Abstract

Viene presentato in questo documento il lavoro svolto per la parte di middleware del progetto PerLa relativa alla connessione "Plug & Play" di nuovi dispositivi. Il progetto si divide in due parti principali, una prima parte in cui sono stati implementati dei driver di alto livello per la comunicazione tra i dispositivi e il sistema, e una seconda parte che permetta di inserire nuovi dispositivi nel sistema a run time.

Contents

List of Figures

Chapter 1

Introduzione

Il lavoro svolto fa parte del progetto PerLa. In questo capitolo verranno date informazioni generali relative al linguaggio PerLa ed alle parti sviluppate nel corso del progetto. Ci si soffermerà maggiormente sulla parte del middleware PerLa, essendo questa l'oggetto del progetto sviluppato.

1.1 Introduzione a PerLa

PerLa (PERvasive LAnguage) [1, 2] è un linguaggio dichiarativo per l'interrogazione di sistemi pervasivi. E' stato progettato per poter supportare sistemi eterogenei e per essere facilmente distribuito su reti di sensori (es. WSN, Wireless Sensors Networks). Grazie a sintassi e semantica ispirate ad SQL, non comporta alcuna difficoltà d'apprendimento per utenti che hanno già esperienza con linguaggi dichiarativi. Oltre al linguaggio vero e proprio, PerLa comprende anche un middleware [3] volto a fornire delle astrazioni logiche di alto livello, chiamate Oggetti Logici, per ogni nodo connesso al sistema che forniscono un'interfaccia comune per l'interrogazione dei nodi collegati alla rete. L'aggiunta di nuovi dispositivi risulta semplificata rispetto ad altri sistemi di gestione di reti wireless, grazie alla presenta di una libreria C portabile che implementa le funzioni richieste per la comunicazione con il middleware, e ad un sistema di aggiunta di nuovi nodi basato su paradigma "Plug & Play".

1.1.1 Il Linguaggio

Come accennato in precedenza, grazie agli Oggetti Logici, la struttura fisica di un dispositivo risulta completamente trasparente al linguaggio PerLa. Ogni volta che deve essere eseguita un'interrogazione, sarà il middleware ad interpretare le richieste inviate dall'utente sotto forma di query e a occuparsi della comunicazione con i dispositivi fisici che costituiscono la rete. Il linguaggio PerLa [4] supporta tre tipi di query:

- Low Level Query: definiscono il comportamento dei nodi della rete. Sono composte da diverse sezioni, le quali descrivono le regole per la scelta dei dispositivi dai quali campionare, le modalità di campionamento e le operazioni da effettuare sui dati campionati.
- High Level Query: permettono la manipolazione di flussi di dati provenienti da query di basso livello o altre query di alto livello.

- Actuation Query: non sono volte alla raccolta di dati, ma all'invio di comandi di attuazione verso i nodi della rete.

1.1.2 Il Middlware

l middleware merita maggior attenzione in quanto è l'oggetto di questo progetto. I compiti principali del middleware sono:

- astrarre i nodi collegati al sistema
- supportare l'esecuzione delle query PerLa.

Di seguito viene fornita una breve introduzione dei suoi componenti:

- Language Parser: parsifica le query testuali inviate al sistema e ne fornisce una rappresentazione ad oggetti.
- Functionality Proxy Component (FPC): implementa le funzionalità dell'Oggetto Logico.
- Logical Object Registry: contiene la lista degli Oggetti Logici correntemente registrati nel sistema.
- High Level Query Executor: esegue le query di alto livello.
- Low Level Query Executor: esegue le query di basso livello.
- Low Level Software: adatta i diversi dispositivi al middleware. E' il componente che fornisce le astrazioni logiche degli oggetti.

Come già accennato in precedenza, il middleware è stato progettato al fine di ridurre il carico di lavoro richiesto all'utente per aggiungere nuovi dispositivi di campionamento. Questo obiettivo ha portato allo sviluppo di una libreria platform-independent, che implementa una serie di funzioni necessarie alla gestione delle risorse hardware di un nodo della rete (es. scheduling, gestione dei timer e dei dispositivi di comunicazione, ecc), ed un sistema per l'aggiunta "Plug & Play" di nuovi dispositivi. Quest'ultima funzionalità è basata su questi componenti:

- Descrittore XML: file XML contenente la descrizione completa di un nodo della rete. Viene memorizzato dai dispositivi ed inviato al middleware.
- FPC Factory: componente del middleware preposto alla creazione "on-the-fly", senza intervento da parte dell'utente, di FPC (Oggetti Logici) in grado di comunicare con i rispettivi dispositivi. Le informazioni necessarie per l'istanziazione di nuovi FPC sono ricavate dai Descrittori XML.

La finestra temporale durante la quale un nuovo sensore si connette presso il middleware è chiamata "Fase di Binding", e prevede:

- l'invio del Descrittore XML da parte di un nodo della rete
- la creazione dell'FPC da parte dell'FPC Factory
- l'invio della conferma dell'avvenuta registrazione verso il nodo

Il middleware comunica con il resto del sistema per mezzo di due interfacce. L'Application Level Interface, interfaccia "verso l'alto", che contiene i metodi per effettuare query testuali al sistema e ricevere i risultati e la Physical Level Interface.

1.2 Physical Level Interface

L'idea alla base della realizzazione di quest'interfaccia consiste nel tentativo di minimizzare il più possibile lo sforzo programmativo di basso livello richiesto ad un utente. Per questo motivo la Physical Level Interface è, di fatto, una libreria C che può essere estesa con nuovi driver e ricompilata a seconda dell'architettura hardware del dispositivo da connettere, ma che contiene già le funzioni comuni di cui tutti i nodi della rete avranno bisogno. La scelta del linguaggio C deriva dalla necessità di supportare dispositivi eterogenei: deve essere possibile connettere al middleware anche nodi che non sono JAVA-enabled. Poiché l'obbiettivo ultimo di PerLa è quello di essere in grado di interrogare sistemi pervasivi di qualsiasi tipo, e poiché esistono vastissime categorie di dispositivi con capacità computazionali estremamente limitate, sviluppare le librerie della Physical Level Interface in un linguaggio come Java precluderebbe l'utilizzo dei nodi meno performanti all'interno di un sistema PerLa (in quanto non sarebbero in grado di eseguire una Java Virtual Machine). In generale, invece, qualsiasi piattaforma hardware è dotata di un compilatore C. Questa parte del progetto mira a sviluppare le funzionalità richieste per la gestione del canale di comunicazione con il middleware. I componenti principali che svolgono quest'operazione sono il Gestore del Canale (Channel Manager) e gli Adapter.

1.2.1 Channel Manager

Il Channel Manager è il componente preposto all'astrazione del canale di comunicazione. I suoi compiti principali sono:

- fornire un'astrazione del canale di comunicazione tramite una semplice interfaccia composta da due semplici primitive di scrittura e lettura di pacchetti
- mascherare ai livelli superiori i problemi relativi alla comunicazione (frammentazione dei dati, complessità protocollare, ecc).

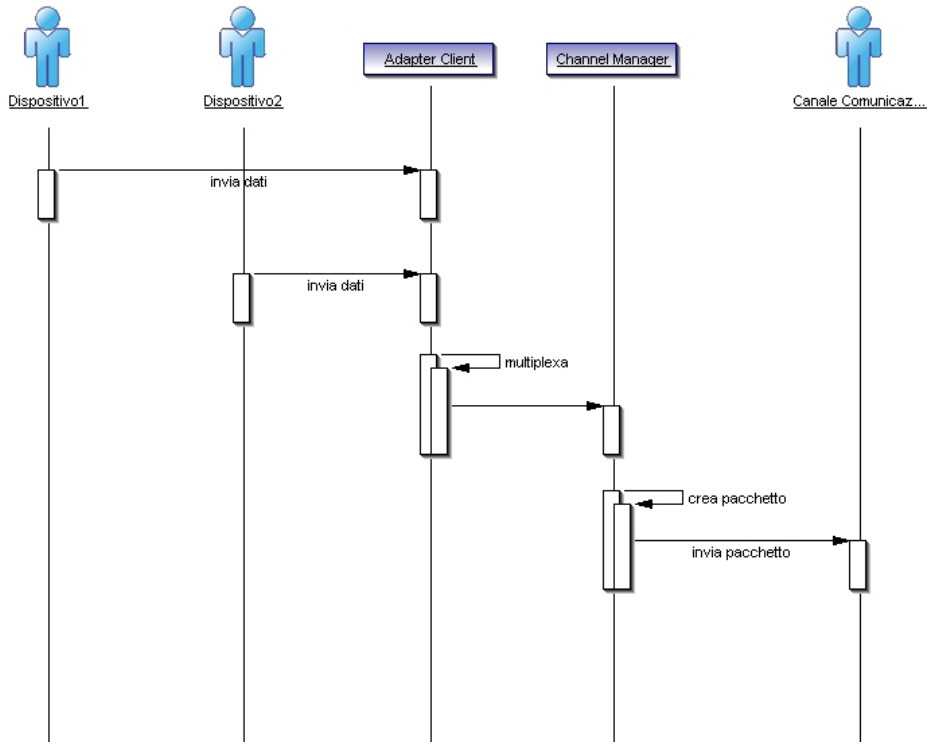

Figure 1.1: Sequence Diagram raffigurante il funzionamento della Physical Level Interface.

A seconda del canale di comunicazione utilizzato, il Channel Manager può scegliere di mandare pacchetti di due tipi diversi:

- Addressful: sono i pacchetti utilizzati nel caso in cui il protocollo del canale di comunicazione sottostante non gestisca autonomamente la trasmissione dell'indirizzo di sorgente e destinazione (ad esempio canali totalmente broadcast privi di alcun tipo di indirizzamento).
- Addressless: in questo caso il Channel Manager lascia la gestione dell'indirizzamento del pacchetto al protocollo del canale (ad esempio nel caso in cui si stia trasmettendo su un canale che utilizza TCP/IP).

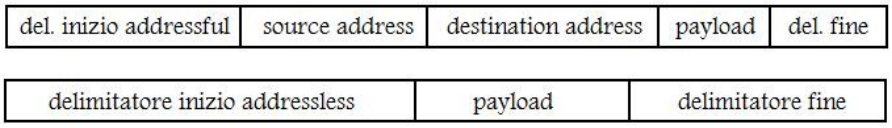

Figure 1.2: I due tipi di pacchetti gestiti dal Channel Manager. Sopra addressful, sotto addressless.

1.2.2 Adapter Server e Adapter Client

Il middleware PerLa offre la possibilità di utilizzare un dispositivo funzionante come aggregato di più nodi. Questo può rendersi necessario nel caso in cui i singoli sensori non abbiano le capacità necessarie per gestire direttamente la comunicazione con il proprio FPC. Anche in situazioni come queste il sistema PerLa fornisce all'utente la possibilità di accedere ai singoli nodi, astraendoli come se ognuno di essi fosse direttamente collegato al proprio FPC. In queste condizioni vengono introdotti nel sistema gli Adapter. Questi dispositivi hanno il compito di creare e gestire diversi canali virtuali all'interno di un singolo canale fisico. Un canale virtuale è quindi l'astrazione che permette ad un nodo della rete di comunicare direttamente con l'FPC che lo controlla, indipendentemente dalla presenza di un canale fisico dedicato.

Un Adapter può essere di due tipi: Adapter Server o Adapter Client. L'Adapter Server viene utilizzato nella parte di alto livello del middleware, ed è il componente autorizzato alla creazione dei canali virtuali; l'Adapter Client viene eseguito direttamente dai nodi della rete, e può creare solo canali virtuali di binding (cioè dei canali virtuali utilizzati solo durante la fase di connessione del sensore al sistema). Nel corso del progetto svolto è stato realizzato solo il cosiddetto Adapter Client, che si occupa del multiplexing di più nodi sullo stesso canale fisico e della fase di setup dei dispositivi collegati; l'Adapter Server infatti è già stato implementato in un altro progetto [5]. L'Adapter Client permette ad ogni nodo collegato di comunicare per mezzo di due tipi di canali:

- Canali Virtuali (VC): sono i canali virtuali usati normalmente dai sensori e

restano attivi finché gli Adapter in comunicazione non cambiano. Ad ogni VC viene associato un particolare identificatore (VCI, Virtual Channel Identifier).

- Canali Virtuali di Binding (BVC): Una volta terminata la fase di binding vengono sostituiti da un VC normale. Sono caratterizzati dal BVCI (Binding Virtual Channel Identifier).

Quindi, ad ogni nuovo nodo collegato, l'Adapter Client provvederà inizialmente ad associare un BVCI, e in seguito alla ricezione del primo messaggio da parte dell'FPC lo sostituirà con un VCI generato dall'Adapter Server.

1.3 Plug & Play

Sicuramente PerLa è un tipo di sistema che beneficia enormemente dalla presenza di un sistema "Plug & Play" per interfacciare nuovi dispositivi. Infatti, in assenza di tale paradigma, ogni qual volta si volesse aggiungere un nuovo nodo alla rete di sensori, sarebbero state necessarie onerose operazioni di aggiornamento della rete. Invece, grazie al "Plug & Play", il middleware Perla è in grado di riconoscere e aggiungere il nuovo nodo automaticamente.

La realizzazione del paradigma "Plug & Play" viene affrontata nella seconda parte di questo documento, e la strategia adottata ai fini progettuali è la seguente:

- Definizione di uno schema XML (XSD), per la validazione dei descrittori XML propri di ogni nodo presente nella rete, o di un nuovo dispositivo che ad essa vuole aggiungersi, che come spiegato nel capitolo 2.5 dovrà essere il più generale possibile, per avere la possibilità di supportare e riconoscere i dispositivi di tipo più eterogeneo possibile. I descrittori XML dovranno necessariamente seguire le regole contenute nell'XSD. Lo schema XML è stato progettato perseguendo l'obbiettivo di essere compatibile con dispositivi più o meno intelligenti, e pensato per contenere strutture dati "C-like".

- Generazione automatica di oggetti Java che rappresentino in tutto e per tutto i descrittori XML dei nuovi nodi connessi. I file Java verranno ottenuti parsificando il documento XML del dispositivo che si vuole aggiungere alla rete, ed eseguendo la "traduzione" delle strutture in esso contenute in linguaggio Java. Alla fine di tale operazione le classi generate conterranno le strutture dati e i parametri comunicati dal nodo tramite il relativo documento XML, i metodi che consentano di potere accedere e modificare tali strutture, e le annotazioni che servono ai componenti di livello superiore per il riutilizzo dei file Java generati.

- Compilazione "on-the-fly" delle classi Java generate in automatico, e quin-

di creazione di un oggetto logico (FPC) che consenta ai componenti di livello superiore di poter effettuare le query richieste dall'utente sul nodo appena inserito nella rete.

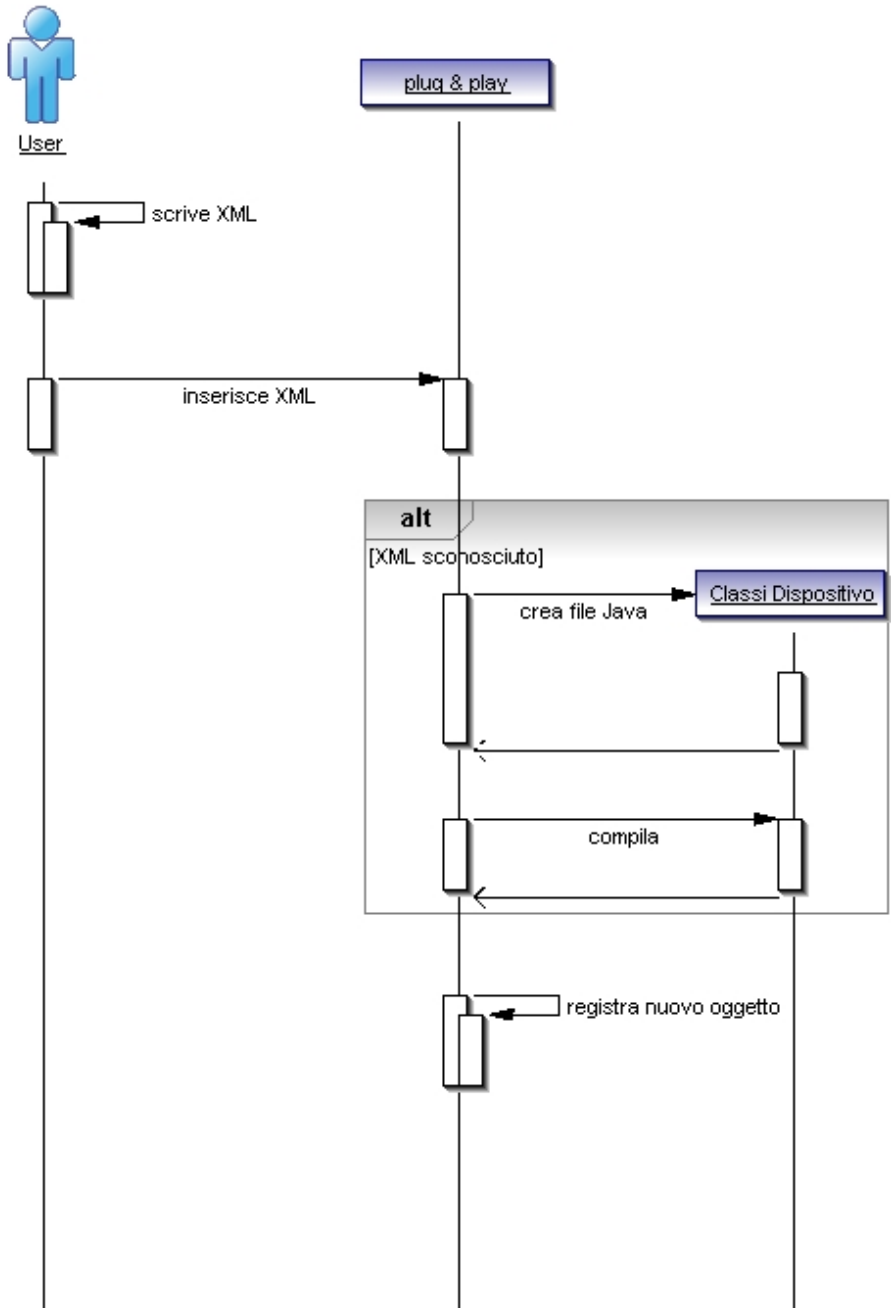

Figure 1.3: Sequence Diagram raffigurante il funzionamento del sistema “Plug & Play”.

Chapter 2

Design del Software

In questo capitolo verrà mostrato come è stato realizzato il progetto.

La prima parte è dedicata alla Physical Level Interface, e si divide in:

- restrizioni imposte all'inizio del progetto (2.1)
- funzionamento della struttura utilizzata per astrarre un flusso di dati, lo Userbuffer (2.2)
- analisi dettagliata dell'implementazione dei componenti Channel Manager (2.3) e Adapter Client (2.4)

Nella seconda parte di questo capitolo invece, viene mostrato come è stato implementato il "Plug & Play" di un dispositivo. Per far in modo che PerLa sia in grado di supportare ciò efficacemente infatti, sono state realizzati diversi componenti di middleware.

In primo luogo si mostra come è stato definito un "XML Schema" (XSD) che serva al sistema per validare il documento XML atto a descrivere un generico sensore (2.5). Si supponga ora che il sistema riceva un documento XML valido da un nuovo dispositivo che tenta di collegarsi a PerLa; possono verificarsi due casi:

Il dispositivo è di un tipo già noto al sistema (ad esempio, viene collegato un nuovo sensore di temperatura ad un sistema che ha già altri sensori di tem-

peratura collegati), in questo caso il server PerLa avrà già delle classi Java che descrivono la struttura dati comunicata dal sensore e si dovrà limitare a crearne delle nuove istanze.

Il dispositivo è di un nuovo tipo, sconosciuto al sistema. L'operazione da svolgere in questo caso è più complessa. Infatti, PerLa deve creare delle classi Java che descrivano la struttura dati comunicata dal sensore a partire dell'XML. Per fare ciò, in questo progetto si è fatto uso delle librerie JAXB (Java Architecture for XML Binding) che, come mostrato in 2.6 sono in grado di effettuare il mapping da XML a oggetti Java di supporto.

Successivamente l'albero delle classi di supporto creato da JAXB viene navigato per costruire dei file .java che rispecchino l'effettiva struttura dati del sensore (2.7). Dopodiché i file .java creati vengono caricati in memoria e compilati (2.8).

L'ultima operazione deve essere eseguita sia nel caso in cui il dispositivo sia già noto al sistema, sia nel caso in cui non lo sia. Essa consiste nell'inserire le classi generate all'interno di un contenitore unico che il sistema PerLa è in grado di interrogare. Questo viene mostrato in 2.9.

2.1 Restrizioni

Come si è già accennato nelle sezioni precedenti, l'interfaccia di livello fisico deve rispettare diversi criteri di generalità e portabilità. Il codice realizzato deve poter essere ricompilato su qualsiasi tipo di dispositivo. Da qui la scelta di realizzare la libreria in linguaggio C: anche una macchina su cui non è possibile eseguire una JVM deve aver accesso alle funzioni realizzate nel progetto. Per rispettare i due criteri citati, si è deciso di:

- non fare mai utilizzo dell'allocazione dinamica di memoria (malloc/calloc): si è visto infatti come certi nodi non abbiano sufficiente memoria per permettere un uso efficace dell'allocazione dinamica, o non dispongano di MMU (Memory Management Unit). Questo comporta l'impossibilità di utilizzare strutture dati dinamiche (liste, code ecc.).

- utilizzare i tipi uintX_t (uint8_t, uint32_t,): questi tipi garantiscono la lunghezza in bit del dato contenuto in essi. Ad esempio, la scelta di "uint8_t" garantisce che ogni variabile così dichiarata abbia una dimensione di esattamente un byte (8 bit) indipendentemente dall'architettura hardware del dispositivo su cui il codice è compilato; se la stessa variabile fosse stata un "char", avrebbe potuto occupare 1 o 2 byte. Considerazioni analoghe si possono fare per tutti gli altri tipi di dati primitivi (int, long, char, ecc.). Per la natura delle funzioni realizzate, conoscere la quantità di memoria occupata da ciascun dato è fondamentale.

2.2 UserBuffer

Lo Userbuffer è la struttura dati utilizzata per contenere uno stream di byte. E' caratterizzata da:

- Buffer*: puntatore alla prima cella di memoria dove sono contenuti i dati.
- Size: numero massimo di byte contenuti dal buffer.
- Cursor: posizione della prossima cella di memoria da leggere.
- LastDataIndex: ultimo byte utile contenuto nel buffer.

Questa struttura dati è inoltre dotata delle funzioni USERBUFFER_commitData e USERBUFFER_getData, utilizzate per laggiunta e la rimozione di informazioni dal buffer. Altre funzioni per la gestione di questa struttura dati sono USERBUFFER_init e USERBUFFER_setBuffer, le quali vengono rispettivamente utilizzate per linizializzazione degli indici di posizione e per laggiunta dellarea di memoria in cui andranno inseriti i dati. Al fine di semplificare il debugging delle applicazioni sviluppate, è stata aggiunta una funzione per la visualizzazione su standard output dellintero contenuto del buffer (USERBUFFER_dump).

E' evidente come una struttura come questa sia ottima per astrarre uno stream di dati provenienti da un canale di comunicazione: infatti, pur non essendo dinamica (size è fissata al momento della creazione), può essere "riempita" con un numero di byte variabile, indicato da lastDataIndex, e viene sempre tenuta traccia della posizione in cui ci si trova per mezzo di cursor. Gli indici LastDataIndex e Cursor garantiscono inoltre che linserimento di nuovi dati non vada a sovrascriverne quelli già presenti nel buffer, e che la lettura riprenda sempre dal primo byte non ancora letto.

2.3 Channel Manager

Il Channel Manager è stato implementato tramite l'utilizzo di due Userbuffer, uno per mandare dati sul canale (ChMgrOutputBuffer) e uno per prelevarli (ChMgrInputBuffer). Le funzioni che si occupano di processare i dati da inviare

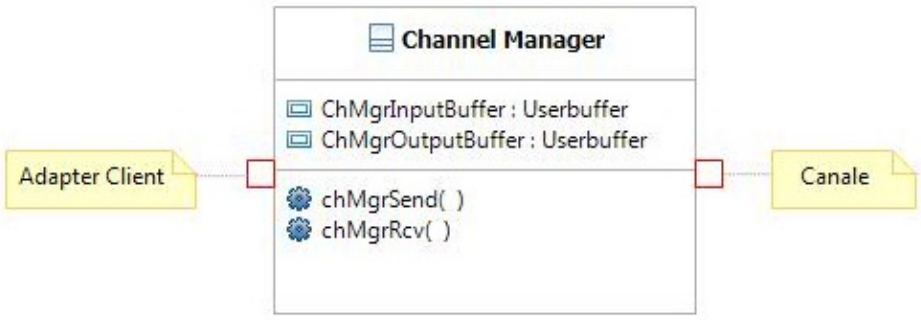

Figure 2.1: Class Diagram relativo ad un Channel Manager.

o ricevuti sono chMgrSend e chMgrRcv, i cui prototipi sono i seguenti:

```
void chMgrSend(UserBuffer* in, UserBuffer* out,
  UserBuffer* temp, bool addressless, uint8\_t address[])
void chMgrRcv(UserBuffer* in, UserBuffer* out,
  UserBuffer* temp, uint8\_t address[])
```

Per entrambe le funzioni in è il buffer da processare, out è il buffer su cui la funzione scrive il proprio output e temp è un buffer temporaneo. Addressless indica a chMgrSend se il pacchetto da inviare deve essere addressless o meno. address[], nel caso di invio, è l'indirizzo del destinatario; nel caso di ricezione è l'indirizzo del nodo stesso che chiama la funzione.

Di seguito viene fornita una breve documentazione delle funzioni:

- ChMgrSend: si occupa di aggiungere i caratteri delimitatori di inizio (STARTAL/STARTAF, rispettivamente per la creazione di un pacchetto addressless o addressful) e fine (END) al contenuto dello Userbuffer in ingresso; nel caso sia stato utilizzato il delimitatore dinizio pacchetto di tipo

addressful, la funzione si occuperà anche di aggiungere gli indirizzi sorgente e destinazione della comunicazione. Queste operazioni sono svolte dalla sottofunzione header. Nel caso in cui un carattere contenuto nel payload del pacchetto sia uguale ad uno dei caratteri delimitatori, prima di copiare il carattere sul canale chMgrSend provvederà ad aggiungere un carattere di escape (definito come SPECIAL nel codice); sarà così possibile per il Channel Manager che riceverà il pacchetto ricostruire il contenuto della comunicazione originale. Naturalmente, questa procedura verrà applicata anche anche nel caso il carattere di escape stesso sia trovato tra i byte da inviare. Queste ultime operazioni sono invece svolte dalla sottofunzione codifica. Es.: se il pacchetto da inviare è [xert/] ed è un pacchetto addressless, il risultato inviato sullo Userbuffer ChMgrOutputBuffer sarà [sx/ert//e], nel caso in cui siano STARTAL=[s], END=[e], SPECIAL=[/].

- ChMgrRcv: esegue la procedura inversa rispetto a quella implementata da ChMgrSend; ovvero, prima rimuove i delimitatori di inizio e fine pacchetto, e poi naviga il payload al fine di ricostruire il contenuto originale. Nel caso in cui la funzione trovi come primo byte del pacchetto il delimitatore di inizio addressful, provvederà anche alla gestione degli indirizzi contenuti. Infatti, scarterà tutti i pacchetti che hanno un indirizzo diverso da quello associato al Channel Manager che ha chiamato la funzione. Anche chMgrRcv è composta da due sottofunzioni, decodifica e rimuoviheader, le quali svolgono le operazioni appena descritte. Riprendendo il caso dell'esempio precedente, se la funzione riceve [sx/ert//e], il risultato sarà [xert/] (si può notare come il Channel Manager si "accorga" che il pacchetto è addressless e rimuova solo il primo byte, che comunque contiene il delimitatore di inizio).

Dato il metodo di funzionamento del Channel Manager, è chiaro che lo Userbuffer in uscita deve essere sovradimensionato rispetto a quello in ingresso. La dimensione minima che deve avere ChMgrOutputBuffer per non avere perdita di dati è di 2N+2+ADDRESSLENGHT, dato un ChMgrInputBuffer di size=N.

Infatti nel caso in cui si voglia inviare un pacchetto di dimensione uguale a N il cui contenuto è composto interamente da caratteri uguali ai delimitatori, si avrà bisogno di altri N byte per i caratteri speciali, di 2 byte per i caratteri delimitatori e di ulteriori ADDRESSLENGHT byte per l'indirizzo di destinazione.

Come risulta dalla struttura delle funzioni appena presentate, il compito principale del Channel Manager è quello di interfacciare il canale di comunicazione sottostante con un Adapter Client, presentato nella sezione successiva.

2.4 Adapter Client

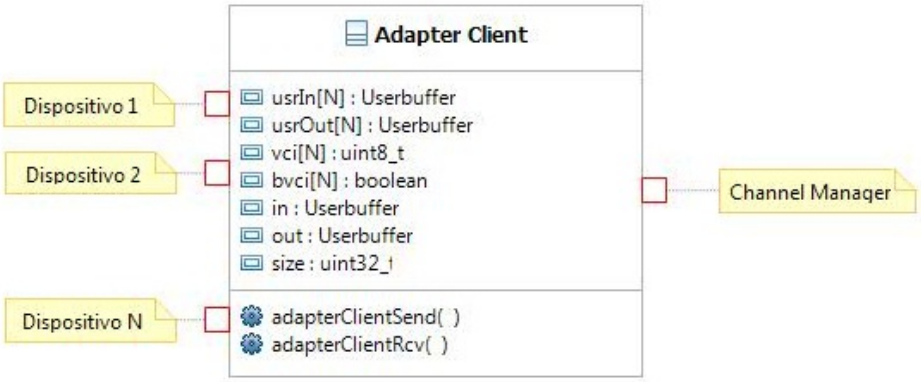

Figure 2.2: Class Diagram relativo ad un Adapter Client.

L'Adapter Client è l'interfaccia utilizzata dai nodi per inviare e riceve dati. Per gestire i vari dispositivi connessi nel progetto è stato implementato utilizzando le seguenti strutture dati:

- ACInputBuffer, ACOutputBuffer: sono i due Userbuffer che l'Adapter Client usa per comunicare con il Channel Manager
- size: numero di dispositivi attualmente connessi
- vci[]: vettore contenente gli identificatori di canale virtuale (VCI) dei dispositivi connessi
- usrIn[]/usrOut[]: vettori di Userbuffer utilizzati dai nodi per comunicare con l'Adapter Client
- bvci[]: vettore di valori booleani che specificano se i VCI memorizzati in vci[] sono di binding o meno

Ogni Adapter Client è dotato di tre funzioni: AdapterClientInit, AdapterClientSend, AdapterClientRcv. La funzione AdapterClientInit, come suggerito dal nome, viene utilizzata per linizializzazione delle strutture dati utilizzate.

AdapterClientSend è una funzione utilizzata esclusivamente dai nodi collegati allAdapter, al fine di notificare la presenza di dati pronti per linvio su uno dei buffer di uscita. Il suo prototipo è:
void AdapterClientSend(UserBuffer* in, AdapterClient* AC) dove in è il buffer del nodo che si attiva per inviare dati, e AC è l'Adapter Client che li dovrà inviare (e sul cui buffer di uscita verrà scritto il risultato).

Le operazioni che esegue sono:

- Ricerca dellindice identificativo del nodo allinterno dellAdapter Client.
- Recupero dellidentificatore di canale virtuale associato al nodo dal vettore vci.
- Invio su ACOutputBuffer il carattere delimitatore definito in BVCI/VCI (a seconda che il canale sia di binding o no), il BVCI/VCI del canale di comunicazione usato, il carattere delimitatore definito in PAYLOAD (per indicare che ciò che segue è il carico), e infine i dati.

La funzione AdapterClientRcv ha il seguente prototipo:
void AdapterClientRcv(AdapterClient* AC), in cui AC è l'Adapter Client su cui vengono svolte le operazioni. Essa viene invocata da un Channel Manager per notificare la ricezione di dati dal canale di trasmissione, ed esegue le seguenti operazioni:

- Naviga all'interno del pacchetto in cerca del VCI contenuto in esso, che verrà utilizzato dall'Adapter Client per identificare a quale dispositivo il pacchetto è indirizzato.
- Quando la funzione ha individuato quale sia il nodo a cui sono indirizzati i dati ricevuti, provvede a spostare i soli dati del carico da ACInputBuffer (Userbuffer di ingresso dell'Adapter Client) allo Userbuffer del sensore destinazione.
- Nel caso in cui il canale virtuale fosse di binding (e quindi identificato da un BVCI), la funzione aggiorna il VCI assegnato al sensore di destinazione con quello contenuto nel pacchetto.

E' importante notare che la ricezione può avvenire da un solo buffer (cè infatti un solo Channel Manager), ma linvio può avvenire da più buffer (ci sono molti nodi). L'Adapter Client risulta quindi, come accennato nelle sezioni precedenti (1.2.2), il componente che permette a più nodi di essere multiplexati attraverso un singolo canale di comunicazione.

2.5 Definizione Schema XML

La definizione di un buon XSD è fondamentale per il corretto funzionamento del sistema. Infatti, definendo un XSD il più generale possibile il sistema sarà in grado di supportare una maggior varietà di dispositivi. Al contrario, definendo un XSD specifico per certi tipi di dispositivi, sarà più semplice la loro definizione e successivamente la gestione dei dati da essi prodotti. Un esempio ovvio è la definizione della grandezza "Temperatura" nell'XSD: tutti i sensori di temperatura conterranno nel loro descrittore XML questa grandezza.

Vengono ora presentate le idee principali che hanno portato alla formulazione del foglio di schema per come è stato poi implementato:

- Generalità: Ovviamente non è possibile a priori definire tutti i tipi di grandezze che un generico dispositivo che si vuole collegare a PerLa può produrre. Bisogna infatti ricordare che PerLa ha l'ambizioso obbiettivo di supportare e gestire dati generati da qualsiasi dispositivo elettronico, dal più semplice sensore capace di misurare dati elementari come la temperatura o l'umidità presente in un dato ambiente fino ad arrivare a dispositivi molto più complessi come un PDA o un computer. Appare quindi evidente che un XSD che vuole gestire una così vasta gamma di dispositivi dovrà essere il più generale possibile.

- Strutture C-like: D'altro canto si possono fare certe assunzioni sulla natura dei dati. Infatti questi proverranno come si è detto da un dispositivo elettronico. Per questo motivo nel corso della progettazione dello schema si è supposto che comunque un sensore sia capace di produrre una struttura dati simil-C. Prendiamo ad esempio due sensori, uno di temperatura e uno di umidità. L'output prodotto da essi avrà ovviamente dei significati molto diversi, ma ha certamente senso dire che entrambi produrranno un numero intero o a virgola mobile, a seconda della loro architettura hardware. Sembra quindi una scelta sensata dire che l'XSD contenga da qualche parte una grandezza "Tipo", che specifichi che tipo di dato sia un

certo parametro prodotto da un generico dispositivo.

Di seguito vengono mostrate le caratteristiche e le grandezze introdotte nell'XSD, presentate sotto forma di <nomeGrandezza : TipoGrandezza> :

- <perlaDeviceElement : PerlaDevice> E' l'elemento radice. Ha un nome e contiene a scelta o un <perlaSingleDevice : PerlaSingleDeviceType> o un <aggregatePerlaDevice : AggregatePerlaDeviceType>.

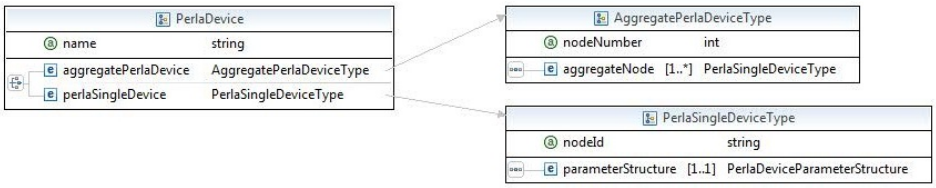

Figure 2.3: Diagramma raffigurante la struttura di un PerlaDevice.

- <aggregatePerlaDevice : AggregatePerlaDeviceType> Questo elemento non è altro che una collezione di <aggregateNode : PerlaSingleDeviceType>.
- <perlaSingleDevice : PerlaSingleDeviceType> Un PerlaSingleDeviceType contiene un ID (<nodeId : string>) e una <parameterStructure : PerlaDeviceParameterStructure>.

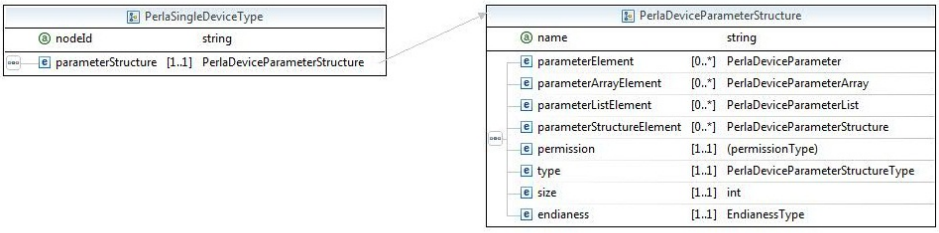

Figure 2.4: Diagramma raffigurante la struttura di un PerlaSingleDevice.

- <parameterStructure : PerlaDeviceParameterStructure> Questo è sicuramente l'elemento più importante e complesso dell'XSD. Innanzitutto una struttura è caratterizzata da nome (<@name : string>), tipo (<type : string>), dimensione (< size : int>), permessi in lettura e scrittura (<permission : (PermissionType)>, in cui PermissionType è un tipo anonimo che può assumere i valori 'r','w', ed 'rw', dall'ovvio significato), endianness (<endianness : EndiannessType>, in cui EndiannessType è una string che può assumere i valori BigEndian o LittleEndian) e la specifica della Constant (2.9) su cui fare il mapping della struttura (<mapping : string>). Come è noto una struttura poi può contenere delle variabili (<parameterElement : PerlaDeviceParameter>), degli array (<parameterArrayElement : PerlaDeviceParameterArray>), delle liste (<parameterListElement : PerlaDeviceParameterList>) e altre strutture annidate al suo interno (<parameterStructureElement : PerlaDeviceParameterStructure>).

Figure 2.5: Diagramma raffigurante la struttura di un PerlaDeviceParameter-Structure.

- <parameterElement : PerlaDeviceParameter> Questo elemento rappresenta una variabile C. Esso ha un nome (<name : string>), dimensione in byte (<lenght : int>), un tipo (<type : PerlaDeviceParameterType>, sostanzialmente un tipo sono due stringhe, una che ne identifica il nome e un'altra per il segno), un valore di default (<value : hexBinary>), attributeType, elemento necessario a PerLa per gestire i dati (<attributeType : AttributeTypeType> in cui AttributeTypeType è una string che può assumere i valori di probing, nonProbing e static), permessi di lettura

e/o scrittura (<permission : (PermissionType)>, in cui PermissionType è identico a quello visto in PerlaDeviceParameterStructure), dei limiti sui suoi valori massimi e minimi (<bounds : Bound> in cui Bound è una struttura contenente due hexBinary che indicano il minimo e il massimo), una scelta tra valori continui (<continuousValue : (continuousValueType)>, in cui continuousValueType è un hexBinary che indica a quale distanza l'uno dall'altro verranno prodotti i dati) o discreti (<discreteValue : discreteValueType>, un elenco impostato dall'utente di hexBinary che i dati prodotti potranno assumere), una funzione di conversione (<conversionFunction : FunctionType>, indica se i dati prodotti dal dispositivo dovranno essere accettati dal dispositivo come sono o dovranno subire un qualche tipo di trasformazione) e l'elemento mapping, analogo a quello presente in PerlaDeviceParameterStructure.

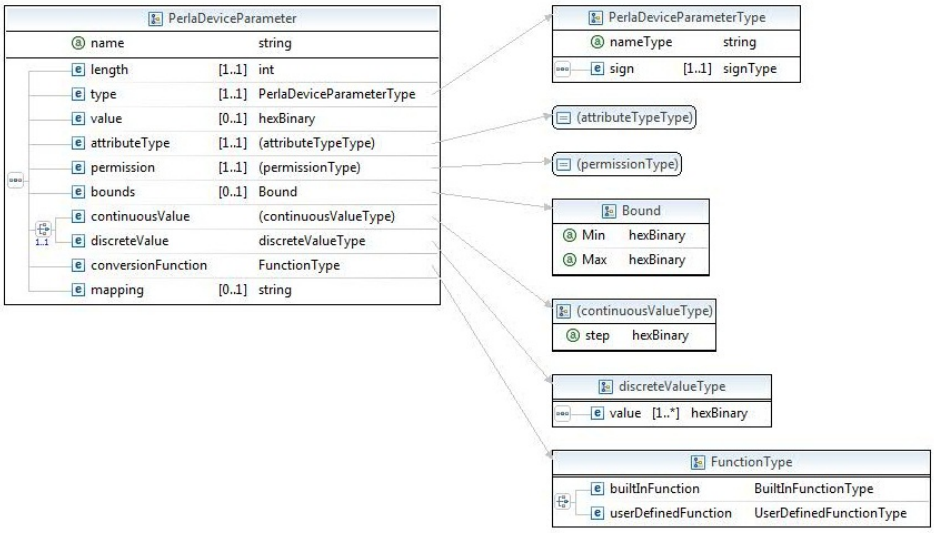

Figure 2.6: Diagramma raffigurante la struttura di un PerlaDeviceParameter.

- <parameterArrayElement : PerlaDeviceParameterArray> Un array ha una dimensione totale (<@size : int>), un padding (<padding : (paddingType)>, in cui paddingType è un tipo anonimo, costituito da due

attributi <@value : hexBinary> e <@position : int>) e una scelta tra parametro (caso di un array semplice, come int[] a), array (caso di matrici o array di ordine superiore, come int[][] a), liste (array di liste) strutture (array di strutture, come temp[] a).

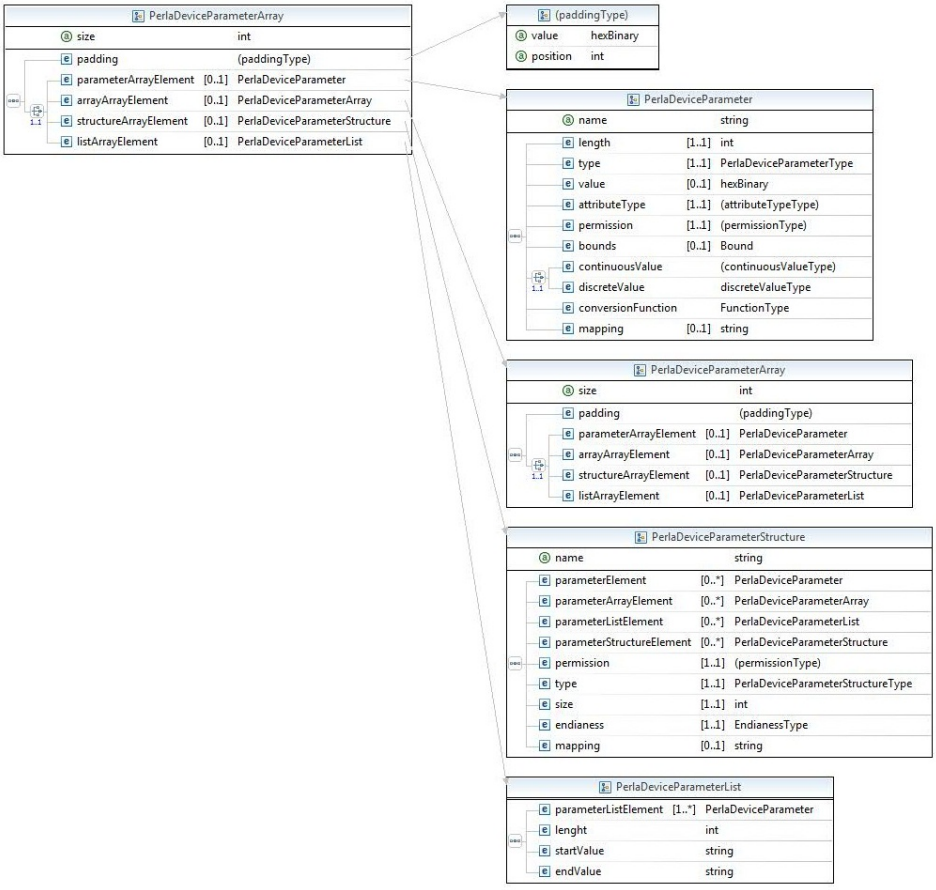

Figure 2.7: Diagramma raffigurante la struttura di un PerlaDeviceParameter-Array.

- <parameterListElement : PerlaDeviceParameterList> Una lista nell'XSD definito è sempre una lista di parametri semplici, con valore di inizio e fine e lunghezza (intesa come numero di elementi contenuti nella lista).

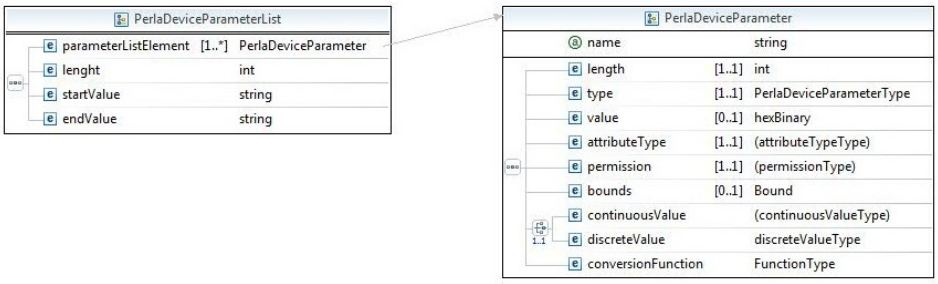

Figure 2.8: Diagramma raffigurante la struttura di un PerlaDeviceParameterList.

2.6 Java Interface for XML Binding

Nel momento in cui viene inserito nel sistema un documento XML valido, questo deve essere letto e mappato su una o più classi Java che ne riflettano la struttura. Per far ciò si è scelto di avvalersi del supporto di JAXB, una tecnologia sviluppata da Sun. Bisogna però notare che l'apporto dato da JAXB al sistema sviluppato, pur non essendo marginale, non è fondamentale. Come si vedrà nelle sezioni successive, tramite l'API di JAXB infatti ci si è limitati a creare delle classi di supporto, le quali poi verranno a loro volta lette per generare a run time il codice Java relativo ad un singolo sensore. Viene ora presentato brevemente il funzionamento generale di JAXB (2.6.1), mentre nelle sezioni successive viene mostrata l'operazione di binding tra uno schema XML e le classi Java (2.6.2) e l'unmarshaling di un documento XML in oggetti Java (2.6.3).

2.6.1 Breve introduzione a JAXB

JAXB [6] è una tecnologia di binding XML-Java che consente la trasformazione tra schemi e oggetti Java e tra documenti XML e istanze di oggetti Java. La tecnologia JAXB è composta da un'API e da strumenti associati che semplificano l'accesso a documenti XML. E' possibile utilizzare gli strumenti e le API JAXB per stabilire associazioni tra classi Java e schemi XML. La tecnologia JAXB fornisce un ambiente di run time che consente di convertire i propri documenti XML in oggetti Java e viceversa. Per accedere ai dati memorizzati in un documento XML, non è necessario comprenderne la struttura.

L'accesso ai documenti XML tramite programmi Java, avviene usualmente tramite l'utilizzo di parser, che scandiscono l'intero documento e lo frammentano logicamente in diverse parti che vengono poi messe a disposizione dell'applicazione Java.

I parser seguono due approcci differenti:

- SAX (Simple API for XML). I parser SAX iniziano a leggere il documento XML dall'inizio, lo suddividono quindi in pezzi e li passano all'applicazione

nell'ordine in cui li trovano, senza salvare nulla in memoria.

- DOM (Document Object Model). I parser DOM, invece, ricostruiscono un albero di oggetti, che rappresenta l'organizzazione dei dati all'interno del documento XML. L'albero viene quindi salvato in memoria, permettendo quindi la manipolazione e modifica dei dati.

JAXB rappresenta un approccio diverso dai due precedenti, e si basa su due operazioni principali che verranno descritte meglio in seguito; ossia le operazioni di binding e unmarshaling. L'operazione di marshaling, raffigurata nello schema e presente in JAXB, non verrà descritta in quanto non è risultata utile ai fini del progetto.

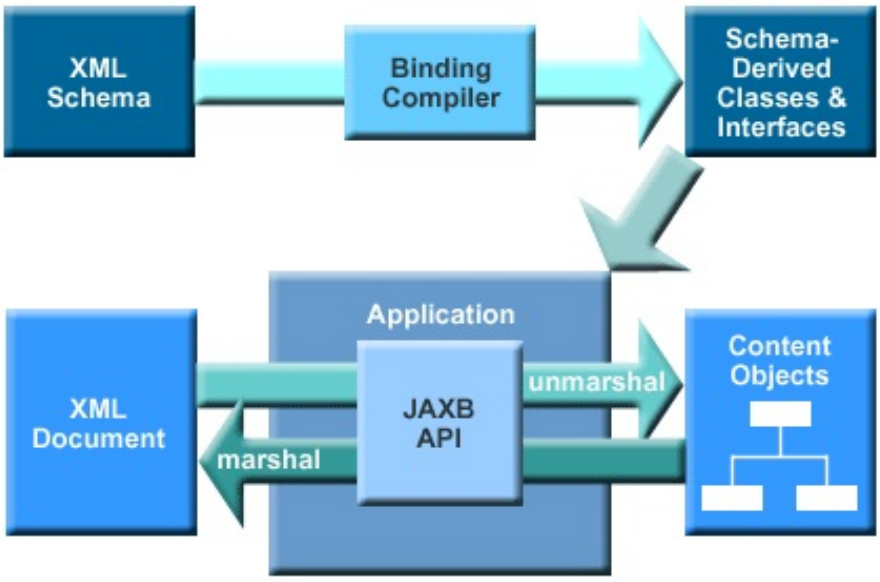

Figure 2.9: Schema rappresentante il funzionamento di JAXB.

In sostanza, JAXB, ricostruisce l'albero degli oggetti a partire dallo schema, e non dal documento XML come avviene nei due approcci precedenti, creando per ognuno degli elementi contenuti nell'XSD, una classe Java che lo rappresenta. In seguito questi oggetti verranno istanziati dall'applicazione, seguendo

uno specifico documento XML, che deve necessariamente rispettare l'XSD da cui è stato creato l'albero delle classi Java.

Tutto ciò permette quindi di accedere e manipolare un documento XML, conoscendo solamente lo schema generale XSD, senza dover implementare parser specifici per XML differenti

2.6.2 Binding

Figure 2.10: Schema rappresentante l'operazione di binding di uno schema.

L'operazione di binding è la prima operazione necessaria per poter convertire un documento XML in un insieme di oggetti Java. Tutto ruota sull'utilizzo di un componente, chiamato Binding Compiler, fornito da JAXB. Il compilatore, preso in ingresso uno schema XML, genera un insieme di classi Java corrispondenti agli elementi presenti nello schema. La sintassi per invocare il compilatore da riga di comando è la seguente:
xjc [opzioni] <schemaXML.xsd>
Come si può vedere, è sufficiente fornire al compilatore il path del file .xsd; l'unica opzione di cui si è fatto uso nel progetto è -p, che permette di specificare il package all'interno del quale devono essere create le classi Java. Per maggiori informazioni è possibile consultare la documentazione di xjc, disponibile presso [7]. Come si può vedere dalla figura, nel package xsdgeneratedclasses, sono stati inseriti tutti i file Java generati, tramite compilatore xjc di JAXB, a partire dallo schema XML progettato e sviluppato nel capitolo 2.5.

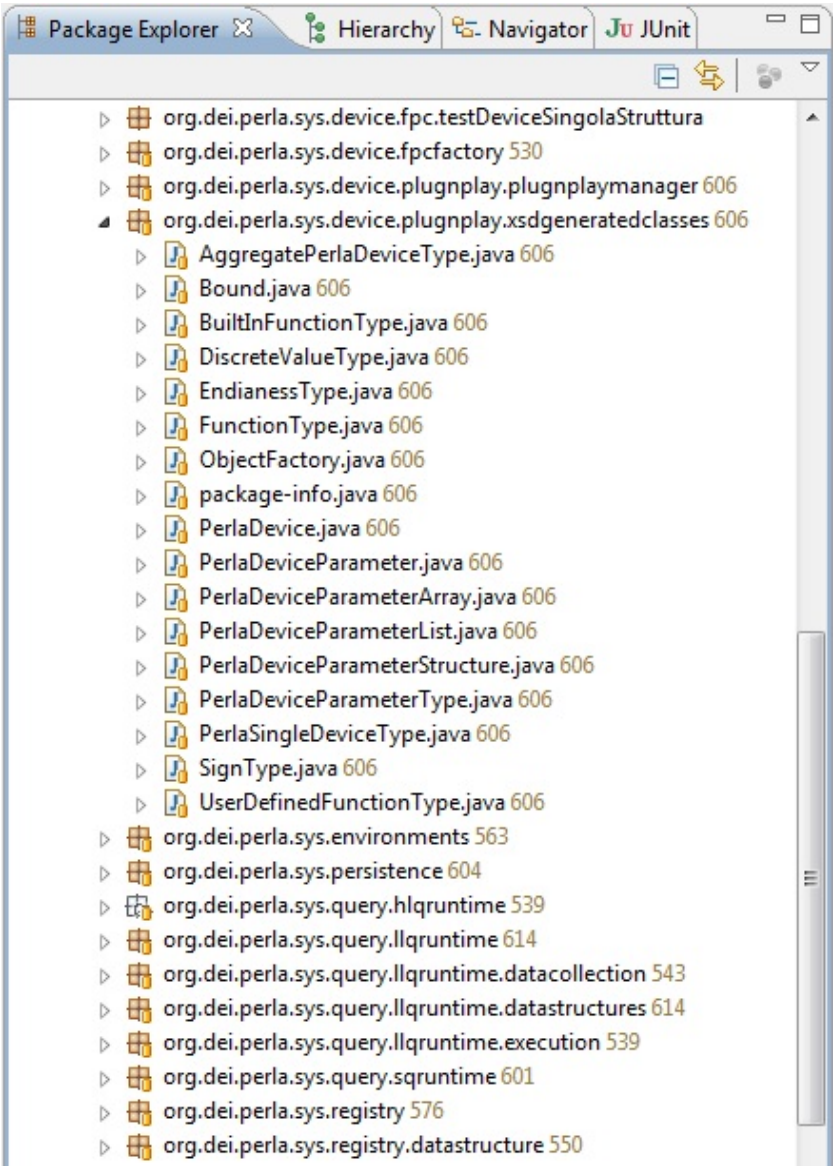

Figure 2.11: Classi generate dal compilatore xjc.

2.6.3 Unmarshalling

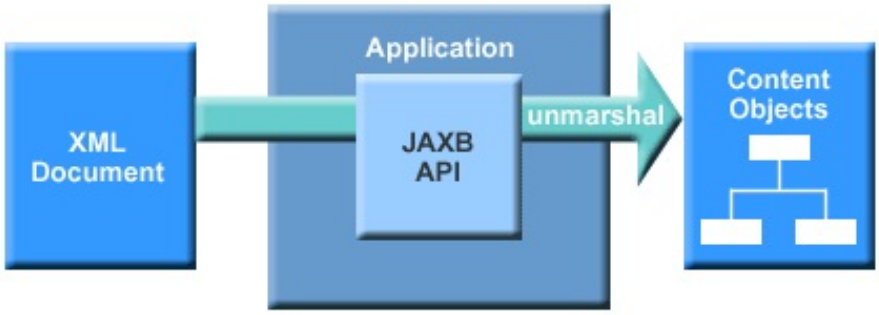

Figure 2.12: Schema rappresentante l'operazione di unmarshalling di un'XML.

L'operazione di unmarshalling provvede alla creazione dell'albero degli oggetti Java, istanziando le classi generate tramite la precedente operazione di binding, e quindi riportate nel package xsdgeneratedclasses, seguendo uno specifico documento xml, che deve necessariamente essere valido rispetto allo schema XML progettato. Una volta eseguito l'unmarshalling del documento sarà possibile navigare all'interno dell'albero degli oggetti Java.

Di seguito viene riportato il codice scritto per poter eseguire l'unmarshalling di un documento XML:

```
public PerlaDevice createClassTree(String pkg, String xml)
  throws Exception {
    JAXBContext jc = JAXBContext.newInstance(pkg);
    Unmarshaller unmarshaller = jc.createUnmarshaller();
    PerlaDevice pd = (PerlaDevice) ((JAXBElement) unmarshaller
      .unmarshal(new File(xml))).getValue();
    return pd;
    }
```

Come si può vedere, è necessario creare un oggetto JAXBContext, il quale rappresenta il punto di accesso all'API di JAXB, e bisogna fornirgli come parametro

il package contenente le classi generate tramite binding; si tratterà quindi del package xsdgeneratedclasses. Dopo aver creato un Unmarshaller, è possibile invocare il metodo unmarshal(), che è l'esecutore materiale dell'operazione di unmarshaling. L'Unmarshaller parsifica il documento XML passatogli come parametro, e restituisce l'elemento radice dell'albero, che è sempre un PerlaDevice.

A questo punto quello che si è ottenuto non è ancora il risultato finale, in quanto gli oggetti creati sono di tipo PerlaDeviceParameter, PerlaDeviceParameterStructure, ecc. Nel risultato finale invece le classi e poi gli oggetti Java generati saranno di tipi uguali ai valori contenuti nei campi tipo, i nomi saranno uguali ai valori dei campi nome, ecc. Quello che si è ottenuto fin'ora è solamente un risultato intermedio.

La scelta di utilizzare JAXB è dovuta al fatto che si è ritenuto fosse più semplice navigare all'interno di un albero di oggetti Java rispetto ad un documento XML. Questo è quello che viene fatto durante la costruzione a run time del codice, argomento affrontato nel prossimo capitolo.

2.7 Generazione del codice

Una delle parti essenziali del progetto consiste nel generare automaticamente le classi Java che rappresentino un nuovo dispositivo che viene connesso a Perla. Nel capitolo precedente si è visto come, con l'ausilio di JAXB, vengano generate le classi di supporto. Il passo successivo consiste nel creare le classi Java che riflettano la struttura vera e propria dei dati inviati dai dispositivi che vogliono connettersi al sistema.

Le classi generate avranno come attributi dei parametri semplici (derivati da PerlaDeviceParameter), piuttosto che array (da PerlaDeviceParameterArray), liste (da PerlaDeviceParameterList) o altre strutture (da PerlaDeviceParameterStructure, queste corrisponderanno a classi pubbliche in Java), e metodi che permettano di accedere a tali attributi. Inoltre, nel codice generato saranno presenti le annotazioni necessarie per consentire il riutilizzo di questi file da parte dei componenti ai livelli superiori di PerLa.

Per ottenere ciò sono state implementate due classi: ClassTreeManager, il componente che si occupa di effettuare l'unmarshaling e in seguito di chiamare i metodi della classe ParameterManager sul PerlaDevice ottenuto. In seguito ParameterManager si occupa di svolgere il lavoro di creazione del codice sorgente.

2.7.1 ClassTreeManager

Come accennato ClassTreeManager svolge l'operazione di unmarshalling, dopodiché chiama i metodi di ParameterManager.

Come si vede in figura, l'unico metodo che la classe espone, oltre al costruttore, il getter del ParameterManager che contiene e il main utilizzato per fini di test, è il metodo createJavaFiles. Sono inoltre presenti tre metodi helper, createClassTree, e due diversi manageTree.

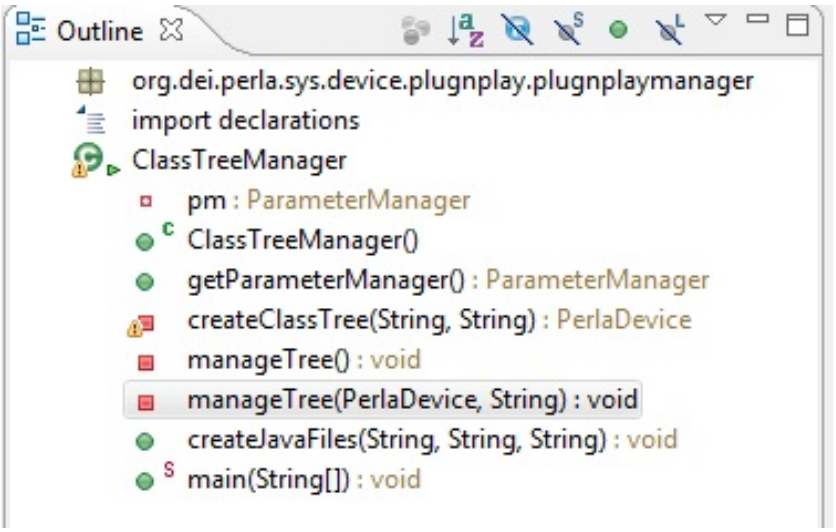

Figure 2.13: Outline della classe ClassTreeManager.

- createJavaFiles(String pkg, String xml, String fileDir):

```
public void createJavaFiles(String pkg,
  String xml, String fileDir) throws Exception {
  manageTree(createClassTree(pkg, xml), fileDir);
}
```

  questo è l'unico punto d'accesso a ClassTreeManager, si limita a chiamare manageTree passandogli come primo parametro il risultato dell'operazione di unmarshalling (sarà un perlaDevice) e come secondo la cartella in cui l'utente vuole che siano inseriti i file .java che verranno creati.

- createClassTree(pkg, xml) si rimanda a 2.6.3 per maggiori dettagli sul funzionamento di questo metodo.

- manageTree(PerlaDevice pd, String fileDir):

```
private void manageTree(PerlaDevice pd, String fileDir){
  this.pm=new ParameterManager(pd);
```

```
if (fileDir==null)
  manageTree();
else {
  pm.setFileDir(fileDir);
  manageTree();
  }
}
```

viene inizializzato il ParameterManager pm presente nella classe passandogli come parametro il PerlaDevice generato dall'unmarshaling; poi viene modificata o meno la cartella di destinazione dei file, dipendentemente dal fatto che sia stata specificata una fileDir diversa da null o meno; quindi viene chiamato manageTree.

- manageTree():

```
private void manageTree(){
  this.pm.printDevice();
  this.pm.toFile();
  this.pm.printLogFile();
  }
```

questo metodo invoca i metodi necessari alla creazione automatica del codice messi a disposizione dal ParameterManager pm presente nella classe.

2.7.2 ParameterManager

La procedura per la creazione automatica del codice è la seguente: per ogni struttura si costruisce una stringa, alla quale ogni metodo chiamato concatenerà la propria stringa. Il risultato sarà una stringa contenente il codice di un intera classe java. Per ogni struttura si procederà alla creazione della propria stringa. Le singole stringhe saranno poi stampate su file .java appositamente creati.

I metodi principali di ParameterManager consistono in:

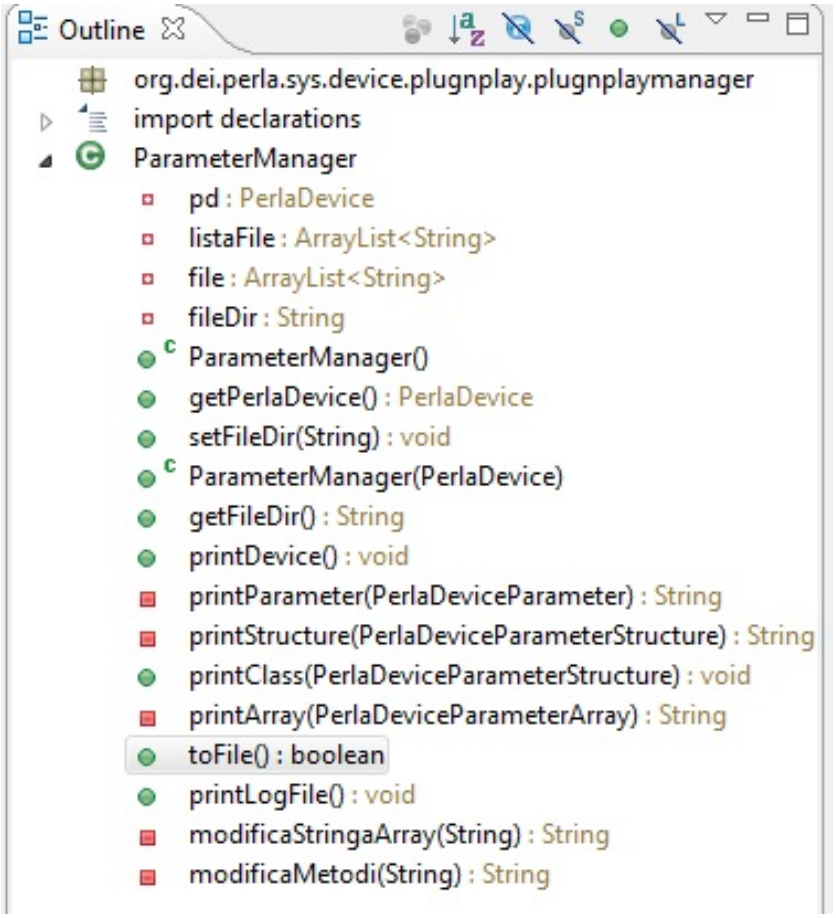

Figure 2.14: Outline della classe ParameterManager.

- printDevice(): viene chiamato da ClassTreeManager. Da il via al processo di creazione del codice chiamando a sua volta printClass()
- printClass(PerlaDeviceParameterStructure str): si occupa della costruzione di una stringa contenente il codice di un intera classe Java. Per far capire come funzioni il processo di costruzione di una stringa partendo dall'albero degli oggetti generati da JAXB viene riportato per questo metodo il codice.

```
public void printClass(PerlaDeviceParameterStructure str) {
  String s = "package org.dei.perla.sys.device.fpc.;" +
  + "\n\nimport org.dei.perla.utils."+
  + "dataconverter.annotations.*;"+
  "\nimport org.dei.perla.utils.dataconverter.enums.*;"+
  "\n\n@StructInfo(endianness = ";
  if (str.getEndianess().toString().equals("BIG_ENDIAN"))
    s += "Endianness.BIG_ENDIAN";
  else
    s += "Endianness.LITTLE_ENDIAN";
  s += ", totalStructSize = " +
    + str.getSize() + ")\npublic class " +
    + str.getType() + "{\npublic " + str.getType() + "(){}";
  for (PerlaDeviceParameter p : str.getParameterElement())
    s += "\n" + printParameter(p);
  for (PerlaDeviceParameterStructure st : str
    .getParameterStructureElement())
    s += "\n" + printStructure(st);
  for (PerlaDeviceParameterArray a :
    str.getParameterArrayElement())
    s += "\n" + printArray(a);
  s += "\n}";
  file.add(s);
}
```

Come si vede dal codice riportato printClass() riceve una PerlaDeviceParameterStructure, e in base agli elementi contenuti nella struttura, procede alla chiamata dei metodi gestori dei diversi tipi di parametro. Tali metodi sono ricorsivi, quindi le stringhe da loro restituite saranno concatenate alla stringa s, che verrà quindi salvata nell'ArrayList file. Prima di procedere alla chiamata di tali metodi vengono innanzitutto inseriti gli import necessari e le annotazioni proprie di una classe, ossia: Endianess (BIG_ENDIAN o LITTLE_ENDIAN) e totalStructSize. Si può notare che, come ci si aspetta da una classe Java, il nome della classe corrisponde al tipo che definisce.

- printParameter(PerlaDeviceParameter p): produce una stringa contenente il codice per un PerlaDeviceParameter, una variabile in Java. La stringa restituita sarà cosi costituita:
  annotazioni + private nomeParametro; + getter + setter. Le annotazioni per un parametro semplice sono sempre del tipo: @SimpleField(size, sign (SIGNED o UNSIGNED))

- printArray(PerlaDeviceParameterArray array): produce una stringa contenente il codice per un ParameterElementArray, un array in Java. Il metodo è ricorsivo, perché si possono avere array di parametri semplici, array di liste, array di strutture o array di array. Il metodo printArray procederà quindi alla chiamata ricorsiva dei vari metodi gestori di parametri a seconda di quello che troverà all'interno dell'elemento array. La stringa risultante sarà:
  annotazioni + private tipoArray[] nomeArray; + getter + setter
  Le annotazioni sono: @FixedLengthArray(length). Il metodo printArray utilizza altri due metodi privati, modificaMetodi per modificare i suoi getter e i setter e modificaStringaArray per decidere dove posizionare "[]" nella stringa.

- printStructure(PerlaDeviceParameterStructure str): come detto in precedenza una struttura corrisponde ad una nuova classe Java, essendo un

parametro composto. Quindi, printStructure procederà alla creazione della stringa contenente il codice della struttura da inserire nella classe "padre" (N.B.: con padre non si intende una relazione di ereditarietà in Java, ma semplicemente che nel documento XML una struttura era contenuta dentro l'altra), e invocherà di nuovo printClass(PerlaDeviceParameterStructure str) per la creazione del codice della nuova classe. Restituirà quindi una stringa così formata:
annotazioni + private tipoStruttura nomeStruttura; + getter + setter
Le annotazioni sono: @CompositeField(size)

In ParameterManager esistono inoltre altri due metodi, che si occupano di generare i file .java relativi alle stringe appena costruite e a scrivere un ulteriore file che contiene i nomi di tutti i file generati. Questi metodi sono:

- toFile(): si occupa di creare per ogni stringa creata, un file .java e di scrivere al suo interno il contenuto di quella stringa. Le stringhe contenenti il codice delle classi vengono prese ciascuna dall'ArrayList file, e passate all'oggetto FileOutputStream, che provvederà alla creazione del file corrispondente.

```
public boolean toFile() {
  int index;
  for (String s : file) {
    FileOutputStream o;
    String nomeFile;
    try {
      File dir = new File(fileDir + this.pd.getName());
      dir.mkdirs();
      nomeFile = s.substring(s.indexOf("public class ")
        + "public class ".length(), s.indexOf("{"));
      listaFile.add(nomeFile);
      index = s.indexOf(";");
      // completa nome del package col nome del device
```

```
      s = s.substring(0, index) + pd.getName() +
        + s.substring(index);
      // sposta il contenuto della stringa nel file
      // che ha come nome il nome della struttura
      o = new FileOutputStream(new File(fileDir +
        + this.pd.getName() +
        + "/" + nomeFile + ".java"));
    } catch (FileNotFoundException e) {
      e.printStackTrace();
      return false;
      }
    try {
      o.write(s.getBytes());
    } catch (IOException e) {
      e.printStackTrace();
      return false;
    }
  }
return true;
}
```

Per ogni stringa, viene creato nella cartella fileDir un file che ha il nome corrispondente al nome della classe Java. Prima di copiare il contenuto della stringa di questa ne viene modificata la dichiarazione del package, in modo che il risultato sia coerente. Poi si procede alla copia dei dati contenuti nella stringa.

- printLogFile(): si occupa di creare un file contenente i nomi delle classi (file .java) create in precedenza da toFile(). Il file creato verrà anch'esso salvato nella cartella specificata in fileDir.

2.8 Compilazione

Una volta generato il codice sorgente relativo ad un dispositivo, per poterlo effettivamente utilizzare a run time, è necessario compilarlo. Ovviamente, non è possibile richiedere l'intervento dell'utente per la compilazione del codice, sia perché sarebbe contro l'idea stessa di "Plug & Play", sia perché vanificherebbe completamente lo sforzo fatto nel generare automaticamente il codice.

Appare quindi evidente l'esigenza di introdurre nel progetto sviluppato un componente che riesca a compilare a run time il codice generato. A livello di progettazione questo componente è stato pensato come un oggetto che, ricevuto un file Java in ingresso, ne generi il suo file .class o, se ci sono degli errori, stampi sullo standard output quali problemi ha riscontrato. Inoltre è necessario che i file .class vengano prodotti all'interno di un path specifico; infatti quello che si mira ad ottenere è una libreria di "driver", uno per ogni tipo di device che nel corso della storia del sistema si è connesso ad esso. In questo modo l'intera procedura di lettura del file XML, generazione dei sorgenti Java e compilazione si renderà necessaria solo nel caso in cui un nuovo dispositivo di tipo sconosciuto al sistema PerLa tenterà di connettersi ad esso.

Per come è stato progettato, il compilatore da sviluppare è molto semplice, e certamente sarebbe sufficiente poter invocare il compilatore di default presente nella JVM. Fortunatamente l'API di Java viene incontro a quest'esigenza in quanto mette a disposizione l'interfaccia JavaCompiler ed una serie di altri tool per invocare il compilatore di default, tutti contenuti nel package javax.tools.

Seguendo questa strada, l'intero processo della compilazione a run time si semplifica molto, e infatti il componente sviluppato consta di un solo metodo. D'altra parte, si deve supporre che un compilatore sia effettivamente presente nella JVM del sistema su cui viene eseguito il progetto; questo comporta che non sia sufficiente installare solo Java Runtime Environment, ma si debba installare l'intero JDK.

Di fatto, la classe Compiler espone il metodo
compileFiles(File f, String[] options)

che opera in questo modo:

- Prende il controllo del compilatore e del gestore dei file di default della JVM tramite

  ```
  JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
  StandardJavaFileManager fileManager = compiler.
    getStandardFileManager(null, null, null);
  ```

  fileManager è un componente necessario per la gestione dei file oggetto.

- Controlla se sono state fornite delle opzioni al compilatore; nel caso non ne sia stata data nessuna, vengono inizializzate delle opzioni di default

  ```
  if (options == null || options.length == 0)
    options = new String[] { "-d","bin" };
  ```

  l'opzione -d bin viene data di default perché bin è la cartella dove poi si andranno a trovare tutti i driver. Ad esempio un sensore di tipo XY, i cui sorgenti .java si trovano in org/dei/perla/sys/device/fpc/XY, avrà i suoi driver contenuti in bin/org/dei/perla/sys/device/fpc/XY.

- Viene creata una lista di file da compilare

  ```
  List<File> sourceFileList = new ArrayList<File>();
    for (File file : f.listFiles())
      if (file.getName().contains(".java"))
        sourceFileList.add(file);
  ```

  Si suppone che al metodo compileFiles venga passato una cartella contenente uno o più file .java, in questo passo questi vengono aggiunti alla lista di file da compilare.

- Il fileManager e il compiler creano una unità di compilazione a partire dalla lista di file, viene creata una CompilationTask e infine questa viene compilata

```
Iterable<? extends JavaFileObject> compilationUnits =
  fileManager.getJavaFileObjectsFromFiles(sourceFileList);
CompilationTask task = compiler.getTask(null, fileManager,
  null, Arrays.asList(options), null, compilationUnits);
boolean result = task.call();
```

A questo punto sono stati generati i class file di tutti i file .java contenuti in una data cartella, e di default si andranno a trovare nella cartella bin.

2.9 Wrapping

Come accennato nell'introduzione, gli strati superiori del middleware PerLa comunicano sempre con i Functionality Proxy Component, o FPC. Un FPC, che non è altro che la rappresentazione logica di un dispositivo per mezzo di un oggetto Java, risulta essere l'interfaccia per mezzo della quale il sistema comunica con un dispositivo ad esso connesso.

Per questo motivo è necessario che tutte le classi generate a partire da un documento XML valido relativo ad un singolo dispositivo si registrino all'interno di singolo un FPC, e che questo sia poi in grado di interrogarle.

La gestione dei dati di un FPC avviene per mezzo della classe astratta FPCDataStructure, la quale contiene a sua volta il metodo getAttributeByName(String parName), che è il metodo da utilizzarsi per recuperare il valore di un attributo a partire dal nome specificato in parName. Questo metodo è il punto cardine dell'interfaccia, infatti esso viene invocato dagli strati superiori del middleware specificando il nome del parametro e deve ritornare il valore di quest'ultimo.

L'implementazione di questo metodo ha posto dei problemi, principalmente dovuti ai due motivi seguenti:

- Due documenti XML diversi producono oggetti Java diversi. Naturalmente questo è vero non solo a livello semantico, ma anche a livello sintattico. Ad esempio, il descrittore sensore1.xml che contiene la struttura di tipo A darà luogo a all'oggetto A, mentre il descrittore sensore2.xml contenente la struttura di tipo B darà origine all'oggetto B.

- Un singolo documento XML può generare più oggetti Java. Come si è visto nei capitoli precedenti infatti, un documento sensore.xml che contiene una struttura di tipo A, la quale a sua volta ha annidata al suo interno un'altra struttura di tipo B viene in tradotta in Java come due classi pubbliche, una di nome A, contenete l'attributo di tipo B, e la classe B. Questo significa che di fatto non è sufficiente estendere FPCDataStruc-

ture aggiungendole un parametro che rappresenti l'analogo di un elemento radice, perché finché un dispositivo non prova a connettersi al sistema non è noto né di che tipo questo sia, né quanti siano gli oggetti prodotti dal suo documento XML.

Inoltre, l'FPC deve soddisfare certi requisiti di performance; gli strati superiori del middleware infatti continuano ad interrogare gli FPC alla ricerca di dati aggiornati. Se quindi questi non rispondono in maniera sufficientemente veloce, rischiano di agire da collo di bottiglia compromettendo di fatto il corretto funzionamento dell'intero sistema.

Una prima soluzione valida sembrerebbe quella di fare un pesante uso del meccanismo di reflection. Sarebbe sicuramente molto semplice, avendo il nome dell'attributo da cercare, utilizzare il metodo getMethod di java.lang.reflect per trovare il getter/setter d'interesse e successivamente invocarlo. Di fatto però, questo aggiungerebbe un pesante overhead alle performance, dovuto appunto al costo intrinseco dell'utilizzo della reflection [8]. Per questo motivo si è cercato di evitare il più possibile l'uso della reflection.

Il problema è stato quindi affrontato nel modo seguente:

- FPCDataStructure è stata modificata, aggiungendole il parametro root di tipo AbstractData
- è stata creata FPCDataStructureImpl, una classe che estende FPCDataStructure e che implementa il metodo getAttributeByName
- tutte le classi generate dinamicamente sono state modificate in modo che risultino delle estensioni di AbstractData
- è stata creata AbstractData, la classe che contiene due Hashtable, le quali contengono a loro volta rispettivamente i metodi getter e setter dei parametri della classe.

In questo modo, quando vengono create delle nuove classi, queste sono sempre l'estensione di AbstractData e non di tipi diversi; inoltre, AbstractData mette

a disposizione le due Hashtable con cui si può cercare il parametro voluto senza dover ricorrere alla reflection. Le due Hashtable vengono popolate tramite il metodo fillTables:

```
Method[] methods = this.getClass().getMethods();
String name = "", tipoMetodo = "";
for (Method m : methods) {
  name = m.getName();
  tipoMetodo = name.substring(0, 3);
  if (tipoMetodo.equals("get")) {
    gettersTable.put(name, m);
    } else if (tipoMetodo.equals("set")) {
      settersTable.put(name, m);
      }
  }
```

Come si può vedere fillTables fa uso della reflection, ma di fatto viene invocato una tantum, pertanto è stato giudicato un costo accettabile.

Una volta creato l'FPC è possibile utilizzare getParameterByName, che opera come segue:

- invoca sul suo oggetto root il metodo lookForParameter, passandogli la stringa che ha ricevuto (che identifica il parametro da cercare)
- lookForParameter legge la stringa che ha ricevuto e cerca nelle Hashtable se il getter del parametro è presente. La funzione termina quando la stringa termina e restituisce il valore del parametro che è stato specificato, o lancia un eccezione se sono stati riscontrati dei problemi (ad es. il parametro specificato non esiste).
- poiché la funzione getParameterByName deve restituire un Constant<?>, del risultato trovato deve essere effettuato il casting al sottotipo di Constant corretto. Questo viene fatto secondo la seguente politica:
  - se l'oggetto ha valore null, viene restituita una ConstantNull

- se l'oggetto ha specificata l'annotazione mapping, viene eseguito un casting al tipo specificato nel valore dell'annotazione. In questo caso l'utente deve aver precedentemente definito la classe come estensione di Constant, altrimenti viene lanciata un'eccezione
- se i casi precedenti non si verificano, allora viene controllato il tipo dell'oggetto restituito da lookForParameter, e viene restituita l'estensione di Constant adatta (ad es. ConstantInteger se l'oggetto era un int, ConstantFloat se era un float, ecc.)
- se nessuna delle estensioni di Constant risulta essere adatta, c'è stato qualche problema e viene lanciata un eccezione

Purtroppo, non si è riuscito ad evitare di utilizzare la reflection all'interno di lookForParameter. Se è infatti vero che il metodo cerca all'interno dell Hashtable il getter specificato, una volta che l'ha trovato lo chiama per mezzo di

```
parameter = gettersTable.get("get" + parName).
  invoke(this, (Object[]) null);
```

Peraltro, utilizzare la reflection solo per invocare un metodo è relativamente (molto) meno costoso che effettuare anche la ricerca del metodo specificato ogni volta. Nonostante questo metodo sia da testare in una situazione di carico reale per verificarne l'efficacia, allo stato attuale sembra comunque una buona soluzione.

2.9.1 Stringa parName

In questa sezione viene presentata la grammatica da utilizzare nelle stringhe parName utilizzate per specificare il parametro che getParameterByName deve cercare. Infatti le classi generate dinamicamente non condividono il namespace, pertanto questo può dar luogo a diverse ambiguità. Ad esempio, data la seguente situazione:

```
public class A {
  private int a;
  private B b;
  private int[] c;
}

public class B {
  private int a;
  private int b;
}
```

deve essere possibile recuperare sia i parametri di A che di B.

Per questo motivo si suppone che l'utente che chiama getParameterByName utilizzi la notazione puntata specificata di seguito:

- volendo trovare un parametro all'interno della classe root, è sufficiente specificare il nome del parametro. Ad es. volendo ottenere 'a' o 'b' contenuti in A è sufficiente chiamare getParameterByName("a") piuttosto che getParameterByName("b")
- nel caso di array, per ottenere il puntatore all'array bisogna specificare il nome, mentre per ottenere un elemento dell'array la sintassi è la solita (nomearray[indice]). Ad es. per ottenere 'c' basta chiamare getParameterByName("c"), per ottenere c[5] bisogna chiamare getParameterByName("c[5]")
- per parametri contenuti in classi annidate, va utilizzata la sintassi classeContenente_classeContenuta, eventualmente più volte. Ad es. per il parametro a contenuto in B (chiamata b) bisogna che l'utente chiami getParameterByName("b_a")

Rispettando queste regole l'utente ha la certezza che qualsiasi parametro richiesto sia raggiungibile, indipendentemente dall'overloading effettuato sui nomi.

Chapter 3

Testing

In questo capitolo vengono mostrati i test effettuati sui componenti sviluppati. Come al solito, la prima parte è dedicata ai driver, mentre la seconda al "Plug & Play".

3.1 Codifica Pacchetti

Vengono definiti e inizializzati dei buffer, due vuoti (tempBuff e outBuff) e uno con dei dati (inBuff). Dopodiché viene chiamata chMgrSend in modo che i dati contenuti in inBuff vengano messi su outBuff con la formattazione opportuna e si verifica attraverso USERBUFFER_dump (funzione che scrive in console il contenuto del buffer passato per parametro) che questo effettivamente accade. Poi viene testata chMgrRcv coi dati presenti in outBuff e ancora una volta attraverso USERBUFFER_dump(inBuff) si controlla che funzioni correttamente. Infine viene inizializzato un Adapter Client con connessi tre nodi, e vengono testate (sempre per mezzo del dump in console) AdapterClientRcv e AdapterClientSend. Nelle sezioni successive vengono analizzati esaustivamente i casi di test presentati, prima quelli riguardanti il Channel Manager (3.1.1), poi quelli riguardanti l'Adapter Client (3.1.2).

3.1.1 Channel Manager

In questa sezione vengono mostrati i due test effettuati sul Channel Manager, per controllare che l'invio e la ricezione di dati vada a buon fine. Scopo dei due test è mostrare che inviando un pacchetto e successivamente ricevendolo, il suo contenuto rimane invariato.

chMgrSend() - Invio di un pacchetto dal Channel Manager

In questo primo test si vuole simulare l'arrivo di un pacchetto di dati da inviare al Channel Manager. In particolare vengono mostrate le fasi di elaborazione ed invio dei dati. Nella tabella 3.1 vengono mostrate le condizioni in cui si svolge il test. Il sequence diagram mostra i passaggi effettuati nell'esecuzione della funzione. Come prima fase vengono inizializzati i buffer coinvolti:

```
USER_BUFFER_init(&buff1);
USER_BUFFER_init(&buff2);
USER_BUFFER_init(&buff3);
```

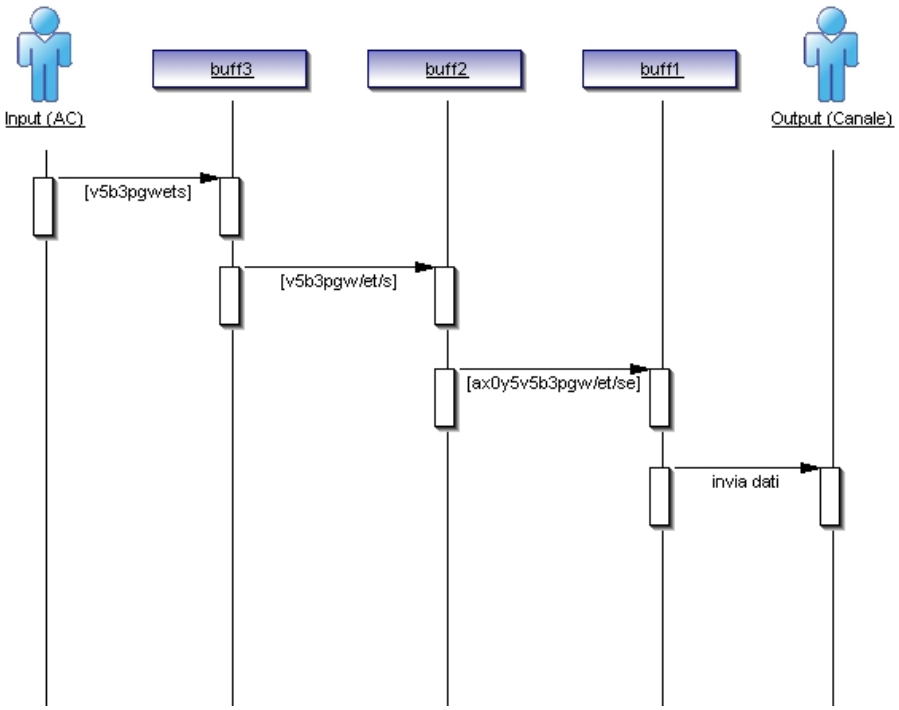

Figure 3.1: Sequence Diagram raffigurante l'invio di un pacchetto.

```
USER_BUFFER_setBuffer(&buff3, v3, 10);
USER_BUFFER_setBuffer(&buff1, v1, 40);
USER_BUFFER_setBuffer(&buff2, v2, 40);
```

buff3 viene inizializzato con questi dati:

```
{'v','5','b','3','p','g','w','e','t','s'};
```

buff1 e buff2 sono invece vuoti. Dopodiché viene eseguita:

```
chMgrSend(&buff3, &buff1, &buff2, false, address1, address2);
```

I primi tre parametri sono i buffer su cui deve operare il Channel Manager, false indica che il pacchetto e addressful e che pertanto il Channel Manager dovrà aggiungere al pacchetto gli indirizzi contenuti nei parametri address1 e address2. Come visto precedentemente ChMgrSend chiama:

Table 3.1: Dati di input del test

Modalità di invio	Indirizzo Sorgente (Ch. Mgr. che chiama funzione)	Indirizzo Destinatario (Ch. Mgr. che riceve dati)	Dati da inviare
ADDRESSFUL	'0'	'5'	'v5b3pgwets'

```
codifica (buff3,buff2);
```

la quale:

- inizializza delle variabili che conterranno i caratteri delimitatori, come mostrato nella tabella 3.2

Table 3.2: Caratteri delimitatori

Nome var.	startal	startaf	e	sp	addrfrom	addrto
Valore	's'	'a'	'e'	'/'	'x'	'y'
Descr.	inizio addressless	inizio addressful	fine	escape	indirizzo sorgente	indirizzo destinatario

- Controlla byte per byte i dati contenuti all'interno di buff3 (2.3).

Quando codifica termina buff2 conterrà [v5b3pgw/et/s]. Dopodiché viene chiamata:

```
header(buff2, buff1, false, address1, address2);
```

che svolge le seguenti operazioni:

- Il pacchetto deve essere addressful, pertanto il primo byte in uscita corrisponderà al delimitatore di inizio addressful. Vengono anche aggiunti su buff1 i byte che compongono e delimitano gli indirizzi sorgente e destinazione (nel nostro caso x0y5).

- ```
  USER_BUFFER_commitData(buff1, &(buff2->buffer[buff2->cursor ]),
  buff2->lastDataIndex  buff2->cursor);
  ```

  Quest'istruzione sposta l'intero contenuto di buff2 in buff1.

- ```
  uint8_t e = 'e';
  USER_BUFFER_commitData(buff1,&e,1);
  ```

  Viene inviato il delimitatore di fine pacchetto su buff1.

A questo punto buff1 sarà un vettore di 15 byte costituito nel modo seguente: [ax0y5v5b3pgw/et/se].

chMgrRcv() - Ricezione di un pacchetto dal Channel Manager

In questo test si simula la ricezione del pacchetto precedentemente inviato da parte di un secondo Channel Manager; verrà mostrato quindi come il Channel Manager in questione processa il pacchetto per estrarne solo i dati utili. La tabella 3.3 mostra le condizioni in cui si svolge il test.

Come nel caso precedente come prima cosa si inizializzano le strutture dati:

```
USER_BUFFER_init(&buff2);
USER_BUFFER_init(&buff3);
USER_BUFFER_setBuffer(&buff2, v2, 40);
USER_BUFFER_setBuffer(&buff3, v3, 10);
```

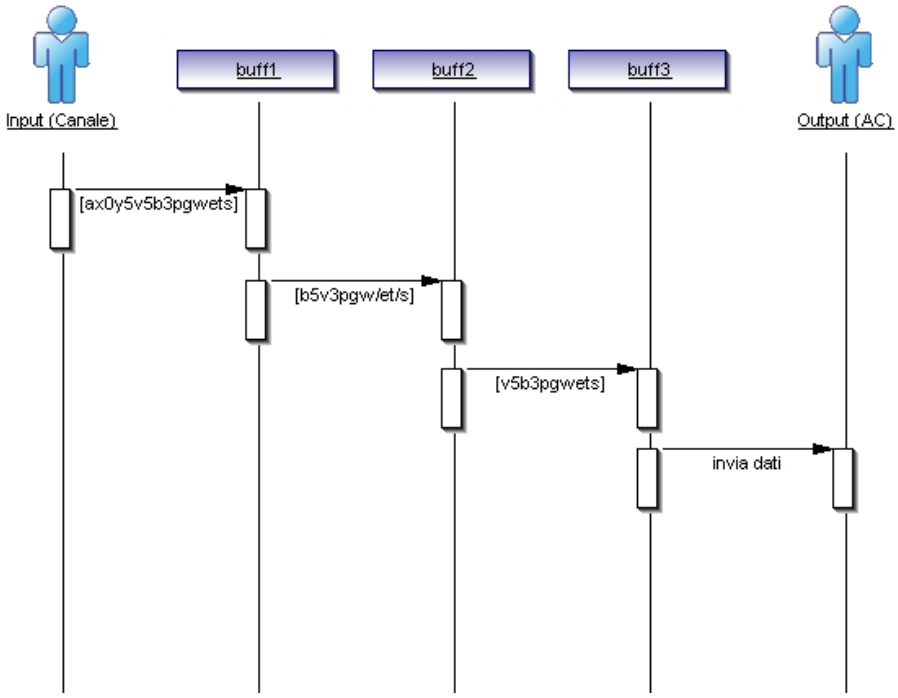

Figure 3.2: Sequence Diagram raffigurante la ricezione di un pacchetto.

Table 3.3: Dati di input del test

Tipo pacchetto	Indirizzo Sorgente (Ch. Mgr. che ha inviato il pacchetto)	Indirizzo Destinatario (Ch.Mgr. che chiama la funzione)	Dati ricevuti
ADDRESSFUL	'0'	'5'	'ax0y5v5b 3pgwets'

Nonostante fossero già state inizializzate nell'esempio precedente si ripete questo passo perché potrebbero essere rimaste "sporche" e quindi potrebbero inficiare l'esito del test. Questa operazione non è necessaria durante il normale utilizzo di un Channel Manager, viene svolta qui solo perché vengono riutilizzati gli stessi buffer del test precedente. Inoltre si può notare come lo Userbuffer buff1 non venga reinizializzato in quanto in questo test lo si vuole riutilizzare come ingresso (e quindi si vogliono mantenere i dati con cui è stato riempito nel corso del test precedente). Poi si passa ad eseguire:

```
chMgrRcv(&buff1,&buff3,&buff2, address2);
```

buff1 è il buffer che la funzione utilizza come ingresso, buff2 è il solito buffer temporaneo, buff3 è il buffer su cui la funzione andrà a scrivere e address2 è l'indirizzo associato al Channel Manager. Come prima cosa chMgrRcv() chiama:

```
rimuoviHeader(buff3, buff2, address2);
```

la quale:

- Inizializza la variabile locale:

  ```
  uint8_t startaf = 'a';
  ```

- Verifica che il pacchetto sia addressless o meno controllando il primo byte. Nel nostro caso il pacchetto è addressful, il che significa che la funzione controllerà anche che l'indirizzo contenuto nel pacchetto corrisponda ad address2 e, nel caso questa verifica vada a buon fine, saltare l'indirizzo.

- `USER_BUFFER_commitData(buff2, &(buff1->buffer[buff1->cursor]), buff1->lastDataIndex - buff1->cursor  1);`

  Quest'istruzione copia il contenuto di buff1 in buff2 partendo dal byte a cui punta il cursore di buff1 e fermandosi prima dell'ultimo byte (che comunque conterrà il delimitatore di fine pacchetto).

All'uscita di rimuoviHeader(), buff2 conterrà [b5v3pgw/et/s]. Dopodiché viene chiamata:

```
decodifica(buff2, buff3);
```

che:

- Controlla la struttura del pacchetto come è stato mostrato nelle sezioni precedenti (2.3). Ad es. "/e" diventa "e", "//" diventa "/", ecc.

Quando tutto il pacchetto è stato controllato decodifica() e chMgrRcv() terminano e il contenuto di buff3 sarà [v5b3pgwets]; si può notare che il contenuto del buffer corrisponde a come era stato inizializzato buff3 nel corso del primo test, ovvero l'invio di un pacchetto da parte di un Channel Manager e la sua ricezione da parte di un altro ha mantenuto invariato il contenuto del pacchetto stesso. Si può pertanto concludere che entrambi i test siano andati a buon fine.

3.1.2 Adapter Client

Come accennato precedentemente, in questa sezione vengono mostrati i due casi di test effettuati sull'Adapter Client. Come prima fase si procede ad inizializzare le strutture dati utilizzate:

```
AdapterClient ac;
uint8_t i;
for (i = 0; i<3; i++) {
  USER_BUFFER_init(&(vcInBuff[i]));
  USER_BUFFER_setBuffer(&vcInBuff[i], vcIn[i], 40);
  USER_BUFFER_init(&(vcOutBuff[i]));
```

```
  USER_BUFFER_setBuffer(&vcOutBuff[i], vcOut[i], 40);
  }
adapterClientInit(&ac, &buff3, &buff1, 3, &vcInBuff, &vcOutBuff,
  vci);
```

Durante il ciclo for vengono inizializzati gli Userbuffer dei dispositivi connessi all'Adapter Client; si è deciso di simulare un Adapter Client con 3 dispositivi, quindi dovranno essere utilizzati 6 buffer. Dopodiché si chiama adapterClientInit() che "riempie" l'Adapter Client ac con:

- buff1/buff3: Userbuffer utilizzati nei casi di test precedenti; nel seguito verranno utilizzati dall'Adapter Client per comunicare con l'esterno. Gli Userbuffer non vengono re-inizializzati perché si vogliono riutilizzare i dati contenuti in buff1.
- 3: size dell'Adapter Client, ovvero il numero di dispositivi connessi.
- vcInBuff/vcOutBuff: vettori di Userbuffer associati ad ogni dispositivo connesso. Sono tutti inizializzati con dei vettori vuoti.
- vci: vettore dei VCI associati ai dispositivi connessi. Contiene {'4', '2', '3'}.

NB: la funzione adapterClientInit() imposta tutti i VCI come BVCI. L'Adapter Client infatti, come visto nelle sezioni precedenti, non è autorizzato a creare dei canali virtuali, ma solo canali virtuali di binding.

AdapterClientRcv() - Ricezione di un pacchetto da parte dell'Adapter Client

In questo caso di test si vuole simulare l'arrivo di un pacchetto all'Adapter Client e il suo indirizzamento al nodo corretto. La tabella 3.4 mostra le condizioni in cui si svolge il test.

La funzione chiamata è:

```
adapterClientRcv(ac);
```

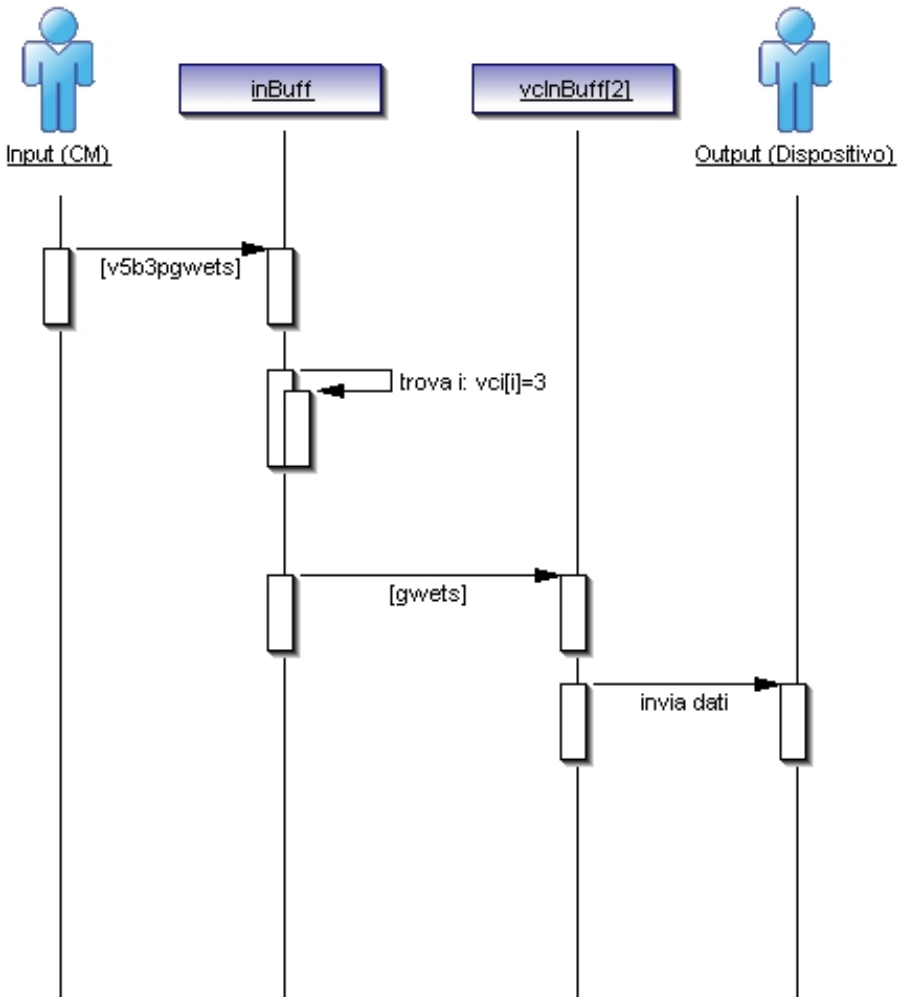

Figure 3.3: Sequence Diagram raffigurante la ricezione di un pacchetto.

Table 3.4: Dati di input del test

Tipo pacchetto	BVCI	VCI	Dati input
ADDRESSFUL	'3'	'5'	'v5b3pgwets'

che riceve come parametro l'Adapter Client appena inizializzato. La funzione svolge le seguenti operazioni:

- Inizializzazione delle variabili locali come mostrato nella tabella 3.5

Table 3.5: Variabili locali

Nome var.	VCI	BVCI	payload	vcinum	bvcinum
Valore	'v'	'b'	'p'		
Descr.	delimitatore precedente il vci	delimitatore precedente il bvci	delimitatore precedente i dati	byte che conterrà il vci	byte che conterrà il bvci

- Viene effettuato un ciclo che scandisce il contenuto del pacchetto arrivato. All'interno del ciclo vengono svolte le seguenti operazioni:
  - `if (AC.buff3->buffer[AC.buff3->cursor] == payload)`
    `exit=true;`

    Quando viene incontrato il byte 'p' il ciclo termina.
  - `if (AC.buff3->buffer[AC.buff3->cursor] == vci)`
    `vcinum = AC.buff3->buffer[AC.buff3->cursor + 1];`

    Quando viene incontrato il byte 'v', il successivo byte conterrà il VCI del canale virtuale utilizzato dal dispositivo destinatario e di conseguenza verrà memorizzato in vcinum. In questo caso vcinum='5'.

–
```
if (AC.buff3->buffer[AC.buff3->cursor] == bvci){
    bvcibool=true;
    bvcinum = AC.buff3->buffer[AC.buff3->cursor + 1];
    }
```

  Se si incontra il byte 'b' il successivo byte conterrà il BVCI del canale virtuale di binding utilizzato dal dispositivo destinatario e di conseguenza verrà memorizzato in bvcinum. In questo caso bvcinum='3'. Viene anche impostato bvcibool a true.

NB: mentre i delimitatori 'p' e 'v' devono essere sempre presenti in un pacchetto affinché venga gestito correttamente dall'Adapter Client, per quanto riguarda 'b' la questione è differente. Si troverà infatti solo il delimitatore 'v' nel caso in cui la fase di binding sia terminata (e quindi il canale è univocamente identificato dal VCI). Si potranno invece trovare sia 'v' che 'b' nel caso in cui si passi dalla fase di binding alla fase "normale"; in questo caso il pacchetto viene inviato al dispositivo associato al canale contenuto in bvcinum e dopodiché l'Adapter Client provvederà a modificare la propria tabella vci[] con il VCI contenuto in vcinum. Il pacchetto analizzato rientra proprio in quest'ultima categoria, pertanto il pacchetto verrà indirizzato all'i-esimo dispositivo con vci[i]='3' e dopodiché vci[i] verrà impostato a '5'.

- Una volta trovato il VCI del canale virtuale a cui è destinato il pacchetto, la funzione cerca nella tabella vci[] dell'Adapter Client quale sia il dispositivo (tra tutti quelli connessi) effettivamente coinvolto nella comunicazione. Per farlo si confronta vci[i] con vcinum/bvcinum (nel nostro caso bvcinum dato che bvcibool=true). Quando l'i-esimo dispositivo viene individuato (vci[2]=3=bvcinum) la funzione chiama:

```
USER_BUFFER_commitData(ac.vcInBuff[2],
  &(AC.buff3->buffer[AC.buff3->cursor]),
  AC.buff3->lastDataIndex  AC.buff3->cursor);
```

che sposta tutti i dati rimanenti (a questo punto saranno solo i dati utili in quanto cursor di inBuff punta al byte successivo a 'p') da buff3 a vcInBuff[2].

- Infine:

```
ac.vci[2] = '5';
ac.bvci[2] = false;
```

Impostano il nuovo valore di VCI contenuto in vcinum e indicano che non è più un canale di binding (ac.bvci[2]= false).

Il contenuto di vcInbuff[2] sarà, a questo punto, [gwets].

AdapterClientSend() - Invio di un pacchetto da parte dell'Adapter Client

In questo caso di test si vuole simulare l'invio di un pacchetto da parte di un nodo e si vuole mostrare come l'Adapter Client aggiunga le informazioni necessarie alla sua gestione. In particolare, si simula che sia il nodo che controlla vcInbuff[2] e vcOutbuff[2] a voler inviare i dati.

Per farlo si procede nel modo seguente:

```
USER_BUFFER_commitData(vcOutBuff[2],
  &(vcInBuff[2]->buffer[vcInBuff[2]->cursor]),
  vcInBuff[2]->lastDataIndex  vcInBuff[2]->cursor);
```

Con questa istruzione si sposta il contenuto del buffer vcInBuff[2] (ovvero [gwets]) in vcOutBuff[2]. Quindi si chiama:

```
adapterClientSend(vcOutBuff[2], &ac);
```

dove vcOutBuff[2] è lo Userbuffer che vuole inviare i dati e ac è il solito Adapter Client inizializzato precedentemente.

La funzione svolge le seguenti operazioni:

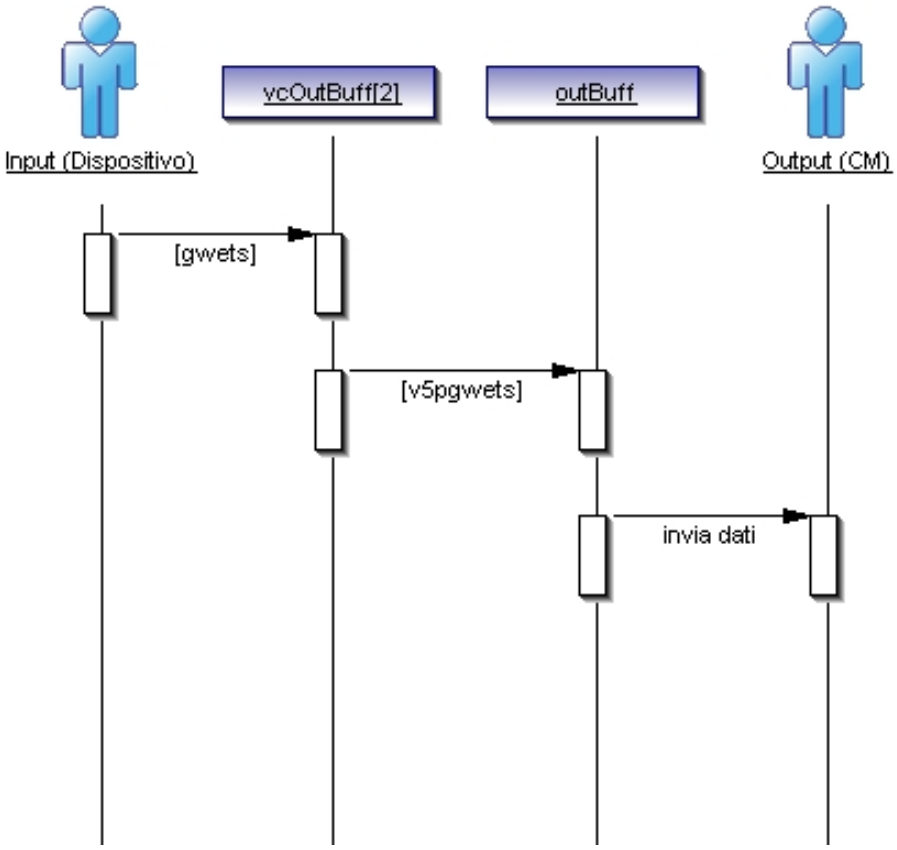

Figure 3.4: Sequence Diagram raffigurante l'invio di un pacchetto.

Table 3.6: Variabili locali

Nome var.	VCI	BVCI	payload	vcinum
Valore	'v'	'b'	'p'	
Descr.	delimitatore precedente il vci	delimitatore precedente il bvci	delimitatore precedente i dati	byte che conterrà il vci

- Inizializzazione delle variabili locali come mostrato nella seguente tabella 3.6.

- Viene effettuato un ciclo che mira a individuare quale sia il VCI del nodo che vuole inviare i dati:

  ```
  if (vcOutBuff[i] == in)
    vcinum = ac->vci[i];
  ```

  Nel nostro caso vcinum verrà impostato a '5'.

- ```
  if (ac->bvci[i - 1] == false)
    USER_BUFFER_commitData(ac->buff1, &vci, 1);
  else
    USER_BUFFER_commitData(ac->buff1, &bvci, 1);
  ```

  Questa istruzione invia 'b' o 'v' a seconda che il canale virtuale del nodo che vuole inviare i dati sia di binding o meno. Nel nostro caso il canale del sensore in considerazione non è più di binding, e infatti bvci[2] vale false; quindi verrà inviato 'v'.

- ```
  USER_BUFFER_commitData(ac->buff1, &vcinum, 1);
  USER_BUFFER_commitData(ac->buff1, &payload, 1);
  ```

  La prima istruzione invia il VCI (ovvero 5) mentre la seconda il delimitatore di payload ('p').

- ```
  vcOutBuff[2]->cursor = 0;
  USER_BUFFER_commitData(ac->buff1,
    &(vcOutBuff[2]->buffer[vcOutBuff[2]->cursor]),
  vcOutBuff[2]->lastDataIndex  vcOutBuff[2]->cursor);
  ```

  Con questa coppia di istruzioni viene inviato l'intero contenuto del buffer vcOutBuff[2] a buff1.

All'uscita della funzione lo Userbuffer buff1 conterrà [v5pgwets].
  ```

3.2 Plug & Play

Come è stato esposto nei capitoli precedenti, nel corso dello sviluppo del sistema sono stati utilizzati diversi strumenti. In particolare, sono stati utilizzati il linguaggio XML per la definizione dello XML Schema e per la conseguente scrittura dei documenti XML relativi a ciascun dispositivo, la libreria JAXB per l'Unmarshaling dei documenti XML nelle classi di supporto e la chiamata al compilatore di default (supposto javac) che deve essere presente nel sistema che andrà ad ospitare PerLa. Pertanto, al momento di decidere quali test case del sistema implementare, sono state effettuate le seguenti scelte:

- Naturalmente, non c'è nessun test per quanto riguarda la parte di definizione dello schema XML, in quanto è una parte di progettazione che ha portato alla definizione di un documento.
- Nessun test sull'uso di JAXB: si suppone infatti che il package sviluppato da Sun sia stato già adeguatamente testato.
- Test case relativo alla compilazione: questo test case è invece stato implementato, in quanto non ci si limita a chiamare il compilatore di default ma come si è visto gli si passano diverse opzioni e vengono compilati più file in una volta.

Al fine di testare il progetto sviluppato in modo sistematico, i test case sono stati definiti tramite Junit 3.0. I test case individuati sono CompilerTest, RuntimeSourceCodeTest e FPCWrapTest e sono contenuti nel package test. I dati di test relativi agli input e ai risultati attesi sono contenuti in nelle cartelle test/testDeviceSingolaStruttura e test/testDevice, come mostrato nella figura. TestDevice e TestDeviceSingolaStruttura sono due dispositivi fittizi molto semplici, di cui sono stati creati i descrittori XML.

Nei prossimi capitoli verrà mostrato il processo di testing per come è stato implementato. In 3.2.1 viene testato il processo di generazione a run time del codice Java, in 3.2.2 si mostra il funzionamento del test case relativo alla compi-

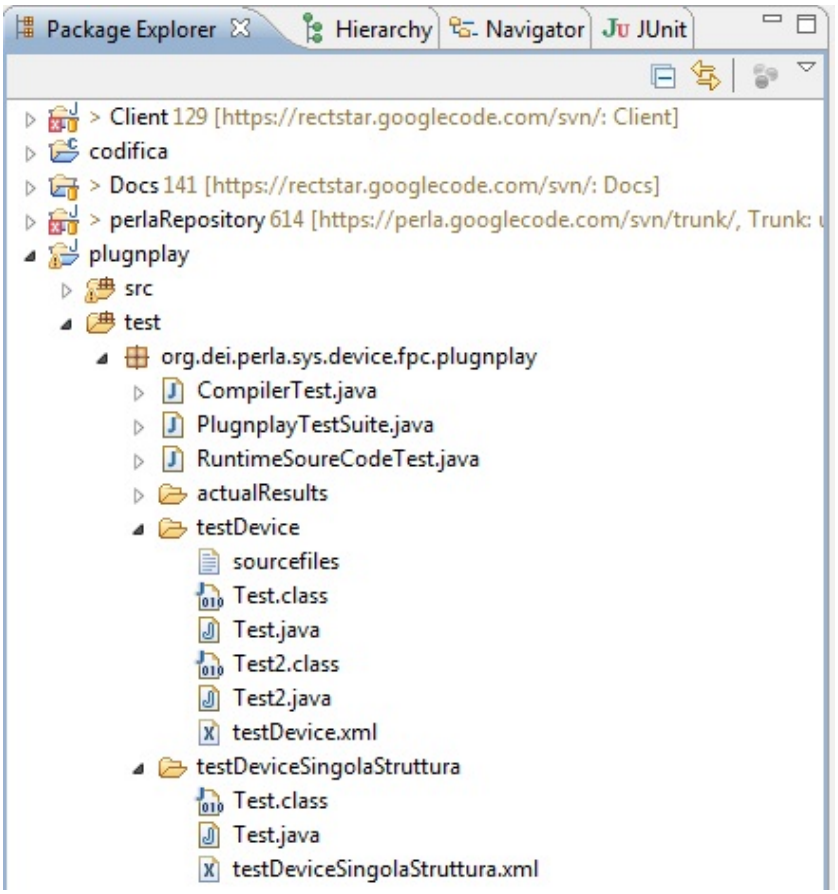

Figure 3.5: Contenuto del package test.

lazione a run time, in 3.2.3 viene presentato il caso di test relativo al wrapping degli oggetti generati in un singolo FPC.

L'ultimo capitolo (3.2.4) non riguarda dei test case JUnit ma bensì la prova del sistema su un dispositivo reale, DSPic.

3.2.1 RuntimeSourceCodeTest

Questo test case è relativo alla generazione dinamica di file .java a partire da un documento XML. Viene ora mostrato il funzionamento del codice che relativo al test case, contenuto nella classe RuntimeSourceCodeTest. Innanzitutto, la funzione di setUp() crea un nuovo ClassTreeManager e imposta la cartella che dovrà contenere i file generati nel corso del testing:

```
protected void setUp() throws Exception {
  super.setUp();
  ctm = new ClassTreeManager();
  fileDir = new
    String("test/org/dei/perla/sys/device/" +
    + "fpc/plugnplay/actualresults/");
}
```

Una volta fatto ciò, viene invocata la funzione che si occupa di fare il testing vero e proprio, testSourceCodeCreation(). Questa funzione genera i file .java per ogni file .xml presente in test/org/dei/perla/sys/device/fpc/plugnplay, nel modo seguente:

```
public void testSourceCodeCreation() {
  //per ogni test case
  for (File f : new
    File("test/org/dei/perla/sys/device/fpc/plugnplay/").
    listFiles()) {
    if (f.isDirectory()) {
      // trova xml relativo al test case e genera i file da
      //testare in src/test
      for (File g : f.listFiles()) {
        if (g.getName().contains(".xml")) {
          try {
            // crea file java nella cartella
```

```
// src/test/actualresults
ctm.createJavaFiles(
  "org.dei.perla.sys.device.plugnplay." +
  + "xsdgeneratedclasses",
  g.getName(), fileDir);
} catch (Exception e) {
  e.printStackTrace();
  fail();
  }
```

in questa prima parte vengono creati i file .java per mezzo di createJavaFiles. Questi si troveranno in ognuno in una cartella denominata col nome del test case che si sta considerando. Il test fallisce nel caso in cui sorga qualche eccezione in questa fase.

```
// per ogni file (cartella) in src/test/actualresults
for (File h : new File(fileDir).listFiles())
  //se si tratta dello stesso test case
  if (h.getName().equals(f.getName()))
    //per ogni file contenuto nel test case che sto esaminando
    for (File i : h.listFiles())
      // per ogni file contenuto nella cartella
      // dei risultati attesi
      for (File k : f.listFiles()) {
        //se hanno lo stesso nome
        if (i.getName().equals(k.getName())) {
```

successivamente, i file devono essere confrontati coi risultati attesi. Di conseguenza in questa parte vengono individuati quali file confrontare.

```
if (i.getName().equals(k.getName())) {
  //vengono confrontati
  f1 = new byte[(int) i.length()];
```

```
  f2 = new byte[(int) k.length()];
  try {
    ActualResultStream = new BufferedInputStream(
      new FileInputStream(i));
    expectedResultStream = new BufferedInputStream(
      new FileInputStream(k));
    actualResultStream.read(f1);
    expectedResultStream.read(f2);
    } catch (IOException e) {
      e.printStackTrace();
      fail("IO Exception");
    }
  assertEquals(new String(f1),new String(f2));
}
```

infine i file individuati vengono confrontati. Vengono creati due array di byte f1 e f2 delle dimensioni dei due file i e k da testare; i due file vengono aperti e letti, il loro contenuto viene spostato rispettivamente in f1 e f2; dopodiché assertEquals controlla che f1 e f2 abbiano effettivamente lo stesso contenuto.

3.2.2 CompilerTest

Questo test case è volto a provare il funzionamento del processo di compilazione a run time. Come si intuisce facilmente dal nome, si occupa di testare la classe Compiler e in particolare il metodo che svolge la compilazione, compileFiles. Il test case sviluppato opera nel modo seguente:

- Nel corso del SetUp() vengono svolte le seguenti operazioni:

  ```
  super.setUp();
  c = new Compiler();
  inputDir = new File("src/org/dei/perla/sys/device/fpc/");
  expectedResultDir = new File(
    "test/org/dei/perla/sys/device/fpc/plugnplay/");
  // compila file .class nella cartella di testing
  options = new String[] {"-d", "test/org/dei/perla/sys/" +
    + "device/fpc/plugnplay/actualResults/"};
  ```

  queste operazioni sono dedicate all'impostazione della cartella da cui il compilatore deve prendere i file .java, della cartella da cui il test case prenderà i file .class compilati precedentemente per confrontarli con quelli generati dal compilatore e delle opzioni da dare al compilatore; in pratica options contiene la cartella in cui il compilatore dovrà inserire i risultati.

- Una volta fatto ciò viene chiamato il metodo testCompiler() il quale

  ```
  // per ogni caso da testare
  for (File f : inputDir.listFiles())
    // per ogni risultato atteso
    for (File g : expectedResultDir.listFiles())
      // se si tratta dello stesso caso di test
      if (f.getName().equals(g.getName())) {
        // compila
        c.compileFiles(f, options);
  ```

come prima cosa compila i file .java relativi al test case che si sta considerando.

- ```
  for (File k : new File(dir + f.getName()).listFiles())
    // per ogni file di risultato
    for (File j : g.listFiles())
      if (k.getName().equals(j.getName())) {
  ```

  individua i file da confrontare, cercando tra i file compilati ora e i file dei risultati attesi (precedentemente compilati, naturalmente invocando il compilatore javac esternamente).

- ```
  // confronta contenuto dei file
  f1 = new byte[(int) k.length()];
  f2 = new byte[(int) j.length()];
  try {
    actualResultStream = new BufferedInputStream(
      new FileInputStream(k));
    expectedResultStream = new BufferedInputStream
      new FileInputStream(j));
    actualResultStream.read(f1);
    expectedResultStream.read(f2);
    } catch (IOException e) {
      e.printStackTrace();
      fail("IO Exception");
    }
  assertEquals(new String(f1), new String(f2));
  ```

  come nel caso del test precedente, vengono creati due array di byte nei quali viene copiato il contenuto dei due file da confrontare. Il test ha successo nel caso i due array siano identici.

3.2.3 FPCWrapTest

Con l'implementazione di questo test case si vuole verificare il corretto funzionamento del wrapping delle classi generate in un FPC, e la capacità di quest'ultimo di gestire i dati contenuti nelle classi ad esso associate. Questo test case, a differenza dei precedenti, svolge delle operazioni elementari, sia perché l'FPC offre un'interfaccia molto semplice, sia perché non essendo ancora stato implementato il metodo setAttributeByName non è possibile riempire una classe generata dinamicamente, pertanto tutte le annidate all'interno di quella più esterna sono impostate a null e tutti i valori sono impostati con dei valori di default. Il test case è stato implementato nel modo seguente:

- creazione di due fpc, uno (fpc) contenente la classe generata in testDeviceSingolaStruttura, e un secondo (fpc2) contenente le classi generate in testDevice

```
fpc = new FPCDataStructureImpl(
  (AbstractData) new org.dei.perla.sys.
  device.fpc.testDeviceSingolaStruttura.Test());
fpc2 = new FPCDataStructureImpl(
  (AbstractData) new org.dei.perla.sys.
  device.fpc.testDevice.Test());
```

- vengono effettuate chiamate a getParameterByName su diversi attributi, e confrontati i risultati con quelli attesi.

```
// intero in testDeviceSingolaStruttura
assertEquals(fpc.getAttributeByName("p").getValueInt(), 0);
// intero in testDevice
assertEquals(fpc2.getAttributeByName("x").getValueInt(), 0);
// array (vuoto) in testDevice
assertEquals(fpc2.getAttributeByName("y").toString(), "NULL");
// struttura (vuota) in testDevice
```

```
assertEquals(fpc2.getAttributeByName("q").toString(), "NULL");
// nome non esistente
try {
  fpc.getAttributeByName("fakeVar");
} catch (NullPointerException e) {
  assertTrue(true);
}
```

3.2.4 Un caso reale: DSPic

In questa sezione si vuole presentare l'utilizzo del sistema su un dispositivo reale, il DSPic.

Come prima cosa si è definito l'XML relativo ai dati prodotti dal dispositivo. I dati prodotti sono stati letti dall'header file (.h) che era stato utilizzato per programmare il DSPic. L'XML così costruito viene salvato come "testDSPic.xml". Poiché i dati che DSPic produce sono molti, per ragioni di spazio viene mostrato nella figura 3.6 il contenuto di una sola delle 3 strutture annidate, lowSamplingRateData.

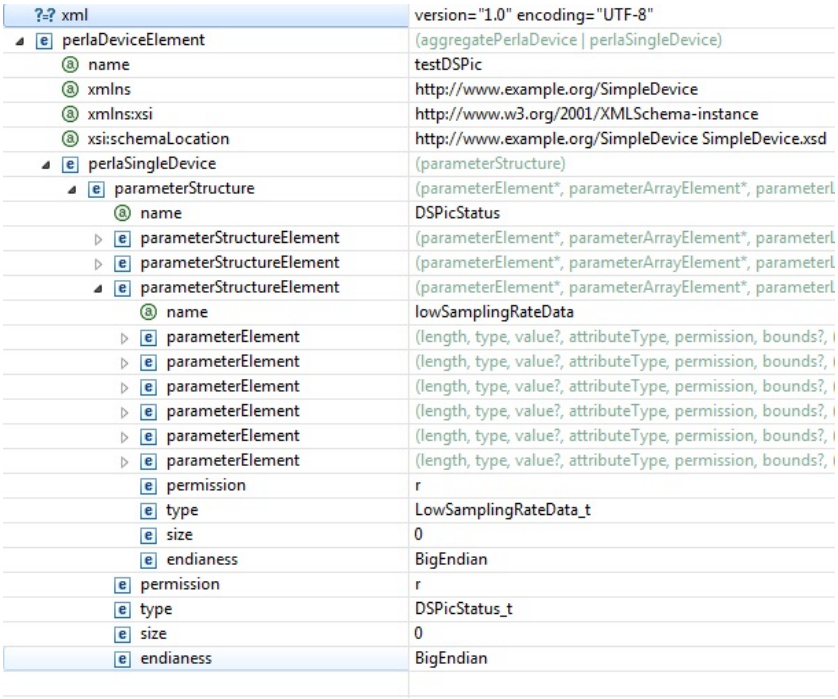

Figure 3.6: XML del DSPic.

Come si può vedere la struttura lowSamplingRateData contiene sei parametri elementari (tutti interi con segno). Poiché l'XSD e le classi di supporto sono già stati creati in precedenza, il passo successivo è la generazione del codice a run time. Viene pertanto invocato il metodo:

```
createJavaFiles("org.dei.perla.sys.device.
  plugnplay.xsdgeneratedclasses", "testDSPic.xml", null);
```

I file generati si trovano, come ci si aspettava, in org.dei.perla.sys.device.fpc.testDSPic. Nella figura 3.7 viene mostrato una parte dell'output di createJavaFiles relativo alla struttura lowSamplingRateData, che si trova nel file LowSamplingRate-Data_t.java

```
1  package org.dei.perla.sys.device.fpc.testDSPic;
2
3  import org.dei.perla.utils.dataconverter.annotations.*;
4  import org.dei.perla.utils.dataconverter.enums.*;
5  import org.dei.perla.sys.device.fpc.AbstractData;
6
7  @StructInfo(endianness = Endianness.BIG_ENDIAN, totalStructSize = 0)
8  public class LowSamplingRateData_t extends AbstractData {
9  public LowSamplingRateData_t(){super();}
10
11 @SimpleField(size = 2, sign = Sign.SIGNED)
12 private int temperature;
13
14 public int gettemperature()
15 {    return temperature ;
16 }
17  public void settemperature(int temperature){
18      this.temperature= temperature;
19 }
20 @SimpleField(size = 2, sign = Sign.SIGNED)
21 private int attitudeX;
22
23 public int getattitudeX()
24 {    return attitudeX ;
25 }
26  public void setattitudeX(int attitudeX){
27      this.attitudeX= attitudeX;
28 }
29 @SimpleField(size = 2, sign = Sign.SIGNED)
30 private int attitudeY;
31
32 public int getattitudeY()
33 {    return attitudeY ;
34 }
35  public void setattitudeY(int attitudeY){
```

Figure 3.7: Codice Java relativo a LowSamplingRateData_t.

I file generati vengono quindi compilati tramite la chiamata a:

```
c.compileFiles("src/org/dei/perla/sys/device/fpc/testDSPic", null);
```

che si occupa di compilare tutti i file .java che trova nella cartella specificata. I class file si troveranno in bin/org/dei/perla/sys/device/fpc/testDSPic. I class file generati sono mostrati nella figura 3.8.

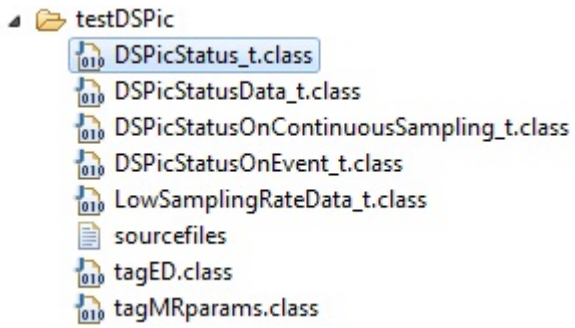

Figure 3.8: Class file generati per il DSPic.

L'ultimo passaggio necessario perché i file siano utilizzabili dagli strati superiori del middleware PerLa è il wrapping in un FPC. Questo viene fatto tramite:

```
FPCDataStructureImpl fpc = new FPCDataStructureImpl(
  (AbstractData) new org.dei.perla.sys.device.
  fpc.testDSPic.DSPicStatus_t());
```

Ora l'FPC è inizializzato, e i suoi dati possono essere recuperati tramite getAttributeByName.

Chapter 4

Conclusioni

Il lavoro svolto si è focalizzato su due parti del middleware PerLa. Per la parte di realizzazione dei driver per gestire il canale di comunicazione, il lavoro può considerarsi concluso. Adapter Client e Channel Manager sono allo stato attuale capaci di supportare qualsiasi tipo di dispositivo (a patto che sia dotato di un compilatore C), abilitandolo alla comunicazione con un sistema PerLa. Naturalmente possono essere estesi con ulteriori funzionalità, ma allo stato attuale svolgono tutte le operazioni strettamente necessarie.

Un discorso diverso va invece fatto per la parte di "Plug & Play". Se infatti si può dire che la progettazione del sistema può considerarsi completa, non tutte le parti sono state ancora implementate. Nell'XSD era infatti stata prevista la possibilità di avere nodi aggregati e che all'interno di un generico nodo possa essere presente una lista di elementi, oltre che parametri semplici, array e strutture. Di fatto questi due elementi non sono ancora gestiti (vengono ignorati durante la generazione a run time del codice). Un'altra parte che dev'essere ancora implementata è la funzione setParameterByName in FPCDataStructure, ovvero il metodo preposto all'impostazione dei parametri di un FPC.

Questo non significa però che il sistema allo stato attuale sia non funzionante, o presenti notevoli lacune. Le parti non ancora implementate infatti sono quelle la cui necessità è stata giudicata a priorità minima. Generalmente infatti, un

sensore è singolo e non un aggregato di più sensori; sembra anche un assunzione ragionevole dire che un generico sensore (solitamente dalle capacità di calcolo limitate) difficilmente produrrà una lista di elementi, e anche se lo facesse, è possibile modellizzarla con un array sovradimensionato. Per quanto riguarda setParameterByName invece, sicuramente la funzione è necessaria, in quanto in sua assenza è impossibile per l'utente inviare dati ai dispositivi. Anche in sua assenza però i nodi possono comunque inviare al sistema i dati che producono, garantendo pertanto ugualmente il funzionamento di quest'ultimo.

Bibliography

[1] Perla. pervasive language. http://perla.dei.polimi.it.

[2] Schreiber F.A., Camplani R., Fortunato M., and Marelli M. Perla: A declarative language and middleware for pervasive systems. 2008.

[3] Schreiber F.A., Camplani R., Fortunato M., and Marelli M. Perla - pervasive language.

[4] Schreiber F.A., Camplani R., Fortunato M., Marelli M., and Pacifici F. Perla: a data language for pervasive systems. 2008.

[5] Maesani A., Magni C. P., and Padula E. Low level architecture of the perla middleware. 2008.

[6] Java architecture for xml binding (jaxb). http://java.sun.com/developer/technicalarticles/webservices/jaxb/.

[7] Java architecture for xml binding - binding compiler (xjc). http://java.sun.com/webservices/docs/1.6/jaxb/xjc.html.

[8] Trail: The reflection api. http://java.sun.com/docs/books/tutorial/reflect/.