ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea in Ingegneria Informatica

TECNOLOGIE AD AGENTI E SISTEMI EMBEDDED

Elaborata nel corso di: Fondamenti di Informatica LB

Tesi di Laurea di:
SIMONE PIRINI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2013–2014
SESSIONE II

PAROLE CHIAVE

sistemi embedded
internet of things
agenti
jacamo
raspberry pi

Indice

Introduzione **vii**

1 Sistemi Embedded ed Internet of Things **1**

1.1 Introduzione 1
1.2 Sistemi embedded 1
1.2.1 Sistemi Operativi 2
1.2.2 System on chip 3
1.2.3 Dal punto di vista del programmatore 3
1.3 Internet of Things 5
1.3.1 Gli agenti 7

2 Framework per programmazione di sistemi embedded **9**

2.1 Middleware software 9
2.2 Oracle Java ME Embedded 10
2.2.1 Java Device I/O 12
2.3 Java Embedded System (JAHASE) 14
2.3.1 Strumenti di JavaES 15
2.3.2 Virtual Embedded Platform 15

3 Linguaggi di programmazione ad Agenti **17**

3.1 Gli agenti 18
3.1.1 Proprietà degli agenti 18
3.1.2 Differenze tra gli agenti e gli oggetti 21
3.1.3 L'ambiente 21
3.2 Architetture ad agenti 22
3.2.1 Architetture reattive ed ibride 23
3.2.2 Modello BDI e Practical reasoning 23

3.2.3 Approccio intenzionale 23
3.2.4 Ciclo di controllo Practical Reasoning System (PRS) 25
3.2.5 Comunicazione tra agenti 26
3.3 Programmazione orientata agli agenti 27
3.3.1 Beliefs 28
3.3.2 Goals 28
3.3.3 Plans 29

4 JaCaMo su Raspberry Pi 31
4.1 Modellazione del sistema embedded con un framework ad agenti 31
4.2 JaCaMo 33
4.2.1 Interazioni tra livelli 34
4.3 Raspberry Pi 35
4.3.1 Hardware 35
4.3.2 General Purpose Input/Output 36
4.3.3 Pi4J 37
4.4 JaCaMo su Raspberry Pi 38
4.4.1 Configurazione RPi 38
4.4.2 Configurazione JaCaMo 40

5 Conclusioni 47

Introduzione

> Information technology (IT) is on the verge of another revolution. Driven by the increasing capabilities and ever declining costs of computing and communications devices, IT is being embedded into a growing range of physical devices linked together through networks and will become ever more pervasive as the component technologies become smaller, faster, and cheaper. [..] These networked systems of embedded computers, referred to as EmNets throughout this report, have the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways.[..] The use of EmNets throughout society could well dwarf previous milestones in the information revolution.[..] IT will eventually become **an invisible component of almost everything** in everyone's surroundings[4].

Con il ridursi dei costi e l'aumentare della capacità di computazione dei componenti elettronici sono proliferate piattaforme che permettono al bambino come all'ingegnere di sviluppare un'idea che trasversalmente taglia il mondo reale e quello virtuale. Una collisione tra due mondi che fino a poco tempo fa era consentita esclusivamente a professionisti. Oggetti che possono acquisire o estendere funzionalità, che ci permettono di estendere la nostra percezione del mondo e di rivalutarne i suoi limiti. Oggetti connessi alla *rete delle reti* che condividono ed elaborano dati per un nuovo utilizzo delle informazioni.

Con questa tesi si vuole andare ad esplorare l'applicazione degli agenti software alle nuove piattaforme dei sistemi embedded e dell'Internet of

Things, tecnologie abbastanza mature eppure non ancora esplorate a fondo. Ha senso modellare un sistema embedded con gli agenti?

Capitolo 1

Sistemi Embedded ed Internet of Things

1.1 Introduzione

La capacità dell'uomo di creare utensili coi quali semplificarsi la vita contribuisce a qualificare la nostra specie come intelligente. L'uomo ha trovato nell'elaboratore elettronico un valido alleato permettendogli non solo di costruire strumenti inermi ma di dotare tali strumenti di funzioni fino a poco tempo fa impensabili.

1.2 Sistemi embedded

I sistemi embedded, o sistemi integrati, sono sistemi elettronici a microprocessore pensati e realizzati per applicazioni specifiche (special purpose). Si differenziano dai più noti (ma meno diffusi) sistemi general purpose per la loro impossibilità ad essere riutilizzati per scopi diversi da quelli per cui sono stati pensati.

Il compito di un sistema embedded non si limita a trasformare dei dati di ingresso in un output, bensì ad interagire con il mondo fisico percependolo e modificandolo in base alle sue funzioni. La stretta relazione tra un sistema embedded ed il mondo fisico con cui il sistema si trova ad interagire pone la progettazione in mano alla persona che più conosce e comprende quel mondo[8].

Alcuni oggetti diventati ormai di uso comune sono in realtà sistemi embedded. Basta pensare al mouse o alla tastiera di un pc, o al forno microonde tutti questi oggetti contengono un processore e un software che permette loro di eseguire le poche funzioni per i quali sono stati progettati.

Sebbene sia possibile realizzare un dispositivo, come quelli sopra citati, senza ricorrere ai sistemi embedded e quindi senza un processore e un software, bensì ricorrendo ai circuiti integrati personalizzati, è anche vero che l'utilizzo di un sistema embedded generalmente offre più flessibilità, risulta più economico e facile da progettare.

É possibile ricondurre l'alba dei sistemi embedded al 1971 quando Intel, in risposta alla domanda di un cliente (Busicom) che richiedeva un circuito integrato per i suoi calcolatori, sviluppa quello che sarà il primo microprocessore della storia interamente contenuto in un solo circuito integrato, Intel 4004, aprendo di fatto l'inizio dell'era dei sistemi embedded.

Le caratteristiche tecniche di un sistema embedded prevedono che i componenti siano dimensionati al minimo indispensabile per svolgere le funzioni per il quale il sistema è stato progettato. Questo, se per alcuni aspetti permette di tenere sotto controllo l'energia utilizzata (aspetto di primo rilievo in sistemi il cui approvvigionamento energetico può essere isolato da una rete elettrica), rappresenta un concreto limite alla programmazione del dispositivo.

1.2.1 Sistemi Operativi

Un sistema embedded si troverà inserito in un ambiente non noto a priori e avrà bisogno dal suo sistema operativo di funzionalità che generalmente, in un sistema desktop, non vengono considerate prioritarie. Tuttavia il sistema embedded userà altre funzioni del sistema operativo sensibilmente meno rispetto ad un pc desktop. Con la proliferazione di schede economiche per lo sviluppo di sistemi embedded ed il diffondersi degli smartphones i sistemi operativi si sono specializzati offrendo versioni che tengono in conto la scarsità delle risorse e la l'autonomia dell'applicativo. Queste nuove versioni dei sistemi operativi tendono a rendere più reattivo il kernel specializzandolo con funzionalità real-time.

1.2.2 System on chip

Un *system on a chip* integra tutti i componenti di cui ha bisogno un computer in un circuito integrato.

1.2.3 Dal punto di vista del programmatore

Sebbene la programmazione dei sistemi embedded può apparire come programmare piccoli computer, questo risulta estremamente non corretto quando si pongono i principali obiettivi di questi sistemi: interagire col mondo fisico. I sistemi embedded richiedono che alcune astrazioni di programmazione vengano integrate e assimilate dal software che si intende sviluppare per rispondere al meglio alle possibili situazioni nelle quali il sistema è calato. Proprietà di un sistema embedded:

- Tempestività
- Concorrenza
- Vivacità
- Interfacce
- Eterogeneità
- Reattività

La programmazione di un sistema embedded richiede, il più delle volte, una stretta interazione tra hardware e software. Avendo a disposizione risorse estremamente limitate (per esempio, un processore la cui potenza computazionale sia solo sufficiente alle esigenze e non sovradimensionato permette un risparmio consistente in termini di potenza elettrica) lo sviluppatore è spinto a conoscere intimamente la piattaforma hardware utilizzata al fine di sfruttarne tutte le capacità e in alcuni casi, lo sviluppatore si troverà persino a modificare le caratteristiche tecniche del sistema per migliorare le proprietà sopra citate relative all'ambiente in cui il suo sistema dovrà operare.

Questo non accade nella totalità dei casi perché esistono piattaforme per sviluppare sistemi embedded con una elevata ed efficace astrazione dell'hardware. Questo permette al programmatore di concentrarsi solo sul

livello applicativo confidando che gli strumenti di sviluppo ottimizzino il software al meglio per il sistema embedded finale[1].

Proprietà del software embedded

- Efficienza del codice:

  La capacità di rendere ogni singola riga di codice il più efficace in termini di prestazioni fa la differenza nel momento in cui si hanno a disposizione limitate risorse.

- Interfacce delle periferiche:

  Le periferiche rappresentano in un sistema embedded le estensioni grazie alle quali è possibile interagire con l'ambiente circostante: percependolo (sensori) e modificandolo (attuatori).

- Robustezza del codice:

  Una delle differenze più entusiasmanti dal punto di vista della programmazione è la necessità che il sistema software sul dispositivo non debba ricorrere ad un riavvio (più o meno forzato che sia) ma che lavori in un periodo virtualmente infinito senza la necessità dell'intervento dell'uomo.

- Riusabilità del codice:

  Sebbene sistemi embedded di produttori diversi abbiano peculiarità a volte molto differenti gli uni dagli altri, la possibilità di riutilizzare il codice per progetti diversi e magari su sistemi hardware diversi rimane uno strumento molto potente in mano allo sviluppatore, che in questo modo potrà, con pochi accorgimenti, copiare funzionalità anche complesse da un sistema al altro in poco tempo.

- Strumenti di sviluppo:

  Gli strumenti per il debugging sono generalmente abbastanza limitati in quanto risulta piuttosto difficile, nonostante il livello di astrazione fornito, riuscire a catturare le peculiarità di una piattaforma di cui non si possono conoscere le finalità. Generalmente il produttore di piattaforme embedded correda di strumenti di emulazione software che fornisce al programmatore un buon inizio per la fase di debugging. Avendo a che fare con ambienti le cui variabili non possono essere

previste in maniera esaustiva all'interno dei modelli utilizzati per l'astrazione del mondo circostante ai fini del proprio progetto, la fase, per ora, di debugging prevede inevitabilmente prove sul "campo".

1.3 Internet of Things

L'Internet of Things (IoT) è l'ennesimo esempio che vede due tecnologie, internet (sistema globale di reti di computer interconnessi) e i sistemi informatici, ampiamente consolidate individualmente unirsi in qualcosa di nuovo che sfrutta i vantaggi e le potenzialità di entrambe le tecnologie. Il termine individua quindi tutti quei sistemi embedded che accedono ad internet per comunicare ed estendere o integrare i propri servizi[14].

Il primo esempio di IoT può essere fatto risalire agli inizi degli anni ottanta nella prestigiosa Università Carnegie Mellon di Pittsburgh (Pennsylvania) dove ad un banale distributore di CocaCola è stato esteso l'accesso all'allora Internet per inviare informazioni sullo stato del rifornimento della nota bevanda zuccherata al fine di non lasciare vuoto il distributore. Già con questo primo esperimento risulta ovvio che se è possibile connettere ad internet un distributore di bevande ed essendo internet una risorsa limitata (l'indirizzamento IPv4, adottato nel 1981, permette di collegare un massimo di poco meno di 2^{32} dispositivi) lo spazio di indirizzamento di internet risulterà insufficiente ed è anche per questo che l'adozione dello standard IPv6, con il conseguente aumento a poco meno di 2^{128} possibili indirizzi, permetterà all'IoT un pieno e libero sviluppo. [11]

La persona che per prima formalizzò il termine *Internet of Things* fu Kevin Ashton: in un articolo per l'*RFID Journal* dopo aver evidenziato i limiti di un internet di computers, in cui viaggiano solo ed esclusivamente idee e contenuti prodotti dall'uomo, mise in evidenza come l'uomo interagisca con gli oggetti e ne sia circondato e come questi rappresentino parte integrante del mondo di ciascuno di noi. Permettere alle *cose* di aver uno spazio su internet in cui condividere e ricevere informazioni ci potrebbe permettere di tracciare praticamente tutto: potremmo sapere se un oggetto è arrivato alla fine della sua vita utile e sostituirlo o ripararlo nel momento più opportuno, potremmo sapere quando innaffiare le piante in base all'umidità del terreno eccetera[9].

Un esempio "familiare" di questa integrazione può essere quello dei cronotermostati di ultima generazione che oltre alle funzioni tradizionali di programmazione oraria e stagionale offrono la possibilità di un controllo remoto attraverso applicativi web-based o mobili. Si rende possibile interpellare e modificare il cronotermostato senza il bisogno di una interazione diretta oppure lasciare al sistema stesso la possibilità di autoregolare il sistema in base alle nostre esigenza solamente "studiando" le nostre abitudini grazie a sensori appositamente interfacciati.

Le *Smart cities* sono un esempio di applicazione su larga scala di IoT. Alcune città offrono al pubblico i dati raccolti in alcuni contesti (mobilità, energia, rifiuti, ecc..) per permettere a sviluppatori o *makers* (artigiani/-hobbisti di sistemi embedded e IoT) di realizzare applicazioni o sistemi embedded.

Scenari più o meno realistici e catastrofici sul futuro prevedono che ogni oggetto (fisico) dell'ambiente possa comunicare attraverso internet.
Con l'IoT stiamo entrando in una era in cui gli utenti di internet saranno miliardi e di questi solo la minoranza sarà composta da esseri umani. Nonostante possa apparire come se si volesse dare semplicemente una connessione internet agli oggetti e quindi essere di utilità limitata se non del tutto assente, quello che si sta già realizzando pare essere di grande impatto. Basti pensare alle applicazioni mediche: l'elettrocardiogramma dinamico, ad esempio, richiede di indossare un dispositivo in grado di misurare l'attività elettrica del cuore nell'arco di un'intera giornata. Tuttavia questo esame avviene tramite un dispositivo che viene letteralmente cablato sul paziente senza fornire informazioni durante il periodo che viene indossato. L'integrazione tra IoT ed wearable devices (dispositivi indossabili) potrebbe permettere ai medici di monitorarne le variazioni in tempo reale ed eventualmente intervenire in casi di anomalie o emergenze, mentre il paziente potrebbe trarne beneficio sia da un punto di vista del comfort che, ovviamente, della salute.
Ma, al di là dell'esempio specifico, l'unico limite allo sviluppo di IoT sarà come sempre la fantasia umana. Se oggi può apparire come l'ennesima forzatura in una società alla ricerca continua di nuovi flussi informativi, domani, l'IoT potrebbe ribaltare i paradigmi comunicativi non solo macchina-macchina bensì macchina-uomo e addirittura uomo-uomo, con risvolti difficilmente prevedibili.

1.3.1 Gli agenti

L'Internet of Things offre una potente piattaforma che permette di collegare sia componenti software tra loro che con servizi internet.
Usando l'IoT gli agenti possono inviare aggiornamenti sul loro stato e richieste di servizi a chiunque sia interessato.

L'architettura che più sembra adattarsi all'IoT è quella *orietata agli eventi*: ovvero un pattern architetturale che in base all'evento rilevato/generato produrrà, rileverà e reagirà in base alle logiche implementate. Come vedremo nel capitolo 3, gli agenti avendo proprietà simili a quelle umane (autonomia, capacità sociali come collaborare, coordinare e negoziare), si candidano ad essere un ottimo alleato per l'IoT. In sistemi complessi con componenti autonomi e forte necessità di comunicazione i sistemi multi-agente si adattano perfettamente. Il connubio tra IoT e sistemi multi-agente sembra essere una naturale evoluzione di entrambe le tecnologie.

Capitolo 2

Framework per programmazione di sistemi embedded

Un numero sempre crescente di applicazioni innovative fa affidamento a piattaforme di architetture di sistemi eterogenei. Nella progettazione di sistemi complessi una delle attività che richiede il maggior impiego di tempo è la definizione delle interfacce di comunicazione tra i diversi componenti appartenenti alle differenti architetture. Oltre ad essere una complessa attività rappresenta anche un limite alla flessibilità del sistema e generalmente la soluzione trovata è di difficile riuso.

Il *middleware* è la parte di software che opera tra il sistema operativo ed il livello applicativo per risolvere le problematiche relative a riuso, portabilità ed affidabilità.

2.1 Middleware software

Un sistema embedded è generalmente composto da un'architettura che comprende sei moduli:

- Applicazioni
- Librerie
- File system

- OS kernel
- Driver periferiche
- Hardware

Questa tipo di architettura presenta degli inconvenienti: problema di riuso di alcuni componenti, la portabilità per varie applicazioni, la dipendenza per piattaforme trasversali, flessibilità durante lo sviluppo del sistema, la scalabilità.

Un *middleware* è un livello di astrazione il cui compito è rendere omogeneo la comunicazione tra i componenti di un sistema distribuito. Possiamo vedere un *embedded system* come un tipo di sistema distribuito e quindi pensare a un *middleware* che armonizzi le comunicazioni nell'architettura interna del sistema embedded. Un sistema embedded può infatti essere visto come se fosse composto da un insieme di risorse eterogenee collegate attraverso una infrastruttura e presentare gli stessi problemi di scalabilità, eterogeneità ecc di cui un middleware può offrire una soluzione [12].

Una possibile architettura di un sistema embedded con middleware dovrebbe inserirsi tra il livello applicativo e quello relativo alle librerie, offrendo un insieme di interfacce a disposizione delle applicazioni.

2.2 Oracle Java ME Embedded

Scrivere programmi in Java ha enormi vantaggi dal punto di vista della riusabilità del codice. Questo è dovuto all'utilizzo della Java Virtual Machine (JVM) che creando un ambiente virtuale permette l'astrazione completa sia dell'hardware(cpu, memorie, periferiche) della specifica macchina che del suo software (sistema operativo e librerie). Ovviamente tutto questo lavoro deve essere supportato da delle buone prestazioni delle macchine su cui la JVM viene installata.

Nei sistemi embedded la JVM si trova a dover lavorare con delle risorse ridotte al minimo. Per questa ragione Oracle ha sviluppato la piattaforma Java Micro Edition Embedded. Questa versione di Java tenta di ridurre il carico della CPU trasferendo i processi più avari di risorse su una macchina *host* (presumibilmente più performante del dispositivo embedded) in cui si potrà sviluppare il software per poi distribuirlo sul dispositivo embedded.

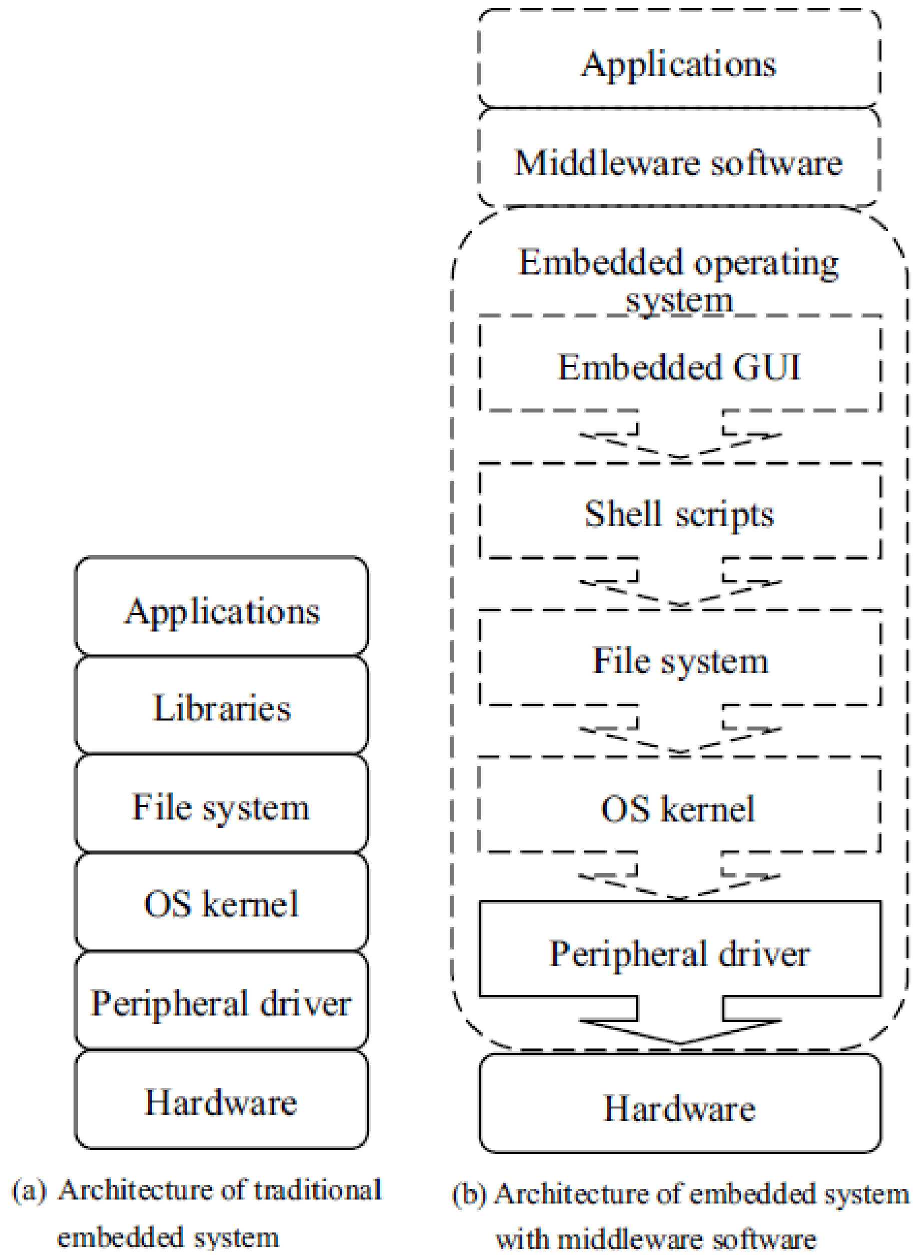

Figura 2.1: [5]

Questa piattaforma a livello logico non essendo vincolata a nessuna risorsa hardware in particolare permette lo sviluppo di applicativi ad alto livello e facilmente portabili da un sistema embedded ad un altro.
Uno degli aspetti più rilevanti ai fini di questa tesi è il livello di astrazione che le librerie della Java ME Embedded offrono per le interfacce delle periferiche General-Purpose I/O, I2C, PWM ed altre[10].

2.2.1 Java Device I/O

Il progetto Device I/O, sviluppato dalla comunità *OpenJDK*, mette a disposizione delle Java Application Programming Interface (API) per accedere a periferiche generiche su dispositivi embedded. Include il supporto per le periferiche di maggiore uso sulle piattaforme embedded disponibili oggi sul mercato: General Purpose Input/Output (GPIO), convertitori ADC e DAC, contatori, dispositivi Inter-Integrated Circuit Bus(I2C), PWM, Universal Asynchronous Receiver/Trasmitter (UART), Serial Peripheral Interface ed altri. Fornendo un controllo di basso livello riesce a coprire anche aspetti di sicurezza, concorrenza e gestione dell'energia consumata.
Il vantaggio di poter lavorare con questa libreria nativa in Java, oltre alla riusabilità del codice come già detto, non è ancora molto evidente essendo questa libreria relativamente "giovane" (giugno 2014). Tuttavia la potenza di una piattaforma di questa portata potrebbe essere paragonabile al processo che ha portato lo standard JDBC sui database a fine anni 90. Essendo la mancanza di standard uno dei motivi per cui molte piattaforme oggi disponibili non vengono utilizzate nel pieno delle loro funzionalità, una libreria come Java Device I/O con il suo livello di astrazione potrebbe coprire il gap presente e, finalmente, legare le periferiche di basso livello ad un linguaggio di programmazione ad alto livello.

Utilizzo di Java device I/O

Un veloce esempio di questo framework può essere rappresentato dalla modellazione di un sensore di movimento: l'HC-SR501 PIR MOTION DETECTOR.
Questo sensore si basa sulla tecnologia ad infrarossi, ha un'alta sensibilità, alta affidabilità, modalità a basso consumo di energia, largamente usato nell'industria. L'integrato si presenta con due trigger per la regolazione del-

la sensibilità e del tempo di reazione; tre pin forniscono l'interfaccia per il collegamento alla linea di alimentazione, di terra e dati.

```java
import jdk.dio.gpio.GPIOPin;
import jdk.dio.gpio.GPIOPinConfig;
import jdk.dio.gpio.PinListener;

public class HCSR501 {
   private GPIOPin pin = null;
/*TEST Local_artifacts_rpiDevs!*/
   public HCSR501(int pinGPIO) {
      try {
         //Establishing the initial conditions
         pin = (GPIOPin) DeviceManager.open(new GPIOPinConfig(0,
             pinGPIO,
               GPIOPinConfig.DIR_INPUT_ONLY,
               GPIOPinConfig.MODE_INPUT_PULL_DOWN,
               GPIOPinConfig.TRIGGER_RISING_EDGE, false));
      } catch (InvalidDeviceConfigException e) {
```

Listing 2.1: Importo i componenti per l'apertura e la gestione dell'interfaccia GPIO, contenuti nelle librerie jdk.dio; implemento un costruttore parametrizzato con il pin dati che apra una connessione con il pin e lo imposta in modalità di input. Inoltre richiede che gli sia notificato quando il dato varia da basso ad alto.

```java
      }
      //I2CUtils.I2Cdelay(3000); // wait for 3 seconds
   }

   //Method that supports motion detection events
   public void setListener(PinListener pirListener) {
      if (pin != null)
         try {
            pin.setInputListener(pirListener);
         } catch (ClosedDeviceException e) {
            e.printStackTrace();
         } catch (IOException e) {
            e.printStackTrace();
```

```
14          }
15    }
16
17    //Closing the pin and freeing the listener
18    public void close() {
19       if (pin != null) {
20          try {
21             pin.setInputListener(null);
22          } catch (ClosedDeviceException e) {
23             e.printStackTrace();
24          } catch (IOException e) {
25             e.printStackTrace();
26          }
27          try {
28             pin.close();
29          } catch (IOException e) {
30             e.printStackTrace();
31          }
32       }
33    }
34 }
```

Listing 2.2: Il metodo *setListener* intercetterà l'evento generato dal PIR mentre il metodo *close* andrà a eliminare la sottoscrizione all'evento e a chiudere la connessione con l'interfaccia GPIO.

2.3 Java Embedded System (JAHASE)

Java Embedded System (JavaES) è un framework sviluppato da Holgado-Terriza ed Viúdez-Aivar, basato su Java che mette a disposizione una piattaforma adattabile e flessibile per lo sviluppo di applicazioni su dispositivi embedded eterogenei. Questa piattaforma fornisce un'astrazione di un sistema embedded che include sia componenti hardware che software. Viene fornito, in oltre, un modello di programmazione dalla semantica ben definita per sviluppare su sistemi embedded con limitate risorse, astraendo la complessità e l'eterogeneità che potrebbe caratterizzare dei sistemi embedded fisici. Per esempio in JavaES vengono offerti meccanismi e strutture di alto livello che rendono possibili un accesso e gestione sicura ed affidabile

del livello hardware (I/O, timers, segnali PWM o memoria).
É possibile ottenere l'indipendenza della macchina virtualizzando sia il sistema hardware (le risorse) che l'infrastruttura software (OS, JVM o i drivers). Questa astrazione sarà effettuata in un livello che divide l'applicazione Java dal dispositivo fisico preservando la portabilità del codice. Questo permette l'integrazione di componenti software e quasi ogni periferica al livello hardware, rendendo possibile la personalizzazione del sistema embedded senza dover programmare un driver o ricorrere alle librerie *native*

2.3.1 Strumenti di JavaES

Il framework JAVAES mette a disposizione una serie di strumenti che permettono un'appropriata astrazione e delle API per sviluppare su dispositivi embedded. Il componente principale è la *Virtual Embedded Platform* (VEP) che definisce una piattaforma funzionale astratta del dispositivo destinatario. La VEP è indipendente dalla macchina di destinazione e completamente configurabile in base ai requisiti dell'applicazione. JavaES mette a disposizione una versione personalizzata del *Runtime Environment* appositamente per la VEP: VEP-RE. La VEP-RE contiene un infrastruttura software con le astrazioni necessarie e i componenti richiesti per un applicazione embedded. Inoltre ha uno strumento, basato sul tool *Ant*, per la distribuzione ed il *building* [7], semplificando lo sviluppo di un applicazione per dispositivi di destinazione diversi.
Completa gli strumenti a disposizione un *debugger*: *JaCoDES*; che permette di aggiungere un *monitor* all'interno dell'immagine dell'applicativo che gli permette di tracciare l'esecuzione del programma. Con lo stesso strumento è possibile realizzare una versione personalizzata del *Runtime Environment* per la VEP specifia dell'applicazione.

2.3.2 Virtual Embedded Platform

A differenza delle applicazioni desktop, un sistema embedded necessita di indirizzare le eventuali problematiche software ed hardware allo stesso tempo dal momento che le sue funzionalità dipendono dalla capacità di esplorare l'ambiente fisico attraverso l'uso delle periferiche di I/O.
Le proprietà d'astrazione e le caratteristiche di un VEP vengono descritte in

un documento: il *VEP Specification*(XML). Tale documento è organizzato in sette sezioni:

- Execution Environment
- I/O Management
- Memory
- Communications
- Quality Attributes
- Time Services
- Debug ToolSet

Capitolo 3

Linguaggi di programmazione ad Agenti

L'ingegneria del software ha progressivamente migliorato la sua conoscenza delle caratteristiche della complessità del software. Le interazioni sono probabilmente la più importante caratteristica di un software complesso.

Gli agenti, come vedremo in questo capitolo, rappresentano uno strumento per la comprensione della società umana: i sistemi multiagente mettono a disposizioni un innovativo strumento per simulare la società aiutando in questo modo a far luce su diversi aspetti dei processi sociali.

La complessità della rete globale richiederà nel futuro una sempre maggiore decentralizzazione e autonomia decisionale. Gli agenti software nonostante siano stati ampiamente studiati non hanno ancora guadagnato, negli ambienti industriali (commerciali), quella fiducia necessaria per la loro implementazione su larga scala. Questo può essere dovuto alla mancanza di standardizzazione. Una standardizzazione delle interfacce nell'Internet of Things può essere la spinta che permetterà al software ad agenti di diffondersi[14].

Ad oggi, l'architettura Belief-Desire-Intention (descritta nel paragrafo 3) rappresenta uno dei migliori approcci per lo sviluppo di agenti dotati di caratteristiche cognitive. Nella ricerca di un linguaggio di programmazione adatto alle caratteristiche degli agenti BDI la scelta, ai fini di questa tesi, di AgentSpeak è dovuta al livello di astrazione che questo linguaggio forniva ed alla consolidata esperienza della comunità.

Jason è un interprete di AgentSpeak, integra la *teoria degli atti lingui-*

stici e la comunicazione inter-agente. Altra caratteristica rilevante sta nella sua implementazione in Java (quindi potenzialmente svincolato da una piattaforma specifica) e nell'essere distribuito sotto licenza libera (GNU LGPL)[3].

3.1 Gli agenti

Il punto focale degli agenti è la loro *autonomia*: la capacità di agire autonomamente in un ambiente al fine di raggiungere gli obiettivi assegnatogli.

Una definizione di agente autonomo può essere fornita distinguendo tra agenti e programmi (procedurali). Se nel caso di questo ultimo la struttura operativa può essere ridotta in *input-computazione-output* (programmi funzionali secondo il modello matematico $f : I \rightarrow O$)nel caso degli agenti non è possibile applicare tale struttura non essendo un semplice elaboratore di input per fornire output. La reattività richiesta ad alcuni sistemi non prevede l'eventualità che il sistema possa terminare, viene richiesto che sia continuamente attivo e rispondente. Vengono chiamati *agenti* perché sono pensati per essere attivi, produttori intenzionali di azioni: quando si trovano nel loro ambiente dovranno raggiungere degli obiettivi per conto dei loro progettisti cercando di trovare il modo migliore per farlo.

È opportuno pensare ad un agente come un individuo monoblocco continuamente in interazione con il suo ambiente:

percepisce - *decide* - *agisce* - *percepisce* - *decide* - ...

Un esempio di agente elementare può essere un termostato: l'obiettivo che gli è stato delegato è mantenere costante la temperatura della stanza e le azioni che ha a disposizione sono l'accensione e lo spegnimento del riscaldamento. Il termostato rappresenta tuttavia un esempio di agente piuttosto banale in quanto il processo decisionale non sarà altro che una struttura condizionata (if-then-else).

3.1.1 Proprietà degli agenti

Al dì là di una definizione univoca, difficilmente formalizzata fino ad ora, in una visione più *opertiva* possiamo individuare le proprietà che accomunano gli agenti software[6]:

- Reattività (Percepire e agire sull'ambiente)
  Capacità di rispondere in modo tempestivo ai cambiamenti dell'ambiente. La maggior parte degli ambienti, infatti, è dinamico e tale caratteristica rende molto difficoltosa la costruzione del software. Il programma dovrà quindi tenere in considerazione la possibilità del fallimento, conseguentemente potrà valutare se vale la pena eseguire una qualche azione.
  Un sistema reattivo dovrà quindi mantenere una continua interazione con il suo ambiente e rispondere ad eventuali cambiamenti dello stesso, tale risposta dovrà essere data in un tempo per cui sia utile.

- Autonomia
  Esercitare controllo rispetto le proprie azioni. Vogliamo che l'agente sia in grado di decidere il miglior modo di agire per raggiungere gli obiettivi che riceverà. Gli agenti autonomi sono situati e pensati per specifici ambienti. Nel momento in cui un agente si trova in un ambiente differente rispetto quello per cui è stato pensato può smettere di essere un agente (perdendone le caratteristiche). Per esempio alcuni sensori potrebbero non essere più efficaci per "sentire" l'ambiente circostante o alcuni attuatori possono risultare inutili.

- Orientamento all'obiettivo (pro-attività intenzionale)
  Non reagire semplicemente in risposta all'ambiente. Reagire all'ambiente non è una cosa difficile (ad uno stimolo faccio corrispondere una regola, quindi un'azione) ma generalmente si vuole che l'agente reagisca facendo qualcosa per noi, sull'ambiente, per sé stesso o per altri agenti.

  Particolarmente difficile sarà riuscire a bilanciare in maniera ottimale il comportamento reattivo con la propensione al raggiungimento degli obiettivi. L'agente dovrà generare e tentare di raggiungere gli obiettivi non soltanto guidato dagli eventi ma prendendo l'iniziativa. Per far ciò dovrà essere in grado di riconoscere le eventuali opportunità.

- Continuità temporale
  Sistema in continua esecuzione.

- Comunicazione (abilità sociale)
  Interagire con altri agenti (e possibilmente anche con umani) al fi-

ne di raggiungere gli obiettivi cooperando, coordinando e negoziando.
La cooperazione consiste nel lavorare insieme, come una squadra, per raggiungere un obiettivo "condiviso". Un agente potrebbe non essere in grado di raggiungere da solo un obiettivo oppure essere consapevole che cooperando con altri agenti potrebbe raggiungere l'obiettivo traendone vantaggi (risparmio di risorse, tempo impiegato, aumento delle conoscenze dell'ambiente, ecc..).
Gli agenti applicano la coordinazione gestendo le interdipendenze tra diverse attivit̀; se ad esempio devono condividere risorse questo sarà il modo migliore per farlo.
a La negoziazione è l'abilità a raggiungere accordi su materie di interesse comune, in modo da poter prendere in considerazione un eventuale peggioramento ai fini di una più probabile riuscita positiva dell'attività negoziata. Il mondo reale può essere astratto come un ambiente multi-agente ed alcuni obiettivi sarà possibile raggiungerli solamente tramite l'interazione con altri agenti.

- Apprendimento (addattabile)
  Cambiare il suo comportamento in base alle esperienze precedentemente "vissute". Tra alternative equamente valide al fine del raggiungimento di un obiettivo la scelta intrapresa potrebbe rivelarsi non ottimale fallendo di fatto la buona riuscita del piano, sarà quindi opportuno fare tesoro dei fallimenti per essere in grado di mettere in campo piani diversi a parità di condizioni iniziali.

- Mobilità
  L'agente sarà in grado di trasportarsi da una macchina ad un'altra.

- Flessibilità
  Le azioni non fanno parte di un copione predefinito.

- Personalità
  Credibile personalità e stato emotivo.

Secondo la nostra definizione ogni agente soddisfa le prime quattro proprietà, mentre aggiungendo le altre proprietà si possono creare classi di agenti in base alle esigenze del problema/ambiente.

Un agente quindi si troverà a dover effettuare delle azioni tramite i propri attuatori per modificare efficacemente la percezione dell'ambiente circostante in modo tale da raggiungere determinati obiettivi.

3.1.2 Differenze tra gli agenti e gli oggetti

Gli agenti a prima vista possono apparire come degli *oggetti* molto elaborati. Tuttavia oltre a provenire da idee concettualmente diverse hanno caratteristiche fondanti differenti: gli oggetti incapsulano uno stato, comunicano attraverso il passaggio di messaggi ed hanno metodi (corrispondenti alle operazioni che è possibile fare sullo stato interno dell'oggetto stesso). Un agente, invece, incapsula un forte concetto di autonomia: in particolare decide da solo se svolgere o meno un'azione su richiesta di un altro agente (non è previsto il trasferimento del flusso di controllo in quanto ogni agente ne possiede uno autonomo).

Gli agenti possono, potenzialmente, avere comportamento intelligente grazie alla loro flessibilità (*reattività*, *proattività*, *sociali*).

Infine gli agenti sono entità attive (agiscono perché vogliono farlo) e non dei fornitori passivi di servizi.

3.1.3 L'ambiente

Un ambiente *accessibile* si ha quando un agente può ottenere informazioni complete, accurate ed aggiornate sullo stato dell'ambiente. Gli ambienti moderatamente complessi sono generalmente *inaccessibili.* La complessità di un agente crescerà all'aumentare della complessità dell'ambiente in cui opera.

Parlando della *reattività* si è detto che l'agente deve essere in grado di gestire un eventuale fallimento, in un ambiente *deterministico*, in cui un azione ha un singolo e garantito effetto, non ci sarà incertezza sullo stato finale derivante dall'esecuzione di un'azione. Tuttavia il mondo fisico può essere a tutti gli effetti considerato *non deterministico* il che rappresenta un grande problema (e sfida) per i progettisti di agenti.

In un ambiente *episodico* l'esecuzione di un'azione dipende dal numero degli episodi discreti senza stabilire un collegamento tra l'esecuzione di un agente in scenari differenti. Questo tipo di ambiente risulta di facile gestione

da parte dello sviluppatore di agenti in quanto l'agente potrà decidere quale azione eseguire in base al solo episodio corrente.

In un ambiente *statico* non ci saranno altre variazioni eccetto quelle apportate dall'esecuzione di un azione dell'agente. In uno *dinamico*, invece, avremo processi operativi che cambiano l'ambiente fuori dal controllo dell'agente.

Un ambiente *discreto* avrà un fisso e finito numero di azioni e percetti.

3.2 Architetture ad agenti

L'architettura di un agente corrisponde alla sua progettazione software, possiamo distinguere all'interno di un'architettura le *strutture dati*, le *operazioni* che è possibile eseguire sulle strutture dati ed il *flusso di controllo* tra le operazioni.

Tipologie di agenti

- Agenti a ragionamento simbolico
  Il processo decisionale si avvale di esplicito ragionamento logico. L'approccio classico alla costruzione di agenti è permesso tramite la visione degli stessi come un tipo particolare di sistema basato su una conoscenza (*knowledge-based system*), in questo paradigma, conosciuto come *symbolic AI*, è possibile definire degli agenti deliberativi contenente un esplicita rappresentazione del modello simbolico del mondo, in grado di prendere decisioni tramite il ragionamento simbolico.

- Agenti reattivi
  Pensati in risposta alle problematiche derivanti dagli *agenti a ragionamento simbolico*. L'agente, generalmente, farà corrispondere un'azione agli stimoli percepiti. L'obiettivo sarà quello di dare sempre una risposta tempestiva.

- Agenti ibridi
  Architettura ibrida che tenta di combinare il meglio delle architetture simboliche e reattive.

3.2.1 Architetture reattive ed ibride

Gli agenti che applicano il *ragionamento simbolico* non risultano avere una risposta adatta alle esigenze dell'ambiente. Infatti nel momento in cui acquisiscono informazioni e le modellano simbolicamente, il mondo circostante potrebbe essere variato rendendo inutile il lavoro fatto dall'agente.

In risposta a queste problematiche è stata sviluppata l'*architettura reattiva*, che consiste nel dare un comportamento intelligente in grado di lavorare senza rappresentazioni simboliche.

Con l'*architettura ibrida* si cerca di coniugare le due architetture sopra citate in modo che i vantaggi di una possano compensare i difetti dell'altra in quanto nè un approccio completamente deliberativo nè uno completamente reattivo è adatto alla costruzione di agenti.

L'idea dell'architettura ibrida è quindi quella di costruire un agente dotato di almeno due sottosistemi: uno *deliberativo*, che sviluppa piani e prende decisioni nel modo proposto da IA simbolica; un altro *reattivo*, che sia in grado di reagire agli eventi senza ricorrere a ragionamenti complessi.

3.2.2 Modello BDI e Practical reasoning

Il modello BDI (beliefs Desires Intentions) permette di individuare le operazioni che competono ad un agente. Il modello si ispira al modello di comportamento umano che prevede tre fasi per raggiungere un obiettivo. La prima fase consiste nel controllare le informazioni a disposizione dell'ambiente. Queste informazioni possono essere state ottenute grazie ai sensori di cui si ha disponibilità, possiamo averle annotate precedentemente o possono esserci state inviate da altri agenti. La fase intermedia è quella caratterizzata dal *desiderio*. Ovvero in questa fase l'agente deve scegliere tra tutti gli obiettivi di cui ha conoscenza. Infine la fase delle intenzioni permette all'agente di attuare dei piani al fine di raggiungere gli obiettivi definiti nella fase precedente.

3.2.3 Approccio intenzionale

L'approccio del modello BDI si differenzia fortemente da quello procedurale o ad oggetti mimando quello umano. Secondo il filosofo Daniel Dennett un sistema il cui comportamento può essere predetto e spiegato in termini di "atteggiamento" (come le belief, desire ed intention) viene definito *sistema*

intenzionale. Gli esseri umani comunicano usando questo meccanismo attraverso altri sistemi di comunicazione. Si tratta, infatti, di informazioni e deduzioni aggiuntive implicitamente contenute in altre comunicazioni. Per esempio la frase "Marco intende scrivere una relazione" non solo ci informa sul comportamento di Marco ma ci da un insieme di informazioni che possono farci prevedere il suo comportamento: non ci sorprenderemmo, per esempio, se trovassimo Marco particolarmente scontroso o se lo vedessimo stare parecchio tempo al computer mentre ci sorprenderebbe trovarlo a far festa fino a tardi. L'*intentional stance* sembra un ottimo meccanismo di astrazione per la gestione di sistemi ad alta complessità. Permette un modo di comunicazione, non tecnico, molto familiare all'uomo e rappresenta il primo paradigma post-dichiarativo.

Non viene più richiesto di specificare esattamente cosa un sistema deve fare scrivendo un dettagliato algoritmo che possa produrre un uscita in base all'ingresso dato, una delle criticità della programmazione procedurale è appunto il livello di dettaglio richiesto in situazioni non chiare a priori. La programmazione dichiarativa, invece, richiedendo un obiettivo che il sistema deve raggiungere lascia alla macchina il controllo di scegliere come raggiungere l'obiettivo. In questo caso il meccanismo di controllo implementa un modello di azione ragionata.

Per produrre delle azioni concrete sull'ambiente il nostro sistema sfrutterà le credenze (beliefs), i desideri (desires) e le intenzioni (intentions) a propria disposizione, il modello decisionale che permette ciò viene definito *practical reasoning*, si tratta di un ragionamento orientato al azione, con un processo il cui fine è capire cosa fare.
Il *ragionamento pratico* dell'essere umano si può dividere nelle attività di *deliberazione* e *ragionamento per obiettivi.*

Deliberazione

L'attività di *deliberazione* corrisponde, nel mondo degli agenti, alla scelta delle *intenzioni.* Le *intenzioni* hanno un ruolo cruciale nel modello BDI: tendono a portare ad un azione; tendono a filtrare l'eventuale adozione di altre intenzioni che potrebbero causare un conflitto. La scelta di perseguire una particolare intenzione è strettamente legata alla *belief base* di ciascun agente. Una volta intrapresa un'intenzione, indipendentemente dal suo successo o fallimento, le implicazioni future si faranno sentire nei futuri

processi decisionali di deliberazione. Se un'intenzione si rivelerà fallimentare in determinate condizioni, l'agente sarà propenso ad evitare la stessa deliberazione al ripresentarsi di quelle condizioni.

Ragionamento per obiettivi

Ragionamento per determinare come raggiungere un obiettivo utilizzando i mezzi a disposizione. Il ragionamento per obiettivi produrrà dei *piani* generati da un algoritmo di pianificazione. L'obiettivo di un piano sarà portare l'ambiente da uno stato iniziale ad uno stato finale, che corrisponderà all'obiettivo/intenzione realizzato. Il compito di un agente è quello di assemblare i vari piani a sua disposizione, sviluppati dal programmatore, per raggiungere l'obiettivo che desidera.

3.2.4 Ciclo di controllo Practical Reasoning System (PRS)

Ciclo di controllo di un agente che applica il *ragionamento pratico*, con controllo per evitare di applicare dei piani che si sono rivelati non più ottimali in base all'evoluzione dell'ambiente:

- percepisce il mondo, aggiorna i propri beliefs;
- *delibera* per decidere quale intenzione perseguire;
- utilizza il ragionamento per obiettivi al fine di trovare un piano col quale raggiungere l'intenzione scelta;
- esegue il piano.

Il *Practical Reasoning System* è la prima architettura per agenti che utilizza il paradigma belief-desire-intention; inizialmente sviluppato da Michael Georgeff ed Amy Lansky allo Stanford Research Institute. La struttura di un piano di un Sistema a Ragionamento Pratico:

> Goal: post condizione. La percezione futura dell'ambiente dopo l'applicazione del piano
>
> Context: pre condizione. La percezione attuale dell'ambiente.

Body: come ottenere il goal se ci troviamo nel context.

- Al momento dell'avviamento, un agente PRS avrà una collezione di piani, con la struttura goal-context-body, e delle informazioni iniziali, sotto forma di beliefs, sul mondo circostante.

- Quando l'agente seleziona un obiettivo andrà ad inserirlo in uno stack (*intention stack*) che conterrà tutti gli obiettivi che devono ancora essere raggiunti.

- Nella libreria contenente i piani a disposizione cercherà quelli la cui post condizione (goal) corrisponda al obiettivo che vuole raggiungere.

- Tra quelli trovati escluderà quelli che non soddisfano le pre condizioni (context) in base a quelle che sono le informazioni che l'agente ha sull'ambiente. I piani rimanenti dopo questa scrematura formeranno le possibili *opzioni* che l'agente avrà a disposizione.

- Selezionare il piano tra quelli possibili corrisponde al processo di *deliberazione*, questa fase viene facilitata dall'utilizzo di meccanismi che attribuiscono dei valori numerici ai vari piani, una sorta di punteggio, cosicchè l'agente potrà selezionare il piano con il punteggio più elevato.

- Quindi il piano verrà eseguito.

3.2.5 Comunicazione tra agenti

Quando si vuole spiegare un'attività umana usiamo delle dichiarazioni che fanno suo di una psicologia popolare che interpretano il comportamento umano in termini di *credenze*, *volontà*, speranze, ecc... La comunicazione tra agenti nei sistemi a multi agente è generalmente basata sulla teoria degli atti linguistici (*speech-act teory*). La teoria degli atti linguistici basa le fondamenta sul principio secondo il quale il *linguaggio è azione*: un agente razionale produce un'espressione nel tentativo di cambiare lo stato dell'ambiente in cui lavora. Ciò che differenzia questa teoria da altre con azioni *non linguistiche* è il dominio dell'atto, ovvero la parte del mondo che l'agente vorrebbe modificare attraverso l'*atto*. Un atto linguistico si struttura

in una prima parte detta di *locuzione*, l'obiettivo, la *illocuzione* ed un'ultima parte, detta *perlocuzione*, in cui avremo l'effetto che l'atto linguistico produce sull'interlocutore. É possibile dividere gli atti linguistici in cinque classi: assertivi, direttivi, commissivi, espressivi e dichiarativi. In termini generali un atto linguistico può essere scomposto in due componenti: un verbo *performativo* ed un contenuto: variando la componente performativa e utilizzando lo stesso contenuto otterremo differenti atti linguistici. La comunicazione per atti linguistici è una comunicazione a livello di conoscenza, molto differente da una comunicazione per invocazione di metodo come avviene per esempio nei linguaggi orientati agli oggetti.

3.3 Programmazione orientata agli agenti

Abbiamo visto come il *Practical Reasoning System* crea l'architettura di un agente. Implementando l'agente andremo a definirne i piani ovvero il comportamento che l'avente dovrà tenere in determinate condizioni. Tuttavia gran parte delle operazioni che l'agente deve compiere gli vengono attribuite dall'architettura stessa senza il bisogno di ridefinirle. Il processo di *deliberazione* o di selezione di un piano non sono funzionalità che devono essere implementate per ogni singolo caso. Essendo sistemi reattivi, gli agenti dovranno essere in grado di rispondere, tempestivamente, agli stimoli percepiti ed agli eventi ricevuti.

```
1 cogitoErgoSum.
2
3 +cogitoErgoSum <- .print(''Hello World'').
```

Listing 3.1: Esempio di agente software in Jason

Questo piccolo esempio mostra un agente software. La sua unica conoscenza iniziale viene dalla riga numero 1, ovviamente non ha un senso compiuto non essendo questo l'obiettivo dell'esempio, ma ciò che l'agente conosce del mondo. Nella riga numero 3 invece è presente un piano: a sinistra della "freccia" abbiamo l'evento scatenante del piano mentre a destra l'azione che l'agente dovrà effettuare. Ovviamente le azioni potranno essere più di una. Quindi nel caso specifico: l'agente parte con la convinzione data dalla riga uno; tale convinzione porta a scatenare l'evento per il quale è previsto un piano nella riga tre con l'effetto di scrivere a video la frase: "Hello World".

Un linguaggio orientato agli agenti si comporrà di *Beliefs*, *Goals* e *Plans*.

3.3.1 Beliefs

L'agente per conoscere l'ambiente in cui si trova avrà bisogno di un modo per rappresentare tale conoscenza. Una *belief base*, ovvero un insieme di espressioni derivanti dalla programmazione logica, rappresenteranno la conoscenza del nostro agente. Queste espressioni sono sottoforma di predicati simbolici, per esempio nel listato qui sotto la parola *favorito* esprime una relazione tra *Marco* e *topolino*.

```
1   favorito(marco, topolino)
```

È possibile aggiungere delle note ai beliefs per aumentarne il contenuto informativo esplicito. Nel listato che segue è possibile intuire che gli impegni di *Davide* termineranno nel *2016*.

```
1   impegnato(davide)[finoa(2016)]
```

3.3.2 Goals

Se con i beliefs l'agente ha una visione del suo ambiente rappresentata da espressioni che ritiene veritiere, i goals sono invece la rappresentazione dell'ambiente che desidera nel suo "futuro". Questa peculiarità fa dei goals elementi non di primo rilievo al fine della modellazione di un agente.

In AgentSpeak i goals vengono divisi in due classi: gli obiettivi e gli obiettivi di prova. Quest ultimi sono generati al fine di ottenere informazioni disponibili nella belief base dell'agente.

```
1 !accendi_luce(camera)
2
3 ?estratto_conto(EC)
```

Nella prima riga l'obiettivo da raggiungere sarà l'accensione della luce nella camera e, se non già accesa, l'agente dovrà agire per realizzarlo. Nella terza riga l'agente interroga la *belief base* e *unirà* il risultato prodotto con la variabile EC.

3.3.3 Plans

Un piano è composto da tre distinte parti: l'evento scatenante, il contesto e il corpo.

```
+Goal : Context
   <- Body.
```

Listing 3.2: Esempio di piano in Jason

Nel listato qui sopra è presente un esempio di piano: sulla sinistra della prima riga abbiamo l'obiettivo da raggiungere mentre alla destra le condizioni che permettono l'applicazione del piano; per convalidare le condizioni l'agente si avvarrà delle sue conoscenze dell'ambiente, ovvero dei beliefs in suo possesso. Nella seconda riga il corpo che rappresenta le azioni da intraprendere al fine di raggiungere l'obiettivo.

```
+!leave(home)
   : not raining & not ~raining
   <- !location(window);
   ?curtain_type(Curtains);
   open(Curtains);
   ..
```

In questo piccolo esempio si può vedere brevemente un piano per uscire di casa: l'agente nel momento in cui deciderà di lasciare la casa controllerà tra le sue conoscenze dell'ambiente (esterno) la condizione particolare secondo cui non ha la certezza che non stia piovendo. In tal caso il piano si articola in un primo sotto-obiettivo che prevede il raggiungimento di una finestra; un secondo che si interroga sul tipo di tenda che oscura la finestra scelta; un ultimo di cui apre la tenda. Il piano ovviamente continua aggiungendo sotto-obiettivi.

Capitolo 4

JaCaMo su Raspberry Pi

Abbiamo visto come i *sistemi embedded* si prestano particolarmente a quelle applicazioni in ambienti fisici in cui è richiesta una buona reattività e continuità di lavoro. In questo quadro l'*Internet of Things* amplifica di non poco le potenzialità dei sistemi embedded aumentandone allo stesso tempo le problematiche relative alla coesistenza di sistemi eterogenei.
Successivamente con gli *agenti* è stato introdotto un approccio diverso alla modellazione del nostro ambiente. Attraverso agli agenti autonomi con un comportamento secondo l'architettura BDI si è visto come un agente assuma comportamenti autonomi, sia reattivo, tenda a lavorare in continuità temporale e gestisca le comunicazioni in un modo più sociale.

Date le premesse si vuole ora esplorare una possibile applicazione della *programmazione orientata agli agenti* direttamente sui *sistemi embedded* utilizzando nello specifico la piattaforma Raspberry Pi (Model B) come sistema embedded e JaCaMo come framework per lo sviluppo degli agenti. In questo capitolo si farà riferimento alle pubblicazioni [13][2].

4.1 Modellazione del sistema embedded con un framework ad agenti

I sistemi embedded vengono sviluppati per essere integrati in un ambiente e interagire con esso. Ricoprono uno specifico e preciso scopo in tale ambiente e dovranno essere in grado di rispondere in maniera tempestiva alle

sollecitazioni percepite. L'Internet of Things permette ai sistemi embedded di condividere i propri dati e sfruttare funzionalità condivise in rete.

I linguaggi di programmazione, di alto e basso livello, che si sono affermati oggi negli ambiti industriali, commerciali e privati non intercettano a livello di architettura problematiche che, anche se previste in tutti i linguaggi, nei sistemi embedded emergono in modo naturale e prepotentemente richiedono un'attenzione particolare. Basti pensare ai problemi relativi alla concorrenza tra processi o sottoprocessi: i linguaggi di programmazione mettono a disposizione strumenti come *semafori*, variabili per il mutuo accesso a risorse, blocchi atomici. L'implementazione di questi strumenti risulta non banale e occorre avere una sensibilità che vada oltre la buona conoscenza del linguaggio di programmazione ma che sconfini anche nelle risorse hardware in cui l'applicativo dovrà girare e tutti i livelli intermedi fino a quello applicativo (librerie, file system, OS kernel, driver periferiche).

Nei sistemi embedded l'accesso alle periferiche, come sensori e attuatori, è generalmente essenziale al buon funzionamento del sistema stesso. Questo rende necessario che la condivisione delle risorse avvenga nel miglior modo possibile e, come sempre quando si tratta di *specific purpose systems*, nel minor tempo possibile, garantendo bassi tempi di latenza nell'accesso a tutti i richiedenti. Occorre quindi che il linguaggio di programmazione scelto e tutto il framework a suo supporto organizzi un'architettura che garantisca una gestione ottimale degli aspetti visti.

L'agente, come concetto astratto è un'entità reattiva e autonoma e si adatta particolarmente bene a **incapsulare il controllo** di un sistema embedded rispettando le prerogative del sistema stesso. L'architettura BDI fornisce un ulteriore vantaggio all'agente che dovrà percepire l'ambiente e agire di conseguenza, il processo ciclico, *beliefs-desires-intentions*, infatti permette un continuo aggiornamento delle proprie conoscenze e una conseguente scelta del piano migliore da portare avanti in base ai propri obiettivi iniziali, generati da eventi o modifiche percepire nell'ambiente, oppure ricevuti attraverso comunicazione (da altri agenti).

Se la parte di controllo viene bene "interpretata" dall'agente, la modellazione del mondo fisico sarà affidata agli **artefatti** (che vedremo nel prossimo paragrafo). Implementare un agente per utilizzare un attuatore o un sensore, anche se possibile, appare uno spreco di risorse. L'architettura di un agente autonomo offre molte funzionalità rispetto a quelle estremamente limitate richieste da una periferica: un *artefatto* sarà quindi molto più

simile a un *oggetto*. L'*Object Oriented Programming* implementando meccanismi di incapsulamento, eredità e polimorfismo si presta egregiamente alla modellazione dei componenti presenti nel mondo fisico.

Avremo quindi un agente che percepisce gli input del sistema embedded in termini di proprità osservabili degli artefatti e dei loro cambiamenti. L'agente potrà agire sull'ambiente in cui opera il sistema embedded in termini di azioni fornite dagli artefatti.

4.2 JaCaMo

JaCaMo è un framework per la programmazione orientata agli agenti che permette lo sviluppo di sistemi multi-agente. Questo framework tiene conto di tre livelli di astrazione: quello degli agenti, l'ambiente e quello organizzativo. Ogni livello è realizzato grazie all'integrazione di altrettante tecnologie basate sugli agenti:

- Jason
  interprete di AgentSpeak per l'implementazione degli agenti.

- CArtAgO
  Common ARTifact infrastructure for AGents Open environments, piattaforma basata sugli *artefatti* per sviluppare l'infrastruttura di gestione del livello organizzativo.

- MOISE
  modello organizzativo per sistemi multi-agente basato su varie entità: gruppo, ruolo, missione, obiettivo, schema sociale e norma.

Un sistema multi-agente in JaCaMo sarà quindi un insieme di agenti autonomi programmati in Jason che lavoreranno in un ambiente condiviso, programmato in CArtAgO, il tutto organizzato tramite la programmazione in MOISE.

Gli agenti sono principalmente ispirati all'architettura BDI, ovvero composti da un insieme di *beliefs*, che rappresentano la conoscenza del mondo in cui si trovano; un insieme di *goals*, che corrispondono alle mansioni che un agente può effettuare; un insieme di *plans*, ovvero il *know-how* per raggiungere gli obiettivi, i *plans* possono essere innescati dagli *eventi*.

L'ambiente, basato su CArtAgO ed un meta-modello ad agenti ed *artefatti*, è composto da uno o più *workspaces*, ognuno dei quali contiene un insieme dinamico di *artefatti*. Un *artefatto* è il componente fondamentale che definisce la struttura ed il comportamento dell'ambiente, mette a disposizione un insieme di operazioni e proprietà osservabili che definiscono l'interfaccia dell'artefatto stesso. L'agente può osservare e operare sull'artefatto grazie a l'interfaccia disponibile. Oltre ai *workspaces* e gli *artefatti* il livello di astrazione dell'ambiente mette a disposizione l'entità *manual*: usata per rappresentare la descrizione delle funzionalità messe a disposizione da un artefatto.

Il livello relativo alla organizzazione, basato su MOISE, si divide in una parte strutturale composta da *group* e *role* ed una funzionale da *social scheme*(per decomporre la struttura dell'organizzazione in sub-goals), *goal* e *mission* (insieme di sub-goals). Infine l'entità *norm* lega i *roles* alle *missions* vincolando il comportamento dell'agente.

4.2.1 Interazioni tra livelli

- A-E: Mappatura agente-ambiente
  L'integrazione tra il livello dell'agente e quello dell'ambiente è dato dalla mappatura delle azioni esterne di un agente alle operazioni e le proprietà osservabili di un artefatto. Gli eventi vengono mappati nelle percezioni dell'agente aggiornandone la *belief base* e producendo eventi. Ogni azione che l'agente effettua sarà quindi messa a disposizione da almeno un artefatto dell'ambiente, nel caso più artefatti avessero la medesima azione tra le proprie interfacce allora l'agente dovrebbe contestualizzare esplicitamente l'artefatto di riferimento. L'agente non dovrà preoccuparsi di aggiornare le sue conoscenze dell'ambiente, ovvero non dovrà interrogare gli artefatti per conoscerne il loro stato aggiornato, perché nel momento che una proprietà osservabile di un artefatto si aggiorna l'agente riceverà dinamicamente l'aggiornamento nella *belief base*. Sarà quindi possibile programmare l'agente per reagire a eventuali cambi delle proprietà osservabili degli artefatti.

- O-E: Mappatura organizzazione-ambiente
  Infrastruttura organizzativa viene pensata come parte dell'ambiente in cui sono posti gli agenti. Le diverse entità volte a gestire lo stato dell'organizzazione in termini di *groups*, *social scheme* e *norms* sono

astratte attraverso artefatti organizzativi ed incapsulando ed attuando il comportamento dell'organizzazione secondo le specifiche sopra discusse. Gli agenti potranno utilizzare le azioni messe a disposizione dagli artefatti organizzativi per prendere parte ad una organizzazione(adottando ruoli, impegnandosi in missioni, comunicando all'organizzazione il raggiungimento di *social goal*). L'interazione tra agenti e organizzazione evita di ricorrere a primitive specifiche e meccanismi per la gestione dell'organizzazione.

- A-O: Mappatura agente-organizzazione
  I goals definiti nel livello dell'organizzazione sono mappati nei goals individuali di un agente che può decidere se adottarli. Dal punto di vista del livello organizzativo avrò un artefatto corrispondente, mentre dal punto di vista dell'agente i goals *imposti* saranno percepiti e se l'agente decide di adottarli verranno creati i corrispettivi goals e, rispettando il principio di autonomia, l'agente potrà decidere come realizzarli adottando eventualmente altri goals individuali.

4.3 Raspberry Pi

Raspberry Pi è un computer dalle dimensioni ridotte all'essenziale (85mm x 56mm x 17mm) che oltre alle tipiche interfacce che ne consentono un uso tradizionale (usb, vga, hdmi) mette a disposizione alcuni strumenti che permettono di interagire con l'ambiente esterno. Raspberry Pi Foundation, l'organizzazione che ha realizzato questo dispositivo, è una fondazione il cui scopo è rendere accessibile ad adulti e bambini un istruzione nel campo dell'informatica.

4.3.1 Hardware

Attualmente (2014) sono presenti sul mercato tre modelli di Raspberry Pi che differiscono lievemente le une dalle altre:

Punti in comune: SoC, HDMI, porte usb (1,2,4), pins expansion header, video composito, audio, GPIO (17, 17, 26), CSI-2, DSI, dimensioni

- Model A
- Model A+

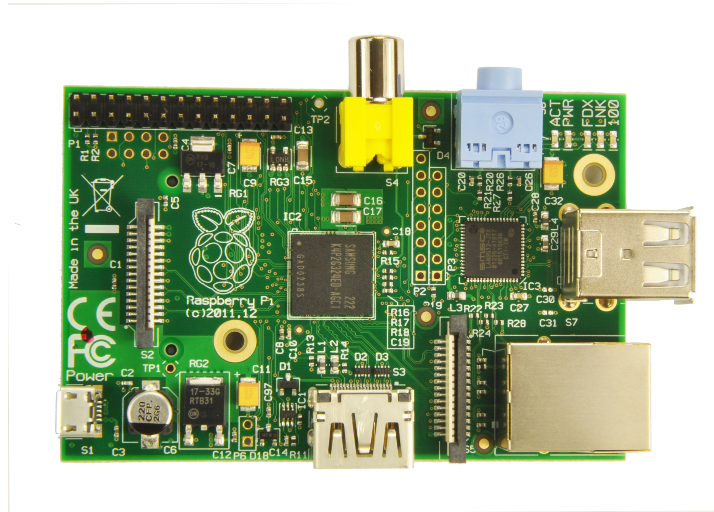

Figura 4.1: Raspberry Pi Model B

- Model B
- Model B+

Tutti i modelli sono dotati di un unico componente (BCM2835) è per questo che è possibile parlare di System-On-Chip, ovvero di Sistema su un singolo chip. Nel chip in questione è presente la CPU (ARM1176JZF-S), la GPU, la SDRAM (256MB o 512MB) ed un DSP (processore di segnale digitale ad uso esclusivo del sistema sono essendo disponibili librerie).

Sebbene tra il Model A ed il Model B le differenze più significative siano la potenza (1,5W, 3,5W, rispettivamente), il numero di porte usb e la presenza di un'interfaccia Ethernet, il Model B+ presenta sostanziali differenze dalle altre due (nonostante il nome): un socket per schede microSD, 26 GPIO, un'uscita video-audio composita (jack 3.5mm), 4 porte usb.

4.3.2 General Purpose Input/Output

Una caratteristica fondamentale di questo dispositivo è la disponibilità di un interfaccia programmabile che permette di ”sentire” il mondo esterno.

Questa interfaccia consiste in una serie di pin (26 nei modelli A e B, 40 nel modello B+), anche se non tutti effettivamente utilizzabili come input o

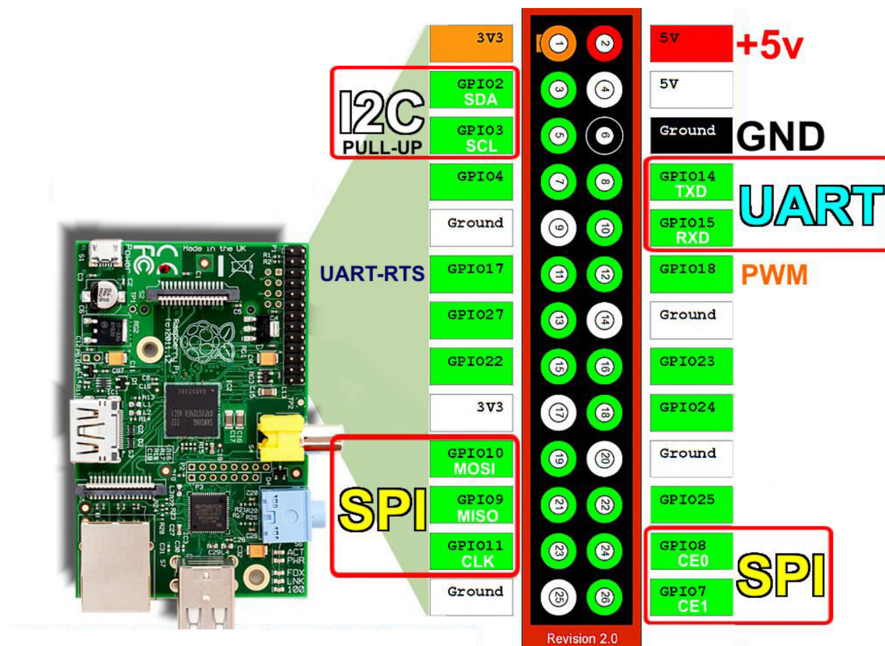

Figura 4.2: Pinout GPIO (Model A/B)

output, che permettono il collegamento delle periferiche con la scheda comunicando quasi direttamente con i registri del processore facendone tuttavia una parte piuttosto delicata del sistema.

L'interfaccia GPIO è esclusivamente digitale, questo limita relativamente all'utilizzo di periferiche con comunicazioni digitali escludendo una buona fetta di sensori ed attuatori analogici che, attraverso opportuni convertitori o il collegamento di microcontrollori (come Arduino), possono essere collegati solo indirettamente.

4.3.3 Pi4J

Tra le librerie a disposizione per l'utilizzo dell'interfaccia GPIO su Raspberry Pi le Pi4J sono probabilmente quelle più complete e performanti. Mettono a disposizione delle API orientate agli oggetti. Si sono affermate nella comunità Java principalmente per il loro alto grado di astrazione rispetto le librerie native.

Pi4J implementa librerie di basso livello (come è possibile vedere in figura 4.3.3) mutuate da WiringPi, un progetto che ha portato la gestione dell'interfaccia GPIO come si può trovare su Arduino. Questo progetto è riuscito ad avvicinare tanti sviluppatori alla piattaforma Raspberry Pi grazie alla familiarità che avevano già acquisito con l'utilizzo di Arduino.

4.4 JaCaMo su Raspberry Pi

Si vuole implementare un semplice esempio per poter testare la fattibilità di un sistema modellato attraverso JaCaMo su Raspberry Pi. L'ambiente in cui verrà inserito il sistema embedded prevederà un sensore e un attuatore, rispettivamente un bottone e un led. Per modellare l'ambiente come visto nei paragrafi precedenti si è scelto di usare gli *artefatti* JaCaMo. Avremo quindi un artefatto che ci fornirà le funzioni del bottone e uno per il Led.

4.4.1 Configurazione RPi

Sul sistema embedded scelto è stato installato il sistema operativo Raspbian: un sistema operativo gratuito basato su una distribuzione Linux, Debian, e ottimizzata per l'hardware del Raspberry Pi (RPi). In relazione al carico di lavoro (JVM) che la piattaforma dovrà sostenere, per motivi legati alle basse prestazioni è stato scelto di utilizzare RPi con comandi da terminale remoto, evitando quindi di installare gli strumenti per sviluppare direttamente l'applicativo di prova. Occorrerà:

- Installare Oracle Java ME Embedded 8.1 for Raspberry Pi Model B (ARM11/Linux)[10].
- Generare le librerie *libdio.so* e *dio.jar*

```
1 hg clone http://hg.openjdk.java.net/dio/dev
2 cd dev
3 export PI_TOOLS=<path to raspberry pi toolchain>
4 export JAVA_HOME=<path to JDK8>
5 make
```

- Ora sarà possibile copiare la libreria dio.jar sul computer dove svilupperemo il sistema software.

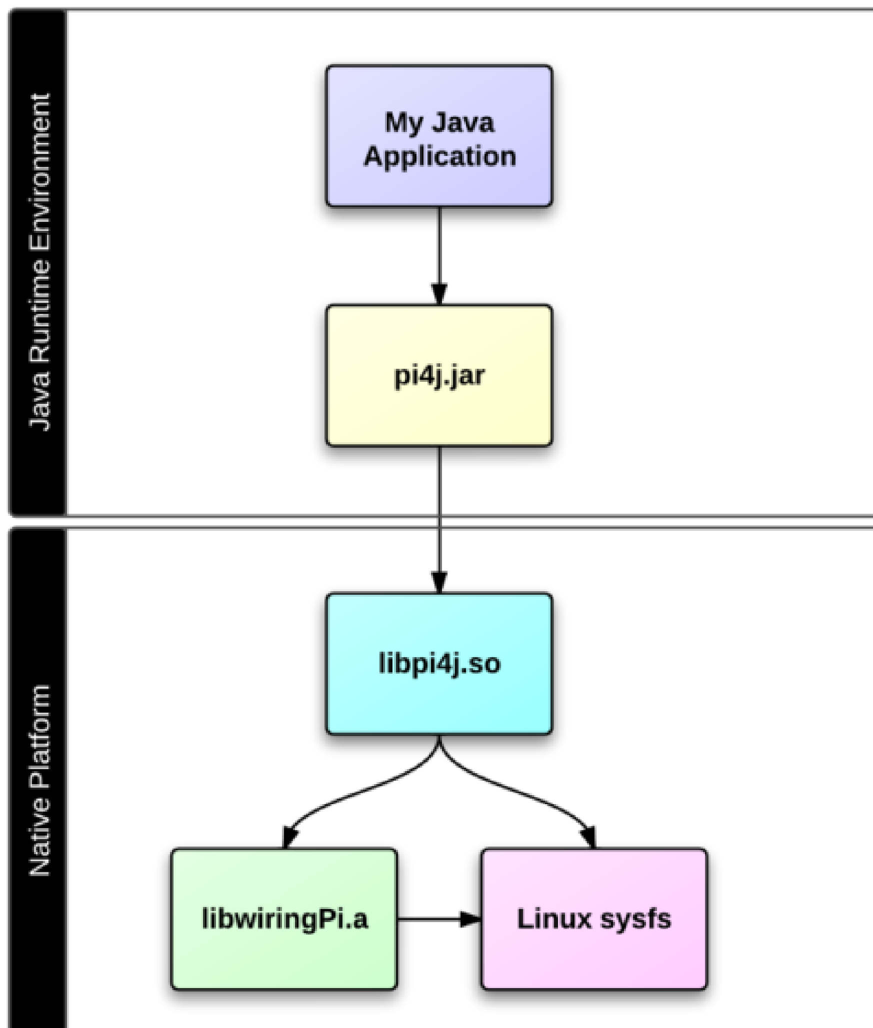

Figura 4.3: Dipendenze Pi4J e WiringPi

- Realizzare il circuito. Scelgo i pin per il bottone e il led e li collego alla GPIO.
- Dopo aver realizzato il sistema software e configurato le piattaforme (JaCaMo e Raspberry Pi) occorrerà esportare una versione del sistema software per essere eseguita sulla piattaforma di destinazione.

4.4.2 Configurazione JaCaMo

Come già detto JaCaMo presenta tre livelli: uno relativo agli agenti, uno all'ambiente e uno all'organizzazione.

Artefatti

L'ambiente in cui il nostro agente opererà è composto da un led ed un bottone. L'agente potrà interagire con questi due componenti attraverso altrettanti artefatti.

L'artefatto del led avrà una proprietà osservabile che indicherà lo stato di accensione o spegnimento.

```
1  package raspi;
2
3  import cartago.*;
4  import com.pi4j.io.gpio.*;
5
6  public class Led extends Artifact {
7
8     private GpioPinDigitalOutput pin;
9     void init(int pinNum) {
10       try {
11          GpioController gpio = GpioFactory.getInstance();
12          pin =
               gpio.provisionDigitalOutputPin(Config.pinMap[pinNum]);
13          this.defineObsProperty("state", "off");
14       } catch (Exception e) {
15          e.printStackTrace();
16       }
17    }
```

Listing 4.1: Artefatto led

L'agente potrà agire accendendo o spegnendo il led tramite due operazioni che l'artefatto mette a disposizione.

```java
@OPERATION
void switchOn() {
   try {
      pin.high();
      this.updateObsProperty("state", "on");
   } catch (Exception e) {
      e.printStackTrace();
   }
}

@OPERATION
void switchOff() {
   try {
      pin.low();
      this.updateObsProperty("state", "off");
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Listing 4.2: Artefatto Led - Operazioni

L'artefatto del bottone avrà uno stato che indicherà se il bottone risulta premuto o no.

```java
package raspi;

import cartago.*;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

import com.pi4j.io.gpio.GpioController;
import com.pi4j.io.gpio.GpioFactory;
import com.pi4j.io.gpio.GpioPinDigitalInput;
import com.pi4j.io.gpio.PinPullResistance;
import com.pi4j.io.gpio.event.GpioPinDigitalStateChangeEvent;
import com.pi4j.io.gpio.event.GpioPinListenerDigital;
```

```
14
15 public class Button extends Artifact {
16
17   private GpioPinDigitalInput pin;
18   private ButtonListener proc;
19
20   void init(int pinNum) {
21     this.defineObsProperty("pressed", false);
22     try {
23       GpioController gpio = GpioFactory.getInstance();
24       pin = gpio.provisionDigitalInputPin(Config.pinMap[pinNum],
25             PinPullResistance.PULL_DOWN);
26       proc = new ButtonListener();
27       execInternalOp("fetchEvents");
28       pin.addListener(proc);
29     } catch (Exception e) {
30       e.printStackTrace();
31     }
32   }
```

Listing 4.3: Artefatto bottone

Il bottone non permette nessuna operazione all'agente. Tuttavia implementerà delle operazioni ad uso dell'architettura, che potrà prelevare dall'artefatto l'evento generato.

```
1   @INTERNAL_OPERATION
2   void fetchEvents() {
3     while (true) {
4       await(proc);
5       Event ev = proc.getCurrentEventFetched();
6       if (ev instanceof ButtonPressed) {
7         this.updateObsProperty("pressed", true);
8       } else if (ev instanceof ButtonReleased) {
9         this.updateObsProperty("pressed", false);
10      }
11    }
12  }
```

Listing 4.4: Artefatto bottone - Operazioni

Alla pressione del bottone viene generato dalla libreria utilizzata un evento che viene gestito dall'artefatto. L'artefatto crea a sua volta un evento distinguendo dal tipo di pressione rilevata, bottone premuto o rilasciato. Questo nuovo evento verrà inserito in una lista di eventi che l'architettura dell'agente potrà utilizzare durante il suo ciclo di *sensing*.

```
1   class ButtonListener implements GpioPinListenerDigital,
       IBlockingCmd {
2
3      protected BlockingQueue<Event> eventQueue;
4      private Event fetched;
5
6      public ButtonListener() {
7         eventQueue = new ArrayBlockingQueue<Event>(50);
8      }
9
10     public void handleGpioPinDigitalStateChangeEvent(
11           GpioPinDigitalStateChangeEvent event) {
12        Event ev = null;
13        System.out.println(" --> GPIO PIN STATE CHANGE: " +
             event.getPin()
14              + " = " + event.getState());
15        if (event.getState().isHigh()) {
16           ev = new ButtonPressed();
17        } else {
18           ev = new ButtonReleased();
19        }
20        eventQueue.offer(ev);
21     }
22
23     public Event getCurrentEventFetched() {
24        return fetched;
25     }
26
27     public void exec() {
28        try {
29           fetched = eventQueue.take();
30        } catch (Exception ex) {
31        }
```

```
        }
    }

}
```

Listing 4.5: Artefatto bottone - Eventi

Agente

L'agente si servirà di due artefatti: uno per interagire con il led ed uno con il bottone. L'agente avrà un obiettivo iniziale fittizio, *start*.

```
/* Initial goals */
!start.
```

Listing 4.6: Agente - Goals

Il piano relativo all'obiettivo *start* gli permetterà di creare i due artefatti, testare il funzionamento del led e registrarsi per ricevere gli eventuali eventi generati dal bottone.

```
+!test  <-
   switchOn;
   .wait(100);
   switchOff.
+!start : true <-
   println("hello.");
   makeArtifact("led","raspi.Led",[4]);
   !test;
   makeArtifact("button","raspi.Button",[0],Button);
   focus(Button).
```

Listing 4.7: Agente - Plans

Quando il bottone viene premuto o rilasciato l'artefatto notificherà l'evento che si concretizzerà in una nuova belief all'agente. L'agente eseguirà le azioni previste dal suo piano: se la belief corrisponde a *pressed(true)* eseguirà l'operazione *switchOn* sul led; se la belief sarà *pressed(false)* l'operazione sarà *switchOff*.

```
+pressed(X) : X == true
```

```
2     <- switchOn.
3
4  +pressed(X) : X == false
5     <- switchOff.
```

Listing 4.8: Agente - Plans

Capitolo 5

Conclusioni

In questa tesi, dopo una panoramica su due tecnologie, i sistemi embedded e gli agenti, abbiamo visto una loro possibile integrazione. Al di là delle specifiche piattaforme prese in considerazione per un risvolto volutamente pragmatico, quello che emerge chiaramente è quanto queste tecnologie abbiano forti potenzialità se pensate in simbiosi tra di loro.

I sistemi embedded da sempre occupano un'importante ruolo all'interno dell'elettronica. I grandi produttori di componenti permettono di sperimentare e progettare prototipi grazie a delle schede modulari e a strumenti per lo sviluppo *ad hoc*. Tuttavia solo recentemente grazie a piattaforme estremamente economiche e versatili (per esempio Arduino) è possibile per chiunque avvicinarsi al mondo dei sistemi embedded. Se da un lato questa nuova linfa si concretizza in community di curiosi con le idee più stravaganti e originali a soluzione dei problemi di vita quotidiana nel mondo fisico, dall'altro significa che a "progettare" e implementare l'apertura automatica del cancello di casa, una mangiatoia in grado di riconoscere l'animale con cui interagisce e scegliere se somministrargli un alimento piuttosto di un altro, oppure ancora un sistema domotico in grado di controllare i consumi delle utenze e in caso di anomalie avvisare, non saranno necessariamente programmatori altamente specializzati in architetture *specific purpose* con esperienza e conoscenza di elettronica, meccanica e di programmazione concorrente, bensì di entusiasti e un po' sprovveduti. Un agente software sembra completare il lavoro di un sistema embedded: riesce ad occuparsi di aspetti estremamente importanti senza la necessità di lasciare tali delicati compiti al programmatore o hobbista.

I sistemi embedded e l'Internet of Things stanno cambiando anche alcuni equilibri economici: se fino a poco tempo fa l'ingegnere con una buona idea, con il classico garage in cui realizzare il primo prototipo, era costretto a una fase di contrattazione al fine di trovare uno o più investitori interessati alla propria idea, oggi il modello viene ribaltato anche grazie alla possibilità di ricorrere al *crowdfunding* e di proporre un "non-prodotto" da trasformare in prodotto vero e proprio a una platea globale che può decidere. Evitare la tagliola degli investitori "tradizionali", legati a logiche spesso conservatrici e poco propense al rischio, permette a tanti progetti di emergere dai cassetti e concretizzarsi. Un'architettura software che riesca ad occuparsi degli aspetti più complessi di una macchina che, anche se piccola, risulta tutt'altro che semplice, potrebbe aiutare a costruire un applicativo più robusto, modulare, stabile ed efficiente.

Con l'*Internet of Things* il sistema embedded entra in una rete di servizi ed oggetti dotati di funzionalità; gli oggetti avranno quindi la possibilità di interagire tra loro scambiando dati ed informazioni, un modellamento di questi sistemi attraverso la tecnologia ad agenti renderebbe più *reattivi* gli oggetti sgravandoli di problematiche relative all'architettura hardware e non inerenti ai loro comportamenti. Nella fattispecie è possibile utilizzare gli artefatti per modellare oggetti che offrono funzionalità, sensori ed attuatori, invece gli agenti per gestire il controllo del sistema embedded. In questo modo è possibile pensare al mondo che ci circonda come un insieme di artefatti disponibili ad essere utilizzati dagli agenti.

Data la sinergica interazione tra l'ambiente e l'agente, possibili sviluppi futuri Risulta molto difficile dire se i sistemi embedded integreranno la tecnologia ad agenti data la grande volatilità del mercato dell'Information Technology. Purtroppo infatti non sempre la migliore tecnologia è quella che sopravvive alla *selezione tecnologica*.

Bibliografia

[1] M. Barr. *Programming Embedded Systems : with C and GNU development tools*. O'Reilly, 2007.

[2] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. - multi-agent oriented programming with jacamo . *Science of Computer Programming* , 78(6):747 – 761, 2013. .

[3] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[4] N. R. C. Committee on Networked Systems of Embedded Computers. Embedded, everywhere: A research agenda for networked systems of embedded computers. The National Academies Press, 2001.

[5] Y.-H. Fan and J.-O. Wu. Middleware software for embedded systems. In *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, pages 61–65, March 2012.

[6] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, ECAI '96, pages 21–35, London, UK, UK, 1997. Springer-Verlag.

[7] J. A. Holgado-Terriza and J. Viúdez-Aivar. A flexible java framework for embedded systems. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 21–30, New York, NY, USA, 2009. ACM.

[8] E. A. Lee. Embedded software. volume 56 of *Advances in Computers*, pages 55 – 95. Elsevier, 2002.

[9] F. Mattern and C. Floerkemeier. From the internet of computers to the internet of things. In K. Sachs, I. Petrov, and P. Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *Lecture Notes in Computer Science*, pages 242–259. Springer Berlin Heidelberg, 2010.

[10] Oracle. Java platform, micro edition embedded (java me embedded) 8.1, 2014. `http://docs.oracle.com/javame/8.1/`.

[11] F. Palermo. Internet of things done wrong stifles innovation, 2014. http://www.informationweek.com/strategic-cio/executive-insights-and-innovation/internet-of-things-done-wrong-stifles-innovation/a/d-id/1279157.

[12] F. Rincon, J. Barba, F. Moya, F. Villanueva, D. Villa, J. Dondo, and J. Lopez. System-level middleware for embedded hardware and software communication. In *Intelligent Solutions in Embedded Systems, 2007 Fifth Workshop on*, pages 127–138, June 2007.

[13] A. Sorici, O. Boissier, G. Picard, and A. Santi. Exploiting the jacamo framework for realising an adaptive room governance application. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, - VMIL'11*, SPLASH '11 Workshops, pages 239–242, New York, NY, USA, 2011. ACM.

[14] D. Uckelmann, M. Harrison, and F. Michahelles. *Architecting the Internet of Things*. Springer Publishing Company, Incorporated, 1st edition, 2011.