ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Dipartimento di Informatica - Scienza e Ingegneria DISI

TESI DI LAUREA

in

Fondamenti di Informatica T-2

Progetto e sviluppo di un middleware per l'interfacciamento di sensori per la domotica in ambiente Java

CANDIDATO
Federico Baldassarre

RELATORE
Chiar.mo Prof. Enrico Denti

CORRELATORE
Dott.ssa Roberta Calegari

Anno Accademico 2015/16

Sessione I

Oggi siamo sul tetto del mondo,
domani scopriremo un palazzo più alto
e vorremo arrivarci,
ma oggi siamo sul tetto del mondo

Un amico

Prefazione

Il presente progetto è stato realizzato come tesi di laurea del corso di Ingegneria Informatica presso l'Università di Bologna. Esso si inserisce nel contesto di ricerca svolto dal laboratorio APICe, integrandosi in particolare con i progetti tuProlog e HomeManager.

Ringraziamenti

In questa giornata si conclude la mia prima esperienza con il mondo universitario e, per estensione, della vita adulta. Durante questi anni non sono maturato solamente dal punto di vista accademico, ma anche e soprattutto personale. Perché se un libro o un corso online possono darti conoscenza, solo l'Università può darti le *persone*. Ed è dunque importante per me ringraziare tutte quelle persone che hanno condiviso con me piccoli e grandi ritagli di questo percorso.

Comincio ringraziando il professor Enrico Denti e la Dottoressa Roberta Calegari, i "dottori" dietro questa tesi. Loro hanno visto in me una potenzialità e hanno chiesto che la sfruttassi, concedendo grande fiducia, libertà e pazienza. Insieme a loro ringrazio anche tutti i professori che in questi anni hanno cercato di dare un volto umano alla conoscenza, trasmettendo valori oltre che conoscenza.

Ringrazio poi di cuore il mio collega e amico Ventu, perché come tanto volte gli ho ripetuto, le persone da sole possono essere dei draghi, ma è solo con la condivisione di sforzi e obiettivi che possiamo tirare fuori il meglio di noi. E affermo ancora una volta che qui oggi ci siamo arrivati *insieme*.

Grazie a Chiara per l'amore, il sostegno e i consigli di questi anni, per tutte le piccole cose viste e vissute insieme, per i momenti facili e quelli difficili.

Questa tesi parla tanto di domotica e di case intelligenti, ma ciò che rende un'abitazione una vera casa sono le persone che ci vivono giorno dopo giorno. Per questo ringrazio infinitamente la mia famiglia, vicini o lontani che siano. Mamma, papà e fratello perché da vicino hanno supportato i miei studi e "sopportato" ogni vaschetta di gelato conquistata. Nonne e nonni, perché, chi da lontano e chi da ancora più lontano, mi hanno sempre fatto sentire il loro orgoglio e il loro amore.

Protagonisti veri della mia crescita sono poi tutti gli amici trovati in questi anni. Grazie agli *Amici Stravaganti* per la vostra gioia, la compagnia e le esperienze vissute insieme. Grazie *Ciao Cari* perché dimostrate che amici è più che compagni di corso.

Un pensiero finale va poi a tutte quelle persone senza volto che inconsapevolmente hanno contribuito a questa tesi con la loro goccia di esperienza, quindi grazie StackOverflow, grazie Internet.

Federico

Contenuti

Indice dei capitoli

Introduzione

La Domotica è una scienza che punta a migliorare la qualità della vita nella casa e più in generale negli ambienti antropizzati. In questo contesto assume grande rilevanza la possibilità di comunicare tra oggetti eterogenei in grado a loro volta di interagire con l'ambiente. Risulta quindi fondamentale l'utilizzo di una tecnologia abilitante che permetta la comunicazione tra gli agenti.

L'obiettivo finale del progetto di tesi è realizzare un *middleware* per sensori distribuiti Java-based chiamato SensorNetwork, che permetta ad un agente domotico di effettuare *sensing* sull'ambiente. Le funzionalità principali del sistema sono:

- Uniformità di accesso a sensori eterogenei distribuiti
- Semplicità di utilizzo ed estensione con nuovi sensori
- Alto livello di automazione del sistema, con caratteristiche di avvio automatico dei nodi e auto discovery dei nuovi sensori
- Ampia libertà di configurazione
- Modularità e sviluppo a componenti, per facilitare l'introduzione di modifiche

Il sistema realizzato è basato su un'architettura a componente-container che permette l'utilizzo di *sensori* all'interno di *stazioni di sensori* e che supporti l'accesso remoto per mezzo di un servizio di *naming* definito *ad-hoc.*

L'esposizione è così strutturata:

- Nel primo capitolo viene descritto il contesto domotico, analizzando il problema della comunicazione tra agenti eterogenei e distribuiti
- Nel secondo capitolo si progetta una soluzione basata sul modello componente-container in grado di rispondere alle esigenze presentate nel capitolo precedente, analizzando nel dettaglio le funzionalità richieste per il sistema dei sensori, delle stazioni e per il servizio di naming
- Nel terzo capitolo si entra nel dettaglio della realizzazione del sistema, con un'introduzione sugli strumenti utilizzati, a cui segue una definizione completa dei concetti di Sensor, Station e Provider, concludendosi poi con l'implementazione di tre sensori come *proof of concept*
- Nel quarto capitolo viene illustrata la realizzazione completa di un Provider basato sulla tecnologia Java RMI, con una descrizione esaustiva delle varie problematiche incontrate e delle tecniche utilizzate per risolverle, un breve esempio ripercorre in modo completo i passi necessari al *deployment* del sistema, concludendo poi con una digressione sulle tecnologie alternative per la realizzazione del Provider
- Nel quinto capitolo si espongono le riflessioni finali sul progetto realizzato, presentando e suggerendo possibili sviluppi per migliorare ed estendere il sistema

1 Analisi del problema

1.1 Smart Home Systems e Internet of Things

La Domotica è una scienza interdisciplinare che si occupa dello studio delle tecnologie atte a migliorare la qualità della vita nella casa e più in generale negli ambienti antropizzati, realizzando case intelligenti, confortevoli, sicure e di semplice fruizione.

In questo contesto vengono utilizzati anche i termini *Smart Home Systems* e *Building Automation*, perché gli impianti realizzati presentano come caratteristica principale un'elevata capacità di apprendimento delle abitudini dell'utente e di adattamento ai suoi bisogni[1].

È fondamentale per un agente domotico poter accedere all'ambiente circostante per misurarlo e agire su di esso. Solo così un maggiordomo virtuale *context-aware* può sostituire le funzioni svolte da un major domus fisico, monitorando lo stato di un'abitazione e le attività dei suoi abitanti, modificando la temperatura degli ambienti, il livello di umidità e di illuminazione, eccetera.
Da questa necessità di operare anche su piccoli oggetti e dispositivi, risulta evidente il forte legame che unisce la domotica ai concetti di *Internet of Things* e *pervasive computing*.

Il termine *Internet of Things*[2] fa riferimento alla tendenza di rendere interconnesso ogni oggetto della vita quotidiana. Se vent'anni fa l'accesso alla rete era tipicamente limitato a personal computer e sistemi informatici di medie-grandi dimensioni, oggi assistiamo all'avvento di dispositivi connessi di più disparata natura: orologi, tablet, semafori, robot da cucina, autoveicoli e navigatori satellitari.

Le entità facenti parte della rete dell'IoT sono spesso chiamate *smart objects* e a differenza dei normali dispositivi possiedono un ruolo attivo nel sistema di comunicazione in cui sono inseriti, in grado in generale di effettuare *sensing* e attuazione sull'ambiente.

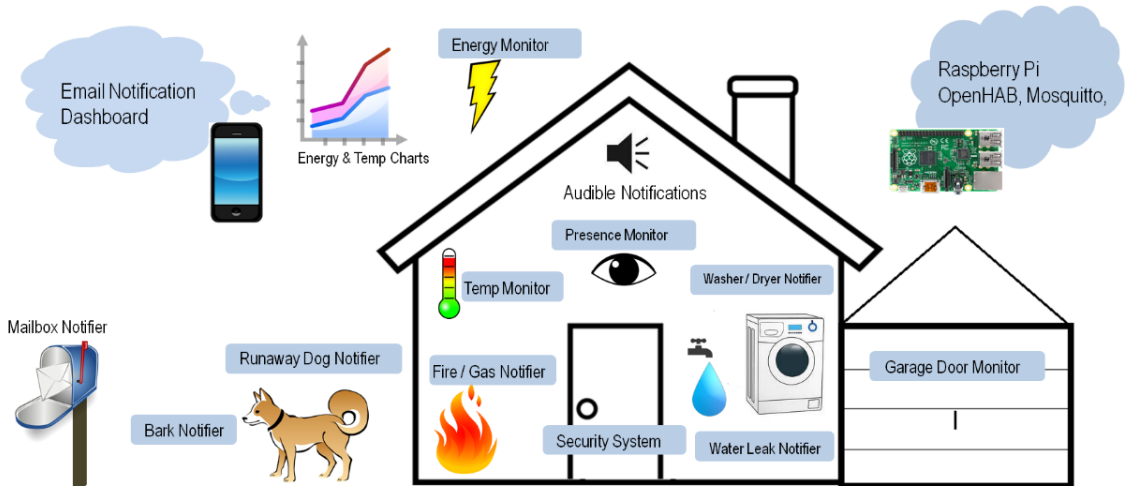


Figura 1 Numerosi ambiti domestici in cui è possibile integrare smart objects

È in particolare sull'attività di *sensing*, ovvero la misura dell'ambiente, che si concentra il *focus* di questa tesi. Al *sensing* è associata la problematica della modalità di accesso a sensori distribuiti.

1.2 Descrizione del problema

La componente software di un sistema domotico può essere realizzata tramite svariati approcci tecnologici. Tra questi uno che dimostra numerosi vantaggi è quello basato sull'integrazione di

[1] https://en.wikipedia.org/wiki/Home_automation
[2] https://en.wikipedia.org/wiki/Internet_of_things

linguaggi orientati agli oggetti e linguaggi di programmazione logica *rule-based*. Dalla collaborazione di questi due paradigmi è possibile realizzare sistemi complessi nei quali la gestione dei sensori è affidata al primo, mentre il secondo è deputato a utilizzare le informazioni ricavate per abilitare ragionamenti e politiche di ordine superiore. A titolo di esempio si consideri il sistema HomeManager[3], realizzato dall'Università di Bologna e basato su linguaggio Prolog.

Indipendentemente dal paradigma di alto livello scelto, l'automatizzazione di una abitazione è fortemente legata alla capacità di pilotare una vasta rete di componenti elettronici e meccanici. In questo contesto il focus è posto di conseguenza a ben più basso livello, occupandosi della realizzazione fisica dei sensori, del sistema di controllo e della tecnologia di comunicazione. Particolare attenzione va posta in proprio quest'ultima, in quanto è necessario abilitare i nodi della rete all'accesso remoto.

La concretizzazione di un ambiente domotico è quindi resa possibile da tecnologie abilitanti come le Wireless Sensor Network[4]. I nodi di una WSN sono i sensori, disposti all'interno di un ambiente, con lo scopo di rilevare determinati dati e inviarli per esempio nodo con capacità di elaborazione superiore.

La realizzazione di reti di sensori e attuatori può essere delegata a produttori specializzati, tuttavia le infrastrutture da esse proposte risultano tipicamente costose per l'utente finale e rigide in quanto difficile incorporare tecnologie di altri produttori. Per contro, gli utenti che volessero realizzare sistemi non proprietari devono affrontare il difficile compito di assemblare componenti non compatibili. Il che richiede un significativo livello di conoscenza e sforzo tecnico, rischiando di sfociare in un sistema più costoso e peggio realizzato di una soluzione proprietaria[5].

Questa tesi intende ovviare a tali problematiche, realizzando un *framewok* che renda semplice la progettazione, l'implementazione e l'accesso remoto a sensori distribuiti.

1.3 Middleware per sensori distribuiti

Dal paragrafo precedente si evidenziano una serie di caratteristiche che devono essere prese in considerazione nella realizzazione del sistema. Di seguito vengono meglio specificate ed esemplificate.

L'hardware dei sensori presenta specificità delle quali si deve tenere conto in maniera locale al sensore, ma vanno necessariamente nascoste all'utente finale. Ad esempio, per ottenere una misura di distanza tramite un sensore ad ultrasuoni può essere necessario inviare al dispositivo un segnale di una determinata forma, ma per un utilizzatore esterno ciò che conta è il risultato ottenuto e non come sia stata effettuata la misura. Si rende dunque necessario astrarre dalle diverse tipologie di sensori, uniformando l'accesso e l'utilizzo. Non è invece possibile ignorare l'eterogeneità dei sensori e i loro bisogni specifici per quanto riguarda la loro configurazione locale.

Una rete di sensori è naturalmente soggetta a grande dinamicità: i nodi che la compongono potrebbero perdere la connessione o essere disabilitati e nuovi elementi potrebbero essere installati per aumentare le funzionalità. Un produttore di soluzioni per l'*home automation,* potrebbe infatti decidere di progettare nuovi dispositivi ed integrarli nelle installazioni preesistenti. Così come un cliente potrebbe acquistare dapprima una versione ridotta del sistema e se

[3] http://apice.unibo.it/xwiki/bin/view/Products/HomeManager
[4] https://en.wikipedia.org/wiki/Wireless_sensor_network
[5] Jarrod, T., Ian, A., & Gilles, G. (2002). *Sensor Abstraction Layer.* SP&E

soddisfatto aggiungere successivamente nuovi elementi. Si vuole dunque che la realizzazione e l'aggiunta di nuovi componenti alla rete sia la più semplice possibile.

In un sistema complesso, costituito da molti nodi, è consigliabile ridurre le operazioni necessarie all'utente preferendo invece un'automatizzazione delle stesse. Introducendo una serie di meccanismi di supporto, l'utente non è costretto a riavviare e riconfigurare ogni nodo della rete a seguito ad esempio di un'interruzione di corrente, ma la rete è in grado di ristabilirsi in maniera quasi completamente autonoma.

In definitiva, questo progetto consiste nella realizzazione di una architettura a *middleware* in grado di offrire una visione uniforme di reti di sensori eterogenei, indipendentemente dalle tecnologie sottostanti.

Si vuole astrarre e nascondere le specificità hardware relative all'accesso, al controllo e all'utilizzo di sensori ed attuatori, offrendo al livello soprastante un'interfaccia stabile e indipendente con funzionalità di gestione remota della rete.

Il risultato finale vuole essere un sistema flessibile, semplice dall'esterno e facilmente estensibile, in grado di alleggerire i linguaggi di programmazione di alto livello utilizzati nei sistemi domotici dal carico di gestire i dettagli specifici dei sensori permettendo comunque una grande libertà di configurazione ed utilizzo.

2 Progettazione del sistema

In questo capitolo si presentano le diverse funzionalità che il sistema deve offrire, definendo più nel dettaglio quanto illustrato al termine del capitolo precedente. Da esse conseguono scelte progettuali che hanno orientato l'effettiva implementazione della rete di sensori verso un'architettura a componente-container che offra facilità di estensione con nuovi sensori, configurazione del sistema a *deployment* time e indipendenza dal meccanismo di utilizzo remoto dei sensori.

2.1 Funzionalità dei sensori e della station

Ad un alto livello di astrazione si può pensare ad un sensore come un'entità in grado di effettuare sull'ambiente circostante interagendo con esso o reperire informazioni da una qualsiasi fonte. Sull'esecuzione di misure è opportuno distinguere la descrizione dell'operazione vista dall'esterno dall'effettivo meccanismo di misura usato dal sensore.

Volendo trarre un esempio dalla vita reale: per misurare la temperatura corporea sono disponibili in commercio moltissimi modelli, eterogenei per principio di funzionamento, tuttavia ognuno di essi offre la medesima operazione di misura. (Figura 2)

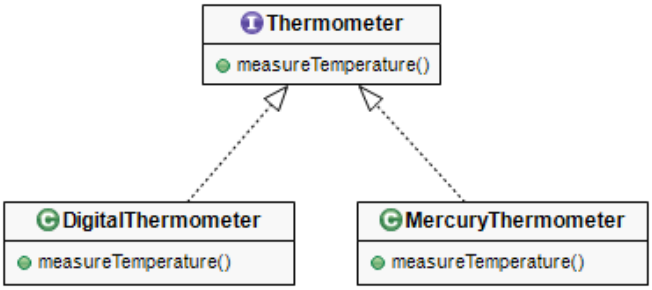


Figura 2 Esempio di ereditarietà tra termometri

In ambiente *object-oriented* si utilizza il concetto di interfaccia per descrivere un'operazione pubblica offerta da diverse entità concrete, sebbene con implementazione diversa[6]. All'insegna di questa astrazione si vuole poter utilizzare tutti i sensori dello stesso tipo in modo uniforme, indipendentemente dall'hardware del sensore vero e proprio e dalla sua logica di controllo.
Sarà dunque necessario separare la gerarchia di interfacce dalla gerarchia di delle implementazioni. Tramite la prima si potranno esprimere le operazioni pubbliche di una certa famiglia di sensori, in modo uniforme e indipendente dal sensore. Tramite la seconda sarà possibile riutilizzare la parte di codice comune ai diversi sensori.

Un sensore inoltre è tipicamente dotato di uno stato che ne descrive sinteticamente la condizione interna. Sarà dunque necessario prevedere un supporto alla gestione dello stato dei sensori, alle transizioni tra i diversi stati e alla notifica di tale transizione ad altre entità. Tale supporto può essere realizzato da una classe base ed ereditato da tutte le implementazioni dei sensori in modo da evitare riscrittura di codice.
Semplificando al massimo la descrizione degli stati, si identificano tre stati che descrivono il sensore in condizione operativa normale, il sensore non avviato e il sensore in stato di errore. Dall'esterno, per un qualsiasi sensore, deve essere possibile leggere lo stato.
In questo modo abbiamo individuato due aspetti della gestione dello stato. Il primo riguardante la logica di gestione dello stato confluirà nella gerarchia delle implementazioni dei sensori, mentre il secondo inerente alla vista dall'esterno dello stato confluirà nella gerarchia delle interfacce (Figura 4 e Figura 4).

[6] https://en.wikipedia.org/wiki/Protocol_(object-oriented_programming)

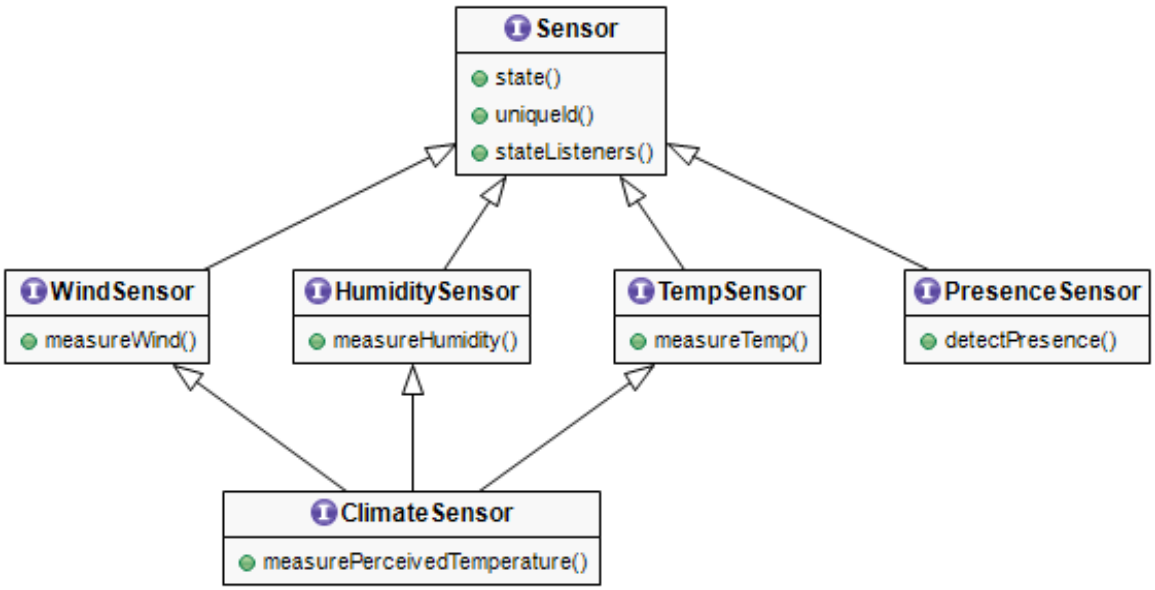


Figura 3 Gerarchia di interfacce per i sensori

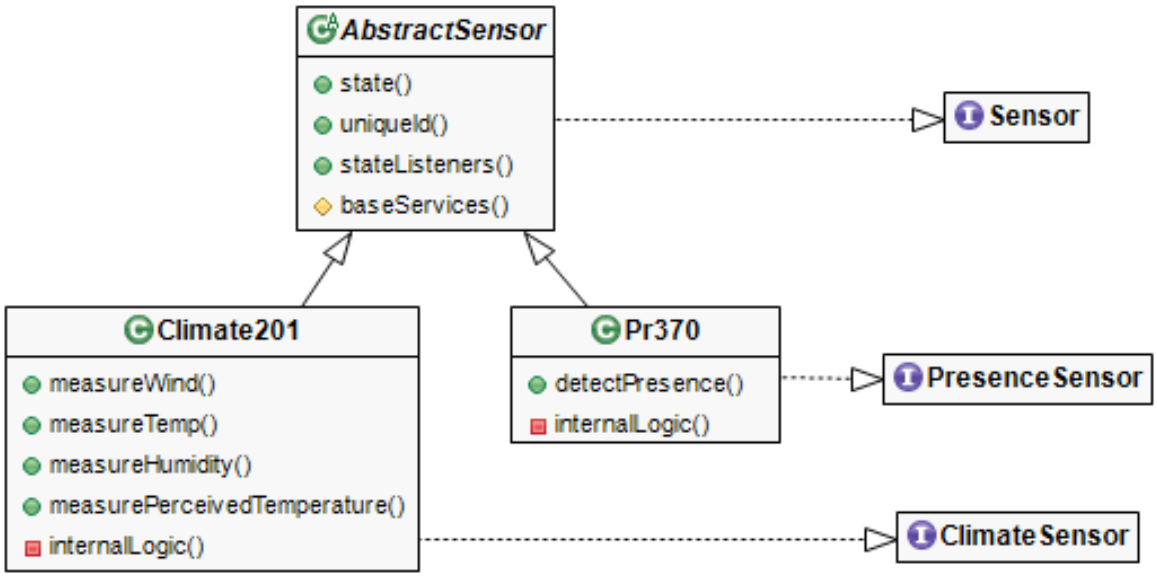


Figura 4 Gerarchia di implementazioni per i sensori

La logica di controllo di uno specifico sensore è strettamente dipendente dal tipo di hardware sul quale il sensore è collocato ed anche dal tipo di collegamento con cui è allacciato. Ad esempio un certo modello di sensore di temperatura potrebbe essere montato su un Raspberry Pi ai pin di GPIO 3 e 4, ma un domani potrebbe essere necessario ricollegarlo ai pin 7 e 8. Non si desidera tuttavia che piccole variazioni di questo tipo obblighino la revisione del codice e la ricompilazione della classe del sensore. Certi parametri devono poter essere configurati con grande semplicità al momento del *deployment*. Si desidera inoltre che i sensori siano in grado di attivarsi autonomamente senza che sia necessario l'intervento dell'utente. Ci si preoccupa quindi di realizzare un semplice meccanismo di *dependency injection* in grado di ottenere certi parametri da descrittori di configurazione e fornirli ai sensori a *runtime*.

Un ulteriore aspetto da prendere in considerazione nel progettare una rete di sensori distribuiti è con quali modalità renderli disponibili ai clienti, in particolare se gestire la concorrenza degli accessi e il caching dei risultati per migliorare l'efficienza del sistema. La decisione presa in questi due casi

è quella di lasciare che le implementazioni dei sensori gestiscano autonomamente il problema nel modo per loro più corretto.
Infatti, visto il livello di astrazione molto elevato sui sensori, non è pensabile prendere una decisione che risulti valida per qualsiasi sensore. Ci sono sensori per i quali il risultato ha significato solo nell'istante del rilevamento e altri che possono restituire la stessa misura a richieste successive per un certo arco di tempo.

È invece un aspetto comune a tutti i sensori quello di poter offrire azioni bloccanti e non bloccanti. Infatti, indipendentemente dal tipo di misura, ci si può aspettare che l'interazione con il mondo fisico comporti dell'attesa. Si deve prevedere che alcuni clienti siano disposti ad aspettare il risultato mentre altri desiderino proseguire l'esecuzione ed essere notificati del risultato in un secondo momento, a seconda del dell'uso che deve essere fatto della misurazione. Sarà necessario quindi rendere disponibile un meccanismo di richieste bloccante e uno non bloccante.

L'ultima considerazione di carattere generico riguarda il meccanismo di identificazione di un sensore. Prevedendo infatti di rendere distribuita la rete dei sensori non è possibile fare affidamento a identificatori come l'indirizzo logico assegnato dalla Java Virtual Machine al sensore ed è necessario prevedere un sistema di identificatori univoci.

Sarebbe possibile lasciare ad ogni sensore la responsabilità di crearsi, caricare la propria configurazione ed avviarsi. Tuttavia, è evidente che i sensori abbiano una caratteristica modulare e necessitino tutti di servizi comuni come la gestione del proprio ciclo di vita e il naming. Per questo motivo si è deciso di far "vivere" i sensori all'interno di un container, in maniera analoga a quanto avviene ad esempio per una stazione metereologica che ospita al suo interno un certo numero di sensori eterogenei. Per loro natura le stazioni devono essere progettate in modo da poter operare su dispositivi a basso consumo e basso costo.

Al container ideato per i Sensor si dà nome di Station (Figura 5). La stazione fornisce il supporto di base al loading e alla autoconfigurazione dei sensori, alla gestione del loro ciclo di vita e al restart automatico in caso di failure di un singolo sensore. L'obiettivo è far sì che, indipendentemente dal tipo di sensore che si vuole introdurre, uno sviluppatore debba solamente produrre l'interfaccia del sensore e implementare la logica specifica di funzionamento. Fatto ciò sarà la Station a prendersi carico del sensore fornendo i servizi necessari al suo funzionamento e rendendolo disponibile ad altri per l'utilizzo.

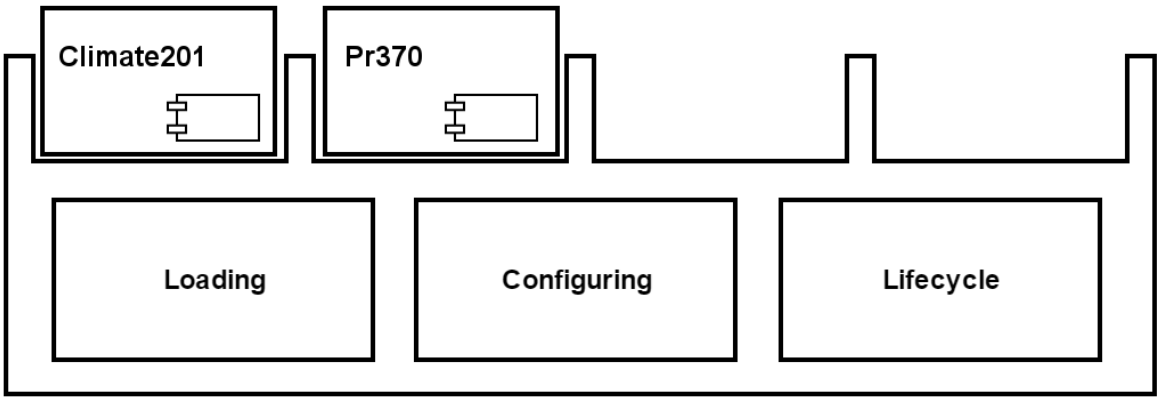


Figura 5 Schema a blocchi del container Station

Tra le caratteristiche desiderate per un container di questo tipo vi sono la tolleranza al guasto di un singolo sensore in termini di isolamento e auto recovering. Questo perché il funzionamento di un

sensore non influisca sugli altri ospitati dalla station e perché sia possibile riavviare in automatico un sensore in stato di fault.

Si vuole inoltre sollevare gli sviluppatori di sensori dal compito di rendere disponibili i sensori ai clienti. Le Station devono pubblicare i sensori da esse ospitati in modo che siano disponibili per i clienti. Si desidera inoltre un alto livello di trasparenza per gli sviluppatori ed i clienti nei confronti di questo meccanismo. I sensori potrebbero essere resi disponibili sia in locale che nel distribuito, senza comportare alcuna necessità di modifica né al codice del sensore, né a quello dell'utilizzatore.

Le stazioni devono infine permettere l'attivazione e la disattivazione di un sensore su richiesta di un utente. Quest'ultima funzionalità vista come caso di utilizzo esperto del sistema da parte di un agente intelligente in grado di determinare la necessità di attivazione o meno di un sensore.

2.2 Funzionalità del sistema distribuito

Una volta definite le caratteristiche dei Sensor e della Station si analizzano le funzionalità desiderate per il sistema distribuito, quello che realizzerà effettivamente la rete di sensori. Come esposto nel capitolo precedente è posta grande attenzione nel rendere trasparente per gli sviluppatori e gli utilizzatori dei sensori la caratteristica distribuita del sistema.

Con la configurazione esposta, i sensori sono accessibili anche da altri sensori, i quali possono utilizzare i primi come fonte di informazioni per sintetizzare risultati più complessi. Questo comportamento è utile quando il sensore intermedio è dotato di potere computazionale maggiore utilizzabile per aggregare misurazioni grezze provenienti da dispositivi meno performanti[7].

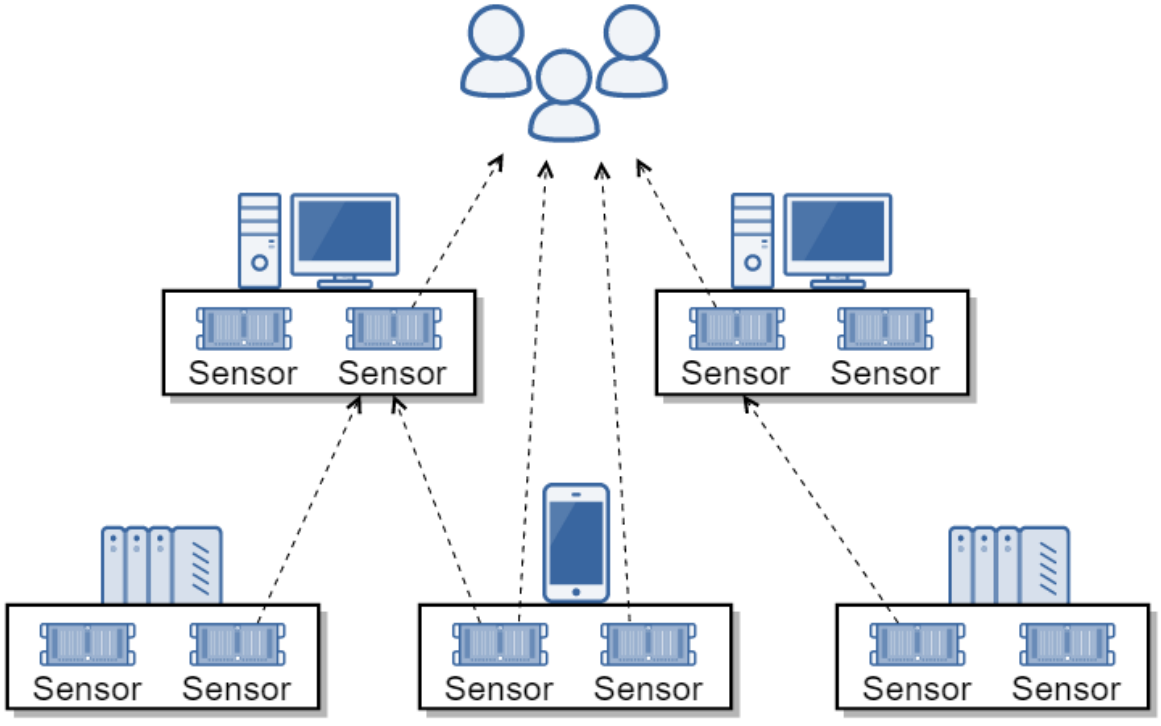


Figura 6 Esempio di configurazione della rete di sensori

[7] Jarrod, T., Ian, A., & Gilles, G. (2002). *Sensor Abstraction Layer.* SP&E

2.2.1 Provider

Per rendere distribuita la rete dei sensori è necessario prevedere un supporto all'interazione tra le diverse stazioni e i sensori da esse ospitati. Primo requisito tra tutti è individuare un sistema di nomi logici per permettere l'identificazione e l'ottenimento delle risorse in gioco.

Tale sistema di nomi e accesso remoto potrebbe essere realizzato in numerosi modi. Non volendo legare il sistema ad un'implementazione specifica del servizio si è introdotta una API generica aperta a diverse realizzazioni tra loro intercambiabili.

Un Provider è l'interfaccia attraverso la quale Station e clienti possono registrare, ricercare e ottenere i sensori all'interno del sistema distribuito. I provider devono garantire che l'utilizzo dei sensori sia trasparente all'accesso locale o remoto. Non potendo prevedere in anticipo la gamma di interfacce e implementazioni dei sensori è bene che i provider siano in grado di lavorare con un qualsiasi sottotipo dell'interfaccia Sensor e della classe base AbstractSensor presentate in precedente, considerando anche di non conoscere a priori i tipi di dati scambiati come risultato delle misurazioni.

Per le caratteristiche di dinamicità del sistema è anche opportuno che i provider possano notificare la registrazione e la deregistrazione, in modo tale che i clienti interessati possano accorgersi subito dell'ingresso e dell'uscita di stazioni e sensori.

È inoltre conveniente prevedere funzionalità di ricerca avanzate per i sensori in aggiunta a semplici lookup di un sensore secondo il suo nome logico. In tal modo si potranno eseguire con facilità ricerche come: *"tutti i sensori di temperatura"*, *"tutti i sensori in stato di fault"* e ancora *"tutti i sensori ospitati dalla stazione X"*.

Non si impongono vincoli invece sulla natura e sulla collocazione del provider, permettendo che esso possa fungere da sistema di nomi locale a un singolo nodo o in modo più ampio per tutti i nodi di una rete, prevedendo inoltre che il servizio possa essere realizzato in modo centralizzato oppure a sua volta distribuito su più nodi.

Si auspica che l'implementazione del Provider sia facilmente scalabile, in grado quindi di gestire efficientemente sia la rete dei sensori di un piccolo appartamento, sia un edificio delle dimensioni di un grattacielo.

2.2.2 Integrazione del servizio di naming nelle Station

Al container Station si va ad aggiungere una funzionalità che consiste nell'interazione con un Provider per registrare e deregistrare sensori.

Si potrebbe lasciare che ogni implementazione di Provider fornisca anche una implementazione di Station adatta a quello specifico provider. Tuttavia, questo implicherebbe anche la necessità di realizzare client specifici per ogni Provider andando ad intaccare una delle caratteristiche prefisse.

Grazie alle astrazioni introdotte dalla API è possibile realizzare una Station in grado di interagire con un generico provider. Il meccanismo utilizzato è simile a quello utilizzato da JDBC per caricare i driver di accesso ad un database[8]: a compile time la Station e i client dipendono solamente

[8] http://www.oracle.com/technetwork/java/javase/jdbc

dall'interfaccia Provider, mentre il caricamento dell'effettiva implementazione è effettuato a *runtime*, differendo in questo modo anche la dipendenza.

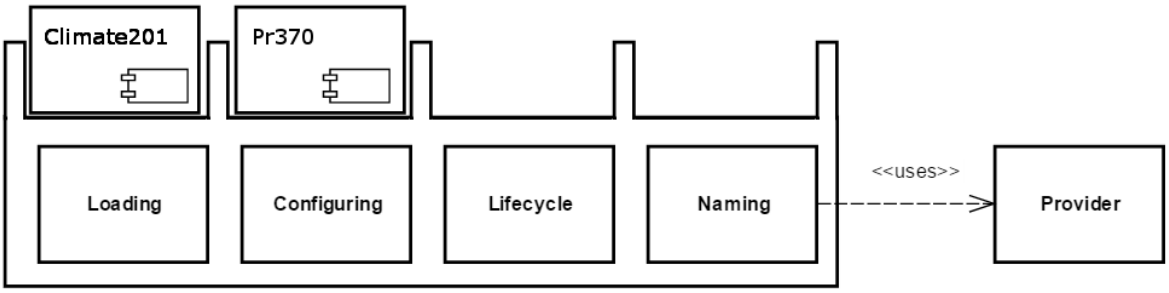


Figura 7 Utilizzo di un Provider da parte della Station per offrire un servizio di naming

2.2.3 Client

Aver separato sin dal primo momento le interfacce dei sensori dalle implementazioni permette solleva i clienti dalla dipendenza da queste ultime. Inoltre questo permette di realizzare meccanismi più complessi nei quali l'utente comunica attraverso l'interfaccia non direttamente con il sensore ma con un proxy al cui interno è nascosta la logica di comunicazione con il sensore remoto.

Per dialogare con un sensore il client dovrà ottenere un Provider in modo analogo a quanto viene fatto dalle Station e quindi utilizzarlo per ricercare i sensori di suo interesse. Sarà il provider a fornire i meccanismi necessari all'utilizzo remoto del sensore.

2.3 Panoramica dei moduli progettuali

Viene qui di seguito presentata una panoramica per visualizzare le entità in gioco nel sistema e le relazioni di dipendenza tra di esse. La suddivisione tra i moduli è pensata per garantire l'estensibilità del sistema e facilitarne l'utilizzo.

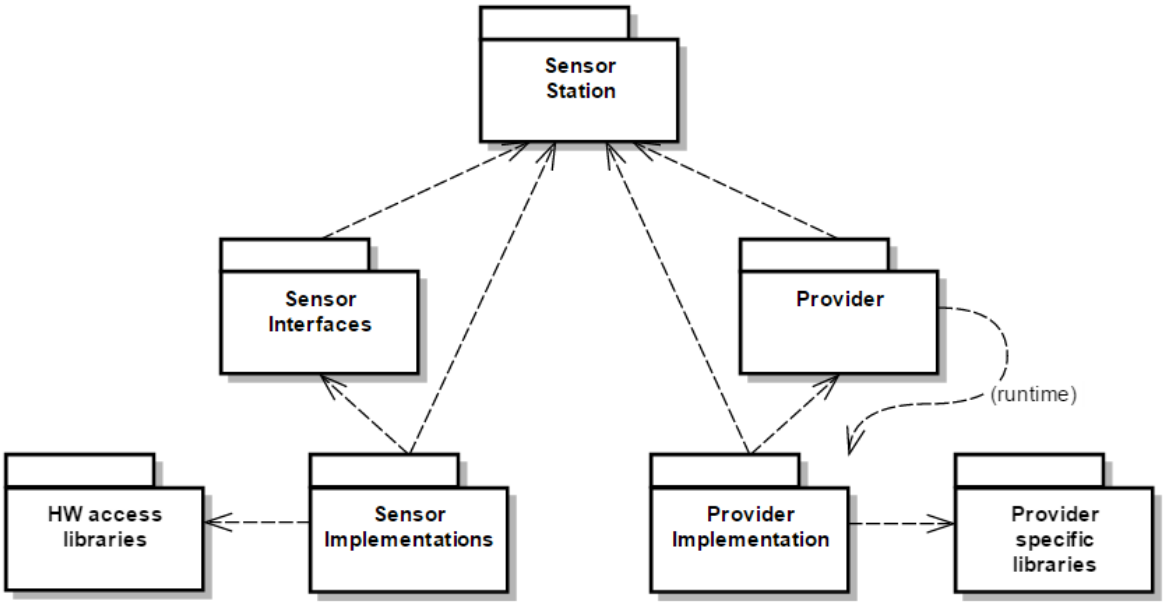


Figura 8 Overview dei moduli del sistema e delle loro dipendenze

SensorStation contiene la descrizione del container Station e dei suoi componenti Sensor, sia in termini di interfacce che in termini di classi base.

SensorInterfaces e **SensorImplementations** rappresentano rispettivamente le interfacce e le implementazioni dei sensori, entrambi dipendono da SensorStation per estendere la gerarchia dei

sensori. Inoltre si prevede che SensorImplementations abbia dipendenze aggiuntiva da librerie necessarie per accedere all'hardware dei sensori ed è corretto che queste dipendenze non vengano ereditate da chi utilizza i sensori.

Provider contiene la definizione dell'interfaccia del Provider e una Station in grado di utilizzare un Provider, dipende da SensorStation in quanto ne estende i servizi. Necessita di ProviderImplementation solamente a *runtime*.

ProviderImplementation rappresenta una generica realizzazione del servizio Provider e dipende dunque da esso. Può avere dipendenze da librerie necessarie al suo funzionamento che non si aggiungono però alle dipendenze di chi usa Provider.

Client rappresenta un generico utilizzatore della rete di sensori, per questo motivo ha necessariamente bisogno di conoscere Provider, SensorStation e SensorInterfaces. A *runtime* avrà bisogno che sia disponibile un'implementazione di Provider.

Tutti i moduli che definiscono dei servizi o delle interfacce devono essere il più possibile chiusi a modifiche, in modo da non provocare variazioni a cascata su tutti i moduli che dipendono da essi. Sono al contrario aperti al cambiamento e all'estensione tutti i moduli che le implementano, permettendo così di costruire con grande libertà nuovi provider e sensori nel rispetto delle interfacce definite stabilmente.

3 Implementazione del sistema

Consolidate le scelte progettuali si procede a concretizzarle in termini di implementazione del sistema. Il capitolo si apre con una breve descrizione degli strumenti hardware e software utilizzati per lo sviluppo, per poi proseguire con un'attenta esposizione dell'architettura del sistema costituito da sensori, stazioni e provider.

3.1 Strumenti di sviluppo software

3.1.1 Java come linguaggio di sviluppo

Il linguaggio di programmazione scelto per lo sviluppo del *middleware* è Java, determinato in seguito alle seguenti considerazioni.

- Java è disponibile per una vasta gamma di sistemi hardware e software, dai microcomputer ai supercomputer, su Windows e su Unix e per questo adatto a contribuire alla realizzazione della caratteristica di uniformità desiderata
- Tramite il progetto tuProlog[9] dell'Università di Bologna, Java è integrato con il linguaggio di programmazione logica Prolog, permettendo così a quest'ultimo di accedere alla rete di sensori e realizzare sistemi domotici intelligenti multiparadigma
- Java offre nativamente RMI come API di remotizzazione e framework di sviluppo di sistemi distribuiti, strumento che è stato insegnato durante il corso di Reti di Calcolatori T e verrà utilizzato per realizzare un'implementazione di esempio delle API del Provider
- La libreria open source Pi4J offre un'interfaccia *object-oriented* per accedere alle componenti hardware del Raspberry Pi, nota piattaforma per la realizzazione di progetti basati su microcomputer
- Come ultima motivazione, ma non per questo meno importante, Java è il linguaggio *object-oriented* sul quale è stata basata la maggior parte della didattica del corso di Ingegneria Informatica da me seguito, per questo motivo risulta il linguaggio nel quale ho maggiore confidenza

3.1.2 Maven per l'automatizzazione del processo di sviluppo

La suddivisione modulare del sistema vista nel capitolo precedente ne comporterà la distribuzione su svariati archivi jar. Viste anche le numerose relazioni tra un pacchetto e l'altro non è certo pensabile gestire le dipendenze "a mano" compilando e copiando i pacchetti ad ogni modifica del codice. Per questo motivo ci si affida a uno strumento di automatizzazione del processo di sviluppo quale Maven[10].

Maven è propriamente un *building automation tool* usato principalmente in contesti Java per i quali è nato, ma flessibile ed estendibile a progetti in C#, Ruby, Scala o altri linguaggi. Maven è nato come evoluzione rispetto ad Ant permettendo una semplificazione del processo di sviluppo grazie al principio di *convention over configuration*, una standardizzazione della struttura dei progetti grazie al concetto di archetipo e al sistema di plugin e un avanzato sistema di risoluzione automatica delle dipendenze.

I vantaggi dell'utilizzo di uno strumento standard sono numerosi. Tra i tanti esempi, il beneficio di non dover comprendere la struttura di un progetto già avviato da parte di un nuovo sviluppatore incluso nel team. E ancora la grande disponibilità di plugin pronti all'uso che sostituiscono la

[9] http://apice.unibo.it/xwiki/bin/view/Tuprolog/
[10] https://maven.apache.org/

scrittura di target Ant scritti a mano. Non ritendendo però necessaria una descrizione completa di Maven, si procede a introdurre in breve le sole funzionalità utilizzate e la loro configurazione.

Il cuore di un progetto Maven è costituito dal Project Object Model, un file XML in cui specificare i metadati del progetto e parametri dei plugin che divergono dalla convenzione di default. La maggior parte delle funzionalità di Maven è fornita dal sistema plugin. Esistono plugin per compilare, testare e gestire i sorgenti, generare documentazione e molti altri ancora. Alcuni plugin di base sono inclusi di default in ogni progetto, altri possono essere specificati e configurati in un'apposita sezione del file POM.

Maven introduce inoltre il concetto di *lifecycle*, ovvero un insieme di *goal* lanciati in modo sequenziale al fine di raggiungere un obiettivo come compile, test o install. Durante il normale *lifecycle* di compilazione Maven svolge l'importante funzione di risoluzione delle dipendenze. Per dipendenze si intendono artefatti come librerie o moduli necessarie al progetto. Le dipendenze vengono specificate nel POM e sottoposte a valutazione da parte di Maven. Ogni artefatto riferito dal progetto corrente viene reperito da repository locali o remoti e messo a disposizione durante la fase in cui è necessario. Vengono valutate anche le dipendenze transitive, cioè artefatti non direttamente inclusi nella lista delle dipendenze ma da esse riferiti (Figura 9). Di fatto viene alleggerito lo sviluppatore dal compito di localizzare e scaricare il pacchetto giusto nella versione giusta di ogni libreria necessaria al progetto.

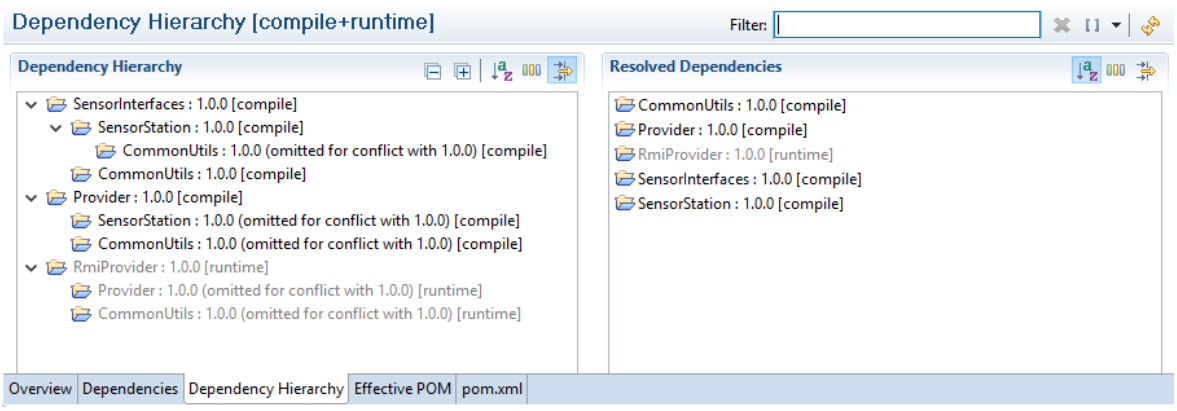


Figura 9 Risoluzione delle dipendenze transitive a compile time e a runtime per un Client del sistema che utilizza la realizzazione RMI del Provider (Maven plugin per Eclipse)

Per concludere, nei progetti che compongono il sistema è stato configurato il plugin di compilazione affinché utilizzasse Java nella sua versione 8 e affinché conservasse i nomi dei parametri dei metodi per favorire la leggibilità dei metadati ottenuti a *runtime* tramite Reflection.

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.3</version>
  <configuration>
    <compilerArgument>-parameters</compilerArgument>
    <testCompilerArgument>-parameters</testCompilerArgument>
    <source>1.8</source>
    <target>1.8</target>
  </configuration>
</plugin>
```

3.1.3 Git per versioning e teamwork

È stato scelto Git[11] come strumento di *version control* per seguire il processo di sviluppo del software nelle sue fasi. In questo modo è stato semplice seguire l'evoluzione del codice, tenere traccia dei cambiamenti e testare nuove funzionalità su *branch* temporanei.

Il codice è stato inoltre frequentemente caricato su una *repository* remota ospitata da BitBucket, per garantire ad altri studenti la possibilità di collaborare al progetto in maniera concorrente e l'uso futuro del sistema realizzato.

3.2 Componenti hardware e software per i sensori

3.2.1 Raspberry Pi e GrovePi StarterKit per i sensori fisici

Il Raspberry Pi è un *single-board computer* sviluppato nel Regno Unito dalla Raspberry Pi Foundation[12]. L'idea di base è la realizzazione di un dispositivo economico, concepito per stimolare l'insegnamento di base dell'informatica e della programmazione nelle scuole. È basato su un hardware open source e sistema operativo *linux-based*, permettendo grande possibilità di customizzazione.

Il suo costo contenuto, il *form factor* ridotto, l'enormità di moduli aggiuntivi e la partecipazione attiva di una grossa community di *makers* hanno contribuito alla sua enorme popolarità sul mercato. Oggi trova applicazione in un'infinità di progetti in ambito di robotica, home automation[13], 3D printing, home security e web server domestici.

Il kit GrovePi è una scheda di estensione per Raspberry Pi sviluppata e commercializzata dalle Dexter Industries[14]. Si propone di rendere facile e veloce la prototipazione di dispositivi per l'IoT, riducendo al minimo il tempo di set up dell'hardware e di configurazione delle librerie software. Lo Starter kit comprende numerosi sensori e attuatori *plug and play*, oltre che accessori per il Raspberry come un dongle WiFi e una scheda SD con il sistema operativo Raspbian, completo delle librerie necessarie all'interfacciamento con l'hardware. Trattandosi di un sistema operativo basato su Debian, non ci sono problemi di compatibilità con il software Java.

3.2.2 Weather Underground API per i sensori virtuali

Il sito wundergound.com espone delle web API per accedere al servizio di dati meteorologici Weather Underground[15]. In questo modo è possibile ottenere informazioni molto complete e dettagliate riguardo alle condizioni meteo e astronomiche di un particolare luogo, con granularità giornaliera oppure oraria. Grazie alle ricche API è possibile inoltre richiedere inoltre una *history* recente e le previsioni future.

L'accesso ai dati è regolato tramite una *key* di licenza, ottenibile tramite registrazione. Sono disponibili diverse forme di registrazione, sia gratuite che a pagamento, che differiscono nel numero massimo di richieste giornaliere e nella natura dei servizi accessibili. Per le finalità puramente esemplificative di questa tesi è sufficiente la versione gratuita di base.

Le query al servizio vengono effettuate tramite richieste HTTP a determinati URL, costituendo così una RESTful API. I dati restituiti sono in formato JSON o XML, ad eccezione di alcuni servizi come le webcam e le mappe che restituiscono immagini.

[11] https://git-scm.com/
[12] https://www.raspberrypi.org/
[13] Carano, M. (2015). *Sperimentazione di tecnologie Raspberry in contesti di home intelligence*
[14] http://www.dexterindustries.com/grovepi/
[15] https://www.wunderground.com/weather/api

3.3 Framework di base per i Sensori

3.3.1 Architettura della Station

Partendo dall'analisi delle funzionalità del container Station effettuata nel capitolo precedente si vanno ora a implementare i servizi necessari. Le classi qui presentate sono contenute nel package SensorStation.jar di cui si dà una visione di insieme in Figura 10

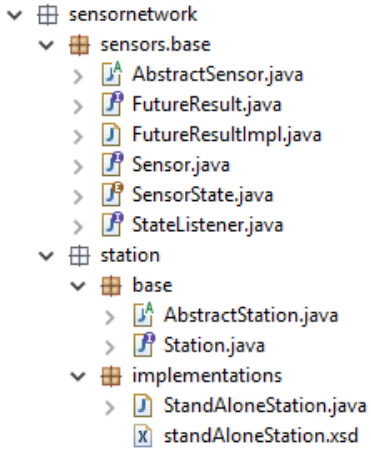


Figura 10 Package SensorStation.jar

Le classi contenute in `sensornetwork.sensors.base` rappresentano la base di partenza per realizzare le gerarchie di interfacce e di implementazioni dei sensori (Figura 11).

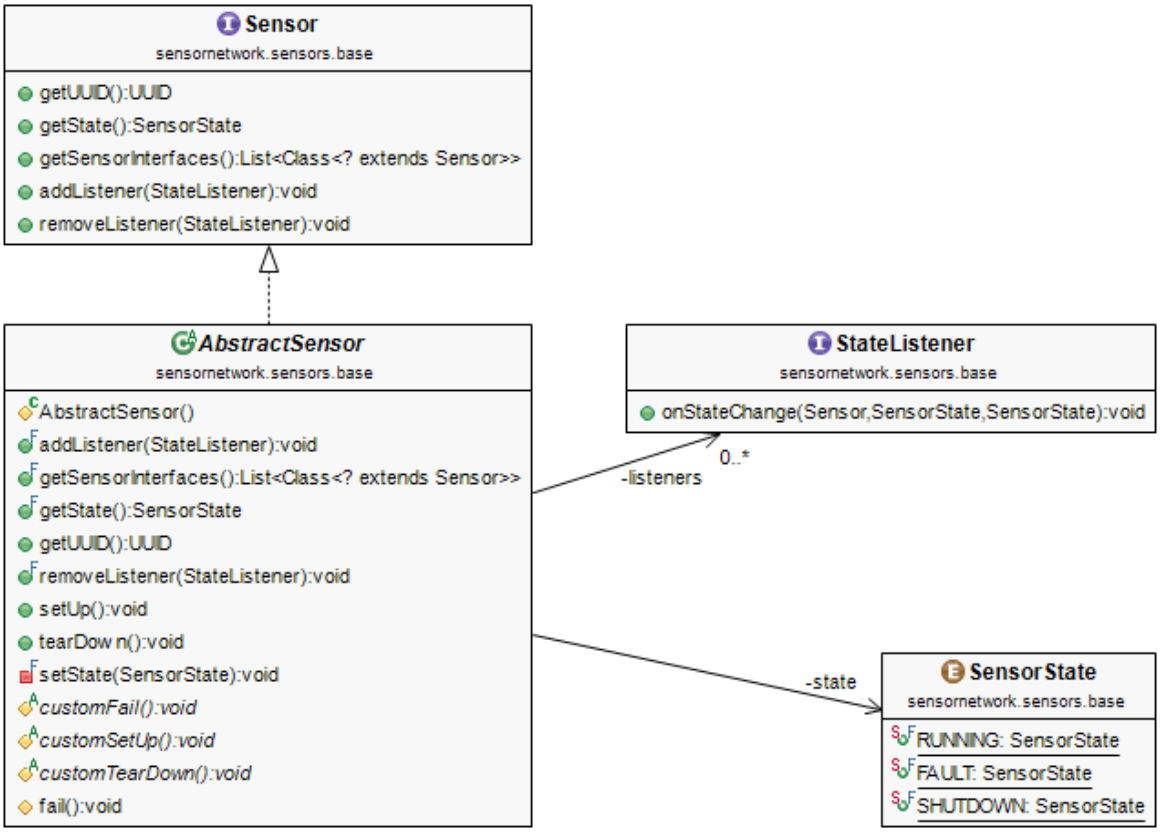


Figura 11 Classe e interfaccia di base per i sensori

L'interfaccia `Sensor` esprime i metodi pubblici fondamentali per il sensore, in particolare quelli che permettono di ottenere il suo identificatore unico, di ottenerne lo stato, gestire i listener per i cambiamenti di stato e ottenere la lista delle sottointerfacce di Sensor implementate. La classe

astratta AbstractSensor contiene la logica comune a tutti i sensori relativa ai metodi di interfaccia e ai metodi utilizzati dal container per gestire i sensori.

Particolare cura è stata data alla gestione dei *listener*: registrazione, deregistrazione e notifica sono gestite dalla classe astratta e non è necessario, oltre che possibile, ridefinirne la logica. Ad ogni cambiamento di stato tutti i listener vengono notificati in maniera parallela e asincrona. In questo modo il sensore è in grado di procedere con il suo funzionamento normale anche se sono presenti numerosi listener e se uno o più listener effettua operazioni di lunga durata.

La descrizione dello stato di un sensore è affidata all'enumerativo `SensorState`, contenente gli stati:

- **`RUNNING`**: sensore avviato, stato di esecuzione normale
- **`FAULT`**: sensore in stato di errore per cause interne, potenzialmente ripristinabile
- **`SHUTDOWN`**: sensore non avviato

La transizione tra uno stato e l'altro è comandata dalla Station tramite i metodi `setUp` e `tearDown`, oppure dal sensore stesso tramite il metodo `fail`. Tuttavia non è permesso al sensore di modificare direttamente lo stato tramite il metodo `setState`. Le sottoclassi possono implementare i metodi `customSetUp`, `customFail` e `customTearDown` per eseguire azioni specifiche in seguito di un cambiamento di stato.

Poiché il sensore è stato progettato per poter essere anche esportato, non è detto che nell'esportazione si mantenga l'univocità dei riferimenti. Per ovviare a questo problema, si inserisce nell'interfaccia base un metodo per ottenere un identificativo univoco del sensore rappresentato da un UUID generato all'instanziazione del sensore da parte della Station. Un UUID è semplicemente un numero di 128 bit la cui univocità è data dall'enorme numero di possibili id, inizialmente era generato a partire dal *MAC address* del dispositivo e dal suo *timestamp* della creazione con granularità di 100 nanosecondi.

Avendo necessità di configurare alcuni parametri di un sensore prima del suo avvio, è possibile marcare i *field* della classe con l'annotazione `@LoadableParameter` lasciando che la Station si occupi del loro caricamento basandosi su file di configurazione esterni. Maggiori dettagli ed esempi riguardo a questo semplice meccanismo di *dependency injection* sono dati in seguito.

Il container `Station` deve offrire le funzionalità di caricamento automatico dei sensori, configurazione tramite *dependency injection* e gestione del loro ciclo di vita. Al fine di permettere l'automatizzazione dell'avvio di una station è necessario racchiudere in un descrittore di *deployment* le informazioni necessarie al processo. Questi descrittori sono realizzati in forma di file XML, tramite una sintassi definita in un file XSD e verificata al momento del *parsing*. Di seguito si commenta un esempio di descrittore:

```
<station>

    <sensor loadAtStartup="false">
          <name>tempAndHumiditySensor</name>
          <class>
           sensornetwork.sensor.implementations.TempAndHumiditySensor
          </class>
          <parameters>temp.properties</parameters>
    </sensor>

    <sensor loadAtStartup="true">
          <name>weatherSensor</name>
          <class>
                sensornetwork.sensor.implementations.Wunderground
          </class>
          <parameters>wundeground.properties</parameters>
    </sensor>

</station>
```

Ogni tag `<sensor>` racchiude la descrizione di un sensore da caricare, comprendendo il suo nome logico, la classe da instanziare e un file `.properties` da utilizzare per il caricamento dei parametri. L'attributo `loadAtStartup` indica se al momento del caricamento il sensore deve essere anche posto in stato RUNNING o meno.

Da notare che il documento XML è caricato solamente all'avvio della stazione, di conseguenza l'aggiunta dinamica di nuovi sensori a una stazione avviata non è possibile. In questa situazione sarà necessario fermare la Station, modificare il file di configurazione e riavviarla, fornendo contestualmente gli archivi `.jar` aggiuntivi contenenti le implementazioni dei nuovi sensori. Si prevede che in futuro si possa modificare questo comportamento facendo sì che una Station rilevi automaticamente nuovi sensori e possa caricarli a *runtime*.

La Station espone inoltre metodi pubblici di interfaccia che permettono un utilizzo avanzato di una stazione (Figura 12). È possibile ad esempio richiedere l'avviamento o lo spegnimento di un sensore a seconda delle necessità del sistema intelligente che governa la rete di sensori.

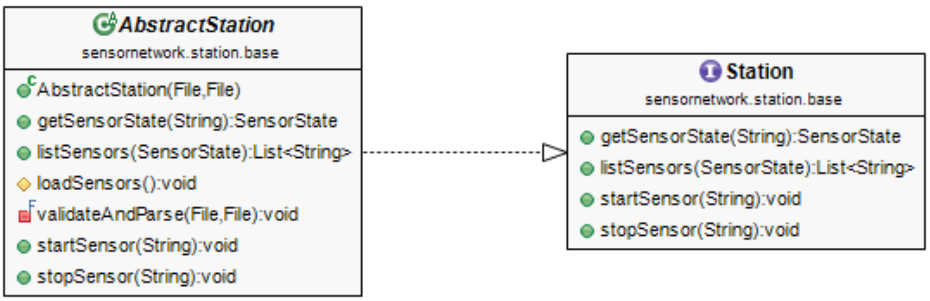


Figura 12 Diagramma della Station

In questo pacchetto si inserisce inoltre una `StandAloneStation`. Non essendo connessa con un Provider, essa non è in grado di fungere da nodo della rete di sensori. Tuttavia rappresenta una base di testing locale per il funzionamento del container e dei sensori, utilizzabile come componente a sé stante in contesti isolati in cui si vogliono collaudare i sensori all'interno di una sola stazione e di una sola JVM.

3.3.2 Implementazione di alcuni sensori

Prima di procedere alla definizione delle API del Provider sono stati realizzati alcuni sensori, in modo tale da poter sperimentare realmente le capacità delle Station. Come *proof of concept* vengono

descritti tre sensori realizzati con strumenti e scopi diversi, mostrando così la grande flessibilità di applicazione dei sensori.

- `Temp5000`: un semplice *mock* per un sensore di temperatura, in grado di fornire letture bloccanti e non bloccanti, con caching dei risultati
- `TempAndHumiditySensor`: sensore di umidità e temperature realizzato su piattaforma Raspberry Pi 2 + GrovePi
- `Wundeground`: sensore virtuale, in grado di reperire dati astronomici e meteorologici utilizzando le web API offerte dal servizio wunderground.com

Le classi qui definite sono contenute nei due pacchetti SensorInterfaces.jar e SensorImplementations.jar. La separazione in due package separati ha fondamentalmente due vantaggi (Figura 13). Per primo la possibilità di realizzare implementazioni alternative dei sensori, le quali rimangono compatibili anche se basate su tecnologie diverse, purché sia rispettata la visione "ai morsetti". In secondo luogo si svincolano gli utilizzatori del package di interfacce dalle dipendenze necessarie alle implementazioni, rappresentate in questo esempio dalle librerie di accesso hardware Pi4J[16], WiringPi e GrovePi.

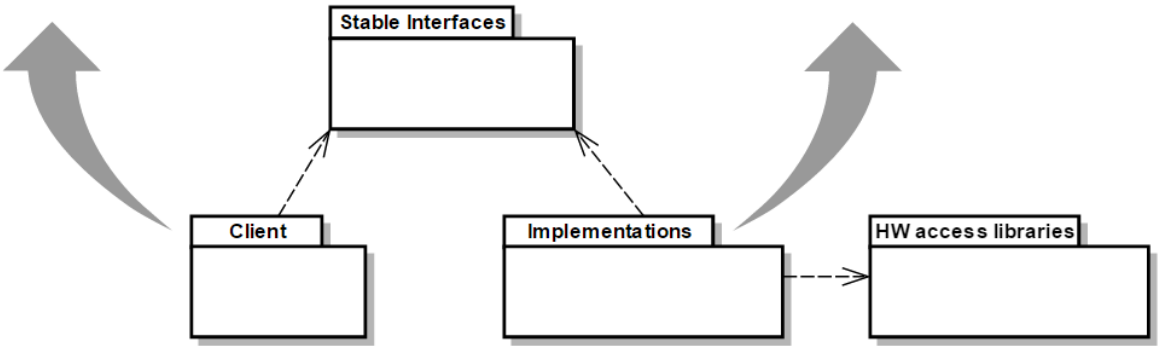


Figura 13 Dipendenze in gioco e possibilità di sviluppo indipendente di clients e implementazioni

Temp5000

Questo *mock* offre dati di temperatura generati in modo casuale. Il suo vero scopo è infatti quello di mostrare la possibilità di eseguire letture bloccanti e non bloccanti, sfruttando inoltre il caching dei risultati per migliorare l'efficienza a fronte di richieste multiple consequenziali.

L'interfaccia pubblica di questo sensore offre metodi per la lettura sincrona e asincrona della temperatura, facendo uso dell'interfaccia `FutureResult` specificata nel package di base dei sensori per realizzare quest'ultima opzione. Internamente si fa uso delle classi standard di Java per il supporto alla concorrenza come `Runnable`, `ExecutorService`, Supplier e `CompletableFuture`, queste ultime due introdotte nella versione 8 insieme a molti altri utili meccanismi di gestione del flusso e della sincronizzazione tra *thread*[17,18].

[16] http://www.savagehomeautomation.com/projects/category/pi4j
[17] http://www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html
[18] https://www.infoq.com/articles/Functional-Style-Callbacks-Using-CompletableFuture

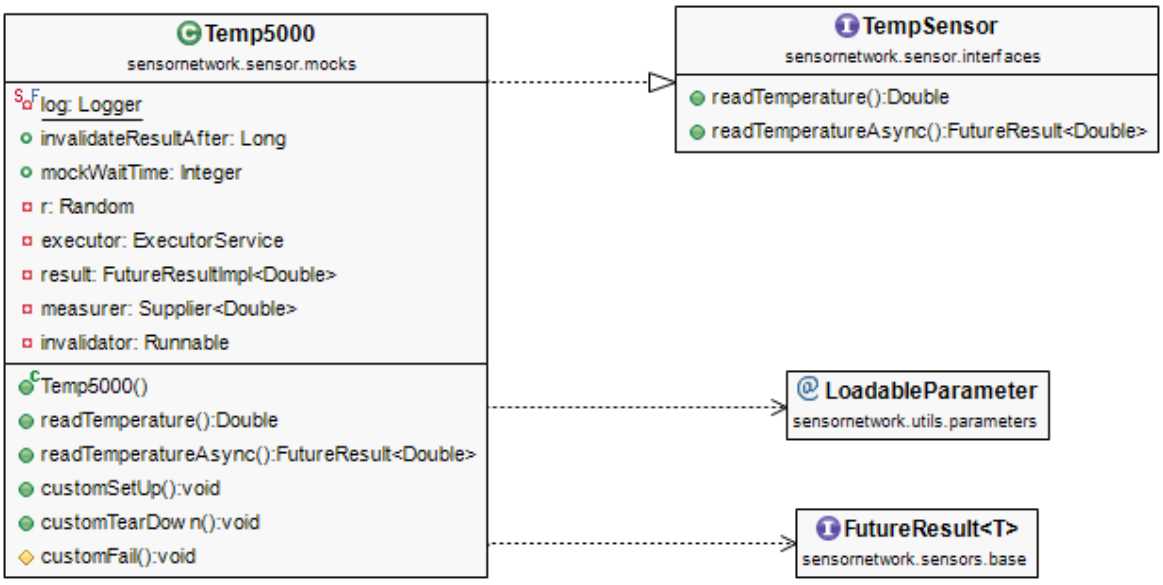


Figura 14 La classe Temp5000 e le principali relazioni con altre classi del sistema

La configurazione di questo sensore prevede il settaggio di due parametri:

```
@LoadableParameter(...)
public Long invalidateResultAfter = 5000L;

@LoadableParameter(...)
public Integer mockWaitTime = 1000;
```

Il primo rappresenta la durata della validità della cache, in altre parole il tempo dopo il quale un'altra misurazione è necessaria in quanto il risultato precedente non può più essere considerate valido. Il secondo è invece utilizzato nel finto processo di misura per simulare l'attesa dovuta all'accesso all'hardware.

TempAndHumiditySensor

La realizzazione di un dispositivo basato su Raspberry Pi 2 Model B corredato dal kit di moduli GrovePi+ StarterKit vuole essere un esempio concretamente utilizzabile in un contesto di *home automation*.

Tra tutti i sensori disponibili si sceglie di implementare una classe per interfacciarsi al sensore di umidità e temperatura. Per quanto riguarda la misura di temperatura viene mantenuta l'interfaccia `TempSensor` mostrata nell'esempio precedente, costruendone una analoga per l'umidità.

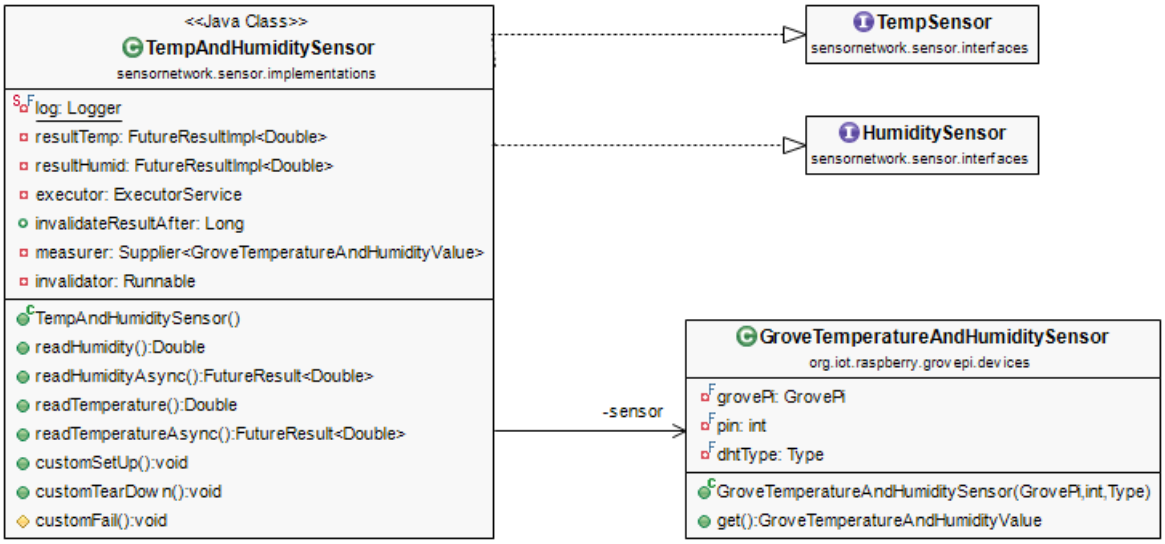


Figura 15 La classe TempAndHumiditySensor e le principali relazioni con le altre classi del sistema

In figura è mostrato lo scheletro della classe `TempAndHumiditySensor`. Trovando già nella classe `GroveTempAndHumiditySensor` una realizzazione della logica di accesso al sensore ed esecuzione della misura, è sufficiente costruire un *wrapper* attorno al sensore per renderlo compatibile con il sistema dei Sensor e convertire il `GroveTemperatureAndHumidityValue` restituito dal sensore in due valori Double per temperatura e umidità[19].

Si sottolinea che ogni sensore è responsabile per l'accesso al suo hardware e deve inoltre regolare accessi multipli concorrenti ai suoi metodi. È dunque necessario che metodi come `readTemperature` e `readHumidity` prevedano un meccanismo di sincronizzazione. Anche per questo sensore è stato previsto un meccanismo di caching del risultato, in modo tale da poter unire più richieste in una sola misura e duplicare il risultato per ogni cliente. Se necessario è possibile effettuare dei check di validità sui dati grezzi letti dal sensore e semplici operazioni di conversione prima di restituire il risultato ai clienti richiedenti.

Wunderground

Con il terzo e ultimo esempio di sensore si dimostra come sia possibile integrare nel sistema sensori virtuali. Il sensore `Wunderground` fa uso delle web API del servizio Weather Underground per restituire dati meteorologici come la temperatura e la velocità del vento e dati astronomici come l'orario di alba e tramonto. L'abilità di un sistema domotico di accedere a tali informazioni potrebbe risultare utile per definire gli orari migliori per azionare tapparelle, condizionatori e termosifoni, basandosi non soltanto su misurazioni locali.

Le query HTTP utilizzate dal sensore hanno la seguente forma:

```
http://api.wunderground.com/api/<key>/conditions/q/<state>/<city>.xml
http://api.wunderground.com/api/<key>/astronomy/q/<state>/<city>.xml
```

Il campo `<key>` dell'URL deve essere sostituito con la chiave di accesso alle API fornita al momento della registrazione al servizio. I campi `<state>` e `<city>` rappresentano rispettivamente lo stato e la città per i quali ottenere le informazioni meteorologiche o astronomiche. Il suffisso .xml al termine dell'URL indica al servizio di inviare i dati in formato XML, in alternativa al formato JSON restituito di default.

[19] https://github.com/InitialState/grovepi/wiki/Part-2.-DHT11-Temperature-and-Humidity-Sensor

Il documento XML ricevuto contiene un numero molto elevato di informazioni, si sceglie per semplicità di parsing di rendere disponibili solamente le più significative. I risultati meteo e astronomici sono rappresentati dalle seguenti classi:

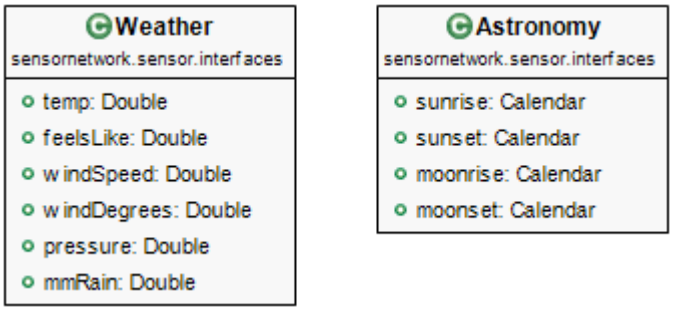


Figura 16 Classi per le rilevazioni meteorologiche e astronomiche

I parametri della chiave di accesso, dello stato e della città sono configurabili con il meccanismo visto in precedenza. Si aggiunge inoltre ai campi configurabili la durata di validità della cache per i risultati meteorologici. I dati astronomici invece sono per loro natura validi fino alla mezzanotte locale.

```
@LoadableParameter(...)
public Long invalidateResultAfter;
@LoadableParameter(...)
public String city;
@LoadableParameter(...)
public String state;
@LoadableParameter(...)
public String key;
```

3.3.3 Package di utilities

L'analisi ha evidenziato delle necessità orizzontali, comuni a tutti i moduli che compongono il sistema. Per questo motivo si decide di racchiudere le classi necessarie al logging e al caricamento dei parametri in un unico package di utilities (Figura 17)

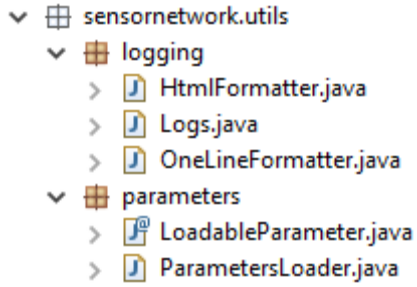


Figura 17 Package CommonUtils.jar

Logging

Per logging si intende il processo di produrre messaggi riguardanti l'esecuzione di un programma e aggregarli in un'unica destinazione persistente. Questa pratica consente di tenere traccia di situazioni di errore, warning e normali messaggi informativi, con il fine di analizzare il comportamento del sistema per risolvere eventuali bug e migliorarne le performances[20].

[20] http://www.vogella.com/tutorials/Logging/article.html

Il package `java.util.logging`, offerto nativamente dal JRE, contiene gli strumenti necessari per configurare il logging, indicando diversi livelli per i messaggi, destinazioni e formati multipli per l'output.

Le classi fondamentali della logging API sono[21]:

- `Logger`: la classe principale utilizzata per scrivere messaggi di logging, ogni logger fa riferimento a una gerarchia di logger nella quale i messaggi vengono mandati dai figli verso i genitori, contiene metodi per loggare messaggi a diversi livelli, insieme a un meccanismo di filtro sui messaggi e la possibilità di settare un livello minimo di errore
- `Handler`: ogni logger può avere associati più handler, su ogni handler il logger scrive i messaggi a lui indirizzati direttamente, previo controllo del livello minimo di logging e del filtro, e i messaggi provenienti dai figli, con il solo controllo sul livello minimo, ogni handler è mappato su una destinazione, tra le più comuni troviamo un file, la console o una socket
- `Formatter`: ogni handler scrive in output il messaggio ricevuto nel formato specificato dal formatter, sono disponibili formatter pronti all'uso come `XMLFormatter` ed è possibile definirne uno personalizzato

Livelli dei messaggi:

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

Utilizzando questi strumenti viene realizzata la classe `Logs` che permette di creare la radice della gerarchia di logger, a cui associa gli handler della console e di un nuovo file nella cartella logs. Il file utilizza un `HtmlFormatter` che incorpora stili CSS al fine di personalizzare il *rendering* del documento da parte di un browser.

Dependency Injection

Per separare la definizione delle classi dei sensori dalla loro configurazione si realizza un semplice meccanismo di *dependency injection*. I componenti necessarie al suo funzionamento sono tre:

- Un modo per indicare quali campi della classe devono essere caricati a *runtime*
- Un file di configurazione esterno
- Un'entità in grado di associare ad ogni campo il parametro letto da file

Tramite l'annotazione `@LoadableParameter` è possibile annotare un campo della classe affinché ne venga fatta injection. L'annotazione prevede che si possa indicare una descrizione *user friendly* del campo e una chiave utilizzata per identificare il valore corretto nel file di configurazione.

```
@LoadableParameter(userDescription="API key", propertyName="Key")
public String key;
```

Il file di configurazione impiega il semplice formato `.properties` nativamente interpretato da Java per mezzo della classe `Properties`. Per specificare i parametri in tale formato si utilizzano coppie chiave-valore:

```
City=Bologna
State=IT
Key=***************
InvalidateResultAfter=10
```

[21] http://tutorials.jenkov.com/java-logging/logger-hierarchy.html

La classe `ParametersLoader` contiene il metodo `loadParameters(File, Object)` che, presi come parametri un oggetto e un File, riempie i campi del primo con i valori contenuti nel secondo. Il meccanismo di caricamento utilizza la Reflection di Java per accedere a *runtime* ai campi pubblici di una classe e alle loro annotazioni. Il parsing dei valori dal file viene effettuato chiamando il metodo statico `valueOf(String)` del tipo corrispondente al campo da caricare, che quindi deve essere definito.

I limiti di questa implementazione naïve di *dependency injection* sono numerosi. Per primo, l'obbligo di dichiarare pubblici tutti i campi annotati e dunque una rottura dell'incapsulamento dei dati. In secondo luogo il parsing tramite il metodo `valueOf` del tipo del campo limita gli sviluppatori nella scelta dei parametri caricabili, costringendoli ad utilizzare solamente tipi come `Integer` e `Double` oppure a implementare un metodo statico con quella *signature*.

Tuttavia, quanto realizzato è sufficiente allo scopo del progetto e rimane comunque aperto per qualsiasi modifica e miglioramento.

3.4 Middleware per sensori distribuiti

3.4.1 Provider e ProviderStation

Una volta realizzate tutte le classi necessarie al funzionamento di una Station si procede a definire il servizio di naming necessario ad utilizzare i sensori all'interno del sistema. La definizione della API di un Provider, insieme ad altre classi accessorie, è contenuta nel pacchetto Provider.jar (Figura 18).

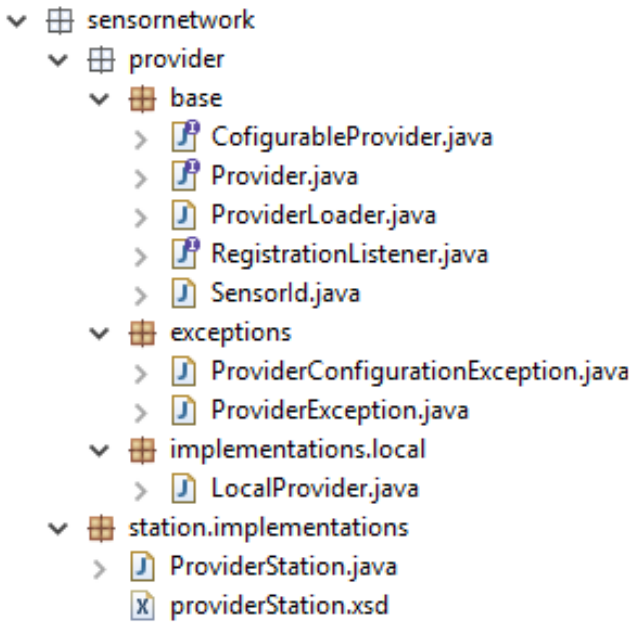


Figura 18 Package Provider.jar

All'interno del package `sensornetwork.provider.base` sono contenuti i tipi principali che compongono l'API Provider. In Figura 19 i diagrammi UML delle interfacce e delle classi.

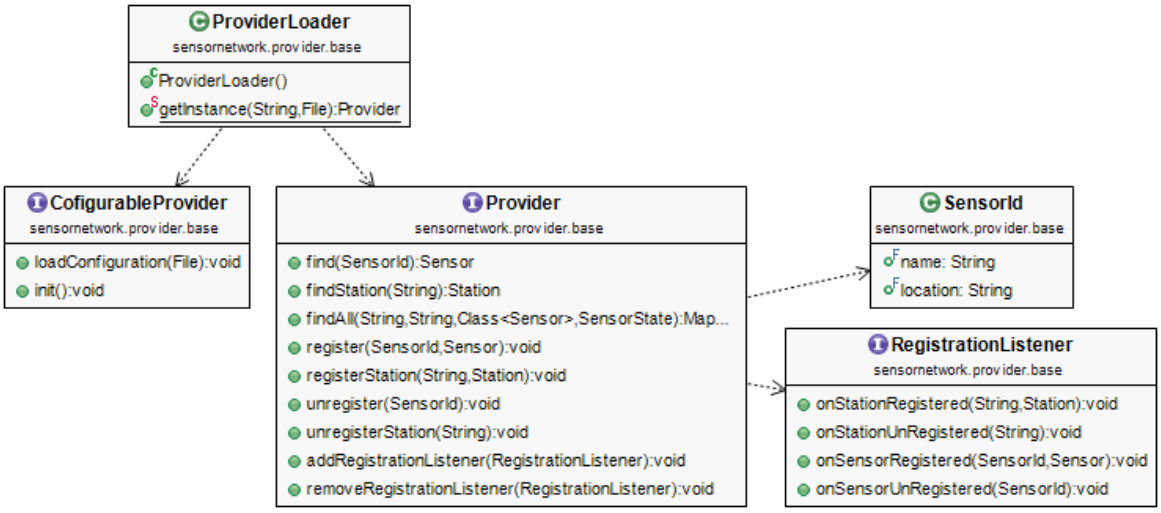


Figura 19 Principali tipi dell'API Provider

La classe `SensorId` rappresenta il nome logico che viene assegnato ad ogni sensore nel momento del suo ingresso nel sistema. Il nome logico è costituito da una stringa rappresentante la Station e una stringa rappresentante il Sensor, in modo da poter costruire nomi significativi come `["Cucina"," TempForno"]`.

L'interfaccia Provider rappresenta il cuore della API e contiene metodi per registrare un sensore, registrare una Station, gestire dei `RegistrationListener` per gli eventi di registrazione, ottenere una Station dato il suo nome, ottenere un sensore dato il suo id o una descrizione parziale delle sue caratteristiche.
Query complesse per ottenere i sensori possono essere effettuate tramite il metodo:

```
Map<SensorId, Sensor> findAll(String name, String location,
                    Class<? extends Sensor> type, SensorState state)
```

Al suo interno sono presenti quattro parametri di ricerca:

- **`name`**
  per indicare il nome del sensore (per cercare, ad esempio, tutti i sensori chiamati "temp", indipendentemente dalla Station su cui si trovano)
- **`location`**
  per indicare il nome della Station
- **`type`**
  è un'istanza della meta-classe `java.lang.Class` che descrive un sottotipo dell'interfaccia Sensor, utilizzabile per ricercare tutti i sensori che implementano una certa interfaccia
- **`state`**
  per indicare lo stato del sensore

Poiché l'interfaccia Provider definisce un servizio la cui effettiva implementazione sarà realizzata a parte e resa disponibile solo a *runtime* è necessario ragionare su quale sia il modo migliore per fornire a una Station il provider da utilizzare. Come prima soluzione si è valutato di realizzare subclassare Station in modo che il costruttore accettasse un'istanza di Provider. Tuttavia questo meccanismo implica che nel codice, ad esempio nel `main`, vengano creati nell'ordine il provider e la stazione. Inoltre, la creazione può essere effettuata solamente nel pacchetto che contiene la classe che implementa il provider o in un pacchetto da lui dipendente. Questo implica che per ogni

implementazione di provider vada anche realizzato un `main`, decentralizzando e replicando il codice inutilmente.

La soluzione migliore è parsa quella di realizzare la classe `ProviderStation` che carichi dal file di configurazione precedentemente descritto il nome della classe da utilizzare come provider e si affidi al metodo statico `getInstance` della classe `ProviderLoader` per ottenerne un'istanza. `ProviderLoader` si comporta di fatto come un service locator che dà accesso ai servizi di un Provider nascondendone l'implementazione.
In questo modo è sufficiente fornire a *runtime* le classi dell'implementazione e non è necessaria una dipendenza da esse a compile time. Un ulteriore vantaggio derivante da questa scelta è che per utilizzare un determinato provider sarà sufficiente specificarlo nell'XML, senza dover produrre ulteriore codice.

È stato inoltre valutato che in un contesto distribuito i provider possano avere bisogno di configurarsi e inizializzarsi, aprendo ad esempio delle connessioni internet, prima di poter essere utilizzati. Per questo motivo, se il provider espone l'interfaccia `ConfigurableProvider`, all'istanza appena creata verrà passato un file di configurazione e ne sarà chiamato il metodo `init`, all'interno del quale è possibile effettuare le inizializzazioni necessarie.

La `ProviderStation` è dunque un'estensione di `StandAloneStation` che fa uso di un provider e gestisce correttamente tutte le registrazioni e deregistrazioni, nonché la notifica di tali eventi ai listener registrati. Analogamente, lo schema XML per la configurazione di una `ProviderStation` deriva da un'estensione di quello precedente e aggiunge le informazioni necessarie all'identificazione della Station, all'instanziazione e all'inizializzazione del provider. Con riferimento al documento XML mostrato in precedenza, è di seguito riportato un nuovo esempio che fa uso del provider RMI descritto nel prossimo capitolo:

```
<station name="Climate Station">

    <provider>
          <class>
          sensornetwork.provider.implementations.
                rmi.client.RmiProviderClient
          </class>
          <parameters>rmiprovider.properties</parameters>
    </provider>

    <sensor loadAtStartup="false">
          <name>tempAndHumiditySensor</name>
          <class>
          sensornetwork.sensor.implementations.TempAndHumiditySensor
          </class>
          <parameters>temp.properties</parameters>
    </sensor>

    <sensor loadAtStartup="true">
          <name>weatherSensor</name>
          <class>
                sensornetwork.sensor.implementations.Wunderground
          </class>
          <parameters>wundeground.properties</parameters>
    </sensor>

</station>
```

3.4.2 Provider locale come *proof of concept*

Ancora una volta si è voluto produrre un prototipo funzionante del sistema per testarne le funzionalità in forma ridotta. `LocalProvider` è in grado di fungere da provider per più Station lanciate sulla stessa virtual machine. Difficile definire questo modello "distribuito", ma vengono già introdotti i concetti di SensorId e di più Station autonome in grado di mettere a disposizione i loro sensori (Figura 20).

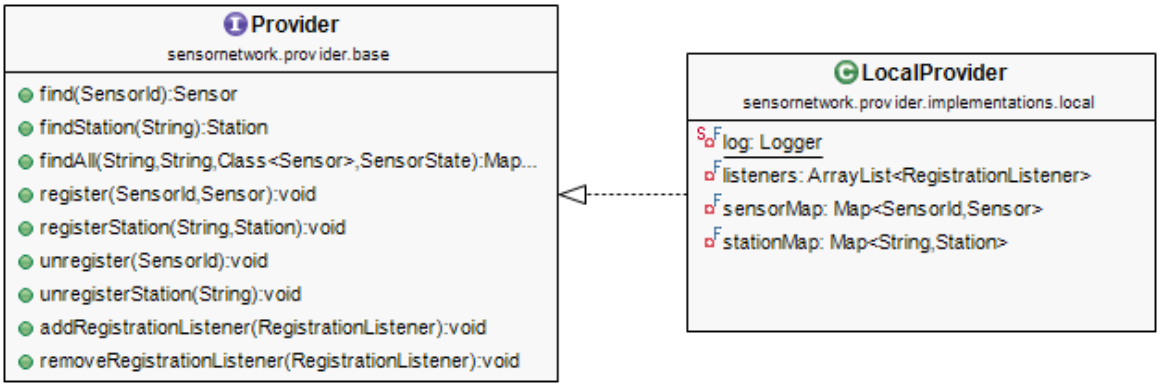


Figura 20 Implementazione locale di Provider

4 Realizzazione di un Provider in Java RMI

Il terzo capitolo conclude la definizione implementativa dell'architettura del sistema, lasciando volontariamente la definizione del Provider a livello di API e non di implementazione. Per dare completezza al progetto si realizza un Provider utilizzando la tecnologia Java RMI. In questo modo sarà possibile testare realmente l'efficacia dell'architettura progettata.

4.1 Tecnologia Java RMI

Java Remote Method Invocation è un'API che permette l'invocazione remota di metodi in un contesto distribuito Java-based. La si può considerare come l'equivalente *object-oriented* di RPC con supporto al trasferimento diretto di classi serializzate e alla *garbage collection* distribuita[22,23].

Questa tecnologia risulta ormai datata, tuttavia è stata scelta per realizzare un'implementazione del Provider per una serie di caratteristiche, di cui si riportano le principali:

- Essendo una soluzione puramente Java risponde all'obiettivo di portabilità definito dei capitoli iniziali, senza aggiungere vincoli dipendenti dallo specifico dispositivo
- L'uniformità garantita dalla piattaforma Java elimina anche la necessità di effettuare *marshalling* e *unmarshalling* dei dati tra un *device* e l'altro.
- Offre una funzionalità di caricamento dinamico delle classi dalla rete che permette ad un utilizzatore di lavorare con tipi di cui non conosce la definizione, garantendo in questo modo la massima libertà di estensione

Utilizzando le API di RMI si possono esportare interfacce di sensori verso altre JVM e tramite esse invocare i metodi di interesse comunicando attraverso una connessione TCP. Per reperire un oggetto esportato è possibile effettuare un lookup su di un `rmiregistry` presso il quale l'oggetto è stato registrato oppure ottenerne il riferimento da un altro oggetto remoto (Figura 21).

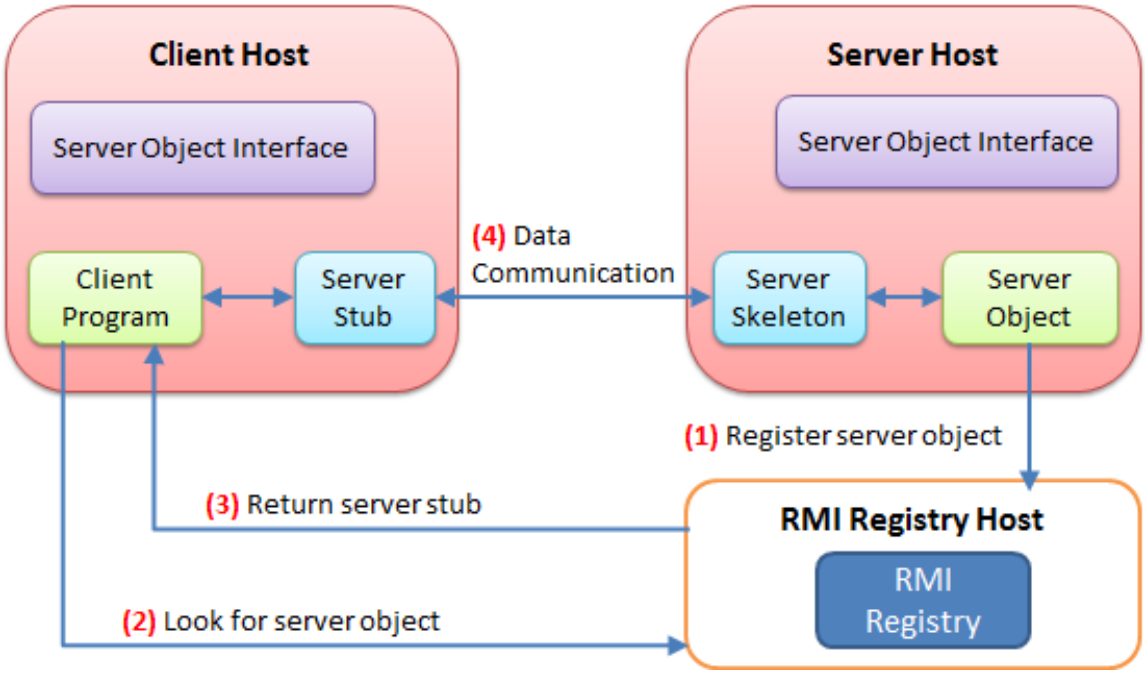


Figura 21 Reperimento di riferimenti remoti e invocazione di metodi tramite RMI

[22] http://java.sun.com/products/jdk/rmi/
[23] Pianciamore, M. (2002). *Programmazione Object Oriented in Java: Java Remote Method Invocation*

4.2 Architettura generale del Provider RMI

Lo schema a blocchi di un provider realizzato tramite RMI è il seguente:

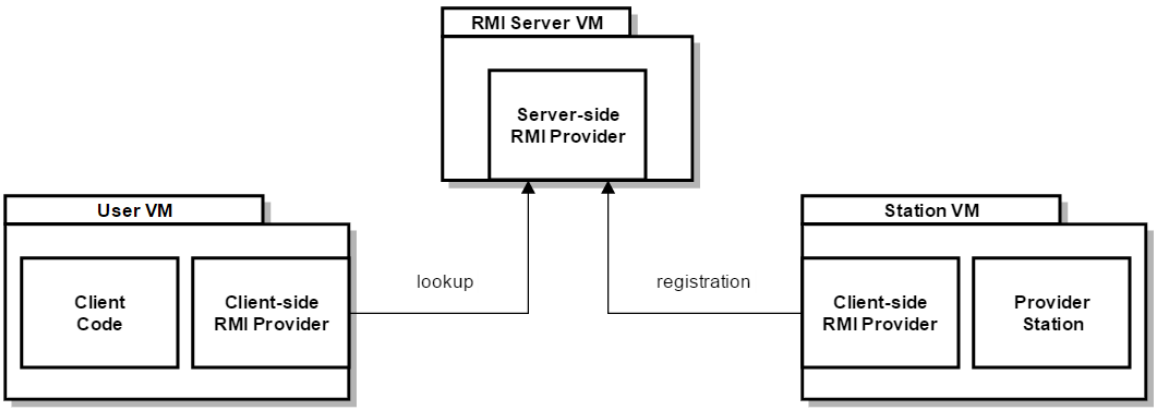


Figura 22 Schema a blocchi del Provider RMI

Si prevede che per ogni utilizzatore del servizio di naming, che sia una Station o un client, sia presente in locale un oggetto che espone i metodi definiti nell'interfaccia `Provider`. Questo oggetto si occuperà delle operazioni necessarie a rendere possibile l'esportazione e l'importazione attraverso RMI e dunque comunicherà con un naming server centralizzato presente come nodo della rete.

4.3 Vincoli imposti da RMI e strategia di soluzione

L'utilizzo di RMI imporrebbe tuttavia dei vincoli molto restrittivi e la necessità di modificare quanto precedentemente fatto. Come problematiche principali si individuano:

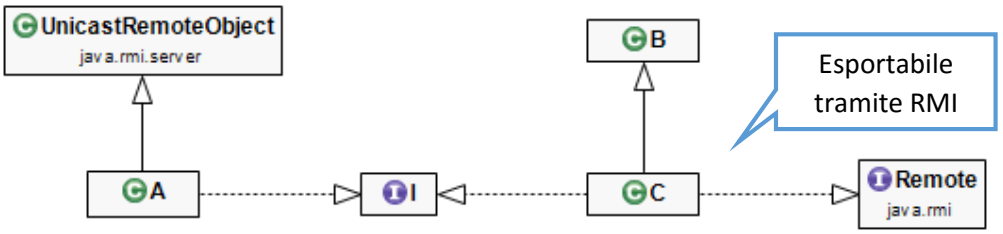
- Ogni classe da esportare deve estendere `UnicastRemoteObject`, imponendo così una modifica alla gerarchia di sensori che ha come radice `AbstractSensor` e alla `AbstractStation`
- Ogni interfaccia da esportare deve estendere `Remote`, con conseguenze simili al punto uno sulle interfacce `Sensor` e `Station`
- I metodi invocabili da remoto devono lanciare `RemoteException`
- Al bootstrap del sistema tutti i nodi devono conoscere le classi e le interfacce che verranno esportate, rendendo così difficile il *deployment* di un nuovo sensore in una rete preesistente

Non è chiaramente pensabile andare a modificare a ritroso le classi già definite per adattarle all'uso con RMI. Agendo in questo modo infatti si "sporcherebbe" un sistema pensato per funzionare con un'implementazione qualsiasi di Provider con vincoli *implementation-specific* di RMI

La soluzione trovata fa uso di funzionalità avanzate di RMI e Java per sottrarsi a questi limiti[24]. Di seguito sono descritti i diversi *step* necessari.

Estendere `UnicastRemoteObject` permetterebbe alle istanze degli oggetti di essere automaticamente esportati, ma come già detti non è un'opzione accettabile. Alternativamente è possibile occuparsi esplicitamente dell'esportazione utilizzando i metodi statici della classe `UnicastRemoteObject`, in particolare `exportObject(Object,int)`. In Figura 23 le differenze:

[24] http://www.javaworld.com/article/2076234/soa/get-smart-with-proxies-and-rmi.html




```
// created and exported automatically
A a = new A();

// created and exported explicitly
C c = new C();
UnicastRemoteObject.exportObject(c, 0);
```

Figura 23 Esportazione di una sottoclasse di UnicastRemoteObject e di una classe non in gerarchia

Esportare i sensori veri e propri ha così successo. Tuttavia, importando l'oggetto con RMI su di un client non è possibile invocare su di esso i metodi definiti nelle interfacce, sono state "perse" nell'esportazione perché non estendevano Remote. Si decide quindi di compilare a *runtime* delle interfacce basate su di esse che estendano anche Remote. In questo modo l'oggetto da esportare non risulta più il sensore, ma un proxy che esponga l'interfaccia remotizzata e deleghi le chiamate al sensore (Figura 24Figura 24).

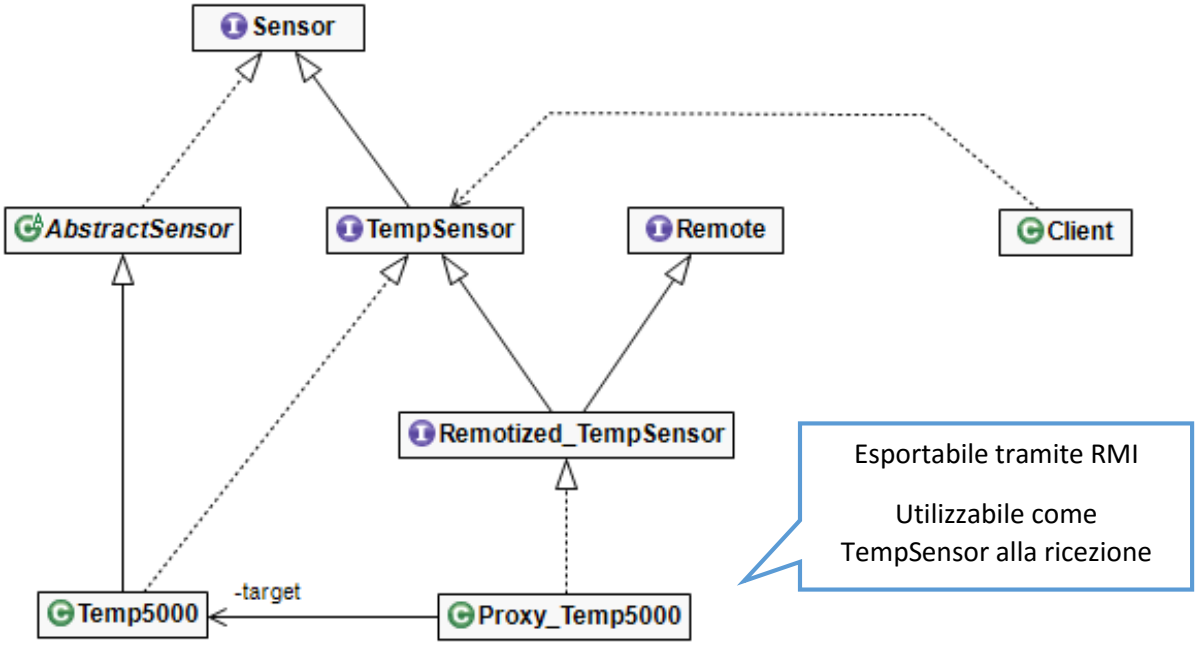


Figura 24 Diagramma dei proxy utilizzati per la remotizzazione

Nel momento in cui il provider RMI riceve una richiesta di registrazione di un sensore da parte della station si occupa di:

1. Generare e compilare le interfacce remotizzate sulla base di quelle esposte dal sensore
2. Creare un proxy che espone queste interfacce e incapsula il sensore
3. Esportare il proxy appena creato tramite `UnicastRemoteObjcet.exportObject`
4. Registrare il proxy presso un `rmiregistry` o un altro servizio di nomi

Il client potrà in maniera trasparente utilizzare l'oggetto ricevuto come un TempSensor, ignorando di star lavorando con un proxy che espone una sottointerfaccia di quella da lui realmente richiesta.

Un ultimo problema che si aggiunge a quelli sopra descritti è la gestione dei listener. In locale, un'entità interessata agli eventi di cambiamento di stato di un sensore può creare un'implementazione dell'interfaccia `StateListener` e passarla al metodo `addListener` del sensore. In seguito il sensore invocherà il metodo `onStateChange` su tutti i listener registrati.
Tuttavia, in un sistema distribuito realizzato con RMI è necessario che l'istanza di `StateListener` venga esportata per poter effettuare un callback su di essa. Quanto fatto per l'esportazione dei sensori deve essere ripetuto per gli `StateListener`, se non che stavolta non è possibile far effettuare le operazioni necessarie al provider locale. Infatti l'aggiunta di un listener viene effettuata invocando `addListener` direttamente sul sensore (propriamente lo stub del sensore che RMI ha deserializzato). Per mantenere questo comportamento è necessario intercettare l'invocazione del metodo sullo stub, creare una versione esportabile dello `StateListener` e utilizzarla per proseguire con l'invocazione (Figura 32 e Figura 33 in appendice).
Riassumendo, vi sono due importanti meccanismi da realizzare: la creazione di un proxy esportabile tramite RMI a partire da un sensore e la creazione di un proxy che remotizza alcuni parametri di invocazione dei suoi metodi a partire dallo stub RMI ricevuto. Insieme essi costituiscono una sorta di *layer* aggiuntivo di remotizzazione.

Realizzare i passi sopra descritti comporta la compilazione di codice Java a *runtime* utilizzando la JavaCompiler API, oltre che la creazione dinamica di proxy tramite la Proxy API. Per poter utilizzar le classi compilate all'interno della JVM locale e da parte dei client remoti è inoltre necessario comprendere il meccanismo del classloading e della funzionalità di codebase remoto di RMI.

Di seguito si presentano in maniera concisa i vari strumenti utilizzati.

4.4 Concetti chiave per la realizzazione del Provider RMI

4.4.1 Class loading

Il concetto di *class loader*, pietra d'angolo della Java Virtual Machine, descrive il processo di caricamento delle informazioni che descrivono una classe. Grazie al concetto di *class loader* Java è in grado di astrarre dai concetti di file e filesystem durante l'esecuzione, permettendo inoltre di caricare le classi da qualsiasi location[25].

Il caricamento di una classe deve essere effettuato in due situazioni: quando nel bytecode viene instanziato un nuovo oggetto appartenente a quella classe tramite l'istruzione new oppure quando nel bytecode viene fatto un riferimento statico alla classe. Dunque il caricamento di una classe avviene in corrispondenza di precise istruzioni nel bytecode di un'altra classe, ma chi carica la prima classe?

Il problema del *bootstrap* della JVM è risolto da un particolare *class loader* che carica le classi di base da una location fidata per la quale non è necessario un processo di verifica. Tra le classi caricate al boot c'è anche il *class loader* principale dell'Application, utilizzato per caricare le classi dai percorsi e dai jar definiti nel classpath. (N.B. il *bootstrap* di una virtual machine è ben più complesso, ma ai fini di questa tesi se ne è semplificata la descrizione)

Quanti *class loader* ci sono quindi? E perché averne più di uno?
Oltre ai *class loader* fin qui descritti è possibile definire dei *class loader* custom e utilizzarli a *runtime* per caricare altre classi. Ad esempio, le Applet fanno uso di un `AppletClassLoader` per caricare le classi necessarie da una repository web utilizzando il protocollo HTTP.

[25] http://www.javaworld.com/article/2077260/learn-java/learn-java-the-basics-of-java-class-loaders.html

Nel nostro sistema verrà utilizzato un *class loader* per caricare classi compilate a *runtime* da una cartella del filesystem locale e RMI utilizza un *class loader* per caricare classi da codebase remoti durante la deserializzazione degli oggetti riferiti.

È importante notare che per la JVM due classi identiche caricate da *class loader* diversi non sono compatibili ed il tentativo di casting di un'istanza da una classe all'altra fallisce. Per questo motivo i *class loader* sono tra loro in gerarchia e alla creazione di un nuovo *class loader* è necessario specificarne il parent. Ogni *class loader*, se non diversamente istruito, al caricamento di una classe delega la richiesta al parent e si occupa del caricamento solo se tra i suoi antenati nessuno ha mai caricato quella classe (Figura 25).

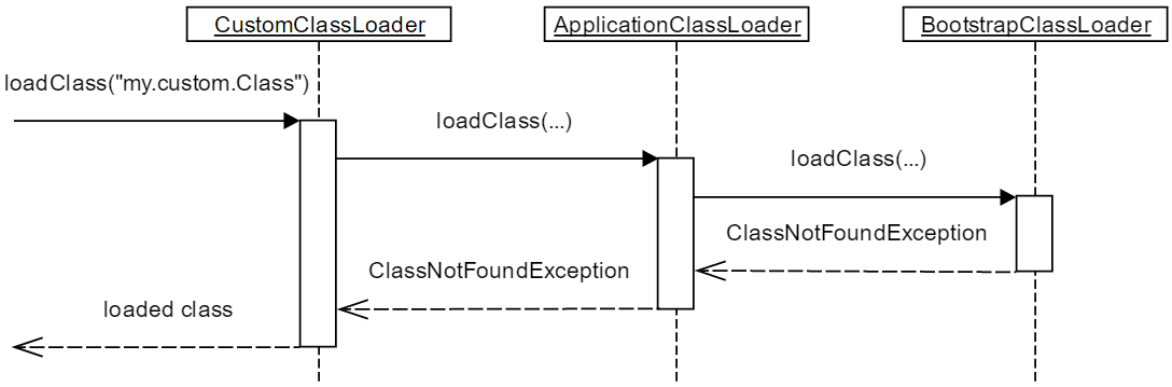


Figura 25 Sequenza di delega tra class loader in gerarchia

In questo modo è garantita la compatibilità tra due classi caricate da *class loader* diversi, purché in gerarchia.

```
// assuming /path/to/extension.jar is not on the classpath
ClassLoader cl1 = new URLClassLoader(new URL[] {
                            new URL("file:/path/to/extension.jar")});
ClassLoader cl2 = new URLClassLoader(new URL[] {
                            new URL("file:/path/to/extension.jar")});
Class c1 = cl1.loadClass("my.custom.Class");
Class c2 = cl2.loadClass("my.custom.Class");
c2.isAssignableFrom(c1); // false
```

4.4.2 Compilazione e caricamento di classi Java a runtime

La compilazione di classi a *runtime* avviene tramite la Java Compiler API introdotta a partire dalla versione 6 di Java[26]. Le responsabilità di generare il codice sorgente, compilarlo e caricarlo sono suddivise tra le seguenti classi (Figura 26).

[26] http://www.beyondlinux.com/2011/07/20/3-steps-to-dynamically-compile-instantiate-and-run-a-java-class/

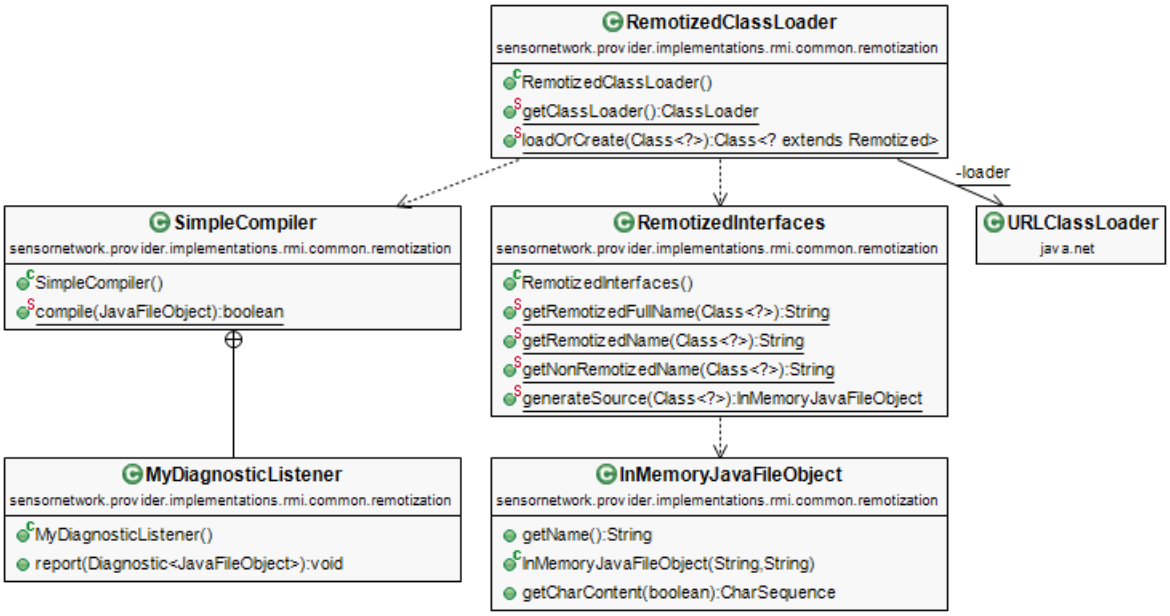


Figura 26 Le classi deputate a compilazione e loading dinamici di interfacce remotizzate

Il *class loader* `RemotizedClassLoader` si occupa di caricare o generare le interfacce remotizzate su richiesta del meccanismo di creazione dei Proxy. Se l'interfaccia è già stata generata essa viene reperita tramite `l'URLClassLoader` contenuto al suo interno. Altrimenti si richiede alla classe statica `RemotizedInterfaces` di generare il codice sorgente della classe sotto forma di un `InMemoryJavaFileObject`. Il sorgente viene quindi compilato da parte del `SingleCompiler` che fa uso della classe innestata `MyDiagnosticListener` per notificare eventuali situazioni di errore. Le classi compilate vengono poste in una directory temporanea creata con `java.nio.file.Files.createTempDirectory`, cartella inclusa tra i percorsi di ricerca del *class loader*.

È importante notare che la compilazione dinamica di classi a *runtime* fa uso di feature presenti solamente nell'installazione di un Java Developement Kit e non di un Java Runtime Environment. Sarà dunque necessario provvedere a fornire un ambiente JDK se si vuole utilizzare il provider realizzato con RMI.

4.4.3 Creazione dinamica di Proxy

Come definizione generale, il *pattern* Proxy prevede che tutte le chiamate dirette verso un oggetto vengano indirizzate verso un oggetto intermedio, il proxy appunto, che delega in maniera trasparente o meno all'oggetto vero e proprio.
In base alla ragione di utilizzo, il pattern prende nomi più specializzati, tra i più comuni: *access* proxy, *façade*, *virtual* proxy o *remote* proxy. Un *access* proxy è usato per imporre una politica di sicurezza nell'accesso ai dati. Una *façade* è una singola interfaccia che nasconde oggetti multipli sottostanti. Un *virtual* proxy si usa per ottenere un caricamento *lazy* o *just in time* dell'oggetto referenziato quando tale operazione risulta costosa, come nel caso di apertura di una connessione verso un database. Un *remote* proxy nasconde l'origine remota dell'oggetto con cui intente comunicare, in particolare questo tipo di proxy è utilizzato da RMI.

In ambiente Java il *pattern* proxy è stato largamente utilizzato fino dalla versione 1. Ma partire da Java 3 è stata introdotta una API per la creazione dinamica di Proxy che permette ancor più flessibilità e libertà. I tipi fondamentali sono la classe `java.lang.reflect.Proxy` e l'interfaccia `java.lang.reflect.InvocationHandler`.

La classe Proxy contiene metodi statici per creare proxy a *runtime* ed è anche la superclasse di tutti i proxy dinamici così creati. Il metodo principale è:

```
Object newProxyInstance(ClassLoader, Class<?>[], InvocationHandler )
```

L'oggetto restituito implementa tutte le interfacce passate al metodo come secondo parametro e delega qualsiasi invocazione all'`InvocationHandler` passato come terzo parametro. Affinché la creazione abbia successo, il `ClassLoader` passato come primo parametro deve essere in grado di risolvere le suddette interfacce.

L'interfaccia `InvocationHandler` viene utilizzata per creare un intercettore per tutte le invocazioni effettuate sul proxy. È un'interfaccia funzionale che definisce un unico metodo:

```
Object invoke(Object proxy, Method method, Object[] args)
```

Il primo parametro è l'oggetto sul quale è stato chiamato il metodo, il secondo parametro è la descrizione del metodo chiamato (nome, numero di parametri, valore di ritorno, classe di definizione ecc.) e il terzo l'elenco degli argomenti. Definendo il corpo di questo metodo è possibile realizzare una vasta gamma di comportamenti: dal semplice logging di ogni invocazione, al *pre-processing* degli argomenti. In seguito si illustrerà come questo meccanismo viene impiegato dal provider RMI.

4.4.4 Codebase remoto e SecurityManager

Una delle maggiori feature di Java RMI è la possibilità di scaricare automaticamente le classi da repository remote di codice dette codebase[27]. In questo modo si elimina la necessità di avere disponibili in locale tutte le classi prima dell'esecuzione. Se per sistemi chiusi questo implica solamente un *deployment* più semplice, per sistemi che vogliono essere aperti a sviluppi futuri o che utilizzano la compilazione a *runtime* questa feature diventa essenziale.

Il concetto di codebase rimanda in parte al concetto di *class loader* introdotto in precedenza. Rappresenta infatti una location tipicamente remota alla quale i *class loader* di RMI possono fare riferimento per scaricare le classi necessarie a deserializzare gli oggetti ricevuti. Ma nello specifico, quali oggetti vengono ricevuti?

Usando RMI le applicazioni possono creare oggetti remoti che accettano invocazioni di metodi da clienti su altre JVM. Per far sì che un client possa invocare un metodo, egli deve avere un modo per comunicare con l'oggetto remoto. Piuttosto che programmare il cliente affinché sappia comunicare con l'oggetto remoto, Java RMI utilizza classi speciale denominate stub che vengono fatte scaricare al cliente. Essi non sono altro che proxy per l'oggetto remoto, utilizzabili come se fossero l'oggetto originale, ma che nascondono al loro interno la logica di comunicazione con protocollo RMI. È evidente dunque che le definizioni degli stub devono essere scaricate dinamicamente da un codebase remoto.

Un'altra situazione per la quale è necessario utilizzare un codebase remoto risulta evidente dall'esempio seguente (Figura 27).

[27] http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/codebase.html

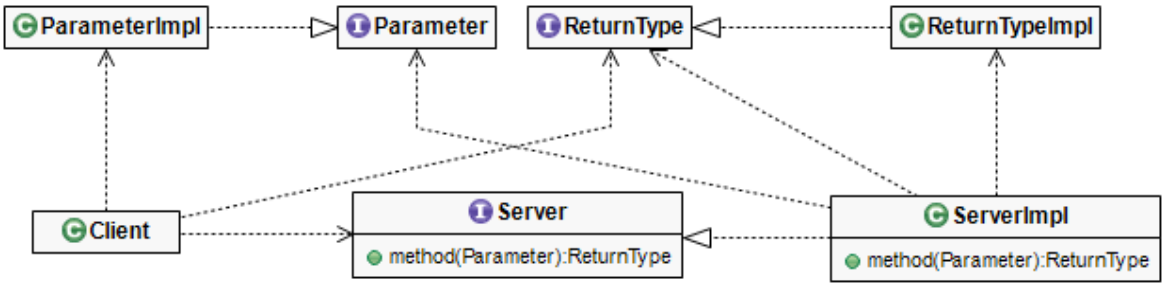


Figura 27 Esempio di applicazione della feature di codebase remoto di RMI

L'interfaccia `Server` espone un metodo che prende come argomento un'istanza di `Parameter` e restituisce un'istanza di `ReturnType`, anch'essi definiti come interfacce. Affinché il client compili è sufficiente rendere disponibili le tre interfacce. Durante l'esecuzione il client invoca il metodo del server passando come parametro un oggetto di tipo `ParameterImpl`. La chiamata avviene sullo stub locale ottenuto da RMI, il quale si occupa di serializzare il parametro e inviarlo al server. Affinché il parametro possa essere deserializzato lato server, dove la definizione di `ParameterImpl` non è nota, è necessario scaricare dinamicamente questa classe da un codebase situato sul client. Viceversa, quando il server restituisce come valore di ritorno un'istanza di `ReturnTypeImpl`, il client deve scaricare la definizione di questa classe prima di poterla deserializzare.

La proprietà `java.rmi.server.codebase` rappresenta una o più location dalle quali è possibile scaricare le definizioni degli oggetti esportati dalla JVM corrente. Utilizza come formato una lista di URL separati da spazi, essi tipicamente fanno riferimento a file locali, a server HTTP o ftp.
Tutte le classi esportate vengono annotate con il contenuto di questa proprietà. L'annotazione viene consultata al momento della deserializzazione se la definizione della classe non viene trovata nel classpath locale o nei codebase specificati dalla stessa proprietà sulla JVM ricevente.

Poiché il caricamento di classi da codebase remoti può introdurre delle brecce nella sicurezza delle applicazioni, è possibile disabilitare il caricamento automatico delle classi dalle location specificate tramite le annotazioni. Con la proprietà `java.rmi.server.useCodebaseOnly` settata a `true` è possibile caricare le classi unicamente dal classpath o dai codebase specificati dalla proprietà `java.rmi.server.codebase` della JVM locale. Nel contesto di questo progetto tuttavia è necessario permettere il download dai codebase specificati tramite annotazione, in quanto non è possibile configurare a priori la virtual machine del client con gli URL di tutti i codebase remoti presenti o futuri.

Per garantire comunque la sicurezza in queste situazioni è possibile fare uso della funzionalità di `SecurityManager` di Java. Un `SecurityManager` controlla le azioni compiute dalle istanze e previene l'esecuzione di azioni *unsafe* lanciando una `SecurityException`. È possibile configurare il `SecurityManager` al momento della creazione tramite file `.policy` nei quali specificare quali azioni considerare *safe* e quali no. Il livello di controllo offerto è molto alto in quanto è possibile garantire e revocare permessi in base all'origine della classe. Nel caso del sistema di sensori è ragionevole che alle classi caricate da RMI vengano concessi i premessi necessari alla comunicazione TCP, ma vengano revocati permessi potenzialmente dannosi come l'accesso al filesystem.

```
grant {
  permission java.security.AllPermission;
};

grant codeBase "http://*" {
  permission java.net.SocketPermission
"*","accept,connect,listen,resolve";
};
```

4.5 Implementazione del Provider RMI

Con riferimento allo schema a blocchi del Provider RMI (Figura 28) per la suddivisione client-side e server-side, vengono qui elencate le classi del package RmiProvider.jar:

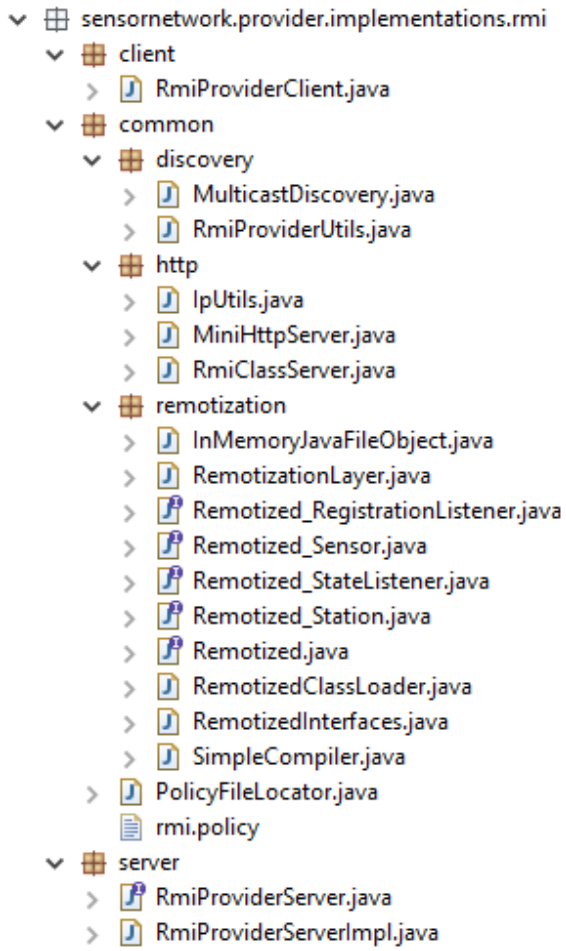


Figura 28 Package RmiProvider.jar

La classe che effettivamente implementa l'interfaccia `Provider` è `RmiProviderClient` (Figura 29), essa è pensata per comunicare con il cliente e servire le richieste del servizio di naming. Al suo interno fa uso della classe `RemotizationLayer` per generare i proxy dei sensori esportabili tramite RMI (`mySkel`) e i proxy che remotizzano i parametri come `StateListener` prima di invocare metodi sugli stub RMI (`myStub`).

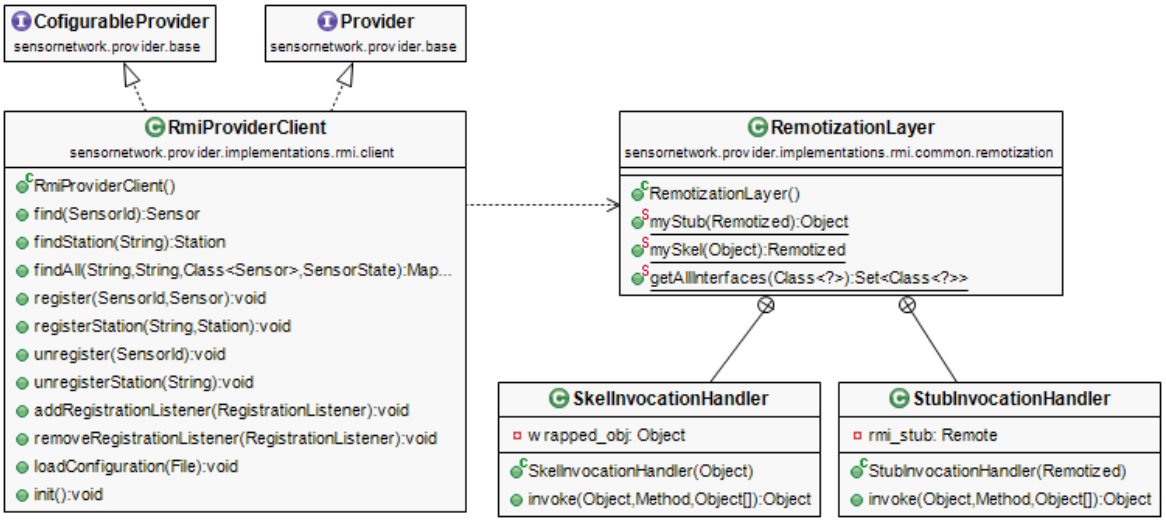


Figura 29 La classe RmiProviderClient e le sue dipendenze principali

All'interno dei metodi di `RemotizationLayer` si fa uso del *class loader* `RemotizedClassLoader`, il cui scopo è provvedere al caricamento dei tipi necessari all'esportazione. Nello specifico, se esso non trova la definizione di un'interfaccia remotizzata la compila sul momento facendo uso di un `SimpleCompiler` (Figura 26).

All'interno del package `sensornetwork.provider.implementations.rmi.common.http` trovano posto le classi necessarie al servizio di codebase remoto. In particolare la classe `RmiClassServer` implementa un web server in grado di fornire le definizioni delle classi ai client. Le richieste vengono gestite da un `MiniHttpServer` che in maniera leggera e naïve risponde alle specifiche di HTTP 1.1 (Figura 30).

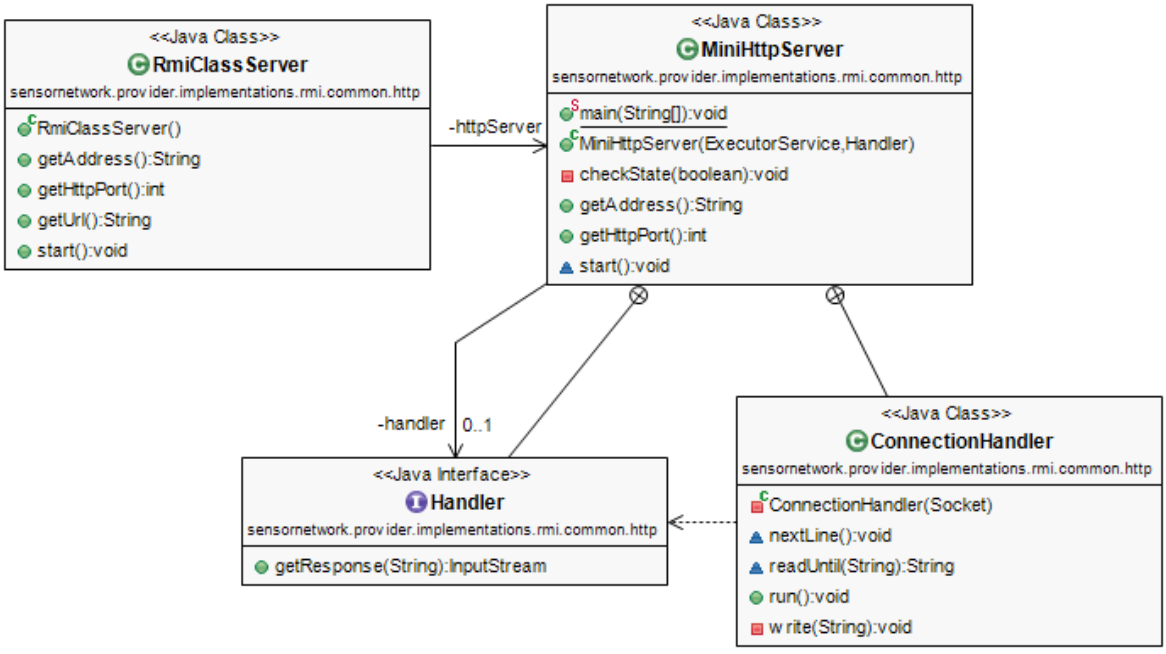


Figura 30 Le classi deputate al servizio di codebase remoto

Al cuore del servizio di naming offerto da questa implementazione delle API di Provider sono le classi contenute in `sensornetwork.provider.implementations.rmi.server` (Figura 31). Esse rappresentano il registro in cui i sensori esportati vengono conservati insieme al nome logico ad esso associato. È a questo registro che fanno riferimento gli `RmiProviderClient` per

effettuare lookup e query. L'interfaccia del servizio è contenuta in `RmiProviderServer` e ricalca l'interfaccia Provider a livello di metodi, con l'unica modifica di accettare e restituire solamente le versioni remotizzate degli oggetti (gli stub RMI ottenuti esportando gli stub esportabili creati da `RemotizationLayer.mySkel`).

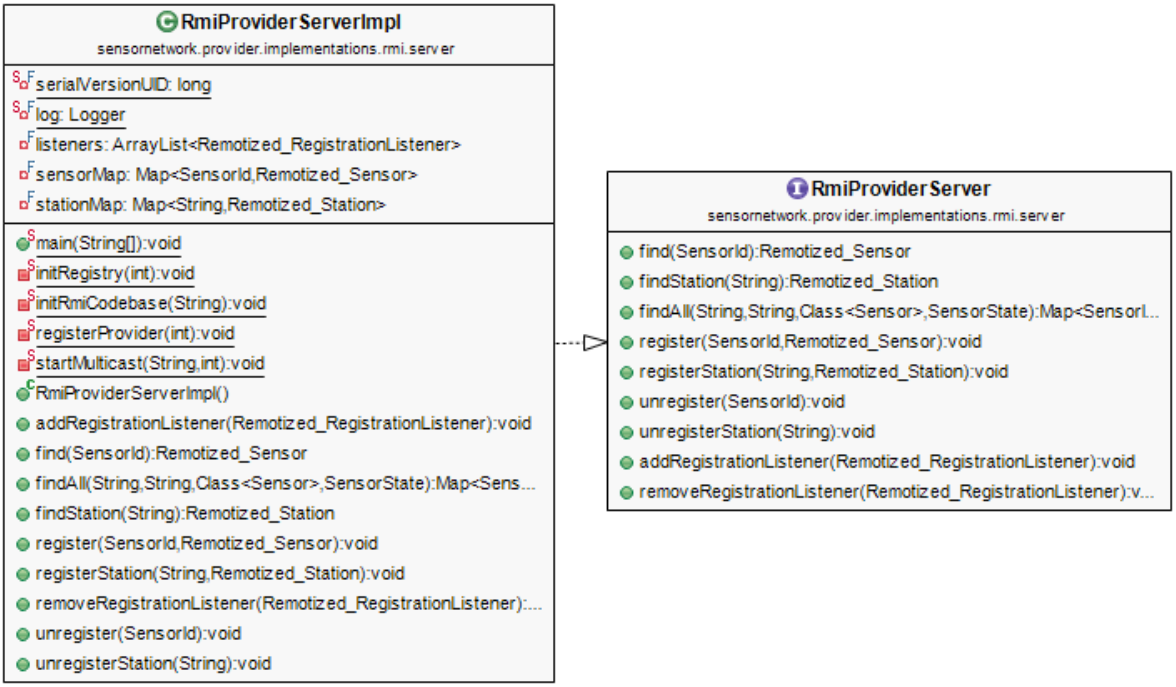


Figura 31 Le classi server-side del Provider RMI

Infine, è stato valutato utile inserire un servizio di *discovery* del server all'interno della rete locale. Ogni `RmiProviderClient` deve infatti ottenere un riferimento al server prima di ogni altra operazione chiamando il metodo statico `Naming.lookup` con l'URL del server:

```
String serverUrl = "rmi://"+host+":"+port+"/RmiProviderServer"
RmiProviderServer server = (RmiProviderServer)
Naming.lookup(serverUrl);
```

Sebbene sia possibile utilizzare il file di configurazione del provider per indicare indirizzo e porta, è verosimile che all'accensione di una Station l'URL del server non sia noto o semplicemente non si voglia doverlo modificare a mano ogni volta.

Il package `sensornetwork.provider.implementations.rmi.common.discovery` realizza un servizio di *discovery* UDP in *multicast* sul gruppo 230.0.0.1. All'accensione il server avvia il servizio `MulticastDiscovery` che è in grado di rispondere con le coordinate del server ad ogni pacchetto *datagram* ricevuto. Ai client sarà sufficiente utilizzare i metodi statici della classe `RmiProviderUtils` per inviare le richieste UDP costruire l'URL da utilizzare nella lookup iniziale.

4.6 Utilizzo del provider RMI

In conclusione alla fase implementativa del progetto si procede a riepilogare gli archivi jar prodotti, le loro funzionalità e le loro relazioni. Si illustrano quindi i passi necessari ad avviare un sistema che utilizzi il Provider RMI e ad effettuare misurazioni attraverso di esso.

Vengono qui elencati gli archivi jar con una breve descrizione del loro contenuto, del loro scopo e delle loro dipendenze:

`CommonUtils.jar`

Contiene le classi di utilità che permettono il logging e l'injection della configurazione all'interno dei sensori

Non ha dipendenze

`SensorStation.jar`

Contiene le classi base necessarie al funzionamento del container Station e dei componenti Sensor, a questo livello non è presente il concetto di sistema distribuito in termini di servizio di naming e accesso remoto, volendo utilizzare la `StandAloneStation` per il testing è necessario fornire a *runtime* gli archivi contenenti le definizioni dei sensori da avviare

Dipende da CommonUtils.jar

`Provider.jar`

Contiene la definizione della API Provider, oltre che una Station in grado di avviare ed utilizzare una realizzazione di Provider per registrare i sensori e renderli disponibili nel distribuito, l'implementazione del provider utilizzato deve essere resa disponibile a *runtime*

Dipende da SensorStation.jar, CommonUtils.jar

`RmiProvider.jar`

Contiene le classi server-side e client-side che implementano la logica di un Provider realizzato tramite RMI

Dipende da Provider.jar, SensorStation.jar, CommonUtils.jar

`SensorInterfaces.jar`

Contiene le interfacce e le strutture che definiscono i sensori di temperatura e di dati astronomici

Dipende da SensorStation.jar

`SensorImplementations.jar`

Contiene le implementazioni dei tre sensori descritti nel capitolo precedente

Dipende da SensorStation.jar, SensorInterfaces.jar e dalle librerie specifiche necessarie ai sensori

`(Client)`

Un utilizzatore del sistema è in grado di ricercare sensori attraverso le API del provider e su di essi comandare delle misurazioni tramite i loro metodi di interfaccia, l'implementazione del provider utilizzato deve essere resa disponibile a *runtime*

Dipende da Provider.jar, SensorStation.jar, SensorInterfaces.jar

Con riferimento ad un Provider realizzato con RMI, un tipico *workflow* di sviluppo, *deployment* e utilizzo di nuovi sensori può essere riassunto dai seguenti passi:

1. Sviluppare i sensori basandosi sull'interfaccia Sensor e sulla classe AbstractSensor producendo l'archivio jar delle implementazioni (dipendente dalle librerie specifiche necessarie ai sensori), l'archivio jar delle interfacce e, se necessari, i file di configurazione `.properties` dei sensori
2. Avviare il server RMI su un nodo qualsiasi della rete, di seguito i messaggi di log prodotti:

```
2016-07-05 12:02:51 INFO
    sensornetwork.provider.implementations.rmi.common.http.RmiClassS
erver start: RmiClassServer started on: http://0.0.0.0:51555

2016-07-05 12:02:51 INFO
    sensornetwork.provider.implementations.rmi.server.RmiProviderSer
```

```
verImpl registerProvider: RmiProviderServer registered on the
rmiregistry

2016-07-05 12:02:51 INFO
    sensornetwork.provider.implementations.rmi.common.discovery.Mult
icastDiscovery run: MulticastDiscoveryServer started on
230.0.0.1:5000, will send 192.168.0.16:1099
```

Output prodotto dall'RMI server

3. Configurare una `ProviderStation` tramite XML affinché faccia uso del Provider RMI e carichi i sensori realizzati al punto 1
4. Avviare la `ProviderStation` fornendo a *runtime* le classi relative al Provider RMI, oltre che le interfacce e le classi relative ai sensori desiderati, di seguito un esempio dei messaggi di log prodotti

```
2016-07-05 12:05:13 INFO
    sensornetwork.station.base.AbstractStation validateAndParse:
RmiTempstation.xml is valid against
providerStation2414710902211036637.xsd

2016-07-05 12:05:13 INFO
    sensornetwork.station.base.AbstractStation loadSensors: Caricato
sensore temp5000

2016-07-05 12:05:13 INFO   sensornetwork.sensors.base.AbstractSensor
setState: Temp5000 started

2016-07-05 12:05:13 INFO
    sensornetwork.station.base.AbstractStation
loadSensors: Caricato sensore temp5000bis

2016-07-05 12:05:13 INFO   sensornetwork.sensors.base.AbstractSensor
setState: Temp5000 started

2016-07-05 12:05:13 INFO
    sensornetwork.provider.implementations.rmi.client.RmiProviderCli
ent init: currentHostName: 192.168.0.16

2016-07-05 12:05:13 INFO
    sensornetwork.provider.implementations.rmi.common.http.RmiClassS
erver start: RmiClassServer started on: http://0.0.0.0:51570

2016-07-05 12:05:13 INFO
    sensornetwork.provider.implementations.rmi.common.discovery.RmiP
roviderUtils findServerUrl: Search for provider started on
230.0.0.1:5000

2016-07-05 12:05:13 INFO
    sensornetwork.provider.implementations.rmi.client.RmiProviderCli
ent init: Looking up server on:
rmi://192.168.0.16:1099/RmiProviderServer

2016-07-05 12:05:13 INFO
    sensornetwork.provider.implementations.rmi.client.RmiProviderCli
ent init: Connessione al server completata

2016-07-05 12:05:17 INFO
    sensornetwork.provider.implementations.rmi.common.remotization.S
impleCompiler compile: Compilation succeeded:
```

```
sensornetwork.provider.implementations.rmi.common.remotization.Remot
ized_TempSensor

2016-07-05 12:05:17 INFO
    sensornetwork.provider.implementations.rmi.common.http.RmiClassS
erver lambda$0: RmiClassServer richiesto url:
/sensornetwork/provider/implementations/rmi/common/remotization/Remo
tized_TempSensor.class [OK]

2016-07-05 12:05:17 INFO
    sensornetwork.provider.implementations.rmi.common.http.RmiClassS
erver lambda$0: RmiClassServer richiesto url:
/sensornetwork/provider/implementations/rmi/common/remotization/Remo
tized_TempSensor.class [OK]

2016-07-05 12:05:17 INFO
    sensornetwork.provider.implementations.rmi.common.http.RmiClassS
erver lambda$0: RmiClassServer richiesto url:
/sensornetwork/sensor/interfaces/TempSensor.class [OK]
```

Output prodotto dalla ProviderStation

```
2016-07-05 12:05:13 INFO
    sensornetwork.provider.implementations.rmi.server.RmiProviderSer
verImpl registerStation: Registered station Mocks Station, 0
listeners to notify
    stub:       com.sun.proxy.$Proxy2
    annotation: http://192.168.0.16:51570/

2016-07-05 12:05:17 INFO
    sensornetwork.provider.implementations.rmi.server.RmiProviderSer
verImpl register: Registered: temp5000@Mocks Station, 0 listeners to
notify
    stub:       com.sun.proxy.$Proxy3
    annotation: http://192.168.0.16:51570/
    interfaces: TempSensor

2016-07-05 12:05:17 INFO
    sensornetwork.provider.implementations.rmi.server.RmiProviderSer
verImpl register: Registered: temp5000bis@Mocks Station, 0 listeners
to notify
    stub:       com.sun.proxy.$Proxy3
    annotation: http://192.168.0.16:51570/
    interfaces: TempSensor
```

Output prodotto dall'RMI server

5. Avviare il client fornendo a *runtime* le classi relative al Provider RMI

```
2016-07-05 12:11:42 INFO
    sensornetwork.provider.implementations.rmi.server.RmiProviderSer
verImpl addRegistrationListener: Added registration listener

2016-07-05 12:11:42 INFO
    sensornetwork.provider.implementations.rmi.server.RmiProviderSer
verImpl find: Requested: temp5000@Mocks Station

2016-07-05 12:11:43 INFO
    sensornetwork.provider.implementations.rmi.server.RmiProviderSer
verImpl findStation: Requested station: Mocks Station
```

```
2016-07-05 12:11:44 INFO
    sensornetwork.provider.implementations.rmi.server.RmiProviderSer
verImpl findAll: Found 2 sensors matching query (null, Mocks
Station,null,RUNNING)
```

Output prodotto dall'RMI server

```
2016-07-05 12:11:42 INFO
    sensornetwork.provider.implementations.rmi.common.http.RmiClassS
erver lambda$0: RmiClassServer richiesto url:
/sensornetwork/provider/implementations/rmi/common/remotization/Remo
tized_TempSensor.class [OK]

2016-07-05 12:11:43 INFO   sensornetwork.sensors.base.AbstractSensor
addListener: Temp5000: added listener

2016-07-05 12:11:43 INFO   sensornetwork.sensor.mocks.Temp5000
lambda$0: Measure done

2016-07-05 12:11:43 INFO
    sensornetwork.provider.implementations.rmi.common.remotization.S
impleCompiler compile: Compilation succeeded:
sensornetwork.provider.implementations.rmi.common.remotization.Remot
ized_FutureResult

2016-07-05 12:11:43 INFO
    sensornetwork.provider.implementations.rmi.common.http.RmiClassS
erver lambda$0: RmiClassServer richiesto url:
/sensornetwork/provider/implementations/rmi/common/remotization/Remo
tized_FutureResult.class [OK]

2016-07-05 12:11:43 INFO   sensornetwork.sensors.base.AbstractSensor
setState: Temp5000 stopped
```

Output prodotto dalla ProviderStation

```
2016-07-05 12:22:35 INFO
    sensornetwork.provider.implementations.rmi.client.RmiProviderCli
ent init: currentHostName: 192.168.0.16

2016-07-05 12:22:36 INFO
    sensornetwork.provider.implementations.rmi.common.http.RmiClassS
erver start: RmiClassServer started on: http://0.0.0.0:51691

2016-07-05 12:22:36 INFO
    sensornetwork.provider.implementations.rmi.common.discovery.RmiP
roviderUtils findServerUrl: Search for provider started on
230.0.0.1:5000

2016-07-05 12:22:36 INFO
    sensornetwork.provider.implementations.rmi.client.RmiProviderCli
ent init: Looking up server on:
rmi://192.168.0.16:1099/RmiProviderServer

2016-07-05 12:22:36 INFO
    sensornetwork.provider.implementations.rmi.client.RmiProviderCli
ent init: Connessione al server completata

...
```

Output prodotto dal client

4.7 Tecnologie alternative per il Provider

A completamento del progetto di tesi si è realizzata un'implementazione delle API del Provider utilizzando la tecnologia Java RMI. Tuttavia, nella comunità informatica è ormai considerata una tecnologia *legacy*, superata da numerose alternative come CORBA o sostituita da approcci differenti come *web services* basati su SOAP o REST.

Sarebbe interessante vedere come queste tecnologie possano essere utilizzare per implementare un Provider e quale di esse comporti i maggiori benefici al sistema.

4.7.1 CORBA

CORBA (Common Object Request Broker Architecture) è una specifica che definisce come oggetti distribuiti possono interoperare. Nato come soluzione C++ ha guadagnato popolarità con l'avvento di Java e del World Wide Web.

Oggetti CORBA possono essere scritti in qualsiasi linguaggio di programmazione supportato come C, C++, Java, Ada o Smalltalk. Questo lo rende molto più portabile di RMI, potendosi integrare sia con oggetti scritti in linguaggi *legacy* non più utilizzati, sia con linguaggi ancora da venire che soddisferanno le specifiche CORBA. L'indipendenza dal linguaggio è ottenuta tramite la definizione di interfacce per l'accesso agli oggetti in Interface Definition Language. In questo modo, per ogni linguaggio specifico è sufficiente la presenza di un "ponte" in grado di convertire da e verso IDL.

Sarebbe così possibile integrare nella rete dei sensori scritti in altri linguaggi, estendendo le possibilità del sistema e la libertà degli sviluppatori.

Su oggetti remoti CORBA è possibile invocare metodi in maniera simile a RMI, utilizzando però come parametri di ingresso e di uscita unicamente dati primitivi e strutture. Non è quindi possibile passare oggetti Java né codice eseguibile, come avviene in RMI scambiando le definizioni di classi sconosciute alla virtual machine ricevente. Per questo aspetto le potenzialità di RMI risultano superiori a CORBA.

Lo scambio di dati tra clienti e servitori in CORBA avviene tramite Object Request Brokers. Tra di essi, gli ORB comunicano tramite protocolli Inter-ORB Protocol, tra i quali si annovera l'Internet IOP in grado di utilizzare *stream* TCP.

CORBA dunque appare come una versione avanzata di RMI che offre numerose caratteristiche aggiuntive per l'invocazione remota di metodi[28]. Tuttavia, nel tempo si è osservata una decadenza della popolarità di tutti i servizi di RPC, che siano essi RMI, CORBA, DCOM o .NET Remoting[29]. Il problema fondamentale è che i sistemi distribuiti sono profondamente diversi da quelli locali. Il concetto di base di RPC è riuscire a nascondere queste differenze e rendere le invocazioni remote identiche a quelle invocazioni locali. Ma le differenze esistono e come tali vanno riconosciute e affrontate.

[28] http://www.javacoffeebreak.com/articles/rmi_corba/
[29] http://stackoverflow.com/questions/3835785/why-has-corba-lost-popularity

Bill Joy, Tom Lyon, L. Peter Deutsch e James Gosling, ingegneri presso la Sun Microsystems durante gli anni Novanta, identificarono quelle che sarebbero diventate note come *"The Eight Fallacies of Distributed Computing"*[30]:

1. La rete è affidabile
2. La latenza è nulla
3. La banda è infinita
4. La rete è in sicurezza
5. La topologia non cambia
6. C'è un solo amministratore
7. Il costo di trasporto è nullo
8. La rete è omogenea

Considerare ingenuamente vere queste assunzioni nel contesto di un sistema distribuito porta inevitabilmente a grandi problemi di sviluppo, se non addirittura al fallimento del progetto, nel momento in cui esse si rivelano false.

Nel caso particolare di CORBA, le maggiori critiche portate a questa architettura riguardano il tentativo di realizzare un'indipendenza dalla locazione fisica degli oggetti riferiti e la complessità della definizione e uso dell'architettura[31].

In CORBA è stato fatto il tentativo di rendere uniforme l'accesso locale e l'accesso remoto. Non potendosi liberare dai vincoli imposti dalla complessità di un accesso remoto, è stato deciso di estendere questa complessità anche agli accessi locali. In questo modo un oggetto locale, residente nello stesso spazio di indirizzamento e accessibile tramite una semplice chiamata a funzione, deve essere trattato nello stesso modo di un oggetto remoto, obbligando così a gestire tutta una serie di situazioni di errore che non possono verificarsi per chiamate locali.
Questo desiderio di trasparenza inoltre nasconde inevitabilmente all'utilizzatore la locazione di un oggetto. Rendendo così impossibile scegliere un'appropriata strategia di uso tra una chiamata locale (1 µs di latenza e valore di ritorno garantito) e una chiamata remota (1 s di latenza, con rischio di errori di rete dei quali si riceve notifica dopo un lungo time out).

Dal momento che l'architettura CORBA prevede l'utilizzo di IDL per definire le interfacce, gli sviluppatori sono obbligati a conoscere un linguaggio in più verso il quale devono essere in grado di tradurre le strutture dati necessarie alla *business logic* della loro applicazione, procedimento che risulta forzato, non certo semplice e non sempre attuabile.
Inoltre, benché la specifica sia open, diversi *vendor* hanno commercializzato il loro protocollo di comunicazione inter-ORB. Ogni implementazione presenta delle peculiarità che è necessario apprendere prima di poter sviluppare un sistema con esse. Per di più, risultando proprietari, i protocolli presentano grandi difficoltà di integrazione.

Ulteriore motivo che ha portato la comunità informatica ad orientarsi verso altri tipi architetture è la diffusione di firewall ad ogni livello delle architetture di rete. In caso di politiche molto restrittive è possibile che i *firewall* blocchino ogni tipo di comunicazione, permettendo unicamente l'accesso a servizi HTTP attraverso la porta 80. Da qui è derivato l'impulso che ha dato ai *web services* la loro grande popolarità in anni recenti.

[30] https://blogs.oracle.com/jag/resource/Fallacies.html
[31] https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture#Problems_and_criticism

4.7.2 Web Services

Un *web service* è un qualsiasi servizio offerto da un dispositivo a un altro, la comunicazione tra i quali è realizzata attraverso il World Wide Web. In un *web service* vengono utilizzate tecnologie come HTTP, progettata inizialmente per la comunicazione uomo-macchina, per attuare lo scambio di dati macchina-macchina in formati come XML o JSON[32].

In un documento del 2004, il W3C ha identificato due tipi di servizi:

- *Arbitrary Web Services*, nei quali il servizio espone un set arbitrario di operazioni
- *REST-compliant Web Services*, il cui scopo primario è la manipolazione della rappresentazione web di risorse utilizzando un set di istruzioni *stateless* uniformi

Esempio della prima tipologia è la tecnologia Simple Object Access Protocol (SOAP), sviluppata da Microsoft e standardizzata dalla Internet Engineering Task Force. SOAP è basata esclusivamente su XML per lo scambio di messaggi e la descrizione dei servizi. Contrariamente a modelli come CORBA che introducono complessità anche per operazioni semplici, SOAP è modulare e si presta facilmente ad espansioni là dove necessario.

Contrariamente a quanto suggerito dal nome, in SOAP l'XML utilizzato per scambiare messaggi e descrivere servizi può risultare laborioso da gestire. Fortunatamente per molti linguaggi esistono librerie che ne semplificano l'uso, addirittura nei linguaggi .NET la gestione dell'XML è completamente automatica. Questa possibilità è data dal Web Services Description Language (WSDL). File in questo formato offrono la descrizione completa di un servizio web, in modo tale che creando un riferimento al servizio tramite un IDE, l'intero processo può essere automatizzato.

Altra feature interessante di SOAP è che non è necessariamente basato su HTTP come protocollo di trasporto. Ad esempio, messaggi SOAP possono essere scambiati anche via mail con protocollo SMTP.

Tuttavia, una delle critiche mosse a SOAP è la grande quantità di dati scambiati per ottenere un servizio. Diversamente da RMI o CORBA, in cui la comunicazione è binaria, SOAP scambia file XML che presentano un grande *overhead* in termini di volume di dati scambiati.

Al contrario, API web basate su REST non necessitano di XML per lo scambio di dati e anche per questo motivo stanno recentemente guadagnando grande popolarità. Representational State Transfer, abbreviato REST, è un concetto presentato nel 2000 da Roy Fielding nella sua tesi di dottorato presso l'Università della California a Irvine[33]. REST non è né un protocollo né uno standard, ma una serie di linee guida per creare un'interfaccia di accesso al servizio basata sui principali predicati HTTP.

Ad ognuno dei verbi GET, POST, PUT, DELETE, inizialmente concepiti per avere una certa semantica in HTTP, viene associato un nuovo significato, facilmente comprensibile. Il costo di invio di una richiesta REST corrisponde a quello di una semplice richiesta HTTP. Allo stesso modo la risposta è molto leggera, tipicamente in formato CSV o JSON, al contrario di un formato XML.

Per fare un esempio, vediamo come SOAP e REST potrebbero interrogare sensore di temperatura per ottenerne la misura istantanea. Nel confronto tra le due alternative si valutino la facilità di comprensione della richiesta e la quantità di dati che è necessario scambiare.

[32] http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/

[33] Thomas, F. R. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* University of California, Irvine

La richiesta SOAP potrebbe essere composta dal seguente XML, inviato ad esempio tramite POST HTTP:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
 <soap:body ts="http://www.sensornetwork.com/tempsensor">
  <ts:GetTemperature>
   <ts:Unit>CELSIUS</ts:Unit>
  </ts:GetTemperature>
 </soap:Body>
</soap:Envelope>
```

La stessa query in formato REST potrebbe ridursi a una semplice richiesta GET a:

```
http://www.sensornetwork.com/tempsensor/Temperature/CELSIUS
```

In conclusione di questa breve panoramica sui più conosciuti modelli di comunicazione distribuita si valuta che un provider basato su API REST potrebbe rappresentare una soluzione in grado di eliminare le barriere costituite da diversi linguaggi di programmazione e dalla presenza di *firewall*, oltre che ridurre l'*overhead* associato ai dati scambiati per richieste semplici come la misura della temperatura in una stanza.

5 Possibili sviluppi

In quest'ultimo capitolo si vogliono elencare alcuni dei possibili sviluppi per il sistema. La maggior parte di essi deriva da *concept* presi in considerazione nelle prime fasi di ideazione, ma non realizzati in questo primo prototipo.

5.1 Protezione e sicurezza

In un sistema domotico di una certa complessità non è certo possibile ignorare il tema della sicurezza. Tema che si può declinare principalmente in due ambiti: quello dell'impedire ad agenti esterni di accedere al sistema e ottenere informazioni private (sicurezza) e quello dell'assegnare diversi livelli di privilegi agli agenti del sistema in modo che possano accedere solo alle informazioni a loro necessarie (protezione).

A titolo di esempio per il secondo ambito si può pensare a componenti come il frigorifero intelligente, le cui informazioni sugli alimenti contenuti devono essere disponibile a tutti gli utenti della casa, e ad altri componenti come l'armadio, le cui informazioni sulle preferenze di una persona siano accessibili solo al proprietario dei vestiti.

Nello specifico caso di un provider realizzato con RMI si possono valutare due miglioramenti relativi alla sicurezza. Per primo, utilizzare una `SocketFactory` custom per *socket* SSL invece che la standard *factory* per TCP *socket*, permettendo così di aggiungere un *layer* di sicurezza sulle connessioni utilizzate per la comunicazione tra le virtual machine. In secondo luogo, curare in maniera più fine i permessi concessi alle classi caricate dai diversi codebase per impedire a un'istanza maligna caricata da remoto di eseguire azioni come leggere dal disco o eliminare file.

5.2 Peer-to-peer discovery

In un contesto distribuito, un nodo centrale che fornisce servizi a tutto il sistema risulta essere il maggiore impedimento alla scalabilità del sistema. Nel realizzare il Provider RMI è stato progettato il registro come server unico accessibile da tutta la rete. Se il numero di sensori e stazioni dovesse crescere in maniera significativa questo elemento incorrerebbe in gravi difficoltà in termini di rapidità di risposta, efficienza nell'uso delle connessioni e spazio di memoria utilizzato.

La soluzione potrebbe essere quella di realizzare un registro a sua volta distribuito. In questo caso ogni nodo della rete, cioè ogni Station, mantiene una parte delle associazioni tra oggetti e nomi logici ed è in grado di condividere le associazioni a lei note con altre Station. Al momento di una registrazione sarà sufficiente registrare il sensore prezzo il frammento locale del registro e lasciare che l'informazione di registrazione venga propagata di Station in Station. Al momento invece di una ricerca ogni Station può cominciare interrogando il registro locale e poi in successione le Station vicine fino all'ottenimento di un risultato.

Sfortunatamente, distribuendo il servizio di naming verrebbero a mancare garanzie sul successo di una ricerca. In caso di fallimento di una lookup infatti il sistema non è in grado di determinare se il sensore non esista o semplicemente non sia stato trovato nei frammenti di registri interrogati.

5.3 Framework di Dependency Injection

Per attuare il caricamento dei parametri di configurazione dei sensori è stato realizzato un semplice strumento di *dependency injection*. Ai fini del progetto questa soluzione si è rivelata sufficiente, tuttavia è ragionevole pensare di integrare nel sistema soluzioni più complete e testate da un'ampia community di utilizzatori.

Nel mondo Java sono infatti numerosi i framework di *dependency injection* disponibili, a cominciare da Dagger o il Spring core per nominarne alcuni[34]. L'integrazione di tali strumenti nel sistema delle Station e dei Provider permetterebbe di sostituire il modello attuale di caricamento dei sensori e dell'implementazione del provider in uso. Il vantaggio risultante sarebbe una riduzione della quantità di codice attualmente necessaria a svolgere queste operazioni, con la conseguenza però di introdurre una dipendenza aggiuntiva e di dover adattare il formato dei file di configurazione utilizzati al framework scelto.

[34] https://keyholesoftware.com/2014/02/17/dependency-injection-options-for-java/

Conclusioni

Un maggiordomo intelligente per l'home automation

Le caratteristiche realizzate in questo progetto lo rendono ideale per la realizzazione di sistemi domotici intelligenti ad alta automazione. Tali sistemi potrebbero prendere il ruolo di maggiordomi all'interno delle abitazioni gestendone i consumi energetici, monitorando gli accessi, regolando la temperatura, eccetera.

Infatti, durante la progettazione sono state tenute in grande considerazione le proprietà di autoconfigurazione e gestione remota delle stazioni e dei sensori, essenziali all'autonomia di un sistema di home automation. L'avvio di una stazione non prevede l'intervento manuale, se non per la prima configurazione, ed è possibile istruire il sistema operativo ospitante per lanciare una Station all'accensione. Inoltre, grazie alle API proposte, un maggiordomo intelligente è in grado non solo di comandare misurazioni, ma anche di gestire l'accensione e lo spegnimento di ogni sensore in base alle necessità (ad esempio, è difficile che in estate serva il sensore di congelamento delle tubature).

Nel contesto dell'Università di Bologna e del progetto HomeManager, la ricerca in ambito di home automation fa uso del linguaggio logico Prolog e del suo interprete tuProlog. Al fine di abilitare tuProlog all'uso del *middleware* di sensori sarà necessario sviluppare una libreria ibrida Java-Prolog che funga da "ponte" tra i due paradigmi.

Bibliografia

(s.d.). Tratto da http://apice.unibo.it/xwiki/bin/view/Products/HomeManager

(s.d.). Tratto da https://it.wikipedia.org/wiki/Raspberry_Pi

(s.d.). Tratto da http://java.sun.com/products/jdk/rmi/

(s.d.). Tratto da http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/codebase.html

(s.d.). Tratto da http://apice.unibo.it/xwiki/bin/view/Tuprolog/

(s.d.). Tratto da http://www.beyondlinux.com/2011/07/20/3-steps-to-dynamically-compile-instantiate-and-run-a-java-class/

(s.d.). Tratto da http://www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html

(s.d.). Tratto da https://www.infoq.com/articles/Functional-Style-Callbacks-Using-CompletableFuture

(s.d.). Tratto da http://winterbe.com/posts/2014/03/16/java-8-tutorial/

(s.d.). Tratto da https://github.com/harry1357931/Remote_Method_Invocation-RMI

(s.d.). Tratto da http://www.vogella.com/tutorials/Logging/article.html

(s.d.). Tratto da http://tutorials.jenkov.com/java-logging/logger-hierarchy.html

(s.d.). Tratto da http://www.javaworld.com/article/2076234/soa/get-smart-with-proxies-and-rmi.html

(s.d.). Tratto da http://www.javaworld.com/article/2077260/learn-java/learn-java-the-basics-of-java-class-loaders.html

(s.d.). Tratto da http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/

(s.d.). Tratto da http://stackoverflow.com/questions/3835785/why-has-corba-lost-popularity

(s.d.). Tratto da http://www.javacoffeebreak.com/articles/rmi_corba/

(s.d.). Tratto da https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture#Problems_and_criticism

(s.d.). Tratto da http://www.javaworld.com/article/2077037/soa/corba-meets-java.html

(s.d.). Tratto da https://keyholesoftware.com/2014/02/17/dependency-injection-options-for-java/

(s.d.). Tratto da Wheather Underground: https://www.wunderground.com/weather/api

(s.d.). Tratto da https://github.com/InitialState/grovepi/wiki/Part-2.-DHT11-Temperature-and-Humidity-Sensor

(s.d.). Tratto da http://www.savagehomeautomation.com/projects/category/pi4j

(s.d.). Tratto da http://www.mokabyte.it/2007/01/maven-1/

(s.d.). Tratto da https://en.wikipedia.org/wiki/Internet_of_things

(s.d.). Tratto da https://en.wikipedia.org/wiki/Home_automation

(s.d.). Tratto da https://en.wikipedia.org/wiki/Protocol_(object-oriented_programming)

(s.d.). Tratto da https://maven.apache.org/

(s.d.). Tratto da https://git-scm.com/

(s.d.). Tratto da http://www.dexterindustries.com/grovepi/

(s.d.). Tratto da https://www.raspberrypi.org/

(s.d.). Tratto da http://www.dexterindustries.com/GrovePi

(s.d.). Tratto da https://blogs.oracle.com/jag/resource/Fallacies.html

Carano, M. (2015). *Sperimentazione di tecnologie Raspberry in contesti di home intelligence.* Università di Bologna.

Grosso, W. (2002). *Java RMI.* O'Reilly.

Jarrod, T., Ian, A., & Gilles, G. (2002). *Sensor Abstraction Layer.* SP&E.

Pianciamore, M. (2002). *Programmazione Object Oriented in Java: Java Remote Method Invocation.*

Thomas, F. R. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* University of California, Irvine.

Appendice A: diagrammi di sequenza per il Provider RMI

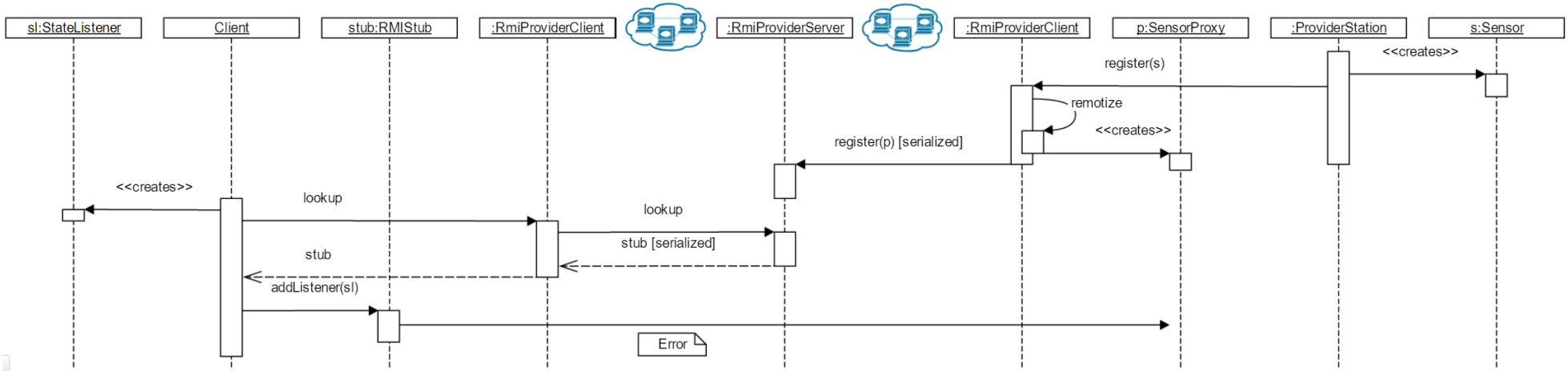


Figura 32 Sequenza di invocazioni per la registrazione di uno StateListener, non funzionante

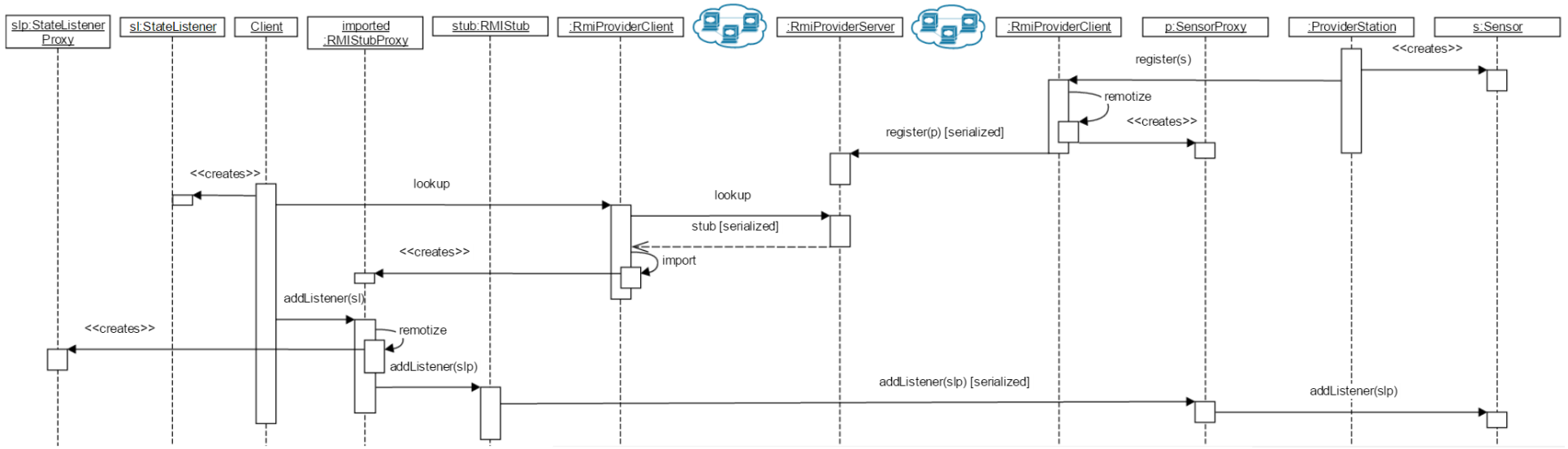


Figura 33 Sequenza di invocazioni per la registrazione di uno StateListener, funzionante