

UNIVERSITÀ DEGLI STUDI DI MESSINA
FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica



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

Tesi di Laurea di:
Antonio CARISTIA

Relatore: Prof. Marco SCARPA
Correlatore: Ing. Francesco LONGO

Anno Accademico 2014–2015

Alla mia famiglia.

Indice

1	Introduzione	1
1.1	Contesto e obiettivi del lavoro	1
1.2	Panoramica generale del progetto	5
1.3	Struttura dati	7
2	La piattaforma Fi-ware	11
2.1	Cos'è Fi-ware	11
2.2	Il Fi-lab	15
2.3	Orion Context Broker	17
2.4	Modello dei dati NGSI	22
2.4.1	updateContext	23
2.4.2	queryContext	25
2.4.3	subscribeContext	27
2.5	Context Broker Federation	31
2.5.1	Push Federation	32
2.5.2	Pull Federation	34
2.6	Introduzione ai big data	39
2.6.1	Scale-up	39
2.6.2	Scale-out	40
2.6.3	Fattori limitanti	40
2.6.4	La dura realtà	41
2.6.5	Di cosa c'è bisogno?	42
2.7	Hadoop	43
2.7.1	HDFS	44
2.7.2	MapReduce	45

2.7.3	HDFS e MapReduce insieme	47
2.7.4	Architettura HDFS e MapReduce	48
2.7.5	Quando usare Hadoop	48
2.7.6	Hive	50
2.8	Cosmos	51
2.8.1	Architettura	53
2.8.2	API	55
2.9	Wirecloud	57
2.10	CKAN	59
2.10.1	Insiemi di dati e risorse	60
2.10.2	Utenti, organizzazioni e credenziali	60
2.10.3	Come si usa	61
3	Implementazione	66
3.1	Descrizione dettagliata del progetto	66
3.1.1	Orion	72
3.1.1.1	Installazione	72
3.1.1.2	Esecuzione e check dell'installazione	74
3.1.2	Rush	75
3.1.3	subscriptionServer.py	77
3.1.4	restServerHive.py	84
3.1.5	Widget	87
3.1.6	ServiceWidget	89
3.1.6.1	init()	91
3.1.6.2	createTable() - createTableOutput()	92
3.1.6.3	retrieveServicesFromServerNGSI()	94
3.1.6.4	handlerReceiveEntity()	97
3.1.6.5	deleteTable()	99
3.1.6.6	updateTable()	99
3.1.7	QueryWidget	100
3.1.7.1	init()	103
3.1.7.2	createTableFooter() - createTableOutPut()	104
3.1.7.3	handlerEntityInput()	106
3.1.7.4	doQuery()	107

3.1.7.5	onQuerySucess()	107
3.1.7.6	deleteTableOutPut()	111
3.1.7.7	updateTableOutPut()	111
3.1.7.8	onQuerySucess1()	112
3.1.8	Wiring e altri widget utilizzati	114
3.1.9	historyHiveToLinearGraph	116
4	Studio delle prestazioni	122
4.1	Scenario e prove effettuate	122
4.2	JMeter	124
4.3	Modellazione dell'applicazione con JMeter	125
4.3.1	Thread Group	127
4.3.2	Once Only Controller	129
4.3.3	Random Variable	129
4.3.4	Loop Controller	130
4.3.5	HTTP Request	132
4.4	Prima serie di test	134
4.5	Seconda serie di test	135
4.6	Seconda serie di test con più ram	139
4.7	Considerazioni finali sui test	141
A	Fogli di stile	142
A.1	service_widget.css	142
A.2	query_widget.css	143
B	script matlab	145
B.1	plot_totale.m	145
B.2	plot_mostra_le_diverse_richieste.m	147

Elenco delle figure

1.1	Livelli del sistema	7
1.2	Struttura dati NGSI	10
2.1	Architettura Fi-ware	13
2.2	Home page del Fi-lab	16
2.3	Sezione Cloud sul Fi-lab	16
2.4	Immagini già pronte sul FI-lab	17
2.5	Modello NGSI	19
2.6	Producer - Broker - Consumer	20
2.7	Comunicazione sincrona	27
2.8	Comunicazione asincrona	28
2.9	Sottoscrizione tra broker	28
2.10	Push Federation	32
2.11	Pull federation	38
2.12	Architettura Hadoop	49
2.13	Architettura Cosmos	55
2.14	Wiring in Wirecloud	58
2.15	Il Marketplace del Fi-lab	59
2.16	CKAN sul Fi-lab	61
2.17	Registrazione di un dataset in CKAN	62
2.18	Aggiunta di una risorsa in CKAN	63
2.19	Completamento della registrazione di un dataset in CKAN .	64
2.20	Pagina di un dataset in CKAN	65
3.1	Il sensore collegato alla scheda	67
3.2	Livello di interfaccia utente	70

3.3	Livello di sensoristica	70
3.4	Livello di backend	71
3.5	Architettura completa del sistema	73
3.6	Rush	75
3.7	Architettura Rush	76
3.8	Connessioni in ascolto	77
3.9	Architettura HiveServer2	85
3.10	ServiceWidget	90
3.11	Interfacce di input/output QueryWidget	101
3.12	QueryWidget	102
3.13	Wiring del mashup	114
3.14	Service Widget	114
3.15	Entità sul MapViewer	115
3.16	Setting History Hive to Linear Graph	116
3.17	Grafico pm10 sul Linear Graph	116
4.1	Prima fase di test	122
4.2	Seconda fase di test	123
4.3	Thread Group	127
4.4	Parametri configurazione Thread Group	128
4.5	Once Only Controller	129
4.6	Random Variable	130
4.7	Loop Controller	130
4.8	Parametri configurazione Random Variable	131
4.9	HTTP Request	132
4.10	Parametri di configurazione HTTP Request	133
4.11	Performance Orion singolo 1-5-10-15-20 utenti	134
4.12	Zoom Orion singolo 1-5-10-15-20 utenti	135
4.13	Federazione singolo thread	136
4.14	Federazione cinque threads	136
4.15	Federazione dieci threads	137
4.16	Federazione tutte le richieste cinque threads	138
4.17	Federazione tutte le richieste dieci threads	139
4.18	Federazione caso dieci e venti thread con più ram	140

ELENCO DELLE FIGURE

vi

4.19 Federazione tutte le richieste 20 threads	140
--	-----

Capitolo 1

Introduzione

1.1 Contesto e obiettivi del lavoro

Il nostro lavoro di tesi è nato dalla congiunta necessità da parte del dipartimento e dell'azienda Arkimedè s.r.l., dove abbiamo svolto l'intero elaborato finale, di acquisire il know how sulle tecnologie offerte dal progetto europeo Fi-ware.

Tutto nasce da uno studio della commissione europea circa le condizioni del mercato ICT nel mondo, i risultati di questa indagine hanno dimostrato una costante crescita del mercato ICT in Asia e negli Stati Uniti, mentre una situazione pressochè ferma nel vecchio continente. Allo stesso tempo si è visto come l'ottanta per cento degli articoli scientifici in questo campo ha autori europei, ma purtroppo solo il dieci per cento di questa massa di produzione scientifica viene brevettata. Per avvicinare il mercato, le aziende e il mondo della ricerca, la commissione europea ha istituito l'European Institute of Technology¹ il cui obiettivo principale è quello di favorire la messa in pratica sul mercato dei risultati della ricerca europea in campo tecnologico.

¹L'EIT è un organismo indipendente dell'Unione Europea istituito nel 2008 per favorire l'innovazione e l'imprenditorialità in tutto il continente. EIT riunisce i principali centri di ricerca e aziende che sviluppano prodotti e servizi innovativi. L'obiettivo di EIT è quello di favorire la collaborazione tra imprenditori e ricercatori, avviare nuove imprese e formare una nuova generazione di imprenditori.

In questo contesto nasce FI-PPP (Future Internet Public Private Partnership): un'iniziativa d'innovazione finanziata dalla comunione europea, guidata dalle principali aziende ICT, il cui scopo è la creazione e messa in produzione di una piattaforma tecnologica per lo sviluppo di applicazioni per l'internet del futuro. Nel 2008 ha avuto inizio la prima fase di questo progetto, che ha coinvolto appunto le principali aziende europee attive in questo settore. Nel 2013 il progetto è entrato nella seconda fase con la messa a disposizione di una infrastruttura cloud su cui è possibile effettuare la sperimentazione di tutta una serie di servizi innovativi.

Lo scopo finale del progetto è molto ambizioso, competere effettivamente sul mercato con aziende leader del calibro di Amazon e Google, fornendo una piattaforma alternativa europea libera da modalità invasive di raccolta di dati sensibili che caratterizzano le soluzioni d'oltreoceano. Favorire lo sviluppo del mercato quindi, fornendo dei servizi già pronti alle piccole aziende ICT per sviluppare applicazioni innovative e allo stesso tempo liberare i dati europei dal controllo americano.

Il progetto ha così generato tre prodotti: **Fi-ware**, **Fi-lab** e **Fi-Ops**.

Fi-ware nasce come progetto all'interno di FI-PPP e possiamo pensarlo come un insieme di tecnologie messe a disposizione agli sviluppatori sia come servizi sia in open source. Queste insieme di tecnologie di base afferiscono a diversi domini (IoT², data context management etc.) che nella terminologia di Fi-ware vengono chiamati capitoli.

Fi-lab è un ambiente o laboratorio dove queste tecnologie vengono messe a disposizione di tutti, inoltre al suo interno vengono forniti i dati in formato open che gli sviluppatori possono usare per costruire le loro applicazioni. Di fatto Fi-lab è un punto di incontro per sviluppatori, aziende e stakeholder.

Fi-Ops è l'infrastruttura su cui queste tecnologie girano: macchine, reti, controllo della qualità del servizio etc.

Nel 2014 ha avuto inizio la fase tre del progetto che mette a disposizione alle piccole e medie imprese cento milioni di euro per sviluppare le proprie

²Con il termine Internet of Things ci si riferisce all'evoluzione della rete internet verso la connessione di oggetti, sensori e dispositivi intelligenti in grado di comunicare tra di loro attraverso la rete, generare dati e fornire nuovi servizi prima inimmaginabili. I principali campi di applicazioni sono molteplici: Domotica, Robotica, Smart City etc.

applicazioni, con però il vincolo di utilizzare i servizi offerti dal progetto Fi-ware.

Arkimedè s.r.l., società con la quale collaboriamo, ha deciso di partecipare ai bandi di gara per questi finanziamenti; il nostro lavoro di tesi è il frutto di questa partecipazione. In questi mesi abbiamo progettato e sviluppato un'applicazione per il monitoraggio ambientale utilizzando le tecnologie sopra citate. Arkimedè s.r.l. ha sviluppato negli anni un sistema per la gestione della flotta taxi dell'intera città di Messina, in particolare su ciascuna autovettura Arkimedè ha installato un tablet attraverso il quale il server interagisce direttamente con il tassista per informarlo sulle corse ad esso assegnate. La disponibilità di un dispositivo su ciascuna autovettura ha permesso ad Arkimedè di specializzarsi nella raccolta di dati molto utili in ambito Smart City, come il traffico cittadino in varie fasce orarie, le condizioni del manto stradale o la copertura della rete dati nelle varie zone della città.

I dati a disposizione potrebbero sensibilmente migliorare la vita dei cittadini, fornendo ad esempio all'automobilista un resoconto in tempo reale del traffico piuttosto che suggerimenti sul percorso verso la destinazione o addirittura prevedere attraverso calcoli statistici i periodi o le zone più congestionate. Questi dati potrebbero essere di utilità non solo per il privato ma anche per l'amministrazione pubblica allo scopo di migliorare la viabilità cittadina. Nell'ottica dunque di sviluppare dei servizi di smart mobility volti al miglioramento della qualità di vita della comunità, Arkimedè ci ha chiesto di realizzare un sistema di monitoraggio ambientale che sfruttasse per effettuare le misurazioni le autovetture a propria disposizione. Brevemente possiamo dire che il sistema ha le seguenti caratteristiche:

- Il sistema è **economico**: è stato infatti realizzato con hardware a basso costo. Per effettuare le misurazioni sono stati scelti dei sistemi embedded facilmente reperibili sul mercato, dai costi molto contenuti che allo stesso tempo garantiscono buona capacità di elaborazione. Anche la scelta della sensoristica è stata rivolta a mantenere bassi i costi, i sensori di particolato sono purtroppo molto costosi. Il nostro approccio è stato quello di risparmiare sulla qualità dei sensori,

pensando di migliorare la precisione della misurazione in un secondo momento attraverso una post elaborazione dei dati. Studi in letteratura [1, 2][1, 2]hanno dimostrato come un elevato numero di sensori a basso costo, con opportune elaborazioni dei dati[3][3], possano garantire delle misurazioni paragonabili a quelle di un singolo sensore di alta qualità molto più costoso.

- Il sistema è **scalabile**: la memorizzazione dei dati è distribuita attraverso filesystem HDFS³ permettendo così lo storing di grosse moli di dati. Quando i dati sono tanti, l'uso di un sistema di memorizzazione centralizzato può rappresentare il collo di bottiglia dell'intera infrastruttura. Inoltre, la distribuzione dei dati in un cluster di macchine, permette operazioni di post elaborazione distribuite molto efficaci e ampiamente utilizzate in ambito big data. Infine, i tempi di risposta del server di comunicazione sono stati studiati sotto diverse condizioni di carico, ottenendo anche in situazioni di stress buoni tempi di risposta.
- Il sistema sfrutta il paradigma RESTful⁴, ciò garantisce un elevato livello di **interoperabilità** tra le varie parti. Il server che permette la comunicazione e lo smistamento delle informazioni tra i livelli è di fatto un web service che fornisce la proprie funzionalità attraverso chiamate rest su protocollo http. Questa scelta lascia massima libertà sul linguaggio di programmazione da utilizzare nell'implementazione di un'applicazione di terze parti che volesse interagire con il sistema; non si è legati a nessuna libreria di uno specifico linguaggio.

³Hadoop Distributed File System è un file system distribuito altamente scalabile. Ogni nodo del cluster è detto DataNode, i dati sono distribuiti e replicati in blocchi di 64MB sui vari DataNode. Esiste un nodo centrale, HeadNode, che si occupa di tenere traccia delle posizioni sui DataNode dei blocchi di un file o della gestione degli errori sul cluster.

⁴Rest è un insieme di linee guida per l'implementazione di web services, i principi di questo paradigma sono: identificazione delle risorse (URI), utilizzo del protocollo http per la comunicazione, comunicazione senza stato, risorse autodescrittive e collegamenti tra risorse tramite link.

- Il sistema è in grado di pubblicare i dati raccolti in maniera automatica in formato **open data**⁵ sullo store del Fi-lab. Il nostro sistema doveva infatti prevedere un metodo semplice per l'accesso ai dati da parte dei programmati che avessero intenzione di sviluppare applicazioni di tipo ambientale. CKAN⁶ e il formato open data sono stati utilizzati proprio a questo scopo. Il vantaggio nell'uso di CKAN è rappresentato dalla possibilità di accedere ai dati delle misurazioni attraverso semplici api. Sviluppatori di terze parti possono dunque utilizzare il sistema per le proprie applicazioni come sorgente di dati ambientali.
- Il sistema fornisce un' **interfaccia grafica** per la consultazione delle misurazioni. I dati non sono accessibili solamente in formato open data per l'elaborazione automatica; attraverso un sistema innovativo di nome Wirecloud⁷ i dati sono consultabili dagli utenti direttamente dal browser.
- Il sistema prevede la possibilità di **aggiungere ulteriori servizi**, permettendo all'utente di effettuare il discovery dei nuovi servizi e delle rispettive informazioni.

1.2 Panoramica generale del progetto

In questo paragrafo proveremo a fornire una descrizione ad alto livello del sistema realizzato riservando i capitoli successivi per una vista più dettagliata delle soluzioni implementative e del codice. Molto in generale possiamo dire che il sistema prevede l'installazione sulle autovetture di dispositivi embedded con sensori appositi per la raccolta di dati di temperatura, pressione e

⁵Con il termine open data si intende riferirsi ad una modalità libera di pubblicazione e diffusione dei dati su internet. Secondo questa modalità i dati sono pubblicati sul web senza restrizioni di copyright che ne limitano la riproduzione o l'accesso. Più rigorosamente l'Open Knowledge Foundation definisce un dato open se chiunque è in grado di utilizzarlo, riutilizzarlo e ridistribuirlo con l'unico vincolo di condividere il dato allo stesso modo.

⁶Consultare il paragrafo 2.10 a pagina 59 per informazioni su questo software di pubblicazione dei dati

⁷Consultare il paragrafo 2.9 a pagina 57 per un' introduzione al software

particolato. Le autovetture quindi devono essere pensate come dei sensori mobili in giro per la città, durante le soste dei tassisti il sistema si attiva iniziando a raccogliere i dati d'interesse. I dati raccolti vengono inviati ad un server che ha il compito di smistare le informazioni in due direzioni: o verso il sistema di storing distribuito o verso l'interfaccia grafica per la consultazione da parte dell'utente.

Il sistema è diviso in tre livelli, rispettivamente dal basso verso l'alto:

- Il livello di **sensoristica**: appartengono a questo livello i dispositivi embedded e i sensori utilizzati per la raccolta dei dati, le operazioni a questo livello consistono fondamentalmente nella raccolta e nell'inoltro delle informazioni al livello superiore che avrà il compito di elaborarle.
- Il livello di **backend**: a questo livello troviamo la logica di funzionamento del sistema, il compito del backend è quello di ricevere i dati dal livello inferiore e inoltrarli al sistema di memorizzazione permanente, passarli al livello superiore per l'interrogazione da parte degli utenti finali. Svolge dunque sia una funzione di comunicazione tra il livello di sensoristica e di interfaccia utente, sia di elaborazione e memorizzazione permanente dei dati.
- Il livello di **interfaccia utente**: come si può intuire facilmente, questo è il livello al quale accedono gli utenti, attraverso di esso è possibile consultare i dati in modo intuitivo. Questo livello nasconde i livelli inferiori e rappresenta quindi l'interfaccia del sistema verso l'utente finale.

La divisione in livelli del sistema ci ha permesso di sviluppare in modo indipendente le varie parti, ciò è risultato indispensabile in quanto abbiamo dovuto collaborare con un altro collega ed è subito sorta la necessità di lavorare in parallelo su diverse parti del sistema. Inoltre, un approccio di questo tipo facilita la possibilità di modificare le varie parti senza che le nuove modifiche abbiano ripercussioni su tutto il resto dell'architettura. Ciascun livello è visto dunque come una scatola nera, definita l'interfaccia di comunicazione, cioè le regole di comunicazione tra i diversi livelli tutta la

logica interna ad un livello è assolutamente irrilevante per il funzionamento e lo sviluppo delle restanti parti.

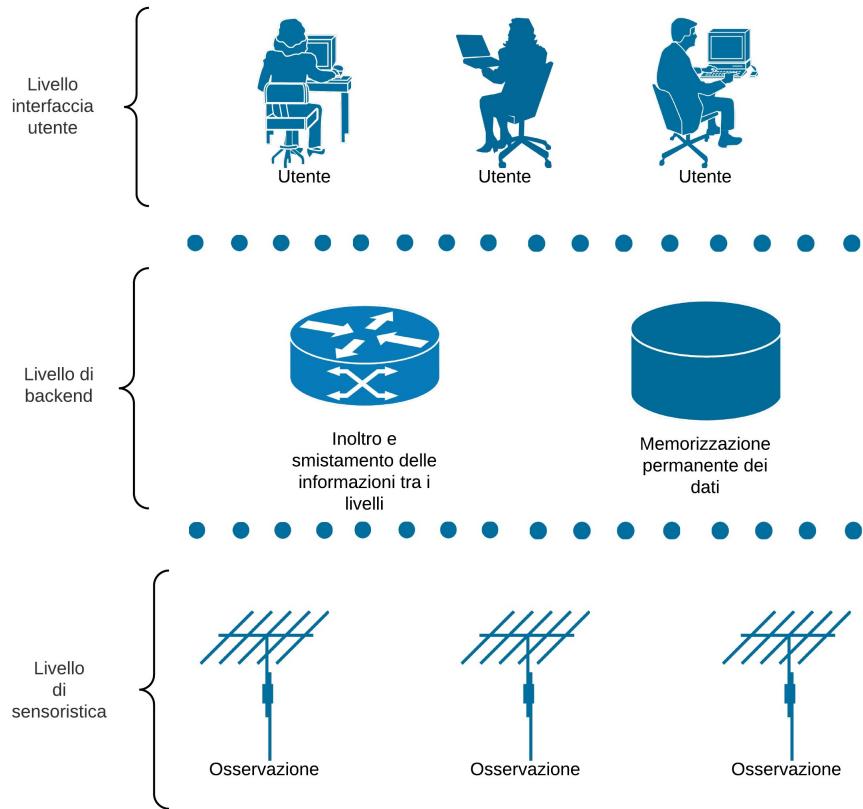


Figura 1.1: Livelli del sistema

1.3 Struttura dati

In questo paragrafo forniamo un elenco delle entità e dei rispettivi attributi che abbiamo creato per rappresentare il nostro dominio di interesse. Per rappresentare i dati abbiamo usato il modello NGSI (New Generation Service Interface).

NGSI è il modello dei dati adottato da Fi-ware, nei capitoli successivi verrà fornita una trattazione più approfondita, in questa sede possiamo

dire molto brevemente che il modello NGSI rappresenta i dati attraverso il concetto di entità, tutto ciò che è rappresentabile nel mondo reale è visto come una singola entità. A ciascuna entità è associato un campo id che la identifica univocamente ed un campo type. Un'entità può avere uno o più attributi; un attributo è composto da una tripla (name, type, value). Oltre agli attributi è possibile associare degli altri campi detti metadati che forniscono informazioni aggiuntive sugli attributi.

L'entità più importante nel nostro sistema è il **Service**, questa entità rappresenta il singolo servizio. I servizi presenti in questo momento sulla piattaforma sono tre: uno per la misurazione di temperatura, uno per la pressione e l'ultimo per la rilevazione del particolato. Il sistema, come già anticipato, prevede la possibilità di aggiungere nuovi servizi e meccanismi di discovery dei servizi per l'utente.

Di seguito sono riportati gli attributi dell'entità **Service** con una breve spiegazione:

1. **description**: una breve descrizione del servizio
2. **typeOfService**: stringa di testo che rappresenta il servizio

Successivamente abbiamo l'entità **Procedure** che rappresenta il singolo sensore montato sul taxi, questi sono i suoi attributi:

1. **description**: una breve descrizione del sensore.
2. **model**: marca e modello del sensore
3. **type**: indica se il sensore è fisso o mobile
4. **datasheet**: link al datasheet del sensore
5. **observed_property**: proprietà fisica misurata dal sensore.

Per rappresentare il taxi esiste l'entità **Taxi** con i seguenti attributi:

1. **ID_SERVICE**: id del servizio a cui partecipa il taxi
2. **procedure**: id del sensore montato sul taxi

3. **position**: posizione attuale occupata dal taxi

Infine abbiamo l'entità **Observations** che rappresenta la singola osservazione effettuata dal sensore. Questi i suoi attributi:

1. **ID_SERVICE**: id del servizio a cui appartiene la misurazione
2. **date**: data della misurazione
3. **duration**: durata della misurazione
4. **server**: ip del server che contiene i dati dell'osservazione
5. **position**: posizione della misurazione
6. **observed_property**: proprietà fisica oggetto della misura
7. **procedure**: id del sensore che ha effettuato la misurazione

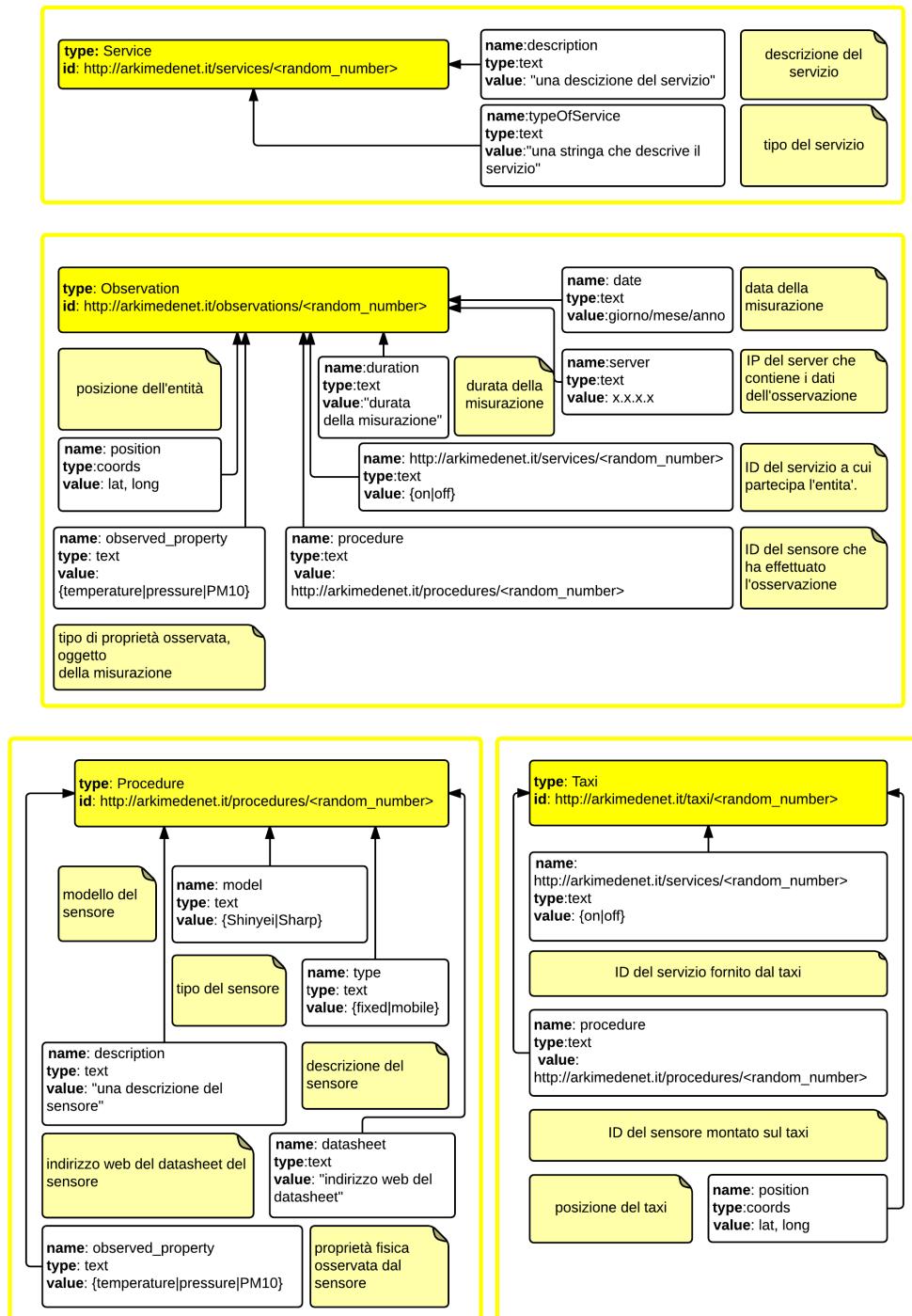


Figura 1.2: Struttura dati NGSI

Capitolo 2

La piattaforma Fi-ware

2.1 Cos'è Fi-ware

Fi-ware è un progetto finanziato dall'Unione Europea che si prefigge l'obiettivo di creare un'architettura aperta ed un insieme di specifiche che aiutino gli sviluppatori, i fornitori di servizi e le organizzazioni nella creazione di prodotti software innovativi per l'internet del futuro. La piattaforma cerca di ridurre gli ostacoli di carattere economico e tecnico che le piccole e medie imprese incontrano nello sviluppo e nell'innovazione dei loro prodotti, fornendo una serie di servizi software già pronti detti **generic enabler**. I generic enabler sono a tutti gli effetti dei web services¹ che rispettano il paradigma RESTful², offrono dunque le loro funzionalità attraverso RE-

¹Un web service è un software che opera in un contesto distribuito, di solito internet, allo scopo di fornire una certa tipologia di servizi alle applicazioni. Il più grosso vantaggio nell'uso dei web services è che questi forniscono un modo semplice per far interagire tra loro applicazioni sviluppate con linguaggi differenti che girano su architetture diverse. Un web service definisce un'interfaccia, una serie di chiamate di funzioni o servizi, che viene esposta alle applicazioni esterne. In questo modo le applicazioni possono richiamare le funzionalità fornite dal servizio. L'interfaccia è raggiungibile attraverso i tradizionali protocolli web come http, ciò rende i servizi indipendenti da uno specifico linguaggio di programmazione o sistema operativo garantendo l'interoperabilità tra componenti diverse di uno stesso sistema.

²Rest è un insieme di linea guida per l'implementazione di web services, i principi di questo paradigma sono: identificazione delle risorse (URI), utilizzo del protocollo http per la comunicazione, comunicazione senza stato, risorse autodescrittive e collegamenti tra risorse tramite link.

ST API³. I servizi, quindi i Generic Enabler, sono erogati attraverso una piattaforma cloud su cui l'intera architettura si basa.

Sono state individuate delle macro categorie all'interno delle quali i GE sono stati catalogati; queste categorie sono dette capitoli. Ogni capitolo identifica una specifica tipologia di servizi erogabili. Di seguito la lista dei domini con una breve descrizione dei servizi offerti per categoria:

- **Cloud Hosting:** il livello che mette a disposizione le risorse di rete, di storage e computazionali sulle quali tutti gli altri servizi girano.
- **Data/Context Management:** questo livello fornisce i servizi per lo scambio, l'accesso, il processamento e l'analisi di grosse quantità di dati, oltre che la trasformazione di questi ultimi in informazioni di contesto disponibili alle applicazioni.
- **Application/Services Ecosystem and Delivery Framework:** infrastruttura per creare, pubblicare e gestire i servizi Fi-ware durante il loro ciclo di vita.
- **Internet of Things (IoT) Services Enablement:** interfaccia di comunicazione tra i servizi Fi-ware e i dispositivi IoT.
- **Interface to Networks and Devices (I2ND):** interfaccia aperta verso i dispositivi e le reti, serve le necessità di connettività dei servizi erogati attraverso la piattaforma
- **Security:** servizi di privacy e sicurezza per la piattaforma

Tutti i servizi nascono dalla definizione di specifiche aperte circa le funzionalità che devono essere erogate e come queste devono essere realizzate. Ciò permette di scindere i servizi dalla loro effettiva implementazione. Si definiscono di fatto le API ed i protocolli che i vari GE devono implementare, e successivamente si procede all'implementazione. Il progetto Fi-ware si è fatto carico di realizzare un'implementazione per tutti i GE dei capitoli.

³Con il termine Application Program Interface ci si riferisce ad un insieme di funzioni messe a disposizione al programmatore per lo svolgimento di un ben determinato compito. Il più delle volte il termine si riferisce alla librerie di un certo linguaggio di programmazione.

Nulla vieta al privato, dato che le specifiche sono aperte, di realizzare una propria implementazione di uno specifico GE e rilasciarla al pubblico o in forma open source o commerciale. Questo vuol dire che ogni servizio, ogni implementazione di un singolo GE, può essere sostituita con altri servizi che rispettano le specifiche. Fi-ware ha scelto di definire specifiche open proprio per rendere il mercato il più libero possibile dal monopolio di un singolo soggetto; un'azienda può scegliere se fornire servizi attraverso i GE disponibili o realizzare una propria implementazione del servizio. L'utente può passare da un fornitore di servizio all'altro secondo il proprio gusto, con la garanzia che il servizio offerto soddisfi determinate caratteristiche definite a priori dalle specifiche. Inoltre, le implementazioni dei servizi sulla piattaforma Fi-ware sono rilasciate in forma open source, il codice è dunque liberamente consultabile on-line.

Una visione completa dell'architettura dell'intera piattaforma con i GE divisi per categoria è mostrata nella figura sottostante.

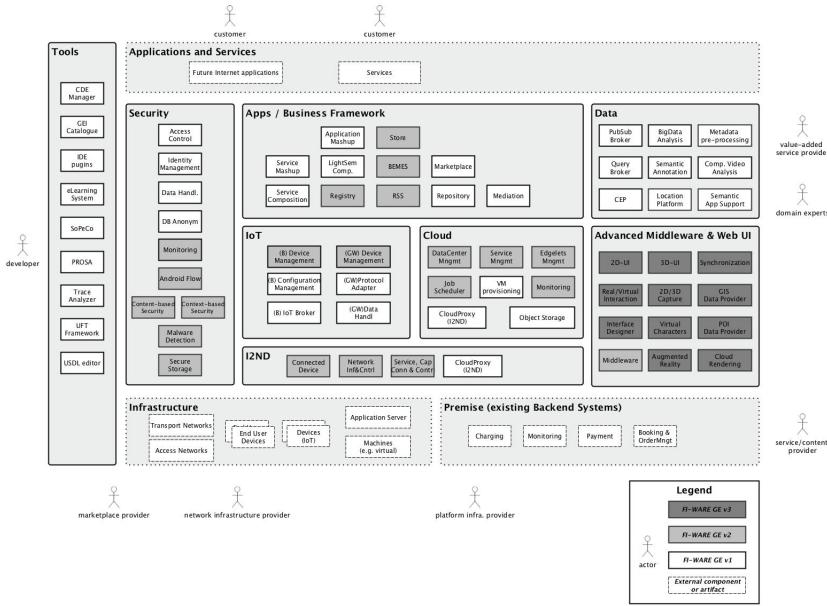


Figura 2.1: Architettura Fi-ware

Una lista completa degli enablers può essere trovata consultando il catalogo on-line all'indirizzo <http://catalogue.fi-ware.org/enablers>. Per esempio, lo sviluppatore impegnato nella creazione di un'applicazione IoT accederà

dal catalogo al capitolo IoT cercando gli enabler che possono aiutarlo nel proprio obiettivo. Ogni GE ha una propria pagina dedicata all'interno della quale si fornisce una descrizione dei servizi offerti dal componente e tutta la documentazione dettagliata sull'installazione e l'uso delle API per il programmatore. Esiste anche un portale di e-learning pensato appositamente per semplificare il percorso di apprendimento della tecnologia Fi-ware; il portale contiene corsi, video e documenti per ogni singolo GE⁴.

I vari GE possono comunicare tra di loro attraverso un modello di dati detto NGSI, questo modello è estremamente semplice e flessibile per rappresentare qualunque dominio d'informazione.

Il progetto Fi-ware è partito nel 2008 ed è arrivato alla terza fase del suo iter, in questa fase vengono messi a disposizione dei finanziamenti per supportare le piccole e medie imprese nello sviluppo di applicazioni attraverso la tecnologia offerta. Per velocizzare ulteriormente il processo, sono stati istituiti gli **acceleratori** Fi-ware il cui compito è quello di selezionare le idee più innovative, finanziarle e supportare le aziende durante tutto il periodo di implementazione con supporto tecnico e tutoring sulle nuove tecnologie. Gli acceleratori sono divisi per categorie e comprendono praticamente tutti i principali business del mercato ICT.

- smart cities
- e-health
- trasporti
- energia e ambiente
- agro-food
- media e contenuti
- produzione e logistica
- sociale e apprendimento

⁴La piattaforma di e-learning può essere raggiunta al seguente indirizzo: <http://edu.fi-ware.org/>

2.2 Il Fi-lab

Il Fi-lab è un laboratorio virtuale pensato appositamente per permettere agli addetti ai lavori di sperimentare le tecnologie messe a disposizione dalla piattaforma Fi-ware. L'intera piattaforma si basa su un'architettura distribuita di nodi federati in tutto il territorio europeo. Ciò vuol dire che, ai fini dell'utilizzo pratico, la piattaforma è assolutamente trasparente per l'utente finale. Come è possibile vedere nell'immagine successiva, l'home page del Fi-lab presenta un menu con sei principali categorie:

1. **Cloud**: attraverso questa sezione è possibile gestire le macchine messe a disposizione dalla piattaforma: richiedere la creazione di macchine virtuali, scegliere l'immagine del sistema operativo da caricare e i software che si desidera installare.
2. **Store**: questa sezione permette l'accesso al marketplace del Fi-lab, attraverso di esso è possibile ricercare i widget e gli operatori forniti dal Fi-lab per la sperimentazione con Wirecloud.
3. **Mashup**: attraverso questa sezione si accede all'istanza Wirecloud del Fi-lab.
4. **Data**: questa sezione permette l'accesso alla piattaforma CKAN per la pubblicazione dei dati in formato open data.
5. **Account**: è la sezione principale alla quale si accede subito dopo aver fatto il log-in. E' questa la sezione mostrata nella figura di sotto.
6. **Help&Info**: sezione relativa alla documentazione del Fi-lab

Le sezioni **Store**, **Mashup** e **Data** non vengono trattate in questa sede in quanto rappresentano le istanze sul fi-lab rispettivamente di **Wirecoud** (Store e Mashup) e di **CKAN**, per avere informazioni su queste due piattaforme invitiamo il lettore a consultare il paragrafo 2.9 a pagina 57 ed il paragrafo 2.10 a pagina 59.

Vediamo invece di entrare nel dettaglio della sezione **Cloud**.

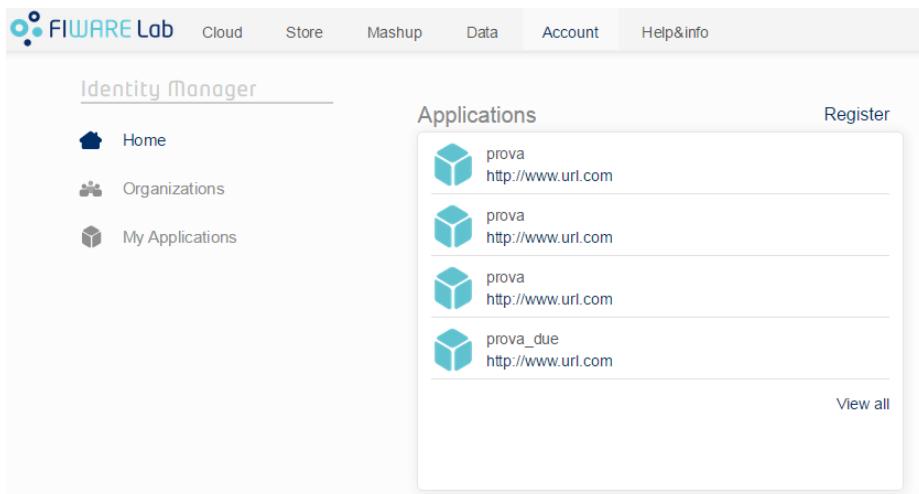


Figura 2.2: Home page del Fi-lab

Instance Name	IP Address	Size	Keypair	Status	Task	Power State
idos	10.0.9.115		antonio	SHUTOFF	None	RUNNING
orionold	10.0.5.92 130.206.85.25		antonio	ACTIVE	None	RUNNING
oriontemplate-orion-1...	10.0.5.214		antonio	SHUTOFF	None	RUNNING

Figura 2.3: Sezione Cloud sul Fi-lab

L’immagine di sopra mostra ciò che viene visualizzato su schermo quando l’utente clicca sulla voce cloud della dashboard. Subito possiamo notare nel frame principale (**Instances**) le macchine che l’utente ha istanziato sul nodo spagnolo. La localizzazione fisica delle macchine viene fornita nel menù a sinistra dal campo **Region**. Come è facile notare ci sono tre macchine istanziate di cui due spente ed una attiva. Alla macchina attiva è associato un ip pubblico (130.20685.25) per raggiungerla dall’esterno della rete del fi-lab. Per richiedere un indirizzo ip o aprire una porta TCP/UDP per un particolare servizio è possibile accedere dal menù a sinistra alla voce **Security**. La voce **Snapshot** invece permette appunto di salvare uno snapshot per le macchine istanziate. Uno snapshot è come una fotografia

dello stato e delle configurazioni della macchina ad un certo istante di tempo. Gli snapshot vengono utilizzati per ripristinare la macchina virtuale ad uno stato precedente in caso errori o cattivi funzionamenti.

Il metodo più facile per istanziare una macchina virtuale sul fi-lab è sicuramente attraverso la sezione **Images**. Cliccando sulla voce corrispondente nel menù a sinistra avremo una lista di macchine già pronte come quelle in figura. Di solito le macchine hanno un nome abbastanza esplicito circa il GE che mettono a disposizione. Per lanciare la macchina basterà cliccare sul bottone **Launch** a sinistra. Cliccando sul nome dell'immagine si accede ad un resoconto circa lo stato e le caratteristiche hardware della macchina selezionata

Name	Type	Status	Visibility	Container Format	Disk Format	Actions
Ubuntu12.04-server-x86_64	baseimages	active	public	OVF	QCOW2	<button>Launch</button>
CentOS-4.3-x86_64	baseimages	active	public	OVF	QCOW2	<button>Launch</button>
Ubuntu Server 14.04.1 [x64]	baseimages	active	public	OVF	QCOW2	<button>Launch</button>
CentOS-6.5-x64	baseimages	active	public	OVF	QCOW2	<button>Launch</button>
CentOS-7-x64	baseimages	active	public	OVF	QCOW2	<button>Launch</button>
repository-image-R3.2	fwware-apps	active	public	AMI	AMI	<button>Launch</button>
wirecloud-img	fwware-apps	active	public	OVF	QCOW2	<button>Launch</button>
wstore-img	fwware-apps	active	public	OVF	QCOW2	<button>Launch</button>

Figura 2.4: Immagini già pronte sul FI-lab

2.3 Orion Context Broker

Se Internet come la conosciamo oggi è stata fondamentalmente pensata per collegare tra loro i calcolatori e di conseguenza gli utenti che li utilizzano, l'internet del futuro o internet delle cose deve essere immaginata come una rete in cui miliardi di dispositivi, oggetti del mondo reale, sensori, sono connessi e trasmettono costantemente informazioni sul loro stato o sull'ambiente circostante. I dati raccolti da questi dispositivi richiedono spesso un' elaborazione successiva per trasformare i dati grezzi in informazione che abbia del valore aggiunto da presentare poi all'utente finale.

E' necessario dunque un livello intermedio tra questi due mondi molto lontani, il dominio della macchine e quello degli utenti.

Chi sviluppa applicazioni in ambito IoT o Smart Mobility⁵ ha quindi spesso bisogno di una logica che sia in grado di ricevere i dati dai dispositivi connessi e inviarli agli utenti, oppure di identificare il verificarsi di determinati eventi o condizioni sui dati ricevuti e avvertire l'utente in tempo reale. Sviluppare una tale logica di **data context management** richiede una certa quantità di risorse in termini di programmatore, conoscenze tecniche ed economiche che spesso rappresenta il principale ostacolo che preclude alle piccole aziende la partecipazione a questo nuovo mercato.

Fi-ware ha dedicato un intero capitolo alla gestione delle informazioni di contesto, fornendo una serie di componenti specifici che svolgono queste operazioni.

Il componente principale di questo dominio è sicuramente l'**Orion Context Broker**: come spiega il nome stesso, Orion si occupa di fare da broker, cioè di smistare le informazioni di contesto tra le applicazioni che producono il dato e quelle che lo usano; di fare da intermediario tra il livello dei dispositivi e il livello delle applicazioni utente. Prima di esaminare nel dettaglio le funzionalità messe a disposizione da questo componente software è necessario introdurre brevemente cosa s'intende per informazioni di contesto.

Le informazioni di contesto nel modello **NGSI** sono rappresentate e scambiate attraverso strutture dati chiamate **context elements**:

Per ciascun context elements il modello prevede:

- Un **id** ed un tipo: *EntityId* e *EntityType*
- Una sequenza di **attributi**: un attributo è una tripla composta da $\langle Name, Type, Value \rangle$
- **Metadati** opzionali riferiti agli attributi $\langle Name, Type, Value \rangle$

⁵Per mobilità intelligente s'intende un modo nuovo di intendere la mobilità delle persone e delle merci che fa uso massiccio di elementi tecnologici quali sensori o dispositivi embedded per realizzare sistemi di trasporto sostenibili a basso costo in grado di minimizzare gli sprechi e i fattori inquinanti.

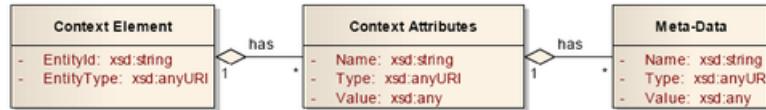


Figura 2.5: Modello NGSI

Quindi tutti i dati riguardanti le entità relative ad una specifica applicazione sono informazioni di contesto; le informazioni di contesto vengono rappresentate e scambiate usando il modello NGSI visto in precedenza. Le informazioni di contesto sono smistate dal broker alle applicazioni che ne hanno bisogno. Esistono due tipologie di applicazioni: produttrice e consumatrice.

L'applicazione produttrice appartiene al livello dei dispositivi che generano i dati; l'applicazione consumatrice appartiene al livello degli utenti che usano le informazioni.

In maniera più formale vediamo di elencare quindi le operazioni che Orion mette a disposizione degli utenti:

- **Register context** producer applications: attraverso questa operazione è possibile registrare l'applicazione che produce il dato (provider)
- **Update context** information: attraverso questa operazione è possibile aggiornare il valore di un'informazione di contesto
- **Notify** when changes on context information: questa operazione permette di essere informati quando un'informazione di contesto viene aggiornata
- **Query context** information: questa operazione permette di conoscere il valore di un'informazione di contesto.

I'Orion context broker è una buona scelta dunque, in tutti quelli scenari in cui nella propria architettura si ha la necessità di un componente che faccia da mediatore tra il produttore dell'informazione di contesto (il sensore) e il consumatore (l'applicazione utente).

Riassumendo molto sinteticamente possiamo dire quindi che sono tre gli attori nel dominio del data context management: il produttore dell'informazione di contesto che produce il dato e lo invia al broker, il consumatore che contatta o viene contattato dal broker per usufruire dell'informazione di contesto e appunto il broker stesso che fa da mediatore tra producer e consumer. Un esempio ancora più chiaro di questa interazione è mostrato nella figura sottostante:



Figura 2.6: Producer - Broker - Consumer

Le operazioni, che da adesso chiameremo API, messe a disposizione dal broker sono divise in due grosse categorie: **NGSI-9** e **NGSI-10** [11].

Le api di tipo NGSI-10 rappresentano l'interfaccia verso il programmatore per manipolare le informazioni di contesto, tutte le operazioni all'interno di questa categoria si occupano dello scambio e la manipolazione delle informazioni di contesto.

NGSI-9 è invece l'interfaccia relativa alla gestione della disponibilità delle informazioni di contesto. A questa categoria appartengono dunque tutte le operazioni che trattano la disponibilità delle informazioni.

Di seguito riportiamo uno schema delle funzionalità appartenenti alle due interfacce sopra citate. Tutte le API in questione sono delle semplici chiamate rest su protocollo http.

- **NGSI-10**

- **updateContext**: creazione e aggiornamento di entità.
- **queryContext**: interrogazione degli attributi di entità.

- **subscribeContext**: sottoscrizione ad una entità; una sottoscrizione informa l'applicazione, in tempo reale, sulle operazioni di update sull'entità.
- **unsubscribeContext**: cancellazione di una sottoscrizione.
- **updateContextSubscription**: aggiornamento di una sottoscrizione.
- **notifyContext**: notifica all'applicazione (che ha fatto richiesta di notifica attraverso una sottoscrizione) di un'operazione di update su una entità.

- **NGSI-9**

- **registerContext**: registrazione di un provider per una entità, Orion non memorizza nel suo db i dati relativi all'entità in questione, bensì contatterà il provider dell'entità per servire le richieste di query.
- **discoverContextAvailability**: permette all'applicazione che ri-chiama la procedura di scoprire il provider per una specifica entità.
- **subscribeContextAvailability**: sottoscrizione ad una entità, questo tipo di sottoscrizione informerà l'applicazione ogni qual volta viene registrato un nuovo provider per l'entità o un certo tipo di entità.
- **unsubscribeContextAvailability**: cancellazione di una sottoscrizione.
- **updateContextAvailabilitySubscription**: aggiornamento di una sottoscrizione.
- **notifyContextAvailability**: notifica all'applicazione (che ha fatto richiesta di notifica attraverso una sottoscrizione) di un'operazione di registerContext di un provider per un'entità.

2.4 Modello dei dati NGSI

L’aspetto centrale del modello dei dati NGSI è il concetto di **entità**. Le entità sono la rappresentazione virtuale di tutti i possibili oggetti fisici nel mondo reale. Esempi di entità fisiche sono le persone, una stanza, le automobili etc. Queste entità virtuali hanno un **identificatore** ed un **tipo**. Per esempio, un’entità virtuale rappresentante una persona di nome Giovanni potrebbe avere l’identificatore «Giovanni» e il tipo «persona».

Qualsiasi possibile informazione circa le entità fisiche è espresso nella forma di **attributi** delle entità virtuali. Gli attributi hanno un **nome** ed un **tipo**. Per esempio, la temperatura corporea di Giovanni potrebbe essere rappresentata come un attributo avente il nome «temperatura_corporea» ed il tipo «temperatura». I valori relativi a questi attributi sono contenuti nel campo **value** dell’attributo. Un attributo è di fatto una tripla composta da $\langle id, tipo, valore \rangle$. Vi sono inoltre altri campi che contengono i **metadati** degli attributi. I metadati sono dati che si riferiscono ai dati; nel nostro esempio della temperatura corporea un metadato potrebbe essere il tempo della misurazione, l’unità di misura e altre informazioni circa il valore dell’attributo «temperatura_corporea».

Esiste anche il concetto di **dominio** degli attributi. Un dominio di attributo raggruppa logicamente un insieme di attributi. Per esempio, il dominio di attributo «stato_di_salute» potrebbe includere gli attributi «temperatura_corporea» e «pressione_sanguigna».

Il contenitore usato per scambiare informazioni circa le entità è il **context element**. Un context element può contenere informazioni circa più attributi di una stessa entità. Il dominio di questi attributi può anche essere specificato all’interno del context element; in questo caso tutti i valori degli attributi appartengono a quello specifico dominio.

Formalmente un context element contiene le seguenti informazioni

- Un EntityId che include un nome ed un tipo.
- Una lista di attributi
- Il nome di un dominio di attributo (opzionale)

- Una lista di metadati che si applicano a tutti i valori degli attributi di un dato dominio.

2.4.1 updateContext

Questa operazione è una delle più importanti tra quelle messe a disposizione dall’interfaccia NGSI-10, la *updateContext* ha infatti la duplice funzione di creare una nuova entità e di aggiornare i valori degli attributi di un’entità precedentemente creata.

Per creare un’entità è necessario richiamare il metodo POST del protocollo http con i seguenti parametri nella richiesta:

- URL: localhost:1026/v1/updateContext
- Content-Type: application/xml
- Accept: application/xml

Questi tre parametri specificano rispettivamente il percorso della chiamata rest da contattare, il formato (in questo caso xml) del body del metodo post e il formato del messaggio di risposta.

Di seguito è mostrato il corpo di una chiamata *updateContext* per la creazione di un’entità di tipo Room con id Room1. L’entità, come prevede il modello di dati NGSI, possiede degli attributi, in questo caso *temperature* e *pressure* che vengono specificati attraverso i tag *<contextAttributeList>* e *<contextAttribute>*.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <updateContextRequest>
3 <contextElementList>
4 <contextElement>
5 <entityId type="Room" isPattern="false">
6 <id>Room1</id>
7 </entityId>
8 <contextAttributeList>
9 <contextAttribute>
10 <name>temperature</name>

```

```

11 <type>float</type>
12 <contextValue>23</contextValue>
13 </contextAttribute>
14 <contextAttribute>
15 <name>pressure</name>
16 <type>integer</type>
17 <contextValue>720</contextValue>
18 </contextAttribute>
19 </contextAttributeList>
20 </contextElement>
21 </contextElementList>
22 <updateAction>APPEND</updateAction>
```

Da notare il tag `<updateAction>` che nel caso di creazione di una nuova entità è valorizzato ad APPEND.

La `updateContext` però, come detto, viene usata anche per l'aggiornamento dei valori degli attributi di un'entità già esistente.

I parametri da passare al metodo POST e i tag per l'operazione di aggiornamento sono gli stessi per la creazione dell'entità con l'unica differenza che il tag `<updateAction>` viene valorizzato ad UPDATE.

Di seguito riportiamo il corpo di una chiamata `updateContext` in cui si aggiornano i valori degli attributi *temperature* e *pressure* dell'entità creata con l'operazione precedente rispettivamente a 26.5 e 763. Ribadiamo il valore UPDATE nel tag `<updateAction>`.

```

1 <updateContextRequest>
2 <contextElementList>
3 <contextElement>
4 <entityId type="Room" isPattern="false">
5 <id>Room1</id>
6 </entityId>
7 <contextAttributeList>
8 <contextAttribute>
9 <name>temperature</name>
10 <type>float</type>
```

```

11 <contextValue>26.5</contextValue>
12 </contextAttribute>
13 <contextAttribute>
14 <name>pressure</name>
15 <type>integer</type>
16 <contextValue>763</contextValue>
17 </contextAttribute>
18 </contextAttributeList>
19 </contextElement>
20 </contextElementList>
21 <updateAction>UPDATE</updateAction>
22 </updateContextRequest>
```

2.4.2 queryContext

Questa chiamata rest può essere utilizzata dall'applicazione consumatrice dell'informazione di contesto per reperire i dati presenti su Orion che sono stati precedentemente generati dall'applicazione produttrice.

La funzionalità è offerta richiamando il metodo POST del protocollo http con questi parametri:

- URL: localhost:1026/v1/updateContext
- Content-Type: application/xml
- Accept: application/xml

Nel riquadro in basso è mostrato il corpo di una chiamata *queryContext* relativa all'entità che abbiamo creato in precedenza attraverso la *updateContext*. Come si può notare, vengono specificati sia il tipo dell'entità che si vuole interrogare che il relativo id. Orion permette anche ricerche più generali in cui è possibile non specificare il tipo o usare espressioni regolari come per esempio `.*` per selezionare tutti i possibili id.

```
<?xml version="1.0" encoding="UTF-8"?>
<queryContextRequest>
```

```

<entityIdList>
<entityId type="Room" isPattern="false">
<id>Room1</id>
</entityId>
</entityIdList>
<attributeList/>
</queryContextRequest>

```

La risposta di una tale query è mostrato di seguito. Come è possibile notare vengono restituiti negli appositi tag i valori degli attributi di cui si è fatta richiesta nella query, in questo caso *temperature* e *pressure*. Inoltre il messaggio contiene un codice di risposta ed una response phrase che può essere usata per il debugging nel caso di insuccesso della richiesta.

```

1 <queryContextResponse>
2 <contextResponseList>
3 <contextElementResponse>
4 <contextElement>
5 <entityId type="Room" isPattern="false">
6 <id>Room1</id>
7 </entityId>
8 <contextAttributeList>
9 <contextAttribute>
10 <name>temperature</name>
11 <type>float</type>
12 <contextValue>23</contextValue>
13 </contextAttribute>
14 <contextAttribute>
15 <name>pressure</name>
16 <type>integer</type>
17 <contextValue>720</contextValue>
18 </contextAttribute>
19 </contextAttributeList>
20 </contextElement>
21 <statusCode>

```

```

22 <code>200</code>
23 <reasonPhrase>OK</reasonPhrase>
24 </statusCode>
25 </contextElementResponse>
26 </contextResponseList>
27 </queryContextResponse>
```

2.4.3 subscribeContext

Le due chiamate che abbiamo visto fino ad ora, *updateContext* e *queryContext*, sono già sufficienti per realizzare qualunque scambio di informazioni tra produttore e consumatore attraverso Orion. Il produttore crea l'entità e valorizza gli attributi con chiamate *updateContext*, il consumatore accede ai dati con le *queryContext*.



Figura 2.7: Comunicazione sincrona

Questo tipo d'interazione è chiaramente sincrona, il consumatore per restare aggiornato sui cambiamenti effettuati dal produttore deve implementare un qualche meccanismo di polling interrogando periodicamente il broker per reperire i dati. La *subscribeContext* mette invece a disposizione una soluzione asincrona per il reperimento delle informazioni da parte del consumatore. Un consumatore può sottoscriversi per la ricezione di informazioni (context elements) al verificarsi di certe condizioni. Una sottoscrizione può avere una durata oltre la quale il consumatore non verrà più informato sul cambiamento delle entità sottoscritte. Il consumatore una volta sottoscritto verrà informato automaticamente attraverso una chiamata *notifyContext*. E' importante notare che l'applicazione che effettua la sot-

toscrizione può anche non essere l'applicazione consumatrice che riceverà il dato.

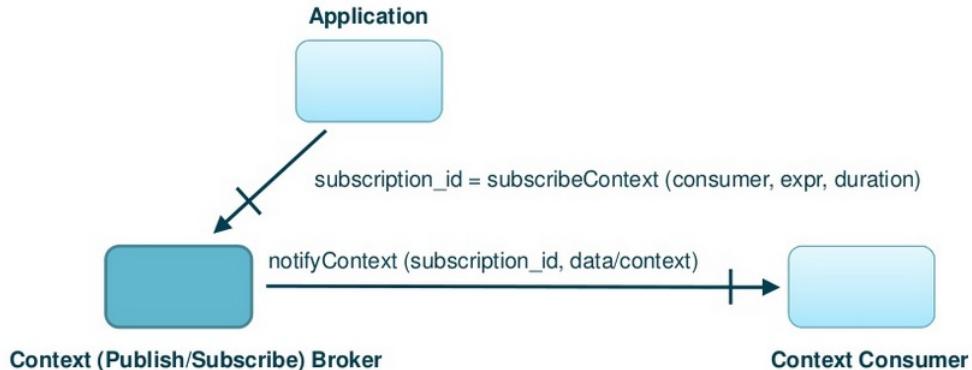


Figura 2.8: Comunicazione asincrona

Le sottoscrizioni possono essere più complicate coinvolgendo anche il context broker. Nella figura sottostante un'applicazione sottoscrive il context broker 2 (CB2) sul context broker 1 (CB1), ciò vuol dire che il CB1 notificherà attraverso *notifyContext* al CB2 gli aggiornamenti delle proprie entità. Il CB2 in questo scenario gioca quindi il ruolo di consumatore. Questo genere di sottoscrizioni sono alla base di una particolare modalità di funzionamento del broker detta push federation che affronteremo nel paragrafo successivo.

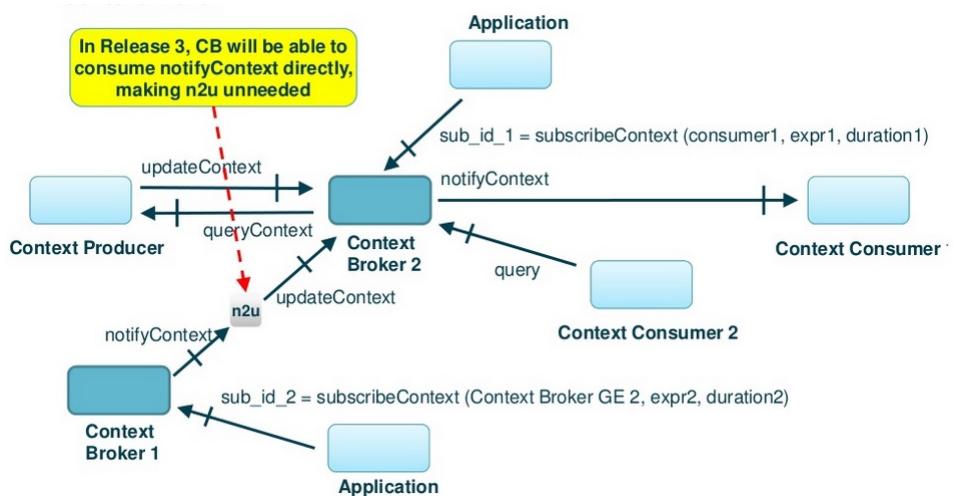


Figura 2.9: Sottoscrizione tra broker

Le tipologie di sottoscrizioni possibili sono due:

- **ONTIMEINTERVAL**: il consumatore verrà contattato a intervalli regolari anche se il dato non è cambiato.
- **ONCHANGE**: il consumatore verrà contattato solo quando l'attributo oggetto della sottoscrizione ha subito un cambiamento.

Indipendentemente dal tipo di sottoscrizione la *subscribeContext* viene richiamata attraverso il metodo POST con i soliti tre parametri.

- URL: localhost:1026/v1/subscribeContext .
- Content-Type: application/xml
- Accept: application/xml

Di seguito riportiamo il corpo di una chiamata di tipo **ONTIMEINTERVAL**. In questo esempio viene effettuata una sottoscrizione all'entità di tipo Room con id Room1. Il tag *<attributeList>* contiene la lista degli attributi che il consumatore vuole siano notificati allo scadere dell'intervalle temporale di notifica. Il tag *<reference>* specifica l'indirizzo ip dell'applicazione consumatrice, dice in pratica al broker a quale indirizzo inviare i dati allo scadere dell'intervalle di tempo. Il tag *<condValue>* permette di definire l'intervalle di notifica, mentre il tag *<duration>* specifica la durata della sottoscrizione nello formato **ISO8601**.

```

1 <?xml version="1.0"?>
2 <subscribeContextRequest>
3 <entityIdList>
4 <entityId type="Room" isPattern="false">
5 <id>Room1</id>
6 </entityId>
7 </entityIdList>
8 <attributeList>
9 <attribute>temperature</attribute>
10 </attributeList>
11 <reference>http://localhost:1028/accumulate</reference>

```

```

12 <duration>P1M</duration>
13 <notifyConditions>
14 <notifyCondition>
15 <type>ONTIMEINTERVAL</type>
16 <condValueList>
17 <condValue>PT10S</condValue>
18 </condValueList>
19 </notifyCondition>
20 </notifyConditions>
21 </subscribeContextRequest>
```

Nel caso di sottoscrizione di tipo **ONCHANGE** è necessario specificare attraverso i tag `<condValueList>` e `<condValue>` quali sono gli attributi al cui cambiamento Orion dovrà generare l'evento di notifica. Nel nostro esempio la notifica del valore dell'attributo *temperature* si attiverà a seguito di un update dell'attributo *pressure*.

```

1 <subscribeContextRequest>
2 <entityIdList>
3 <entityId type="Room" isPattern="false">
4 <id>Room1</id>
5 </entityId>
6 </entityIdList>
7 <attributeList>
8 <attribute>temperature</attribute>
9 </attributeList>
10 <reference>http://localhost:1028/accumulate</reference>
11 <duration>P1M</duration>
12 <notifyConditions>
13 <notifyCondition>
14 <type>ONCHANGE</type>
15 <condValueList>
16 <condValue>pressure</condValue>
17 </condValueList>
18 </notifyCondition>
```

```
19 </notifyConditions>
20 <throttling>PT5S</throttling>
21 </subscribeContextRequest>
```

2.5 Context Broker Federation

In questo paragrafo parleremo di una particolare configurazione che è possibile attiva sul broker: la federazione. In questa modalità il contenuto informativo di un broker può essere riprodotto, attraverso meccanismi di sottoscrizione e notifica, su un secondo broker. I motivi per attivare una configurazione del genere possono essere principalmente due: si vuole integrare il contenuto informativo di due domini/applicazioni differenti; ad esempio in ambito smart city potremmo pensare di federare due broker appartenenti a città differenti per accedere dallo stesso broker alle informazioni di entrambe le città. Un altro buon motivo nell'utilizzare la federazione è la distribuzione del carico, potremmo pensare infatti di utilizzare due broker con lo stesso contenuto informativo e distribuire in modo omogeneo le query degli utenti per migliorare i tempi di risposta dalla parte dell'applicazione consumatrice.

La federazione può essere realizzata in due modi: **push** e **pull** mode. La differenza sostanziale tra questi due tipi di federazione sta nel fatto che in push mode tutti i broker che prendono parte alla federazione aggiornano il proprio stato locale a seguito di un update su un'entità; nel caso pull invece, solo uno degli broker federati aggiorna effettivamente il dato. La differenza sarà più chiara nei paragrafi successivi.

Nei due paragrafi seguenti affronteremo entrambe le configurazioni. Inoltre, anticipiamo che il caso di federazione in push mode è stato effettivamente sperimentato ai fini di questo lavoro di tesi, è stato anche svolto uno studio delle prestazioni per la nostra applicazione in questa configurazione. Per maggiore informazioni rimandiamo al capitolo quattro.

2.5.1 Push Federation

Prendiamo in considerazione il seguente scenario: tre broker (A, B e C) sulla stessa macchina, in attesa di connessioni rispettivamente sulle porte **1030**, **1031** e **1032**. Naturalmente lo stesso esperimento può essere riprodotto lanciando le tre istanze su macchine differenti.

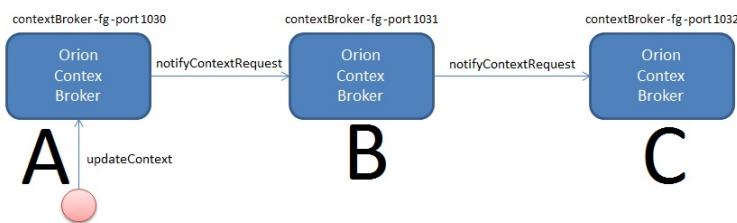


Figura 2.10: Push Federation

A questo punto inviamo una sottoscrizione attraverso la **subscribeContext** al broker A, in modo da inviare al broker B gli aggiornamenti effettuati sul broker A. Nel riquadro in basso è mostrato il corpo della richiesta. Da notare il tag *<reference>* attraverso il quale specifichiamo l'indirizzo del broker a cui mandare la notifica al cambiamento dell'attributo *temperature*. Da notare il fatto che il tag *<type>* è valorizzato a **ONCHANGE**. Per comprendere meglio il significato dei tag della *subscribeContext* rimandiamo il lettore al paragrafo 2.4.3 a pagina 27

Molto brevemente quello che stiamo facendo è di chiedere al broker A di inviare al broker B i valori degli attributi presenti nei tag *<condValueList>* ad ogni aggiornamento degli attributi presenti nei tag *<notifyConditions>* e *<notifyCondition>*.

```

1 <?xml version="1.0"?>
2 <subscribeContextRequest>
3 <entityIdList>
4 <entityId type="Room" isPattern="false">
5 <id>Room1</id>
6 </entityId>
7 </entityIdList>
8 <reference>http://localhost:1031/v1/notifyContext</reference>

```

```

9 <duration>P1M</duration>
10 <notifyConditions>
11 <notifyCondition>
12 <type>ONCHANGE</type>
13 <condValueList>
14 <condValue>temperature</condValue>
15 </condValueList>
16 </notifyCondition>
17 </notifyConditions>
18 <throttling>PT5S</throttling>
19 </subscribeContextRequest>
```

Facciamo la stessa cosa tra i broker B e C, inviamo una sottoscrizione al broker B affinchè inoltri automaticamente una notifica al broker C ad ogni aggiornamento delle proprie entità.

```

1 <subscribeContextRequest>
2 <entityIdList>
3 <entityId type="Room" isPattern="false">
4 <id>Room1</id>
5 </entityId>
6 </entityIdList>
7 <reference>http://localhost:1032/v1/notifyContext</reference>
8 <duration>P1M</duration>
9 <notifyConditions>
10 <notifyCondition>
11 <type>ONCHANGE</type>
12 <condValueList>
13 <condValue>temperature</condValue>
14 </condValueList>
15 </notifyCondition>
16 </notifyConditions>
17 <throttling>PT5S</throttling>
18 </subscribeContextRequest>
```

A questo punto creiamo un'entità sul broker A, quello che ci aspettiamo

che avvenga è che la creazione dell’entità sul broker A attivi la notifica verso il broker B che a sua volta notificherà l’evento al broker C a seguito delle operazioni di sottoscrizioni svolte in precedenza. Il risultato della federazione è che adesso l’entità A è replicata su tutte e tre le istanze di Orion anche se abbiamo effettuato l’operazione di creazione solo sul broker A. Naturalmente il processo vale allo stesso modo anche per le operazione di aggiornamento di una entità in modalità federata.

```

1 <updateContextRequest>
2 <contextElementList>
3 <contextElement>
4 <entityId type="Room" isPattern="false">
5 <id>Room1</id>
6 </entityId>
7 <contextAttributeList>
8 <contextAttribute>
9 <name>temperature</name>
10 <type>float</type>
11 <contextValue>23</contextValue>
12 </contextAttribute>
13 </contextAttributeList>
14 </contextElement>
15 </contextElementList>
16 <updateAction>APPEND</updateAction>
17 </updateContextRequest>
```

2.5.2 Pull Federation

Il caso di **pull** federation differisce dal **push** mode in quanto l’informazione di contesto non viene replicata su tutti i broker che partecipano alla federazione, bensì ogni broker gestisce localmente un certo numero di entità; se viene fatta richiesta ad un broker di informazioni gestite da un altro broker, il primo semplicemente contatta l’Orion appropriato per reperire l’informazione e poi smistarla all’utente. Quindi l’utente effettua una query o un

update su un’entità ad un determinato broker, se questa è gestita direttamente dal broker quest’ultimo serve personalmente la richiesta, altrimenti contatta il broker che gestisce l’informazione inoltrandogli l’operazione di query o di update da effettuare. Si dice che il broker in questa configurazione agisce da NGSI proxy. Vediamo un esempio pratico.

Per funzionare, la pull federation sfrutta una chiamata dell’interfaccia **NGSI-9**: la **registerContext**. Attraverso questa operazione è possibile specificare il context provider per una specifica entità. Il context provider è di solito un altro Orion che memorizza effettivamente i dati relativi all’informazione di contesto. Se Orion riceve una query o un update per un’entità per cui non esiste corrispondenza nel proprio database locale ma un context provider è registrato per quella data entità, il broker inoltrerà l’operazione di query o di update al provider. Questo processo è completamente trasparente all’applicazione che effettua la richiesta. Il provider non deve essere necessariamente un altro broker, ma qualsiasi applicazione che implementi le operazioni di query e update secondo gli standard previsti da **NGSI-10**.

1. **Messaggio n.1**, l’applicazione regista il context provider sul broker per l’attributo *temperature* dell’entità *Street4*. Il provider ha indirizzo <http://sensor48.mycity.com/ngsi10> come si può vedere dal tag *<providingApplication>*

```
(1) <?xml version="1.0"?>
2  <registerContextRequest>
3  <contextRegistrationList>
4  <contextRegistration>
5  <entityIdList>
6  <entityId type="Street" isPattern="false">
7  <id>Street4</id>
8  </entityId>
9  </entityIdList>
10 <contextRegistrationAttributeList>
11 <contextRegistrationAttribute>
12 <name>temperature</name>
```

```

13 <type>float</type>
14 <isDomain>false</isDomain>
15 </contextRegistrationAttribute>
16 </contextRegistrationAttributeList>
17 <providingApplication>http://sensor48.mycity.com/ngsi10
18 </providingApplication>
19 </contextRegistration>
20 </contextRegistrationList>
21 <duration>P1M</duration>
22 </registerContextRequest>
```

2. Messaggio n.2: un client effettua un query sull'entità Street4

```

(1) <queryContextRequest>
2 <entityIdList>
3 <entityId type="Street" isPattern="false">
4 <id>Street4</id>
5 </entityId>
6 </entityIdList>
7 <attributeList>
8 <attribute>temperature</attribute>
9 </attributeList>
10 </queryContextRequest>
```

3. Messaggio n.3: il broker non ha memorizzato nel proprio database locale l'entità Street4, ma ha registrato il context provider per l'entità all'indirizzo <http://sensor48.mycity.com/ngsi10>, così inoltra la query al provider.

```

(1) <queryContextRequest>
2 <entityIdList>
3 <entityId type="Street" isPattern="false">
4 <id>Street4</id>
5 </entityId>
```

```

6 </entityIdList>
7 <attributeList>
8 <attribute>temperature</attribute>
9 </attributeList>
10 </queryContextRequest>
```

4. **Messaggio n.4:** il provider risponde alla richiesta

```

(à) <?xml version="1.0"?>
2 <contextElementResponse>
3 <contextElement>
4 <entityId type="Street" isPattern="false">
5 <id>Street4</id>
6 </entityId>
7 <contextAttributeList>
8 <contextAttribute>
9 <name>temperature</name>
10 <type>float</type>
11 <contextValue>16</contextValue>
12 </contextAttribute>
13 </contextAttributeList>
14 </contextElement>
15 <statusCode>
16 <code>200</code>
17 <reasonPhrase>OK</reasonPhrase>
18 </statusCode>
19 </contextElementResponse>
```

5. **Messaggio n.5:** il broker inoltra la risposta al client. Notare nella risposta il tag *<details>* che informa il client sul provider che ha effettivamente servito la richiesta.

```

(à) <contextElementResponse>
2 <contextElement>
```

```

3 <entityId type="Street" isPattern="false">
4 <id>Street4</id>
5 </entityId>
6 <contextAttributeList>
7 <contextAttribute>
8 <name>temperature</name>
9 <type>float</type>
10 <contextValue>16</contextValue>
11 </contextAttribute>
12 </contextAttributeList>
13 </contextElement>
14 <statusCode>
15 <code>200</code>
16 <details>Redirected to context provider
17 http://sensor48.mycity.com/ngsi10
18 </details>
19 <reasonPhrase>OK</reasonPhrase>
20 </statusCode>
21 </contextElementResponse>
```

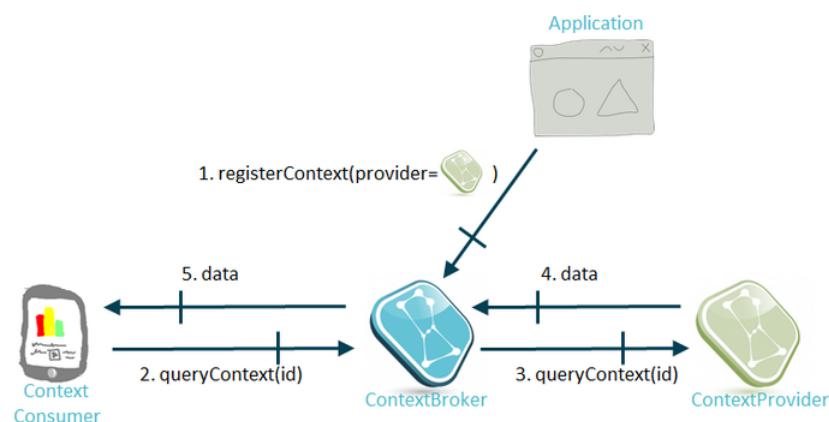


Figura 2.11: Pull federation

2.6 Introduzione ai big data

Se diamo uno sguardo alle tecnologie e ai servizi che oggi troviamo in rete è facile arrivare alla conclusione che tutto gira attorno ai dati. Come utenti svolgiamo un duplice ruolo, siamo sia produttori che consumatori di grosse quantità di dati. Svolgendo le normali operazioni di tutti i giorni sul web: effettuare una ricerca su google, acquistare un prodotto da amazan, prenotare un volo su expedia, contribuiamo più o meno consapevolmente alla generazione di dati, lasciando tracce indelebili sulla nostra identità gusti ed interessi. Al giorno d'oggi inoltre, non solo la quantità di dati generati cresce in modo esponenziale ma aumenta considerevolmente anche la frequenza con la quale questi dati vengono generati. La sfida e il business risiedono nella capacità di estrarre da questa enorme mole informe di dati delle informazioni utili: che rappresentano il valore aggiunto sui dati, ciò che permette di individuare i trends di un determinato mercato o le relazioni tra set di dati diversi.

Le grandi compagnie, in primis google, hanno da tempo capito come i dati a loro disposizione possono migliorare il servizio offerto all'utente; basti pensare come google mostra le pubblicità in funzione alle nostre ricerche o come amazon mostri i nuovi prodotti in base ai nostri gusti e interessi pregressi.

La necessità di memorizzare grosse quantità di dati prima inimmaginabili e di elaborarli nel breve periodo ha mutato l'idea stessa di storage e come i dati vengono strutturati per la memorizzazione. Il problema principale che gli ingegneri hanno dovuto risolvere è stato quello di come scalare il proprio sistema nel momento in cui i dati iniziano a crescere oltre misura come nel fenomeno dei big data. Vediamo quindi una breve panoramica dei possibili approcci a questo problema centrale.

2.6.1 Scale-up

Secondo questo paradigma il problema dell'elaborazione di grosse quantità di dati viene risolto utilizzando calcolatori dalle capacità di elaborazione impressionanti ma anche dai costi altrettanti importanti. Quando il volume

dei dati aumenta, l'approccio è quello di spostarsi su un server ancora più potente. Il vantaggio di questa soluzione è che l'architettura non richiede modifiche durante il processo di crescita dei dati. Il limite è evidente, per quanto potente possa essere un singolo host, ad un certo punto, il volume dei dati potrebbe crescere oltre le capacità di elaborazione messe a disposizione dalle moderne tecnologie. Il principale svantaggio rimane comunque il costo eccessivo.

2.6.2 Scale-out

Invece di espandere il sistema verso hardware sempre più performante, l'approccio scale-out prevede di distribuire il processo di memorizzazione ed elaborazione su più e più macchine. Se il volume dei dati raddoppia semplicemente si usano due server al posto di uno. Il beneficio evidente di questo approccio è che i costi rimangono molto più bassi rispetto al paradigma scale-up a parità di volumi dei dati. Lo svantaggio risiede nella necessità di sviluppare strategie per dividere i dati tra i diversi server dell'infrastruttura che di solito richiedono soluzioni software complesse.

2.6.3 Fattori limitanti

Dunque, qualunque dei due paradigmi si scelga per scalare il sistema verso il nuovo volume dei dati è possibile individuare per entrambe le parti dei fattori limitanti che ne ostacolano l'effettiva implementazione.

- **Scale-out:** le difficoltà introdotte dalla complessità di gestione della concorrenza nel sistema diventano significanti. Utilizzare in concorrenza diverse cpu o diversi host è compito complesso, che ricade sul progettista software e richiede risorse umane e tecniche considerevoli e quindi ulteriori costi.
- **Scale-up:** la crescita, secondo la legge di Moore, delle capacità elaborative delle CPU è molto più veloce rispetto alla crescita delle velocità di risposta dei dischi e delle reti di comunicazione. Il rischio è dunque quello di costruire sistemi con imponenti capacità elaborative che non

posso essere sfruttati in quanto il sistema di storage non è in grado di fornire i dati ad una frequenza sufficiente da tenere occupate le cpu per la maggior parte del tempo.

2.6.4 La dura realtà

Idealmente un'architettura scale-out è composta da singoli host ciascuno impegnato nell'elaborazione di un sottoinsieme dell'insieme dei dati globale per produrre la sua porzione di risultato finale. La realtà raramente è così semplice. Spesso gli host possono avere bisogno di comunicare tra di loro per scambiarsi porzioni di dati o risultati parziali. Questa necessità pone il sistema in una situazione di vulnerabilità da due punti di vista: i **colli di bottiglia** e il **rischio di fallimento**. Se una porzione di dati è richiesta da ogni host nel sistema, c'è un rischio di conflitto e di ritardi, in quanto i diversi client concorrenti devono accedere ai dati comuni. Se, per esempio, in un sistema con 25 host, un singolo host deve essere consultabile da tutti i restanti, le prestazioni complessive del sistema saranno determinate dalle capacità di quest'ultimo. Peggio ancora, se durante il processo si verifica un errore; tutto il processo di elaborazione cade a pezzi. Studi su questi tipi di cluster hanno dimostrato tale rischio; inoltre anche se il carico di lavoro è stato distribuito tra gli host, spesso viene usato un sistema di storage condiviso per contenere tutti i dati. Invece di condividere risorse, i singoli componenti di un sistema dovrebbero essere più indipendenti possibili gli uni dagli altri, in modo da consentire a ciascuno di procedere in modo assolutamente indipendentemente.

Immaginiamo di avere un grande insieme di dati, ad esempio, 1000 terabyte e che sia necessario eseguire una serie di quattro operazioni su ogni pezzo di dati dell'insieme. Vediamo i diversi modi di risolvere questo problema. Una soluzione tradizionale scale-up vedrebbe l'utilizzo di un grande server collegato a un altrettanto impressionante sistema di archiviazione, quasi certamente avremmo un uso di tecnologie per la comunicazione quali fibra ottica in modo da ottimizzare la larghezza di banda di archiviazione. Il sistema eseguirà il compito, ma è soggetto ad un limite di I/O; infatti anche gli switch di fascia alta hanno un limite su quanto velocemente i dati pos-

sono essere consegnati. In alternativa, l'approccio di tipi scale-out sarebbe quello di prevedere un gruppo di 1.000 macchine, ciascuna con 1 terabyte di dati diviso in quattro quadranti, in cui ciascun quadrante di macchine abbia assegnato uno dei quattro compiti da svolgere. Il software di gestione di cluster dovrebbe coordinare il movimento dei dati all'interno del cluster per garantire che ogni porzione dei dati subisca tutte e quattro le fasi di lavorazione. Ogni porzione dei dati verrà eseguita su un determinato host per subire una delle quattro operazioni, ma successivamente avrà bisogno di essere spostato per l'elaborazione sugli altri tre quadranti, in questo modo vi è un consumo elevatissimo di banda di rete per eseguire il task. Ricordando che la potenza di elaborazione è aumentata più velocemente rispetto alle velocità di rete o delle operazioni di I/O di un disco, quello che è giusto chiedersi è se questo è il modo migliore per affrontare il problema? L'esperienza recente ci dice che la risposta è no, e che un approccio alternativo è quello di **evitare di spostare i dati e invece spostare l'elaborazione**. Utilizzare quindi un cluster come appena accennato, ma senza suddividerlo in quadranti; invece, fare in modo che ciascuno dei nodi sia in grado di eseguire tutte le quattro fasi di elaborazione sui dati in possesso localmente, spostando la quantità più piccola possibile di dati. Con un po' di fortuna, si potrebbe avere la necessità di trasmettere i dati una sola volta e le uniche cose che viaggiano attraverso la rete saranno i binari del programma e rapporti di stato o di errore.

2.6.5 Di cosa c'è bisogno?

Ripensando allo scenario mostrato nel paragrafo precedente, si commetterebbe un errore a pensare che il problema sia risolto gestendo l'aspetto del movimento ed dell'elaborazione dei dati. In realtà la pianificazione dei processi, la gestione degli errori, e il coordinamento tra i nodi sono i punti più delicati della questione. Se dovessimo implementare da zero i meccanismi necessari per l'individuazione dei nodi sui quali eseguire l'elaborazione, la logica per l'esecuzione distribuita, e quella che unisce tutti i risultati parziali nel risultato complessivo, non avremmo guadagnato molto dal vecchio modello. Prima avevamo bisogno di gestire in modo esplicito il partiziona-

mento dei dati; ora i dati non si muovono ma dobbiamo gestire ben altri problemi. Il risultato sarebbe semplicemente quello di aver scambiato un problema difficile con uno altrettanto complesso.

Abbiamo dunque bisogno di un sistema che gestisce la maggior parte della logica del nostro cluster in modo trasparente e in questo modo consente allo sviluppatore di pensare esclusivamente in termini di soluzione del problema di business. A queste chiare necessità risponde Hadoop che introdurremo nel paragrafo successivo.

2.7 Hadoop

Tutto ebbe inizio quando Google, tra il 2003 e il 2004, rilasciò due articoli accademici che descrivevano la piattaforma che l'azienda proponeva per processare i dati su larga scala in modo altamente efficiente. I due articoli trattavano rispettivamente di **GFS** [5](Google File System) e **MapReduce** [6].

Doug Cutting iniziò allora un'effettiva implementazione di entrambi i due componenti del sistema di Google; Cutting chiamò il sistema Hadoop: nome del pupazzo di pezza di suo figlio. Hadoop è quindi una piattaforma o framework open source che mette a disposizione un'implementazione sia di GFS che di MapReduce, permettendo il processamento di insiemi di dati molto grossi attraverso un sistema clusterizzato di hardware a basso costo. Hadoop crebbe molto rapidamente anche grazie al supporto economico e tecnico di aziende leader come Yahoo che è tutt'oggi il maggiore contributore del progetto.

Al giorno d'oggi il progetto Hadoop è composto da numerose componenti software e sotto progetti, è più lecito infatti parlare di ecosistema Hadoop. I due principali componenti restano però l' Hadoop Distributed File System (**HDFS**) e **MapReduce**, che come abbiamo già anticipato, sono l'effettiva implementazione delle tecnologie proposte da Google.

- **HDFS**: è un filesystem distribuito in grado di immagazzinare insiemi di dati molto grandi usando il paradigma scale-out attraverso cluster di nodi.

- **MapReduce:** è un paradigma di processamento dei dati che definisce delle specifiche stringenti su come i dati devono essere forniti in input ed in output durante due passi successivi di elaborazione chiamati appunti *map* e *reduce*.

Sia HDFS che MapReduce godono delle caratteristiche di cui si era parlato nel paragrafo 2.6.5 e cioè:

- Entrambi sono pensati per essere eseguiti su un cluster di nodi server.
- Entrambi sono in grado di scalare le loro capacità attraverso l'aggiunta di più server secondo il paradigma scale-out.
- Entrambi hanno meccanismi per l'identificazione e la gestione degli errori.
- Entrambi offrono i loro servizi in modo trasparente all'utente, permettendo allo sviluppatore di concentrarsi sul problema effettivo da risolvere.
- Entrambi hanno un'architettura dove un software gira su un server fisico e controlla tutti gli aspetti dell'esecuzione del sistema.

Ciò di cui c'era bisogno adesso c'è.

2.7.1 HDFS

HDFS è un filesystem abbastanza diverso da quelli che siamo abituati ad utilizzare sui nostri sistemi desktop. HDFS non implementa lo standard POSIX, è un filesystem distribuito nel senso che distribuisce i blocchi dei file attraverso più nodi del cluster. Le sue caratteristiche principali sono:

- HDFS memorizza i file in blocchi tipicamente di almeno 64 MB, una valore molto maggiore rispetto agli usuali 4-32 KB dei filesystem tradizionali.
- HDFS è altamente efficiente su richieste di lettura per file molto grandi, ma fornisce scarse prestazioni su richieste per file di piccole dimensioni.

- HDFS è ottimizzato per processi che sono detti di write-once e read-many.
- Su ciascun nodo del cluster gira un processo chiamato **DataNode** che memorizza i blocchi dei file. I DataNode del cluster sono coordinati da un processo master detto **NameNode** che gira su di un host separato. Il NameNode ha il compito di tenere traccia della posizione dei blocchi dei file sui DataNode e della gestione degli errori.
- HDFS gestisce i fallimenti del disco attraverso la replicazione di blocchi. Ciascun blocco di un file è memorizzato su più nodi all'interno del cluster, il NameNode monitora costantemente i report inviati da ciascun DataNode per assicurarsi che i fallimenti per ciascun blocco non superino un certo fattore di replica. Se questo accade, il NameNode schedula l'aggiunta di un'altra copia del blocco all'interno del cluster.

2.7.2 MapReduce

Anche se MapReduce come tecnologia è relativamente nuova, in realtà si basa su concetti consolidati nella scienza dell'informazione, in particolare si ispira ad una tipologia di approccio in cui si cerca di esprimere le operazioni che dovrebbero poi essere applicate a ogni elemento in un insieme di dati. Infatti i concetti di funzioni chiamati map e reduce provengono dai linguaggi di programmazione funzionali.

Un altro concetto chiave alla base di MapReduce è quello del "divide et impera", in cui un singolo problema è diviso in più singole attività secondarie. Questo approccio diventa ancora più potente quando le attività secondarie vengono eseguite in parallelo; in un caso ideale, un compito che richiede 1.000 minuti potrebbe essere completato in 1 minuto da 1.000 attività secondarie parallele.

MapReduce è un paradigma di elaborazione che si basa su questi principi; fornisce una serie di trasformazioni da una sorgente di dati in ingresso ad un insieme di dati risultanti in uscita. Nel caso più semplice, i dati di ingresso sono dati in pasto alla funzione map mentre i dati temporanei

risultanti dalle operazioni di map alimentano la una funzione reduce. Lo sviluppatore definisce solo le trasformazioni dei dati tra le operazioni di map e di reduce in quanto MapReduce gestisce anche il processo di applicazione delle trasformazioni sui dati nel cluster in parallelo. Sebbene i concetti su cui si basa Hadoop non sono certo nuovi in letteratura, un punto di forza di Hadoop è nel modo in cui ha portato questi principi insieme in una piattaforma facilmente accessibile e ben progettata. A differenza dei tradizionali database relazionali che richiedono dati strutturati con schemi ben definiti, MapReduce e Hadoop funzionano meglio su dati semi-strutturati o non strutturati. Invece di dati conformi agli schemi rigidi, il requisito è invece che i dati siano forniti alla funzione map come una serie di coppie di valori di chiave. L'output della funzione map è una serie di altre coppie di valori chiave, e la funzione reduce esegue un'operazione di aggregazione sui risultati parziali per raccogliere i risultati finali. Hadoop inoltre fornisce un'interfaccia standard per le funzioni map e reduce, le implementazioni di funzioni map e reduce vengono dette mapper e reducer. Un job tipico di MapReduce consisterà in un certo numero di mapper e di reducer, di solito sia i mapper che i reducer hanno dei compiti relativamente semplici. Lo sviluppatore si deve concentrare solo su come esprimere la trasformazione tra la sorgente di dati e l'insieme finale dei risultati; Hadoop gestisce tutti gli aspetti relativi all'esecuzione del lavoro, di parallelizzazione, e coordinamento dei nodi.

Questo ultimo punto è forse l'aspetto più importante di Hadoop. La piattaforma si prende la responsabilità di ogni aspetto riguardante l'esecuzione e il processamento di tutti i dati. Dopo che l'utente definisce i criteri chiave per il job, tutto il resto diventa responsabilità del sistema. Dal punto di vista della dimensione dei dati, lo stesso job in MapReduce può essere applicato a insiemi di dati e di cluster di qualsiasi dimensione. Se i dati hanno dimensione di 1 GB ed il cluster è composto da un singolo host, Hadoop programmerà l'elaborazione di conseguenza. Allo stesso modo se i dati sono di 1 PB e sono distribuiti tra mille macchine, Hadoop fa del suo meglio per utilizzare tutti gli host ed eseguire il job nel modo più efficiente possibile. Dal punto di vista dell'utente, la dimensione dei dati e del cluster è trasparente, e, a parte influenzare il tempo necessario per elaborare il

processo, non c'è differenza sul modo in cui l'utente interagisce con Hadoop nei due casi.

2.7.3 HDFS e MapReduce insieme

E' possibile usare HDFS e MapReduce separatamente, ma questi hanno anche maggiore efficienza se combinati assieme. HDFS usato senza MapReduce è un'ottima piattaforma di storage su larga scala. Anche se MapReduce può leggere dati da sorgenti non HDFS, la metodologia del suo processamento dei dati si allinea così bene con HDFS che l'utilizzo congiunto di questi due componenti rappresenta la casistica più diffusa in Hadoop.

Quando un job in MapReduce è eseguito, Hadoop necessita di decidere dove eseguire il codice in modo da processare i dati nel modo più efficiente possibile. Se tutti gli host del cluster su cui gira MapReduce prelevano i loro dati da un singolo host di storage condiviso, il modo in cui Hadoop esegue i job non ha molta importanza in quanto il sistema di storage è una risorsa condivisa che causa contesa. Ma se il sistema di storage è HDFS, ciò permette a MapReduce di eseguire il processamento dei dati che contengono il dato d'interesse, sfruttando il principio per cui è meno costoso muovere i dati di processamento che i dati stessi.

Il più comune modello di deployment per Hadoop prevedere un cluster HDFS ed un cluster MapReduce sullo stesso insieme di server. Ogni host che ospita i dati attraverso HDFS ospita anche un componente MapReduce che è in grado di schedulare ed eseguire il processamento dei dati. Quando un job è sottomesso su Hadoop, esso utilizza un processo di ottimizzazione che tenta il più possibile di schedulare il processamento dei dati sugli host dove i dati risiedono localmente, minimizzando il traffico di rete e massimizzando le performance.

Nello scenario precedente nel quale si processava un task composto da quattro step differenti su 1 PB di dati distribuiti tra mille host del cluster, il modello MapReduce di Hadoop eseguirebbe il processamento attraverso una funzione map su ciascuna porzione di dati sull'host dove il dato effettivamente risiede e successivamente riutilizzerebbe il cluster per eseguire la

funzione reduce per raccogliere i risultati individuali nell'insieme finale dei risultati.

L'abilità del progettista sta dunque nel suddividere il problema nella migliore combinazione di funzioni map e reduce che possono essere eseguite parallelamente sul cluster. La soluzione migliore è spesso quella di usare più job MapReduce dove l'output del primo si trasforma nell'input del secondo e così via.

2.7.4 Architettura HDFS e MapReduce

Sia HDFS che MapReduce, come spiegato sopra, sono software di cluster che mostrano un'architettura con caratteristiche comuni:

- Entrambi seguono un architettura dove un cluster di nodi slave (lavoratori) sono gestiti da un nodo coordinatore/master.
- Il nodo master in entrambi i casi (**NameNode** per HDFS e **JobTracker** per MapReduce) monitora lo stato del cluster e gestisce i fallimenti, sia spostando i blocchi dei dati tra i nodi, sia rischedulando i job falliti.
- I processi (**DataNode** per HDFS e **TaskTracker** per MapReduce) sono responsabili di effettuare il lavoro sui nodi fisici: memorizzazione dei blocchi ed esecuzione dei job, ricevendo istruzioni dal NameNode o dal JobTracker e riportando lo stato del job.

2.7.5 Quando usare Hadoop

Hadoop è una bel prodotto, ma è necessario capire i contesti nel quale può essere conveniente usarlo. Per prima cosa è necessario dire che Hadoop è un sistema batch, di conseguenza esso non fornirà un risultato finchè tutto il lavoro sarà completato. In un cluster abbastanza grande il risultato può essere generato relativamente in fretta, ma rimane il fatto che le risposte non sono immediate. Conseguentemente Hadoop da solo non è la soluzione per query con bassi tempi di latenza come quelle di un sito web o di sistemi in tempo reale. Quando Hadoop è in esecuzione su grossi insiemi di

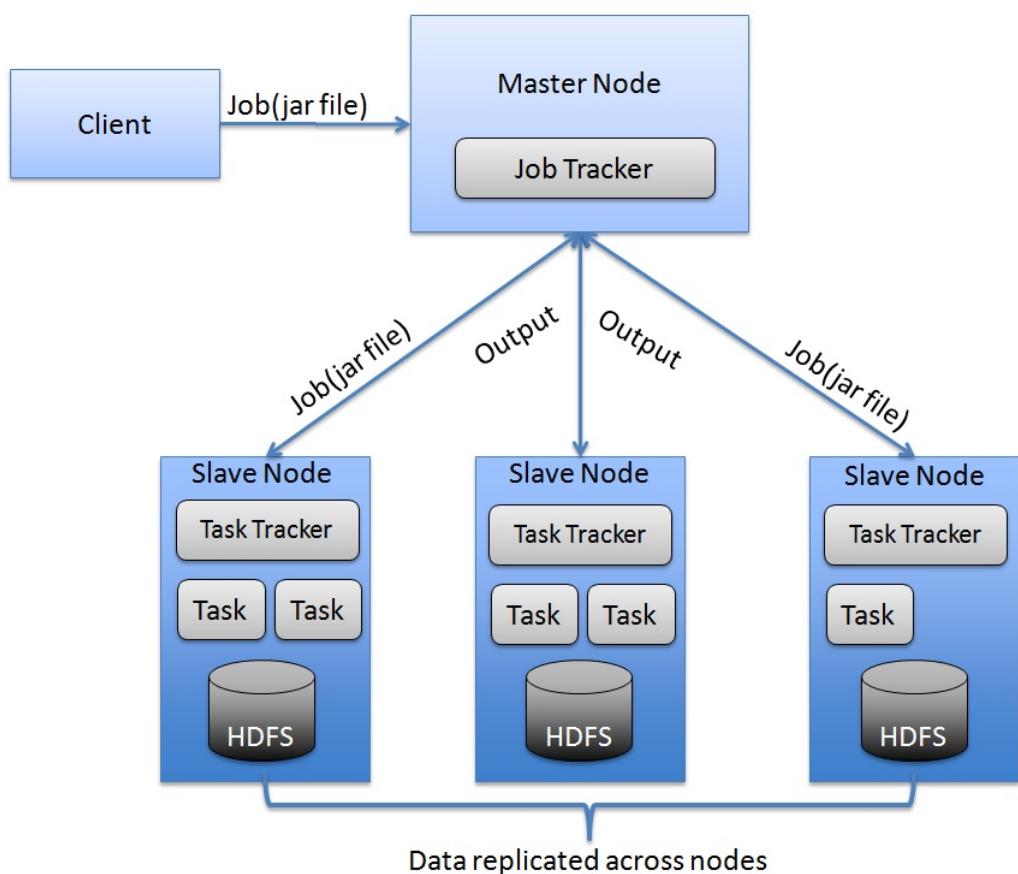


Figura 2.12: Architettura Hadoop

dati l'overhead dovuto all'impostazione del job, necessario per determinare quali task devono essere eseguiti su ciascun nodo, è una parte trascurabile del tempo totale di esecuzione. Nel caso invece di piccoli insiemi di dati l'overhead si avverte soprattutto nell'esecuzione di semplici job MapReduce.

Come fanno allora Google o Yahoo ad avere tempi risposta inferiori? La soluzione è usare Hadoop insieme ad altre tecnologie, ad esempio Google attraverso il proprio web crawler recupera gli aggiornamenti sulle pagine web, successivamente MapReduce processa questa enorme quantità di dati e da questo insieme si producono gli indici che un insieme di server MySQL usa per servire gli utenti finali.

2.7.6 Hive

Il paradigma MapReduce si è rivelato una soluzione vincente per l'elaborazione distribuita di grosse quantità di dati. Rimane il fatto che la maggior parte dei programmatori non è avvezza a questa modalità di accesso ed elaborazione dei dati. Inoltre, i dati risultanti dalle operazioni di reduce sono scritti in modo distribuito su HDFS ed in teoria, per essere consultati, necessitano di altri job MapReduce. Tutti i programmatori che scrivono applicazioni che interrogano dati hanno sicuramente molta più familiarità nell'uso di un linguaggio SQL-like piuttosto che MapReduce. Infine MapReduce impone allo sviluppatore di utilizzare java per accedere e manipolare i dati sul cluster. Per tutta questa serie di motivazioni è nato il progetto Hive.

Hive permette di accedere a dati strutturati e non strutturati, distribuiti sul cluster, attraverso un linguaggio SQL-like di nome HiveQL. Di fatto Hive aggiunge un ulteriore livello di astrazione sopra il paradigma MapReduce, permettendo al programmatore di accedere ai dati senza bisogno di conoscere java o MapReduce. Le query SQL vengono tradotte in operazioni MapReduce da un componente di Hive detto HDInsight e quindi eseguite sul cluster.

Nel nostro progetto abbiamo utilizzato Hive per prelevare, per conto dei widget di wirecloud, i dati delle misurazioni. E' stato realizzato un server

(restServerHive) che accetta richieste provenienti dai widget di wirecloud, effettua le query e restituisce i dati al widget in un formato adeguato.

2.8 Cosmos

Cosmos è l'implementazione del Big Data Generic Enabler [13], gira su un cluster di macchine che eseguono task in parallelo allo scopo di processare grosse quantità di dati in modo altamente efficiente. Cosmos offre anche delle funzionalità che facilitano il deployment del cluster. Cosmos di fatto si basa sull'ecosistema Hadoop, al quale aggiunge proprie API per la gestione sia del cluster che dei servizi e degli utenti. In breve possiamo dire che Cosmos permette di:

- effettuare operazioni di I/O su uno **storage cluster** (Infinity) basato su filesystem HDFS.
- creazione, uso ed eliminazione di **computing cluster** basati su MapReduce.
- accesso e consultazione dei dati sullo storage cluster attraverso sistemi SQL-like come Hive o Pig.
- gestione della piattaforma in tutti i suoi aspetti: gestione dei servizi, degli utenti, dei cluster di storage e di computing attraverso Cosmos API o CLI.

Cosmos gestisce quindi due tipologie di cluster: storage e computing. La prima deve essere pensata come un insieme di macchine usate per effettuare esclusivamente operazioni di memorizzazione di grosse quantità di dati in modo distribuito, la seconda invece come un'insieme di risorse di calcolo per operazioni di processamento dati.

Cosmos deve essere usato in tutti quei contesti in cui risulta necessario processare grosse quantità di dati, sia che questi siano stati memorizzati precedentemente (Big Data Batch Processing) sia che questi siano ricevuti in tempo reale (Big Data Streaming Processing) con l'obiettivo di estrarre

dai dati informazioni che erano nascoste nei dati originali. L'overhead introdotto dalle operazioni di gestione del cluster, identificazione delle risorse, parellizzazione dei task, gestione degli fallimenti sul cluster, determinano tempi di risposta che possono essere troppo elevati in particolari contesti. Se infatti in scenari di batch processing questo può non essere un problema, il caso di processamento in tempo reale può risultare sensibile a ciò e sarà necessario integrare Cosmos con altre tecnologie.

Attraverso i computing cluster, Cosmos mette a disposizione servizi di computing sia di tipo sia tipo batch che real time, Cosmos fornisce anche servizi di storage attraverso il cluster Infinity.

Il servizio di computing permette la creazione, configurazione e coordinamento della macchine del cluster attraverso CLI o REST API. Se il volume dei dati è sufficientemente elevato da permettere un efficiente strategia di processamento in parallelo dei dati, Cosmos garantisce grossa efficienza nell'elaborazione, fornendo risultati in tempi impensabili per altri sistemi di elaborazione. L'efficienza di processamento è ottenuta sfruttando il paradigma MapReduce, come detto di fatto Cosmos è un ecosistema Hadoop ampliato. I dati sono distribuiti in un elevatissimo numero di macchine e processati localmente da singoli task in modo del tutto trasparente all'utente sfruttando le funzioni map e reduce del paradigma. Questa semplicità da d'uso del cluster ha un prezzo: i tempi di latenza. Le operazione di reduce non possono iniziare se non dopo che le operazioni di map abbiano terminato, inoltre un certo periodo di tempo è usato per l'identificazione delle risorse e i dati nel cluster per il processamento. Tutto questo aumenta il tempo di risposta e come già anticipato ciò richiede, in contesti di risposta stringenti, l'integrazione con altre tecnologie.

Se il servizio di computing permette di lavorare sui dati in parallelo, il servizio di storage fa in modo che i dati siano disponibili per le operazioni di computing. Di solito il servizio di storage ha elevatissima capacità di memorizzazione e offre dei modi per accedere ai dati distribuiti sul cluster in modo efficiente. Un sistema di dati distribuito è una rete di computer dove l'informazione è memorizzata in più nodi, ciascun nodo possiede una certa parte dell'informazione. I sistemi di storage distribuiti fanno uso di repli-

cazione su più nodi dei dati, lo stesso dato dunque è distribuito e replicato in nodi differenti della rete allo scopo di facilitare il recovery dell'informazione quando un nodo fallisce. Questo tipo di sistema di memorizzazione è usato in quanto garantisce scalabilità e grosse capacità di memorizzazione che altrimenti non si potrebbero raggiungere con sistemi centralizzati. Il sistema prevede un nodo centrale di gestione che fornisce anche una vista unitaria dello storage totale. Questa vista unitaria fornisce servizi per la creazione, modifica e cancellazione dei file distribuiti. Per fare ciò il sistema di gestione mantiene una lista dei file e delle posizioni dei blocchi dei file sui vari nodi. In Cosmos lo standard per il sistema di storage è HDFS, ma qualsiasi altro sistema di storage distribuito può essere utilizzato come ad esempio: CFS (Cassandra File System) o GFS (Google File System).

L'uso dei due servizi: computing e storage è assolutamente indipendente l'uno dall'altro, anche se l'utilizzo congiunto è la configurazione raccomandata.

2.8.1 Architettura

Il Big Data GE e quindi anche Cosmos estendono l'architettura dell'ecosistema Hadoop. Nei paragrafi precedenti è stata fatta una trattazione dettagliata di questa architettura. Molto brevemente possiamo ribadire che Hadoop è un framework che permette il processamento distribuito di ampi insiemi di dati su cluster di computer sfruttando il paradigma MapReduce. Hadoop è pensato per scalare facilmente da un singolo server fino a migliaia di macchine; hadoop inoltre offre un sistema di identificazione e gestione dei fallimenti attraverso la replica dei dati distribuiti. Due sono i componenti alla base del sistema Hadoop: HDFS per lo storage ditribuito e MapReduce per l'elaborazione in parallelo dei dati.

HDFS rappresenta il livello più basso dell'architettura; i dati sono divisi in blocchi di 64MB e distribuiti su diversi nodi della rete detti DataNode. Esiste un nodo speciale detto NameNode che ha il compito di memorizzare la posizione dei blocchi di ciascun file, e effettuare la gestione del filesystem: creazione, cancellazione, rinomina dei file. A questo livello, qualunque filesystem distribuito supportato da Hadoop può essere utilizzato per sostituire

HDFS; alcuni esempi sono Amazon S3, CloudStore, FTP FileSystem, GFS etc.

Sopra HDFS si trova il livello di MapReduce composto da una serie di nodi detti TaskTracker che effettuano operazione di map e di reduce sui dati presenti in HDFS. Esiste un nodo speciale detto JobTracker che decide quale task e quale dato assegnare a ciascun TaskTracker. La lista dei task da eseguire è contenuta in un file .jar che è replicato diverse volte su HDFS dal JobTracker. La macchina sulla quale viene effettuato il carimento del file .jar ed invocata l'esecuzione dei task è detta JobClient.

L'architettura può variare in base alle dimensioni del cluster o delle performance richieste. Ad esempio, se il cluster è piccolo, due o più nodi potrebbero essere uniti in uno unico con conseguente diminuzione delle prestazioni della CPU ma anche con un miglioramento delle latenze sulla transazioni tra i due nodi. E' normale trovare sulla stessa macchina sia il TaskTracker che il DataNode o anche avere il JobTracker eseguito sulla stessa macchia del NameNode. Situazione estreme possono prevedere JobTracker, NameNode, TaskTracker e Datanode insieme.

Quando il job ha terminato, i risultati vengono scritti su HDFS; se i dati devono essere acceduti dagli utenti con tempi latenza bassi è preferibile storare i dati su sistemi NoSQL.

Fin qui abbiamo descritto molto brevemente quello che già sapevamo; cosa aggiunge Cosmos all'architettura Hadoop?

L'architettura di Cosmos si basa su un nodo Mater (Master Node) che esegue il software di gestione e si comporta da interfaccia verso l'utente finale. L'utente finale è solitamente un'applicazione che attraverso le Cosmos API d'amministrazione richiede la creazione dei cluster di computing o di storage.

La richiesta di creazione di un cluster determina la creazione di un Head Node. L'Head Node è responsabile esclusivamente della gestione del cluster di storage o del computing cluster, esso quindi non memorizza e non elabora nulla per conto del cluster. Gli Slave Node, anch'essi creati alla richiesta di creazione del cluster, hanno invece il compito di eseguire i task o memorizzare i dati.

Come è possibile vedere dalla figura in basso, sull'Head Node girano il NameNode e il JobTracker, mentre sugli slave node troviamo il DataNode e il TaskTracker. Questi servizi Hadoop possono essere usati per configurare Head e Slave Node di Cosmos, ma molte altre soluzioni possono essere usate allo stesso modo:

- CFS (Cassandra File System) + MapReduce
- HDFS (Hadoop Distributed File Sysmte) + HBase (NoSQL distributed Database)
- HDFS + MapReduce + Cascading
- HDFS + MapReduce + Hive

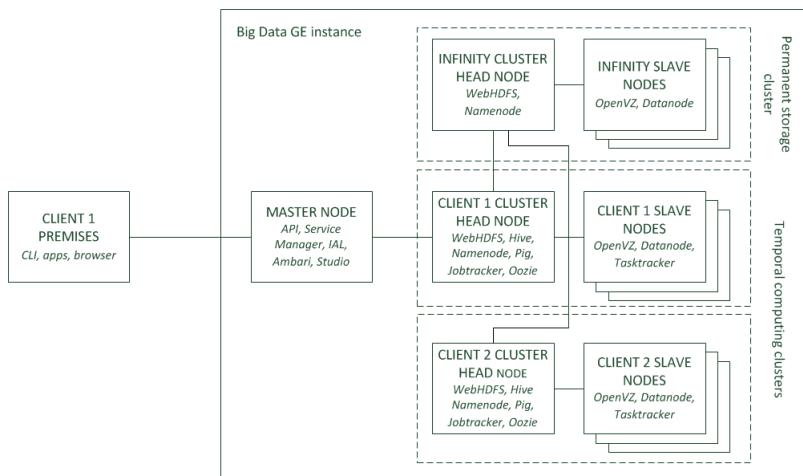


Figura 2.13: Architettura Cosmos

2.8.2 API

Le API messe a disposizione da Cosmos [14, 15]sono molteplici, ne forniamo di seguito un elenco:

- **Admin API:** Queste API sono usate per compiti di amministrazione della piattaforma come creazione, modifica, cancellazione dei cluster e degli utenti oppure amministrazione dei task o degli spazi di storage.

Queste funzionalità sono raggiungibili attraverso il nodo Master della piattaforma che è comune a tutti gli utenti.

- **Comandi Hadoop:** CLI di Hadoop usata sia per amministrare HDFS sia per l'esecuzione di job MapReduce. Hadoop può essere completamente governato attraverso un'interfaccia CLI⁶. Tra questi comandi il più importante è *hadoop fs* che è usato per l'amministrazione del filesystem HDFS: creazione, cancellazione, modifica di file e directory.
- **Hadoop API:** Java API usate per programmare client HDFS o applicazioni MapReduce⁷
- **HDFS RESTful API:** API per l'amministrazione di HDFS. Queste API includono i protocolli WebHDFS, HttpFS e Infinity.
 - **WebHDFS:** Quando non è possibile accedere direttamente alle macchine sul cluster ed utilizzare su di esse l'interfaccia CLI per la gestione del filesystem è possibile usare questa tipologia di API. Attraverso di esse è possibile gestire i permessi, i proprietari, creare, rinominare, leggere file e directory. Le API sono raggiungibili sulla porta TCP 50070 del NameNode del cluster⁸. E' importante sottolineare che alcune operazioni come CREATE e APPEND richiedono un processo a due step. Nel primo passo si invia la richiesta di I/O al NameNode del cluster, la risposta dal NameNode a questa richiesta indica l'URL del DataNode sul quale il dato può essere scritto o letto. Chiaramente WebHDFS gestisce esclusivamente il filesystem distribuito, quindi non sono previste operazioni per l'esecuzioni di job MapReduce.
 - **HttpFS:** implementa le stesse API di WebHDFS ma il suo comportamento è leggermente differente. Quando si usa WebHDFS,

⁶Una lista completa dei comandi Hadoop è consultabile all'indirizzo <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/CommandsManual.html>

⁷La documentazione completa delle Java Hadoop API è consultabile al seguente indirizzo: <http://hadoop.apache.org/docs/current/api/overview-summary.html>

⁸La documentazione completa di WebHDFS è consultabile al seguente indirizzo: <http://archive.cloudera.com/cdh/3/hadoop/webhdfs.html>

sia il NameNode (Head Node) sia tutti i DataNode (Slave Node) devono possedere un indirizzo IP pubblico. Il NameNode deve avere ip pubblico per poter effettuare la richiesta di I/O, il DataNode invece per poter accedere al dato. Quest'obbligo implica l'uso di un elevato numero di indirizzi IP pubblici. HttpFS risolve questo problema agendo da gateway. In HttpFS è il NameNode stesso che effettua le operazioni di I/O sul DataNode per conto dell'utente; in questo modo non è necessario assegnare ip pubblici a tutti i nodi del cluster ma solo al NameNode.

2.9 Wirecloud

Wirecloud è un modo innovativo di realizzare applicazioni Web [16], la piattaforma è stata pensata per permettere lo sviluppo di web applications anche da parte di chi non ha particolari conoscenze di programmazione. Il concetto chiave in Wirecloud è il **mashup**; possiamo pensare al mashup come un insieme di piccoli moduli, detti **widget** o **operatori**, connessi tra di loro. Il widget di solito ha un interfaccia grafica verso l'utente, mentre l'operatore svolge esclusivamente della logica di backend e non interagisce direttamente con l'utente. Ogni widget o operatore ha uno specifico compito e fornisce un'interfaccia in ingresso ed in uscita ben definita; in questo modo è possibile collegare tra loro i diversi moduli, attraverso un processo detto wiring e sfruttare le singole funzionalità dei moduli già esistenti per realizzare nuove applicazioni: i mashup appunto. La figura di sotto mostra un esempio di wiring, come è possibile vedere i vari componenti sono connessi attraverso le loro interfacce di ingresso e uscita, queste sono sfruttate dai moduli per scambiarsi le informazioni necessarie allo svolgimento dei loro compiti. Molto brevemente, in questo esempio il primo dei componenti a partire da sinistra seleziona un servizio da una lista e passa attraverso il wiring l'informazione al quey widget; quest'ultimo interroga un server per conoscere i fornitori del servizio. L'informazione è così passata al modulo NGSI Entity to PoI che effettua una trasformazione dei dati che così vengono passati al Map Viewer per la rappresentazione dei punti di interesse

sulla mappa. Come si è visto è stata implementata una semplice applicazione di geolocalizzazione dei servizi usando moduli già pronti senza dover mettere mani ad alcuna riga di codice; nulla vieta di poter riutilizzare uno o tutti questi moduli in un'altra applicazione per scopi differenti. E' questa la filosofia che sta alla base del progetto Wirecloud.



Figura 2.14: Wiring in Wirecloud

Conoscendo le operazioni svolte dal widget, la propria interfaccia d'ingresso e d'uscita, è possibile connetterlo e sfruttare le funzionalità messe a disposizione senza bisogno di scrivere codice. Chiaramente, chi ha conoscenze di programmazione web, ha la possibilità di sviluppare i propri widget e operatori se non esiste uno già pronto che implementa le funzionalità necessarie. Nello sviluppo di questi moduli bisognerebbe tenere sempre presente che i componenti (widget o operatori) dovrebbero avere caratteristiche più generali possibili per favorire il loro riutilizzo in altri mashup in configurazioni con altri widget e operatori.

I mashup sono realizzati all'interno dei **workspaces**. Un workspace consiste nell'insieme di widget e operatori a disposizione che possono essere connessi tra loro per realizzare il mashup.

Wirecloud prevede anche un luogo virtuale: lo **store**, dove i mashup o i singoli widget e operatori possono essere scaricati o acquistati. Uno store appartiene ad uno specifico proprietario ed è pensato esclusivamente per mettere a disposizione i widget/operatori/mashup di una sola organizzazione. Un insieme di store fanno parte di un **marketplace**; il marketplace è una piattaforma pensata per ospitare differenti store di diverse compagnie

e permette di confrontare e cercare facilmente i componenti di cui c'è bisogno. Nei capitoli successivi verranno fornite le informazioni necessarie per la realizzazione dei semplici widget o operatori.

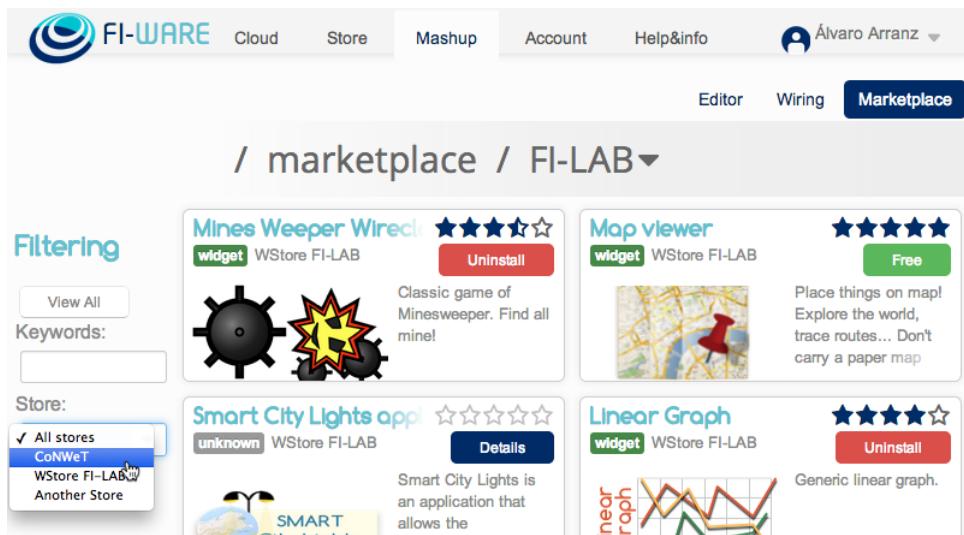


Figura 2.15: Il Marketplace del Fi-lab

2.10 CKAN

CKAN [19] è un tool per creare siti web basati su open data. Possiamo pensare a CKAN come ad un software di pubblicazione di contenuti tipo WordPress ma per la pubblicazione dei dati piuttosto che di pagine web. Il software permette dunque di amministrare e pubblicare collezioni di dati. CKAN può essere usato da chiunque voglia mettere in condivisione i propri insiemi di dati: istituti di ricerca, pubblica amministrazione, organizzazioni o aziende. Una volta che il dato è stato pubblicato, gli utenti possono usare l'interfaccia del software per cercare i dati di cui hanno bisogno. I dati sono presentati all'utente sia attraverso semplici elenchi in tabelle o anche attraverso mappe o veri e propri grafici.

2.10.1 Insiemi di dati e risorse

I dati su CKAN sono pubblicati in unità chiamate datasets. Un dataset è pacchetto di dati: per esempio le misurazioni provenienti da diverse stazioni meteo rappresentano un dataset. Il risultato di una ricerca sarà sempre un certo numero di dataset, quando l'utente cerca i suoi dati ottiene sempre un insieme di datasets come risultato. Un dataset contiene:

- Informazioni (metadati) circa i dati pubblicati; per esempio il nome dell'organizzazione che ha pubblicato il dato, la data di pubblicazione, il formato dei dati, la licenza etc.
- Un numero di risorse che contengono i dati. In CKAN non importa in quale formato è il dato. Una risorsa può essere un file CSV, un file PDF, linked data in formato RDF etc. Le risorse possono essere salvate internamente o esternamente facendo riferimento ad essa attraverso un semplice link. Un dataset inoltre può contenere qualsiasi numero di risorse. Per esempio, differenti risorse possono contenere i dati per differenti anni, o possono contenere lo stesso dato in differenti formati.

2.10.2 Utenti, organizzazioni e credenziali

Gli utenti, per utilizzare il software, devono solamente registrare il proprio account ed effettuare il log in. Di solito le credenziali non sono necessarie per cercare i dati, ma sono richieste per la pubblicazione dei contenuti: creazione e modifica dei datasets. Ciascun dataset appartiene ad un'organizzazione. CKAN non prevede un limite alle organizzazioni. Ciascuna organizzazione presente su CKAN può avere le proprie autorizzazioni, permettendo così di gestire il processo di pubblicazione da parte degli utenti. Infatti l'amministratore di un'organizzazione può aggiungere singoli utenti, con differenti ruoli in funzione del livello di autorizzazione. Chiaramente un utente di un'organizzazione può creare un dataset che apparterrà esclusivamente alla propria organizzazione. Di default, un dataset appena creato è privato e quindi visibile esclusivamente agli utenti della stessa organizzazione. Una volta pronti per la pubblicazione, i dati possono essere resi pubblici

dall’utente che ha il livello di autorizzazione necessario: di solito l’amministratore. I dataset, solitamente, sono creati all’interno delle organizzazioni. E’ possibile impostare CKAN per permettere ai dataset di non appartenere ad alcuna organizzazione. Questi dati possono però essere modificati da qualunque utente connesso al sistema. Questa possibilità permette di creare una sorta di sistema di pubblicazione e editing di dati condiviso detto wiki-like data hub.

2.10.3 Come si usa

Attraverso il FI-LAB, è possibile interagire con CKAN semplicemente accedendo alla sezione Data.



Figura 2.16: CKAN sul Fi-lab

Attraverso questa sezione è possibile effettuare ricerche di dati in base a specifiche parole chiave ed eventualmente scaricare i dati di interesse. Per poter aggiungere o modificare un dataset si deve necessariamente appartenere ad un’organizzazione; durante la fase di registrazione del nuovo insieme di dati CKAN chiede una serie di informazioni relative al dataset stesso, che sono:

- Un titolo: che deve essere unico, breve e quanto più specifico possibile.
- Una descrizione: che racchiuderà tutta una serie di informazioni aggiuntive sul dataset che possono tornare utili all’utente.
- Dei tags: che aiuteranno le persone nella ricerca dei dati.
- La licenza: per informare l’utente su quale tipo di licenza è stata usata per rilasciare tali dati.

- La visibilità: che può essere pubblica, rendendo i dati accessibili a tutti gli utenti, o privata, consentendo quindi l'accesso solo ai membri dell'organizzazione.
- La ricercabilità: che può consentire l'accesso diretto ai dati o l'accesso tramite URL specifica.

The screenshot shows the CKAN 'Create dataset' interface. The top navigation bar has three steps: 1. Create dataset (highlighted in green), 2. Add data, 3. Additional info. The main form fields are:

- Title:** eg. A descriptive title. Below it is a URL field: * URL: data.lab.fi-ware.org/dataset/<dataset> with an 'Edit' button.
- Description:** eg. Some useful notes about the data. A note below says: You can use Markdown formatting here.
- Tags:** eg. economy, mental health, government
- License:** License Not Specified. A note to the right says: License definitions and additional information can be found at [opendefinition.org](#).
- Visibility:** Private. A note to the right says: Private datasets can only be accessed by certain users, while public datasets can be accessed by anyone.
- Searchable:** True. A note to the right says: Searchable datasets can be searched by anyone, while non-searchable datasets can only be accessed by entering directly its URL.
- Allowed Users:** Allowed Users

Figura 2.17: Registrazione di un dataset in CKAN

Una volta concluso l'inserimento di queste informazioni è possibile procedere aggiungendo la risorsa o le risorse che conterranno i dati del dataset. Per ogni risorsa, sia essa un file o un link, viene richiesto l'inserimento di:

- Un nome
- Una descrizione
- Il formato (CSV, XML, JSON, PDF etc.)

Infine, la creazione del dataset viene completata tramite l'inserimento di ulteriori informazioni riguardanti:

- La sorgente dei dati (può essere ad esempio un link).
- La versione dei dati.

The screenshot shows the CKAN 'Add data' interface. At the top, a green progress bar indicates the current step: '1 Create dataset' (highlighted), '2 Add data', and '3 Additional info'. Below the progress bar, there are two upload options: 'File:' with a 'Upload' button and 'Link'. A 'Name:' field contains the placeholder 'eg. January 2011 Gold Prices'. A 'Description:' field contains the text 'Some useful notes about the data', with a note below stating 'You can use Markdown formatting here'. A 'Format:' dropdown menu is shown. At the bottom, there are navigation buttons: 'Previous', 'Save & add another' (disabled), and a blue 'Next: Additional Info' button.

Figura 2.18: Aggiunta di una risorsa in CKAN

- L'autore, che corrisponde al nome della persona o dell'organizzazione responsabile della produzione di tali dati.
- L'email dell'autore.
- Il maintainer, che corrisponde ad una seconda persona responsabile dei dati
- L'email del maintainer.
- Una serie di campi personalizzati che il creatore del dataset può aggiungere in base ai suoi scopi.

La ricerca di un dataset può essere fatta dalla sezione Data, digitando nel form di ricerca le parole chiave desiderate. CKAN mostrerà i risultati, e l'utente potrà scegliere di restringere la ricerca solo ad alcuni dataset che per esempio posseggano determinati tags o usare dei filtri di ricerca specifici. Se si vuole effettuare la ricerca di un dataset tra quelli di una specifica organizzazione, è possibile accedere alla pagina dell'organizzazione stessa su CKAN e digitare qui le parole chiave per la ricerca.

The screenshot shows a step-by-step process for creating a dataset. The current step is 'Additional info'. The form contains the following fields:

- Source:** http://example.com/dataset.json
- Version:** 1.0
- Author:** Joe Bloggs
- Author Email:** joe@example.com
- Maintainer:** Joe Bloggs
- Maintainer Email:** joe@example.com
- Custom Field:** Key: [] Value: []
- Custom Field:** Key: [] Value: []
- Custom Field:** Key: [] Value: []

Figura 2.19: Completamento della registrazione di un dataset in CKAN

Se si è interessati all'esplorazione di uno dei dataset ottenuti tra i risultati, basta selezionare il dataset d'interesse e CKAN ne visualizzerà la pagina. Qui si potranno consultare informazioni come il nome, la descrizione del dataset e brevi descrizioni di ognuna delle risorse del dataset. È possibile da qui scaricare i dati o semplicemente visualizzarli (solo per determinati formati). Nella pagina del dataset sono presenti generalmente altri tre pannelli che sono:

- Groups: dove vengono elencati i gruppi associati al dataset in questione.
- Activity Stream: dove è possibile consultare lo storico dei recenti cambiamenti avvenuti sul dataset.
- Related: dove vengono elencati i collegamenti alle pagine web relative a questo dataset.

The screenshot shows the CKAN dataset page for 'Barcelona: List of events dialy'. At the top, there's a navigation bar with links for 'Dataset', 'Groups', 'Activity Stream', and 'Related'. Below the navigation, the title 'Barcelona: List of events dialy' is displayed. A 'Follow' button is present. On the left, there are sections for 'Organization' (Barcelona) and 'Social'. The main content area includes a 'Data and Resources' section with a 'Dialy agenda' resource (updated every 30 minutes), an 'Explore' button, and tabs for 'Culture and Leisure' and 'Daily agenda'. An 'Additional Info' table provides details about the source and maintainer.

Field	Value
Source	Ajuntament de Barcelona
Maintainer	Jurgen Harms

Figura 2.20: Pagina di un dataset in CKAN

Un utente interessato ad un dataset può richiedere al sistema l'invio di una notifica in seguito ad una modifica sui dati cliccando semplicemente sul tasto Follow.

Capitolo 3

Implementazione

3.1 Descrizione dettagliata del progetto

In questo paragrafo forniremo una descrizione dettagliata dell’architettura del sistema da noi implementato, descrivendo quali componenti software sono state utilizzate e come queste interagiscono tra di loro.

Nel primo capitolo avevamo anticipato che il sistema è stato diviso in tre livelli, spiegando anche le motivazioni di questa scelta ed il compito di ciascuna delle parti. Rimandiamo quindi al paragrafo 1.2 a pagina 5 per un’ introduzione al progetto.

A partire dal livello più basso abbiamo:

- livello di **sensoristica**
- livello di **backend**
- livello di **interfaccia utente**

Per realizzare la componente hardware del livello di sensoristica, sono state utilizzate due schede molto diffuse sul mercato con buone capacità d’elaborazione e dai costi contenuti, la **Raspberry pi B+** e la **Linino One**. Per la misurazione del particolato abbiamo collegato alla scheda un sensore: lo **Shinyei PPD42NS**. Inoltre è stato utilizzato un dispositivo **OBD** per prelevare direttamente dall’autovettura i valori di temperatura e pressione

ambientale. La scheda e i sensori scelti ci hanno permesso di realizzare un sistema di monitoraggio a basso costo.

L'immagine di sotto mostra una delle schede utilizzate (Raspberry) ed il sensore di particolato connessi attraverso una breadboard.

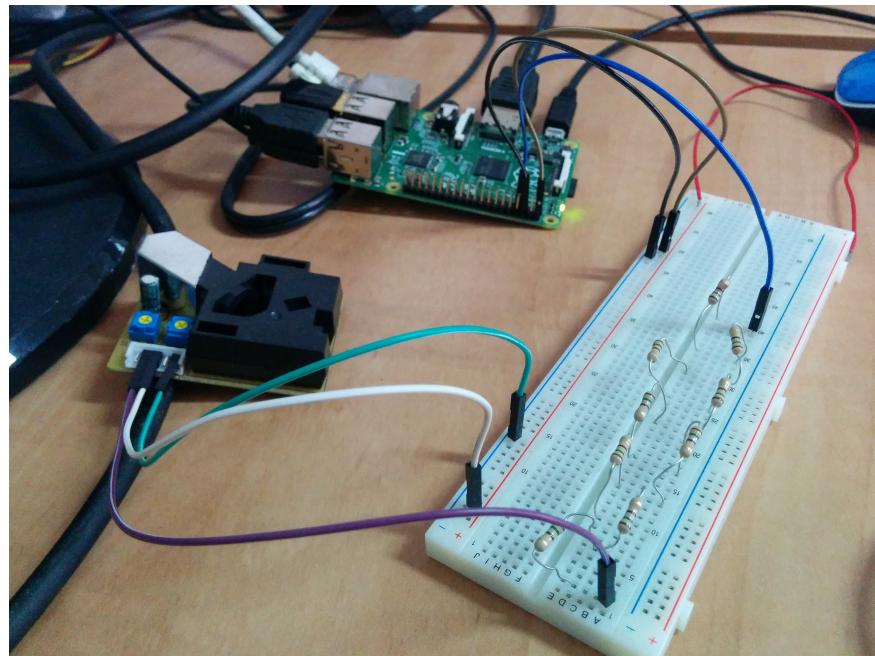


Figura 3.1: Il sensore collegato alla scheda

Il software che gestisce la componente di sensoristica è composto da tre script che girano direttamente sul dispositivo embedded (Linino o Raspberry):

- **Obd.py:** legge i valori di temperatura e pressione dal dispositivo obd e li scrive su un file.
- **Shinyei.py:** legge i valori di PM10 dal sensore Shinyei e li scrive su un file.
- **Send_measures.py:** rileva un aggiornamento dei file e invia i nuovi dati ad Orion.

Il codice è stato scritto in Python in quanto l'interprete per questo linguaggio di programmazione era disponibile su entrambe le schede; il linguaggio

inoltre è stato arricchito dalla comunità dei suoi utenti con un elevato numero di librerie. Alcune di queste ci hanno facilitato il compito soprattutto per la lettura dei dati dai pin GPIO della raspberry e nell'interazione con il dispositivo OBD.

Il secondo livello, come anticipato, ha il compito di permettere la comunicazione tra l'applicazione produttrice dei dati (nel nostro caso il livello di sensoristica) e l'applicazione consumatrice (l'interfaccia grafica che si trova al terzo livello). I dati di temperatura, pressione e del particolato devono essere smistati dal primo al terzo livello e memorizzati per una futura consultazione o post elaborazione. Per svolgere questi compiti abbiamo utilizzato esclusivamente moduli software messi a disposizione dalla tecnologia Fi-ware, nello specifico:

- **Orion:** è il modulo software centrale del livello di backend, ha il compito di ricevere le informazioni/misurazioni, di smistare i valori misurati alle applicazioni che ne hanno fatto richiesta attraverso sottoscrizioni, di inviare i dati a Cosmos/CKAN per una memorizzazione permanente. Nel paragrafo 2.3 a pagina 17 è possibile trovare una trattazione dettagliata circa le caratteristiche di Orion.
- **Cosmos:** mette a disposizione un file-system distribuito in cui salvare i dati, il componente fa anche uso della tecnologia Hadoop MapReduce per una elaborazione efficiente dei dati su un cluster di macchine. Il lettore può trovare informazioni su questo modulo software al paragrafo 2.8 a pagina 51.
- **CKAN:** permette la pubblicazione e la consultazione di informazioni in formato open data. I dati raccolti dal sistema verranno quindi pubblicati sullo store del Fi-lab attraverso le api CKAN. I dati quindi potranno essere venduti o fruiti gratuitamente attraverso lo store. Consultare il paragrafo 2.10 a pagina 59 per ulteriori informazioni.

L'ultima componente del sistema è l'interfaccia con l'utente, attraverso di essa è possibile selezionare il servizio d'interesse e visualizzare sulla mappa le entità che prendono parte al servizio selezionato. Nel caso specifico,

cioè l'applicazione per la misurazione del particolato, l'utente visualizzerà sulla mappa i taxi che dispongono dei sensori e le zone della città in cui sono disponibili delle misurazioni. Per la realizzazione dell'interfaccia grafica abbiamo utilizzato **Wirecloud** (vedi paragrafo 2.9 a pagina 57) e implementato due nuovi widget per i nostri scopi:

- **ServiceWidget**: mostra all'utente i servizi attivi e permette di selezionarne uno.
- **QueryWidget**: individua tutte le entità di un servizio e le inoltra alla mappa per la visualizzazione.

I widget e gli operatori già esistenti che utilizziamo sono i seguenti:

- **MapView**: mostra le entità di un servizio su una mappa.
- **LinearGraph**: mostra il grafico di una misurazione.
- **NgsiEntityToPoi**: usato per trasformare i dati in uscita dal QueryWidget in un formato adatto per il MapView.

Passiamo adesso alla descrizione di come le varie parti interagiscono tra loro.

Per quanto riguarda la **parte grafica**:

- Il **ServiceWidget** contatta Orion per conoscere i servizi disponibili, effettua una sottoscrizione per essere aggiornato in modo asincrono sulla creazione di nuovi servizi o circa l'aggiornamento di servizi già esistenti
- Il **ServiceWidget** mostra all'utente i servizi appena scoperti, l'utente seleziona il servizio a cui è interessato cliccando con il mouse sul nome del servizio.
- Il **ServiceWidget** invia al QueryWidget il servizio che l'utente ha selezionato
- Il **QueryWidget** contatta Orion per selezionare tutte le entità che appartengono al servizio richiesto, nel nostro caso le entità mostrate saranno i taxi e le misurazioni effettuate.

- Le entità trovate sono mostrate sulla mappa dal **MapViewer**.

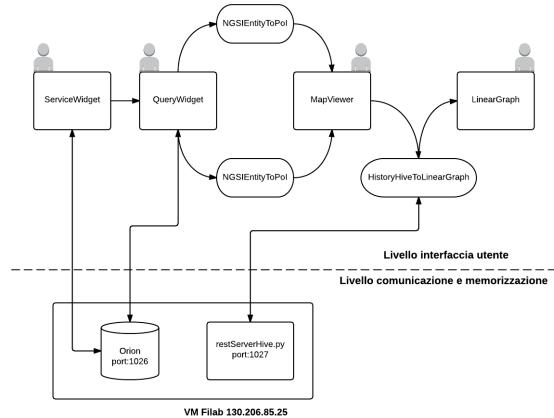


Figura 3.2: Livello di interfaccia utente

Per la parte di sensoristica:

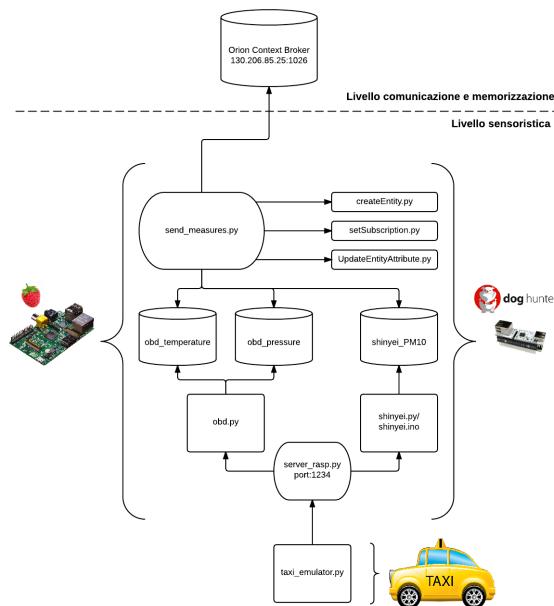


Figura 3.3: Livello di sensoristica

- Il tassista attiva la misurazione attraverso il tablet presente sulla propria autovettura. Per fare ciò contatta un server scritto in python (ser-

ver_rasp.py) presente direttamente sulla scheda. Il server, ricevuto il comando, eseguirà gli script necessari per effettuare la misurazione.

- Ogni trenta secondi gli script **Obd.py**, **Shinyei.py**, **Send_measures.py**, **UpdateEntityAttribute.py** si occupano di effettuare le misurazioni di temperatura pressione e PM10 e inviare i dati aggiornati al server Orion.

Per la **logica di comunicazione e memorizzazione**:

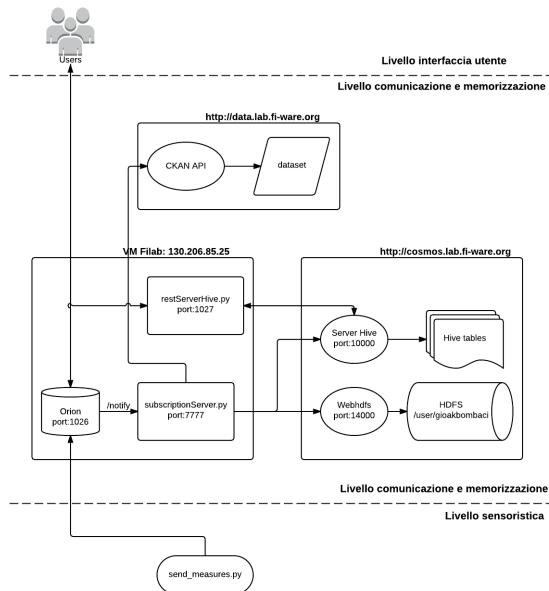


Figura 3.4: Livello di backend

- Il **subscriptionServer** resta in ascolto per aggiornamenti sui dati, quando riceva una notifica di update (nuove misurazioni) dal server Orion, invia il nuovo dato a Cosmos. Contemporaneamente il subscriptionServer si occuperà di salvare i dati su CKAN usando le api appropriate.
- **Cosmos**, ottenuti i dati inviatigli dal subscriptionServer, li memorizza su un filesystem distribuito.
- Abbiamo detto che attraverso il **LinearGraph** è possibile visualizzare in un grafico i dati relativi ad una misurazione, il LinearGraph per fare

questo contatta il **RestServerHive**. Il compito del RestServerHive è quello di fare da proxy tra il LinearGraph e Cosmos: il LinearGraph richiede i dati al server, quest'ultimo effettua una query Hive su Cosmos e restituisce i dati al widget in un formato opportuno. I passaggi appena descritti risultano necessari in quanto le api NGSI di Wirecloud non mettono a disposizione delle procedure per contattare direttamente Cosmos.

- Abbiamo dunque due modalità di accesso ai dati; la prima, descritta sopra, attraverso i widget di wirecloud, la seconda tramite i dataset pubblicati su CKAN.

3.1.1 Orion

In questo paragrafo verranno mostrati brevemente i passaggi per installare [12] e lanciare correttamente il broker. Gli esempi mostrati di sotto sono stati realizzati su una distribuzione CentOS, questo vuol dire che per l’installazione del software abbiamo utilizzato il sistema di gestione dei pacchetti yum. Purtroppo in questo momento il broker è pacchettizzato solo in formato .rpm, se si volesse installarlo su distribuzioni differenti una soluzione potrebbe essere alien¹.

E’ importante ricordare che il broker, per memorizzare le proprie informazioni, utilizza un database NoSQL mongodb; si assume quindi in questa sede che sulla macchina in cui si sta effettuando l’installazione sia già presente il software in questione².

3.1.1.1 Installazione

L’installazione di Orion è molto semplice e può essere effettuata sia manualmente attraverso l’apposito pacchetto rpm, sia in modo automatico attraverso il repository di Fi-ware.

¹<http://joeyh.name/code/alien/>

²Per chi non sapesse come installare mongodb su un sistema CentOS è possibile consultare questa guida sul sito del progetto: <http://docs.mongodb.org/manual/tutorial/install-mongodb-on-red-hat-centos-or-fedora-linux/>

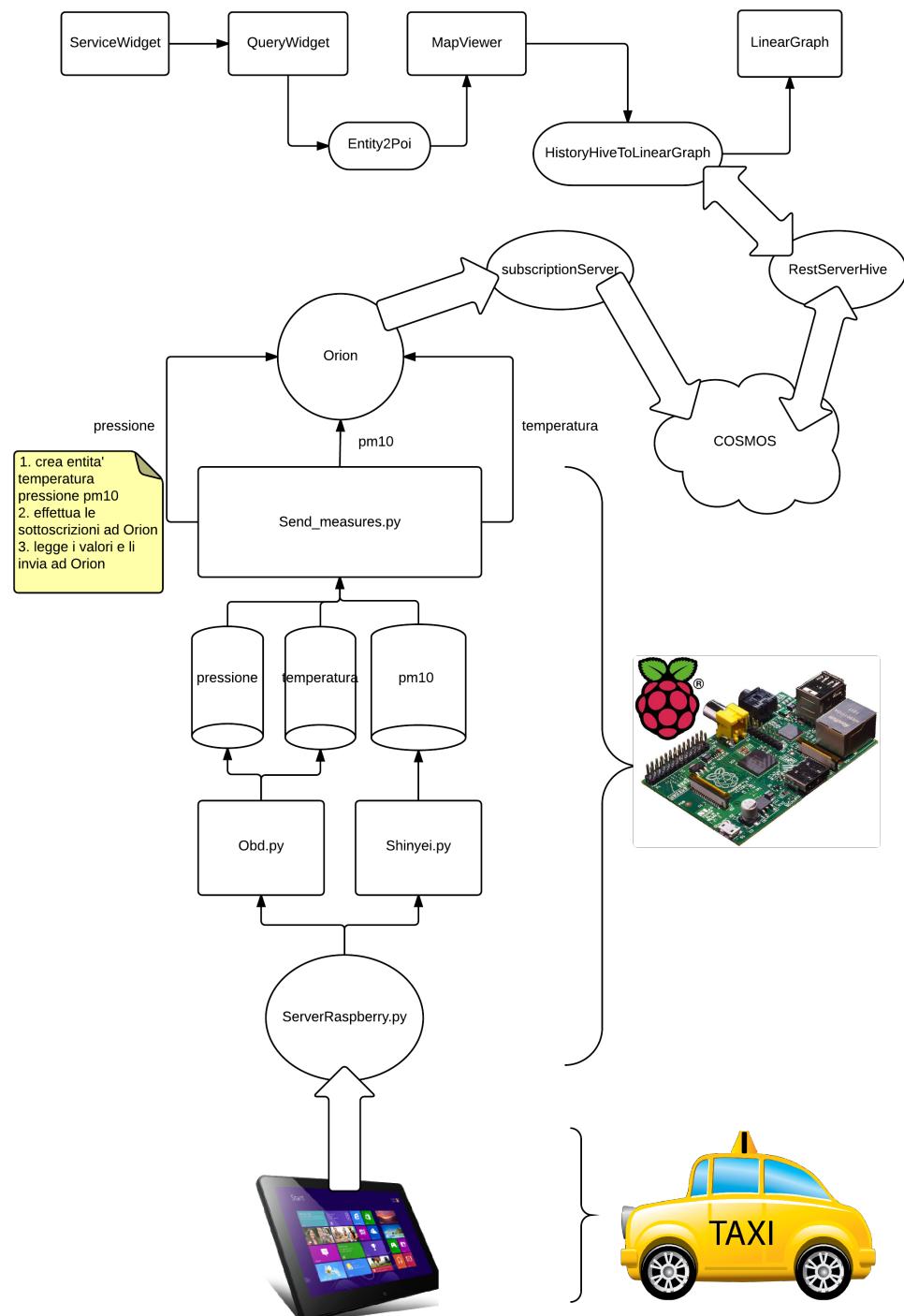


Figura 3.5: Architettura completa del sistema

Per effettuare un'installazione manuale bisogna:

1. Scaricare il pacchetto rpm contextBroker-X.X.X-x.x86_64.rpm al seguente indirizzo: https://forge.fiware.org/frs/?group_id=7
2. Installare il pacchetto con il comando rpm -i contextBroker-X.X.X-x.x86_64.rpm.

Per installare Orion attraverso il repository è sufficiente aggiungere nella cartella **/etc/yum.repos.d** il file **testbed-fiware.repo** con le seguenti righe:

```
[ testbed-fi-ware ]
name=Fiware Repository
baseurl=http://130.206.80.64/repo/rpm/x86_64/
gpgcheck=0
enabled=1

[ testbed-fi-ware-new ]
name= Fiware Repo two
baseurl=http://repositories.testbed.fi-ware.org/repo/rpm/x86_64/
gpgcheck=0
enabled=1
```

A questo punto per far partire l'installazione digitare il seguente comando:

```
yum install contextBroker
```

3.1.1.2 Esecuzione e check dell'installazione

Una volta completata l'installazione sarà possibile lanciare il broker con il seguente comando:

```
/etc/init.d/contextBroker start
```

Si può verificare il corretto funzionamento del software richiedendo al server orion in esecuzione il proprio numero di versione. Per fare questo utilizziamo curl:

```
curl localhost:1026/version
```

La risposta in caso di corretto funzionamento dovrebbe essere la seguente:

```
<orion>
<version>0.8.0-next</version>
<uptime>0 d, 0 h, 0 m, 11 s</uptime>
<git_hash>3fdb55b96913b3e4d9f9a344e990164650f69b91</git_hash>
<compile_time>Wed Oct 30 15:31:29 CET 2013</compile_time>
<compiled_by>fermin</compiled_by>
<compiled_in>centollo</compiled_in>
</orion>
```

3.1.2 Rush

Rush è un http relayer che permette di:

- Effettuare richieste http asincrone
- Tenere traccia dello stato della richieste inoltrate
- Effettuare ritrasmissioni automatiche delle richieste http
- Effettuare chiamate di funzioni al termine di una richiesta http

Rush sostanzialmente effettua delle richieste http per conto dell'applicazione che lo richiede. E' stato necessario utilizzare Rush in quanto wirecloud utilizza, per la comunicazione con Orion, una connessione https. Orion purtroppo non è in grado di inviare le notifiche e le sottoscrizioni attraverso https.



Figura 3.6: Rush

Per svolgere i propri compiti Rush utilizza un'architettura relativamente semplice. Anche in questo caso abbiamo dei produttori e dei consumatori. I consumatori comunicano con i produttori attraverso un redis server. Ciascun consumatore utilizza una serie di worker che implementano le funzionalità rush sopra descritte.

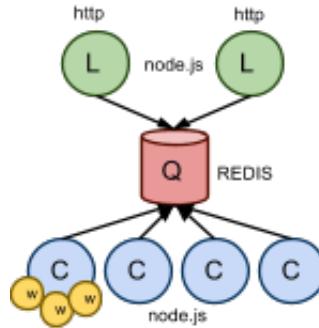


Figura 3.7: Architettura Rush

Nella figura di sopra vediamo in verde i produttori (listeners) che generano le richiesta http, la richiesta è inoltrata ai worker (in giallo) attraverso redis (in rosso). I consumatori (in blu) servono la specifica richiesta attraverso uno o più worker. Come detto, l'installazione di Rush è stata necessaria per permettere la comunicazione tra Orion e i widget installati su wirecloud. La comunicazione http tra i widget su wirecloud e orion è criptata attraverso SSL, questo vuol dire che le notifiche http che Orion invia al widget a seguito di una sottoscrizione devono essere di tipo https. Purtroppo Orion nativamente non è in grado di inoltrare notifiche https, per fare ciò si appoggia appunto a rush. L'installazione di rush è stata quindi indispensabile. Di seguito mostriamo brevemente i passi da seguire per installare sulla propria macchina il software.

Per prima cosa è necessario installare il redis-server che permette la comunicazione tra i nodi produttori e i nodi consumatori. E' importante sottolineare che Orion funziona esclusivamente con versioni di redis-server superiori alla 2.6³. Una volta installato il redis-server si può passare all'installazione di rush. Di seguito mostriamo i passi da seguire:

1. Scaricare una versione stabile dal sito del progetto⁴
2. Installare le dipendenze con il comando: `npm install --production`

³Redis è il più diffuso db di tipo key value, è possibile scaricare redis al seguente indirizzo <http://download.redis.io/releases/>

⁴<https://github.com/telefonicaid/Rush>

3. Settare nel file *configBase.js* la variabile *gevlsnrMongo* per informare rush sulla macchina dove è installato mongodb
4. Lanciare il redis-server
5. Lanciare il listener nella cartella bin in questo modo : ./listener
6. Lanciare il consumer nella cartella bin in questo modo: ./consumer
7. Lanciare Orion con l'opzione –rush in questo modo: /usr/bin/contextBroker –dbhost localhost –db orion –rush localhost:5001

Al termine di questi passi dovreste avere in ascolto i seguenti software sulle seguenti porte:

1. Orion 1026
2. Redis-server 6379
3. Rush 5001
4. MongoDB 27017/28017

Active Internet connections (only servers)						
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	0.0.0.0:5001	0.0.0.0:*	LISTEN	31399/node
tcp	0	0	0.0.0.0:27017	0.0.0.0:*	LISTEN	1704/mongod
tcp	0	0	0.0.0.0:6379	0.0.0.0:*	LISTEN	17340./redis-serv
tcp	0	0	0.0.0.0:5901	0.0.0.0:*	LISTEN	1755/Xvnc
tcp	0	0	0.0.0.0:111	0.0.0.0:*	LISTEN	1253/rpcbind
tcp	0	0	0.0.0.0:6001	0.0.0.0:*	LISTEN	1755/Xvnc
tcp	0	0	0.0.0.0:28017	0.0.0.0:*	LISTEN	1704/mongod
tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN	26010/sshd
tcp	0	0	127.0.0.1:631	0.0.0.0:*	LISTEN	1453/cupsd
tcp	0	0	127.0.0.1:25	0.0.0.0:*	LISTEN	1658/master
tcp	0	0	0.0.0.0:53371	0.0.0.0:*	LISTEN	1271/rpc.statd
tcp	0	0	0.0.0.0:1026	0.0.0.0:*	LISTEN	17605/contextBroker

Figura 3.8: Connessioni in ascolto

3.1.3 subscriptionServer.py

Il subscriptionServer svolge il ruolo di proxy tra Orion e i due sistemi di memorizzazione usati dal nostro sistema: Cosmos e CKAN. Come abbiamo ampiamente detto, Orion non è in grado di memorizzare i dati a lungo termine; i valori degli attributi vengono sovrascritti ad ogni nuovo aggiornamento.

E' stato quindi necessario utilizzare altri meccanismi per la memorizzazione permanente dei dati. Cosmos, usando la tecnologia Hadoop, permette al sistema di sfruttare un filesystem distribuito in grado di memorizzare grosse quantità di dati. Inoltre, attraverso MapReduce, si offre la possibilità di effettuare elaborazioni efficienti sui dati distribuiti che altri sistemi non sono in grado di svolgere. CKAN invece è pensato per la pubblicazione dei dati in formato open data, dobbiamo pensare a questo sistema come un luogo in cui gli utenti o gli sviluppatori possono cercare ed accedere ai dati di cui hanno bisogno. Entrambi i sistemi dunque memorizzano i dati, Cosmos ha la funzione di memorizzazione e post elaborazione mentre CKAN è un luogo di condivisione e scambio dei dati.

Il subscriptionServer riceve gli aggiornamenti sulle misurazioni direttamente da Orion e scrive il dato sia sul filesystem distribuito di Cosmos in formato .txt sia sul dataset CKAN in formato .csv. Per essere continuamente informato sugli aggiornamenti dei dati, il subscriptionServer sfrutta un sistema di sottoscrizioni. Il *server_rasp.py*, che gira sulla scheda a cui è connesso il sensore, si occupa, prima di iniziare una misurazione, di effettuare uno sottoscrizione ad Orion sulla nuova entità observations per conto del subscriptionServer. Ricordiamo che un'entità observations rappresenta la singola osservazione. In questo modo Orion invierà automaticamente al subscriptionServer tutti gli aggiornamenti della misurazione, campione dopo campione.

Vediamo adesso di spiegare a grandi linee il codice del server. La prima cosa da fare è estrarre dal payload del messaggio di sottoscrizione le informazioni sul dato e cioè:

- **sub_id**: è l'id della sottoscrizione, tutti i messaggi relativi ad una sottoscrizione avranno un identificativo univoco che li rappresenta.
- **entity_type**: è il tipo dell'entità a cui si riferisce il dato in arrivo.
- **entity_id**: è l'id dell'entità.
- **att_type**: è il tipo dell'attributo.
- **att_name**: è il nome dell'attributo.

- **att_value:** è il campo più importante in quanto contiene il nuovo valore del dato.

Di seguito il codice che esegue queste operazioni.

```

1 payload=request.data
2 payload_obj=json.loads(payload)
3 sub_id=payload_obj[ 'subscriptionId' ]
4 entity_type=payload_obj[ 'contextResponses' ][0][
    'contextElement'][ 'type' ]
5 entity_id=payload_obj[ 'contextResponses' ][0][
    'contextElement'][ 'id' ]
6 is_pattern=payload_obj[ 'contextResponses' ][0][
    'contextElement'][ 'isPattern' ]
7 att_type=payload_obj[ 'contextResponses' ][0][
    'contextElement'][ 'attributes' ][0][ 'type' ]
8 att_name=payload_obj[ 'contextResponses' ][0][
    'contextElement'][ 'attributes' ][0][ 'name' ]
9 att_value=payload_obj[ 'contextResponses' ][0][
    'contextElement'][ 'attributes' ][0][ 'value' ]

```

Successivamente vengono creati il nome della cartella che risiederà sul filesystem distribuito, i nomi dei file txt e csv e il nome della tabella hive che il restServerHive consulterà per reperire i dati per conto dei widget di Wirecloud.

```

1 dir_on_cosmos = entity_id + '_' + entity_type
2 name_file = dir_on_cosmos + ".txt"
3 name_file_ckan = dir_on_cosmos + ".csv"
4 name_table_hive = 'gioakbombaci_' + dir_on_cosmos

```

A questo punto se è la prima volta che riceviamo la sottoscrizione dobbiamo:

- inserire l'id della sottoscrizione nel dizionario (variabile *list_sub_id*) che tiene traccia di tutte le sottoscrizioni servite dal server
- creare l'intestazione per i file csv

- creare la cartella sul filesystem distribuito

- inizializzare la tabella hive caricando i dati dal file txt appena creato.

Per fare questo si utilizzano le librerie Thrift⁵ per Hive. Attraverso queste librerie siamo in grado di aprire una connessione con HiveServer2 all’indirizzo 130.206.80.46 sulla porta 10000 del cluster Infinity.

Per informazioni su HiveServer2 consultare il paragrafo successivo.

Indipendentemente se la sottoscrizione è la prima o meno, sarà necessario creare i file txt e csv con il dato aggiornato (il nuovo campione della misurazione) e inviarlo su Cosmos. Come è possibile notare nel codice di sotto, alle righe 10 e 11, l’invio del file su Cosmos avviene in due passi, nel primo si effettua la richiesta di I/O all’ HeadNode e successivamente si esegue l’operazione di scrittura sul DataNode che conterrà effettivamente il dato. Per le operazioni di scrittura abbiamo utilizzato le api WEBHDFS di Cosmos, utilizzando curl per richiamare le procedure.

```

1 if (check == False):
2     #e' la prima sottoscrizione
3     list_sub_id[sub_id] = timestamp
4     #creo l'intestazione per il file csv
5     create_header = "echo timestamp," + att_type + " >>
6                     " + name_file_ckan
6     os.system(create_header)
7
8     create_file = "echo " + timestamp + "," + att_value +
9                     " >> " + name_file
9     create_file_ckan = "echo " + timestamp + "," +
10                    att_value + " >> " + name_file_ckan
10    send_file_step1 = 'curl -i -X PUT "http://cosmos.lab.
11                      fi-ware.org:14000/webhdfs/v1/user/gioakbombaci/
12                      observations/" + dir_on_cosmos + '/' + name_file +
13                      '?op=CREATE&user.name=gioakbombaci"'
```

⁵Thrift è un framework che permette di accedere alla CLI di Hive attraverso chiamate remote. Thrift è disponibile per diversi linguaggi di programmazione tra cui Java, C++, Python e Ruby.

```
11 send_file_step2 = 'curl -i -X PUT -T ' + name_file +
   '--header"content-type:application/octet-stream"'
   + "http://cosmos.lab.fi-ware.org:14000/webhdfs/v1/
   user/gioakbombaci/observations/' + dir_on_cosmos +
   '/' + name_file + '?op=CREATE&user.name=
   gioakbombaci&data=true',
12
13 os.system(create_file)
14 os.system(create_file_ckan)
15 if (check == False):
16     #prima sottoscrizione
17     # se e' la prima sottoscrizione
18     # -1- creo la dir
19     # -2- creo la tabella
20     create_dir = 'curl -i -X PUT "http://cosmos.lab.fi-
       ware.org:14000/webhdfs/v1/user/gioakbombaci/
       observations/' + dir_on_cosmos + '?op=MKDIRS&user.
       name=gioakbombaci"',
21     #inizio codice hive
22     os.system(create_dir)
23     os.system(send_file_step1)
24     os.system(send_file_step2)
25     #creo la tabella
26 try:
27     transport = TSocket.TSocket('130.206.80.46', 10000)
28     transport = TTransport.TBufferedTransport(transport)
29     protocol = TBinaryProtocol.TBinaryProtocol(transport
           )
30     client = ThriftHive.Client(protocol)
31     transport.open()
32     query = 'create external table ' + name_table_hive +
           " (time string, observed_property string) row
           format delimited fields terminated by ','
```

```

        location '/user/gioakbombaci/observations/" +
        dir_on_cosmos + "'"
33    client.execute(query)
34    transport.close()
35    except Thrift.TException,
36 tx:
37    print '%s' % (tx.message)

```

Se invece non si tratta della prima sottoscrizione, è necessario cancellare il file txt e caricare il nuove file con il dato aggiornato.

```

1 else:
2 #se non e' la prima sottoscrizione
3 #-1- cancello il file
4 #-2- reinvio il file
5 delete_file = 'curl -i -X DELETE "http://cosmos.lab.fi
-ware.org:14000/webhdfs/v1/user/gioakbombaci/
observations/' + dir_on_cosmos + '/' + name_file +
'?op=DELETE&user.name=gioakbombaci"',
6 os.system(delete_file)
7 os.system(send_file_step1)
8 os.system(send_file_step2)

```

L'ultima operazione da fare, al termine dell'intera misurazione, è la scrittura del file .csv sul dataset CKAN. Il file viene scritto alla fine, e non campione dopo campione, in quanto CKAN non è in grado di servire le richieste di scrittura alla frequenza con la quale il server riceve i singoli dati. Si è preferito dunque, scrivere l'intero file alla conclusione della misurazione con una sola operazione di scrittura, piuttosto che aggiornare il file su CKAN ad ogni aggiornamento con più operazioni di scrittura. E' il server python *server_rasp.py* presente sulla scheda che invia il comando di caricamento del file .csv sul dataset al termine della misurazione.

Per svolgere queste operazioni si usano le API messe a disposizione da CKAN [18].

```
1 @app.route('/send_ckan', methods=['POST'])
```

```
2 def send_ckan():
3     id_ckan_resource = request.args.get('id_obs', '')
4     file_temperature = "temperature_" + id_ckan_resource
5         + "_observations.csv"
6     file_pressure = "pressure_" + id_ckan_resource + "
7         _observations.csv"
8     file_pm10 = "pm10_" + id_ckan_resource + "
9         _observations.csv"
10    create_temperature = "curl -H'Authorization: 30
f4d771-ef48-4050-a35b-de965247f8aa' 'https://data
.lab.fiware.org/api/action/resource_create' --
form upload=@" + file_temperature + " --form
package_id=provadataset --form name=\"" +
file_temperature + "\" --form format=csv"
11    create_pressure = "curl -H'Authorization: 30f4d771-
ef48-4050-a35b-de965247f8aa' 'https://data.lab.
fiware.org/api/action/resource_create' --form
upload=@" + file_pressure + " --form package_id=
provadataset --form name=\"" + file_pressure + "\" --
form format=csv"
12    create_pm10 = "curl -H'Authorization: 30f4d771-ef48
-4050-a35b-de965247f8aa' 'https://data.lab.fiware
.org/api/action/resource_create' --form upload=@"
+ file_pm10 + " --form package_id=provadataset
--form name=\"" + file_pm10 + "\" --form format=csv"
13    print("ckan create resource " + file_temperature)
14    os.system(create_temperature)
15    time.sleep(30)
16    print("ckan create resource " + file_pressure)
17    os.system(create_pressure)
18    time.sleep(30)
19    print("ckan create resource " + file_pm10)
20    os.system(create_pm10)
```

3.1.4 restServerHive.py

Come anticipato nel paragrafo 3.1 a pagina 66 questo server svolge funzioni di proxy tra il LinearGraph di Wirecloud e i dati delle misurazioni presenti su Cosmos. La libreria NGSI messa a disposizione da Wirecloud non prevede ancora delle funzionalità per accedere direttamente, tramite hive, ai dati presenti sul filesystem distribuito. Le stessa libreria NGSI permette però di effettuare delle semplici richieste http. Abbiamo quindi pensato di realizzare un server rest che, utilizzando le librerie Thrift per hive, fornisse chiamate rest (richiamabili dai widget) per reperire i dati su HDFS da Wirecloud.

Dopo aver importato le librerie Thrift necessarie per interagire con hive da python, si definisce nella variabile *urls* l’interfaccia rest verso i widget, cioè i percorsi delle chiamate richiamabili dall’esterno. In questo modo si associa il percorso `http://localhost/observations` alle classe `get_observations`.

```

1 from hive_service import ThriftHive
2 from hive_service.ttypes import HiveServerException
3 from thrift import Thrift
4 from thrift.transport import TSocket
5 from thrift.transport import TTransport
6 from thrift.protocol import TBinaryProtocol
7
8 urls = (      '/taxi/Taxi1', 'get_taxi_obs',
9           '/observations', 'get_observation'
10 )

```

La classe `get_observations` si occupa di prelevare i dati delle misurazioni su Cosmos e passarli al widget. Il primo passo consiste nell’aprire una connessione sulla porta 10000 con l’host 130.206.80.46 (`cosmos.lab.fi-ware`) che rappresenta l’indirizzo dell’ HeadNode del cluster Infinity. Ricordiamo che Infinity è il cluster di storage del Fi-lab. Quello che stiamo facendo è contattare l’HiveServer2 presente sul nodo HeadNode; l’HiveServer fa parte del progetto Hive di Hadoop e fornisce un’ interfaccia che permette a client remoti di eseguire query attraverso Hive CLI o tramite API che sfruttano driver ODBC/JDBC. La versione 2 di HiveServer fornisce meccanismi di

autenticazione e connessioni multiple concorrenti.

```

1 class get_observation:
2     def GET( self ):
3         try:
4             transport = TSocket.TSocket( '130.206.80.46' , 10000 )
5             transport = TTransport.TBufferedTransport( transport )
6             protocol = TBinaryProtocol.TBinaryProtocol( transport )
7             result = ""
8
9             client = ThriftHive.Client( protocol )
10            transport.open()

```

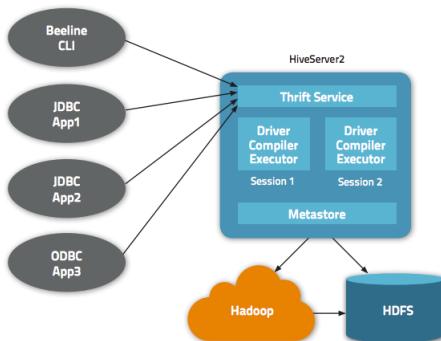


Figura 3.9: Architettura HiveServer2

A questo punto siamo connessi ad HiveServer2 e possiamo eseguire query sui dati usando la sintassi HiveQL. Prima però dobbiamo effettuare il parsing dei parametri⁶ passati dal widget nella chiamata al metodo rest `http://localhost/observations`. I parametri sono due: l'id ed il tipo dell'entità su cui effettuare la query su Cosmos. Nel nostro caso l'entità a cui siamo interessati è la singola osservazione.

```

1 parameter = web.input()
2 ent_id = parameter.id

```

⁶Per effettuare il parsing dei parametri di una chiamata rest abbiamo utilizzato la libreria web.py, la documentazione dei metodi è consultabile al seguente indirizzo <http://webpy.org/>

```
3 ent_type = parameter.type
```

Una volta connessi a HiveServer2 e prelevato l'id sui cui effettuare la query, non rimane altro che interrogare la tabella che contiene i dati della misurazione in questo modo:

```
client.execute("select * from gioakbombaci_"+ent_id+"_"+ent_type)
```

A questo punto si preleva una riga della tabella alla volta attraverso un ciclo *while*. La tabella delle osservazioni è composta da due colonne: la prima contiene il timestamp e la seconda il valore numerico del campione. Questi due valori sono memorizzati rispettivamente nelle variabili *date* e *value*.

```
1 while (1):
2     row = client.fetchOne()
3     if (row == ""):
4         break
5     r_o_w = row.split()
6     if (len(r_o_w) != 2):
7         continue
8     if (r_o_w[0]):
9         print r_o_w[0]
10    date = r_o_w[0]
11    if (r_o_w[1]):
12        print r_o_w[1]
13        value = r_o_w[1]
```

Ciascuna riga deve essere anche formattata in json e inserita in lista (variable *table*).

```
1 currentRow = Row_obs(date, value)
2 provaJ = json.dumps(currentRow.__dict__)
3 table.append(provaJ)
```

Al termine del ciclo *while*, all'interno della variabile *table* avremo memorizzato tutte le righe della tabella. Non resta altro che ritornare la lista al widget formattandola prima in formato json.

```

1 Jtable = json.dumps(table)
2 return Jtable

```

3.1.5 Widget

In questo paragrafo spiegheremo brevemente quale è la struttura da seguire per realizzare un widget [17] per wirecloud. Un widget è formato da tre componenti:

- Il **template** che descrive il widget: i suoi punti di ingresso e di uscita, i parametri di default e tutti i riferimenti alle risorse del widget.
- Il **codice** composto dalle pagine html, dai file javascript e css. Questi file contengono la descrizione del comportamento del widget.
- Le **risorse statiche**, come le immagini o la documentazione.

Verrà mostrato un template di esempio e spiegati i tag che lo compongono. In questo paragrafo non vengono analizzati gli altri due componenti di un widget in quanto sia il codice che le risorse statiche dipendono dalla specifica implementazione. Tutti i widget dunque devono contenere al loro interno un template che descrive il componente e le risorse ad esso associate. Abbiamo tre sezioni principali:

- **ResourceDescription**: descrittiva del widget stesso, in questa sezione vengono specificati il nome del widget, l'autore, il numero di versione etc.
- **Platform.Preferences**: vengono specificati i parametri del widget con i valori di default.
- **Platform.Wiring**: sono descritti i punti di ingresso e di uscita, le interfacce di cui avevamo parlato nell'introduzione a Wirecloud.

Iniziamo dalla **ResourceDescription**:

```

<?xml version="1.0" encoding="UTF-8"?>
<Template xmlns="http://wirecloud.conwet.fi.upm.es/ns/template#">

```

```

<Catalog . ResourceDescription>
<Vendor>arkimede</Vendor>
<Name>Esempio</Name>
<DisplayName>Esempio</DisplayName>
<Author>ciccio cappuccio</Author>
<Version>0.0.11</Version>
<Mail>ciccio@cappuccio</Mail>
<Description>esempio</ Description>
<ImageURI>images / catalogue . png</ImageURI>
<iPhoneImageURI>images / catalogue . png</iPhoneImageURI>
<WikiURL></WikiURL>
<Requirements>
<Feature name="NGSI" />
</Requirements>
</Catalog . ResourceDescription>

```

I tag sono autoesplicativi, vale la pena però spiegarne almeno due:

- **DisplayName:** è il nome che comparirà sullo store di Wirecloud.
- **Requirements:** specifica le librerie esterne che verranno utilizzate dal widget, in questo caso si fa riferimento alle funzioni NGSI.

Passiamo alla sezione **Platform.Preferences**:

```

<Platform . Preferences>
<Preference name="ngsi_server" type="text"
description="server che contiene la lista dei servizi"
label="NGSI Server" default="http://orion.lab.fi-ware.org:1026/">
</Preference>
<Preference name="ngsi_proxy" type="text"
description="server proxy" label="NGSI proxy"
default="https://ngsiproxy.lab.fiware.org">
</Preference>
</Platform . Preferences>

```

In questa sezione vengono specificati i parametri del widget e impostati i corrispettivi valori di default. I parametri del widget sono visibili cliccando

sull'icona setting. Nel caso specifico vengono definiti i parametri ngsi_server ed ngsi_proxy.

```
<Platform . Wiring>
<OutputEndpoint name="outputService" type="text"
description="ritorna il servizio selezionato nella tabella"
label="Selected Service" friendcode="service">
</OutputEndpoint>
</Platform . Wiring>
<Platform . Link>
<XHTML href="index . html" contenttype="text/html"/>
</Platform . Link>
<Platform . Rendering width="10" height="20"/>
</Template>
```

In ultimo, la sezione **Platform.Wiring**. Qui vengono dichiarati i punti di uscita e di ingresso del widget che serviranno nelle operazioni di wiring con gli altri widget e operatori. Inoltre viene specificata la pagina iniziale index.html e le dimensioni della finestra del widget.

3.1.6 ServiceWidget

Il ServiceWidget è stato interamente sviluppato da noi allo scopo di mostrare all'utente i servizi disponibili sul server. L'utente attraverso il widget è in grado di scoprire tutti i servizi disponibili e selezionare con un click il servizio a cui è interessato. Il ServiceWidget è connesso ad un altro widget, introdotto nel paragrafo successivo, che effettuerà le interrogazioni su tutte le entità che partecipano al servizio selezionato dall'utente. In questo modo le entità selezionate verranno visualizzate su una mappa attraverso un altro widget già esistente (MapView). Il ServiceWidget è composto da due tabelle: nella tabella in alto vengono mostrati i messaggi che il widget scambia con il server: una sorta di log delle operazioni che il widget e il server effettuano in background. La seconda tabella mostra invece i servizi disponibili, questa è composta da tre colonne: la prima contiene l'id del servizio, la seconda il tipo del servizio, la terza una descrizione del servizio.

Number of messages received 4		
[18:13:27] [onNotify] [serId=Service3,serType=findPeople]		
id	typeOfService	description
Service3	findPeople	descrizine servizion 3
Service2	streamingVideo	descrizine servizion 2
Service1	misuraPM	prova sa sa provaaaa

Figura 3.10: ServiceWidget

Cliccando su una riga, come anticipato, l’utente selezionerà il servizio e tutte le sue entità per la visualizzazione sulla mappa. Il costruttore del widget contiene le dichiarazioni delle variabili, di seguito una spiegazione di quelle più importanti:

- **my_body**: è la variabile che contiene il body della pagina html
- **contenitore**: è il div che contiene la tabella dei messaggi scambiati tra widget e server
- **contenitore_tabella**: è il div che contiene la tabella dei servizi
- **table**: tabella che mostra i servizi disponibili
- **table_output**: tabella che contiene i messaggi scambiati tra il widget e il server
- **serviceList**: contiene la lista dei servizi disponibili

La struttura della pagina è quindi composta dal body principale e due div: **contenitore** e **contenitore_tabella**. Il primo div contiene la tabella in alto (messaggi) il secondo ospita la tabella in basso (servizi).

Di seguito il codice del costruttore:

```

1 var ServiceWidget = function ServiceWidget() {
2     this.my_body = null;
3     this.contenitore = null; /* stdout */
4     this.contenitore_tabella = null; /* div per la tabella */
5     this.connection = null;

```

```

6   this.ngsi_server = null;
7   this.ngsi_proxy = null;
8   this.subscriptionId = null;
9   this.table = null;
10  this.table_output = null;
11  this.serviceList = null;
12 };

```



Tabella 3.1: Div Service Widget

3.1.6.1 init()

Le variabili dichiarate nel costruttore sono inizializzate dalla funzione init:

```

1 ServiceWidget.prototype.init = function init() {
2     this.serviceList = [];
3     var my_body = document.getElementsByTagName("body")[0];
4     var contenitore = document.createElement("div");
5     contenitore.setAttribute("id","idContenitore");
6     var contenitore_tabella = document.createElement("div");
7     contenitore_tabella.setAttribute("id","tabellaContenitore");
8     this.my_body = my_body;
9     this.contenitore = contenitore;
10    this.contenitore_tabella=contenitore_tabella;
11    this.my_body.appendChild(this.contenitore);
12    this.my_body.appendChild(this.contenitore_tabella);

```

```

13     createTableOutPut . call ( this );
14     createTable . call ( this );
15     this . contenitore . appendChild ( this . table _ output );
16     this . contenitore _ tabella . appendChild ( this . table );
17     retrieveServicesFromServerNGSI . call ( this );
18 }

```

Si inizializza la variabile *my_body* assegnandole il riferimento al <body> della pagina, si creano i due tag <div> e li si assegna alle variabili *contenitore* e *contenitore_tabella*; si creano le due tabelle richiamando le funzioni *createTableOutPut()* e *createTable()*. Si inserisce la tabella dei messaggi (*table_output*) nel <div> *contenitore* e la tabella dei servizi nel <div> *contenitore_tabella*. In ultimo si richiama la funzione *retrieveServicesFromServerNGSI()* che effettua una sorta di discovery dei servizi presenti sul server attraverso una **subscribeContext**.

3.1.6.2 **createTable()** - **createTableOutput()**

Le due funzioni creano le due tabelle della pagina. In entrambe le chiamate vengono creati i seguenti tag:

- <**table**> è il tag html per la creazione della tabella
- <**tbody**> è il tag che rappresenta il corpo della tabella
- <**tr**> è il tag che rappresenta la singola riga della tabella
- <**th**> è il tag che rappresenta la singola colonna della tabella

E' importante dire che le funzioni si limitano a creare l'intestazione delle tabelle, le righe verranno aggiunte durante l'esecuzione del widget dalla funzione *updateTable()* che verrà presentata più avanti.

```

1 var createTableOutPut = function createTableOutPut () {
2     var table _ output = document . createElement ( "table" );
3     table _ output . setAttribute ( "id" , "outputTable" );
4     var tbody = document . createElement ( "tbody" );
5     table _ output . appendChild ( tbody );

```

```
6      var row_header = document.createElement("tr");
7      var new_cell = document.createElement("th");
8      new_cell.innerHTML = 'Number of messages received ' + tot_mess;
9      row_header.appendChild(new_cell);
10     tbody.appendChild(row_header);
11     this.table_output = table_output;
12   };
13
14 var createTable = function createTable() {
15   /* creo l'intestazione della tabella
16   * <table id="servicesTable">
17   * <tr>
18   * <th>id</th>
19   * <th>typeOfService</th>
20   * <th>description</th>
21   * </tr>
22   * </table> */
23   var num_col = 3;
24   var table = document.createElement("table");
25   table.setAttribute("id", "servicesTable");
26   var tbody = document.createElement("tbody");
27   table.appendChild(tbody);
28   var row_header = document.createElement("tr");
29   for(var i=0; i<num_col; i++){
30     var new_cell = document.createElement("th");
31     new_cell.innerHTML = tableHeaders[i];
32     row_header.appendChild(new_cell);
33   }
34   tbody.appendChild(row_header);
35   this.table = table;
36 };
```

3.1.6.3 retrieveServicesFromServerNGSI()

Questa funzione ha il compito di eseguire il discovery dei servizi sul server, per fare questo effettua una sottoscrizione ad Orion con id qualunque e tipo pari al valore Service⁷; in questo modo il broker invierà immediatamente un messaggio al widget con tutte le entità di tipo Service presenti fino a quel momento sul server, inoltre tutte le volte che un nuovo servizio verrà creato o uno già esistente aggiornato, il broker invierà un messaggio di notifica al widget⁸. Il widget in questo modo resterà sempre aggiornato sui servizi disponibili sul server e mostrerà i risultati nella tabella dei servizi che abbiamo visto nei paragrafi precedenti. Come prima cosa vengono settate le variabili per la connessione che sono:

- **ngsi_server**: rappresenta l'indirizzo ip del broker
- **ngsi_proxy**: rappresenta l'indirizzo ip del proxy ngsi
- **connection**: rappresenta la connessione al broker

Per settare *ngsi_server* e *ngsi_proxy* si fa uso della funzione *MashupPlatform.prefs.get()* presente nella libreria javascript di Wirecloud ; questa funzione permette di accedere al valore dei parametri di default del widget: in questo caso ai parametri *ngsi_server* e *ngsi_proxy*. Per creare la connessione e inizializzare la variabile *conenction* si usa un'altra funzione della medesima libreria *NGSI.Connection()*. Fatto questo siamo connessi al server Orion.

```

1 var retrieveServicesFromServerNGSI = function
2   retrieveServicesFromServerNGSI()
3 {
4   this.ngsi_server = MashupPlatform.prefs.get('
5     ngsi_server');
6   this.ngsi_proxy = MashupPlatform.prefs.get('
7     ngsi_proxy');
```

⁷Per comprendere la struttura dati utilizzata dal sistema si può consultare il paragrafo 1.3 a pagina 7

⁸La logica di funzionamento di una sottoscrizione è ampiamente trattata nel paragrafo 2.4.3 a pagina 27

```

5   this.connection = new NGSI.Connection(this.
6     ngsi_server,
{ use_user_fiware_token: true, ngsi_proxy_url: this.
  ngsi_proxy });

```

A questo punto dobbiamo effettuare la sottoscrizione; il primo passo consiste nell'inizializzare i parametri per la richiesta che sono i seguenti:

- **entityList**: lista delle entità su cui effettuare la sottoscrizione; in questo caso la lista è un'espressione regolare con id=.* e type=Service. In questo modo otteniamo una sottoscrizione a tutte le entità di tipo Service, cioè a tutti i servizi.
- **attributeList**: la lista degli attributi che verranno restituiti a seguito di un evento di notifica, nel nostro caso il vettore è nullo quindi Orion, nel messaggio di notifica, ritornerà tutti gli attributi delle entità a cui ci siamo sottoscritti.
- **duration**: la durata della sottoscrizione
- **throttling**: intervallo di tempo entro cui non considerare aggiornamenti successivi
- **notifyConditions**: rappresenta la lista delle condizioni che generano l'evento di notifica; nel nostro caso l'evento di notifica verrà generato al cambiamento (type=ONCHANGE) di uno degli attributi specificati nel vettore condValues(var condValues = ['typeOfService', 'description'])
- **options**: opzioni di connessione, in questa variabile si definiscono le funzioni da richiamare all'esito della sottoscrizione (**OnSuccess**) o in caso di notifica circa la creazione o l'aggiornamento di un servizio (**OnNotify**).

```

1 /* valorizzo i parametri da passare alla
   sottoscrizione
2 * entityList :

```

```
3 * attributeList :  
4 * duration :  
5 * throttling :  
6 * notifyConditions :  
7 * options :  
8 */  
9 var type = 'Service';  
10 var entityList = [] ; /* lista delle entità a cui  
    sottoscriversi */  
11 var attributeList = null;  
12 var duration = 'PT3H';  
13 var throttling = null;  
14 /* mi sottoscrivo a tutte le entità di tipo Service */  
15 var notifyConditions = [];  
16 var entity = { id: '.*' , type: type , isPattern: true  
    };  
17 entityList.push(entity);  
18 var conditions = { 'type': 'ONCHANGE' , 'condValues':  
    condValues }  
19 notifyConditions.push(conditions);  
20 var options = {  
21 flat: true ,  
22 onNotify: handlerReceiveEntity.bind(this) ,  
23 onSuccess: function (data)  
24 {  
25 this.subscriptionId = data.subscriptionId;  
26 printDebug.call(this, 'onSuccess', data);  
27 this.refresh_interval=setInterval(  
    refreshNGSISubscription.bind(this),1000*60*60*2);  
28 window.addEventListener("beforeunload",function()  
29 {  
30 this.connection.cancelSubscription(this.  
    subscriptionId);
```

```

31 } . bind( this ) ;
32 } . bind( this )
33 } ;
34 this . connection . createSubscription( entityList ,
    attributeList , duration , throttling , notifyConditions ,
    options ) ;
35 } ;

```

Una volta che tutti i parametri per la sottoscrizione sono stati settati non rimane altro che eseguire la richiesta al server. L'ultima riga di codice richiama proprio la funzione che effettua la sottoscrizione: *createSubscription()*.

3.1.6.4 handlerReceiveEntity()

Questa funzione viene richiamata dalla *retrieveServicesFromNGSI()* quando il server invia al widget un messaggio di notifica circa le entità sulle quali abbiamo effettuato la sottoscrizione. Il messaggio di notifica può essere di due tipi: o si riferisce ad un servizio già esistente i cui attributi hanno subito un aggiornamento, oppure si riferisce ad un nuovo servizio che è stato appena creato. La funzione dunque, come prima cosa, si occupa di capire a quale delle due tipologie appartiene il messaggio in arrivo. Nel primo caso (servizio aggiornato) scorre la lista dei servizi per selezionare il servizio che è cambiato e aggiornare il suo contenuto. Nel caso di nuovo servizio, viene aggiunto un nuovo elemento nella lista dei servizi. In entrambi i casi viene richiamata la funzione *printDebug()* che si occupa di stampare nella tabella dei messaggi le informazioni relative al nuovo messaggio di notifica ricevuto. In ultimo, vengono richiamate in successione le funzioni *deleteTable()* e *updateTable()* per aggiornare la tabella dei servizi con i nuovi valori. Infatti i valori di notifica relativi ad un servizio preesistente o ad un nuovo servizio sono in questo momento salvati nella lista dei servizi; per aggiornare la tabella dunque, basterà cancellare le righe della tabella con la *deleteTable()* e scorrere la lista dei servizi per creare la nuova tabella sfruttando la *updateTable()*.

```
1 var handlerReceiveEntity = function
2   handlerReceiveEntity(data)
3 {
4   var serviceDataList = data.elements;
5   for (var serviceId in serviceDataList) {
6     var service = serviceDataList[serviceId];
7     /*
8      * controllo se il servizio è nella lista dei
9       servizi
10     * in caso affermativo devo aggiornare i dati nella
11       lista
12     * altrimenti devo inserire un nuovo servizio nella
13       lista
14   */
15   if(serviceId in this.serviceList){
16     //vecchio servizio aggiornato
17     delete this.serviceList[serviceId];
18     var updated_id = service.id;
19     var updated_description = service.description;
20     var updated_type = service.typeOfService;
21     var updated_service = {};
22     updated_service.id = updated_id;
23     updated_service.description =
24       updated_description;
25     updated_service.typeOfService = updated_type;
26     this.serviceList[serviceId] = updated_service;
27     printDebug.call(this, 'onNotify', service);
28   } else{//nuovo servizio
29     var new_id = service.id;
30     var new_description = service.description;
31     var new_type = service.typeOfService;
32     var new_service = {};
33     new_service.id = new_id;
```

```

29         new_service.description = new_description;
30         new_service.typeOfService = new_type;
31         this.serviceList[serviceId] = new_service;
32         printDebug.call(this, 'onNotify', new_service);
33     }
34 }
/*Quando ho finito di aggiornare la serviceList
 * cancello tutte le righe della tabella
 * e scrivo le nuove righe con i valori aggiornati
 * in serviceList */
39 deleteTable.call(this);
40 updateTable.call(this);
41 };

```

3.1.6.5 deleteTable()

La funzione è molto semplice, controlla se il numero di righe è maggiore di uno, cioè se c'è qualche riga oltre quella d'intestazione e in caso affermativo cancella tutte le righe della tabella dei servizi.

```

1 var deleteTable = function deleteTable(){
2     var table = document.getElementById("servicesTable");
3     var nrows = table.rows.length;
4     if(nrows>1){
5         //se c'è qualche riga (row=1 header)
6         for(var i in this.serviceList) { table.
7             deleteRow(1); }
8     }

```

3.1.6.6 updateTable()

La funziona scorre tutti gli elementi presenti nella lista dei servizi (variabile *serviceList*) e crea una nuova riga per ogni singolo elemento.

```

1 var updateTable = function updateTable() {
2     /* inserisce in testa tutti i servizi
3      * presenti in this.serviceList
4     */
5     var table = document.getElementById("servicesTable")
6         ;
7     for(var i in this.serviceList){
8         var row = table.insertRow(1);
9         var cell_id = row.insertCell(0);
10        var cell_type = row.insertCell(1);
11        var cell_desc = row.insertCell(2);
12        var tmp_service = this.serviceList[i];
13        cell_id.innerHTML = tmp_service.id;
14        cell_type.innerHTML = tmp_service.typeOfService;
15        cell_desc.innerHTML = tmp_service.description;
16        row.addEventListener("click", handleClickRow.bind
17            (this, row), false);
18    }
19 };

```

3.1.7 QueryWidget

Il QueryWidget ha il compito di selezionare tutte le entità di uno specifico servizio. Il widget riceve in ingresso, dal ServiceWidget a cui è connesso attraverso l'interfaccia d'ingresso *ServiceEntity*, il servizio che l'utente ha selezionato e effettua le query per identificare tutte le entità che prendono parte al servizio d'interesse. Anche questo widget è stato interamente scritto da zero per i nostri scopi. Il modulo, oltre ad un ingresso di cui abbiamo già parlato, ha anche due interfacce di uscita: *ListOfEntities* e *ListOfEntitiesDel* (vedi figura). Queste interfacce di uscita devono essere connesse agli ingressi di un altro widget MapViewer, di cui parleremo successivamente, per permettere la rappresentazione delle entità su una mappa. La prima interfaccia (*ListOfEntities*) fornisce in uscita tutte le entità identificate che devono essere mostrate sulla mappa; la seconda interfaccia (*ListOfEntitiesDel*)

sDel) è invece usata per eliminare dalla mappa le entità del servizio precedentemente selezionato. Infatti, quando l'utente seleziona il nuovo servizio d'interesse, le entità del servizio precedente devono essere cancellate dalla mappa; l'interfaccia *ListOfEntitiesDel* svolge proprio il compito di fornire al MapViewer la lista delle entità da eliminare dalla mappa prima di inserire quelle relative al nuovo servizio da mostrare.

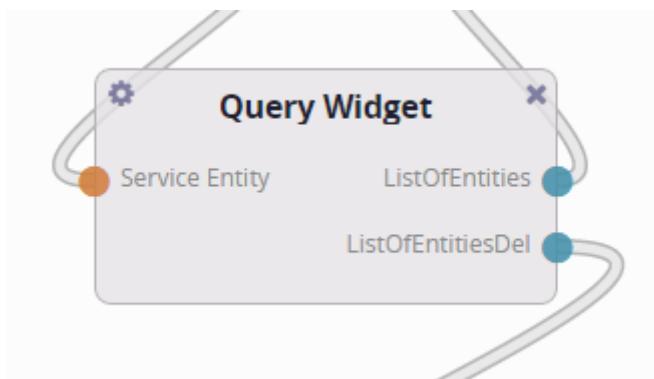


Figura 3.11: Interfacce di input/output QueryWidget

Come è possibile notare dalla figura successiva, anche questo widget è composto da due tabelle: quella superiore, usata a scopi diagnostici, mostra i messaggi scambiati con il server; la tabella inferiore invece mostra i risultati delle query effettuate, in altri termini mostra tutte le entità che il widget ha identificato per un particolare servizio.

Il costruttore contiene la dichiarazione delle variabili più importanti:

- **ngsi_server**: indirizzo del server Orion da contattare
- **serviceId**: id del servizio selezionato dall'utente. Questo valore è ottenuto o dai parametri di impostazione del widget o dall'interfaccia di ingresso verso il ServiceWidget
- **table**: tabella di output che contiene i risultati delle query
- **table_footer**: tabella di diagnostica che contiene i messaggi scambiati con il server
- **contenitore**: div che contiene la tabella delle query

Query Widget																																																																											
[16:32:53] [onSuccess]			2																																																																								
[16:32:52] [The selected service is Service1]	0																																																																										
[16:32:39] [onSuccess]			1																																																																								
<table><thead><tr><th>id</th><th>type</th><th>position</th><th>idQuery</th></tr></thead><tbody><tr><td>pm10_142306557487</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142306536171</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_14230652865</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142306510095</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142306496762</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142306484855</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142306463167</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142306448282</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142306434912</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142297318867</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142297176941</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142297053045</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_14229692767</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142296811378</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>pm10_142296657428</td><td>observations</td><td>undefined</td><td>2</td></tr><tr><td>Taxi2</td><td>Taxi</td><td>undefined</td><td>2</td></tr><tr><td>Taxi1</td><td>Taxi</td><td>undefined</td><td>2</td></tr></tbody></table>				id	type	position	idQuery	pm10_142306557487	observations	undefined	2	pm10_142306536171	observations	undefined	2	pm10_14230652865	observations	undefined	2	pm10_142306510095	observations	undefined	2	pm10_142306496762	observations	undefined	2	pm10_142306484855	observations	undefined	2	pm10_142306463167	observations	undefined	2	pm10_142306448282	observations	undefined	2	pm10_142306434912	observations	undefined	2	pm10_142297318867	observations	undefined	2	pm10_142297176941	observations	undefined	2	pm10_142297053045	observations	undefined	2	pm10_14229692767	observations	undefined	2	pm10_142296811378	observations	undefined	2	pm10_142296657428	observations	undefined	2	Taxi2	Taxi	undefined	2	Taxi1	Taxi	undefined	2
id	type	position	idQuery																																																																								
pm10_142306557487	observations	undefined	2																																																																								
pm10_142306536171	observations	undefined	2																																																																								
pm10_14230652865	observations	undefined	2																																																																								
pm10_142306510095	observations	undefined	2																																																																								
pm10_142306496762	observations	undefined	2																																																																								
pm10_142306484855	observations	undefined	2																																																																								
pm10_142306463167	observations	undefined	2																																																																								
pm10_142306448282	observations	undefined	2																																																																								
pm10_142306434912	observations	undefined	2																																																																								
pm10_142297318867	observations	undefined	2																																																																								
pm10_142297176941	observations	undefined	2																																																																								
pm10_142297053045	observations	undefined	2																																																																								
pm10_14229692767	observations	undefined	2																																																																								
pm10_142296811378	observations	undefined	2																																																																								
pm10_142296657428	observations	undefined	2																																																																								
Taxi2	Taxi	undefined	2																																																																								
Taxi1	Taxi	undefined	2																																																																								

Figura 3.12: QueryWidget

- **footer:** div che contiene la tabella dei messaggi

Questo è il codice del costruttore:

```

1 var QueryWidget = function QueryWidget() {
2     this.ngsi_server = null;
3     this.connection = null;
4     this.attribute_name = null;
5     //this.serviceId = "Service1";
6     this.serviceId = null;
7     this.contenitore = null;
8     this.my_body = null;
9     this.table = null;
10    this.footer = null;
11    this.table_footer = null;
12 };

```

3.1.7.1 init()

In questa funzione si inizializzano le variabili del widget. Si noti come l'indirizzo del server è impostato attraverso la funzione *MashupPlatform.prefs.get()* che permette di leggere il valore dai parametri di impostazione del widget, stesso discorso vale per l'id del servizio (variabile *serviceID*). Questo permette all'utente di impostare il servizio da ricercare e il server da contattare attraverso il menù setting del widget. Successivamente si procede alla creazione delle due tabelle attraverso le chiamate *createTableFooter()* e *createTableOutPut()*. La prima funzione crea la tabella dei messaggi mentre la seconda inizializza la tabella delle query. Inoltre si provvede ad inizializzare la connessione (variabile *connection*) con il server attraverso la funzione *create_ngsi_connection()*. Si registra la funzione *handlerEntityInput()* da richiamare in caso di ricezione di un nuovo servizio proveniente dall'interfaccia di ingresso *InputEntity*; si fa ciò attraverso la funzione *MashupPlatform.wiring.registerCallback()*. Infine si richiama la funzione *standardRoutine()* che controlla se il *serviceID* è stato settato nelle impostazioni del widget e in caso affermativo esegue la query. L'avvio del processo di inter-

rogazione può partire quindi sia dalla funzione *standardRoutine()*, sia dalla funzione *handlerEntityInput()*.

```

1 QueryWidget.prototype.init = function init() {
2     this.ngsi_server = MashupPlatform.prefs.get('
3         ngsi_server');
4     this.serviceId = MashupPlatform.prefs.get('
5         service_id');
6     var my_body = document.getElementsByTagName("body")
7         [0];
8     var contenitore = document.createElement("div");
9     contenitore.setAttribute("id","idContenitore");
10    this.my_body = my_body;
11    this.contenitore = contenitore;
12    createTableFooter.call(this);
13    this.footer = document.createElement("div");
14    this.footer.setAttribute("id","footer");
15    this.footer.appendChild(this.table_footer);
16    this.my_body.appendChild(this.footer);
17    this.connection = create_ngsi_connection();
18    MashupPlatform.wiring.registerCallback("InputEntity"
19        , handlerEntityInput.bind(this));
20    MashupPlatform.prefs.registerCallback(
21        handlerPreferences.bind(this));
21    standardRoutine.call(this);
21 };

```

3.1.7.2 `createTableFooter()` - `createTableOutPut()`

Le due funzioni si occupano rispettivamente di creare l'intestazione per la tabella dei messaggi e delle query. La prima tabella è molto semplice, contiene infatti una sola colonna in cui viene scritto il messaggio. La tabella

delle query ha invece quattro colonne: l'id dell'entità, il tipo, la posizione dell'entità e un id relativo alla query effettuata. Per creare l'intestazione della tabella è necessario aggiungere il tag `<tr>` che rappresenta appunto la riga d'intestazione e successivamente appendere a questo un numero di tag `<th>` pari al numero di colonne della tabella.

```

1 var createTableFooter = function createTableFooter()
2 {
3     var table_footer = document.createElement("table")
4         ;
5     table_footer.setAttribute("id","footerTable");
6     var tbody = document.createElement("tbody");
7     table_footer.appendChild(tbody);
8     this.table_footer = table_footer;
9 };
10
11 var createTableOutPut = function createTableOutPut()
12 {
13     var num_col = 3;
14     var table_output = document.createElement("table")
15         ;
16     table_output.setAttribute("id","outputTable");
17     var tbody = document.createElement("tbody");
18     table_output.appendChild(tbody);
19     var row_header = document.createElement("tr");
20     for(var i=0; i<num_col; i++)
21     {
22         var new_cell = document.createElement("th");
23         new_cell.innerHTML = taxi_attributes[i];
24         row_header.appendChild(new_cell);
25     }
26     //creo la cella dell'id della query
27     var idquery_cell = document.createElement("th");
28     idquery_cell.innerHTML = "idQuery"
29     row_header.appendChild(idquery_cell);
30 }
```

```

25      tbody.appendChild(row_header);
26      this.table = table_output;
27  };

```

3.1.7.3 handlerEntityInput()

Questa è la funzione che viene richiamata alla ricezione di un servizio dall’interfaccia d’ingresso del widget. Il suo compito è quello di scrivere sulla tabella dei messaggi quale servizio è stato ricevuto e in quale istante. Fatto questo la funzione richiama la *doQuery()* che avrà il compito di effettuare la query per trovare tutte le entità appartenenti al servizio.

```

1 var handlerEntityInput = function handlerEntityInput(
2     serviceEntityString){
3     var serviceEntity = JSON.parse(serviceEntityString);
4     var serviceID = serviceEntity.id;
5     this.serviceId = serviceID;
6     /*
7      * scrivo sull' header della outputTable il servizio
8      * selezionato
9     */
10    var d = new Date();
11    var h = d.getHours();
12    var m = d.getMinutes();
13    var s = d.getSeconds();
14    var current_date = '[' +h+':'+m+':'+s+]';
15    var footer = document.getElementById("footerTable");
16    var row = footer.insertRow(0);
17    var mgs = row.insertCell(0);
18    mgs.innerHTML = current_date + '[' +'The selected
19    service is ' + this.serviceId + ']';
20    var idMsg = row.insertCell(1);
21    idMsg.innerHTML = "0";
22    doQuery.call(this);
23  };

```

3.1.7.4 doQuery()

Per effettuare la ricerca delle entità viene usata la funzione *query()* messa a disposizione dalla libreria javascript NGSI di wirecloud. La query è molto semplice: si ricercano tutte le entità di qualunque tipo e qualunque id che però abbiamo un attributo pari al contenuto della variabile *servID*; *servID* contiene l'identificativo del servizio d'interesse. In questo modo vengono selezionate tutte le entità di qualunque tipo che prendono parte al servizio richiesto. Infine si registrano le funzioni da richiamare in caso di successo della query (**onSuccess**) e in caso di fallimento (**onFailure**).

```

1 var doQuery = function doQuery () {
2     var servID = this.serviceId ;
3     var position = "position" ;
4     this.connection.query (
5         [{ type : "" ,
6             isPattern : true ,
7             id : ".*" }] ,
8         [servID] ,
9         { flat : true ,
10             onSuccess: onQuerySuccess . bind ( this ) ,
11             onFailure: onQueryFail . bind ( this )
12         }
13     );
14 };

```

3.1.7.5 onQuerySucess()

Questa funzione è richiamata al successo della prima query eseguita per trovare le entità appartenenti al servizio. Il parametro passato in ingresso contiene il risultato della query, cioè la lista di tutte le entità trovate. In primis si inserisce una nuova notifica nella tabella dei messaggi per informare dell'esito positivo della query, successivamente si aggiorna la tabella delle query per mostrare le nuove entità: per fare ciò si richiama la funzione *deleteTableOutPut()* e *updateTableOutPut()*, inoltre si invia al MapViewer la

lista delle entità relative al servizio precedente in modo che il widget li canelli dalla mappa per far posto alle nuove entità che sono appena state trovate. Infine si effettua un ciclo for per eseguire una query su ogni tipo di entità trovata in modo da ottenere tutti gli attributi. Ricapitolando: si esegue prima una query, attraverso la funzione doQuery(), per trovare le entità del servizio e successivamente, all'interno della funzione onQuerySuccess(), si effettua una query per ciascuna entità trovata.

```

1 var onQuerySuccess = function onQuerySuccess(data){
2     var entitiesList = data;
3     id_query = id_query + 1;
4     var d = new Date();
5     var h = d.getHours();
6     var m = d.getMinutes();
7     var s = d.getSeconds();
8     var current_date = '['+h+':'+m+':'+s+]';
9     //aggiorno la tabella footer
10    var footer = document.getElementById("footerTable")
11        );
12    var row = footer.insertRow(0);
13    var mgs = row.insertCell(0);
14    mgs.innerHTML = current_date + '[+' + 'onSuccess' +
15        ']';
16    var mgsQueryId = row.insertCell(1);
17    mgsQueryId.innerHTML = id_query;
18    /*aggiungo l'handler per la riga,
19     * quando verrà cliccata la riga verranno
20     * caricate le response della query nella
21     * tabella outputTable
22     */
23    row.addEventListener("click", handleClickRow.bind
24        (this, row), false);
25    //salvo in lista i risultati della query
26    var oneElement = {};

```

```
24     oneElement.response = data; /* oggetti taxi */
25     oneElement.id = id_query;
26     listDataQuery[id_query] = oneElement;
27     //richiamo la deleteTableOutPut()
28     //cancello tutte le righe tranne l'header
29     deleteTableOutPut.call(this);
30     //richiamo la updateTableOutPut()
31     updateTableOutPut.call(this, oneElement);
32     //prima di fare la query per le nuove entita'
33     //invio il comando di eliminazione delle
34     //vecchie entità nella mappa
35     if(id_query > 1) {
36         var id_del = id_query-1;
37         var old_data = listDataQuery[id_del];
38         for(var e in old_data.response){
39             var entity_to_del = old_data.response[e];
40             var json_entity = JSON.stringify(entity_to_del);
41             MashupPlatform.wiring.pushEvent(
42                 "OutputEntitiesDel", json_entity);
43             }
44         }
45         for(var attr in entitiesList){
46             //var entity = JSON.stringify(entitiesList[
47             //attr]);
48             var entity = entitiesList[attr];
49             var taxiID = "";
50             var taxiType = "";
51             taxiID = entity.id;
52             taxiType = entity.type;
53             var ent_ID = entity.id;
54             var ent_Type = entity.type;
```

```
54     //new code test
55     var position = "position";
56     var temperature = "temperature";
57     var pressure = "pressure";
58     var pm10 = "PM10";
59     if(ent_Type == "Taxi"){
60         this.connection.query([{
61             type : ent_Type,
62             isPattern : false ,
63             id : ent_ID }], [
64             [this.serviceId ,position] ,
65             { flat: true ,
66             onSuccess:onQuerySuccess1.bind(this) ,
67             onFailure: onQueryFail1.bind(this)
68             }
69         );
70     }
71     else{
72         this.connection.query([{
73             type : ent_Type,
74             isPattern : false ,
75             id : ent_ID }],null ,
76             { flat:true ,
77             onSuccess: onQuerySuccess1.bind(this) ,
78             onFailure: onQueryFail1.bind(this)
79             }
80         );
81     }
82 }
83
84 };
```

3.1.7.6 deleteTableOutPut()

La funzione cancella tutte le righe della tabella delle query esclusa la prima riga d'intestazione.

```

1 var deleteTableOutPut = function deleteTableOutPut(){
2     var table = document.getElementById("outputTable")
3         ;
4     var nrows = table.rows.length;
5     if(nrows>1){ // se c'è almeno una riga (row=1 è il
6         // header)
7         for(var i=0; i<nrows-1; i++){ // l'header non
8             // lo cancello
9             table.deleteRow(1);
10        }
11    }
12 };

```

3.1.7.7 updateTableOutPut()

La funzione si occupa di inserire nella tabella delle query tutte le entità di un servizio. Il parametro *data*, fornito in ingresso, contiene un id (variabile *data.id*) che identifica la query effettuata e tutti i risultati di risposta: cioè le entità di un servizio (variabile *data.response*). I risultati della query vengono copiati nella variabile *queryList* e successivamente si effettua un ciclo for per inserire ciascun elemento della lista in tabella. Per ogni elemento si inseriscono attraverso la funzione *insertCell()*: l'id dell'entita, il tipo, la posizione (latitudine e longitudine) e l'id della query.

```

1 var updateTableOutPut = function updateTableOutPut(
2     data){
3     /* inserisce tutti i risultati della query
4      * selezionata nella tabella outputTable
5      * data è un oggetto oneElement contenuto in
6      * listDataQuery
7      * oneElement.id -> id della query
8 };

```

```

6      * oneElement.response -> oggetti taxi .id .type .
7          position
8      */
9      var queryList = data.response;
10     var id = data.id;
11     var table = document.getElementById("outputTable")
12     ;
13     for(var i in queryList){
14         var row = table.insertRow(1);
15         var cell_id = row.insertCell(0);
16         var cell_type = row.insertCell(1);
17         var cell_pos = row.insertCell(2);
18         var cell_idquery = row.insertCell(3);
19         var tmp = queryList[i];
20         cell_id.innerHTML = tmp.id;
21         cell_type.innerHTML = tmp.type;
22         cell_pos.innerHTML = tmp.taxiPosition;
23         cell_idquery.innerHTML = id;
24     }
25 };

```

3.1.7.8 onQuerySuccess1()

Questa funzione viene richiamata ogni volta che la query sulla singola entità ha avuto successo. Il parametro *data1*, ricevuto in ingresso, contiene tutti gli attributi dell'entità oggetto della query. Si esegue quindi un ciclo for per prelevare un passo alla volta tutti gli attributi dell'entità. Alla fine del ciclo sarà stato completamente aggiornato l'elemento corrispondente all'entità nella lista da mandare al MapViewer per la cancellazione (variabile *listaDataQuery*). Questa operazione è necessaria in quanto la prima query valorizza solamente l'attributo Service di tutte le entità, la seconda query recupera invece tutti gli attributi dell'entità tra cui la posizione che è indispensabile al MapViewer per cancellare le entità sulla mappa. Infine si invia

l'entità (completamente valorizzata) al MapViewer attraverso l'interfaccia di uscita *OutputEntities* in modo che questa sia visualizzata sulla mappa.

```
1 var onQuerySuccess1 = function onQuerySuccess1(data1){  
2     var taxiEntity = "";  
3     var Entity = "";  
4     for(var attr1 in data1){  
5         taxiEntity = JSON.stringify(data1[attr1]);  
6         /*devo aggiornare le entita' nella lista  
7          *in quanto la prima query valorizza solamente  
8          *il campo Service, la seconda query prende  
9              tutti  
10             gli altri valori, anche la posizione che e'  
11             *indispensabile per la cancellazione nella  
12                 mappa  
13             *att1 = id entita, data1[attr1] oggetto entita  
14                 'in lista  
15             */  
16             var response = listDataQuery[id_query].  
17                 response;  
18             var element_to_update = response[attr1];  
19             delete listDataQuery[id_query].response[attr1]  
20             ;  
21             listDataQuery[id_query].response[attr1] =  
22                 data1[attr1];  
23             Entity = JSON.stringify(data1[attr1]);  
24             MashupPlatform.wiring.pushEvent(  
25                 "OutputEntities", Entity);  
26         }  
27     };
```

3.1.8 Wiring e altri widget utilizzati

I due widget descritti in precedenza sono stati realizzati da noi in quanto non esistevano moduli che implementassero le funzionalità che ci servivano. In questo paragrafo invece mostreremo i widget già esistenti che abbiamo utilizzato per creare il nostro mashup. La figura di sotto mostra l'immagine del wiring di tutti i componenti connessi insieme.

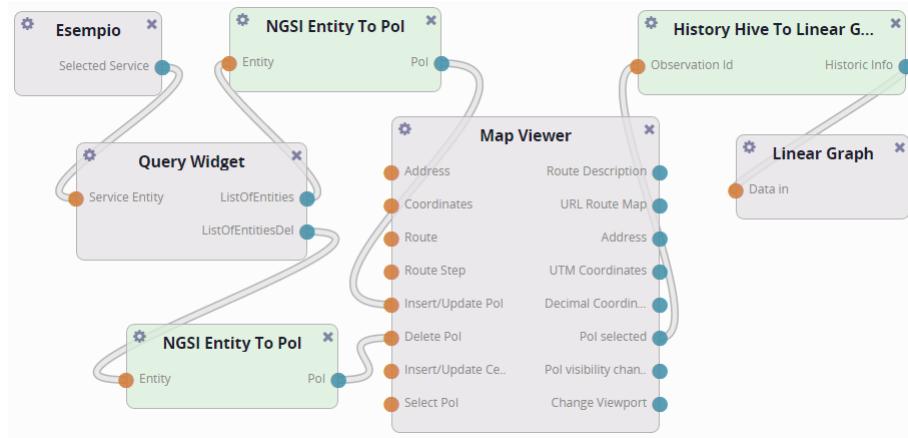


Figura 3.13: Wiring del mashup

Il flusso di scambio di messaggi si svolge in quest'ordine:

- Il **ServiceWidget** mostra i servizi disponibili in tabella (figura di sotto), l'utente clicca sul nome di un servizio e il modulo invia al **QueryWidget** l'id del servizio che l'utente ha selezionato.

Number of messages received 7		
[12:54:9] [onNotify] [serId=Service3,serType=misuraPressione]		
[12:54:9] [onNotify] [serId=Service2,serType=misuraTemperatura]		
[12:54:9] [onNotify] [serId=Service1,serType=misuraPM]		
[10:54:8] [onNotify] [serId=Service3,serType=misuraPressione]		
[10:54:8] [onNotify] [serId=Service2,serType=misuraTemperatura]		
[10:54:8] [onNotify] [serId=Service1,serType=misuraPM]		
[10:54:8] [onSuccess] [subId=54e4613e985d83c7352b41fb,duration=PT3H]		

id	typeOfService	description
Service3	misuraPressione	misurazioni pressione in città
Service2	misuraTemperatura	misurazioni temperatura in città
Service1	misuraPM	misurazioni pm10 in città

Figura 3.14: Service Widget

- Il **QueryWidget** effettua le query e individua così tutte le entità che partecipano a quel dato servizio. La lista delle entità è poi passata attraverso interfaccia d'uscita **ListOfEntities** all'operatore NGSI Entity To PoI. Sulla seconda interfaccia **ListOfEntitiesDel** viene invece inviata la lista delle entità appartenenti al servizio precedentemente selezionato: queste entità verranno cancellate dalla mappa per far posto a quelle in uscita dall'interfaccia **ListOfEntities**.
- **NGSI Entity To PoI**: questo è un operatore, il suo compito è quello di trasformare i dati in ingresso (entità) in un formato adatto alle interfacce d'ingresso del MapViewer (Point of Interest PoI). Le entità quindi sono trasformate in PoI e passate alle due interfacce d'ingresso del MapViewer **Insert/Update Poi** (inserimento nuove entità) e **Delete Poi** (cancellazione vecchie entità).
- Il **MapViewer**, ricevute le liste delle entità in formato PoI sulla due interfacce, cancellerà dalla mappa quelle in ingresso sulla **Delete Poi** e inserirà quelle provenienti dalla **Insert/Update Poi**. Nella figura di sotto un esempio di osservazioni di particolato mostrate sul MapViewer.

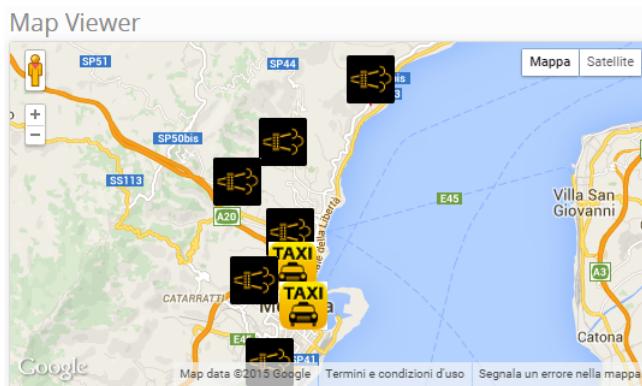


Figura 3.15: Entità sul MapViewer

- L'utente a questo punto visualizzerà sulla mappa le entità del servizio che aveva selezionato sul ServiceWidget, cliccando su una delle entità presenti sulla mappa potrà visualizzare il grafico della misurazione.

A seguito del doppio click dell’utente, l’id dell’entità selezionata verrà inviata all’operatore **History Hive to Linear Graph** attraverso l’interfaccia d’uscita **Poi selected**. L’Hisotry Hive to Linear Graph è un operatore che ha il compito di contattare il **restServerHive** (scritto da noi) per richiedere i dati della misurazione che sono memorizzati su Cosmos. Il server da contattare può essere impostato attraverso il campo setting dell’operatore, come è mostrato nella figura in basso.

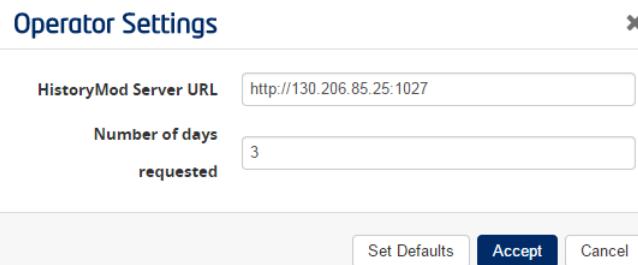


Figura 3.16: Setting History Hive to Linear Graph

- L’ History Hive To Linear Graph passa infine i dati della misurazione al Linear Graph che ne mostrerà il grafico.



Figura 3.17: Grafico pm10 sul Linear Graph

3.1.9 historyHiveToLinearGraph

Questo operatore era già implementato, ma è stato necessario modificarne il codice per adattarlo ai nostri scopi. Il suo compito, come anticipato

nel paragrafo precedente, è quello di ricevere dall’ interfaccia d’ ingresso il PoI della misurazione, contattare il restServerHive per prelevare i dati dell’osservazione e passare questi ultimi al LinearGraph per la visualizzazione. Vediamo in breve il codice del modulo.

La prima cosa da fare è registrare la funzione *handlerSensorIdInput()* affinchè questa sia richiamata ogni volta che l’operatore riceve un PoI dalla propria interfaccia d’ingresso *ObservationIdInput*.

```
1 MashupPlatform.wiring.registerCallback( "
    ObservationIdInput" , handlerSensorIdInput . bind( this
) );
```

Alla ricezione di un PoI, la funzione si limita a controllare se l’oggetto è non nullo e in caso affermativo a richiamare la funzione *changeObs()* passando in ingresso il PoI ricevuto.

```
1 var handlerSensorIdInput = function
    handlerSensorIdInput( sensorId ) {
2     if ( sensorId ) {
3         changeObs . call( this , sensorId );
4     }
5 };
```

La funzione *changeObs()*, ricevuto in ingresso il PoI, effettua una parsing json del dato, memorizzando l’id dell’entità nella variabile *poiID* e i suoi attributi nella variabile *entityDataPOI*. Se il tipo dell’entità è observations si controlla a quale tipo particolare di osservazione questa appartiene: pm10 (Service1), temperature (Service2), pressure (Service3). Service1, Service2 e Service3 sono gli id associati rispettivamente ai servizi di misurazione del particolato, della temperatura e della pressione e altro non sono che attributi dell’entità observations. Una trattazione completa della struttura dati utilizzata nel sistema è presentata nel paragrafo 1.3 a pagina 7.

```
1 var changeObs = function changeObs( sensorPOI ) {
2     var sensorId , from , to , sensorParsed , numberOfDays
        , today;
3     var entityDataPOI;
```

```

4
5     sensorParsed = JSON.parse(sensorPOI);
6     entityDataPOI = sensorParsed.poi.data;
7     poiID = sensorParsed.poi.id;
8     //se l'entità cliccata sulla mappa è un
9     //observations
10    if(entityDataPOI.type == "observations"){
11        poiType = "observations";
12        //controllo qual'è l'observed_property dell'
13        //observations
14        //sulla base della presenza dell'attributo
15        //Service1/2/3
16        if(entityDataPOI.Service1)
17            observed_prop = "pm10";
18        else if(entityDataPOI.Service2)
19            observed_prop = "temperature";
20        else if(entityDataPOI.Service3)
21            observed_prop = "pressure";
22    }
23
24    //se l'entità cliccata sulla mappa è un taxi
25    else if(entityDataPOI.type == "Taxi"){
26        poiType = "Taxi";
27    }

```

Al termine di questi passi avremo in *poiType* il tipo dell'entità (observations o Taxi) e in *observed_prop* il tipo specifico di observations: pm10, temperature o pressure. A questo punto si devono inserire nella variabile *requestParams* i parametri della richiesta rest che sarà effettuata al restServerHive, nello specifico vengono impostati il metodo http da eseguire (GET) e le funzioni da richiamare in caso di successo della richiesta (**onSuccess**) e di fallimento (**onFailure**). Successivamente, si invia al LinearGraph, connesso sull'interfaccia d'uscita *OutputStatus*, un messaggio di notifica circa il PoI che è stato identificato, e, nel caso in cui il PoI sia di tipo observations,

si effettua la chiamata REST attraverso la funzione *makeRequest()* messa a disposizione dalla libreria nativa di Wirecloud. E' importante notare che la *makeRequest()* invia al restServerHive altri due parametri: l'id dell'entità (*poiID*) ed il tipo (*observations*), questi parametri saranno utili al server al fine di selezionare la tabella corretta che contiene i dati dell'osservazione.

```

1      var requestParams = {
2          "method": "GET",
3          "onSuccess": processHistoricData.bind(this),
4          "onFailure": onFailure.bind(this)
5      };
6
7      // Send loading msg by OutputStatus endpoint
8      var loadMsg = 'loading ' + sensorId;
9      MashupPlatform.wiring.pushEvent("OutputStatus",
10         JSON.stringify(loadMsg));
11
12     var url = MashupPlatform.prefs.get(
13         'historymod_hiveserver');
14
15     if(poiType.match('Taxi')){
16         loadMsg = "It's not possible to graph data of
17             a taxi!\nClick on an observation on the map
18             ";
19         MashupPlatform.wiring.
20             pushEvent("OutputStatus", JSON.stringify(
21                 loadMsg));
22     }else if(poiType.match('observations')){
23         MashupPlatform.http.makeRequest(url + "/"
24             "observations?id=" + poiID + "&type=
25             observations", requestParams);
26     }
27 };

```

Se la richiesta ha esito negativo si invia al LinerGraph, sull'interfaccia di comunicazione *OutputStatus*, un messaggio di errore.

```

1 var onFailure = function onFailure(/*sensorId*/) {
2     var failMsg = 'Error: Hive server unreacheable!';
3     MashupPlatform.wiring.pushEvent("OutputStatus",
4         JSON.stringify(failMsg));

```

Altrimenti si procede ad elaborare i dati richiamando la funzione *processHistoricData()*. Per prima cosa si provvede ad effettuare un parsing json dei dati (variabile *response*) di risposta restituiti dal restServerHive. La variable *parsedResponse* conterrà dunque tutte le misurazioni dell'osservazione selezionata in formato json. Successivamente si impostano i parametri di configurazione del LinearGraph (le label per ascisse e ordinate, il range di variazione dei dati) e li si inviano sull'interfaccia di uscita attraverso la funzione *pushEvent()*.

```

1 var processHistoricData = function processHistoricData
2     (response) {
3         var i, parsedResponse, data, id, lampObservationId
4             , config;
5
6         var parsedResponseText = response.responseText;
7         parsedResponse = JSON.parse(response.responseText)
8             ;
9
10        if (poiType.match('observations')) {
11            // Axis config
12            if(observed_prop == "temperature"){
13                config = {
14                    'axisConfig': [
15                        {
16                            axisId: 0,
17                            label: observed_prop,
18                            color: '#00A8F0',
19                            max: 300,
20                            min: -300,
21                            type: 'linear'
22                        }
23                    ]
24                }
25            }
26        }
27    }
28
29    MashupPlatform.wiring.pushEvent("OutputStatus",
30        JSON.stringify(parsedResponse));
31
32    onFailure();
33}

```

```

18           //ticks: [-300, -150, -100, -50, 0,
19             50, 100, 150, 200, 250, 300]
20           ticks : null
21       },
22   'title': poiID,
23   'leyend': {
24     position: 'ne'
25   },
26   'oneData': true
27 };
28
29 MashupPlatform.wiring.pushEvent("OutputStatus",
  JSON.stringify(config));

```

Infine attraverso un ciclo for si inviano al LinearGraph i dati della misurazione un campione per volta.

```

1   data = [];
2   for (i = parsedResponse.length - 1; i > 0; i -= 1)
3   {
4     var elem = JSON.parse(parsedResponse[i]);
5     data.push({
6       'id': 0,
7       'value': [elem.date, elem.value],
8       'label': observed_prop,
9       'axis': 1,
10      'color': '#00A8F0'
11    });
12 }

```

Capitolo 4

Studio delle prestazioni

4.1 Scenario e prove effettuate

Per lo studio delle prestazioni dell'applicazione da noi realizzata abbiamo deciso di effettuare una serie di test al fine di indagare le capacità di risposta del server Orion sotto diverse condizioni di carico.

La prima serie di test è stata effettuata usando un solo server Orion, studiando i diversi tempi di risposta rispettivamente con uno, cinque, dieci, quindici e venti utenti connessi contemporaneamente. Questa prima fase di prove è stata realizzata per determinare il volume di carico critico per il server, cioè il numero di utenti oltre il quale il ritardo di risposta del server risultasse elevato (superiore ad 1-2 sec).

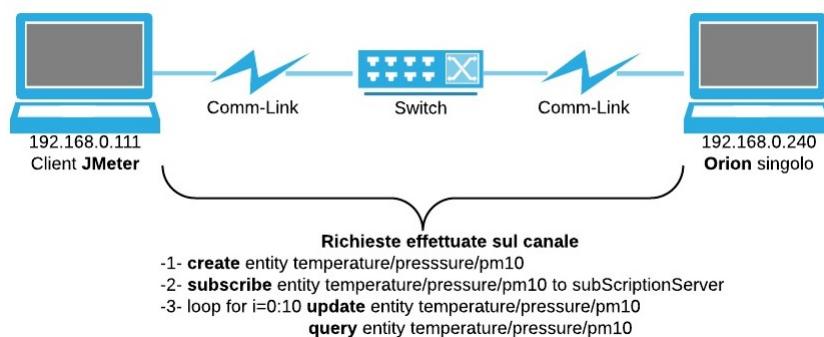


Figura 4.1: Prima fase di test

Individuato il valore di carico critico, abbiamo effettuato una seconda serie di prove con due Orion connessi tra di loro in federazione. Il server Orion infatti, mette a disposizione una particolare modalità, detta appunto federazione, attraverso la quale è possibile replicare le entità presenti sul server su un secondo server allo scopo di distribuire il carico delle query.

Nella seconda tipologia di test, il server Orion principale è stato usato per effettuare le operazioni di creazione, aggiornamento e sottoscrizione delle entità, mentre il secondo server è stato usato esclusivamente per servire le query degli utenti.

La seconda fase di test ha portato ad alcune interessanti considerazioni sul consumo di risorse da parte della macchina slave in questa particolare configurazione.

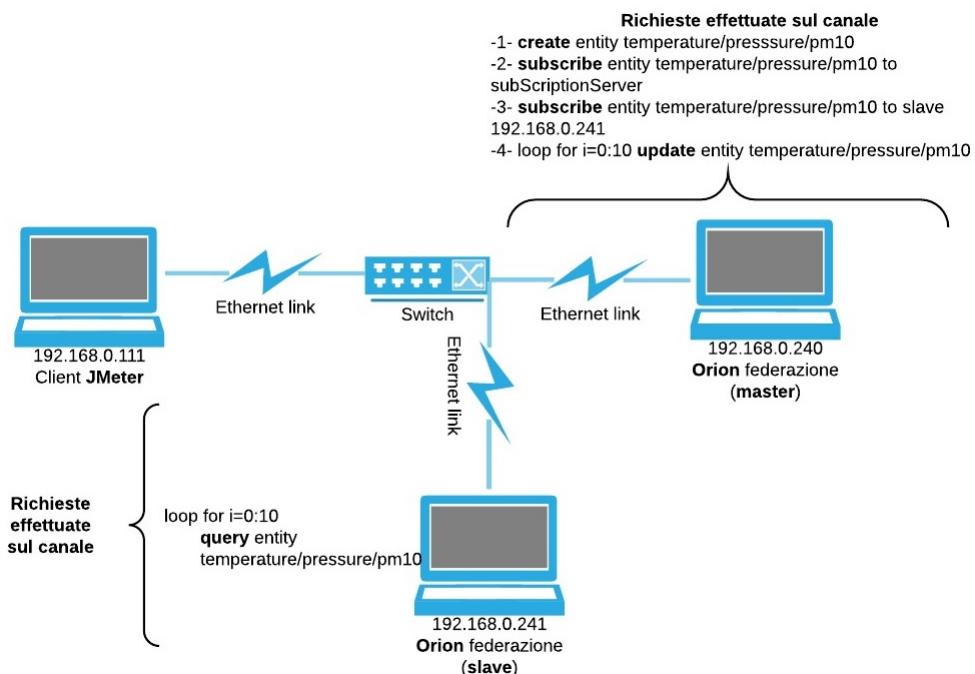


Figura 4.2: Seconda fase di test

Per realizzare i nostri test abbiamo utilizzato tre macchine virtuali Linux CentOS 6.6 con queste caratteristiche hardware:

- Server principale (**master**): 2.5 GB

- Server secondario (**slave**): 2.5 GB
- Macchina per simulare il comportamento dell'applicazione (**client**): 4 GB

4.2 JMeter

Per simulare il comportamento della nostra applicazione sotto diverse condizioni di carico, abbiamo utilizzato un software open source specifico per lo studio delle performance di nome JMeter.

JMter è ampiamente utilizzato per studiare le prestazioni di differenti protocolli e risorse; attraverso di esso è possibile effettuare test di prestazioni o di carico su un singolo server, gruppi di server o intere reti. JMeter supporta differnti tipi di server e protocolli tra cui:

- Web - HTTP, HTTPS
- SOAP
- FTP
- JDBC
- LDAP
- POP3, IMAP, SMTP
- MongoDB

JMeter è scritto in Java, può quindi essere eseguito su qualsiasi piattaforma dotata di JVM, inoltre mette a disposizione un'interfaccia grafica attraverso la quale è possibile modellare il proprio piano di test. Infine JMeter simula un gruppo di utenti che inviano richieste simultanee ad un server e ritorna statistiche che mostrano le performance di risposta attraverso tabelle grafici o semplici file.

4.3 Modellazione dell'applicazione con JMeter

Il primo passo da effettuare è stato quello di schematizzare il comportamento dell'applicazione di noi realizzata per poi riprodurlo su JMeter attraverso il piano di test.

Il comportamento dell'applicazione (lato sensoristica) può essere riassunto nei seguenti passi:

1. Creazione dell'entità temperatura
2. Creazione dell'entità pressione
3. Creazione dell'entità pm10
4. Sottoscrizione al subScribtionServer dell'entità temperatura
5. Sottoscrizione al subScribtionServer dell'entità pressione
6. Sottoscrizione al subScribtionServer dell'entità pm10
7. Loop (10 volte) .
 - (a) update entity temperature
 - (b) update entity pressure
 - (c) update entity pm10
 - (d) attendi dieci secondi

Il punto sette si riferisce ad una sessione di misurazione, il loop si conclude quando termina la sessione di raccolta dati. Per simulare una sessione di raccolta, abbiamo deciso di ripetere il processo di update (update entity temperature/pressure/pm10) per dieci volte, aspettando al termine di ogni iterazione dieci secondi per modellare l'intervallo di tempo che passa tra un campione ed il successivo della misurazione. Infine, un ulteriore passo può essere individuato nelle operazioni di query delle entità da parte degli utenti. Quest'ultimo però non fa parte delle richieste effettuate a livello di sensoristica; le query infatti sono svolte al livello più alto di interfaccia utente.

JMeter mette a disposizione tutta una serie di componenti grafici che possono essere usati per modellare il comportamento del proprio piano di test. L'elemento principale è il **Thread Group** al quale vengono collegati tutti gli altri componenti, inoltre attraverso di esso è possibile definire il numero di threads in parallelo da eseguire. Successivamente abbiamo i Controllers: **Samplers Controllers** e **Logical Controllers**. I Samplers Controllers specificano l'invio di una richiesta ad un server, per esempio un **HTTP Request Controller** permette di effettuare una richiesta HTTP. I Logical Controller permettono di definire la logica da usare per inviare le richieste; possiamo pensare a questi componenti come i costrutti *if,else*, *for, while* di un linguaggio di programmazione. Infine abbiamo i **Listeners** che permettono di accedere alle statistiche raccolte da JMeter durante i test; i risultati possono essere mostrati sotto forma di tabella, di albero o su un semplice file di testo. Per una completa trattazione sui componenti disponibili su JMeter si invita a consultare la documentazione sul sito del progetto¹.

I componenti che abbiamo dovuto utilizzare per modellare la successione di richieste sopra elencata sono i seguenti:

- Thread Group
- Once Only Controller
- Random Variable
- HTTP Request
- Loop Controller
- Simple data Writer

Nei paragrafi che seguono effettueremo una breve panoramica di questi componenti.

¹<http://jmeter.apache.org/usermanual/index.html>

4.3.1 Thread Group

Il Thread Group è il componente principale al quale vengono aggiunti tutti gli altri elementi (le diverse richieste). Il suo scopo, oltre ad essere il componente padre di tutto il test plan, è quello di definire il numero di threads da eseguire in parallelo. In breve, in una prima fase si modella il comportamento dell'applicazione aggiungendo al thread group le diverse richieste da eseguire in successione; questo blocco di richieste, modellanti il comportamento dell'applicazione, verranno eseguite da un singolo thread. E' quindi possibile simulare diverse istanze dell'applicazione attive sul server, aumentando il numero di thread da eseguire in parallelo attraverso il campo **Number of Threads**. E' anche possibile, attraverso il campo **Ramp-Up Period**, definire un intervallo di tempo entro il quale distribuire l'esecuzione dei vari threads. Spesso infatti l'arrivo simultaneo di tutti i threads, soprattutto se in numero elevato, potrebbe falsare le misurazioni dei campioni iniziali. Gli altri parametri sono abbastanza intuitivi: ad esempio il campo **Loop Count** permette di impostare il numero di ripetizioni successive per il test plan, alternativamente è possibile selezionare **Forever** per impostare un valore indefinito, in questo modo il piano di test dovrà essere terminato manualmente dall'utente. Un altro parametro utile è **Action to be taken after a Sampler error** che permette di definire l'azione da intraprendere in caso di errore.

Thread Group	
Name:	Thread Group
Comments:	
Action to be taken after a Sampler error	
<input checked="" type="radio"/> Continue <input type="radio"/> Start Next Thread Loop <input type="radio"/> Stop Thread <input type="radio"/> Stop Test	
Thread Properties	
Number of Threads (users):	1
Ramp-Up Period (in seconds):	0
Loop Count:	<input type="checkbox"/> Forever 1
<input type="checkbox"/> Delay Thread creation until needed	
<input type="checkbox"/> Scheduler	

Figura 4.3: Thread Group

Parameters	Attribute	Description	Required
Name	Descriptive name for this element that is shown in the tree.	Determines what happens if a sampler error occurs, either because the sample itself failed or an assertion failed. The possible choices are:	No
Action to be taken after a Sampler error		<ul style="list-style-type: none"> • Continue - ignore the error and continue with the test • Start Next Loop - ignore the error, start next loop and continue with the test • Stop Thread - current thread exits • Stop Test - the entire test is stopped at the end of any current samples • Stop Test Now - the entire test is stopped abruptly. Any current samplers are interrupted if possible. 	No
Number of Threads	Number of users to simulate.		Yes
Ramp-up Period		How long JMeter should take to get all the threads started. If there are 10 threads and a ramp-up time of 100 seconds, then each thread will begin 10 seconds after the previous thread started, for a total time of 100 seconds to get the test fully up to speed.	Yes
Loop Count		Number of times to perform the test case. Alternatively, "forever" can be selected causing the test to run until manually stopped.	Yes, unless forever is selected
Delay Thread creation until needed		If selected, threads are created only when the appropriate proportion of the ramp-up time has elapsed. This is most appropriate for tests with a ramp-up time that is significantly longer than the time to execute a single thread. I.e. where earlier threads finish before later ones start. If not selected, all threads are created when the test starts (they then pause for the appropriate proportion of the ramp-up time). This is the original default, and is appropriate for tests where threads are active throughout most of the test.	Yes
Scheduler		If selected, enables the scheduler If the scheduler checkbox is selected, one can choose an absolute start time. When you start your test, JMeter will wait until the specified start time to begin testing. Note: the Startup Delay field over-rides this - see below.	Yes
Start Time		If the scheduler checkbox is selected, one can choose an absolute end time. When you start your test, JMeter will wait until the specified start time to begin testing, and it will stop at the specified end time. Note: the Duration field over-rides this - see below.	No
End Time		If the scheduler checkbox is selected, one can choose a relative end time. JMeter will use this to calculate the End Time, and ignore the End Time value.	No
Duration (seconds)		If the scheduler checkbox is selected, one can choose a relative startup delay. JMeter will use this to calculate the Start Time, and ignore the Start Time value.	No
Startup delay (seconds)			No

Figura 4.4: Parametri configurazione Thread Group

4.3.2 Once Only Controller

Questo elemento è usato per inserire al suo interno altri componenti (ad esempio una richiesta HTTP) che devono essere eseguiti una sola volta nel flusso di esecuzione del test plan.

Il componente è stato utilizzato per inserire le richieste che vengono eseguite all'inizio del flusso di esecuzione, cioè le operazioni di creazione e sottoscrizione delle entità. Come si può vedere dalla figura, questo componente non ha nessun parametro di configurazione.



Figura 4.5: Once Only Controller

4.3.3 Random Variable

Questo componente è usato per generare dei numeri casuali durante l'esecuzione del test plan. Il campo **Variable Name** permette di definire il nome della variabile generata che verrà usata come reference all'interno degli altri componenti. È possibile definire il range di generazione attraverso i parametri **Minimum Value** e **Maximum Value**. È anche possibile, attraverso il campo **Per Thread (User)**, scegliere se utilizzare un solo generatore condiviso per tutti i thread in esecuzione, oppure è possibile definire il seme per l'inizializzazione della sequenza attraverso il campo **Seed for Random function**. Infine si può scegliere anche il formato di uscita del numero generato impostando il campo **Output Format**.

Il Random Variable è stato utilizzato per generare gli id univoci delle entità da creare, attraverso il Variable Name è infatti possibile fare riferimento al numero casuale generato da questo componente all'interno di altri componenti; per fare questo basta utilizzare il simbolo del dollaro prima del nome della variabile (impostato nel campo **Variabile Name**) in questo modo: \${IDTEMP}. Questo simbolo è stato quindi inserito all'interno di un http request per impostare l'id univoco dell'entità da creare.

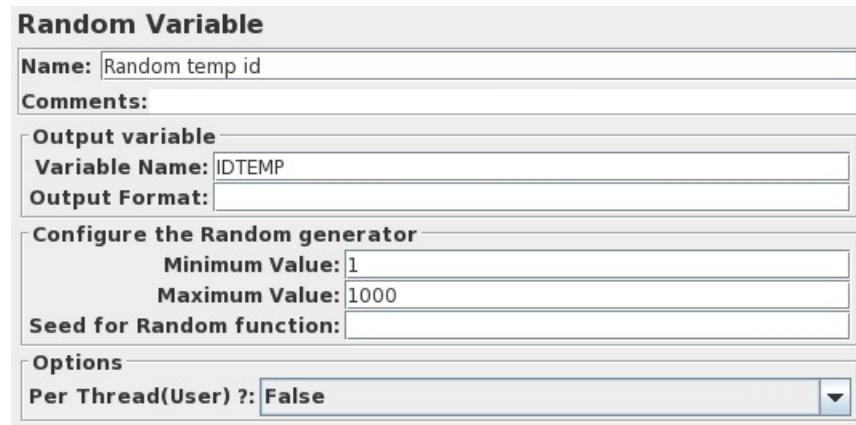


Figura 4.6: Random Variable

4.3.4 Loop Controller

Questo è il componente che permette di ripetere per un certo numero di volte una serie di operazioni. Il numero di iterazioni da effettuare viene definito attraverso il campo **Loop Count**.

Abbiamo utilizzato il Loop Controller per inserire al suo interno tutte le richieste di aggiornamento che devono essere eseguite più volte durante il flusso di esecuzione per simulare una sessione di raccolta dati da parte del livello di sensoristica e le richieste di query per simulare una sessione di ricerca da parte di un utente interessato ai dati.



Figura 4.7: Loop Controller

Parameters		Attribute	Description	Required
Name	Descriptive name for this element that is shown in the tree.	Name		Yes
Variable Name	The name of the variable in which to store the random string.	Variable Name		Yes
Format String	The Java text DecimalFormat string to be used. For example "000" which will generate numbers with at least 3 digits, or "USER_000" which will generate output of the form USER_mnn. If not specified, the default is to generate the number using Long.toString()	Format String		No
Minimum Value	The minimum value (long) of the generated random number.	Minimum Value		Yes
Maximum Value	The maximum value (long) of the generated random number.	Maximum Value		Yes
Random Seed	The seed for the random number generator. Default is the current time in milliseconds. If you use the same seed value with Per Thread set to true, you will get the same value for each Thread as per Random class.	Random Seed		No
Per Thread(User)?	If False, the generator is shared between all threads in the thread group. If True, then each thread has its own random generator.	Per Thread(User)?		Yes

Figura 4.8: Parametri configurazione Random Variable

4.3.5 HTTP Request

Questo componente permette di effettuare una richiesta http impostando il campo **Server Name or IP** e **Port Number**, il campo **Path** definisce il percorso della richiesta, il campo **Method** permette di scegliere il tipo di richiesta da effettuare (**POST GET PUT** etc.). Attraverso un tabbed pane è possibile impostare i parametri della richiesta ed il corpo del metodo. Nel nostro caso i parametri sono i tre di cui abbiamo ampiamente discusso nei paragrafi relativi alle **updateContext** e **queryContext** del capitolo due. Il corpo del metodo invece contiene il codice xml della richiesta (nel caso specifico una **updateContext**), da notare come il tag `<id>`, che contiene l'id dell'entità da creare, faccia riferimento attraverso il simbolo `${IDTEMP}` al numero casuale generato dal componente **Random Variable**.

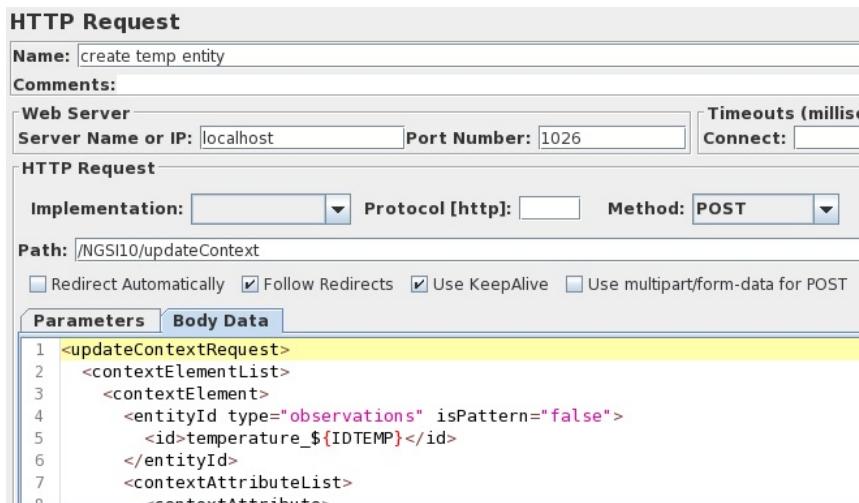


Figura 4.9: HTTP Request

Questo elemento è stato usato per effettuare tutte le richieste http al server Orion, di seguito i parametri di configurazione principali:

Parameters	Attribute	Description	Required
	Name	Descriptive name for this sampler that is shown in the tree.	No
Server		Domain name or IP address of the web server, e.g. www.example.com. [Do not include the http:// prefix.] Note: in JMeter 2.5 (and later) if the "Host" header is defined in a Header Manager, then this will be used as the virtual host name.	Yes, unless provided by HTTP Request Defaults
Port		Port the web server is listening to. Default: 80	No
Connect Timeout		Connection Timeout. Number of milliseconds to wait for a connection to open.	No
Response Timeout		Response Timeout. Number of milliseconds to wait for a response. Note that this applies to each wait for a response. If the server response is sent in several chunks, the overall elapsed time may be longer than the timeout. A Duration Assertion can be used to detect responses that take too long to complete.	No
Server (proxy)		Hostname or IP address of a proxy server to perform request. [Do not include the http:// prefix.]	No
Port		Port the proxy server is listening to.	No, unless proxy hostname is specified
Username		(Optional) username for proxy server.	No
Password		(Optional) password for proxy server. (N.B. this is stored unencrypted in the test plan)	No
Implementation	jmeter.httpsampler	Java: HttpClient3.1, HttpClient4: If not specified (and not defined by HTTP Request Defaults), the default depends on the value of the JMETER property jmeter.httpsampler . failing that, the HttpClient4 implementation is used.	No
Protocol		HTTP, HTTPS or FILE. Default: HTTP	No
Method		GET, POST, HEAD, TRACE, OPTIONS, PUT, DELETE, PATCH (not supported for JAVA implementation). With HttpClient4, the following methods related to WebDav are also allowed: COPY, LOCK, MKCOL, MOVE, PROPFIND, PROPPATCH, UNLOCK.	Yes

Figura 4.10: Parametri di configurazione HTTP Request

4.4 Prima serie di test

Come già anticipato, la prima serie di test che abbiamo effettuato è stata rivolta all'identificazione del numero di utenti critico per il sistema. Una volta modellata l'applicazione con i componenti visti nei paragrafi precedenti, abbiamo lanciato le simulazioni variando di volta in volta il numero di thread in parallelo attraverso il campo del componente Thread Group.

Il grafico sottostante mostra chiaramente che il caso (in verde) di venti utenti connessi contemporaneamente è il valore di soglia critico che cercavamo. Chiaramente questo valore dipende dalle risorse della macchina che ospita il server Orion e potrebbe variare usando sistemi con capacità hardware migliori.

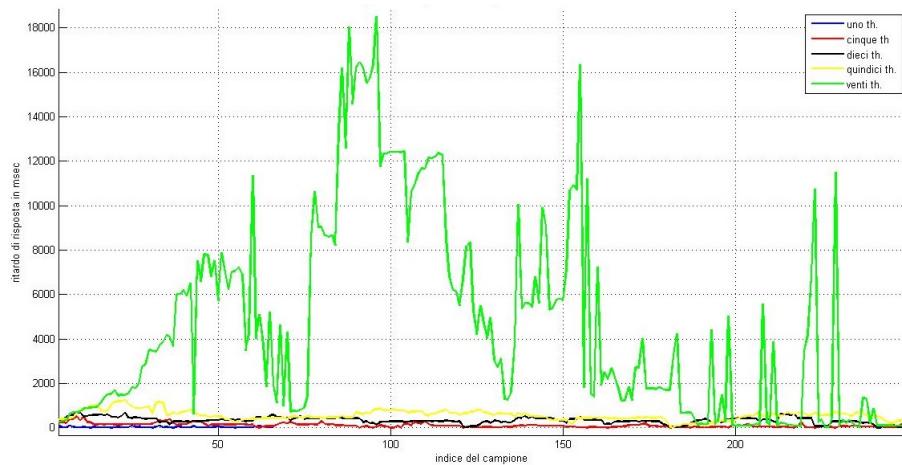


Figura 4.11: Performance Orion singolo 1-5-10-15-20 utenti

Il grafico sottostante è uno zoom dell'immagine precedente e mostra chiaramente che fino al caso di quindici utenti (giallo) i ritardi non salgono oltre il secondo e mezzo, ritardo che riteniamo accettabile per la nostra applicazione.

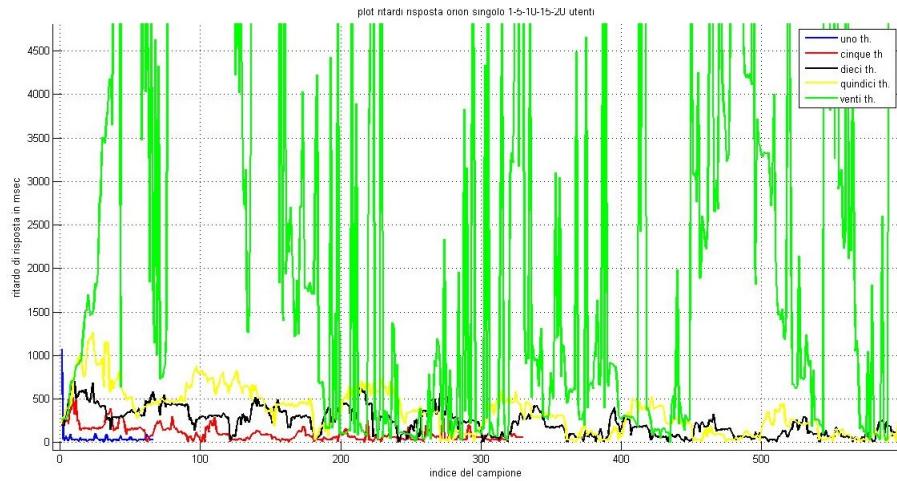


Figura 4.12: Zoom Orion singolo 1-5-10-15-20 utenti

4.5 Seconda serie di test

La seconda serie di prove sono state effettuate per studiare il comportamento del server Orion in modalità federata. La federazione è una particolare configurazione che permette di replicare le entità presenti su un server Orion su un altro server in modo da distribuire il carico tra le due macchine.

Nel nostro scenario potremmo pensare di utilizzare la federazione per distribuire le richieste tra due diversi server Orion nel caso critico di venti utenti. Abbiamo quindi effettuato una serie di prove per studiare le prestazioni in modalità federazione nel caso di uno, cinque, dieci e venti utenti.

Il grafico sottostante si riferisce al caso di un singolo utente, come era logico aspettarsi le prestazioni in modalità federazione (rosso) sono peggiori rispetto al caso di server singolo (blu). Questo comportamento è dovuto all'overhead introdotto dalla federazione; il server master per garantire la coerenza dei dati sul server slave deve inviare, ad ogni update ricevuta, delle richieste http (notifyContext) per aggiornare i dati anche sul secondo server; inoltre il server slave deve gestire queste richieste, aggiornare i propri dati e rispondere ad eventuali query sui dati aggiornati.

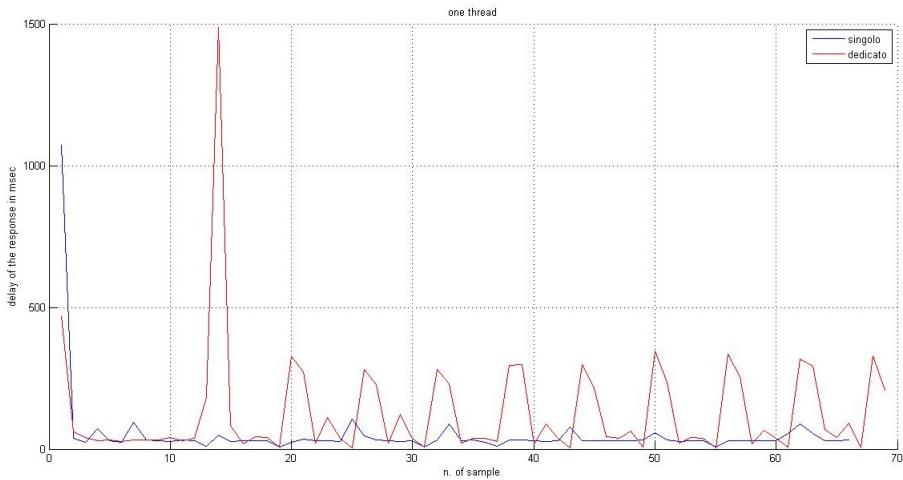


Figura 4.13: Federazione singolo thread

Il grafico di sotto si riferisce al caso di cinque utenti connessi contemporaneamente, anche in questo caso non si ha vantaggio nell'usare due server in federazione. Inoltre i ritardi di risposta delle diverse richieste risultano particolarmente elevati: possiamo notare facilmente picchi che raggiungono anche i trenta secondi. Ritardi di questa entità ci fanno pensare che molto probabilmente ci sia un problema di eccessivo consumo di memoria con conseguente swapping del processo su disco. Solo in questo modo si possono giustificare ritardi così elevati.

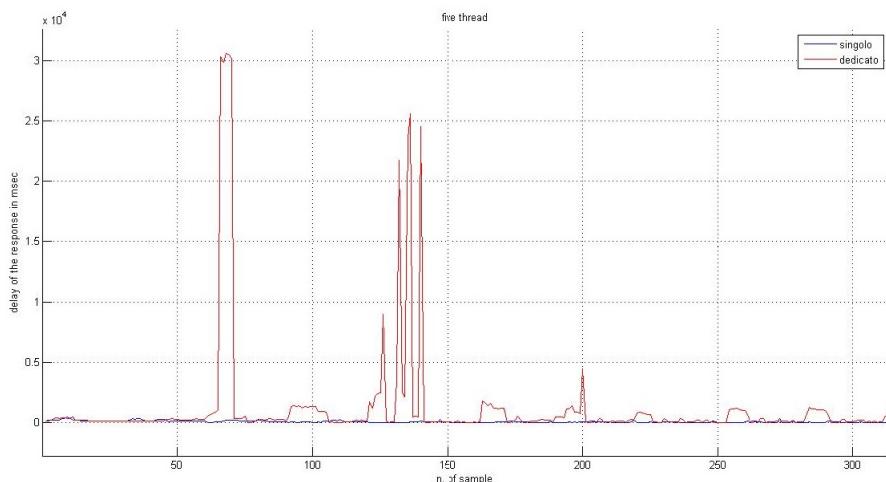


Figura 4.14: Federazione cinque threads

Anche il caso di dieci utenti presenta gli stessi problemi. Se le cose stesse in questi termini non si avrebbe mai vantaggio ad usare le federazione. Come detto sopra è evidente un problema di consumo eccessivo di risorse; andiamo quindi a studiare meglio il problema nei grafici successivi.

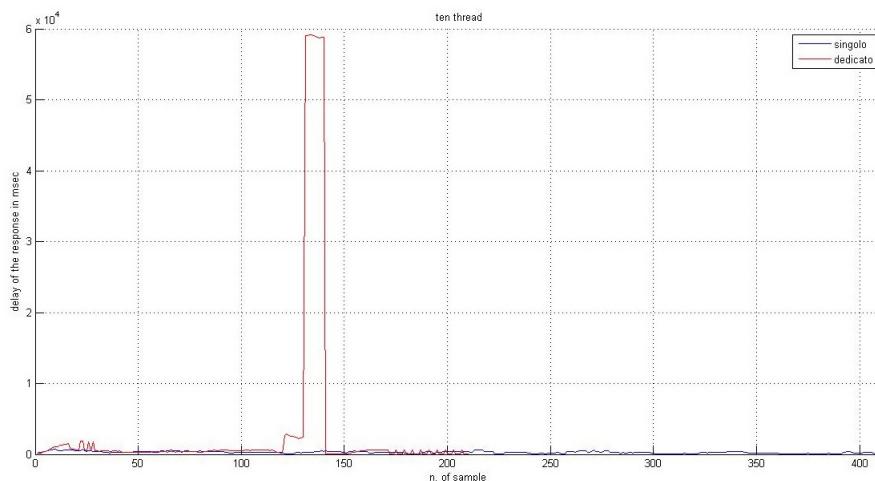


Figura 4.15: Federazione dieci threads

I grafici seguenti mostrano gli andamenti dei tempi di risposta nel caso di cinque e dieci utenti differenziando le singole richieste con colorazioni differenti. In questo modo siamo in grado di individuare le richieste che presentano i tempi di risposta più elevati. Di seguito un elenco del significato dei diversi colori usati per identificare le diverse tipologie di richieste.

- **Blu:** creazione entità
- **Nero:** sottoscrizione al subScribtionServer
- **Rosso:** aggiornamento entità
- **Verde:** query entità

Inoltre per chiarezza dobbiamo dire che sono state studiate due modalità di federazione: una detta **alternata** in cui il server master oltre ad occuparsi della creazione e aggiornamento delle entità risponde ad una porzione delle

query degli utenti; l'altra invece **dedicata** in cui il master si occupa esclusivamente delle create e delle update, mentre lo slave risponde alle query degli utenti.

La figura mostra chiaramente che la federazione soffre di ritardi elevati esclusivamente sulle query, cioè sulle richieste che dovrebbe gestire il secondo server in federazione (slave).

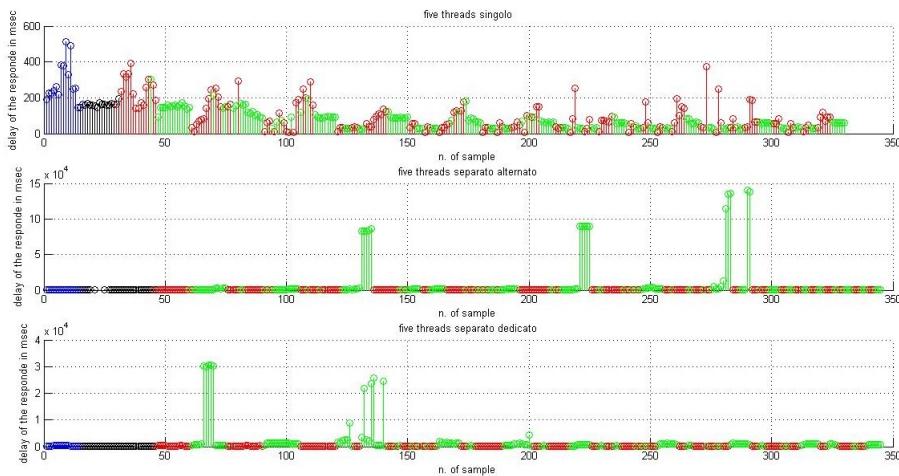


Figura 4.16: Federazione tutte le richieste cinque threads

Anche nel caso di dieci utenti i ritardi maggiori si hanno sulle richieste di query al server slave in federazione. Addirittura per certi campioni possiamo vedere che le richieste ritornano un messaggio di errore (crocetta al posto del pallino sopra il campione).

Quello che sta accadendo è che il secondo server in federazione consuma completamente la memoria disponibile e viene killato dal sistema operativo.

Avendo utilizzato due server con caratteristiche hardware simili possiamo dire che: mentre in modalità orion singolo i casi a cinque e dieci thread vengono gestiti senza eccessiva richiesta di memoria, nel caso di federazione con dieci thread si satura completamente la ram.

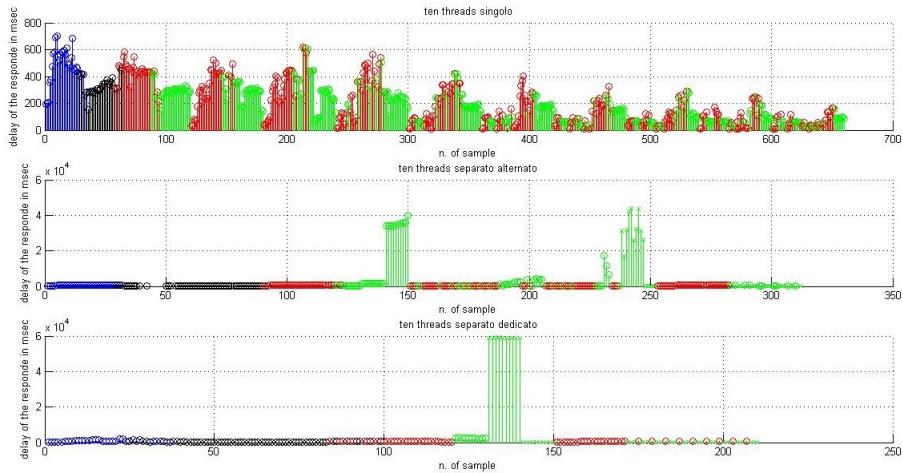


Figura 4.17: Federazione tutte le richieste dieci threads

Questo vuol dire solo una cosa: se si vuole usare l’orion in modalità di federazione è necessario che il secondo server (slave) abbia una quantità di memoria superiore rispetto al server principale; nel caso specifico quasi il doppio.

4.6 Seconda serie di test con più ram

Abbiamo quindi effettuato un’altra serie di test usando per il server centrale (master) sempre la stessa macchina mentre per il server slave abbiamo usato una macchina con il doppio della ram (4GB).

I grafici sottostanti si riferiscono alle prove per dieci e venti utenti. Il caso di dieci utenti, che precedentemente causava la saturazione completa della ram, sembra questa volta paragonabile al caso di Orion singolo. Ancora però non abbiamo vantaggio a federare. C’è anche da dire che nel caso di dieci utenti non avevamo il bisogno di federare in quanto, come mostrato nel primo grafico del capitolo, i risultati nel caso di orion singolo fino a quindici thread mostravano ritardi di risposta accettabili.

Il grafico nel caso di venti thread mostra invece dei tempi di risposta più bassi in modalità federazione, quindi solo in questo caso si ha un vantaggio effettivo nell’uso di due server piuttosto che di uno solo.

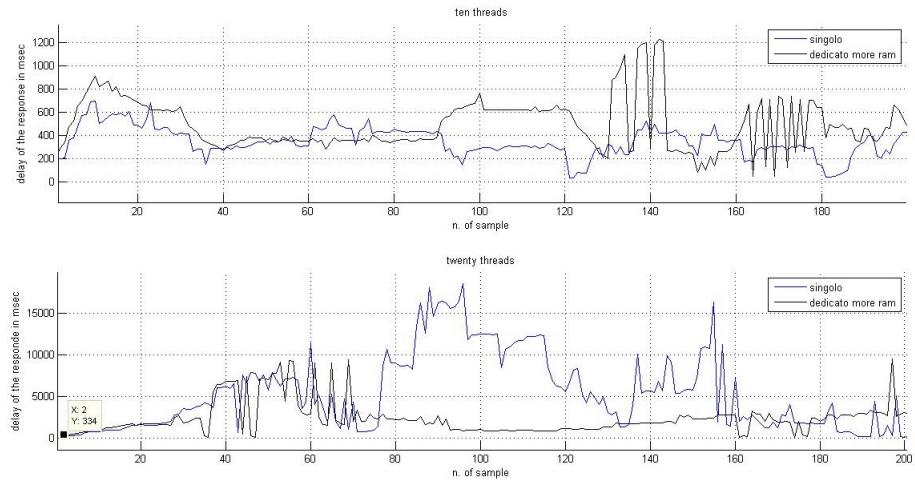


Figura 4.18: Federazione caso dieci e venti thread con più ram

Per concludere mostriamo il grafico con tutte le richieste differenziate per il caso di venti utenti sia in configurazione federazione che singolo. Il significato dei colori è lo stesso di quello visto al paragrafo 4.5 a pagina 137.

Come possiamo vedere ora il server master è libero dal carico delle query, svolte dal server slave, e tra i campioni ottanta e centoquaranta non abbiamo più i ritardi elevati oltre i quindici secondi.

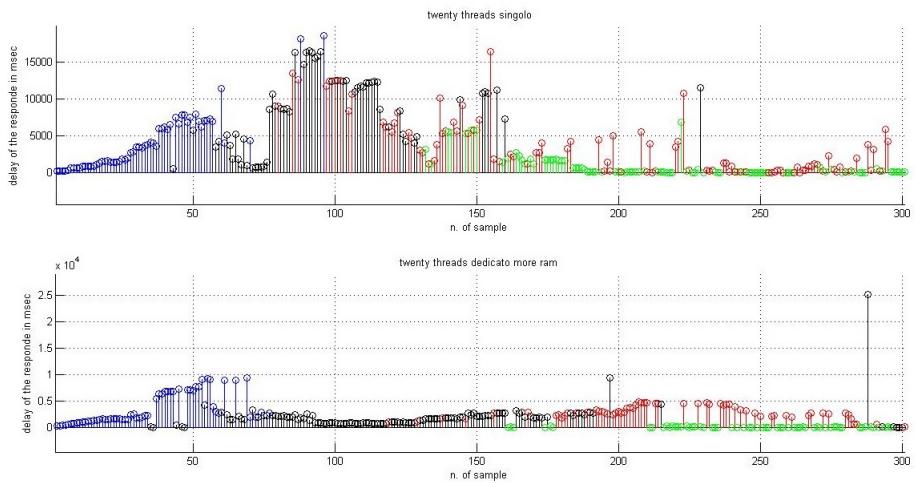


Figura 4.19: Federazione tutte le richieste 20 threads

4.7 Considerazioni finali sui test

Abbiamo prima studiato le prestazioni di un singolo server Orion nel caso di uno, cinque, dieci, quindici e venti threads. Abbiamo individuato il caso di venti utenti come critico in quanto presenta tempi di risposta eccessivamente elevati.

Abbiamo usato una seconda macchina con caratteristiche simili alla prima macchina sulla quale avevamo svolto i test per il caso di server singolo e abbiamo studiato il comportamento della federazione.

La federazione ha mostrato prestazioni inferiori sia per il caso di uno, cinque e dieci utenti. Addirittura nel caso di dieci utenti tutta la memoria della seconda macchina veniva completamente saturata causando il killing del server. Questo comportamento ci ha permesso di individuare il server slave della federazione come il collo di bottiglia del sistema.

Abbiamo quindi installato il server slave su una macchina con più memoria (4GB) e rieseguito i test per dieci e venti utenti. I risultati in questo caso hanno mostrato dei tempi di risposta inferiori sulle query per il caso federato, evidenziando quindi un effettivo vantaggio nell'uso di questa configurazione. In estrema sintesi possiamo dire che: finchè i tempi di risposta si mantengono accettabili (1-2 sec) non conviene federare, quando però questi crescono eccessivamente è possibile federare e ottenere dei tempi di risposta inferiori a patto però di usare una macchina per il server slave con un quantità di memoria sufficiente (nel nostro caso almeno 4 GB) e comunque superiore a quella che risulterebbe necessaria in configurazione orion singolo.

Appendice A

Fogli di stile

A.1 service_widget.css

```
1 * { margin: 0;
2         padding: 0;
3 }
4
5 #idContenitore{
6         font-family: Courier New;
7         font-size: 13px;
8         font-weight: bold ;
9         background: black ;
10        color : #00FF00 ;
11        margin: 10px ;
12        height: auto ;
13        border-radius: 10px 10px 10px 10px;
14 }
15
16 #tabellaContenitore {
17         font-family : Verdana ;
18         font-size: 13px ;
19         color : #464646;
20         margin: 10px ;
```

```
21     table-layout: fixed ;
22         width: 650px;
23         text-align: center ;
24 }
25
26 #servicesTable{
27     table-layout: fixed ;
28     width: 650px;
29     text-align: center ;
30     #border: 1px solid #98bf21 ;
31     #border-radius: 10px 10px 10px 10px ;
32 }
33
34 #servicesTable tr:hover {
35     background-color: #ffff99 ;
36     font-weight: bold ;
37 }
```

A.2 query_widget.css

```
1 #footer{
2     font-family: Courier New;
3     font-size: 13px;
4     font-weight: bold ;
5     background: black ;
6     color: #00FF00 ;
7     margin: 10px;
8     height: auto ;
9     border-radius: 10px 10px 10px 10px ;
10 }
11
12 #footer tr:hover {
13     color: white ;
14     font-weight: bold ;
```

```
15  }
16
17 #idContenitore {
18     font-family: Verdana;
19     font-size: 13px;
20     color: #464646;
21     margin: 10px;
22     table-layout: fixed ;
23     width: auto ;
24     height: auto ;
25     text-align: center ;
26 }
27
28 #idContenitore tr:hover {
29     background-color: #ffff99 ;
30     font-weight: bold ;
31 }
```

Appendice B

script matlab

B.1 plot_totale.m

```
1 totale_uno= './1thread_10loop/totale.csv';
2 totale_cinque= './5thread_10loop/totale.csv';
3 totale_dieci= './10thread_10loop/totale.csv';
4 totale_quindici= './15thread_10loop/totale.csv';
5 totale_venti= './20thread_10loop/totale.csv'
6 fid_uno = fopen(totale_uno);
7 fid_cinque = fopen(totale_cinque);
8 fid_dieci = fopen(totale_dieci);
9 fid_quindici = fopen(totale_quindici);
10 fid_venti = fopen(totale_venti);
11
12 uno = textscan(fid_uno , '%d%d%s%d%s%s%s%d%d' , '
    delimiter' , ' , ' );
13 cinque = textscan(fid_cinque , '%d%d%s%d%s%s%s%d%d%' 
    d' , 'delimiter' , ' , ' );
14 dieci = textscan(fid_dieci , '%d%d%s%d%s%s%s%d%d%' 
    , 'delimiter' , ' , ' );
15 quindici = textscan(fid_quindici , '%d%d%s%d%s%s%s%d%d%' 
    d%d' , 'delimiter' , ' , ' );
```

```
16 venti = textscan(fid_venti, '%d%d%s%d%s%s%d%d',  
17     'delimiter', ',' );  
18 fclose(fid_uno);  
19 fclose(fid_cinque);  
20 fclose(fid_dieci);  
21 fclose(fid_quindici);  
22 fclose(fid_venti);  
23  
24 timestamps_uno=uno{1};  
25 elapsed_uno=uno{2};  
26 label_uno=uno{3};  
27 responsecode_uno=uno{4};  
28 responsemessage_uno=uno{5};  
29 threadname_uno=uno{6};  
30 sucess_uno=uno{8};  
31  
32 elapsed_cinque=cinque{2};  
33 elapsed_dieci=dieci{2};  
34 elapsed_quindici=quindici{2};  
35 elapsed_venti=venti{2};  
36  
37 figure;  
38 hold on;  
39 grid on;  
40  
41 plot(elapsed_uno, 'b', 'LineWidth', 2); %blue  
42 plot(elapsed_cinque, 'r', 'LineWidth', 2); %red  
43 plot(elapsed_dieci, 'k', 'LineWidth', 2); %black  
44 plot(elapsed_quindici, 'y', 'LineWidth', 2); %yellow  
45 plot(elapsed_venti, 'g', 'LineWidth', 2); %green  
46  
47 legend('uno th.', 'cinque th.', 'dieci th.', 'quindici th.
```

```

    , 'venti th.') ;
48 title( 'plot ritardi risposta orion singolo
    1-5-10-15-20 utenti') ;
49 xlabel( 'indice del campione') ;
50 ylabel( 'ritardo di risposta in msec') ;

```

B.2 plot_mostra_le_diverse_richieste.m

```

1 %namefile = './1thread_10loop/totale.csv' ;
2 %namefile = './5thread_10loop/totale.csv' ;
3 %namefile = './10thread_10loop/totale.csv' ;
4 %namefile = './15thread_10loop/totale.csv' ;
5 %namefile = './20thread_10loop/totale.csv' ;
6 namefile = './20thread_10loop/totale.csv' ;
7 %namefile = './30thread_10loop/totale.csv' ;
8
9 fid = fopen(namefile) ;
10 csv = textscan(fid, '%d%d%s%d%s%s%s%d%d%d', 'delimiter', ', , ') ;
11 timestamps=csv{1} ;
12 elapsed=csv{2} ;
13 label=csv{3} ;
14 responsecode=csv{4} ;
15 responsemessage=csv{5} ;
16 threadname=csv{6} ;
17 success=csv{8} ;
18
19 figure ;
20 hold on ;
21 grid on ;
22
23 for i=1:size(timestamps)
24     tmp_success=success{i} ;
25     tmp_label=label{i} ;
26     if(strcmp(tmp_success, 'true'))

```

```
27     switch tmp_label
28         case 'createtempentity'
29             stem(i, elapsed(i), 'ob')
30         case 'createpressentity'
31             stem(i, elapsed(i), 'ob')
32         case 'createpm10entity'
33             stem(i, elapsed(i), 'ob')
34         case 'subscribetempentity'
35             stem(i, elapsed(i), 'ok')
36         case 'subscribepressentity'
37             stem(i, elapsed(i), 'ok')
38         case 'subscribepm10entity'
39             stem(i, elapsed(i), 'ok')
40         case 'updatetempvalue'
41             stem(i, elapsed(i), 'or')
42         case 'updatepressvalue'
43             stem(i, elapsed(i), 'or')
44         case 'updatepm10value'
45             stem(i, elapsed(i), 'or')
46         case 'querytemp'
47             stem(i, elapsed(i), 'og')
48         case 'querypress'
49             stem(i, elapsed(i), 'og')
50         case 'querypm10'
51             stem(i, elapsed(i), 'og')
52     end
53 else
54     switch tmp_label
55         case 'createtempentity'
56             stem(i, elapsed(i), 'xb')
57         case 'createpressentity'
58             stem(i, elapsed(i), 'xb')
59         case 'createpm10entity'
```

```
60                      stem(i , elapsed( i ), 'xb ')
61      case 'subscribetempentity'
62                      stem(i , elapsed( i ), 'xk ')
63      case 'subscribepressentity'
64                      stem(i , elapsed( i ), 'xk ')
65      case 'subscribepm10entity'
66                      stem(i , elapsed( i ), 'xk ')
67      case 'updatetempvalue'
68                      stem(i , elapsed( i ), 'xr ')
69      case 'updatepressvalue'
70                      stem(i , elapsed( i ), 'xr ')
71      case 'updatepm10value'
72                      stem(i , elapsed( i ), 'xr ')
73      case 'querytemp'
74                      stem(i , elapsed( i ), 'xg ')
75      case 'querypress'
76                      stem(i , elapsed( i ), 'xg ')
77      case 'querypm10'
78                      stem(i , elapsed( i ), 'xg ')
79      end
80  end
81 end
82
83 title( 'plot_singole_richieste_orion_singolo_20_utenti' );
84 xlabel( 'indice del campione' );
85 ylabel( 'ritardo di risposta in msec' );
```

Bibliografia

- [1] M. Budde, R. E. Masri, T. Riedel, M. Beigl (2013), “*Enabling Low-Cost Particulate Matter Measurement for Participatory Sensing Scenarios*”. In 12th International Conference on Mobile and Ubiquitous Multimedia (MUM 2013).
- [2] M. Budde, M. Busse, M. Beigl (2012), “*Investigating The Use of Commodity Dust Sensors for Embedded Measurement of Particulate Matter*”. In Networked Sensing Systems (INSS), 2012 Ninth International Conference on.
- [3] D.M. Holstius, A. Pillarisetti, K.R. Smith, E. Seto (2014), “*Fileld Calibration of Low-Cost Aerosol sensor at a Regulatory Monitoring Site in California*”. In Atmospheric Measurement Techniques (AMT) 2014.
- [4] Open Mobile Alliance (2012), “*NGSI Context Management*”. In Open Mobile Alliance 2012.
- [5] S. Ghemawat, H. Gobioff, S. Leung (2003), “*The Google File System*”. In 19th ACM Symposium on Operating System Principles, October 2003.
- [6] J. Dean, S. Ghemawat (2004), “*MapReduce: Simplified Data Processing on Large Clusters*”. In OSDI’04: Sixth Symposium on Operating System Design and Implementation, December 2004.
- [7] T. White (2012), “*Hadoop: The Definitive Guide 3rd Edition*”. O'Reilly Media 2012.

- [8] D. Flanagan (2011), “*Javascript: The Definitive Guide 6th Edition*”. O'Reilly Media 2011.
- [9] B. Rhodes, J. Goerzen (2014), “*Foundations of Python Network Programming 3rd Edition*”. Apress 2014.
- [10] S. Aggarwal (2014), “*Flask Framework Cookbook*”. Packt Publishing 2014.
- [11] Fi-ware wiki (2015), “*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”](https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-Publish/Subscribe_Broker_-_Orion_Context_Broker_-_User_and_Programmers_Guide)
- [12] Fi-ware wiki (2015), “*Orion Context Broker - Installation and Administration Guide*”, [“https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-Publish/Subscribe_Broker_-_Orion_Context_Broker_-_Installation_and_Administration_Guide”](https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-Publish/Subscribe_Broker_-_Orion_Context_Broker_-_Installation_and_Administration_Guide)
- [13] Fi-ware wiki (2015), “*Fi-ware Architecture Description Big Data*”, [“https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-FIWARE.ArchitectureDescription.Data.BigData”](https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-FIWARE.ArchitectureDescription.Data.BigData)
- [14] Fi-ware wiki (2015), “*Big Data Analysis - User and Programmer Guide*”, [“http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-BigData_Analysis_-_User_and_Programmer_Guide”](http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-BigData_Analysis_-_User_and_Programmer_Guide)
- [15] Fi-ware wiki (2015), “*Big Data Analysis - Quick Start for Programmers*”, [“http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-BigData_Analysis_-_Quick_Start_for_Programmers”](http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-BigData_Analysis_-_Quick_Start_for_Programmers)
- [16] Fi-ware wiki (2015), “*Application Mashup - Wirecloud - User and Programmer Guide*”, [“http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-Wirecloud_-_User_and_Programmer_Guide”](http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-Wirecloud_-_User_and_Programmer_Guide)

- https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/-Application_Mashup_-_Wirecloud_-_User_and_Programmer_Guide
- [17] Conwet Web Site (2015), “*Developer’s Guide*”,
<http://conwet.fi.upm.es/wirecloud/developer>
- [18] CKAN Web Site (2015), “*API Guide*”,
<http://docs.ckan.org/en/latest/api/index.html>
- [19] CKAN Web Site (2015), “*User guide*”,
<http://docs.ckan.org/en/latest/user-guide.html>
- [20] JMeter Web Site (2015), “*JMeter User’s Manual*”,
<http://jmeter.apache.org/usermanual/index.html>