** script che si occupa di inviare i dati generati dai sensori al livello di comunicazione e memorizzazione.
- **Server_rasp.py:** script che si occupa di lanciare gli script per la raccolta dei dati (Shinyei.py/Shinyei.ino, Obd.py) e lo script per l'invio

Capitolo 3: Descrizione del progetto ed implementazione

delle misure (`Send_measures.py`), ogni volta che il taxi risulta in sosta, e di killare gli stessi script quando invece è in movimento.

- **CreateEntity.py**: script che consente di creare un attributo di un'entità su un istanza Orion.
- **SetSubscription.py**: script che consente di sottoscrivere un attributo di un'entità su un istanza Orion.
- **UpdateEntityAttribute.py**: script che consente di aggiornare il valore di un attributo di un entità su un istanza Orion.

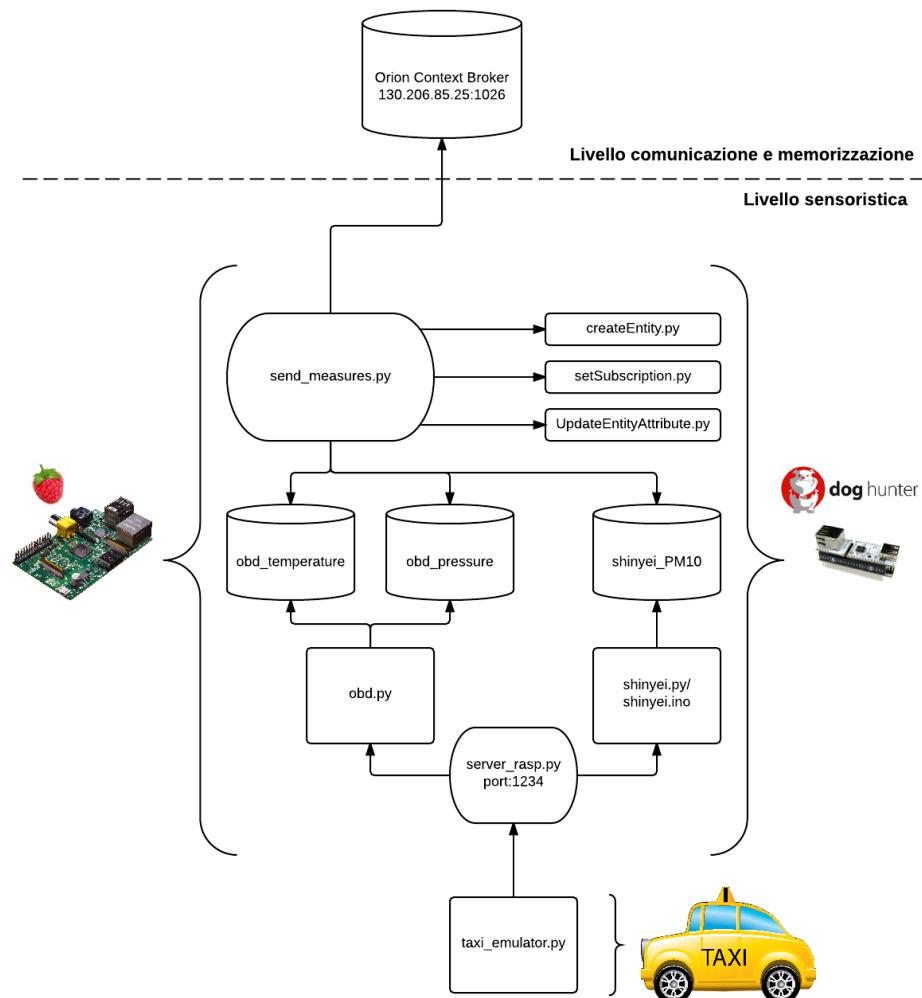


Figura 3.2: Schema livello sensoristica

Capitolo 3: Descrizione del progetto ed implementazione

Proviamo a questo punto, aiutandoci con la figura 3.2, a seguire il flusso delle operazioni a partire dall'inizio di una nuova misurazione fino ad arrivare al passaggio dei dati al livello di comunicazione e memorizzazione. Il *server_rasp.py* resta in ascolto sulla porta 1234, in attesa di ricevere un comando dal *taxi_emulator.py* che può essere o di chiusura dell'osservazione corrente (il taxi si è messo in movimento) o di inizio di una nuova osservazione (il taxi si è fermato). Nel primo caso semplicemente vengono killati gli script per la rilevazione delle misure e per l'invio dei dati (*obd.py*, *shinyei.py*, *send_measures.py*), nel secondo caso, invece, vengono inviate le coordinate GPS correnti del taxi e mandati in esecuzione gli stessi script. Il motivo per cui le misurazioni vengono raccolte soltanto durante la fase di sosta del taxi scaturisce da problemi di sensibilità del sensore Shinyei al movimento, che porterebbero ad una invalidazione dei risultati ottenuti. Ciò che avviene durante la fase di sosta del taxi è che, gli script *obd.py* e *shinyei.py/.ino* misurano periodicamente, nella fattispecie ogni 30 secondi, i valori di temperatura, pressione e pm10, andando ad aggiornare rispettivamente i file *obd_temperature*, *obd_pressure* e *shinyei_PM10* con i nuovi valori. Simultaneamente, lo script *send_measures.py*, una volta create le entità di temperatura, pressione e pm10 relative all'osservazione corrente sulla nostra istanza Orion e fatte le sottoscrizioni per tutte e tre le entità verso il nostro *subscriptionServer* (componente software che appartiene al secondo livello e che analizzeremo nel prossimo paragrafo), non farà altro che prelevare dai file suddetti, ogni 30 secondi, gli ultimi valori misurati e quindi aggiornare le entità su Orion.

3.2.2 Livello di comunicazione e memorizzazione

Il livello di comunicazione e memorizzazione possiamo vederlo come lo strato di backend della nostra applicazione e, come detto nell'introduzione del paragrafo 3.2, è costituito da tre componenti principali:

- **L'Orion Context Broker:** è la componente che fa da punto di collegamento con il livello di sensoristica, ed è in esecuzione su una nostra macchina virtuale deployata tramite il portale Cloud del Filab sul nodo spagnolo di Fiware. In qualche modo, potremmo definirlo il modulo software fulcro del nostro sistema, in quanto, si occupa di ricevere le informazioni sulle osservazioni effettuate, e di smistarle sia alle applicazioni che ne fanno richiesta in modo asincrono, sia a quelle che ne hanno fatto richiesta mediante sottoscrizione.
- **Il cluster Cosmos:** è la componente tramite la quale riusciamo a salvare le informazioni sulle osservazioni effettuate in maniera permanente, in quanto ci mette a disposizione una certa quantità di spazio di archiviazione su un file system distribuito (HDFS). Oltre al salvataggio dei dati in formato file di testo sul nostro spazio utente riservato dell'HDFS, questa componente ci mette a disposizione vari tool, tra cui Hive, il quale ci consente la creazione di tabelle SQL-like da associare ad uno o più file di testo contenenti le nostre informazioni e che possiamo facilmente interrogare, anche da remoto, tramite il linguaggio di interrogazione HiveQL.
- **CKAN:** è il modulo software che ci consente di pubblicare in formato opendata, i dati grezzi relativi alle osservazioni effettuate. La pubblicazione dei dati viene effettuata all'interno di un nostro dataset precedentemente creato attraverso l'interfaccia utente del modulo sul Filab.

Capitolo 3: Descrizione del progetto ed implementazione

Appartengono a questo livello altre due componenti software da noi sviluppate. Tali componenti sono sostanzialmente due script python che svolgono un ruolo di server RESTful, e si trovano costantemente in esecuzione sulla stessa macchina virtuale ospitante il nostro Orion Context Broker. Gli script in questione sono:

- Il **subscriptionServer.py**: questo script riceve da Orion le notifiche di aggiornamento sulle osservazioni effettuate. Per ogni nuova osservazione, crea il rispettivo file di testo sul nostro cluster Cosmos e la rispettiva tabella Hive sul master node di Cosmos. Per ogni notifica di aggiornamento ricevuta, aggiorna il rispettivo file di testo sull'HDFS, ed il rispettivo file in formato *csv* da inviare a CKAN.
- Il **restServerHive.py**: questo script resta in attesa di comandi dal livello interfaccia utente. Momentaneamente, l'unico comando (tramite chiamata HTTP) a cui risponde è quello di prelevamento dei dati grezzi relativi ad una determinata osservazione sul nostro cluster Cosmos.

Capitolo 3: Descrizione del progetto ed implementazione

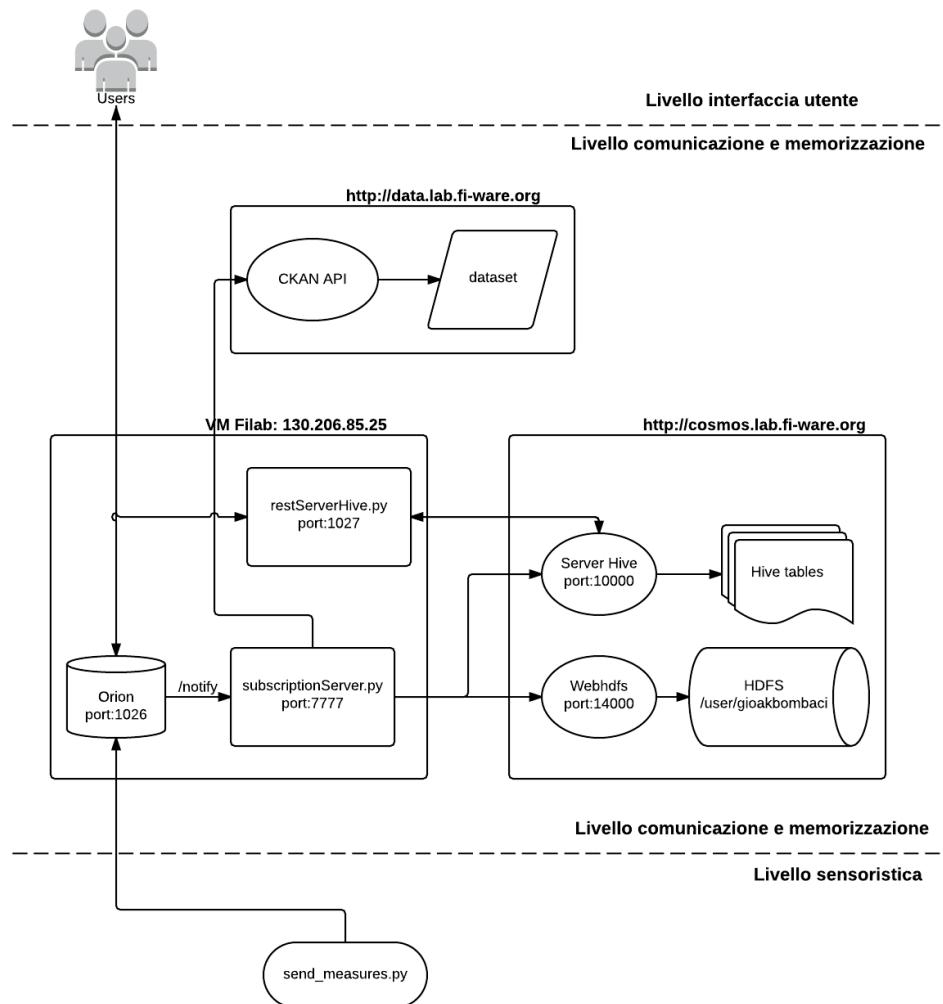


Figura 3.3: Schema livello comunicazione e memorizzazione

Proviamo anche per il secondo livello, aiutandoci con la figura 3.3, a seguire il flusso delle operazioni dall'inizio di una nuova osservazione fino alla sua chiusura. L'Orion all'indirizzo 130.206.85.25 in ascolto sulla porta 1026, riceve in sequenza sull'interfaccia NGSI-10: un comando di creazione dell'entità osservazione e un comando di sottoscrizione della stessa entità al *subscriptionServer.py*. Successivamente per tutta la durata dell'osservazione riceverà (sempre sull'interfaccia NGSI-10) ogni 30 secondi, il valore di misura

Capitolo 3: Descrizione del progetto ed implementazione

aggiornato, che grazie alla sottoscrizione inoltrerà di volta in volta al *subscriptionServer.py*. Come abbiamo già spiegato nel Capitolo 1, Orion non consente di salvare lo storico dei dati, memorizzando solo l'ultimo valore ricevuto. Il *subscriptionServer.py*, in ascolto anch'esso sulla macchina 130.206.85.25 sulla porta 7777, all'arrivo di una nuova notifica con un nuovo id di sottoscrizione, genera localmente il file TXT ed il file CSV per la nuova osservazione, da inviare rispettivamente a Cosmos e a CKAN. Per ogni notifica di aggiornamento dell'osservazione, invia i nuovi dati sul nostro spazio utente Cosmos tramite le *Webhdfs* API, e su CKAN tramite le CKAN REST API. L'interfaccia *Webhdfs* è in ascolto sul master node di Cosmos all'indirizzo <http://cosmos.lab.fi-ware.org> sulla porta 14000.

3.2.3 Livello di interfaccia utente

Il livello di interfaccia utente rappresenta il front-end del nostro sistema, ed è stato implementato interamente in javascript sfruttando uno degli strumenti messi a disposizione dalla piattaforma. In special modo, lo sviluppo del front-end è consistito nella realizzazione di un mashup di widget, attraverso Wirecloud. Wirecloud è uno strumento che consente di creare dei piccoli moduli software che si distinguono in *widget* e *operator*. Entrambi i due tipi di moduli devono essere sviluppati con l'ottica della riusabilità, dispongono di un certo numero di ingressi e di uscite dai quali ricevono ed inviano dati, e tramite i quali si possono collegare tra loro a formare un applicazione (mashup). L'unica differenza sta nel fatto che i *widget* dispongono di una vista utente (definiscono un file HTML che fa da punto d'ingresso per il codice javascript), mentre gli *operator* non possiedono un front-end per gli utenti, in quanto svolgono in genere operazioni di servizio per agevolare il collegamento tra i *widget*.

Capitolo 3: Descrizione del progetto ed implementazione



Figura 3.4: Wiring del mashup realizzato

L’obiettivo principale del mashup implementato è quello di consentire agli utenti finali di poter conoscere i servizi di misurazione disponibili, selezionare le osservazioni effettuate e poterle di graficare. Per la realizzazione del nostro mashup abbiamo riutilizzato o modificato alcuni dei *widget* e *operator* a disposizione sullo store di Fiware, altri invece, li abbiamo ideati e sviluppati interamente da zero. Elenchiamo di seguito, fornendo una breve descrizione, i moduli software utilizzati. I *widget* da noi sviluppati sono:

- **Il ServiceWidget:** contatta la nostra macchina Orion per conoscere i servizi di misurazione disponibili e attraverso una sottoscrizione richiede di essere aggiornato per l’eventuale modifica o creazione di un nuovo servizio. Inoltre mostra all’utente i servizi disponibili.
- **Il QueryWidget:** interroga la nostra macchina Orion per ottenere tutte le osservazioni che appartengono ad un determinato servizio di misurazione (temperatura, pressione, pm10).

I *widget* che abbiamo riutilizzato sono invece:

Capitolo 3: Descrizione del progetto ed implementazione

- **Il MapViewer:** consente la visualizzazione delle osservazioni effettuate su una mappa.
- **Il LinearGraph:** consente di graficare i dati di misura relativi ad una determinata osservazione

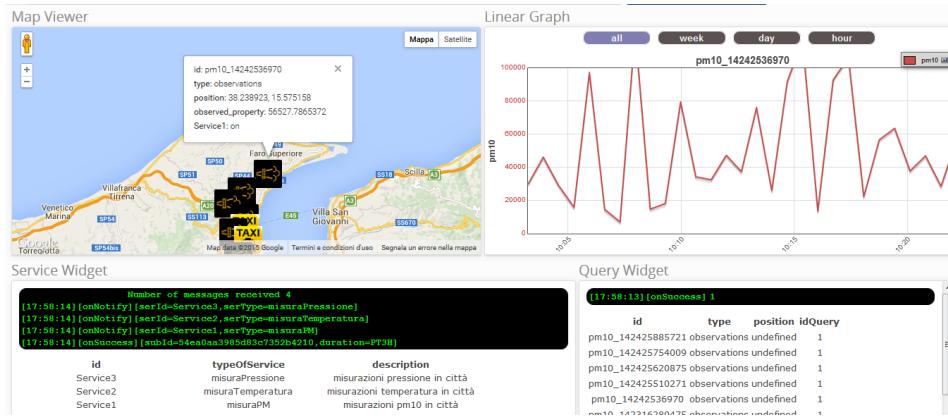


Figura 3.5: Mashup vista utente

Oltre ai due *widget* abbiamo anche realizzato un *operator* che è:

- **L'HistoryHiveToLinearGraph:** contatta il server Hive sul master node di Cosmos per il recupero dei dati relativi ad una determinata osservazione.

L'*operator* che, invece, abbiamo riusato e modificato per le nostre esigenze è:

- **NGSIEntityToPoI:** trasforma un oggetto entità in un oggetto PoI (Point of Interest), affichè possa essere visualizzato su una mappa.

Capitolo 3: Descrizione del progetto ed implementazione

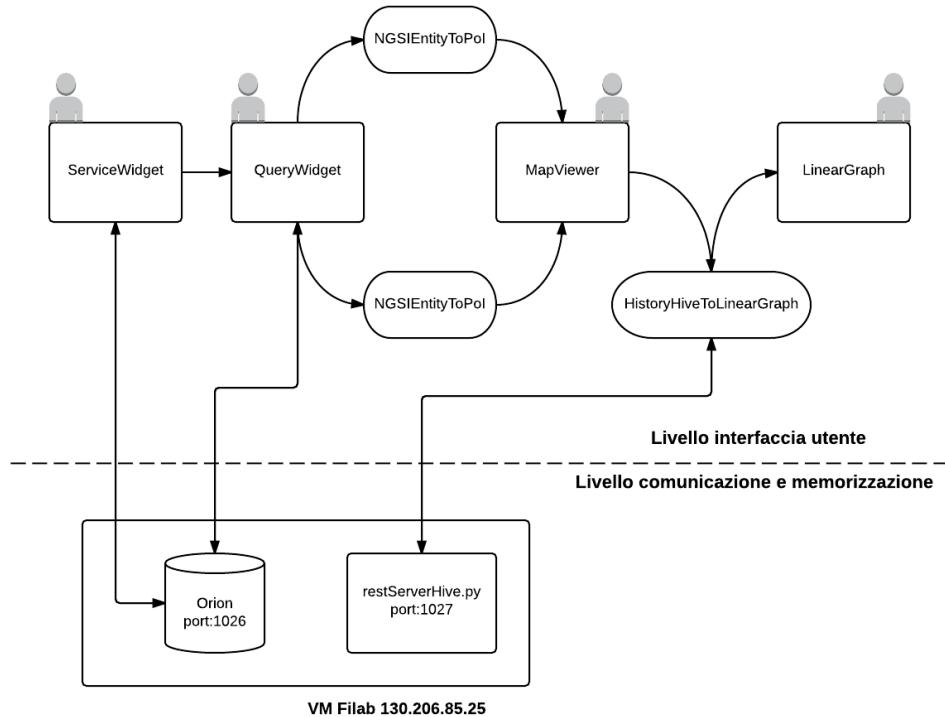


Figura 3.6: Schema livello interfaccia utente

Anche per il livello di interfaccia utente, aiutandoci con la figura 3.6, proviamo a seguire il flusso delle operazioni a partire dalla selezione di un servizio da parte di un utente, fino ad arrivare alla visualizzazione del grafico di un'osservazione. Il *ServiceWidget* contatta la nostra macchina Orion per conoscere i servizi di misurazione disponibili e attraverso una sottoscrizione richiede di essere aggiornato per l'eventuale modifica o creazione di un nuovo servizio. Passa poi al *QueryWidget* il servizio selezionato dall'utente. Il *QueryWidget* contatta la nostra macchina Orion per ottenere tutte le osservazioni (ma in generale tutte le entità) che partecipano a quel servizio, e le passa all'*NGSIEntityToPoI*, che le trasforma in oggetti di tipo PoI. L'*NGSIEntityToPoI operator* passa la lista di oggetti PoI al *MapViewer*, che permette all'utente di visualizzare e selezionare le osservazioni d'interesse su

Capitolo 3: Descrizione del progetto ed implementazione

una mappa. Usiamo due istanze dell’NGSI *operator*, in quanto uno serve per il passaggio delle nuove osservazioni da visualizzare, l’altro per la cancellazione delle osservazioni precedenti. Il *MapViewer* inoltra all’*HistoryHiveToLinearGraph* l’osservazione selezionata dall’utente. L’*HistoryHiveToLinearGraph* contatta quindi il server Hive sul master node di Cosmos all’indirizzo 130.206.80.46 (<http://cosmos.lab-fi-ware.org>) sulla porta 10000, per ricevere i dati relativi all’osservazione e inviarli all’entry point del *LinearGraph*. Quest’ultimo consentirà all’utente di visualizzare i dati dell’osservazione selezionata su un grafico, in modo da facilitarne l’analisi.

3.3 Struttura Dati

In questo paragrafo andiamo ad illustrare la struttura dati ideata ed utilizzata per la realizzazione del nostro sistema. Chiaramente il modello dei dati è stato progettato con riferimento ai concetti di base (*Entities*, *Attributes*, *ContextElements*) concernenti l’architettura del ContextBroker GE, e seguendo quindi le regole del modello NGSI (vedi Capitolo 1). La struttura dati consta di quattro tipi differenti di entità e per ognuno di questi tipi di entità abbiamo definito degli attributi specifici, che secondo il modello NGSI definiscono tutti i campi *name*, *type* e *value*:

- **Service:** rappresenta un determinato servizio (es: misurazione_temperatura). I suoi attributi sono:
 - **description:** descrizione del servizio
 - **typeOfService:** tipo del servizio
- **Procedure:** rappresenta un determinato sensore (es: ShinyeiPPd42NS). I suoi attributi sono:
 - **description:** descrizione del sensore
 - **model:** marca e modello del sensore

Capitolo 3: Descrizione del progetto ed implementazione

- **type:** tipo del sensore (es. fisso o mobile)
- **datasheet:** link al datasheet del sensore
- **observed_property:** grandezza fisica misurata dal sensore (es: temperatura)
- **Taxi:** rappresenta un determinato taxi. I suoi attributi sono:
 - **ID_SERVICE:** id del servizio offerto dal taxi
 - **procedure:** id del sensore montato sul taxi
 - **position:** posizione corrente del taxi
- **Observation:** rappresenta una determinata osservazione effettuata con una Procedure. I suoi attributi sono:
 - **ID_SERVICE:** id del servizio cui l'osservazione appartiene
 - **date:** data dell'osservazione
 - **duration:** durata dell'osservazione
 - **position:** posizione in corrispondenza della quale è stata effettuata l'osservazione
 - **observed_property:** grandezza fisica misurata durante l'osservazione

Capitolo 3: Descrizione del progetto ed implementazione

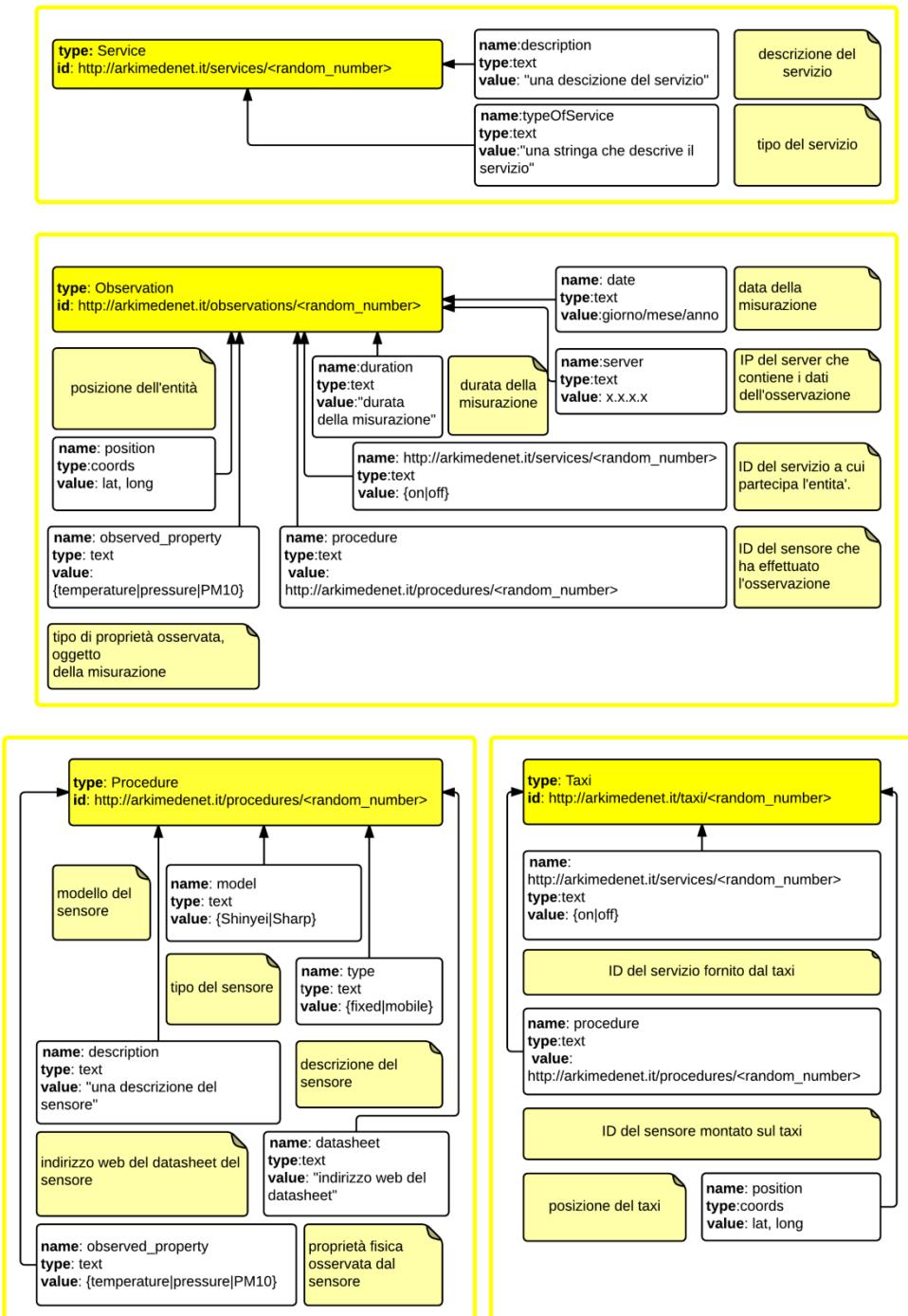


Figura 3.7: Struttura dati del sistema

3.4 Livello Sensoristica: collegamento dei sensori

Nel Capitolo 2 abbiamo riportato i dettagli tecnici e delle descrizioni generali relative a tutti i dispositivi hardware utilizzati per la realizzazione del nostro sistema di monitoraggio. Andiamo a questo punto ad analizzare come abbiamo connesso tra loro i dispositivi per il rilevamento (Shinyei, OBD adapter) con le schede gateway (Raspberry e Linino) .

3.4.1 Interfacciamento dei sensori con la Raspberry

Sulla Raspberry utilizzata per il nostro sistema abbiamo installato il sistema operativo consigliato sul sito ufficiale, ossia il Raspbian OS, su cui troviamo già preinstallato il pacchetto python con tutte le librerie principali di supporto al linguaggio.

Shinyei

Come abbiamo accennato nel Capitolo 2 i pin del sensore Shinyei che abbiamo utilizzato per l’interfacciamento con la scheda sono:

- Il pin1: pin di massa.
- Il pin3: pin di alimentazione del sensore (5V) .
- Il pin4: pin sul quale si misura il segnale d’uscita (digitale).

I pin della GPIO della raspberry che abbiamo utilizzato sono invece:

- Il pin2: pin di alimentazione (5V).
- Il pin6: pin di massa.
- Il pin7 (o GPIO4): pin a cui collegiamo il segnale d’uscita del sensore.

Capitolo 3: Descrizione del progetto ed implementazione

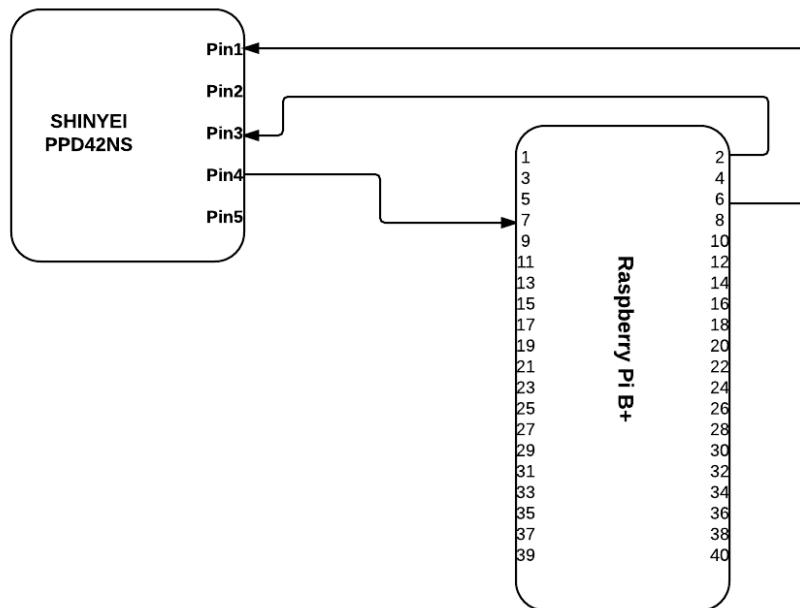


Figura 3.8: Schema di interconnessione shinyei – Raspberry

Il problema principale che abbiamo incontrato nell’interfacciamento del sensore Shinyei con la Raspberry, è stato causato dal fatto che non è possibile applicare ad un pin GPIO una tensione maggiore di 3.3V, in quanto potrebbe portare a malfunzionamenti o addirittura alla rottura del modulo. Purtroppo, misurando con un tester il livello di tensione raggiungibile dal segnale d’uscita del sensore al pin4, ci siamo accorti che questi arrivava a raggiungere valori superiori ai 4V. Per ovviare a questo problema, ci siamo muniti di:

- una breadboard: strumento utilizzato per creare prototipi di circuiti elettrici.
- un po di resistenze da 75 ohm.
- dei cavetti jumper (maschio/femmina) per i collegamenti.

Capitolo 3: Descrizione del progetto ed implementazione

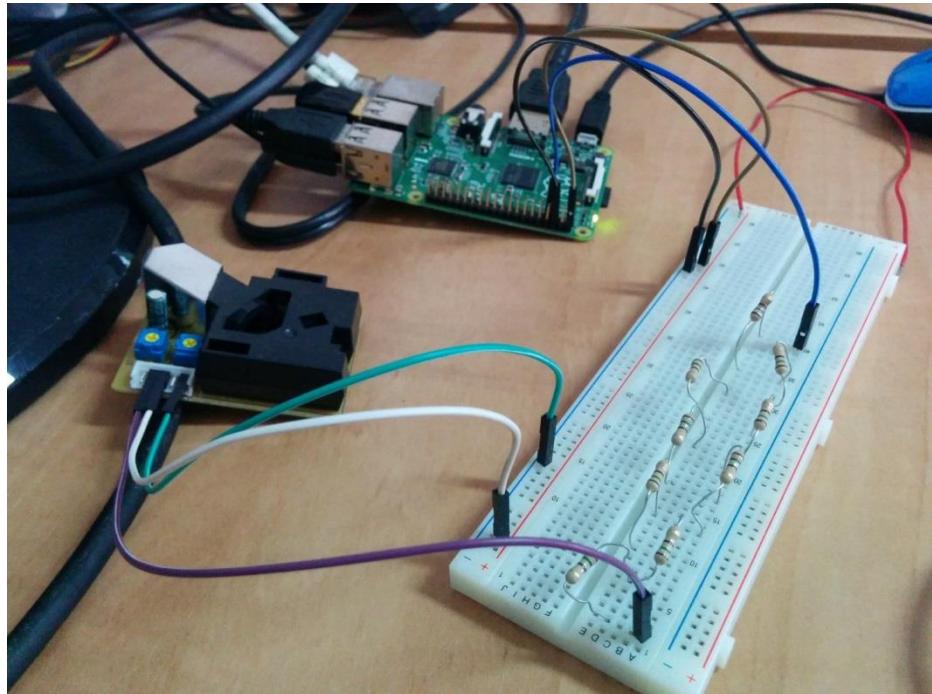


Figura 3.9: Foto del collegamento tra lo Shinyei e la Raspberry

Per ridurre la tensione del segnale d'uscita del sensore, abbiamo utilizzato un metodo sperimentale. Il metodo è consistito nel mettere in serie al segnale di uscita del sensore una serie di resistenze, e misurando tramite il tester il valore di tensione a valle delle resistenze. Abbiamo gradatamente aggiunto resistenze in serie finché il segnale d'uscita, a valle delle resistenze, non superava mai i 3.3V. Questo segnale è quello che abbiamo collegato poi al pin GPIO4 della Raspberry.

Adattatore OBD2

Per la fase di testing del nostro sistema, purtroppo, non abbiamo realmente utilizzato questo dispositivo per la lettura dei dati di pressione e temperatura dalla centralina di un auto, bensì abbiamo utilizzato un approccio diverso, ossia

Capitolo 3: Descrizione del progetto ed implementazione

quello dell'uso di un simulatore OBDII. I motivi che ci hanno portato a scegliere questo tipo di approccio sono stati diversi. Il motivo principale è sicuramente legato a problemi tempistici. Testare l'adattatore sul campo (quindi su un automobile) avrebbe comportato risolvere diversi problemi, tra cui: l'alimentazione della Raspberry, la connessione alla rete, l'installazione del sistema (breadboard compresa) all'interno dell'auto e via dicendo. Oltre al problema tempistico si sono aggiunti ulteriori problemi che andrebbero presi in considerazione se si volesse portare avanti il progetto e portarlo alla commercializzazione. Ci siamo accorti, difatti, che l'adattatore collegato al connettore OBDII dell'auto, quando questa è a motore spento, tende a scaricare molto velocemente la batteria. Inoltre andrebbe migliorato il sistema di sicurezza, in quanto chiunque tramite un dispositivo bluetooth potrebbe facilmente associarsi all'adattatore leggendo ed inviando comandi alla centralina. Per tutta questa serie di motivi abbiamo deciso di utilizzare un simulatore software chiamato **OBDSim**. Questo simulatore consente di simulare il funzionamento dell'ELM327, di simulare connessioni seriali e bluetooth, di generare insiemi di dati ECU¹¹-like e di utilizzare il protocollo OBD per l'interrogazione della centralina. Consente inoltre di usare sorgenti di dati multiple, nel caso si volessero simulare più ECU simultaneamente. Infine è anche un software multipiattaforma, difatti funziona su sistemi Linux, Windows e OSX.

L'installazione di questo software è stata praticamente indolore, in quanto è disponibile, sui repository apt di Raspbian, un pacchetto precompilato contenente OBDSim.

```
> apt-get install obdgpslogger
```

¹¹ Il termine ECU è l'acronimo di Electronic Control Unit, e viene generalmente utilizzato per denotare la centralina elettronica di un'automobile.

Capitolo 3: Descrizione del progetto ed implementazione

Questo simulatore può essere lanciato da riga di comando e può prendere diversi parametri d'ingresso. Il comando con cui abbiamo lanciato il simulatore OBDSim per i nostri test sulla Raspberry è:

```
> obdsim -g Cycle
```

dove il parametro `-g` viene usato per indicare il tipo di generatore da utilizzare. Il generatore *Cycle* consente di generare i valori (della centralina) ciclicamente sulla base del tempo corrente, per cui verranno generati dei valori che aumenteranno linearmente al passare del tempo, e che vengono azzerati solo alla conclusione del ciclo. Il simulatore può essere lanciato usando diversi altri tipi di generatori (anche chiamati plug-in). Tra questi plug-in ne troviamo uno che consente di lanciare un'interfaccia grafica molto semplice da cui è possibile visualizzare graficamente alcuni dei parametri della centralina simulata. Una volta lanciato, il simulatore inizierà a generare dati e risponderà ai comandi OBD inviati sulla pseudo porta seriale `/dev/pts/0`.

3.4.2 Interfacciamento dei sensori con la Linino

Sulla scheda Linino utilizzata per il nostro sistema abbiamo installato l'ultima versione del sistema operativo LininoOS, così come consigliato sul sito ufficiale.

```
$ cd /tmp  
$ wget  
http://download.linino.org/linino_distro/master/latest/openwrt-  
ar71xx-generic-linino-one-16M-squashfs-sysupgrade.bin  
$ sysupgrade -v openwrt-ar71xx-generic-linino-one-16M-  
squashfs-sysupgrade.bin
```

Il Sistema operativo LininoOS è una ottimizzazione per la scheda Linino della distribuzione linux OpenWRT.

Capitolo 3: Descrizione del progetto ed implementazione

Shinyey

Anche per la Linino, i pin del sensore Shinyei che abbiamo usato per l’interfacciamento con la scheda sono:

- Il pin1: pin di massa.
- Il pin3: pin di alimentazione del sensore (5V) .
- Il pin4: pin sul quale si misura il segnale d’uscita (digitale).

Mentre l’unico pin che abbiamo utilizzato sulla Linino è stato:

- Il pin 8: pin a cui collegiamo il segnale d’uscita del sensore.

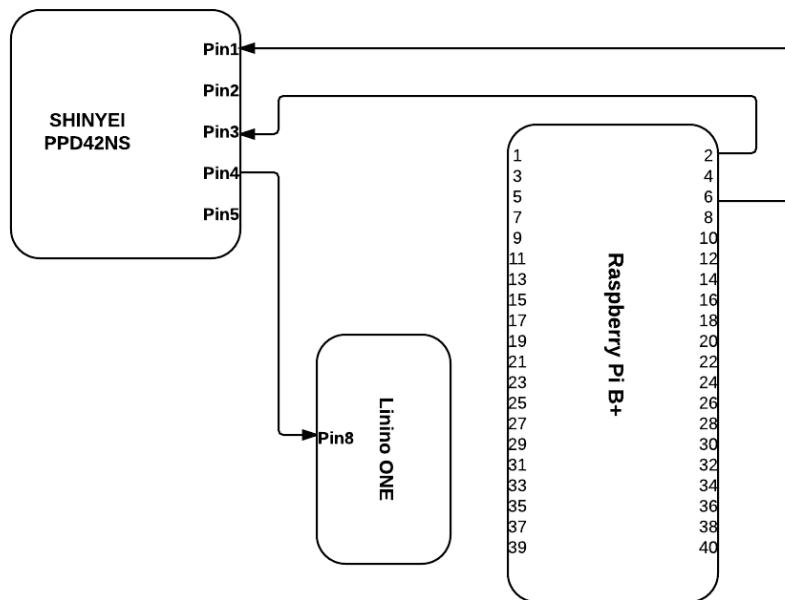


Figura 3.10: Schema di interconnessione Shinyei – Linino

Come è possibile notare dallo schema in figura 3.10, non usiamo i pin di massa e di alimentazione a 5V della scheda Linino per alimentare il sensore, in quanto, la shield per avere l’interfaccia di rete ethernet e la shield che ci consente di disporre di una presa usb e dello slot per la connessione della microSD, occupano praticamente quasi l’intera GPIO della scheda. Per poter

Capitolo 3: Descrizione del progetto ed implementazione

alimentare il sensore attraverso la Linino, senza togliere le due shield, avremmo potuto installare la scheda sulla breadboard facendo combaciare opportunamente i pin inferiori della scheda con i fori della breadboard stessa. Purtroppo, la distanza tra i pin della scheda non combaciava perfettamente con i fori della breadboard a nostra disposizione, per cui abbiamo deciso di alimentare il sensore sfruttando i pin di alimentazione e di massa della Raspberry, e indirizzando sulla GPIO della Linino soltanto il segnale d'uscita del sensore.

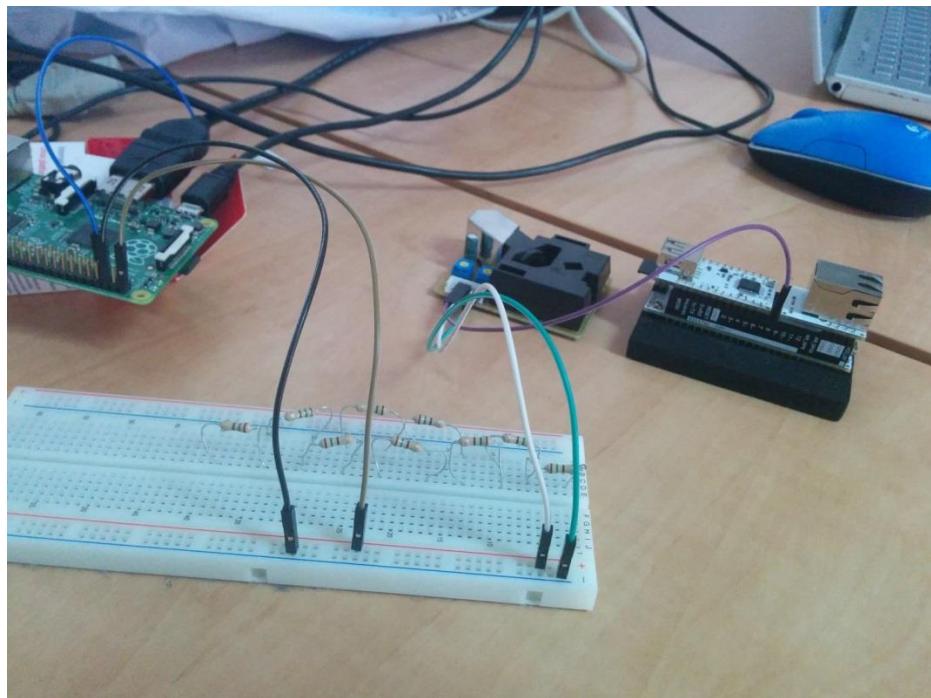


Figura 3.11: Foto del collegamento tra lo Shinyei e la Linino

Adattatore OBD2

Per gli stessi motivi elencati nel paragrafo precedente, anche con la scheda Linino non abbiamo realmente interfacciato l'adattatore OBDII. Solo che a differenza della Raspberry, per la scheda Linino non è stato rilasciato un

Capitolo 3: Descrizione del progetto ed implementazione

pacchetto precompilato contente il simulatore OBDSim, per cui per evitare una lenta procedura di compilazione manuale dei sorgenti abbiamo preferito adottare un approccio diverso. I dati di pressione e temperatura li abbiamo, infatti, generati direttamente tramite la libreria *random* di python.

3.5 Livello Sensoristica: Implementazione software

Nel paragrafo 3.2.1 abbiamo già fornito una visione generale di quelli che sono gli script che sono stati implementati in questo livello. Il linguaggio di riferimento utilizzato su entrambe le boards (Raspberry e Linino) è stato il python. I motivi della scelta sono da ricondurre a quelli che sono i punti di forza del linguaggio stesso, ossia, la facilità d'uso, la portabilità su differenti piattaforme e il sempre crescente numero di librerie in dotazione. Inoltre abbiamo visto come sia su Raspbian sia su LininoOS il pacchetto python sia già preinstallato e pronto all'uso. L'unico script che non è stato realizzato in python è lo Shinyei.ino, ossia lo sketch sviluppato per il microcontrollore della scheda Linino e quindi realizzato seguendo la sintassi del linguaggio di programmazione di Arduino. In questo paragrafo, proveremo a scendere un po' più nel dettaglio dell'implementazione software del sistema, andando a fornire descrizioni più esaustive per ognuno degli script realizzati e riportando le principali sezioni di codice. Iniziamo con l'analisi dei due script Shinyei.py e Shinyei.ino che ci consentono di interagire rispettivamente con le GPIO della Raspberry e della Linino per la lettura dei valori di pm10 rilevati dal sensore Shinyei. Proseguiremo poi con la descrizione di tutti gli altri script python comuni ad entrambe le schede, segnalando all'interno dei rispettivi sottoparagrafi le eventuali lievi differenze di implementazione.

Capitolo 3: Descrizione del progetto ed implementazione

3.5.1 Shinyei.py

Questo script consente di leggere tramite il sensore Shinyei PPD42NS i valori di PM10 dell'aria circostante, quando il sensore è collegato alla GPIO della Raspberry PI. Per la lettura dei dati, abbiamo utilizzato la libreria *RPi.GPIO*¹², la quale mette a disposizione diversi metodi per l'interazione tra il codice sorgente e i pin della Raspberry. Il codice sorgente è stato così organizzato:

Dopo aver importato la libreria RPi.GPIO e le altre librerie necessarie, abbiamo impostato come prima cosa, in quanto obbligatorio, il sistema di numerazione dei pin da utilizzare all'interno del codice, nel nostro caso abbiamo utilizzato il sistema BCM (numerazione relativa adottata dalla Broadcom SoC della rasp).

```
import time
import RPi.GPIO as gpio
import math
gpio.setmode(gpio.BCM)
```

Dopodiché abbiamo settato il pin di ingresso per la lettura dei dati (segnale d'uscita del sensore). In particolar modo abbiamo usato come ingresso il pin GPIO4 e settata la resistenza di pull_up_down in maniera da rilevare inizialmente in ingresso un livello logico alto.

```
gpio.setup(4, gpio.IN, pull_up_down=gpio.PUD_UP)
```

Segue poi la dichiarazione e definizione delle variabili e delle funzioni utilizzate durante la procedura. In particolar modo le funzioni definite sono due:

¹² La documentazione relativa alla libreria RPi.GPIO è consultabile all'indirizzo <https://pypi.python.org/pypi/RPi.GPIO>

Capitolo 3: Descrizione del progetto ed implementazione

- **Start_LOW_pulse:** che resta in attesa finché non si verifica un fronte di discesa del segnale di ingresso al pin GPIO4 e salva l’istante in cui si è verificato il fronte nella variabile **start_time**.
- **Stop_LOW_pulse:** che resta in attesa finché non si verifica un fronte di salita del segnale di ingresso al pin GPIO4 e salva l’istante in cui si è verificato il fronte nella variabile **stop_time**.

```
sample_time=30
def start_LOW_pulse():
    global start_time
    gpio.wait_for_edge(4, gpio.FALLING)
    start_time=time.time()
    #print("GPIO4: Sono LOW!")

def stop_LOW_pulse():
    global stop_time
    gpio.wait_for_edge(4, gpio.RISING)
    stop_time=time.time()
    #print("GPIO4: Sono HIGH!")
```

Così come suggerito nel datasheet del sensore, abbiamo usato un intervallo di campionamento di 30 secondi per la stima della concentrazione di PM10 nell’aria circostante. Per cui, la procedura vera e propria è rappresentata da un loop infinito, all’interno del quale ogni 30 secondi viene stimata la concentrazione di PM10 e archiviato il valore calcolato su dei file. La stima del valore di concentrazione ad ogni passo di campionamento avviene in questo modo:

- Attraverso le funzioni *start_LOW_pulse()* e *stop_LOW_pulse()* vengono calcolate le durate di tutti gli intervalli di tempo all’interno dei quali il segnale d’uscita del sensore si trova a livello logico basso. La durata di ognuno di questi singoli intervalli di tempo, che possiamo chiamare per una migliore identificazione intervalli di LOW, viene salvata nella variabile **duration**.

Capitolo 3: Descrizione del progetto ed implementazione

- Nella variabile **low_pulse_occupancy**, si andranno ad addizionare i valori delle singole duration, finchè non si raggiunge la fine dell'intervallo di campionamento (30 secondi). La variabile **low_pulse_occupancy**, al termine dei 30 secondi, conterrà quindi l'effettiva quantità di tempo (nei 30 secondi) durante la quale il segnale d'uscita del sensore si è trovato sul livello logico basso.
- Dalla variabile **low_pulse_occupancy** viene calcolata la percentuale di tempo durante la quale il livello del segnale è rimasto basso e salvato nella variabile **ratio**. Tramite questa variabile, viene poi calcolato il valore di concentrazione di PM10, sfruttando la funzione di trasformazione riportata nel datasheet del sensore, e tale valore di concentrazione viene salvato nella variabile **concentration**.

```
while 1:  
    try:  
        start_LOW_pulse()  
        stop_LOW_pulse()  
        duration=stop_time - start_time  
        low_pulse_occupancy=low_pulse_occupancy +  
duration  
        if((time.time() -  
start_time_measure)>sample_time):  
            print("Fine misurazione!")  
  
            ratio=(low_pulse_occupancy/sample_time)*100  
            concentration=1.1*math.pow(ratio,3) -  
3.8*math.pow(ratio,2)+520*ratio+0.62  
            out_f=open("shinyei_PM10", "w")  
            out_f.write(str(concentration) + "\n")  
            out_f_log.write(str(concentration) +  
"\n")  
            out_f.close()  
            low_pulse_occupancy=0  
            start_time_measure=time.time()  
            print("Inizio nuova misurazione!")
```

Viene infine salvato il campione di concentrazione appena stimato nei file *shinyei_PM10* e *shinyei_PM10.log*. Il file *Shinyei_PM10.log* viene utilizzato in

Capitolo 3: Descrizione del progetto ed implementazione

modalità *append* per salvare lo storico delle rilevazioni effettuate, mentre il contenuto del file *Shinyei_PM10* viene di volta in volta riscritto con l'ultimo valore di pm10 misurato. Infine verranno riazzerate tutte le variabili e si darà inizio ad una nuova misurazione.

3.5.2 Shinyei.ino

Questo sketch consente di leggere tramite il sensore Shinyei PPD42NS i valori di PM10 dell'aria circostante, quando il sensore è collegato alla GPIO della Linino One. Come si intuisce dall'estensione del file, l'interazione con la GPIO Linino è stata realizzata attraverso il microcontrollore ed i linguaggio di programmazione di Arduino. Come tutti gli sketch Arduino, la struttura di base della procedura si sviluppa sulla definizione di due funzioni:

- *void setup()*: funzione utilizzata per l'inizializzazione delle variabili utilizzate, per impostare lo stato dei pin, per l'impostazione delle comunicazioni seriali e per far partire le librerie da usare durante il ciclo d'esecuzione.
- *void loop()*: funzione che viene eseguita ciclicamente e che contiene la procedura vera e propria, nel nostro caso quindi, l'insieme di istruzioni che ci consentono di leggere ed elaborare il segnale d'uscita del sensore per stimare i valori di concentrazione di PM10 rilevati dal sensore.

Per la scrittura dei valori di concentrazione stimati, sul file system della microSD, importiamo per prima cosa l'header file *FileIO.h*.

```
#include <FileIO.h>
```

Capitolo 3: Descrizione del progetto ed implementazione

Dopodiché definiamo ed inizializziamo tutte le variabili, e settiamo l'intervallo di campionamento (**sample_time_ms**) a 30 secondi.

```
int pin = 8;
unsigned long duration;
unsigned long starttime;
unsigned long sampleteime_ms = 30000;
unsigned long lowpulseoccupancy = 0;
float ratio = 0;
float concentration = 0;

void setup() {
    Bridge.begin();
    Serial.begin(9600);
    FileSystem.begin();
    pinMode(8, INPUT);
    starttime = millis();
}
```

All'interno della funzione *setup()*, facciamo partire: la libreria *Bridge* per facilitare la comunicazione tra il microcontrollore ed il microprocessore, la libreria *FileSystem* per la scrittura dei dati sulla microSD e la libreria *Serial* per la comunicazione seriale che consente la scrittura sull'interfaccia seriale dell'*ArduinoIDE*. Settiamo inoltre il pin 8 come pin d'ingresso per la ricezione del segnale d'uscita del sensore. All'interno della funzione *loop()*, calcoliamo le durate di tutti gli intervalli di tempo (intervalli di LOW) all'interno dei quali il segnale d'uscita del sensore si trova a livello logico basso, attraverso la funzione *pulseIn()*. Allo stesso modo di come è stato fatto nello script *Shinyei.py*, si sommano all'interno della variabile **low_pulse_occupancy** tutti gli intervalli di LOW calcolati all'interno di un intervallo di campionamento, e a partire da questa quantità viene calcolata la percentuale di tempo durante il quale il segnale d'uscita del sensore è rimasto sul livello logico basso (variabile **ratio**). Sfruttando la funzione di trasformazione riportata sul datasheet del sensore, calcoliamo quindi il valore di concentrazione di PM10.

```
void loop() {
```

Capitolo 3: Descrizione del progetto ed implementazione

```
duration = pulseIn(pin, LOW);
lowpulseoccupancy = lowpulseoccupancy+duration;

if ((millis()-starttime) > sampletime_ms)
{
    File dataFile =
FileSystem.open("/mnt/sda1/script_py/shinyei_PM10",
FILE_WRITE);
    if(dataFile)
    {
        ratio = lowpulseoccupancy/(sampletime_ms*10.0);
concentration = 1.1*pow(ratio,3)-
3.8*pow(ratio,2)+520*ratio+0.62
Serial.println(concentration);
        Serial.flush();

        dataFile.print(concentration);
        dataFile.flush();
        dataFile.close();
        lowpulseoccupancy = 0;
        starttime = millis();
    }
    else
    {
        Serial.println("Error opening file!!!");
        Serial.flush();
        delay(3000);
    }
}
}
```

Prima dell'inizio di un nuovo passo di campionamento e quindi di una nuova misurazione viene salvato il valore di concentrazione calcolato sul file system della microSD all'interno del file *shinyei_PM10*, viene azzerato il valore della **low_pulse_occupancy** e viene risettata la variabile **starttime** all'istante di tempo corrente per il conteggio del prossimo intervallo di campionamento.

3.5.3 Obd.py

Lo script `obd.py` consente l’interfacciamento tra i gateway (Raspberry e Linino) e l’adattatore OBDII per l’interrogazione ed il recupero dei dati dalla centralina di un auto. Tale interfacciamento è stato simulato sia sulla Raspberry, sia sulla Linino, seguendo però due approcci differenti (vedi paragrafo 3.4). Nel caso della Raspberry, dove abbiamo utilizzato un simulatore dell’ELM327, abbiamo simulato l’interfacciamento tramite l’utilizzo della libreria `obd0.1.0`¹³, la quale mette a disposizione metodi per l’instaurazione di una connessione seriale e vari metodi per l’invio di comandi OBD all’ELM327. I comandi che inviamo alla centralina dell’auto tramite lo script sono quelli per ottenere i dati di pressione e temperatura.

```
import obd
import time

old_temperature = 0
old_pressure = 0

connection = obd.OBD("/dev/pts/0")

temperature = obd.commands['AMBIANT_AIR_TEMP']
pressure = obd.commands['BAROMETRIC_PRESSURE']

temp_file_log = open("obd_temperatore.log", "a+")
pres_file_log = open("obd_pressure.log", "a+")
```

All’inizio dello script, una volta importata la libreria `obd`, instauriamo una connessione seriale (oggetto **connection**) con la porta virtuale `/dev/pts/0` su cui risponde il simulatore OBDSim. Salviamo poi i comandi per il recupero dei dati di temperatura e pressione, attraverso il metodo *commands* della libreria. Eseguite queste operazioni iniziali, lo script si riduce ad un ciclo infinito, all’interno del quale, ogni 30 secondi (tempo di campionamento usato per la

¹³ La documentazione relativa alla libreria `obd0.1.0` è consultabile all’indirizzo <https://pypi.python.org/pypi/obd/0.1.0>

Capitolo 3: Descrizione del progetto ed implementazione

misurazione del PM10), viene inviato tramite il metodo *query()* dell’oggetto **connection** il comando per il prelevamento del valore corrente di temperatura e pressione. Tali valori vengono scritti rispettivamente nei file *obd_temperature*, *obd_temperature.log*, *obd_pressure* e *obd_pressure.log*.

```
while 1:  
    if connection.has_command(temperature):  
        response = connection.query(temperature)  
        print response.value, response.unit  
        temp_file = open("obd_temperature", "w")  
        temp_file.write(str(response.value))  
        temp_file.close()  
        temp_file_log.write(str(response.value) + "\n")  
        temp_file_log.flush()  
    else:  
        print "temperature not supported"  
  
    if connection.has_command(pressure):  
        response = connection.query(pressure)  
        print response.value, response.unit  
        pres_file = open("obd_pressure", "w")  
        pres_file.write(str(response.value))  
        pres_file.close()  
        pres_file_log.write(str(response.value) + "\n")  
        pres_file_log.flush()  
    else:  
        print "temperature not supported"  
  
    time.sleep(30)
```

Allo stesso modo di come è stato fatto per l’archiviazione dei valori di PM10, abbiamo utilizzato i file *.log* in modalità *append*, per la memorizzazione dello storico dei dati di temperatura e pressione, mentre i restanti file sono stati utilizzati per il salvataggio dell’ultimo valore di lettura delle due grandezze.

Nel caso della scheda Linino, il codice dello script è stato leggermente modificato, in quanto, non abbiamo utilizzato il simulatore OBDSim per la generazione dei valori di temperatura e pressione (vedi paragrafo 3.4), bensì la libreria *random* di python.

Capitolo 3: Descrizione del progetto ed implementazione

```
while 1:  
    temperature = random.randint(-300,300)  
    pressure = random.randint(0,300)  
  
    temp_file = open("obd_temperature", "w")  
    temp_file.write(str(temperature))  
    ...  
    ...  
    time.sleep(30)
```

La differenza sostanziale tra le due versione sta nel fatto che, nel caso Linino, non utilizziamo la libreria *obd0.1.0*, quindi non instauriamo alcuna connessione seriale, ed invece di richiedere i dati tramite una query al simulatore OBDSim, li generiamo direttamente tramite la funzione *randint()* della libreria *random*.

3.5.4 Script NGSI per la connessione ad Orion

Per la connessione alla nostra macchina Orion utilizziamo rispettivamente tre script che sono:

- *createEntity.py*: crea un'entità con un attributo su una macchina Orion.
- *setSubscription.py*: esegue una sottoscrizione per un attributo di un'entità su una macchina Orion.
- *UpdateEntityAttribute.py*: aggiorna il valore di un attributo di un'entità su una macchina Orion.

Lo scopo di questi script è quello di modellare ed eseguire delle richieste HTTP, secondo le regole del protocollo NGSI, in modo da consentirci l'interfacciamento con la nostra macchina Orion per la creazione, sottoscrizione e aggiornamento di ogni entità *observation* generata dal nostro sistema di monitoraggio. Ognuno dei tre script utilizza il file di configurazione

Capitolo 3: Descrizione del progetto ed implementazione

config.ini, per la lettura dei dati necessari all’instaurazione della connessione alla nostra macchina Orion.

```
[user]
username=gioakbombaci@gmail.com
token=0FOTNLpYPECnYeYIbocgxFCI5BmcviGoqPsVXLi7uNmE1tNJ
ROtATmh43gYf5WgzmNexfrlTDaiFmLdQU5D6cg

[contextbroker]
host=130.206.85.25
port=1026
OAuth=yes
```

Nel file vengono specificati: l’indirizzo ip della macchina che ospita l’istanza del Context Broker, la porta su cui Orion è in ascolto, l’account utente ed il rispettivo token da utilizzare per l’autenticazione sulla piattaforma Fiware. In ognuno dei tre script viene quindi importata la libreria *ConfigParser* per il parsing del file *config.ini*, oltre che tutte le altre librerie necessarie, tra le quali troviamo la libreria *requests* per l’esecuzione delle richieste HTTP.

```
import ConfigParser
import requests, json
import io

CONFIG_FILE = "config.ini"

with open(CONFIG_FILE, 'r+') as f:
    sample_config = f.read()
    config =
ConfigParser.RawConfigParser(allow_no_value=True)
    config.readfp(io.BytesIO(sample_config))

CB_HOST=config.get('contextbroker', 'host')
CB_PORT=config.get('contextbroker', 'port')
CB_AAA=config.get('contextbroker', 'OAuth')
if CB_AAA == "yes":
    TOKEN=config.get('user', 'token')
else:
    TOKEN="NULL"
```

L’header delle richieste HTTP viene costruito allo stesso modo per tutti e tre gli script.

Capitolo 3: Descrizione del progetto ed implementazione

```
HEADERS = {'content-type':  
'application/json', 'accept': 'application/json', 'X-Auth-  
Token' : TOKEN}
```

Tutti e tre gli script prendono inoltre dei parametri in ingresso, e definiscono l'URL ed il PAYLOAD della richiesta HTTP secondo le regole del protocollo NGSI, in base al tipo di operazione che deve essere eseguita.

createEntity.py

Questo script consente la creazione di un'entità con un determinato attributo su Orion. I parametri che prende in ingresso sono:

- L'**ID** dell'entità da creare.
- Il **tipo dell'entità** da creare.
- Il **nome dell'attributo dell'entità** da creare.
- Il **tipo dell'attributo dell'entità** da creare.
- Il **valore dell'attributo**.

```
if NUM_ARG==6:  
    ENTITY_ID=sys.argv[1]  
    ENTITY_TYPE=sys.argv[2]  
    ENTITY_ATTR=sys.argv[3]  
    ENTITY_ATTR_TYPE=sys.argv[4]  
    ENTITY_ATTR_VALUE=sys.argv[5]  
  
    CB_URL = "http://"+CB_HOST+":"+CB_PORT  
    URL = CB_URL + '/ngsi10/updateContext'  
    PAYLOAD = '{ \  
        "contextElements": [ \  
            { \  
                "type": "'"+ENTITY_TYPE+"'", \  
                "isPattern": "false", \  
                "id": "'"+ENTITY_ID+"'", \  
                "attributes": [ \  
                    { \  
                        "name": "'"+ENTITY_ATTR+"'", \  
                        "type": "'"+ENTITY_ATTR_TYPE+"'", \  
                        "value": "'"+ENTITY_ATTR_VALUE+"'"  
                    }  
                ]  
            }  
        ]  
    }  
    PAYLOAD = PAYLOAD.replace("\\", "\\\\")
```

Capitolo 3: Descrizione del progetto ed implementazione

```
        "value": "'"+ENTITY_ATTR_VALUE+"'\" \
    } \
], \
"updateAction": "APPEND" \
}' \
r = requests.post(URL, data=PAYOUT, headers=HEADERS)
```

Dopo aver salvato i parametri d'ingresso nelle rispettive variabili, vengono creati l'URL ed il PAYLOAD, per la corretta creazione dell'entità. Il PAYLOAD contiene il testo in formato JSON. Infine viene inviata la richiesta POST HTTP di creazione ad Orion, tramite il metodo *post* della libreria *requests*, passandogli come parametri l'URL, il PAYLOAD e l'HEADER precedentemente costruiti.

setSubscription.py

Questo script consente di eseguire la sottoscrizione di un attributo di un entità su Orion. I parametri che prende in ingresso sono:

- L'**ID** dell'entità da sottoscrivere.
- L'**attributo** di cui si vuole notificare il cambiamento.
- L'**URL** del server che deve ricevere le notifiche.

```
if NUM_ARG==4:
    ENTITY_ID=sys.argv[1]
    ENTITY_ATTR=sys.argv[2]
    SERVER_URL=sys.argv[3]

MIN_INTERVAL = "PT5S"
DURATION = "P1M"
ENTITY_TYPE = "observations"
ENTITY_ATTR_WATCH = ENTITY_ATTR
ENTITY_ATTR_NOTIFY = ENTITY_ATTR

CB_URL = "http://"+CB_HOST+":"+CB_PORT
```

Capitolo 3: Descrizione del progetto ed implementazione

```
URL = CB_URL + '/ngsi10/subscribeContext'

PAYLOAD = '{ \
    "entities": [ \
        { \
            "type": "'+ENTITY_TYPE+'", \
            "isPattern": "false", \
            "id": "'+ENTITY_ID+'" \
        } \
    ], \
    "attributes": [ \
        "'+ENTITY_ATTR_NOTIFY+' \
    ], \
    "reference": "'+SERVER_URL+'", \
    "duration": "'+DURATION+'", \
    "notifyConditions": [ \
        { \
            "type": "ONCHANGE", \
            "condValues": [ \
                "'+ENTITY_ATTR_WATCH+' \
            ] \
        } \
    ], \
    "throttling": "'+MIN_INTERVAL+' \
}'

r = requests.post(URL, data=PAYOUT, headers=HEADERS)
```

Vengono salvati i parametri d'ingresso nelle rispettive variabili, e vengono inoltre definiti: il tipo dell'entità da sottoscrivere, il tempo di durata della sottoscrizione ed il minimo intervallo di tempo che deve trascorrere tra due successivi cambiamenti del valore dell'attributo, affinchè questo venga notificato al server definito nel tag *reference*. Anche in questo caso vengono creati opportunamente l'URL ed il PAYLOAD, per la corretta sottoscrizione dell'entità. Il PAYLOAD contiene il testo in formato JSON. Infine viene inviata la richiesta POST HTTP di sottoscrizione ad Orion, tramite il metodo *post* della libreria *requests*, passandogli come parametri l'URL, il PAYLOAD e l'HEADER precedentemente costruiti.

UpdateEntityAttribute.py

Capitolo 3: Descrizione del progetto ed implementazione

Questo script consente l'aggiornamento di un'entità con un determinato attributo su Orion. I parametri che prende in ingresso sono:

- L'**ID** dell'entità da aggiornare.
- Il **tipo dell'entità** da aggiornare.
- Il **nome dell'attributo dell'entità** da aggiornare.
- Il **tipo dell'attributo dell'entità** da aggiornare.
- Il **nuovo valore dell'attributo**.

```
if NUM_ARG==6:  
    ENTITY_ID=sys.argv[1]  
    ENTITY_TYPE=sys.argv[2]  
    ENTITY_ATTR=sys.argv[3]  
    ENTITY_ATTR_TYPE=sys.argv[4]  
    ENTITY_ATTR_VALUE=sys.argv[5]  
  
    CB_URL = "http://"+CB_HOST+":"+CB_PORT  
    URL = CB_URL + '/ngsi10/updateContext'  
    PAYLOAD = '{ \\\n        "contextElements": [ \\\n            { \\\n                "type": "'"+ENTITY_TYPE+"', \\\n                "isPattern": "false", \\\n                "id": "'"+ENTITY_ID+"', \\\n                "attributes": [ \\\n                    { \\\n                        "name": "'"+ENTITY_ATTR+"', \\\n                        "type": "'"+ENTITY_ATTR_TYPE+"', \\\n                        "value": "'"+ENTITY_ATTR_VALUE+"' \\\n                    } \\\n                ] \\\n            } \\\n        ], \\\n        "updateAction": "UPDATE" \\\n    }'  
  
    r = requests.post(URL, data=PAYOUT, headers=HEADERS)
```

Capitolo 3: Descrizione del progetto ed implementazione

Dopo aver salvato i parametri d’ingresso nelle rispettive variabili, vengono creati l’URL ed il PAYLOAD, per il corretto aggiornamento dell’attributo dell’entità. Il PAYLOAD contiene il testo in formato JSON. Infine viene inviata la richiesta POST HTTP di update ad Orion, tramite il metodo *post* della libreria *requests*, passandogli come parametri l’URL, il PAYLOAD e l’HEADER precedentemente costruiti.

3.5.5 Send_measures.py

Questo script viene mandato in esecuzione dal *server_rasp.py*, ogni qual volta viene iniziata una nuova osservazione. Lo script può essere diviso in tre fasi che sono:

- Fase di creazione delle tre entità *observations* (temperatura, pressione e pm10)
- Fase di sottoscrizione dell’attributo *observed_property* delle tre entità, necessaria per l’invio delle notifiche di cambiamento al *subscriptionServer.py* (livello comunicazione e memorizzazione)
- Fase di aggiornamento durante la quale viene inviato ciclicamente ad Orion il valore dell’attributo *observed_property* delle tre entità, leggendo di volta in volta i nuovi valori di misura dai rispettivi file di appoggio (*Shinyei_PM10*, *obd_temperature*, *obd_pressure*)

Il *send_measures.py* prende tre parametri in ingresso che sono:

- Il **timestamp** relativo all’istante di inizio dell’osservazione: verrà inserito come parte dell’ID delle entità *observations*
- Il **tipo delle entità** da creare: *observations*
- La **posizione** del taxi: espressa sotto forma di coppia di valori (latitudine, longitudine)

Capitolo 3: Descrizione del progetto ed implementazione

Importate le librerie necessarie, vengono dapprima letti i parametri d'ingresso, inizializzate a 0 le variabili per il controllo dell'istante di ultima modifica dei file di appoggio e creati gli id per le tre entità di temperatura, pressione e pm10 sulla base del *timestamp* passato da riga di comando.

```
if NUM_ARG==4:  
    ENTITY_ID=sys.argv[1]  
    ENTITY_TYPE=sys.argv[2]  
    POSITION=sys.argv[3]  
  
    OBD_TEMP_PATH = "obd_temperature"  
    OBD_PRES_PATH = "obd_pressure"  
    SHINYEI_PM10_PATH = "shinyei_PM10"  
  
    lastModifyTemp = 0  
    lastModifyPres = 0  
    lastModifyPM10 = 0  
  
    id_temp = "temperature_" + ENTITY_ID  
    id_pres = "pressure_" + ENTITY_ID  
    id_pm10 = "pm10_" + ENTITY_ID
```

Inizia a questo punto la fase di creazione delle entità, che consiste fondamentalmente nel lanciare lo script *createEntity.py*, tramite il metodo *system()* della libreria *os* di python. Questo script verrà richiamato un numero di volte pari al numero di entità da creare moltiplicato per il numero di attributi che si vogliono creare per ciascuna entità.

```
    create_temp = "python2.7 createEntity.py" + " " +  
    id_temp + " " + ENTITY_TYPE + " " + "position " + "coords  
" + POSITION #+ " &"  
    os.system(create_temp)  
    create_temp = "python2.7 createEntity.py" + " " +  
    id_temp + " " + ENTITY_TYPE + " " + "observed_property "  
    + "temperature " + "NaN" #+ " &"  
    os.system(create_temp)  
    create_temp = "python2.7 createEntity.py" + " " +  
    id_temp + " " + ENTITY_TYPE + " " + "Service2 " + "text "  
    + "on" #+ " &"  
    os.system(create_temp)
```

Capitolo 3: Descrizione del progetto ed implementazione

```
create_press = "python2.7 createEntity.py" + " " +
id_pres + " " + ENTITY_TYPE + " " + "position " + "coords
" + POSITION #+ " &
os.system(create_press)
...
...
```

Alla fase di creazione delle entità segue quella di sottoscrizione, che consiste nel lanciare lo script *setSubscription.py* per sottoscrivere l'attributo *observed_property* delle tre entità verso il *subscriptionServer*.

```
sub_temp = "python2.7 setSubscription.py" + " " +
id_temp + " " + "observed_property " +
"http://130.206.85.25:7777/accumulate"
sub_press = "python2.7 setSubscription.py" + " " +
id_pres + " " + "observed_property " +
"http://130.206.85.25:7777/accumulate"
sub_pm10 = "python2.7 setSubscription.py" + " " +
id_pm10 + " " + "observed_property " +
"http://130.206.85.25:7777/accumulate"

os.system(sub_temp)
os.system(sub_press)
os.system(sub_pm10)
```

Nell'ultima fase lo script entra in un loop infinito, all'interno del quale viene implementata la fase di aggiornamento. Questa fase consiste nel controllare ogni 30 secondi (intervallo di campionamento), tramite il metodo *os.path.getmtime()*, se la data di ultima modifica di ognuno dei file di appoggio è cambiata, e quindi se è disponibile un valore di misura più aggiornato. In questo caso viene lanciato lo script *UpdateEntityAttribute.py* per l'aggiornamento della/e entità.

```
if lastModifyTemp !=  
os.path.getmtime(OBD_TEMP_PATH):  
    lastModifyTemp =  
os.path.getmtime(OBD_TEMP_PATH)  
    temp_file = open(OBD_TEMP_PATH, "r")  
    value = temp_file.readline()
```

Capitolo 3: Descrizione del progetto ed implementazione

```
parameters = "python2.7  
UpdateEntityAttribute.py" + " " + id_temp + " " +  
ENTITY_TYPE + " observed_property temperature " +  
str(value)  
os.system(str(parameters))  
...  
...  
time.sleep(30)
```

3.5.6 Server_rasp.py

Questo script è stato realizzato nell'ottica di consentire, per gli sviluppi futuri, l'interfacciamento delle boards (Raspberry e Linino) con il tablet installato sul taxi. Questo script è stato sviluppato per comportarsi come un REST server, accettando chiamate GET HTTP da parte di un client (nel nostro caso il *taxi_emulator.py*), sulle risorse definite al suo interno. Le funzionalità REST sono state implementate utilizzando la libreria *web* di python, che consente la creazione di una risorsa del REST server attraverso la semplice definizione di una classe python e dei suoi metodi (GET,POST,PUT...). Le risorse principali del *server_rasp.py* sono quelle che permettono l'avvio e lo stop di una osservazione, attraverso il lancio degli opportuni script. Nella parte iniziale dello script viene definita una tupla, tramite la quale la libreria *web* esegue l'associazione tra l'URL delle risorse e le classi che le gestiscono.

```
urls = (  
    '/start/observation', 'start_obs',  
    '/stop/observation', 'stop_obs',  
    '/quit', 'quit'  
)
```

Le classi associate alle due risorse (avvio e stop misurazioni) sono la *start_obs* e la *stop_obs*. In entrambe viene definito il metodo GET, che conterrà rispettivamente le istruzioni per avviare e fermare gli script necessari

Capitolo 3: Descrizione del progetto ed implementazione

ad effettuare le misurazioni ed inviare i dati sulla piattaforma (*obd.py*, *shinyei.py*, *send-measures.py*).

```
class start_obs:

    def GET(self):
        tablet_data = web.input()
        id_entity = tablet_data.id
        id_type = tablet_data.type
        position = tablet_data.position
        tokens = position.split()
        position = tokens[0] + '\\\\' + tokens[1]

        start_obd = "python2.7 obd.py > obd.out &"
        start_pm10 = "python2.7 Shinyei.py >
shinyei.out &" 
        start_send_measures = "python2.7
send_measures.py" + " " + id_entity + " " + id_type + " "
+ position + " &"

        os.system(str(start_pm10))
        os.system(str(start_obd))
        os.system(str(start_send_measures))
```

Abbiamo utilizzato il metodo *input()* della libreria *web* per il recupero dei parametri inviati all'interno dell'url dal *taxi_emulator.py*, che sono:

- Il **timestamp** d'inizio osservazione: che sarà parte dell'id delle entità da creare.
- Il **tipo delle entità** da creare: *observations*.
- Le **coordinate GPS** del punto in cui il taxi si è fermato per l'inizio della nuova *observations*.

```
class stop_obs:

    def GET(self):
        python_process = []
        list_process = psutil.pids()

        for pid in list_process:
```

Capitolo 3: Descrizione del progetto ed implementazione

```
p = psutil.Process(pid)
command = p.cmdline()
if len(command) == 0:
    continue
if command[0] == 'python2.7' and
len(command) > 1:
    if command[1] == "obd.py" or
command[1] == "Shinyei.py" or command[1] ==
"send_measures.py":
        print "killed process " +
command[1]
        p.kill()

python_process.append(command[1])
```

All'interno della classe *stop_obs* abbiamo utilizzato i metodi della libreria *psutil* per effettuare il killing dei processi *obd.py*, *shinyei.py* e *send_measure.py*. L'applicazione server viene lanciata da riga di comando passando come primo parametro la porta (1234) su cui accetta le connessioni HTTP.

```
if __name__ == "__main__":
    app = web.application(urls, globals())
    app.run()
```

Sulla scheda Linino il *server_rasp.py* è stato leggermente modificato per due motivi. Il primo motivo è legato al fatto che utilizziamo lo sketch Arduino per la rilevazione del PM10, per cui questo resterà sempre in esecuzione all'interno del microcontrollore. Il secondo motivo è legato al fatto che nei repository Linino non è disponibile il pacchetto precompilato della libreria *psutil* di python usata sulla Raspberry per il killing dei processi. Per cui la fase di killing dei processi è stata implementata tramite l'uso di uno script bash (*kill_all.sh*).

```
#!/bin/sh

obd=$(pgrep -f "python2.7 obd.py")
sendmeasures=$(pgrep -f "python2.7 send_measures.py")
```

Capitolo 3: Descrizione del progetto ed implementazione

```
kill -9 $obd $sendmeasures
```

È quindi stato modificato opportunamente il metodo GET della classe *stop_obs*.

```
class stop_obs:

    def GET(self):
        #pdb.set_trace()
        kill_all_proc = "./kill_all.sh"
        os.system(str(kill_all_proc))
```

3.5.7 Taxi_emulator.py

Questo script è stato realizzato per automatizzare la fase di testing del nostro sistema, emulando sia lo spostamento del taxi in base ad una lista di coordinate predefinita, sia la comunicazione con il *server_rasp.py*. La logica che si è voluta realizzare è che, ogni volta che il taxi resta fermo sulle stesse coordinate per più di un certo intervallo di tempo, entra nello stato di STOP e deve essere inviato il comando d'inizio nuova osservazione alla board (Raspberry o Linino) installata sul taxi stesso. Ogni volta che il taxi esce dallo stato di STOP, deve essere lanciato il comando di stop dell'osservazione. Lo script è stato implementato seguendo un approccio multi-threading grazie all'utilizzo della libreria *threading* di python. In particolar modo, *taxi_emulator.py* genera e lancia due thread:

- Un thread che si occupa di simulare il cambio delle coordinate
- Un thread che si occupa di controllare periodicamente lo stato del taxi (fermo o in movimento) e inviare di conseguenza i comandi di *start/observation* e *stop/observation* al *server_rasp.py*.

Il codice dello script può essere consultato in Appendice A.2.

3.6 Livello di comunicazione e memorizzazione

Il livello di comunicazione e memorizzazione rappresenta il livello centrale del nostro sistema, ossia quello che mette in collegamento il livello fisico dei sensori con il livello dell’interfaccia utente. Come spiegato in maniera generale nel paragrafo 3.2, questo livello comunica dal basso con il livello di sensoristica, prelevando e memorizzando in maniera permanente i dati di misura effettuati dal sistema di monitoraggio (fisico), mentre comunica dall’alto con il livello di interfaccia utente, fornendo su richiesta le informazioni memorizzate. Questo livello racchiude praticamente tutte le risorse, hardware e software, che lavorano “dietro le quinte”, ma che risultano fondamentali per la funzionalità, l’utilizzabilità e la coesione di tutte le singole componenti dell’intero sistema. Proprio per questo motivo, questo livello può anche essere denominato come il livello di Back-end. Le principali componenti che prendono parte a questo livello sono:

- La nostra macchina virtuale deployata sul portale Cloud del Filab, sulla quale stanno in esecuzione, l’**Orion Context Broker**, il **subscriptionServer.py** ed il **restServerHive.py**.
- Il nostro **cluster Cosmos**.
- Il **CKAN engine**.

Nel prosieguo del paragrafo illustreremo quali sono stati i passaggi necessari per il deployment e la configurazione della VM sul Filab. Le altre componenti del livello verranno trattate più approfonditamente dal collega Antonio Caristia.

Capitolo 3: Descrizione del progetto ed implementazione

3.6.1 La nostra VM sul Filab

Il nodo centrale di tutto il sistema è costituito dalla macchina virtuale che abbiamo deployato sulla piattaforma Fiware attraverso il portale Cloud del Filab. Tale VM l'abbiamo deployata attraverso la sezione *Images*, all'interno della quale troviamo uno svariato numero di immagini di sistema, configurate diversamente l'una dall'altra, contenenti diversi GEi già preinstallati, e pronte per essere istanziate. Per le nostre esigenze, non abbiamo avuto bisogno di creare un template ad hoc, ma abbiamo potuto sfruttare l'*orion-psb-image-R3.4* dalle *Images* disponibili.

Actions						
Name	Type	Status	Visibility	Container Format	Disk Format	Actions
wirecloud-img	firmware:apps	active	public	OVF	QCOW2	<button>Launch</button>
orion-psb-image-R3.4	firmware:data	active	public	OVF	QCOW2	<button>Launch</button>
kurento-image-R5.0.33	firmware:data	active	public	OVF	QCOW2	<button>Launch</button>
cep-r3.3.3-img	firmware:data	active	public	OVF	QCOW2	<button>Launch</button>
orion-psb-image-R4.1	firmware:data	active	public	OVF	QCOW2	<button>Launch</button>
kurento-image-R5.0.4	firmware:data	active	public	OVF	QCOW2	<button>Launch</button>
ofnic-image-R2.3	firmware:2nd	active	public	AMI	AMI	<button>Launch</button>
airnode-hrmn	firmware:link	active	public	AMI	AMI	<button>Launch</button>

Figura 3.12: Immagine utilizzata per la nostra VM

L'immagine *orion-psb-image-R3.4* è fondamentalmente l'immagine di una *Centos* release 6.3, con già preinstallata la versione 0.17.0 dell'Orion Context Broker. Una volta lanciata la VM tramite l'apposito tasto *Launch*, è possibile controllare lo stato della nostra istanza dalla sezione *Instances*. Abbiamo a questo punto richiesto l'assegnazione di un indirizzo ip pubblico alla nostra macchina virtuale attraverso la sezione *Security*, in modo da renderla raggiungibile dai “gateway” del livello di sensoristica (Raspberry e Linino).

Capitolo 3: Descrizione del progetto ed implementazione

Instances							
	Instance Name	IP Address	Size	Keypair	Status	Task	Power State
<input type="checkbox"/>	idas	10.0.9.115	2048 MB RAM 1 VCPU 10GB Disk	antonio	SHUTOFF	None	RUNNING
<input type="checkbox"/>	orionold	10.0.5.92 130.206.85.25	2048 MB RAM 1 VCPU 10GB Disk	antonio	ACTIVE	None	RUNNING
<input type="checkbox"/>	oriontemplate-orion-l...	10.0.5.214	2048 MB RAM 1 VCPU 10GB Disk	antonio	SHUTOFF	None	RUNNING

Figura 3.13: Istanza della nostra VM

Attraverso la sottosezione *Keypairs* della sezione *Security*, abbiamo creato una chiave per l'accesso remoto alla macchina, che usiamo per connetterci alla VM in SSH. Inoltre, dalla sottosezione *Security Groups*, sempre della sezione *Security*, abbiamo creato un gruppo di sicurezza, all'interno del quale abbiamo specificato le porte TCP da cui è possibile accettare connessioni.

Edit Security Group Rules				
Security Group Rules				
IP Protocol	From Port	To Port	Source	Action
TCP	443	443	0.0.0.0/0 (CIDR)	<button>Delete Rule</button>
TCP	1026	1026	0.0.0.0/0 (CIDR)	<button>Delete Rule</button>
TCP	7777	7777	0.0.0.0/0 (CIDR)	<button>Delete Rule</button>
TCP	1027	1027	0.0.0.0/0 (CIDR)	<button>Delete Rule</button>
TCP	8080	8080	0.0.0.0/0 (CIDR)	<button>Delete Rule</button>

Displaying 13 items

Add Rule

IP Protocol: From Port *: To Port *: Source Group: CIDR:

Figura 3.14: Regole di sicurezza della nostra VM

L'accesso alla macchina viene eseguito digitando su un terminale il comando

```
> ssh -i chiave.pem root@130.206.85.25
```

Capitolo 3: Descrizione del progetto ed implementazione

L’istanza dell’Orion Context Broker installata, fa da collante con il livello di sensoristica e gioca il ruolo di centro di smistamento delle informazioni verso tutte le direzioni. Si occupa di ricevere le informazioni sulle osservazioni effettuate, e di inviarle sia alle applicazioni che ne fanno richiesta dal livello superiore, sia a quelle che ne hanno fatto richiesta mediante sottoscrizione dal livello inferiore. Oltre all’istanza Orion la VM ospita anche:

- Il **subscriptionServer.py**: che consente di instaurare una comunicazione virtuale tra Orion e Cosmos e tra Orion e CKAN. Nella fattispecie, lo script consente di memorizzare in maniera permanente le informazioni raccolte dal sistema di monitoraggio sia su Cosmos si su CKAN e di tenerle costantemente aggiornate attraverso le regolari notifiche ricevute da Orion.
- Il **restServerHive.py**: che consente il prelevamento di informazioni dal cluster Cosmos ed il rendimento di quest’ultime alla componente che ne ha fatto richiesta.

Capitolo 4

Studio delle prestazioni

4.1 Introduzione

Come ben sappiamo, lo sviluppo di sistemi software sempre più complessi e farraginosi, hanno fatto sì che il passo di progettazione relativo allo studio prestazionale di un sistema sia uno step doveroso e inevitabile.

Nei capitoli precedenti abbiamo descritto in maniera dettagliata gli strumenti utilizzati e le varie fasi implementative del progetto ai vari livelli. In questo capitolo proveremo, invece, ad illustrare e spiegare quanto abbiamo prodotto nell'ultima fase del nostro lavoro, ossia quella dedicata allo studio delle prestazioni del nostro applicativo. L'approccio che abbiamo utilizzato è stato di tipo simulativo, in quanto lo scarso numero di sensori e board a disposizione, nonché i vari problemi (installazione in auto, sicurezza, ecc...) discussi nel Capitolo 3, non ci ha permesso di poter testare e stressare

l'applicazione direttamente sul campo. I risultati prestazionali sono stati ottenuti seguendo due diverse tecniche di studio.

La prima tecnica è stata quella di utilizzare un tool grafico chiamato JMeter, in grado di eseguire delle analisi prestazionali di software, soprattutto di software orientati al web. Tramite JMeter è possibile simulare connessioni a database e servizi web con svariati tipi di protocolli, tra cui il protocollo HTTP.

La seconda tecnica, invece, è stata quella di realizzare direttamente degli script python che fungessero da simulatore del nostro sistema e tramite i quali è stato possibile eseguire un'analisi delle prestazioni, ricorrendo all'uso della libreria *time* del linguaggio stesso.

Quest'ultimo metodo di studio è quello che illustreremo più approfonditamente nel prosieguo di questo capitolo e di cui riporteremo statistiche e risultati finali.

4.2 Simulazione in Python

Come è possibile evincere dai capitoli precedenti, il componente software che più di tutti gli altri potrebbe causare un rallentamento e conseguentemente una decaduta delle prestazioni del sistema è senza dubbio l'Orion Context Broker. Come abbiamo già visto l'Orion Context Broker è il componente che si occupa di gestire richieste di Update e creazione di nuovi contenuti, di rispondere a richieste di Query sui dati, e di accettare richieste di sottoscrizioni, notificando opportunamente a chi di dovere gli aggiornamenti delle entità sottoscritte. È quindi facile intuire che la bontà prestazionale della nostra applicazione è strettamente legata al numero di connessioni contemporanee a cui l'Orion Context Broker è in grado di rispondere. Proprio per questo motivo il nostro lavoro di analisi si è concentrato prevalentemente nello studio delle prestazioni di questo componente, andando a valutare, posto un determinato

scenario iniziale, le condizioni che portano ad ottenere dei colli di bottiglia e quindi dei rallentamenti dell'intero sistema.

L'analisi delle prestazioni è stata fatta direttamente all'interno del codice python con cui abbiamo implementato il nostro simulatore (vedi Appendice A.1). Tutte le richieste che il Broker è in grado di gestire (Update, Query, Subscription e Notify) sono fondamentalmente richieste HTTP, per cui la valutazione delle prestazioni è stata eseguita semplicemente valutando i tempi di risposta di ogni singola richiesta tramite la funzione *time()* della libreria *time* di python. Tale funzione non fa altro che restituire un floating point rappresentante il numero di secondi trascorsi dall'1 gennaio 1970, da cui si può facilmente estrapolare il valore in millisecondi moltiplicando per mille il risultato restituito. Richiamando tale funzione prima e dopo ogni singola richiesta HTTP effettuata al Broker, e attraverso una semplice operazione di differenza, riusciamo ad ottenere i tempi di risposta alle singole richieste e poter quindi valutare sotto quali condizioni l'Orion Context Broker non riesce più a rispondere alle varie utenze in tempi ragionevoli. I tempi di risposta ed i confronti tra i vari scenari testati sono stati graficati attraverso Matlab; gli script realizzati possono essere consultati in Appendice B.

4.2.1 Flusso del processo e parametri di simulazione

Lo script python di simulazione che abbiamo realizzato è stato strutturato in maniera tale da riprodurre il più fedelmente possibile quelli che sono i passi di esecuzione e di interazione del nostro applicativo con l'Orion Context Broker. Lo script richiede dei parametri d'ingresso obbligatori che rappresentano fondamentalmente i parametri di simulazione, e tramite i quali è possibile riprodurre i più svariati scenari in cui l'applicativo stesso si può ritrovare a dover operare.

Capitolo 4: Studio delle prestazioni

I parametri di simulazione che vengono passati allo script simulatore direttamente da riga di comando sono:

- NUM_ORION: rappresenta il numero di Orion Context Broker da utilizzare durante il processo di simulazione.
- NUM_TAXI: rappresenta il numero di Taxi, ed in generale il numero di utenti che si vuole rendere attivi durante il processo di simulazione.
- NUM_UPDATE: rappresenta il numero di richieste di Update e di Query che verranno eseguite su ogni singola entità creata in fase di simulazione.
- TIME_TO_UPDATE: rappresenta l'intervallo di tempo che deve trascorrere tra due successive richieste di Update e di Query su una stessa entità durante il processo di simulazione.

Il processo di simulazione segue un determinato flusso di esecuzione che può essere suddiviso in più fasi. Di seguito riportiamo un diagramma che rappresenta la struttura del processo.

Capitolo 4: Studio delle prestazioni

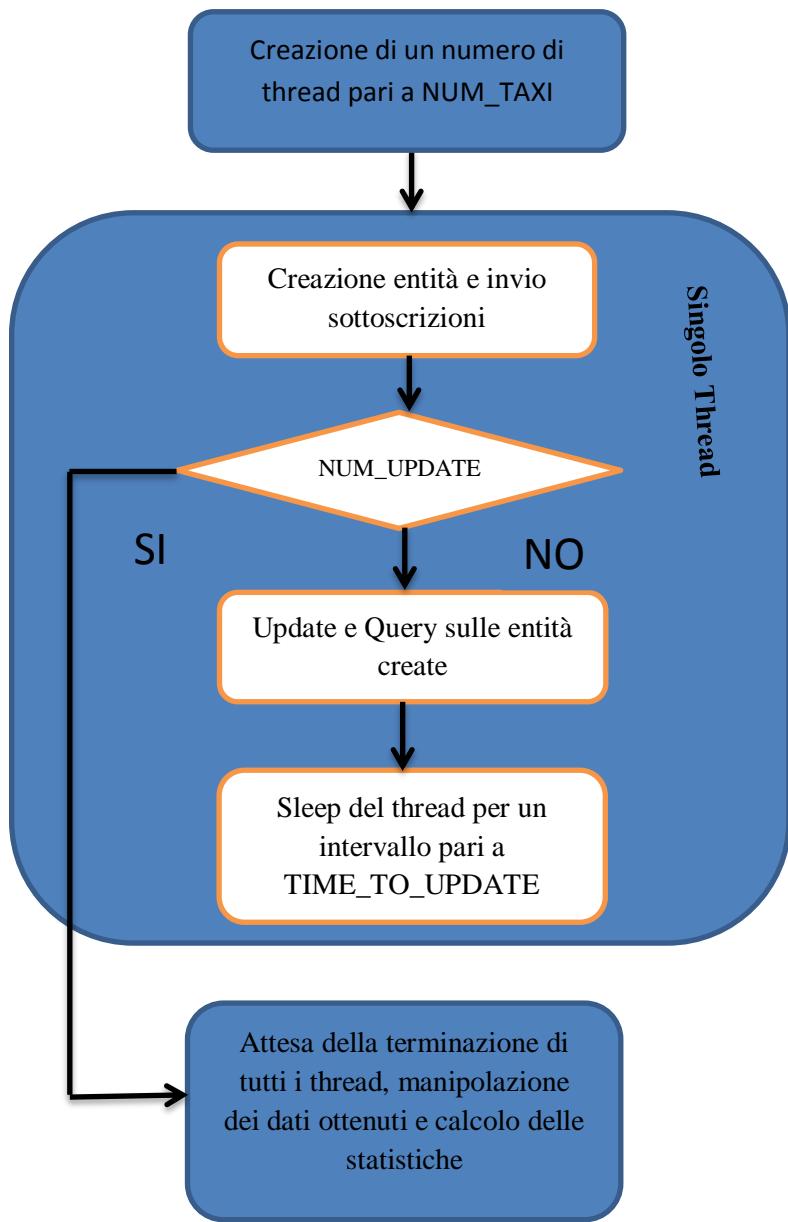


Figura 4.1: Diagramma di flusso del simulatore python

Capitolo 4: Studio delle prestazioni

La prima fase è quella di creazione dei thread, ogni thread rappresenta un utente ed esegue sistematicamente un certo numero di operazioni. Le istruzioni eseguite da ogni thread rappresentano fondamentalmente il flusso di operazioni che vengono eseguite su un singolo taxi in un determinato intervallo di tempo. Anche il flusso di esecuzione di un thread possiamo quindi dividerlo in fasi. Vengono inizialmente create tre entità che rappresentano le observation di temperatura, pressione e pm10. Successivamente vengono eseguite le sottoscrizioni verso un server “fake” (necessario per simulare le sottoscrizioni al subSubscriptionServer della nostra applicazione) per tutte e tre le entità. Una volta eseguita la creazione e la sottoscrizione delle entità, ogni thread entra in un loop all’interno del quale vengono generati randomicamente i valori di temperatura, pressione e pm10. Ad ogni passo del loop vengono quindi eseguite una richiesta di Update e una di Query per ognuna delle tre entità create. La condizione di uscita dal loop sarà data dal raggiungimento di un numero di passi pari a NUM_UPDATE, e l’uscita dal loop determinerà anche la terminazione del singolo thread. Chiaramente, come già anticipato precedentemente, vengono calcolati e salvati su appositi file i tempi di risposta dell’Orion Context Broker per ogni singola richiesta HTTP effettuata. Infine, durante l’ultima fase di simulazione, si attende che tutti i thread abbiano finito la propria esecuzione per poter iniziare la manipolazione dei dati salvati e ottenere così statistiche e risultati finali.

4.3 Configurazione Singolo Orion: Scenario e casi di studio

Per studiare al meglio il comportamento dell'Orion Context Broker quando sottoposto ad un carico di lavoro elevato abbiamo deciso di installare il componente su un host locale, in modo da eludere eventuali aleatorietà sui ritardi causate dal traffico della rete internet. L'host su cui abbiamo installato e configurato il Broker è una macchina virtuale con le seguenti caratteristiche:

- SO: Centos 6.6 a 64 bit
- RAM: 4 GB

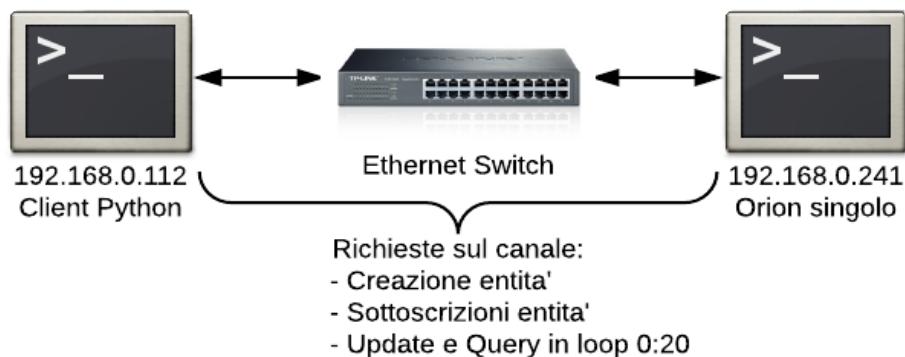


Figura 4.2: Configurazione usata nel caso di Orion singolo

Per stressare il più possibile l'Orion Context Broker e portarlo a delle condizioni limite di lavoro, abbiamo impostato i parametri di simulazione in modo da riprodurre uno scenario diverso da quello relativo al normale funzionamento della nostra applicazione. In particolar modo abbiamo scelto uno scenario in cui abbiamo lasciato fissi due parametri di simulazione:

Capitolo 4: Studio delle prestazioni

- NUM_UPDATE: 20
- TIME_TO_UPDATE: 10 (secondi)

Abbiamo invece giocato su un terzo parametro per aumentare o diminuire il numero di connessioni al Broker. Precisamente abbiamo aumentato in maniera più o meno lineare il numero di utenti fino a quando abbiamo riscontrato il raggiungimento di punti critici di funzionamento.

- NUM_TAXI: 5-10-20-30

Sono stati quindi lanciati quattro processi di simulazione, con 5, 10, 20 e 30 taxi/utenti (numero di thread).

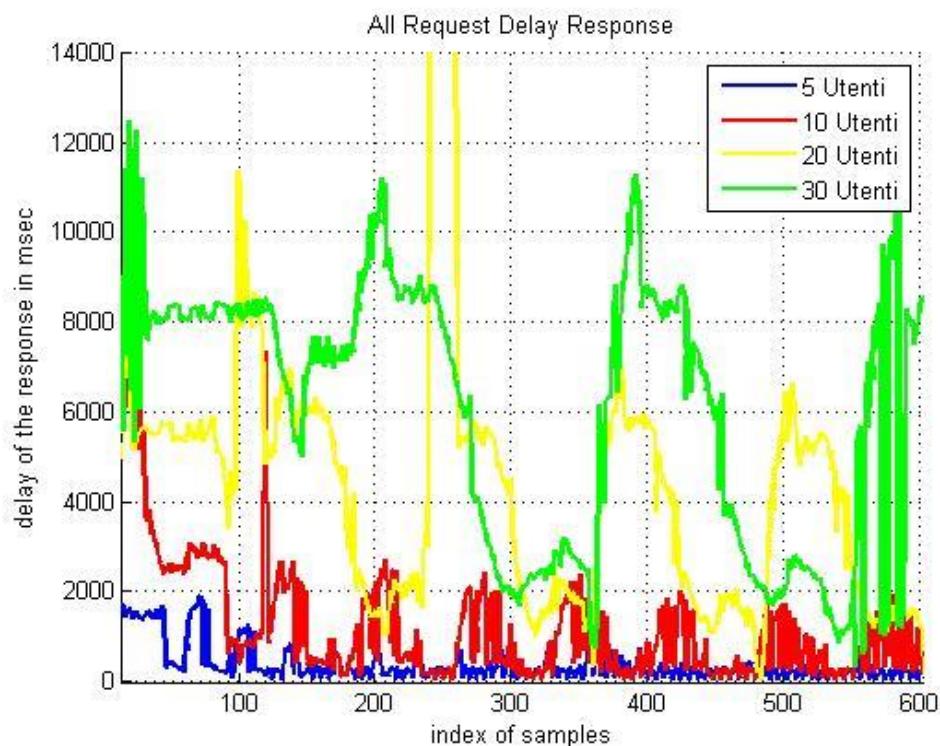


Figura 4.3: Confronto dei ritardi di risposta dell'Orion Context Broker con 5-10-20-30 utenti

Sull'asse delle ascisse della figura 4.3 sono riportati i campioni, ossia tutte le singole richieste HTTP inoltrate al Broker in ordine cronologico (Create, Subscription, Update e Query). Sull'asse delle ordinate è riportato, invece, il ritardo di risposta di ogni singola connessione espresso in millisecondi. Come ci si aspetta, aumentando il numero di utenti che interagiscono attivamente con il Broker aumentano anche i ritardi di risposta del Broker stesso, fino ad arrivare a ritardi dell'ordine dei 12 secondi nel caso di 30 utenti. In questo caso specifico, il Broker si trova a dover lavorare in condizioni di carico eccessivo, che possono provocare perdite di dati.

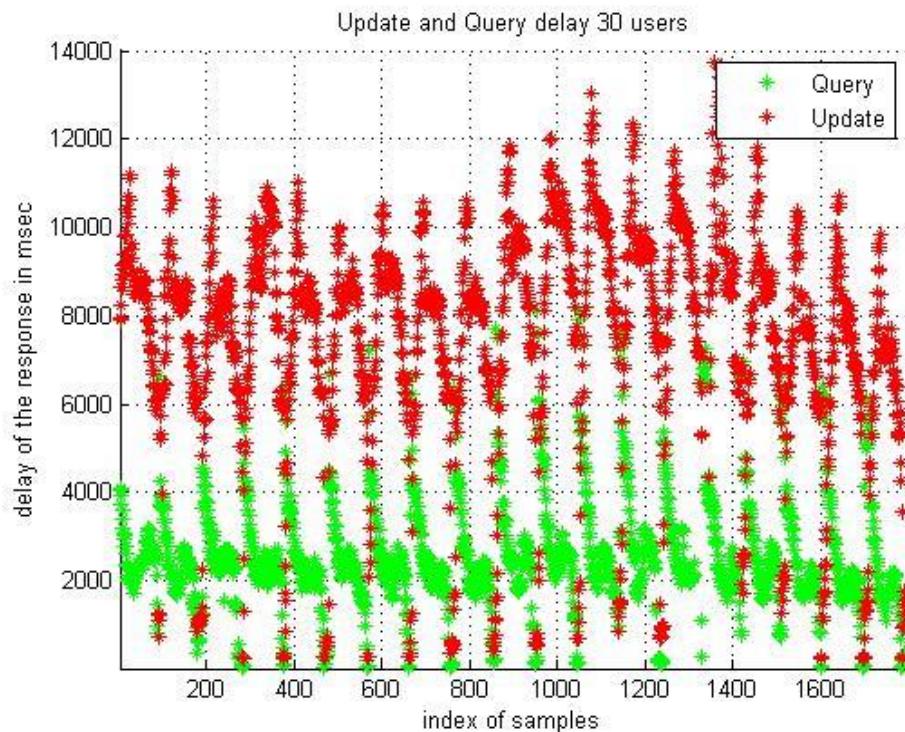


Figura 4.4: Ritardi di risposta di Orion nel caso di richieste di Query e Update con 30 utenti

Difatti, analizzando opportunamente la figura 4.4, ci accorgiamo che tutte le richieste di Query vengono processate dall’Orion con un ritardo inferiore al TIME_TO_UPDATE (10 secondi), ma gran parte delle richieste di Update non riescono ad essere processate all’interno di quest’intervallo, con conseguente perdita di dati. Il passo successivo della nostra fase di test e studio delle prestazioni è stato quindi quello di utilizzare due Orion Context Broker federati tra loro, per provare a migliorare le prestazioni a parità di carico di lavoro.

4.4 Federazione e studio della scalabilità

Nel Capitolo 1, abbiamo presentato e descritto in maniera abbastanza dettagliata le caratteristiche del Context Broker, tra cui la possibilità di poter federare tra loro più componenti di questo tipo. Non avendo trovato un’esaustiva documentazione a riguardo, abbiamo deciso di fare dei test per studiarne sia la funzionalità che la scalabilità. Il tipo di federazione che abbiamo testato ed analizzato è stato il tipo “push”, che come già anticipato nei capitoli precedenti ci permette di avere due Orion contenenti i dati di nostro interesse e che cambiano entrambi il proprio stato locale ad ogni aggiornamento di entità. Ciò che cambia nel flusso di esecuzione della simulazione è soltanto il fatto che per ogni singola entità creata da ogni thread, vengono eseguite due sottoscrizioni sull’Orion primario: una verso il server “fake”, l’altra verso l’Orion federato.

Capitolo 4: Studio delle prestazioni

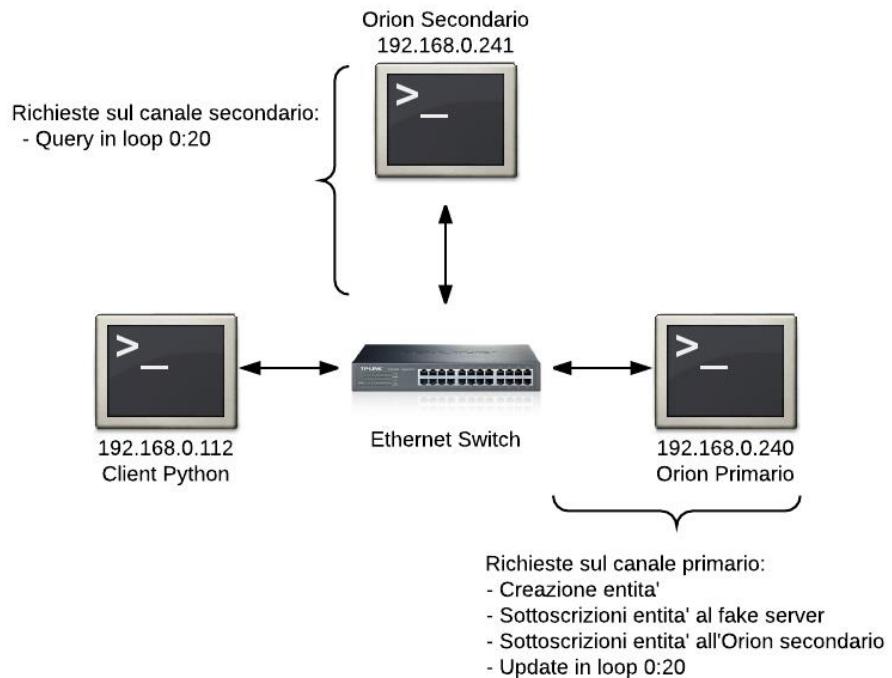


Figura 4.5: Configurazione usata nel caso di due Orion federati

Chiaramente per poter sfruttare i vantaggi della federazione abbiamo opportunamente bilanciato il carico di lavoro tra i due Orion utilizzati. In special modo abbiamo utilizzato l'Orion primario per le operazioni di creazione e sottoscrizione delle entità, nonché per l'invio delle richieste di aggiornamento. Il secondo Orion l'abbiamo invece dedicato alle operazioni di Query sulle entità. Anche in questo caso, per sottrarci ad eventuali ritardi causati dal traffico della rete internet abbiamo preferito eseguire le simulazioni utilizzando due Orion installati all'interno di una rete LAN locale. L'Orion primario è stato installato su una macchina virtuale avente le seguenti caratteristiche:

- SO: Centos 6.6 a 64 bit
- RAM: 2.6 GB

Capitolo 4: Studio delle prestazioni

mentre l'Orion secondario sulla stessa macchina virtuale con la quale abbiamo eseguito i test con un singolo Orion:

- SO: Centos 6.6 a 64 bit
- RAM: 4 GB

Anche in questo caso sono stati presi in considerazione i quattro scenari adottati per i test sul singolo Orion quindi:

- NUM_UPDATE: 20
- TIME_TO_UPDATE: 10 (secondi)
- NUM_TAXI: 5-10-20-30

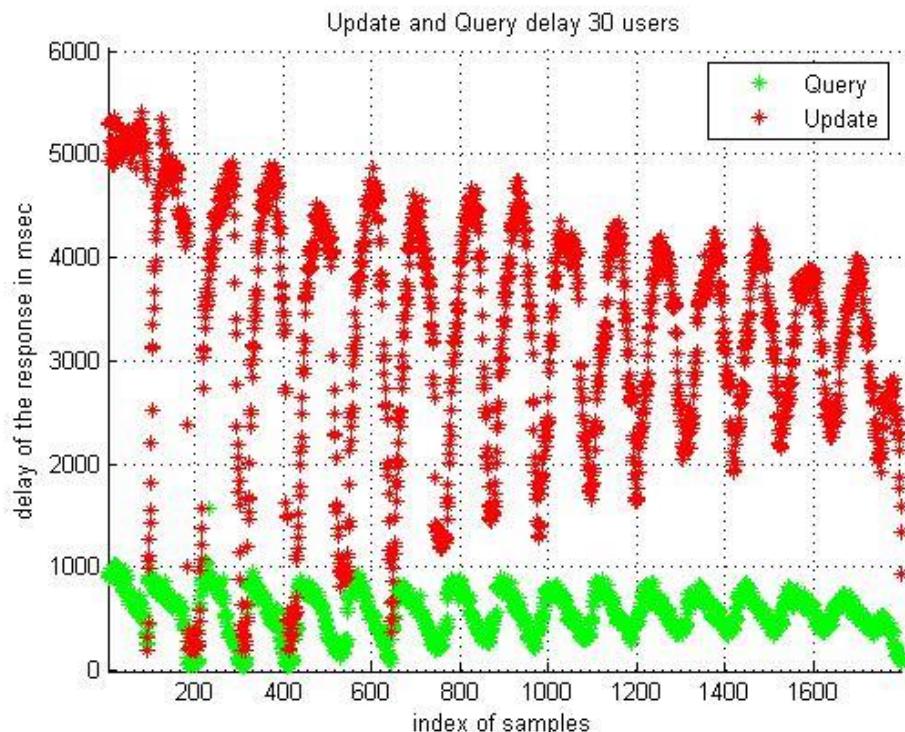


Figura 4.6: Ritardo di risposta di due Orion federati con 30 utenti. Richieste di Update all'Orion primario, richieste di Query all'Orion secondario.

Dalla figura 4.6 possiamo evincere immediatamente che il sistema non si trova più in una condizione critica di funzionamento, in quanto, sia le richieste di Query, sia quelle di Update, vengono processate con tempi di ritardo in tutti i casi inferiori rispetto al TIME_TO_UPDATE. Viene in questo caso mantenuta la correttezza dei dati.

Il primo risultato che abbiamo ottenuto da questa fase di sperimentazione è quindi che, utilizzando la tecnica della “push federation”, e bilanciando opportunamente il carico di lavoro tra gli Orion federati è possibile scalare, quindi mantenere delle buone prestazioni anche all'aumentare del numero di utenti.

4.5 Confronti: Un Orion vs Federazione

Gli altri risultati di questa fase di sperimentazione li abbiamo conseguiti effettuando dei confronti sui ritardi delle richieste di Query e di Update nelle due rispettive configurazioni (1 Orion – 2 Orion federati). La domanda che ci poniamo a questo punto è: considerando i casi di studio, conviene sempre usare la federazione?

Avendo considerato e simulato quattro differenti scenari, andremo ad analizzare i confronti per ognuno dei quattro casi, una volta per le richieste di Query, un'altra per le richieste di Update.

4.5.1 Query

Relativamente alle richieste di Query, possiamo rispondere alla precedente domanda analizzando i grafici sotto riportati.

Capitolo 4: Studio delle prestazioni

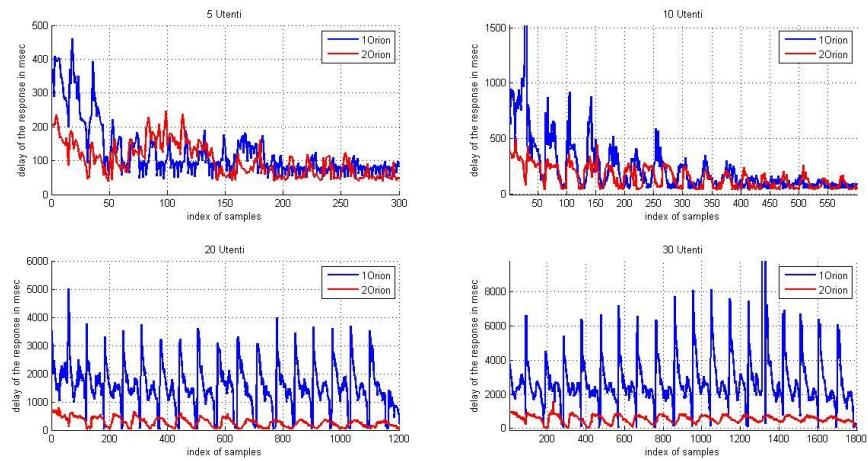


Figura 4.7: Confronto tra i ritardi di risposta delle Query nel caso di uso di un singolo Orion e nel caso di utilizzo di due Orion federati

Esaminando la figura 4.7, possiamo affermare che, posti gli specifici scenari hardware e software presi in considerazione, nei casi con 5 e 10 utenti si potrebbe tranquillamente risparmiare l'utilizzo del secondo Orion, in quanto i ritardi di risposta nelle due differenti configurazioni (1 Orion – 2 Orion federati), soprattutto una volta raggiunto il valore di regime, sono più o meno equivalenti. Possiamo maggiormente avvalorare quanto affermato mostrando anche i grafici relativi alle medie dei ritardi di risposta.

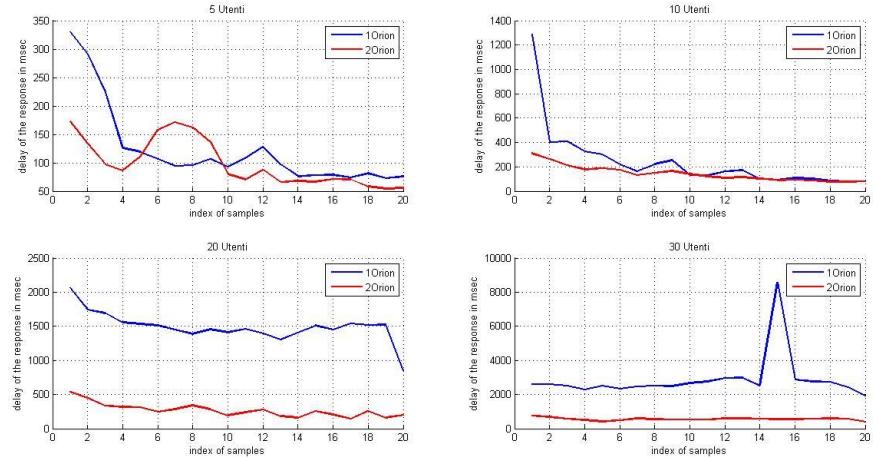


Figura 4.8: Confronto tra i ritardi medi di risposta delle Query nel caso di uso di un singolo Orion e nel caso di utilizzo di due Orion federati

La figura 4.8 riporta un’interpolazione dei tempi di ritardo medi relativi alle richieste di Query. I tempi di ritardo medi sono stati calcolati ad ogni passo del loop di aggiornamento (NUM_UPDATE), mediando chiaramente tra tutte le richieste di Query effettuate da tutti i thread durante il processo di simulazione. Abbiamo quindi ottenuto NUM_UPDATE=20 campioni di tempi di ritardo medi sull’asse delle ascisse (un campione di ritardo medio ad ogni passo del ciclo), mentre abbiamo rappresentato al solito il ritardo in millisecondi sull’asse delle ordinate. Analizzando i quattro grafici relativi ai quattro casi di studio è evidente come nei primi due casi (5-10 utenti) i tempi di ritardo medi nelle due configurazioni sono abbastanza simili, mentre nei secondi due casi (20-30 utenti) si nota un netto risparmio di tempo usando la federazione.

4.5.2 Update

Anche relativamente alle richieste di Update possiamo rispondere alla domanda precedente analizzando i grafici sotto riportati.

Capitolo 4: Studio delle prestazioni

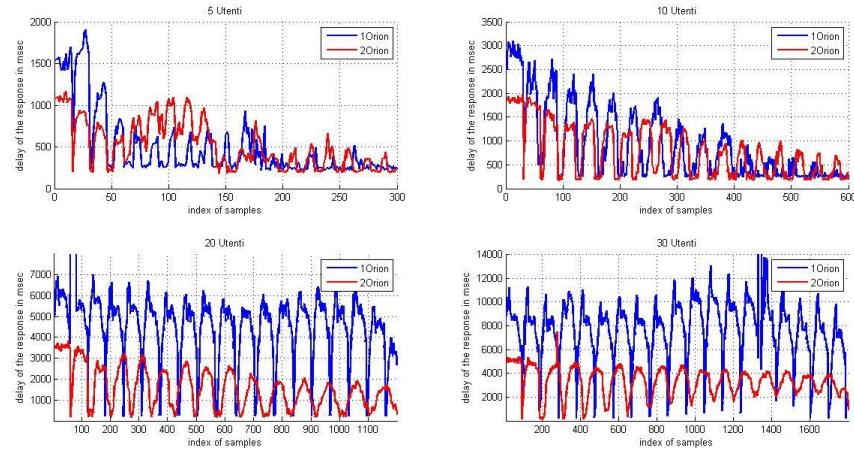


Figura 4.9: Confronto tra i ritardi di risposta delle Update nel caso di uso di un singolo Orion e nel caso di utilizzo di due Orion federati

Esaminando la figura 4.9, possiamo affermare anche in questo caso che, ha senso usare la federazione tra due Orion solo nei casi con 20 e 30 utenti, in quanto i ritardi dei tempi di risposta risultano essere decisamente inferiori rispetto alla configurazione con un Orion singolo. Riportiamo di seguito i grafici relativi alle medie dei tempi di ritardo per confermare ulteriormente quanto appena detto.

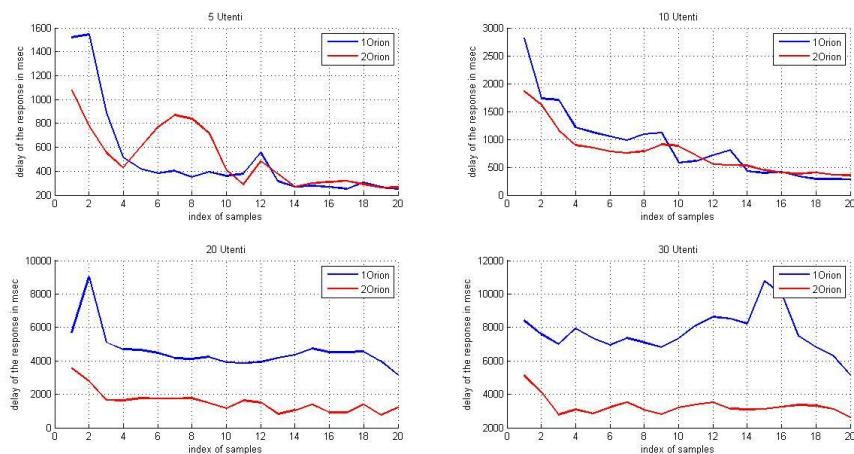


Figura 4.10: Confronto tra i ritardi medi di risposta delle Query nel caso di uso di un singolo Orion e nel caso di utilizzo di due Orion federati

Allo stesso modo della figura 4.8, la figura 4.10 riporta un’interpolazione dei tempi di ritardo medi relativi però alle richieste di Update. Analizzando i quattro grafici relativi ai quattro casi di studio è evidente come nei primi due casi (5-10 utenti) i tempi di ritardo medi sono abbastanza simili, non traendo quindi vantaggi dall’utilizzo dei due Orion federati, mentre nei secondi due casi (20-30 utenti) l’uso di un maggior numero di risorse viene compensato da una netta diminuzione dei ritardi di risposta.

Purtroppo, per mancanza di ulteriore tempo, non siamo riusciti ad eseguire altre simulazioni, magari considerando scenari ancora diversi e federando tra loro un numero ancora maggiore di Orion. Ciò non ci ha consentito di studiare nel dettaglio quello che è l’andamento della scalabilità di questa componente (se lineare, esponenziale ecc...), ma tale lavoro può senza dubbio essere inserito tra gli studi e sviluppi futuri.

4.6 Conclusioni e sviluppi futuri

In questo paragrafo, faremo un breve resoconto di quelli che sono stati gli obiettivi inizialmente prefissati, quali di questi sono stati raggiunti e quali sono stati abbandonati lungo il percorso, per arrivare infine ad una breve ma interessante esposizione su quelli che potrebbero essere i perfezionamenti e le evoluzioni future di questo lavoro. Il primo obiettivo raggiunto è stato sicuramente quello di aver captato la logica e la filosofia che stanno alla base del progetto Fiware, e di aver imparato successivamente, ad utilizzare gli strumenti offerti dalla piattaforma agli sviluppatori. Siamo riusciti nell’intento di realizzare un’applicazione per la piattaforma che rispettasse i requisiti e le

Capitolo 4: Studio delle prestazioni

politiche fondamentali del progetto. Abbiamo visto come, grazie all’uso di tecnologie per sistemi distribuiti, siamo riusciti a rendere scalabile l’applicazione rispetto alle quantità di dati da memorizzare. Siamo riusciti ad interagire con quei moduli della piattaforma che ci hanno consentito di pubblicare i dati raccolti sotto forma di dati “aperti”. Grazie all’uso del paradigma RESTful abbiamo reso l’applicazione facilmente interfacciabile, a tutti i suoi livelli, con applicazioni di terze parti, garantendo simultaneamente le peculiarità di riusabilità ed estensibilità del sistema. Purtroppo, come già anticipato nei capitoli precedenti, un obiettivo che non siamo riusciti a conseguire, ma che può sicuramente essere annoverato tra i possibili sviluppi futuri, è quello della mancata realizzazione di una reale installazione del sistema di monitoraggio fisico (sensori e gateway) all’interno di un autoveicolo. Nei capitoli precedenti abbiamo elencato i diversi problemi che ci hanno impedito il conseguimento di tale obiettivo nel breve periodo, tra cui i più importanti relativi al problema della sicurezza nell’utilizzo dell’adattatore OBD II, all’utilizzo di moduli UMTS per l’invio dei dati sulla rete e alla non così immediata installazione dei vari dispositivi necessari all’interno del veicolo. Oltre a tutti questi problemi rimasti in sospeso, possiamo sbizzarrirci con l’immaginazione se volessimo pensare ai possibili sviluppi futuri per questo lavoro. Si potrebbe pensare, banalmente, di ingrandire il sistema di monitoraggio ambientale attraverso l’uso di ulteriori sensori quali, sensori di umidità, sensori per la rilevazione di CO₂ e altri gas nocivi, sensori crepuscolari o di visibilità. Si potrebbe pensare di fornire attraverso il nostro sistema diversi altri tipi di servizi, quali ad esempio: la mappatura delle buche sul suolo cittadino tramite l’uso di accelerometri, la prenotazione di uno specifico taxi in base alla vicinanza rispetto al punto in cui si trova l’utente, la raccolta di dati sul traffico attraverso lo studio della velocità del veicolo durante i suoi spostamenti e così via. Si potrebbe pensare, inoltre, di rendere il sistema ancora più scalabile e reattivo all’aumentare del traffico dati,

Capitolo 4: Studio delle prestazioni

semplicemente prevedendo l'uso di una federazione di istanze Orion, che possano aumentare o diminuire in base al numero di utenti e di servizi offerti, magari dedicando un'istanza ad ogni livello del sistema o addirittura un'istanza per ogni servizio offerto. Ciò consentirebbe ad applicazioni e sistemi terzi di interagire o meglio riutilizzare i dati ed i servizi offerti dal nostro sistema per i propri scopi, e contribuire quindi al miglioramento della piattaforma e dello stile di vita degli utenti partecipanti. Sarebbero innumerevoli gli ulteriori miglioramenti, le idee innovative ed i nuovi servizi che potremmo elencare relativamente a questo lavoro, ma preferiamo non proseguire oltre e lasciare lo spazio al lettore di dare un po' di sfogo alla propria immaginazione.

Appendice A

Codice Python per lo studio delle prestazioni del nostro sistema

A.1 Taxi_emul_toBenchOrionFederation.py

```
import sys
import time
import threading
import random
import os
import pdb

NUM_ARG=len(sys.argv)
COMMAND=sys.argv[0]

if NUM_ARG==5:
    NUM_ORION_FEDERATION=sys.argv[1]
    NUM_TAXI=sys.argv[2]
    NUM_UPDATE=sys.argv[3]
    TIME_TO_UPDATE=sys.argv[4]

else:
    print 'Usage: '+COMMAND+' [NUM_ORION_FEDERATION] '
    [NUM_TAXI] [NUM_UPDATE] [TIME_TO_UPDATE] '
    print
    sys.exit(2)

#print "Numero TAXI: "+NUM_TAXI + "\nIntervallo di aggiornamento: "+ TIME_TO_UPDATE + " [sec]"

#creo la Dir principale del processo
main_dir =
NUM_ORION_FEDERATION+"Orion_"+NUM_TAXI+"Taxi_"+NUM_UPDATE
+"update_ogni_"+TIME_TO_UPDATE+"sec"
os.system("mkdir "+main_dir)

#inizializzo l'array bidimensionale per il calcolo della media dei ritardi
#delle richieste ad ogni passo di aggiornamento e per ogni thread(Taxi)
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
# [NUM_UPDATE] [NUM_TAXI]
mean_update_array = [[0 for x in range(int(NUM_TAXI))] for x in range(int(NUM_UPDATE))]
mean_query_array = [[0 for x in range(int(NUM_TAXI))] for x in range(int(NUM_UPDATE))]

def taxi_behavior(thread_id):

    try:
        global coords
        global main_dir
        global mean_update_array
        global mean_query_array

        thread_dir = main_dir+"/"+str(thread_id)

        #creo la dir del thread
        os.system("mkdir "+thread_dir)

        ts = str(time.time())
        tokens = ts.split('.')
        #pdb.set_trace()
        ENTITY_ID = tokens[0] + tokens[1]
        ENTITY_ID = str(int(ENTITY_ID) +
int(thread_id))
        ENTITY_TYPE = "observations"
        POSITION =
coords[1]#coords[random.randint(0,len(coords)-1)]
        tokens_pos = POSITION.split()
        POSITION = tokens[0] + '\\ ' + tokens[1]

        #creo gli id per le tre entita' da ENTITY_ID
        generale del server
        id_temp = "temperature_" + ENTITY_ID
        id_pres = "pressure_" + ENTITY_ID
        id_pm10 = "pm10_" + ENTITY_ID

        #Creo il file dove salvare i ritardi delle
CreateEntity
        file_create =
open(thread_dir+"/"+ENTITY_ID+"_create_"+str(thread_id),
"a+")
        file_create.write("#Timestamp,delay\n")

        #Creo il file dove salvare i ritardi delle
Subscription
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
file_sub =
open(thread_dir+"/"+ENTITY_ID+"_sub_"+str(thread_id),
      "a+")
file_sub.write("#Timestamp,delay\n")

#pdb.set_trace()
#fase di creazione delle entita'
create_temp = "python2.7 createEntity.py" + " "
+ id_temp + " " + ENTITY_TYPE + " " + "position " +
"coords " + POSITION #+ " &
#os.system(create_temp)
create_temp = "python2.7 createEntity.py" + " "
+ id_temp + " " + ENTITY_TYPE + " " + "observed_property"
" + "temperature " + "NaN" #+ " &
start_temp = time.time()*1000
os.system(create_temp)
sample_temp = (time.time()*1000) - start_temp
create_temp = "python2.7 createEntity.py" + " "
+ id_temp + " " + ENTITY_TYPE + " " + "Service2 " + "text
" + "on" #+ " &
#os.system(create_temp)

create_press = "python2.7 createEntity.py" + " "
+ id_pres + " " + ENTITY_TYPE + " " + "position " +
"coords " + POSITION #+ " &
#os.system(create_press)
create_press = "python2.7 createEntity.py" + " "
+ id_pres + " " + ENTITY_TYPE + " " +
"observed_property " + "pressure " + "NaN" #+ " &
start_pres = time.time()*1000
os.system(create_press)
sample_pres = (time.time()*1000) - start_pres
create_press = "python2.7 createEntity.py" + " "
+ id_pres + " " + ENTITY_TYPE + " " + "Service3 " +
"text " + "on" #+ " &
#os.system(create_press)

create_pm10 = "python2.7 createEntity.py" + " "
+ id_pm10 + " " + ENTITY_TYPE + " " + "position " +
"coords " + POSITION #+ " &
#os.system(create_pm10)
create_pm10 = "python2.7 createEntity.py" + " "
+ id_pm10 + " " + ENTITY_TYPE + " " + "observed_property"
" + "PM10 " + "NaN" #+ " &
start_pm10 = time.time()*1000
os.system(create_pm10)
sample_pm10 = (time.time()*1000) - start_pm10
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
        create_temp = "python2.7 createEntity.py" + " "
+ id_pm10 + " " + ENTITY_TYPE + " " + "Service1" + "text
" + "on" #+ " &
        #os.system(create_temp)

        #carico i ritardi delle createEntity nel file
        file_create.write(str(long(start_temp)) + "," +
str(sample_temp)+"\n")
        file_create.write(str(long(start_pres)) + "," +
str(sample_pres)+"\n")
        file_create.write(str(long(start_pm10)) + "," +
str(sample_pm10)+"\n")
        file_create.flush()

#sottoscrizioni a tutte e tre le entita'

        #sub_temp = "python2.7 setSubscription.py" + "
" + id_temp + " " + "observed_property" +
"http://130.206.85.25:7777/accumulate"
        #sub_press = "python2.7 setSubscription.py" + "
" + id_pres + " " + "observed_property" +
"http://130.206.85.25:7777/accumulate"
        #sub_pm10 = "python2.7 setSubscription.py" + "
" + id_pm10 + " " + "observed_property" +
"http://130.206.85.25:7777/accumulate"

        sub_temp = "python2.7 setSubscription.py" + " "
+ id_temp + " " + "observed_property" +
"http://fakeurl.omp"
        sub_press = "python2.7 setSubscription.py" + " "
+ id_pres + " " + "observed_property" +
"http://fakeurl.omp"
        sub_pm10 = "python2.7 setSubscription.py" + " "
+ id_pm10 + " " + "observed_property" +
"http://fakeurl.omp"

        start_temp = time.time()*1000
os.system(sub_temp)
sample_temp = (time.time()*1000) - start_temp
start_press = time.time()*1000
os.system(sub_press)
sample_press = (time.time()*1000) - start_press
start_pm10 = time.time()*1000
os.system(sub_pm10)
sample_pm10 = (time.time()*1000) - start_pm10

#carico i ritardi delle subscription nel file
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
        file_sub.write(str(long(start_temp)) + "," +
str(sample_temp)+"\n")
        file_sub.write(str(long(start_pres)) + "," +
str(sample_pres)+"\n")
        file_sub.write(str(long(start_pm10)) + "," +
str(sample_pm10)+"\n")
        file_sub.flush()

        #sottoscrizioni all'orion federato
        sub_temp = "python2.7 setSubscription.py" + " "
+ id_temp + " " + "observed_property" +
"http://192.168.0.241:1026/ngsi10/notifyContext"
        sub_press = "python2.7 setSubscription.py" + " "
+ id_pres + " " + "observed_property" +
"http://192.168.0.241:1026/ngsi10/notifyContext"
        sub_pm10 = "python2.7 setSubscription.py" + " "
+ id_pm10 + " " + "observed_property" +
"http://192.168.0.241:1026/ngsi10/notifyContext"

        start_temp = time.time()*1000
        os.system(sub_temp)
        sample_temp = (time.time()*1000) - start_temp
        start_press = time.time()*1000
        os.system(sub_press)
        sample_press = (time.time()*1000) - start_press
        start_pm10 = time.time()*1000
        os.system(sub_pm10)
        sample_pm10 = (time.time()*1000) - start_pm10

        #carico i ritardi delle subscription per la
federazione nel file
        file_sub.write(str(long(start_temp)) + "," +
str(sample_temp)+"\n")
        file_sub.write(str(long(start_pres)) + "," +
str(sample_pres)+"\n")
        file_sub.write(str(long(start_pm10)) + "," +
str(sample_pm10)+"\n")
        file_sub.flush()

        #Attendo 10 secondi prima di iniziare la fase
di update
        #Creo il file dove salvare i ritardi delle
Update
        file_update =
open(thread_dir+"/"+ENTITY_ID+"_update_"+str(thread_id),
"a+")
        file_update.write("#Timestamp,delay\n")
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
#Creo il file dove salvare i ritardi delle
Query
    file_query =
open(thread_dir+"/"+ENTITY_ID+"_query_"+str(thread_id),
      "a+")
    file_query.write("#Timestamp,delay\n")

    for j in range(int(NUM_UPDATE)):
        #aggiorno il valore di temperatura
        value = random.randint(-300,300)
        parameters = "python2.7
UpdateEntityAttribute.py" + " " + id_temp + " " +
ENTITY_TYPE + " observed_property temperature " +
str(value)
        ts_temp = time.time()
        #pdb.set_trace()
        start_temp = time.time()*1000
        #print str(start_temp)
        os.system(str(parameters))
        sample_temp = (time.time()*1000) -
start_temp

        #aggiorno il valore di pressione
        value = random.randint(0,300)
        parameters = "python2.7
UpdateEntityAttribute.py" + " " + id_pres + " " +
ENTITY_TYPE + " observed_property pressure " + str(value)
        ts_pres = time.time()
        start_pres = time.time()*1000
        os.system(str(parameters))
        sample_pres = (time.time()*1000) -
start_pres

        #aggiorno il valore di PM10
        value = random.randint(0,10000)
        parameters = "python2.7
UpdateEntityAttribute.py" + " " + id_pm10 + " " +
ENTITY_TYPE + " observed_property PM10 " + str(value)
        ts_pm = time.time()
        start_pm = time.time()*1000
        os.system(str(parameters))
        sample_pm = (time.time()*1000) - start_pm

        #carico i ritardi delle update nel file
        file_update.write(str(long(start_temp)) +
"," + str(sample_temp)+"\n")
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
        file_update.write(str(long(start_pres)) +  
", "+ str(sample_pres)+"\n")  
        file_update.write(str(long(start_pm)) +  
", "+ str(sample_pm)+"\n")  
        file_update.flush()  
  
        #faccio la media dei ritardi delle update  
per le 3 entita'  
        mean_update = (sample_temp + sample_pres  
+ sample_pm)/3  
        mean_update_array[j][thread_id] =  
mean_update  
  
        #faccio le query dividendole tra i due  
orion federati  
        #se il passo di iterazione e' pari quero  
il primo orion (port:1026)  
        #se no quero il secondo (port:1027)  
        query_temp = ""  
        query_pres = ""  
        query_pm10 = ""  
  
        """  
        if (j%2)==0:  
            query_temp = '(curl  
localhost:1026/v1/queryContext -s -S --header \'Content-  
Type: application/xml\' -d @- | xmllint --format -)  
<<EOF\n<?xml version="1.0" encoding="UTF-  
8"?>\n<queryContextRequest>\n\t<entityIdList>\n\t\t<entityId type="observations"  
isPattern="false">\n\t\t<id>' +id_temp+ '</id>\n\t\t</ent  
ityId>\n\t</entityIdList>\n\t<attributeList/>\n</queryContextRequest>\nEOF'  
            query_pres = '(curl  
localhost:1026/v1/queryContext -s -S --header \'Content-  
Type: application/xml\' -d @- | xmllint --format -)  
<<EOF\n<?xml version="1.0" encoding="UTF-  
8"?>\n<queryContextRequest>\n\t<entityIdList>\n\t\t<entityId type="observations"  
isPattern="false">\n\t\t<id>' +id_pres+ '</id>\n\t\t</ent  
ityId>\n\t</entityIdList>\n\t<attributeList/>\n</queryContextRequest>\nEOF'  
            query_pm10 = '(curl  
localhost:1026/v1/queryContext -s -S --header \'Content-  
Type: application/xml\' -d @- | xmllint --format -)  
<<EOF\n<?xml version="1.0" encoding="UTF-  
8"?>\n<queryContextRequest>\n\t<entityIdList>\n\t\t<entityId type="observations"
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
isPattern="false">\n\t\t\t<id>' + id_pm10 + '</id>\n\t\t</entityId>\n\t</entityIdList>\n\t<attributeList/>\n</queryContextRequest>\nEOF'
else:
"""
query_temp = '(curl
192.168.0.241:1026/ngsi10/queryContext -s -S --header
\'Content-Type: application/xml\' -d @- | xmllint --
format -) <<EOF\n<?xml version="1.0" encoding="UTF-
8"?>\n<queryContextRequest>\n\t<entityIdList>\n\t\t<entityId type="observations"
isPattern="false">\n\t\t\t<id>' + id_temp + '</id>\n\t\t</entityId>\n\t</entityIdList>\n\t<attributeList/>\n</queryContextRequest>\nEOF'
query_pres = '(curl
192.168.0.241:1026/ngsi10/queryContext -s -S --header
\'Content-Type: application/xml\' -d @- | xmllint --
format -) <<EOF\n<?xml version="1.0" encoding="UTF-
8"?>\n<queryContextRequest>\n\t<entityIdList>\n\t\t<entityId type="observations"
isPattern="false">\n\t\t\t<id>' + id_pres + '</id>\n\t\t</entityId>\n\t</entityIdList>\n\t<attributeList/>\n</queryContextRequest>\nEOF'
query_pm10 = '(curl
192.168.0.241:1026/ngsi10/queryContext -s -S --header
\'Content-Type: application/xml\' -d @- | xmllint --
format -) <<EOF\n<?xml version="1.0" encoding="UTF-
8"?>\n<queryContextRequest>\n\t<entityIdList>\n\t\t<entityId type="observations"
isPattern="false">\n\t\t\t<id>' + id_pm10 + '</id>\n\t\t</entityId>\n\t</entityIdList>\n\t<attributeList/>\n</queryContextRequest>\nEOF'

#pdb.set_trace()
start_temp = time.time()*1000
os.system(query_temp)
sample_temp = (time.time()*1000) -
start_temp
start_pres = time.time()*1000
os.system(query_pres)
sample_pres = (time.time()*1000) -
start_pres
start_pm10 = time.time()*1000
os.system(query_pm10)
sample_pm10 = (time.time()*1000) -
start_pm10

#carico i ritardi delle query nel file
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
        file_query.write(str(long(start_temp)) +
", "+ str(sample_temp)+"\n")
        file_query.write(str(long(start_pres)) +
", "+ str(sample_pres)+"\n")
        file_query.write(str(long(start_pm10)) +
", "+ str(sample_pm10)+"\n")
        file_query.flush()

    #faccio la media dei ritardi delle query
    per le 3 entita'
    mean_query = (sample_temp + sample_pres +
sample_pm10)/3
    mean_query_array[j][thread_id] =
mean_query

    time.sleep(float(TIME_TO_UPDATE))

    #creo il file contenente tutte le update
    requests
    #file_update_all =
open(main_dir+"/_update_all", "a+")
    #os.system()

except KeyboardInterrupt:
    sys.exit(2)

threads = []
for i in range(int(NUM_TAXI)):
    t = threading.Thread(target=taxi_behavior,
args=(i,))
    threads.append(t)
    t.start()

#attendo che tutti i thread finiscano la loro esecuzione
[x.join() for x in threads]

#creo i files contenenti tutte le update e le query
requests
#file_update_all = open(main_dir+"/_update_all", "a+")
#file_query_all = open(main_dir+"/query_all", "a+")

for k in range(int(NUM_TAXI)):
    temp_path_file = main_dir+"/"+str(k)+"/"
    #concateno gli update_file e i query_file
    os.system("cat "+temp_path_file+"*update* >>
"+main_dir+"/update_all")
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
os.system("cat "+temp_path_file+"*query* >>
"+main_dir+"/query_all")

#ordino i due file in base al timestamp
os.system("sort "+main_dir+"/update_all >
"+main_dir+"/update_all_sort")
os.system("sort "+main_dir+"/query_all >
"+main_dir+"/query_all_sort")

#creo i files contenenti le medie delle update e query
request per ogni passo di aggiornamento
mean_update_file = open(main_dir+"/update_mean", "a+")
mean_query_file = open(main_dir+"/query_mean", "a+")

for i in range(int(NUM_UPDATE)):
    mean_up_i =
    (sum(mean_update_array[i]))/int(NUM TAXI)
    mean_qu_i = (sum(mean_query_array[i]))/int(NUM TAXI)
    mean_update_file.write(str(i*int(TIME_TO_UPDATE))+",",
    "+str(mean_up_i)+"\n")
    mean_query_file.write(str(i*int(TIME_TO_UPDATE))+",",
    "+str(mean_qu_i)+"\n")
    mean_update_file.flush()
    mean_query_file.flush()
```

A.2 taxi_emulator.py

```
import time
import threading
import requests
import random

coords = ["'38.175474, 15.546313'", "'38.204752,
15.552321'", "'38.193892, 15.541850'", "'38.216149,
15.536957'", "'38.184650, 15.552321'", "'38.193555,
15.570603'", "'38.190519, 15.548287'", "'38.202729,
15.553780'", "'38.198682, 15.554982'", "'38.219251,
15.558673'", "'38.201717, 15.556527'", "'38.173079,
15.543566'", "'38.188192, 15.556098'", "'38.206404,
15.557471'", "'38.170381, 15.541984'", "'38.228524,
15.567047'", "'38.225001, 15.549966'", "'38.238923,
15.575158']"]

taxi_id = "Taxi1"
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
rand_index = random.randint(0,len(coords)-1)      # Calcolo
random dell'indice della coordinata iniziale
current_point = coords[rand_index]
old_point = current_point
move = True                                         # Stato del TAXI
(true --> in movimento, false --> fermo)

# Thread che simula il cambio delle coordinate del TAXI
def change_coords():
    global rand_index
    global current_point
    while 1:
        time_to_move = random.randint(1000,1500)
        #tempo di cambio della coordinata
        time.sleep(time_to_move)
        rand_index = (rand_index + 1) % len(coords)

        current_point = coords[rand_index]
        #nuova coordinata corrente
        print "Cambio Coordinata: " + current_point +
"\nTempo passato per il cambio della coordinata: "+
str(time_to_move) + "\n"

# Thread che controlla lo stato del TAXI e il cambio
della coordinata per calcolare il nuovo stato del TAXI

def check_state():
    global move
    global current_point
    global old_point

    cont = 0
    while 1:
        if old_point != current_point:
            if move == False:
                move = True
                old_point = current_point
                cont = 0
                #stop sensoristica
                r =
requests.get('http://localhost:1234/stop/observation')
                print "Sto partendo: invio lo stop
per la sensoristica\n"
            else:
                old_point = current_point
```

Appendice A: Codice Python per lo studio delle prestazioni del nostro sistema

```
        cont = 0
        print "Sono in movimento\n"

    else:
        if move == False:
            print "Sono fermo\n"
        else:
            cont = cont + 1
            print "Ero in movimento e sono fermo
da " + str(15*cont) + " secondi\n"
            if cont >= 6:
                move = False
                #start sensoristica
                ts = str(time.time())
                tokens = ts.split('.')
                ts = tokens[0] + tokens[1]
                r =
requests.get('http://localhost:1234/start/observation?id=
'+ ts +'&type=observations&position='+current_point)
                print "Sono fermo da piu' di 90
secondi: invio lo start per la sensoristica\n"+
"id_observation=" + ts + "\ncurrent_position="+
current_point +"\\n"

                time.sleep(15)                                #ogni 15
secondi controllo se sono cambiate le coordinate
                print "CONTROLLO STATO:\\n"

t1 = threading.Thread(target=change_coords)
t2 = threading.Thread(target=check_state)
t1.start()
t2.start()
```

Appendice B

Script Matlab

B.1

diff_singolo_orion_5_10_20_30_utenti_all_request.m

```
singolo_cinque='./1Orion5ThServerGioNoSleep/all_request_s
ort';
singolo_dieci='./1Orion10ThServerGioNoSleepUltimo/all_req
uest_sort';
singolo_venti='./1Orion20ThServerGioNoSleep/all_request_s
ort';
singolo_trenta='./1Orion30ThServerGioNoSleep/all_request_
sort';

fs_cinque = fopen(singolo_cinque);
fs_dieci = fopen(singolo_dieci);
fs_venti = fopen(singolo_venti);
fs_trenta = fopen(singolo_trenta);

s_5 = textscan(fs_cinque, '%d%f', 'delimiter', ',');
s_10 = textscan(fs_dieci, '%d%f', 'delimiter', ',');
s_20 = textscan(fs_venti, '%d%f', 'delimiter', ',');
s_30 = textscan(fs_trenta, '%d%f', 'delimiter', ',');

fclose(fs_cinque);
fclose(fs_dieci);
fclose(fs_venti);
fclose(fs_trenta);

delay_s5=s_5{2};
delay_s10=s_10{2};
delay_s20=s_20{2};
delay_s30=s_30{2};

figure('name','All Request Delay Response');
hold on;
grid on;

plot(delay_s5, 'b','LineWidth',2);
plot(delay_s10, 'r','LineWidth',2);
plot(delay_s20,'y','LineWidth',2); %yellow
plot(delay_s30, 'g','LineWidth',2); %green
```

Appendice B: Script Matlab

```
legend('5 Utenti','10 Utenti','20 Utenti','30 Utenti');
title('All Request Delay Response');
xlabel('index of samples');
ylabel('delay of the response in msec');

hold off;
```

B.2 diff_query_update.m

```
singolo_cinque='./2Orion30ThServerGioNoSleep/query_all_so
rt';
doppio_cinque='./2Orion30ThServerGioNoSleep/update_all_so
rt';

fs_cinque = fopen(singolo_cinque);
fd_cinque = fopen(doppio_cinque);

s_5 = textscan(fs_cinque, '%d%f', 'delimiter', ',');
d_5 = textscan(fd_cinque, '%d%f', 'delimiter', ',');

fclose(fs_cinque);
fclose(fd_cinque);

delay_s5=s_5{2};
delay_d5=d_5{2};

figure('name','Update and Query delay 30 users');
hold on;
grid on;

plot(delay_s5, 'g*');
plot(delay_d5, 'r*');

title('Update and Query delay 30 users')
legend('Query','Update');
xlabel('index of samples');
ylabel('delay of the response in msec');

hold off;
```

B.3 diff_singolo_doppio_orion_query_mean_totale.m

```

singolo_cinque='./1Orion5ThServerGioNoSleep/query_mean';
doppio_cinque='./2Orion5ThServerGioNoSleep/query_mean';
singolo_dieci='./1Orion10ThServerGioNoSleepUltimo/query_mean';
doppio_dieci='./2Orion10ThServerGioNoSleep/query_mean';
singolo_venti='./1Orion20ThServerGioNoSleep/query_mean';
doppio_venti='./2Orion20ThServerGioNoSleep/query_mean';
singolo_trenta='./1Orion30ThServerGioNoSleep/query_mean';
doppio_trenta='./2Orion30ThServerGioNoSleep/query_mean';

fs_cinque = fopen(singolo_cinque);
fd_cinque = fopen(doppio_cinque);
fs_dieci = fopen(singolo_dieci);
fd_dieci = fopen(doppio_dieci);
fs_venti = fopen(singolo_venti);
fd_venti = fopen(doppio_venti);
fs_trenta = fopen(singolo_trenta);
fd_trenta = fopen(doppio_trenta);

s_5 = textscan(fs_cinque, '%d%f', 'delimiter', ',');
d_5 = textscan(fd_cinque, '%d%f', 'delimiter', ',');
s_10 = textscan(fs_dieci, '%d%f', 'delimiter', ',');
d_10 = textscan(fd_dieci, '%d%f', 'delimiter', ',');
s_20 = textscan(fs_venti, '%d%f', 'delimiter', ',');
d_20 = textscan(fd_venti, '%d%f', 'delimiter', ',');
s_30 = textscan(fs_trenta, '%d%f', 'delimiter', ',');
d_30 = textscan(fd_trenta, '%d%f', 'delimiter', ',');

fclose(fs_cinque);
fclose(fd_cinque);
fclose(fs_dieci);
fclose(fd_dieci);
fclose(fs_venti);
fclose(fd_venti);
fclose(fs_trenta);
fclose(fd_trenta);

delay_s5=s_5{2};
delay_d5=d_5{2};
delay_s10=s_10{2};
delay_d10=d_10{2};
delay_s20=s_20{2};
delay_d20=d_20{2};
delay_s30=s_30{2};
delay_d30=d_30{2};

```

Appendice B: Script Matlab

```
figure('name','Query Mean Delay Response');
subplot(2,2,1);
hold on;
grid on;

plot(delay_s5, 'b','LineWidth',2);
plot(delay_d5, 'r','LineWidth',2);

legend('1Orion','2Orion');
title('5 Utenti');
xlabel('index of samples');
ylabel('delay of the response in msec');

subplot(2,2,2);
hold on;
grid on;
plot(delay_s10, 'b','LineWidth',2);
plot(delay_d10, 'r','LineWidth',2);

legend('1Orion','2Orion');
title('10 Utenti');
xlabel('index of samples');
ylabel('delay of the response in msec');

subplot(2,2,3);
hold on;
grid on;
plot(delay_s20, 'b','LineWidth',2);
plot(delay_d20, 'r','LineWidth',2);

legend('1Orion','2Orion');
title('20 Utenti');
xlabel('index of samples');
ylabel('delay of the response in msec');

subplot(2,2,4);
hold on;
grid on;
plot(delay_s30, 'b','LineWidth',2);
plot(delay_d30, 'r','LineWidth',2);

legend('1Orion','2Orion');
title('30 Utenti');
xlabel('index of samples');
ylabel('delay of the response in msec');
```

Appendice B: Script Matlab

```
hold off;
```

Bibliografia e Sitografia

[1] Overall Fi-ware Vision -

http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Overall_FI-WARE_Vision - (consultato il 18.02.2015).

[2] Fi-ware Architecture -

http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FI-WARE_Architecture - (consultato il 18.02.2015).

[3] Data/Context Management Architecture -

http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Data/Context_Management_Architecture - (consultato il 18.02.2015).

[4] Fiware.ArchitectureDescription.Data.ContextBroker -

<http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.ArchitectureDescription.Data.ContextBroker> - (consultato il 19.02.2015).

[5] Fiware.ArchitectureDescription.Data.BigData -

<http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.ArchitectureDescription.Data.BigData> - (consultato il 19.02.2015).

[6] Orion Context Broker – User and Programmers Guide -

https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Publish/Subscribe_Broker_-_Orion_Context_Broker_-_User_and_Programmers_Guide - (consultato il 19.02.2015).

[7] BigData Analysis – User and Programmers Guide -

http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/BigData_Analysis_-_User_and_Programmer_Guide - (consultato il 19.02.2015).

[8] NGSI-9/NGSI-10 information model -

http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/NGSI-9/NGSI-10_information_model - (consultato il 19.02.2015).

[9] Fiware NGSI-10 Open RESTful API Specification -

http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FI-WARE_NGSI-10_Open_RESTful_API_Specification - (consultato il 19.02.2015).

Bibliografia e Sitografia

WARE_NGSI-10_Open_RESTful_API_Specification - (consultato il 19.02.2015).

[10] S. Monk, *Raspberry Pi Cookbook. Software and Hardware problems and solutions*. Sebastopol, O'Reilly Media, 2013.

[11] T. Cox, *Raspberry Pi Cookbook for Python Programmers*, Birmingham, Packt Publishing Ltd, 2014.

[12] Raspberry Pi Documentation -
<http://www.raspberrypi.org/documentation/> - (consultato il 24.02.2015).

[13] Wiki Linino - <http://wiki.linino.org/doku.php> - (consultato il 25.02.2015).

[14] Guide to the Arduino Yun - <http://arduino.cc/en/Guide/ArduinoYun#toc2> - (consultato il 25.02.2015).

[15] D. M. Holstius, A. Pillarisetti, K. R. Smith, e E. Seto, *Field Calibrations of a low-cost aerosol sensor at a regulatory monitoring site in California*, in Atmospheric Measurement Techniques journal, 27 Gennaio 2014.

[16] Air Quality Monitoring -
<http://www.howmuchsnow.com/arduino/airquality/grovedust/> - (consultato il 19.11.2014).

[17] T. Allen, De-construction of the Shinyei PPD42NS dust sensor, in http://takingspace.org/wp-content/uploads/ShinyeiPPD42NS_Deconstruction_TracyAllen.pdf, Maggio 2013.

[18] Particle Sensor model PPD42NS - <http://www.sca-shinyei.com/pdf/PPD42NS.pdf> - (consultato il 9.12.2014)

[19] M. Budde, M. Busse, e M. Beigl, *Investigating the Use of Commodity Dust Sensors for the Embedded Measurement of Particulate Matter*, in Networked Sensing Systems (INSS), Giugno 2012

[20] On-Board Diagnostic II Systems -
<http://www.arb.ca.gov/msprog/obdprog/obdfaqa.htm> - (consultato il 02.03.2015).

Bibliografia e Sitografia

[21] ELM327DS - <http://elmelectronics.com/DSheets/ELM327DS.pdf> -

(consultato il 26.11.2014)

[22] OBD-II adapter (ELM) bluetooth -

<http://www.cartft.com/catalog/PDF/1859/OBD-II%20Adapter.pdf> –

(consultato il 12.12.2014)

[23] M. Summerfield, *Programming in Python 3. A Complete Introduction to the Python Language. Second Edition.* Crawfordsville, Addison-Wesley, 2010