ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

UN MIDDLEWARE JAVA PER L' INTEGRAZIONE DI TECNOLOGIE AD AGENTI CON LA PIATTAFORMA ARDUINO

Elaborata nel corso di: Sistemi Operativi

Tesi di Laurea di:
LORENZO CORNEO

Relatore:
Prof. ALESSANDRO RICCI

Co-relatori:
Dott.Ing ANDREA SANTI

ANNO ACCADEMICO 2010–2011
SESSIONE III

PAROLE CHIAVE

Arduino

Middleware

Java

Agenti

Sistemi Embedded

Ai miei cari, che sempre mi hanno sostenuto e sempre
mi sosterranno.

Indice

Introduzione **ix**

1 Arduino: caratterisiche hardware e software **1**
1.1 Caratteristiche hardware 1
1.1.1 Ethernet Shield 2
1.2 Sensori ed Attuatori 3
1.3 Arduino IDE 4
1.4 Caratteristiche software 4

2 Progetto e sviluppo del middleware **9**
2.1 Analisi dei requisiti 9
2.1.1 Glossario dei termini usati per l'applicazione 9
2.1.2 Modello del dominio 10
2.1.3 Casi d' uso 12
2.2 Analisi del problema 13
2.2.1 Modello del dominio. 13
2.2.2 Architettura logica. 15
2.2.3 Modello di interazione 17
2.3 Modalità di interazione elaboratore-board 17
2.3.1 PULL mode 17
2.3.2 PUSH mode 20
2.4 Progetto 20
2.4.1 Progetto di VirtualBoard 20
2.4.2 Progetto del Middleware lato elaboratore 23
2.5 Progetto del Comunicator 25
2.6 Progetto del protocollo elaboratore-board 27
2.7 Progetto del middleware lato Arduino 28

3 Middleware lato Arduino **31**
3.1 Realizzazione server UDP/TCP 31
3.1.1 Server UDP 31
3.1.2 Server TCP 32
3.2 Gestione del protocollo 33
3.3 Gestione notifiche PUSH 34

4 Uso del middleware e collaudo **37**
4.1 Configurazione del sistema 37
4.2 Threshold in modo PULL 40
4.3 Threshold in modo PUSH 41
4.4 Applicazione di test 43
4.5 Perfomance 46

5 Integrazione di tecnologie ad agenti **49**
5.1 Introduzione agli agenti 49
5.2 JaCa 50
5.3 Livello applicativo in JaCa 52
5.4 Un altro esempio applicativo 55

6 Conclusioni **59**

Introduzione

Obiettivo della tesi è lo studio e la realizzazione prototipale di un middleware Java che permetta di controllare sistemi basati su Arduino, una piattaforma molto flessibile e sempre più diffusa oggi per la sperimentazione e realizzazione a basso costo di sistemi embedded. Il middleware permette da un lato di poter sviluppare i programmi di controllo in un linguaggio moderno, di alto livello come Java, dall'altro di poter sperimentare l' utilizzo di linguaggi di programmazione e tecnologie di ricerca - nella fattispecie ad agenti. Per il progetto si sono posti i seguenti requisiti:

- Realizzare una board virtuale che rappresenti i dispositivi fisici effettivamente montati su Arduino e che garantisca all' utente finale di accedervi tramite un metodo *get*.
- Realizzare il middleware per la comunicazione tra l' elaboratore e Arduino con i protocolli UDP e TCP. Il middleware deve inoltre supportare la modalità PUSH e la modalità PULL.
- Realizzare un programma di test per verificare il corretto funzionamento del middleware.
- Integrare nel progetto la tecnologia ad agenti al fine di rendere autonomo il sistema.
- Realizzare un programma di test per il sistema ad agenti.

Concludo questa introduzione con qualche nota tecnica. La piattaforma utilizzata è Arduino UNO messa in serie con l' Ethernet Shield che null'altro è se non la scheda di rete che mette in comunicazione la board con l'elaboratore. L'ambiente di sviluppo lato microcontrollore è l'IDE ufficiale della piattaforma Arduino giunto alla versione stabile (versione 1.0). Questo

IDE è scritto in Java ed offre una vasta gamma di API che coprono tutte le funzionalità proposte dalla piattaforma. L'IDE usato lato elaboratore è il noto Eclipse INDIGO (versione 3.7) con il quale sarà implementato il middleware e tutto il resto del sistema. Per la modellazione UML viene utilizzato Enterprise Architect (versione 8.0).

Capitolo 1

Arduino: caratterisiche hardware e software

Arduino è una piattaforma di prototipazione elettronica open-source che si basa su hardware e software flessibili e facili da utilizzare. Arduino è basato su una scheda di I/O e su un ambiente di sviluppo, che usa programmi in C/C++ da fare eseguire sulla scheda, il cui microprocessore è programmabile tramite il linguaggio Wiring; Arduino è in grado di comunicare con l'ambiente attraverso una moltitudine di sensori analogici e digitali azionando contemporaneamente qualsiasi tipo di attuatore[4].

1.1 Caratteristiche hardware

Arduino UNO è una scheda di input/output basata sul microcontrollore ATmega328. Questo dispone di 14 pin I/O, 6 ingressi analogici, un oscillatore al cristallo con frequenza 16MHz, una connesione USB, un cavo di alimentazione ed un tasto di reset. Per una utilizzazione veloce basta collegare la board tramite cavo USB di tipo A-B ad un dispositivo alimentato (es. computer), oppure utilizzare un convertitore AC/DC, oppure una pila. Nel caso di utilizzo di una pila i connettori vanno inseriti nei piadini Gnd e Vin. I pin di alimentazione sono:

- **Vin**: il voltaggio di input della scheda quando si utilizza un'alimentazione esterna[1].

[1] Si ricorda che l'alimentazione della board tramite cavo USB di tipo A-B è di 5V.

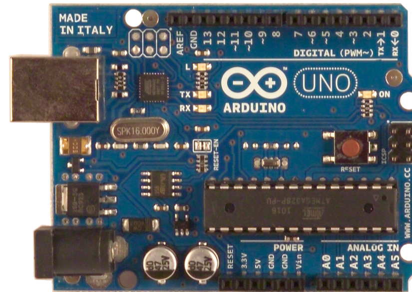

Figura 1.1: Arduino UNO.

- **5V**: l'alimentazione regolata che fornisce corrente alla board e ai suoi componenti. Proviene dal pin Vin attraverso un regolatore della scheda oppure può essere fornita dal cavo USB di tipo A-B o da un'altra alimentazione fissata a 5V.

- **3V3**: sorgente a 3.3V generata dal chip FTDI della scheda. La corrente fornita non può essere maggiore di 50mA.

- **Gnd**: pin della messa a terra.

La memoria interna dell'ATmega328 è di 32 KB (dei quali 0.5 KB usati per il bootloader). In aggiunta sono presenti 2 KB di SRAM e 1KB di EEPROM.

1.1.1 Ethernet Shield

Arduino Ethernet Shield[2] consente alla board Arduino di connettersi a internet ed è basata sul chip ethernet Wiznet W5100 che fornisce un insieme di protocolli di rete compatibili sia con TCP che UDP. Altra importante funzionalità dell'Ethernet Shield è un lettore di schede microSD che consente di avere una grande capacità in termini di memoria disponibile che sopperisce alla limitata memoria interna del microcontrollore. Si pensi ad

[2]Gli shield sono delle piastre che si collegano alla piattaforma Arduino e consentono di estendere le funzionalità di base del prodotto.

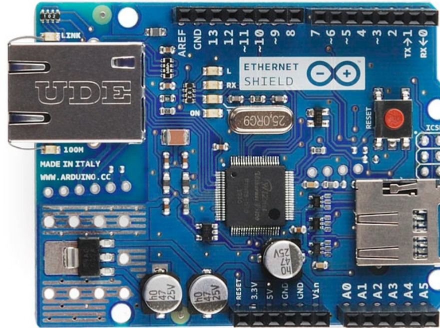

Figura 1.2: Arduino Ethernet Shield.

uno scenario in cui Arduino agisca da Web Server che ha bisogno di costruire pagine dinamiche di grande dimensione: con la scarsa memoria interna sarebbe impossibile.

1.2 Sensori ed Attuatori

I sensori sono componenti elettronici in grado di convertire una grandezza fisica in una grandezza elettrica misurabile da Arduino. I sensori presi in considerazione per lo svolgimento del progetto sono di due tipi: digitali ed analogici. Un sensore digitale si può trovare solamente in due stati: acceso o spento. Un sensore analogico, invece, realizza una variazione continua in base al valore della grandezza che sta misurando. Nel progetto il sensore è quella entità che ci permette di acquisire un valore sul quale effettuare opportune elaborazioni.

Gli antagonisti dei sensori sono gli attuatori: dispositivi che a partire da un segnale elettrico producono un effetto nello spazio fisico. Anche gli attuatori si dividono in analogici e digitali. Un attuatore digitale può assumere valori discreti [0, 1] , un attuatore analogico potrà assumere valori nell' intervallo discreto [0, 255].

1.3 Arduino IDE

L' ambiente di sviluppo integrato (IDE) di Arduino è un' applicazione multipiattaforma scritta in Java, ed è derivata dall' IDE creato per il linguaggio di programmazione Processing e per il progetto Wiring [5]. Per permettere la stesura del codice sorgente, l' IDE include un editor di testo dotato inoltre di alcune particolarità, come il syntax highlighting, il controllo delle parentesi, e l' indentazione automatica. L' editor è inoltre in grado di compilare e lanciare il programma eseguibile in una sola passata con un solo click. L' ambiente di sviluppo integrato di Arduino è fornito di una libreria software C/C++ chiamata “Wiring”(dall' omonimo progetto): la disponibilità della libreria rende molto più semplice implementare via software le comuni operazioni di input/output. I programmi di Arduino sono scritti in C/C++. A supporto dell' IDE sono presenti librerie che regolano il corretto funzionamento della board e di tutti i suoi componenti. Sia le librerie che l' IDE sono spesso soggette a frequenti cambiamenti da parte degli sviluppatori e degli stessi utenti.

Inizialmente per il progetto è stata usata la versione 0019 dell' IDE che presentava diversi problemi nella gestione del protocollo UDP. La versione finale utilizzata è la versione stabile 1.0 che ha apportato sostanziali cambiamenti nelle librerie (specialmente in quella UDP dove cambia il modo di leggere e scrivere pacchetti dati) e ha inoltre portato il cambio dell' estensione del formato Arduino da .pde a .ino. Nel passaggio tra le versioni dell'IDE sono stati risolti diversi problemi e sono stati necessari dei cambiamenti al codice.

1.4 Caratteristiche software

Per creare un file eseguibile dalla piattaforma non è richiesto all' utente di scrivere un programma in C, ma solo di definire due funzioni:

- **setup()**: funzione che viene invocata una sola volta, all' inizio del programma, e di solito viene usata per definire i settaggi iniziali.

- **loop()**: funzione chiamata ripetutamente la cui esecuzione viene interrotta solamente dalla mancanza di alimentazione.

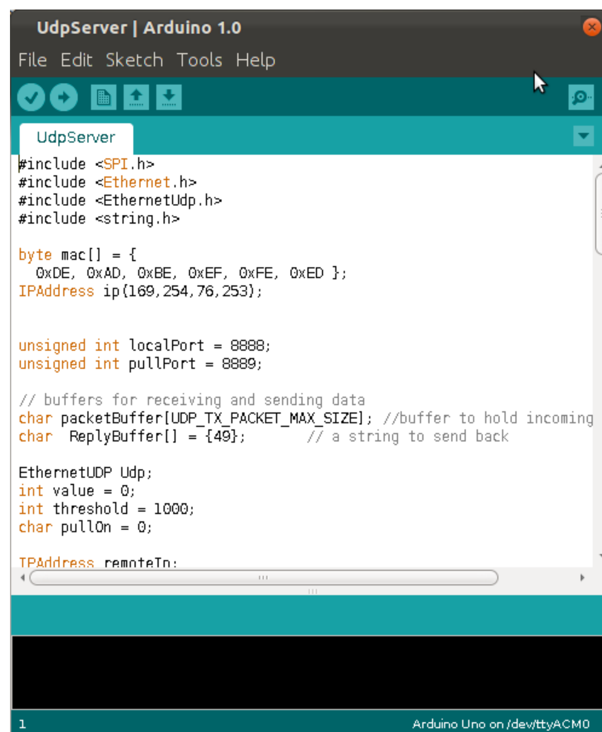

Figura 1.3: Screenshot dell' IDE Arduino .

Il metodo più facile per capire il funzionamento di Arduino è un esempio pratico. Di seguito si riporta il codice di un server di echo con il protocollo UDP.

```
#include <SPI.h>
#include <Ethernet.h>
#include <EthernetUdp.h>

byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 0, 177);
byte gateway[] = {192, 168, 0, 1};

// Local port to listen on
unsigned int localPort = 8888;

// Buffer for receiving and sending data
char packetBuffer[UDP_TX_PACKET_MAX_SIZE];

// An EthernetUDP instance to let us send and receive packets over UDP
EthernetUDP Udp;

void setup() {
  // start the Ethernet and UDP:
  Ethernet.begin(mac,ip, gateway);
  Udp.begin(localPort);

  Serial.begin(9600);
}

void loop() {
  // If there's data available, read a packet
  int packetSize = Udp.parsePacket();
  if(packetSize)
  {
    Serial.print("Received packet of size ");
    Serial.println(packetSize);
    Serial.print("From ");
    IPAddress remote = Udp.remoteIP();
    for (int i =0; i < 4; i++)
    {
```

```
      Serial.print(remote[i], DEC);
      if (i < 3)
      {
        Serial.print(".");
      }
    }
    Serial.print(", port ");
    Serial.println(Udp.remotePort());

    // Read the packet into packetBufffer
    Udp.read(packetBuffer,UDP_TX_PACKET_MAX_SIZE);
    Serial.println("Contents:");
    Serial.println(packetBuffer);

    // Send a reply, to the IP address and port that sent
    // us the packet we received
    Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
    Udp.write(packetBuffer);
    Udp.endPacket();
  }
  delay(10);
}
```

Capitolo 2

Progetto e sviluppo del middleware

In questo capitolo si fornisce la descrizione del processo di progettazione e sviluppo del middleware.

2.1 Analisi dei requisiti

Si riporta di seguito l' insieme degli artefatti che vogliono definire i requisiti funzionali e che devono essere soddisfati dall'applicazione.[1]

2.1.1 Glossario dei termini usati per l'applicazione

Il glossario serve per non creare ambiguità nel lessico utilizzato nella produzione dell' applicazione.

Middleware	Sistema software che funge da intermediario tra il software lato Arduino e il software lato elaboratore. Tramite questo l' utente accede ai dispositivi virtuali e configura il sistema.
Comunicator	Canale di comunicazione usato dal middleware per l' interazione tra i due dispositivi (elaboratore e board).
VirtualDevice	Astrazione virtuale del dispositivo fisico posto sulla board Arduino. Tramite questo l' utente accede alle funzionalità del dispositivo.
Sensore	I sensori sono componenti elettronici in grado di convertire una grandezza fisica in una grandezza elettrica misurabile da Arduino. Possono essere digitali se il loro dominio ammette solo 2 valori (alto / basso o zero / uno), analogici se ammettono valori discreti nell' intervallo [0, 1023].
Attuatore	Dispositivi che a partire da un segnale elettrico producono un effetto nello spazio fisico. Possono essere digitali (alto / basso o zero / uno) o analogici se ammettono valori discreti nell' intervallo [0, 255].
Device	Dispositivo generico del quale non se ne specifica la natura (attuatore / sensore) ed è contraddistindo da un nome simbolico ed il numero di pin (piedino sulla board).
Led	Attuatore che può essere digitale o analogico, sta all' utente finale decidere questa caratteristica. Questo dispositivo viene usato per i test (fisici).
Potenziometro	Sensore analogico usato per i test (fisici).

2.1.2 Modello del dominio

Dopo una attenta analisi dei requisiti richiesti per il progetto ho steso un primo modello delle entità (vedi figura 2.1) di cui è composto il dominio mettendo in rilievo le relazioni che intercorrono tra queste.

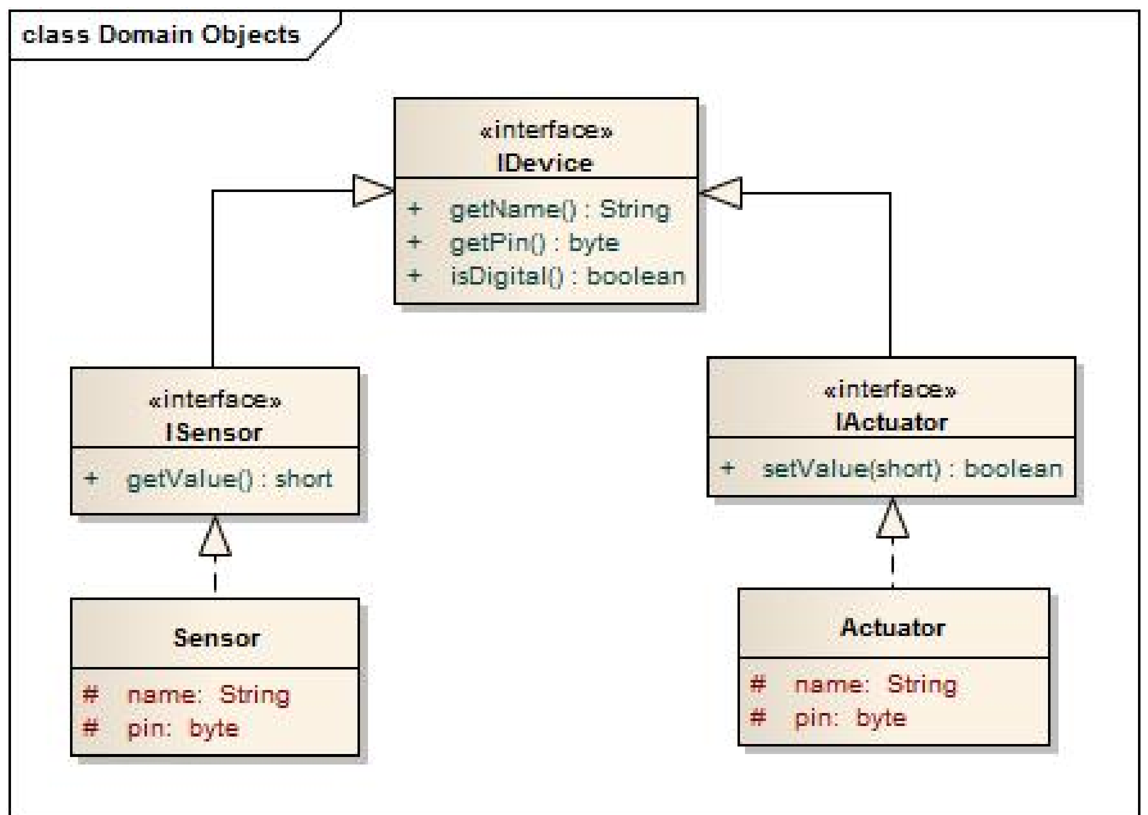

Figura 2.1: Modello del dominio.

2.1.3 Casi d' uso

Tramite l'analisi dei casi d' uso (vedi figura 2.2) si riesce a comprendere meglio quali sono le funzionalità importanti dal punto di vista dell' utente finale (Actor) e quali funzionalità interne non devono condizionargli l' esperienza d' uso.
Di seguito si riporta la descrizione tabellare degli use cases.

ID	UC1
Descrizione	L' utente, al primo utilizzo dell' applicazione e alla prima disposizione dei componenti sulla board Arduino deve effettuare la configurazione tramite questa funzionalità.
Attori	L' utente.

ID	UC2
Descrizione	Accesso ai dispositivi del sistema per potere leggere e scrivere valori di sensori ed attuatori.
Attori	L' utente.

ID	UC1a
Descrizione	Questa funzionalità permette all' utente di inserire i dispositivi (virtuali) che sono stati montati (fisicamente) sulla board Arduino. Dopo questo passaggio viene generato un file di configurazione che contiene le informazione sui dispositivi.
Attori	Il configuratore del sistema.

ID	UC1b
Descrizione	Dopo avere eseguito il caso d' uso UC1a è possibile leggere il file di configurazione, incamerare le informazioni e quindi popolare il Middleware (di dispositivi virtuali).
Attori	Il configuratore del sistema.

ID	UC2a
Descrizione	Si richiede al TransportLayer un' operazione di scrittura o un' operazione di lettura su un dispositivo.
Attori	Il Middleware.

ID	UC2b
Descrizione	Si richiede al Comunicator di potere inviare o ricevere una sequenza di bytes.
Attori	Il TransportLayer.

ID	UC2c
Descrizione	Si richiede ad Arduino lo svolgimento di una operazione.
Attori	Il Comunicator.

2.2 Analisi del problema

L'analisi del problema si basa completamente sui requisiti che questo progetto si è dato e serve per mettere in luce gli aspetti significativi dei quali tenere conto in fase di progetto.

2.2.1 Modello del dominio.

Si faccia riferimento a figura 2.1.

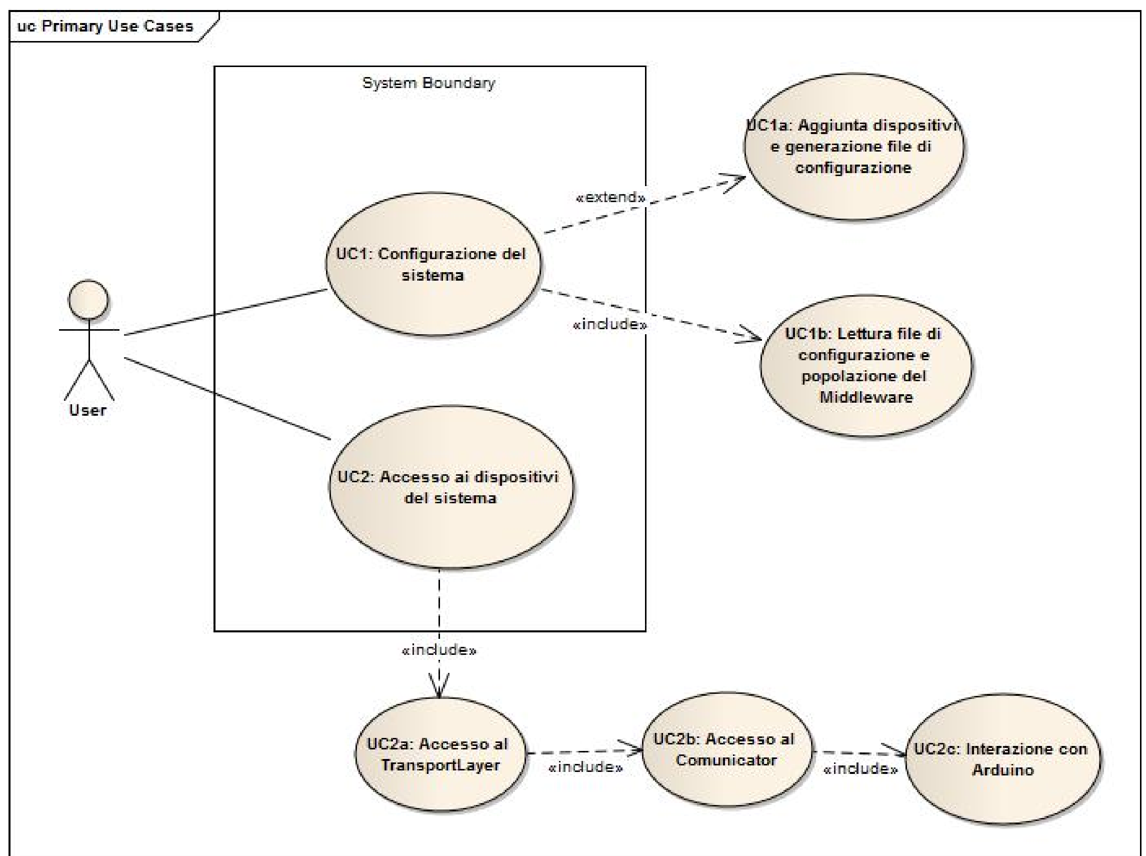

Figura 2.2: Casi d' uso

domain.IDevice	Questa interfaccia definisce caratteristiche comuni a tutti i dispositivi che vengono montati sulla board Arduino. In particolare questa richiede che il componente che la implementa possa restituire un nome simbolico, il numero del piedino su quale è montato, se è digitale o no.
domain.ISensor	Tutti i sensori misurano una grandezza dall'ambiente esterno e ne restituiscono il valore. Questa interfaccia chiede la restituzione di un valore *short*.
domain.IActuator	Tutti gli attuatori eseguono una operazione in base al tipo di valore che viene loro imposto. L' interfaccia si propone di settare un valore di tipo *byte* sul sensore.
domain.Sensor	Questa classe implementa le proprietà fondamentali di un sensore.
domain.Actuator	Questa classe implementa le proprietà fondamentali di un attuatore.

L' interfaccia doamin.ISensor nel metodo *getValue()* restituisce un valore short perchè il numero massimo che Arduino puó misurare su un sensore analogico è 1023.

L' interfaccia domain.IActuator nel metodo *setValue(int)* chiede come parametro un valore intero ma bisogna tenere presente che può assumere solamente valori nell' intervallo discreto [0, 255], si è scelto di utilizzare il parametro intero per non costringere tutte le volte l' utente ad inserire un cast a byte.

2.2.2 Architettura logica.

Per la realizzazione del progetto scelgo una architettura a strati (vedi figura 2.3) che mi garantisce i seguenti vantaggi:

- Separation concern: separazione tra servizi di alto livello e basso livello e tra servizi generali e quelli più specifici.
- La complessità relativa ai vari concern è incapsulata e può essere disaccoppiata.

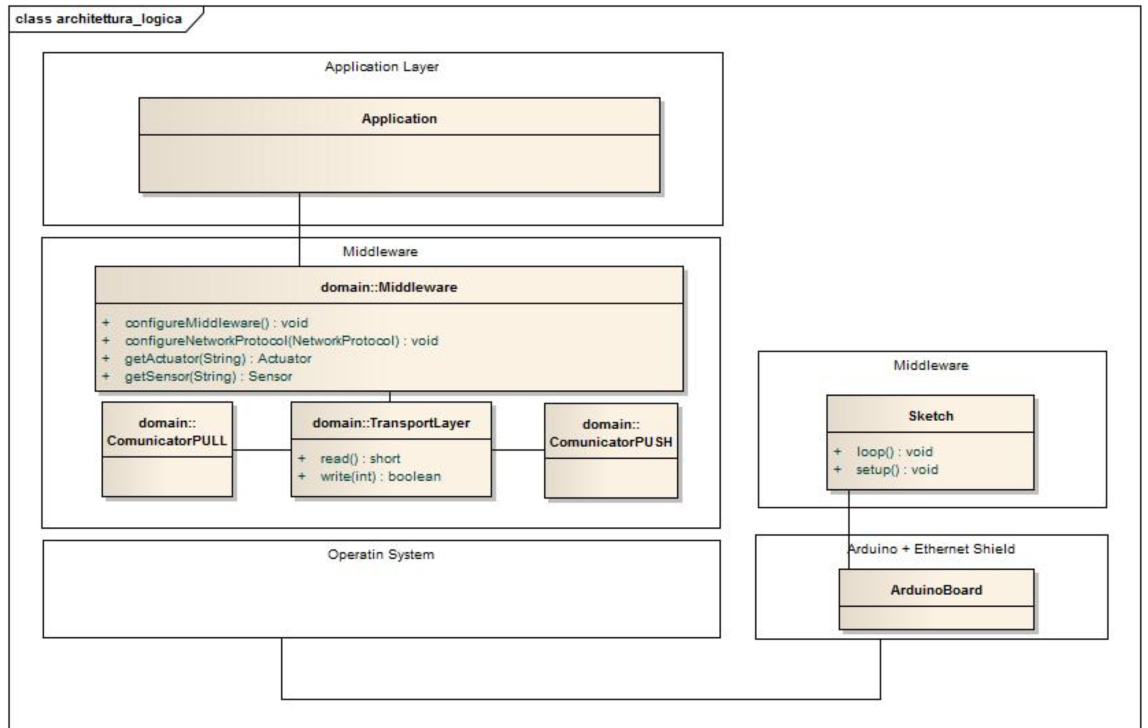

Figura 2.3: Architettura logica del sistema.

- Gli strati più “alti”possono essere sostituiti da nuove implementazioni.
- Gli strati più “bassi”contengono funzioni maggiormente riusabili.

Nel caso in esame definisco tre *layer*. In ordine crescente (dal più basso al più alto):

1. **Sistema operativo**: è la base per la costruzione dell’ applicazione.
2. **Middleware**: questo livello è composto da quattro sottolivelli:
   - un livello di presentazione che concede all’ utente la possibilità di accesso ai dispositivi virtuali.

   - un livello che incapsula l' insieme di tutti i dispositivi virtuali fisicamente montati su Arduino presentando all' utente i riferimenti a questi ultimi.
   - un livello che si occupa di trasformare le richieste del livello successivo in stringhe di bytes e viceversa.
   - un livello che invia stringhe di bytes alla board e viceversa.

3. **Application layer**: questo strato coincide con l' applicazione che l' utente finale, utilizzatore del middleware, realizza.

2.2.3 Modello di interazione

Il modello dell' interazione ci fa capire come deve funzionare il sistema, in modo particolare ci indica chi interagisce con chi e ci fa comprendere il flusso di esecuzione di certe parti del software.

Le figure 2.4 e 2.5 ci mostrano cosa deve fare l'utente per ottenere rispettivamente il riferimento ad un dispositivo virtuale e il valore di un sensore. Per assegnare un valore ad un attuatore si utilizza la stessa sequenza ma l'attore dovrà invocare il metodo *setValue(byte)*:boolean su un attuatore che dovrà invocare il metodo *write(byte)*.

2.3 Modalità di interazione elaboratore-board

2.3.1 PULL mode

La modalità PULL si verifica quando l' elaboratore effettua il polling sulla board Arduino. L' elaboratore deve quindi inviare ripetutamente delle richieste di lettura/scrittura. Il software caricato sulla board elabora la richiesta e spedisce indietro il risultato.

Questo tipo di modalità può essere molto costosa in termini di traffico di rete se la frequenza con la quale vengono richiesti i dati è molto elevata. Bisogna inoltre tenere conto che la frequenza di riferimento sia quella del microcontrollore al fine di evitare l' effetto collo di bottiglia.

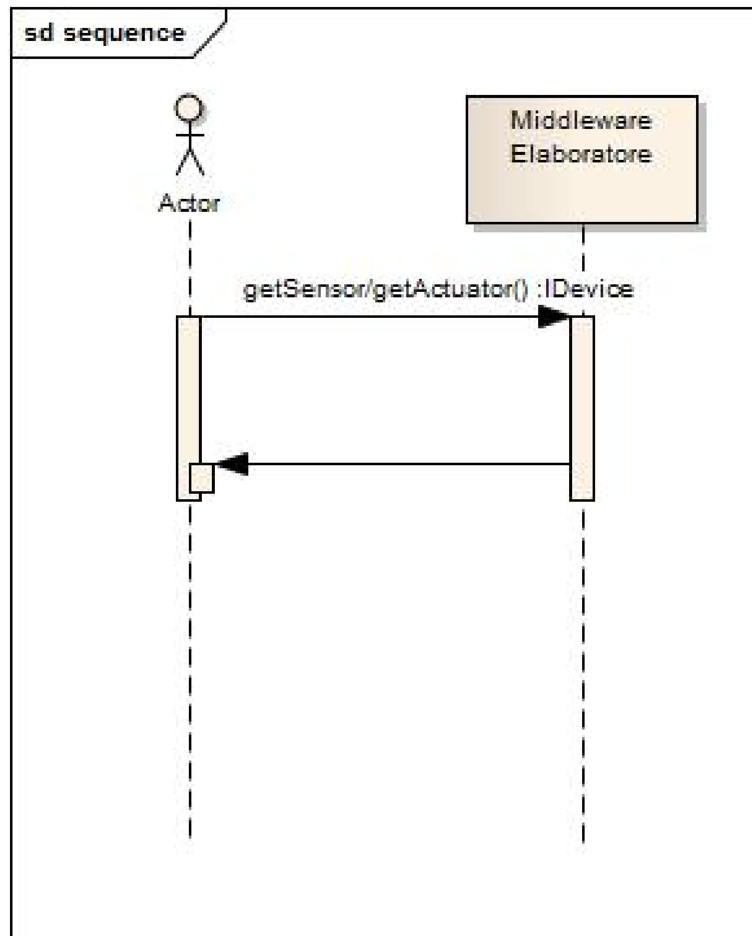

Figura 2.4: Sequenza di operazioni per ottenere il riferimento ad un dispositivo dal sistema virtuale.

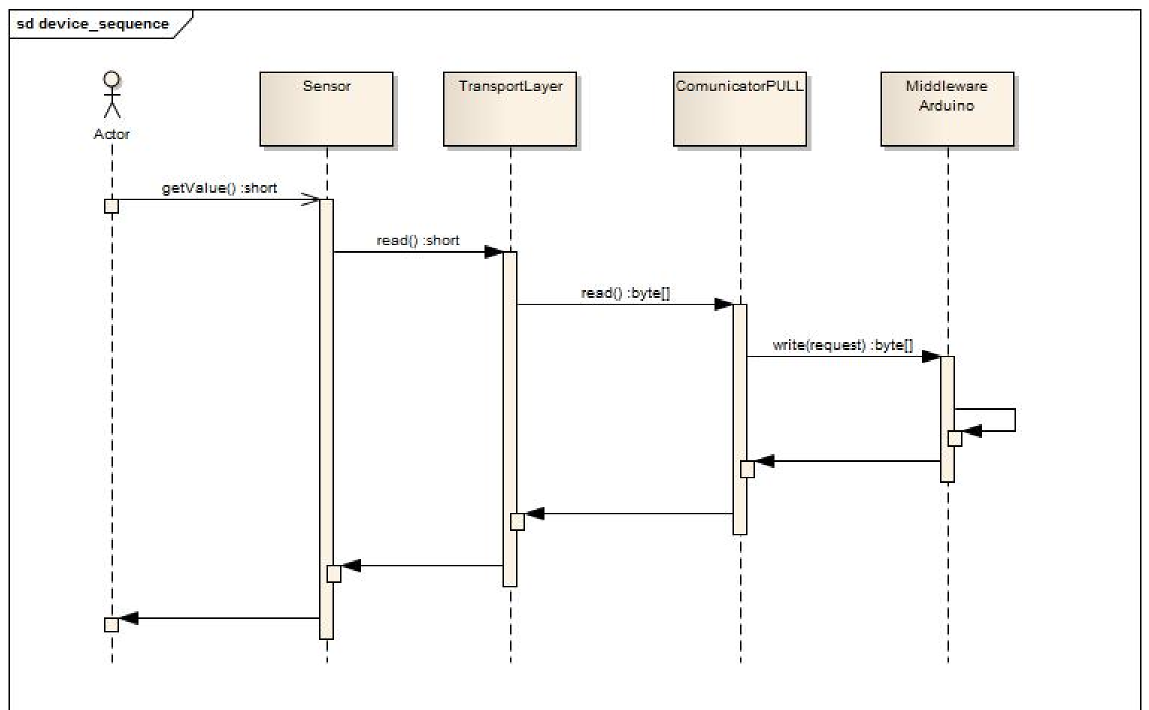

Figura 2.5: Sequenza per ottenere il valore di un sensore.

2.3.2 PUSH mode

La modalità PUSH si ha quando il microcontrollore, al verificarsi di un evento esplicitato nel software caricato (ad esempio il superamento, da parte di un sensore, di un fissato valore), invia all' elaboratore una notifica del cambiamento di stato di un componente montato sulla board.

Questa modalità risulta vantaggiosa, ad esempio, quando all' elaboratore interessa solamente l' intercettazione di un valore di picco di un sensore: il vantaggio sta nel non dovere fare il polling.

2.4 Progetto

Lo scopo della fase di progetto è quello di affinare l' architettura logica discussa nell' analisi del problema in modo da definire un insieme di parti e le relazioni che intercorrono tra di esse tenendo conto della tecnologia che verrà utilizzata nella fase di implementazione. La figura 2.6 mostra, tramite diagramma UML, la struttura scelta per la fase di progetto.

2.4.1 Progetto di VirtualBoard

Come accennato nei requisiti le proprietà fondamentali di questo componente sono la configurazione del sistema e la possibilità di accedere ai dispositivi virtuali. La fase di configurazione si articola nelle seguenti sottofasi:

- Aggiunta dei dispositivi virtuali, dove viene specificato il tipo di device, il nome simbolico del device, il numero del pin del device sulla board, ed un valore booleano che specifica se il device è digitale o no.

- Generazione di un file di configurazione dove vengono memorizzate tutte le informazioni richieste nel punto precedente. La sintassi di tale file è la seguente:

  ```
  Tipo#Nome#Pin#isDigital
  ```

  dove il cancelletto è un carattere di separazione.

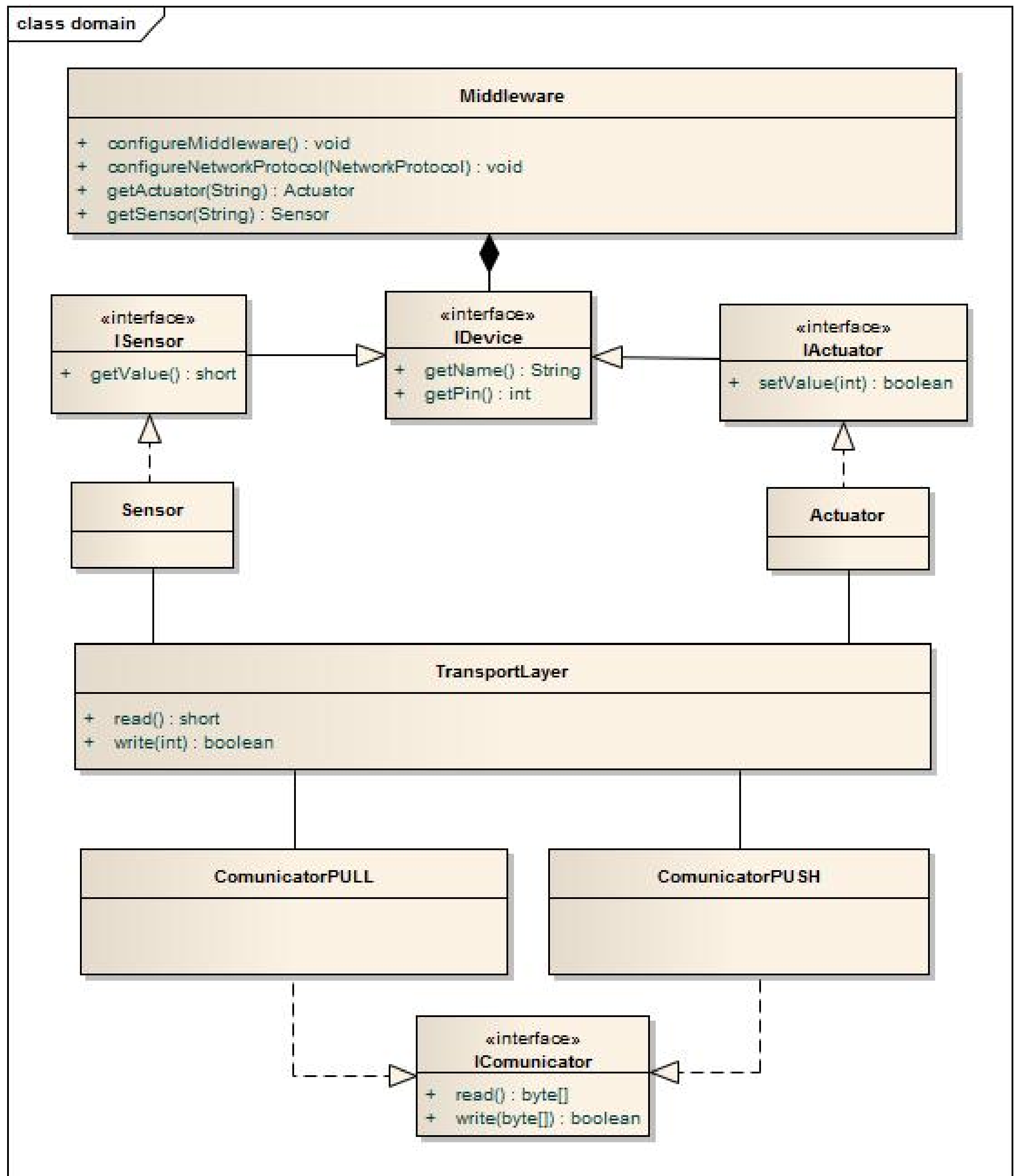

Figura 2.6: Progetto del sistema.

- Popolazione di dispositivi virtuali nella VirtualBoard sulla base della lettura del file di configurazione. Questa operazione è molto importante perchè consente all' utente di prelevare un sensore od un attuatore in base al suo nome simbolico.

- Configurazione del Middleware, le informazioni principali richieste sono: il protocollo di rete da utilizzare, l' indirizzo ip dell' Ethernet Shield Arduino, la porta del servizio offerto dal software caricato sul microcontrollore, la porta di comunicazione lato elaboratore. Si rammenta che il Middleware può operare in PULL mode ed in PUSH mode, in questo ultimo caso è necessario specificare la porta di comunicazione dell' elaboratore. Il progetto di queste due modalità viene descritto in seguito.

La fase di accesso alle risorse della VirtualBoard avviene tramite due metodi get: uno per i sensori ed uno per gli attuatori. Quest' ultimo fatto mi indirizza a creare una lista per i sensori ed una lista per gli attuatori. Di seguito riporto la prima stesura di codice sorgente atto alla realizzazione della suddetta funzionalità:

```
private List<Sensor> sensors_list = new ArrayList<Sensor>();
private List<Actuator> actuators_list = new ArrayList<Actuator>();

public Sensor getSensor(String device_name) {
  for(Sensor s : this.sensors_list) {
    if(s.getName().equals(device_name)) {
      return s;
    }
  }
  return null;
}

public Actuator getActuator(String device_name) {
  for(Actuator a : this.actuators_list) {
    if(a.getName().equals(device_name)) {
      return a;
    }
  }
  return null;
}
```

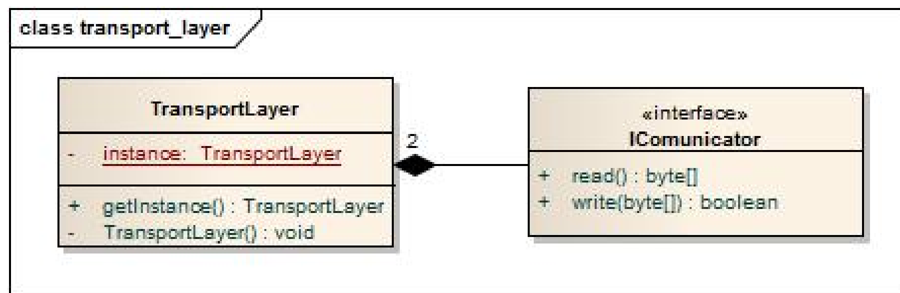

Figura 2.7: Modello di un ComunicationLayer Singleton composto da due Comunicator: uno PUSH e uno PULL.

2.4.2 Progetto del Middleware lato elaboratore

Nell'architettura logica prima esposta si nota che il TransportLayer è uno solo condiviso per tutti gli IDevice incapsulati dal Middleware e questo suggerisce l' utilizzo del pattern Singleton.

Singleton

Il Singleton è un design pattern strutturale che impone ad una determinata classe di essere istanziata una ed una sola volta e di fornire un punto di accesso globale alla stessa. Vi sono diverse implementazioni, una di queste è quella ad inizializzazione preventiva che dichiara una istanza statica all' atto del caricamento in memoria della classe, un costruttore privato ed un metodo *get* che restituisce il riferimento all' istanza.

```
public class SingletonPattern {
  private final static SingletonPattern instance =
      new SingletonPattern();

  private SingletonPattern() {}

  public static SingletonPattern getInstance() {
    return instance;
  }
}
```

Riprendendo il discorso della PUSH mode sento la necessità di introdurre in concetto di evento relativo alla variazione del valore di un sensore. Il com-

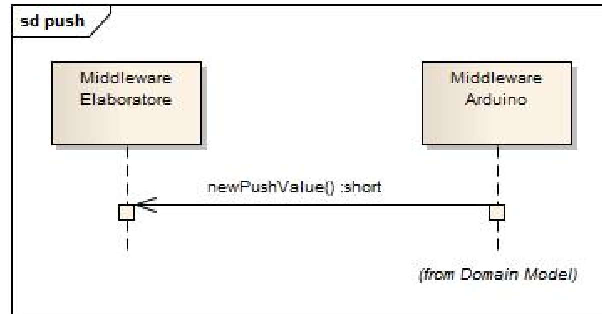

Figura 2.8: Sequenza di una notifica PUSH.

pito del TransportLayer è di dare un senso alle stringhe di byte che arrivano al livello di comunicazione, e nel caso delle notifiche push questo componente dovrà essere ascoltatore di quello che succede nel livello sottostante. La parola *ascoltatore* suggerisce un altro noto pattern: EventListener, che definisce eventi di tipo eterogeneo relativi ad una sorgente.

Event-Listener

Si articola in 3 componenti:

- La classe Event che modella l'evento da intercettare.
- L' interfaccia EventSource fornisce i metodi per aggiungere e rimuovere ascoltatori interessati ad un determinato evento. Sarà la classe che implementa questa interfaccia a scatenare l'evento quando necessario.
- L' interfaccia EventListener chiede l' implementazione di un metodo che viene richiamato ogni qual volta si verivica l'evento Event.

Diversi scenari applicativi possono portare alla coesistenza della PUSH mode con la PULL mode e questo implica che vi siano due "canali"di comunicazione: il primo che esegue richieste PULL ed il secondo sempre in ascolto di notifiche PUSH. Questa coesistenza non sarebbe possibile in quanto la funzione di lettura di un canale di comunicazione è funzione bloccante e bloccherebbe quindi il flusso del programma. Il linguaggio Java mette a

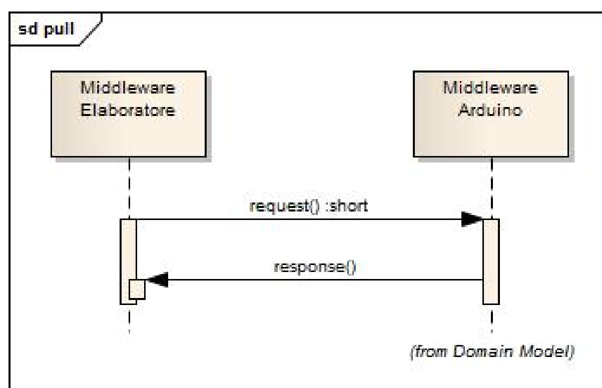

Figura 2.9: Sequenza di un' operazione PULL.

disposizione la classe Thread che permette di eseguire operazioni parallelamente ad un altro flusso di esecuzione. Il TransportLayer incapsulerà quindi un Comunicator per le richieste PULL ed un Comunicator PUSH, che estende la classe Thread, sempre in ascolto su una porta (senza bloccare il flusso di esecuzione).

2.5 Progetto del Comunicator

Il Comunicator è il componente che interagisce direttamente con l' Ethernet Shield tramite uno scambio di bytes e per questo la sua interfaccia offre all' utente due metodi: uno di scrittura ed uno di lettura. Per requisito lo scambio di bytes deve avvenire mediante i protocolli UDP e TCP. Definisco, per la PULL mode, due classi che implementano l' interfaccia IComunicator e all' interno delle quali verranno definite le socket per l' opportuno protocollo. Le classi che verranno utilizzate per la PUSH mode, oltre ad implementare IComunicator, estenderanno la classe Thread col fine di agire parallelamente alla PULL mode (vedi figura 2.10).

Abbiamo prima detto che il TransportLayer osserva il livello sottostante (cioè il Comunicator PUSH) per carpire l'evento che si scatena quando arriva una nuova notifica PUSH al socket attivo. Per rendere possibile il tutto è necessario che anche il Comunicator implementi le funzionalità del pattern Event-Listener (vedi figura 2.11).

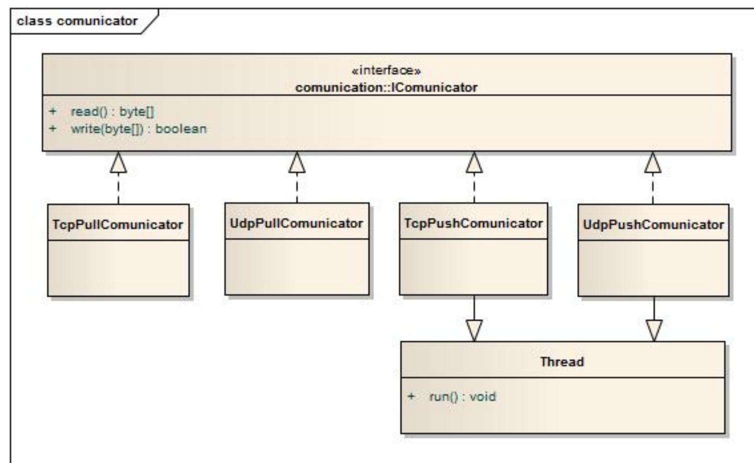

Figura 2.10: Struttura dei comunicator.

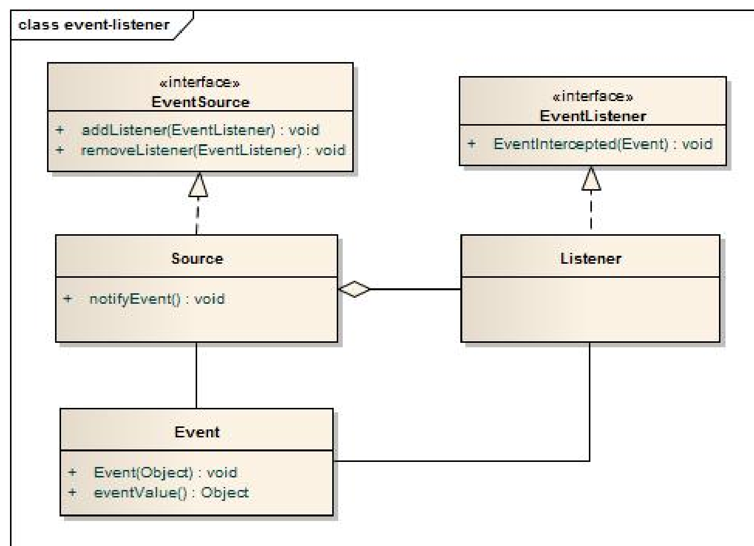

Figura 2.11: Struttura del pattern Event-Listener.

2.6 Progetto del protocollo elaboratore-board

Per designare questo protocollo ho ritenuto necessario includere le seguenti informazioni:

1. Numero del PIN

2. Tipo di operazione: lettura/scrittura

3. Tipo di dispositivo: analogico/digitale

4. Valore da assegnare in caso di operazione di scrittura.

Tutte queste informazioni possono essere condensate in 2 bytes (16 bits), partendo dal bit più significativo:

- 4 bits per il pin: il pin con valore più alto è il pin 13.

- 1 bit per il tipo di operazione: 0 per lettura, 1 per scrittura.

- 1 bit per il tipo di dispositivo: 0 analogico, 1 digitale.

- 10 bits per il valore: in realtà, per la scrittura analogica, ne servono solo 8 perchè il valore massimo assegnabile è 255; ne servono invece 10 quando viene restituito il valore di lettura analogica che al massimo può assumere il valore 1023.

Per la modalità PUSH è necessario sfruttare una determinata configurazione: pin + read + analog/digital più 8 zeri e gli ultimi due bits che fanno da selettore; se i bit selettori valgono 01 si richiede la notifica per il dispositivo sul pin specificato, con 10 se ne richiede la rimozione.

Per il protocollo UDP, non essendovi una connessione, Arduino non può accorgersi che l' elaboratore si è disconnesso ed è per questo che deve essere definita una sequenza di reset per consentire al microcontrollore di accettare nuove richieste con nuovi parametri, questa configurazione prevede che tutti i bits del pin siano ad 1.

Esempi

Voglio richiedere, in modo PULL, al microcontrollore il valore del dispositivo analogico montato sul pin 2.

```
00100000 00000000
```

Ammettendo che il sensore restituisca valore massimo, Arduino risponde:

```
00000011 11111111
```

Voglio settare il valore 255 sul pin 3:

```
00111000 11111111
```

Richiesta di notifiche PUSH per il pin 9:

```
10010000 00000001
```

Richiesta di rimozione delle notifiche PUSH sul pin 9:

```
10010000 00000010
```

2.7 Progetto del middleware lato Arduino

Voglio realizzare un software che garantisca all' utente finale che utilizza il Middleware di non dovere mettere mano al codice Wiring sul microcontrollore. Per fare questo devo analizzare la casistica di tutto ciò che è consentito fare dall' elaboratore:

- Ricordando che i protocolli di interazione sono basati su TCP ed UDP bisogna utilizzare sia un server UDP che un server TCP. Vi sono due modi per fare ciò: realizzare due sketch[1] differenti o creare un selettore in uno sketch unico ed includerli tutte e due nel medesimo.
- Bisogna tenere nota del protocollo di comunicazione tra le due piattaforme e quindi nello scketch devono essere definite delle maschere per le operazioni bit a bit col fine di potere estrapolare correttamente le informazioni (pin, lettura/scrittura, analogico/digitale, valore). Queste informazioni devono essere prelevate tramite opportune procedure in modo da poter creare una libreria Arduino e riutilizzarla per eventuali estensioni dell' applicazione.

[1]Lo sketch è il programma che gira sul microcontrollore.

- Lo sketch deve garantire sia la PULL mode che la PUSH mode all' elaboratore. La PULL mode è di semplice realizzazione in quanto è solamente un polling da parte del computer. Per la PUSH mode creo una struttura (struct) dove inserisco le proprietà indispensabili per i dispositivi che richiedono questa modalità: un flag che mi dice se un determinato pin è registrato per ottenere le notifiche, il valore precedente.

- Procedure di registrazione e rimozione dei dispositivi che richiedono notifiche PUSH col fine di eliminare controlli inutili al microcontrollore (non ha senso scorrere la lista dei dispositivi se nessuno di questi è interessato alle notifiche PUSH).

Gli elementi sopra citati sono alla base del capitolo successivo dove illustrerò la realizzazione del middleware lato Arduino.

Capitolo 3

Middleware lato Arduino

3.1 Realizzazione server UDP/TCP

3.1.1 Server UDP

Vengono dichiarate le variabili globali:

```
// MAC Address of Arduino Ethernet Shield
byte mac[] = {
  0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
// IP Address of Arduino Ethernet Shield
IPAddress ip(192, 168, 0, 13);
// Listening UDP Port
unsigned int localPort = 8888;
// Listening UDP Port of the elaborator's socket
unsigned int pushPort = 8889;
IPAddress pushIp;
// Buffer where are contained incoming data
char packetBuffer[UDP_TX_PACKET_MAX_SIZE];
```

All' interno del metodo setup() vengono inizializzati l' Ethernet Shield con indirizzo mac e ip dichiarati come variabili globali e viene creato il socket UDP in ascolto sulla porta relativa:

```
Ethernet.begin(mac, ip);
Udp.begin(localPort);
```

Nel metodo loop() è localizzato il core del nostro server:

```
int packetSize = Udp.parsePacket();
if(packetSize) {
  Udp.read(packetBuffer, UDP_TX_PACKET_MAX_SIZE);
  . . .
  sendPacket(getReadValue(return_value), Udp.remoteIP(), Udp.remotePort());
}

void sendPacket(char *buffer, IPAddress target_ip, int target_port) {
  Udp.beginPacket(target_ip, target_port);
  Udp.write(buffer);
  Udp.endPacket();
}
```

La funzione getReadValue è una funzione di libreria che prende in ingresso un intero e ritorna un array di due char che può essere spedito tramite la funzione Udp.write(char *buffer).

3.1.2 Server TCP

Vengono dichiarate le seguenti variabili globali, col server Tcp in ascolto sulla porta 9000:

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192,168,0, 13);

EthernetServer server(9000);

char buffer[2];
```

Nel metodo setup() viene inizializzato il server TCP:

```
// start the Ethernet connection and the server:
Ethernet.begin(mac, ip);
server.begin();
```

Nel metodo loop() è presente il codice per gestire le connessioni:

```
EthernetClient client = server.available();
  if (client) {
    int buffer_count = 0;
    while (client.connected()) {
```

```
      if (client.available()) {
        buffer[buffer_count++] = client.read();
        . . .
      }
    }
  }
  client.stop();
```

La funzione server.available() controlla se sono presenti dei dati, se tale condizione è vera il microcontrollore ne inizia la lettura. Si ricorda che le richieste sono di due bytes e che il ciclo viene ripetuto fino a che il client resta connesso al server.

3.2 Gestione del protocollo

Per prima cosa vengono definite le maschere che serviranno ad estrapolare i dati incapsulati nei 2 byte della richiesta:

```
#define WRITE_OP 1
#define READ_OP  0

#define ANALOG_OP  0
#define DIGITAL_OP 1

#define ANALOG_NUM  6
#define DIGITAL_NUM 14

#define PUSH_ON  1
#define PUSH_OFF 0

// This masks refers to most significative part of the word (16bits)
#define PIN_MASK 0xF0
#define RW_MASK  0x08
#define AD_MASK  0x04
```

In seguito vengono definite le funzioni di libreria che restituiscono i dati. La funzione getPin(char *buffer) restituisce il pin del dispositivo della richiesta:

```
int getPin(char *buffer) {
  return  ((buffer[1] & PIN_MASK) >> 4) & 0x000F;
}
```

La funzione isWriteOperation(char *buffer) ci dice se la richiesta è di scittura o lettura:

```
int isWriteOperation(char* buffer) {
    return ((buffer[1] & RW_MASK) >> 3) & 0x000F;
}
```

La funzione isDigitalOperation(char *buffer) ci dice se la richiesta vale per un dispositivo analogico o digitale:

```
int isDigitalOperation(char *buffer) {
   return ((buffer[1] & AD_MASK) >> 2) & 0x000F;
}
```

La funzione getOperationValue(char *buffer) restituisce il valore da assegnare al pin specificato:

```
int getOperationValue(char *buffer) {
  return buffer[0] & 0x00FF;
}
```

Queste funzioni vengono usate sia dal server UDP che dal server TCP.

3.3 Gestione notifiche PUSH

Tengo a precisare il fatto che il linguaggio Wiring permette di leggere/scrivere su pin sia analogici che digitali. Fatta questa premessa ricordo che ho definito la seguente struttura per la gestione di questa modalità:

```
typedef struct push_device {
  int isPush;
  int prev_value;
} push_array;
```

Per il motivo precedentemente citato definisco 2 array: uno per i dispositivi analogici ed uno per i dispositivi digitali:

```
push_array push_analog[ANALOG_NUM];
push_array push_digital[DIGITAL_NUM];
```

Per come è stato progettato il protocollo ho bisogno di un blocco di codice che, prima di compiere qualsiasi operazione, deve controllare se la richesta arrivata sulla porta locale è di registrazione o cancellazione di notifiche push. Questo blocco risolve la stringa di byte e setta l' opportuna variabile della struttura sia per la registrazione/cancellazione sia per il valore precedente[1]:

```
// verify PUSH registration
if(rw == READ_OP && val == 1) {
  if(ad == ANALOG_OP) {
    push_analog[pin].isPush = 1;
    // initialization
    push_analog[pin].prev_value = 0;
    Serial.println("Richiesta push intercettata!");
  } else {
    push_digital[pin].isPush = 1;
    push_digital[pin].prev_value = 0;
  }
  push_device++;
  pushIp = Udp.remoteIP();
}

// verify PUSH deregistration
if(rw == READ_OP && val == 2) {
  if(ad == ANALOG_OP) {
    push_analog[pin].isPush = 0;
  } else {
    push_digital[pin].isPush = 0;
  }
  push_device--;
  // pass next cycle's step
  return;
}
```

Per quanto riguarda il processo di elaborazione dei dati da mandare al computer ho un blocco sia per i dispositivi analogici sia per quelli digitali dove scorro gli array dei dispositivi push e, quando il flag è settato ed il valore precedente è diverso da quello attuale, invio la notifica:

[1]Ho bisogno di memorizzare il valore precedente perchè le notifiche giungono all' elaboratore solo se vi è stata una variazione.

```
// PUSH
if(push_device != 0) {
  // analog
  for(int i = 0; i < ANALOG_NUM; i++){
     int read_value = analogRead(i);
     if(push_analog[i].isPush == 1 &&
          read_value != push_analog[i].prev_value) {
       char val[2];
       val[0] = *((char*)&read_value);
       val[1] = *(((char*)&read_value) + 1);
       val[1] |= (i << 4);
       sendPacket(val, pushIp, pushPort);
       push_analog[i].prev_value = read_value;
     }
  }

  //digital
  for(int i = 0; i < DIGITAL_NUM; i++) {
    int read_value = digitalRead(i);
    if(push_digital[i].isPush == 1 &&
        read_value != push_digital[i].prev_value) {
      char val[2];
      if(read_value == HIGH) {
        val[0] = 1;
      } else {
        val[0] = 0;
      }
      val[1] = (i << 4) | (DIGITAL_OP << 2);
      sendPacket(val, pushIp, pushPort);
      push_digital[i].prev_value = read_value;
    }
  }
```

Il primo blocco if serve a saltare computazioni inutili quando non è presente nessuno dispositivo che richiede notifiche push.

Capitolo 4

Uso del middleware e collaudo

In questo capitolo vengono visualizzati gli esempi applicativi ed i metodi coi quali sono stati condotti i test del middleware.

I test sono stati svolti con un esempio di riferimento: il Threshold[1]. Per realizzare il suddetto esempio abbiamo bisogno di due dispositivi elettronici:

- 1 Potenziometro che determina il valore di input.
- 1 LED che deve accendersi al superamento del valore di input dato dal potenziometro.

I componenti sopra elencati vanno disposti sulla piattaforma come mostrato in figura 4.1.

4.1 Configurazione del sistema

La prima operazione che l' utente deve compiere per potere utilizzare il middleware è la configurazione. L' oggetto che ci permette questa operazione è la classe Middleware, creiamo i nostri dispositivi virtuali tramite le seguenti operazioni:

```
system.Middleware system = new Middleware();
system.addDevice("actuator", "l0", 9, true);
system.addDevice("sensor", "p0", 2, false);
```

[1]Constiste nell' individuazione di un valore di soglia.

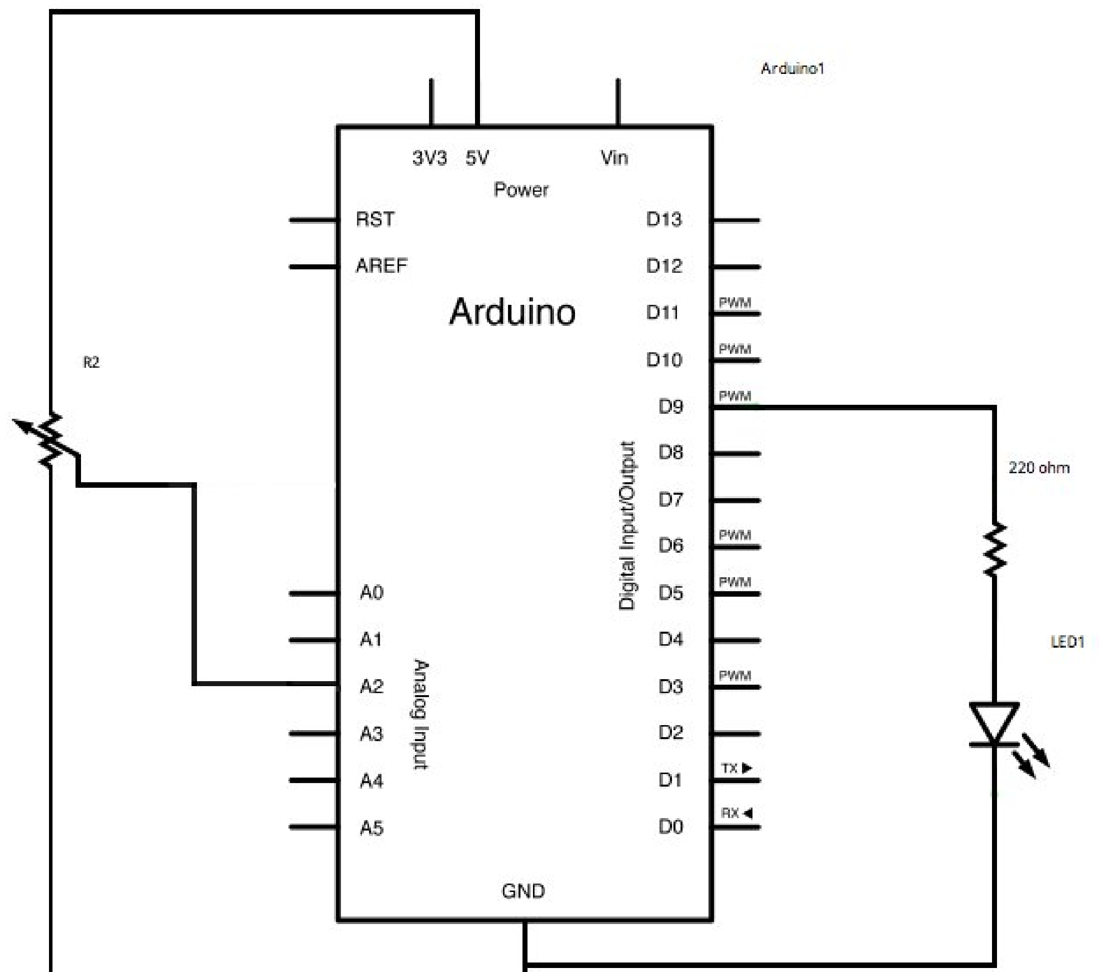

Figura 4.1: Schema elettrico dell'esempio di riferimento.

Il primo parametro del metodo addDevice ci dice se il dispositivo è un sensore o un attuatore, il secondo parametro è il numero del pin, il terzo indica se il dispositivo è digitale o no. Il passo successivo alla creazione dei dispositivi è la generazione del file di configurazione:

```
system.generateConfigurationFile();
```

A questa operazione segue la configurazione del Middleware:

```
system.configureMiddleware();
```

Tramite questa operazione vengono popolate le liste di senori ed attuatori in base al contenuto del file di configurazione. L' ultimo passo è la configurazione del middleware che richiede in input il protocollo di rete da utilizzare:

```
system.configureNetworkProtocol(NetworkProtocol.UDP);
```

Ricordo che l'aggiunta dei dispositivi e la generazione del file di configurazione non è sempre necessaria, ad esempio se utilizziamo sempre la stessa configurazione di Arduino non dobbiamo riconfigurare tutto: basta solamente configurare la VirtualBoard ed il middleware di comunicazione:

```
system.Middleware system = new Middleware();
system.configureMiddleware();
system.configureNetworkProtocol(NetworkProtocol.UDP);
```

Inoltre caricando sulla board il codice descritto nel capitolo precedente è possibile programmare Arduino direttamente con un linguaggio di alto livello come il Java. Per fare ciò è necessario ottenere il riferimento al VirtualDevice. Nel nostro esempio se voglio accendere il led sul piedino 9 mi basta utilizzare la funzione getActuator(name:String) del Middleware:

```
(system.getActuator("l0")).setValue(1);
```

dove "l0"è il nome simbolico del led.

4.2 Threshold in modo PULL

In questo esempio l'elaboratore non fa altro che richiede continuamente alla board il valore del potenziometro, se quest' ultimo è maggiore della soglia il sistema accende il led, altrimenti lo spegne. La classe che implementa questo esempio è una classe attiva ed estende la classe Thread perchè voglio potere eseguire altre operazioni parallelamente. Il costruttore richiede un riferimento al Middleware e la soglia, successivamente prende i riferimenti del led, del potenziometro e setta il threshold. Di seguito si riporta il codice:

```
package example;

import domain.Actuator;
import domain.Sensor;

import system.Middleware;

public class ThresholdPull extends Thread{

private Actuator led;
private Sensor potentiometer;
private int threshold;
private boolean stop;
private boolean is_led_on;

public ThresholdPull(Middleware system, int threshold) {
    this.threshold = threshold;

    this.led = system.getActuator("l0");
    this.potentiometer = system.getSensor("p0");

    this.is_led_on = false;
    this.stop = false;
  }

  @Override
  public void run() {
    while(!this.stop) {
      int value = potentiometer.getValue();
```

```
      if(value > this.threshold) {
          if(!this.is_led_on) {
            this.led.setValue(1);
            this.is_led_on = true;
        }
      } else {
          if(this.is_led_on) {
            this.led.setValue(0);
            this.is_led_on = false;
        }
      }

      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    return;
  }

  public void stopExample() {
    this.stop = true;
  }
}
```

4.3 Threshold in modo PUSH

In questo esempio l'elaboratore resta in attesa dei cambiamenti del valore del potenziometro. Ricordando che la classe dell' esempio implementa SensorListener e può quindi osservare i cambiamenti del valore del potenziometro, quando viene scatenato l'evento l' apposito metodo esegue le operazioni descritte nella sezione precedente. Anche in questo caso il costruttore della classe d' esempio richiede un riferimento al Middleware e la soglia; il Middleware è necessario per prendere il riferimento ai VirtualDevices, questa volta però dobbiamo abilitare le notifiche PUSH sul potenziometro. Di seguito si riporta il codice della classe di esempio:

```
package example;

import system.Middleware;
import domain.Actuator;
import domain.Sensor;
import domain.SensorEvent;
import domain.SensorListener;

public class ThresholdPush extends Thread implements SensorListener {

private Actuator led;
private Sensor potentiometer;
private int threshold;
private BoundedBuffer buffer;
private boolean is_led_on;

  public ThresholdPush(Middleware system, int threshold) {
    this.threshold = threshold;

    this.led = system.getActuator("l0");
    this.potentiometer = system.getSensor("p0");
    this.potentiometer.enablePushNotification();
    this.potentiometer.addSensorListener(this);

    this.is_led_on = false;
    this.buffer = new BoundedBuffer();
  }

  @Override
  public void run() {
      while(true) {
          try {
                int value = (Short)this.buffer.get();
                if(value >= this.threshold) {
                    if(!this.is_led_on) {
                        this.led.setValue(1);
                        this.is_led_on = true;
                    }
                } else {
```

```
                    if(this.is_led_on) {
                        this.led.setValue(0);
                        this.is_led_on = false;
                    }
                }
            } catch(Exception e) {
                    e.printStackTrace();
            }
        }
    }

    @Override
    public void valueChanged(SensorEvent e) {
        try {
          this.buffer.put(e.valueChanged());
      } catch(Exception e) {
          e.printStackTrace();
      }
    }
}
```

In questo esempio il metodo run() resta in attesa che il BoundedBuffer venga riempito dei valori inviati dal middleware lato Arduino. In questo esempio viene risparmiato il polling e si ha un minor costo computazionale.

4.4 Applicazione di test

Una semplice applicazione di testing a console che utilizza gli esempi descritti nelle sezioni precedenti è riportata nel codice sottostante:

```
package application;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import comunication.NetworkProtocol;

import example.ThresholdPull;
```

```
import example.ThresholdPush;
import system.Middleware;

public class Application {

/**
 * @param args
 */
public static void main(String[] args) {
  system.Middleware system = new Middleware();

  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  try {
    System.out.println("Welcome in Arduino Java Middleware.");
    System.out.println("Do you want to configure the Middleware ? [y/n]");

    String line = "";

    if(in.readLine().toLowerCase().equals("y")) {
      System.out.println("Add device with syntax: (actuator|sensor)
          NAME PIN IS-DIGITAL(True|False),
           type \"stop\" for ending configuration.");
  while(!(line = in.readLine()).toLowerCase().equals("stop")) {
    String[] split = line.split(" ");
    if(split[0].toLowerCase().equals("sensor"))
      system.addDevice("sensor", split[1], Integer.parseInt(split[2]),
             (split[3].equals("true")?(true):(false)));
    else if(split[0].toLowerCase().equals("actuator"))
      system.addDevice("actuator", split[1], Integer.parseInt(split[2]),
             (split[3].equals("true")?(true):(false)));
    else {
      System.out.println("Error: bad type.");
    }
  }

  system.generateConfigurationFile();
  system.configureMiddleware();
}
```

```
system.configureMiddleware();

System.out.println("Choose one example:");
System.out.println("1 - UDP Threshold PULL.");
System.out.println("2 - UDP Threshold PUSH.");
System.out.println("3 - TCP Threshold PULL.");

line = in.readLine();
if(line.equals("1") || line.equals("2")) {
  system.configureNetworkProtocol(NetworkProtocol.UDP);
  System.out.print("Enter threshold: ");
  if(line.equals("1")) {
    ThresholdPull example = new ThresholdPull(system,
        Integer.parseInt(in.readLine()));
    example.start();
  } else if(line.equals("2")) {
    ThresholdPush example = new ThresholdPush(system,
        Integer.parseInt(in.readLine()));
    example.start();
  }
} else if(line.equals("3")) {
  system.configureNetworkProtocol(NetworkProtocol.TCP);
  System.out.print("Enter threshold: ");
  ThresholdPush example = new ThresholdPush(system,
      Integer.parseInt(in.readLine()));
  example.start();
}

  System.out.println("Example started, type \"quit\" for exit.");

  while(!line.equals("quit")) {
    line = in.readLine();
  }
} catch (IOException e) {
    e.printStackTrace();
}

    System.exit(0);
}
```

```
}
```

4.5 Perfomance

Una analisi molto interessante è l'analisi delle performance che mette in rilievo le latenze introdotte dal middleware progettato in questo studio. Ho effettuato le rilevazione del RTT[2] per una richiesta PULL di lettura da un sensore, la figura 4.2 riporta l' analisi. Si può notare dal grafico che il ritardo introdotto dal middleware è inversamente proporzionale al periodo con cui le richieste vengono effettuate. Questo fenomeno è la diretta conseguenza dell' effetto collo di bottiglia. Dal grafico si nota una significativa riduzione della latenza quando le richieste vengono mandate con un periodo di ripetizione pari a 300 ms.

In molte applicazioni che non richiedono un aggiornamento molto frequente dei valori dei sensori questo framework potrebbe essere un valido strumento per programmare velocemente il microcontrollore, tenendo presente che, anche per applicazioni richiedenti un refresh elevato, una latenza di 90 ms non è poi così intollerabile per ottenere il valore dei sensori ogni millisecondo.

[2]Round-Trip-Time: tempo di andata e ritorno.

	Rilevazione										
Periodo (ms)	1	2	3	4	5	6	7	8	9	10	Media (ms)
1	34	103	98	104	103	101	89	101	100	72	90,5
5	43	97	97	97	98	97	97	97	97	97	91,7
10	28	92	91	92	92	92	92	93	92	92	85,6
20	59	81	82	82	82	82	82	82	82	82	79,6
50	71	71	49	55	47	59	43	50	46	46	53,7
100	20	2	102	102	2	2	1	2	102	102	43,7
150	14	51	52	53	52	52	52	52	52	16	44,6
250	11	52	51	52	53	52	52	52	52	53	48
300	97	3	101	2	3	3	3	3	3	2	22
500	28	2	3	3	2	2	3	3	2	2	5
1000	50	4	4	4	3	3	4	3	4	4	8,3

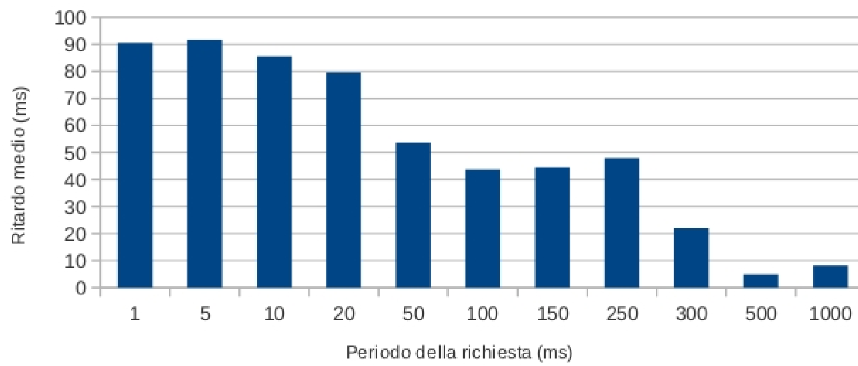

Figura 4.2: Rilevazione del RTT con diversi periodi di ripetizione.

Capitolo 5

Integrazione di tecnologie ad agenti

Il capitolo si propone di introdurre il concetto di agenti e di vedere come si integrano col middleware progettato e implementato nei capitoli precedenti.

5.1 Introduzione agli agenti

Per "agente"si intende un componente software reattivo che esibisce diversi gradi di autonomia per svolgere dei compiti, delegati o dall' utente o da un altro agente, che vengono eseguiti da questo nel migliore dei modi in base alle sue *conoscenze* [3]. Gli agenti, in quanto componenti autonomi e reattivi, possono essere chiamati a formare un sistema multi-agente (MAS) che è un sistema software delimitato da un ambiente (environment) ed ogni agente si occupa di eseguire operazioni su porzioni (artifact) di questo, ed eventualmente gli agenti possono scambiarsi informazioni (tramite veri e propri messaggi) per raggiungere un macro-obiettivo.

Un esempio significativo di Agents and Artifacts (AeA) [2] è la metafora del fornaio. Nel negozio del fornaio (environment) ci sono diversi lavoratori (agenti) ognuno dei quali ha delle precise mansioni da svolgere su "porzioni"di negozio (artefatti): troviamo chi prende gli ordini al banco, chi fa il pane, chi le torte e così via. Questi agenti cooperano interagendo tra di loro per raggiungere il macroobiettivo di far funzionare nel migliore dei modi l' attività. Questa metafora è un esempio di un MAS dove, nello stesso environment, convivono diversi agenti atti a compiere attività con-

correnti, coordinate, distribuite in spazio e tempo per il raggiungimento di un *goal* comune.

Gli agenti godono delle seguenti proprietà:

- **Autonomia**: è quella capacità che permette all' agente di compiere azioni senza che nessuno gliele imponga: è lui stesso che, secondo le sue credenze (beliefs), al verificarsi di certi eventi reagisce secondo dei piani (plans) più o meno prestabiliti. In parole povere vogliamo assegnare degli obiettivi (goals) all' agente che autonomamente deciderà il modo migliore per raggiungerli. L' agente è in grado di perseguire un obiettivo mettendo in atto dei piani che consistono in un insieme di azioni da svolgere.
- **Reattività**: essere reattivi vuole dire rispondere ai cambiamente dell' ambiente.
- **Abilità sociale**: permette agli agenti di coordinarsi e cooperare per raggiungere un obiettivo comune: questo vuol dire che gli agenti possono scambiarsi le conoscenze, i piani e gli obiettivi.

5.2 JaCa

JaCa nasce dall' unione di:

- Jason: linguaggio per programmare gli agenti.
- CArtAgO[1]: linguaggio che permette di creare ed eseguire artefatti utilizzabili dagli agenti.

Fino ad ora si è parlato di artefatti solo come porzioni di ambiente e come entità utilizzabili dagli agenti. L' artefatto mette a disposizione dell' utilizzatore un set di operazioni, questo ci fa capire che le operazioni eseguibili da un agente in un determinato ambiente dipendono dalle operazioni offerte dagli artefatti dell' environment. In particolare l' artefatto mette a disposizione:

[1]Common ARTifact infrastructure for AGents Open environments

- **proprietà osservabili**: usate dagli agenti per ottenere informazioni sullo stato corrente dell' ambiente.
- **eventi osservabili**: rappresentano possibili segnali generati dall' esecuzione di un evento.

Per comprendere meglio possiamo pensare alla proprietà osservabile come il display della macchinetta del caffè, ed il beep sonoro emesso da questa quando l' erogazione della bevanda è completa come l' evento osservabile.

Jason invece permette di specificare le caratteristiche degli agenti specificate nella sezione precedente. E' infatti possibile definire:

- **Beliefs**:

  ```
  our_belief(param)
  ```

- **Goals**:

  ```
  !our_goal(param)
  ```

- **Plans**:

  ```
  ourevent : context <- body
  ```

  Quando si verifica l'evento *ourevent* in un determinato contesto *context*, il nostro agente esegue le operazioni contenute in *body*.

Riportiamo di seguito un semplice esempio dove l' agente stampa il valore di ritorno di una azione messa a disposizione dall' artefatto.
Specifichiamo le entità del sistema:

```
MAS example03_output_param {

  environment:
  c4jason.CartagoEnvironment

  agents:
```

```
  calc_user agentArchClass c4jason.CAgentArch;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

Creiamo l' artefatto:

```
public class Calc extends Artifact {

  private String value;

  void init(String value) {
    this.value = value;
  }

  @OPERATION String getValue() {
    return this.value;
  }
}
```

Codice dell' agente, il cui obiettivo è quello di stampare il valore di ritorno della funzione getValue():

```
!print_value.

+!print_value : true <-
    println(getValue);
```

5.3 Livello applicativo in JaCa

Il punto finale di questo studio è l' integrazione della tecnologia JaCa con il middleware.

Definisco un artefatto, chiamato Board.java, che implementa una proprietà osservabile state: questa ha il compito di rendere visibile all' agente il valore del sensore potenziometro. La classe Board.java estende la classe Artifact, in ingresso al costruttore vuole il periodo di ripetizione della richiesta pull di aggiornamento del valore del sensore. Nel costruttore viene definita la proprietà osservabile e viene configurato tutto il middleware. Sono definite inoltre due operazioni, eseguibili dall' agente, che servono per accendere e spegnere il led; viene creata anche un' operazione interna che ha il

compito di prelevare il valore del potenziometro ed aggiornare la proprietà osservabile state. Di seguito se ne riporta il codice:

```
package test;

import comunication.NetworkProtocol;

import domain.*;
import system.*;
import cartago.*;

public class Board extends Artifact {

  private system.Middleware system;
  private boolean isFetching;
  private Actuator myLed;
  private Sensor myPotentiometer;
  private int period;

  void init(int period){
    this.defineObsProperty("state", 0);
    system = new Middleware();
    system.configureNetworkProtocol(NetworkProtocol.UDP);
    myLed = system.getActuator("l0");
    myPotentiometer = system.getSensor("p0");
    this.period = period;
    isFetching = true;
    this.execInternalOp("fetchSensorValue");
    log("configured.");
  }

  @OPERATION void on() {
    myLed.setValue(1);
  }

  @OPERATION void off() {
    myLed.setValue(0);
  }

  @INTERNAL_OPERATION void fetchSensorValue(){
```

```
      log("fetching.");
      while (isFetching){
         await_time(period);
         int value = myPotentiometer.getValue();
         this.updateObsProperty("state", value);
      }
   }
}
```

Il primo obiettivo che l' agente deve raggiungere è quello di test, che implica un nuovo sotto obiettivo chiamato setup il quale ha il compito di selezionare l' artefatto prima creato. Ogni volta che la proprietà osservabile cambia l' agente reagisce all' evento eseguendo le operazioni contenute nel corpo del piano *state* che viene selezionato sulla base del contesto. Di seguito si riporta il codice:

```
!test.

+!test : true <-
  println("Started.");
  !setup.


+state(V) : V > 400 & current_state(off) <-
  println("on - ",V);
  on;
  -current_state(off);
  +current_state(on).

+state(V) : V < 200 & current_state(on) <-
  println("off - ",V);
  off;
  -current_state(on);
  +current_state(off).


+state(V) <-
  println("State changed: ",V).
```

```
+!setup <-
  makeArtifact("myBoard","test.Board",[100],Id);
  focus(Id);
  println("setup done.");
  +current_state(off).
```

Descrivo in dettaglio i piani dell' agente:

- Ogni volta che la proprietà osservabile state si aggiorna ne viene stampato il valore a console.
- Ogni volta che il valore di state supera i 400 e lo stato corrente del led ci diche che è spento, allora l'agente lo accende rimuovendo la credenza che il led è spento ed aggiunge la credenza che il led è acceso.
- Ogni volta che il valore di state scende sotto i 200 e lo stato corrente del led ci dice che è acceso allora l' agente lo spegne rimuovendo la credenza che il led è acceso ed aggiunge la credenza che il led è spento.

5.4 Un altro esempio applicativo

Immaginiamo di dovere progettare un sistema di allarme all' interno di un monolocale di circa trenta metri quadrati con un bagno, tre finestre e una porta di ingresso. Possiamo pensare di utilizzare Arduino con il middleware descritto nei capitoli precedenti, quattro sensori PIR[2] ed una sirena.

Disponiamo un sensore pir nel bagno, e gli altri tre li puntiamo nella direzione delle due finestre e della porta di ingresso così da avere una protezione adeguata. Dopo avere collegato tutto alla board decido di creare un artefatto che rappresenta la casa e creo una proprietà osservabile "motionDetected"che mette in OR logico il valore dei sensori pir, inoltre metto a disposizione dell'agente un' azione che permette di fare partire la sirena. Il seguente codice realizza l' artefatto:

```
package test;

import comunication.NetworkProtocol;
```

[2]Passive InfraRed sensor che permette di rilevare il movimento delle persone individuando il calore umano.

```
import domain.*;
import system.*;
import cartago.*;

public class Board extends Artifact {

  private system.Middleware system;
  private boolean isFetching;
  private Actuator alarm
  private Sensor pir1;
  private Sensor pir2;
  private Sensor pir3;
  private Sensor pir4;
  private int period;

  void init(int period){
     this.defineObsProperty("motionDetected", false);
     system = new Middleware();
     system.configureNetworkProtocol(NetworkProtocol.UDP);
     pir1 = system.getSensor("p1");
     pir2 = system.getSensor("p2");
     pir3 = system.getSensor("p3");
     pir4 = system.getSensor("p4");
     alarm = system.getActuator("a0");
     this.period = period;
     isFetching = true;
     this.execInternalOp("fetchSensorValue");
     log("configured.");
  }

  @OPERATION void on() {
    alarm.setValue(1);
  }

  @OPERATION void off() {
    alarm.setValue(0);
  }

  @INTERNAL_OPERATION void fetchSensorValue(){
```

```
      log("fetching.");
      while (isFetching){
          await_time(period);
          int pir1val = pir1.getValue();
          int pir2val = pir2.getValue();
          int pir3val = pir3.getValue();
          int pir4val = pir4.getValue();
          if((pir1val + pir2val + pir3val + pir4val) == 0) {
            this.updateObsProperty("motionDetected", false);
          } else {
            this.updateObsProperty("motionDetected", true);
          }
       }
   }
}
```

Invece l' agente sarà così implementato:

```
!start.

+!start : true
     <- println("Started.");
        !setup.

+motionDetectede(V) : V == true & current_state(off)
     <- on;
        -current_state(off);
        +current_state(on).

+motionDetected(V) : V == false & current_state(on)
     <-off;
      -current_state(on);
      +current_state(off).

+!setup
    <-  makeArtifact("myBoard","test.Board",[100],Id);
        focus(id);
        +current_state(off).
```

Capitolo 6

Conclusioni

Questo studio ha portato alla realizzazione di un middlware Java che permette di assegnare e leggere valori di dispositivi montati su Arduino tramite i protocolli UDP e TCP. Inoltre il middleware mette a disposizione una interfaccia davvero semplice ed intuitiva (un metodo per la configuazione, uno per la scelta del protocollo di trasporto, due per prelevare sensori ed attuatori).

Il protocollo UDP realizza i modi di comunicazione PUSH e PULL, il protocollo TCP il modo PULL (questo perchè, nello scenario applicativo di una rete LAN, la probabilità di perdita dei pacchetti è veramente bassa ed ho quindi preferito privilegiare UDP).

È stato realizzato un protocollo di comunicazione tra l' elaboratore ed Arduino che invia/riceve **pacchetti di dimensione di soli 2 bytes**. Questo permette al middleware di essere reattivo e veloce: i test fatti sul Round-Trip-Time hanno evidenziato che nel **peggiore dei casi**, ovvero quando richiediamo un valore **una volta ogni millisecondo**, il tempo medio di andata e ritorno è di **9 centesimi di secondo**!

Per la gestione del middleware lato Arduino sono stati realizzati due sketch: uno per il protocollo UDP ed uno per il protocollo TCP che opportunamente integrati col protocollo di comunicazione sopra citato ed il middleware permettono la **programmazione del microcontrollore con un linguaggio di alto livello**.

Oltre ad avere creato un livello applicativo con il paradigma della programmazione orientata agli oggetti (OOP), ne ho creato un' altro con il paradigma di programmazione orientata agli agenti (AOP) sfruttando la

piattaforma JaCa.

Ringraziamenti

Un rigraziamento speciale ai miei genitori, Laura e Tiziano, che mi hanno permesso di portare a termine questo corso di studi e mi hanno sempre sostenuto ed incoraggiato. Ringrazio anche il resto della famiglia e gli amici per il supporto ricevuto.

Un altro ringraziamento speciale è per Nadia, sempre presente, sempre disponibile a spendere parole di incoraggiamento e ad illuminarmi il cammino sostenendomi anche nei momenti più difficili di questo percorso.

Infine ringrazio il mio relatore, Alessandro Ricci, ed il mio co-relatore, Andrea Santi, coi quali ho portato a conclusione questa tesi di laurea.

Bibliografia

[1] A.Natali and A.Molesini. *Costruire sistemi software. Dai modelli al codice.* Esculapio, 2009.

[2] A.Ricci and A.Santi. Agents as a paradigm for computer programming and software development. 2012.

[3] Bordini, Hubner, and Wooldridge. *Programming multi agent system in AgentSpeak using Jason.* Wiley, 2007.

[4] http://arduino.cc, 2012.

[5] http://it.wikipedia.org, 2012.