ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Eterogeneità dei sistemi di Aggregate Programming: estensione del sistema ScaFi per l'uso di robot Thymio

Tesi di laurea in:
Objective Oriented Programming

Relatore
Chiar.mo Prof. Mirko Viroli

Candidato
Elvis Perlika

Correlatore
Dott. Gianluca Aguzzi

III Sessione di Laurea
Anno Accademico 2023-2024

Abstract

Max 2000 characters, strict.

Qualsiasi tecnologia sufficientemente avanzata è indistinguibile dalla magia.
Terza Legge di Arthur C. Clarke

Contents

Abstract **iii**

1 Introduction **1**

2 Background **3**
2.1 Paradigma dell'Aggregate Programming 3
2.1.1 Evoluzione dei paradigmi di programmazione 3
2.1.2 Sistemi collettivi ed eterogenei 5
2.1.3 Stato dell'arte 7
2.2 ScaFi 10
2.3 Macroswarm 14
2.4 Thymio e tdmclient 21
2.5 Aruco Tag 26

3 Analisi **29**
3.1 Obiettivi 29
3.2 Requisiti 29
3.3 Dominio 30

4 Design **33**
4.1 Architettura server Flask 33
4.2 File di configurazione 34

5 Implementazione **35**
5.1 Implementazione del server Flask 35
5.2 Esempi di Algoritmi AP applicati ai Robot Wave e Thymio nello stesso ambiente 35
5.3 Fancy formulas here 35

6 Conclusione **37**

39

Bibliography **39**

List of Figures

2.1 Architettura del mondo IoT 6
2.2 Droni in formazione in ambiente notturno 7
2.3 Architettura del framework ScaFi 11
2.4 Architettura di Macroswarm 16
2.5 Struttura di Macroswarm e moduli interni 17
2.6 Modalità Round Based 17
2.7 Modalità Long Standing 18
2.8 Simulazioni dei blocchi di movimento eseguita su Alchemist. 19
2.9 Simulazioni delle formazioni eseguite su Alchemist, in ordine: linea, formazione a V, cerchio. 21
2.10 Robot Thymio 22
2.11 Interprete vs Compilatore 24
2.12 Tdmclient workflow 25
2.13 Aruco markers identificati da un algoritmo di visione artificiale . . . 27

3.1 Robot Wave 30

4.1 Architettura del server Flask 34

List of Listings

2.1 Esempio di funzione pura in Scala 4
2.2 Esempio di funzione non pura in Scala 4
2.3 Interfaccia per la definizione di un campo di calcolo in Scala 13
2.4 Esempio di codice Aseba con eventi 24
2.5 Esempio di invio di comando (foo) ad un robot Thymio 25
2.6 Esempio di programma interno a Thymio per la ricezione di eventi (foo) 26
2.7 Esempio di transpile di un programma Python (print.py) in Aseba . 26
3.1 Classe AggregateIncarnation 31
3.2 Interfaccia Enviroment 31
3.3 Definizione dell'ambiente 31
listings/HelloWorld.java 35

Chapter 1

Introduction

Write your intro here.

Elvis Perlika: Add sidenotes in this way. They are named after the author of the thesis

You can use acronyms that your defined previously, such as Internet of Thing (IoT). If you use acronyms twice, they will be written in full only once (indeed, you can mention the IoT now without it being fully explained). In some cases, you may need a plural form of the acronym. For instance, that you are discussing Virtual Machines (VMs), you may need both VM and VMs.

Structure of the Thesis

Elvis Perlika: At the end, describe the structure of the paper

Chapter 2

Background

2.1 Paradigma dell'Aggregate Programming

2.1.1 Evoluzione dei paradigmi di programmazione

L'Objective Oriented Programming è un paradigma nel senso stretto del termine poiché rappresenta un modo di organizzare e rappresentare un mondo. Il paradigma in questione deve la sua potenza nella capacità di simulare entità reali ed è riassumibile con la frase *Everything is an Object*. È rilevante parlare di OOP in quanto il paradigma di programmazione funzionale, che è alla base di ScaFi, è un'estensione di esso. Il potere della programmazione ad Oggetti (OOP), come detto precedentemente, risiede nella capacità di simulare un mondo e permette di farlo grazie agli "oggetti", essi sono istanze di Classi, le quali a loro volta sono strutture dati astratte che permettono ad ogni loro istanza di avere uno stato (definito dai *fields*) e un comportamento (definito dai *methods*). I pilastri della programmazione ad oggetti sono l'incapsulamento, l'ereditarietà e il polimorfismo.

Il difetto della programmazione ad oggetti è che richiede un particolare impegno nel gestire il sistema da realizzare all'aumentare della sua complessità. Altri paradigmi, come la programmazione funzionale, possono essere più adatti a determinati problemi.

Nel caso della OOP abbiamo riassunto il paradigma con la frase *"Everything is an Object"*, per la programmazione funzionale possiamo riassumerla con *"Everything is a Function"*. Quando si parla di funzioni nel ambito della Functional

Programming (FP) si intendo **funzioni pure** cioè senza effetti collaterali. Per effetti collaterali si intende che la funzione fa altro oltre a restituire un risultato. Un paio di esempi, presi da `Functional Programming in Scala`, sono:

- Modifica di una variabile
- Modifica del campo di un oggetto
- Leggere da o scrivere su un file
- "Disegnare" sullo schermo

Si potrebbe pensare che con l'uso della FP si possano costruire solo programmi semplice, nella realtà non c'é alcuna limitazione sulla complessità del software da costruire poiché il paradigma della FP esprime un nuovo modo di pensare e scrivere il codice.

Nel dettaglio, per **funzione pura** si intende una funzione $f : A \to B$, (una funzione che prende un input di tipo A e restituisce un output di tipo B) che mette in relazione ogni elemento di A con esattamente un valore di B. Qualsiasi altra operazione che non sia utile a calcolare $f(a) = b$ con $a \in A$ e $b \in B$ deve essere intesa come effetto collaterale della funzione e quindi evitata se si vuole creare una funzione pura.

Un esempio di funzione pura, senza effetti collaterali, è la funzione di somma che prende in input due valori e ne restituisce la loro somma listing 2.1.

Listing 2.1: Esempio di funzione pura in Scala

```
1  def sum(a: Int, b: Int): Int = a + b
```

Un esempio di funzione non pura, con effetti collaterali, è la funzione che prende in input un valore e lo stampa a schermo listing 2.2.

Listing 2.2: Esempio di funzione non pura in Scala

```
1  def print(a: Int): Unit = println(a)
```

Formalmente si può definire una funzione pura con il concetto di Referential Transparency (RT):

> Una funzione f è Referentially Transparent se per ogni contesto C nel quale la funzione viene inserita, essa può essere sostituta dal risultato della stessa funzione f senza condizionare il risultato di C.

È proprio questa proprietà che permette ad un programma progettato con approccio funzionale di essere maggiormente scalabile e mantenibile.

2.1.2 Sistemi collettivi ed eterogenei

Per comprendere il potenziale del paradigma dell'Aggregate Programming (AP) è fondamentale definire quale tipo di problema si vuole risolvere. La crescita del mondo IoT ha portato alla presenza di numerosi dispositivi connessi tra loro di ogni tipo e caratterizzati da protocolli di comunicazione eterogenei. Il mondo dell'IoT in continua evoluzione promette di migliore la produzione, gli spazi di lavoro e la sanità attraverso la raccolta e l'analisi di grandi quantità di dati. È nato di conseguenza il bisogno di progettare sistemi resistenti e resilienti che permettessero di gestire queste grandi quantità di dispositivi. È necessario che questi sistemi siano capaci di: adattarsi a cambiamenti imprevedibili, permettere l'implementazione di nuovi comportamenti, permettere una capacità di rilevazione e attuazione nel ambiente in modo distribuito e coordinato. Allo stato attuale della tecnologia è difficile progettare sistemi di questo tipo poiché non esistono framework che permettano di gestire in modo semplice e intuitivo la complessità di questi sistemi.

Un'astrazione grafico del mondo IoT è presente in fig. 2.1. È possibile comprendere come ci sia uno strato contente i nodi (dispositivi)[1], uno strato che rappresenta la rete (internet) mediante la quale comunicano[2] ed uno strato contente il mondo Cloud dove vengono raccolte tutte le informazioni prodotte dai nodi della rete e computate attraverso applicazioni complesse per l'analisi, la pianificazione, l'ottimizzazione ed altri scopi.

Il problema di questa architettura è che inviare queste ingenti quantità di dati da un numero elevato di nodi può avere effetti negati sui costi, le disponibilità, la scalabilità e la latenza delle comunicazioni. Una delle soluzione trovate è quella di

[1] HMI: Human-Machine-Interaction.

[2] Prima di accedere alla rete internet le informazioni dei sensori vengono raccolte dai Edge Gateway che si occupano di instradare le informazioni.

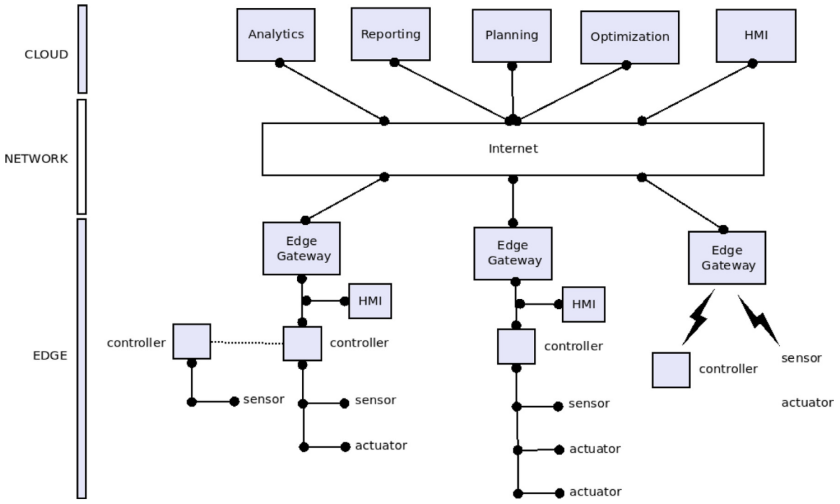

Figure 2.1: Architettura del mondo IoT

spostare la computazione più vicino possibile ai nodi della rete, in modo da ridurre la quantità di dati da inviare e la latenza delle comunicazioni. Questo approccio è chiamato *Edge Computing* non è sempre conveniente applicarlo, dipende dal contesto (non sarebbe adatto nel caso i nodi abbiamo specifiche tecniche, come memoria e capacità di calcolo, deboli).

Nella progettazione di sistemi per il controllo di sistemi aggregati[3] si sottolinea la necessità di meccanismi che rendano possibile ed intuitivo il riutilizzo e la capacità di composizione dei componenti così da poter costruire API[4] a più strati.

Ci sono diversi esempi di sistemi aggregati che necessitano di un controllo distribuito e coordinato, un esempio è il controllo di droni in formazione. In questo caso, ogni drone deve essere in grado di comunicare con i propri vicini per mantenere la formazione, evitare collisioni e raggiungere l'obiettivo comunefig. 2.2. Un altro esempio è il controllo di un gruppo di robot che devono esplorare un ambiente sconosciuto con lo scopo di raccogliere informazioni.

Per andare in contro a queste necessità nasce il paradigma dell'AP che da la nascita al concetto di *intelligenza collettiva*.

[3]Sistemi caratterizzati da un insieme di dispositivi che collaborano tra di loro per raggiungere un obiettivo comune.

[4]Application Programming Interface

Figure 2.2: Droni in formazione in ambiente notturno

2.1.3 Stato dell'arte

Nei capitoli precedenti si è esaminata l'evoluzione dal paradigma Objective Oriented Programming (OOP) a quello FP, la quale, ha permesso di gestire in modo più pratico progetti più complessi e semplificandone la manutenibilità. Una ulteriore, più specifica, evoluzione è quella portata dal paradigma dell'AP. Quest'ultimo mira a rendere la progettazione, manutenzione e testing nell'ambito del controllo di dispositivi hardware di larga-scala (anche conosciuti come **collective adaptive systems (CAS)**).

L'Aggregate Computing (AC) si basa sulla mappatura di un mondo cyber-fisico in un diverso modello logico. Da un punto di vista strutturale possiamo vedere un sistema aggregato come una rete di dispositivi capaci di computare ed equipaggiati con (o sui quali è possibile equipaggiare) sensori ed attuatori, che corrispondono ai nodi. I nodi sono connessi tra loro secondo una logica di vicinato, che può essere definita in base alla distanza fisica o alla comunicazione. Da un punto di vista comportamentale, invece, ogni nodo interpreta il programma aggregato solo in relazione al proprio contesto locale. Da un punto di vista delle interazioni, i nodi estraggono informazione dal proprio vicinato e propagano le proprie nello stesso contesto. Queste interazioni son ciò che permettono al contesto locale e globale di influenzarsi reciprocamente.

La tecnica dell'AP si basa su 3 principi fondamentali per la costruzione di sistemi robusti e resilienti:

- I dettagli implementativi dei sistemi hardware che si vuole andare a manipolare devono essere nascosti così da permettere ai programmatori di concentrarsi solo sulla logica di alto livello del sistema, in alcuni casi è possibile che questa astrazione delle specifiche di basso livello sia tale da poter essere pensato come uno spazio continuo, invece di un insieme di dispositivi separati. Per esempio, invece di immaginarci una stanza piena di sensori, la trattiamo come una area unificata definita da un flusso continuo di dati.

- Il programma manipola le strutture dati della rete di dispositivi sia in funzione della loro estensione spaziale che di quella temporale. Approccio particolare utile in sistemi in cui l'informazione cambia in base al luogo.

- Ogni dispositivo della rete esegue le operazioni necessarie in autonomia, coordinandosi con i dispositivi vicini attraverso meccanismi robusti e resilienti. In questo modo il sistema rimane fluido ed efficiente anche in situazioni anomale, ad esempio causate dal guasto di un qualche dispositivo.

L'approccio AP nella progettazione di sistemi è profondamente diverso classico approccio "dispositivo centrico". Nel secondo caso, ogni dispositivo della rete ha il compito di compiere tutte le operazioni richieste per il raggiungimento della soluzione e simultaneamente comunicare con gli altri dispositivi della rete. Questo approccio, seppur semplice, è poco scalabile e difficile da mantenere al crescere della complessità del sistema. Il nuovo paradigma, invece, prevede la scomposizione in componenti del programma assegnato ad ogni dispositivo, ogni componente esegue un compito specifico e comunica principalmente solo con i componenti dello stesso tipo presenti nel suo intorno per eseguire un servizio ed eventualmente con un altro componente presente nello stesso dispositivo al fine di realizzare una nuova soluzione inerente ad un'altro servizio. Le componenti sono una astrazioni dei *campi di calcolo*.

Il paradigma AP, il quale ha permesso la creazione di framework come ScaFi, Protelis e MacroSwarm, si basa sul concetto dei sopracitati *campi di calcolo*. Per *campo di calcolo* o *computationals field* si intende la mappatura degli elementi del dominio in cui si opera in un insieme di valori. I campi di calcolo vengono utilizzati per standardizzare le interazioni tra i dispositivi, permettendo di analizzare

e progettare sistemi distribuiti in modo più semplice e intuitivo. L'idea di campo prende ispirazione dai campi fisici, ad esempio quello magnetico. Ogni dispositivo della rete è considerato come un punto nello spazio ed il campo rappresenta un dato valore assegnato ad ognuno di questi punti. L'insieme di tutti i valori, che chiameremo *campi* o *fields* rappresentano lo stato di un sistema in un dato istante.

Questo approccio è generalmente utilizzato per un controllo prolungato dei dispositivi della rete sulla quale si vogliono eseguire task che necessitano di eseguire un gran numero di rilevazioni (ad esempio utilizzando sensori), computazioni ed attuazioni.

Ogni dispositivo del proprio sistema aggregato esegue una computazione in modo asincrono attraverso i **sense-compute-(inter)act rounds**, ogni round è composto da tre fasi concettuali:

1. **Aggiornamento del contesto (sense)**: ogni nodo memorizza il suo stato precedente, i dati ambientali restituiti da eventuali sensori e le più recenti informazioni ricevute dai nodi vicini.

2. **Esecuzione del Programma Aggregato (compute)**: il campo di ogni nodo viene computato in base al contesto locale e produce un valore di output detto *export* da condividere con il proprio vicinato ed un output utile per l'attuazione.

3. **Azione (inter-act)**: in questa fase il nodo effettua le seguenti azioni:

   (a) **Condivisione**: il nodo condivide il proprio *export* verso i nodi del suo vicinato in formato broadcast

   (b) **Controllo di attuatori**: l'output del nodo viene utilizzato per l'esecuzione di uno o più attuatori nel proprio ambiente

Posizionandoci in un livello più implementativo troviamo i seguenti costrutti per la manipolazione dei campi:

- **Functions**: funzioni

  $$b(e_1, \ldots, e_n)$$

  applicate agli argomenti $e_1, \ldots, e_n$. Possono essere sia funzioni matematiche, logiche o algoritmiche ma possono anche rappresentare sensori o attuatori.

- **Dynamics**:

$$rep(x \leftarrow v)s_1; \ldots; s_n$$

  rappresenta un una variabile di stato locale x, inizialmente inizializzata con il valore v e che può essere modificata dalle istruzioni $s_1, \ldots, s_n$. In questo modo si definisce un campo dinamico.

- **Interaction**:

$$nbr(s)$$

  rappresenta il vicinato di un dispositivo, ovvero l'insieme dei dispositivi con cui è possibile interagire.

- **Restrictions**:

$$\begin{array}{l} \text{if } e \\ \qquad s_1; \quad \ldots \quad s_n; \\ \text{else} \\ \qquad s'_1; \quad \ldots \quad s'_m; \end{array}$$

  permette di andare a definire sotto spazio del campo principiale in base ad una condizione e sul quale poi andare ad eseguire certe istruzioni invece di altre. È importante che le istruzioni riferite ad un certo sotto spazio non abbiano effetti su altri sotto spazi.

Come viene eseguito un modello di AP?

Questo paradigma, proprio come quelli precedenti, non è privo di difetti. Implementazioni di questo tipo sono difficili da integrare in sistemi programmati in modo più tradizionale, inoltre, dal punto di vista della programmazione sono mancanti i meccanismi per la gestione della concorrenza e della computazione dinamica dei campi. Il framework ScaFi, che vedremo nel prossimo capitolo, cerca di risolvere questi problemi.

2.2 ScaFi

ScaFi è un toolkit open-source scritto in linguaggio Scala per l'AC per la progettazione ad alto livello di sistemi aggregati. Il termine ScaFi deriva dalla unione di

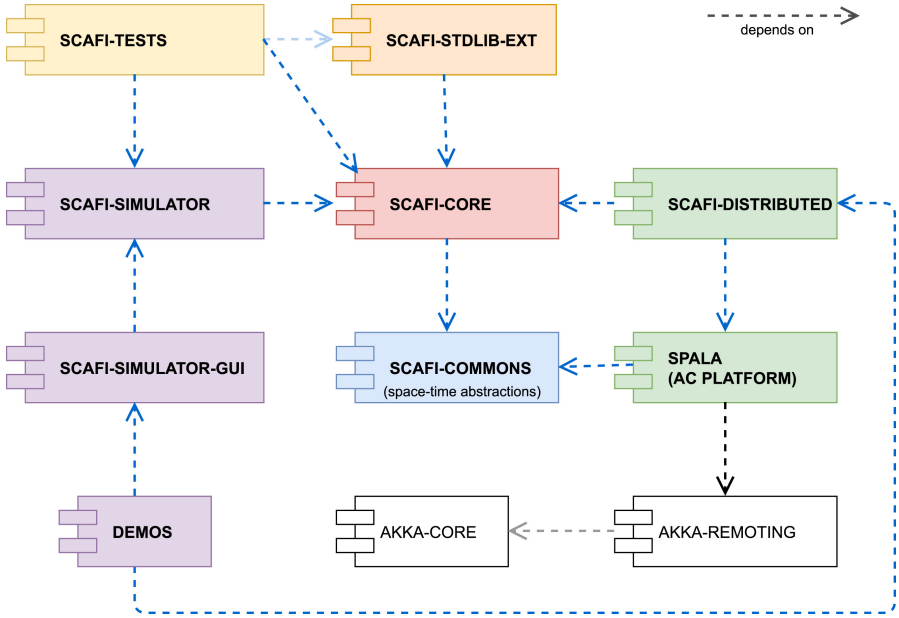

Figure 2.3: Architettura del framework ScaFi

Scala e Fields. Scala è un linguaggio java-like moderno e flessibile progettato per unificare il paradigma OOP e quello FP. È stato progettato per essere coinciso e facilmente leggibile, inoltre si basa sulla Java Virtual Machine, la quale permette l'utilizzo delle librerie Java. L'utilizzo della JVM e la flessibilità del linguaggio sono state le caratteristiche chiave per il raggiungimento della popolarità di questo linguaggio. Il termine Field invece deriva dai *field calculus* (campi di calcolo) di cui abbiamo parlato nei capitoli precedenti.

L'Architettura di ScaFi è riassumibile nella immagine fig. 2.3.

Ispezionando la figura, troviamo:

- **scafi-commons**: fornisce le entità di base, ad esempio astrazioni temporali e spaziali
- **scafi-core**: rappresenta il core del framework fornendo il DSL[5] per la progettazione di sistemi aggregati; con il supporto di librerie standard.

[5]Domain Specific Language: linguaggio di programmazione progettato specificamente per risolvere problemi o esprimere concetti di un dominio ristretto. Il DSL di ScaFi (libreria di programmazione) è quindi un linguaggio specifico per creare e valutare programmi di tipo aggregato, utilizzando una sintassi e una semantica pensate appositamente per questo tipo di calcoli e simulazioni.

- **scafi-stdlib-ext**: librerie extra
- **scafi-simulator**: fornisce un supporto per la simulazione dei sistemi di AP sviluppati
- **scafi-simulator-GUI**: fornisce un'interfaccia grafica per la simulazione
- **spala**: ("spatial scala") si tratta di un *middleware* che permette di eseguire programmi AC basati sugli attori ed è indipendente dal DSL di ScaFi
- **scafi-distributed**: layer integrativo per l'utilizzo di **spala** in ScaFi

Perché usare Scala?

Il motivo per cui Scala è un linguaggio particolarmente adatto per la programmazione di sistemi di AC è dovuto proprio alla sua flessibilità, essa permette di lavorare sui campi di calcolo, che necessitano di essere supportati dalle seguenti proprietà, in modo molto naturale:

- una sintassi coincisa per definire funzioni sui campi, le quali in Scala sono le espressioni standard
- meccanismi di controllo sul *quando* e *come* le espressioni vengono valutate
- capacità di calcolare lo stato di un campo basandosi sulle informazioni più recenti dei nodi presenti nel suo intorno

In Scala ogni valore è un oggetto ed i comportamenti vengono definiti attraverso i metodi. Per implementare un campo di calcolo in Scala andremo a definire gli operato dei campi attraverso la definizione di metodi che verranno interpretati dagli oggetti sui quali sono stati chiamati. Gli oggetti, in questo modo, possono essere visti come macchine virtuali in locale per l'esecuzione degli operatori di campo.

Di seguito l'interfaccia fondamentale per la definizione di un campo di calcolo in Scala (listing 2.3):

Gli elementi principali di questa interfaccia sono:

Listing 2.3: Interfaccia per la definizione di un campo di calcolo in Scala

```
trait Constructs {
  def rep[A](init: => A)(fun: A => A): A
  def nbr[A](expr: => A): A
  def foldhood[A](init: => A)(acc: (A, A) => A)(expr: => A): A
  def aggregate[A](f: => A): A

  def branch[A](cond: => Boolean)(th: => A)(el: => A)
  def share[A](init: => A)(fun: (A, () => A) => A): A

  def mid: ID
  def sense[A](sensorName: String): A
  def nbrvar[A](name: CNAME): A
}
```

- `rep(init)(f)` cattura l'evoluzione di stato di un valore inizializzato ad `init` e aggiornato ad ogni round con la funzione `f`
- `nbr(expr)`: rileva le comunicazioni relative al valore computato dalla espressione `expr` dei nodi vicini
- `foldhood(init)(acc)(expr)`: si occupa, partendo da un valore iniziale `init`, di accumulare i valori computati dai nodi vicini attraverso la funzione di accumulazione `acc` e settare i valori computati dalla espressione `expr` verso i propri vicini
- `branch(cond)(th)(el)`: cattura una partizione (spazio-temporale) del dominio in base alla condizione `cond`, se questa è rispettata esegue l'espressione `th` altrimenti `el`
- `mid`: provvede ad identificare un certo nodo
- `sense(sensorName)`: permette di accedere (ad alto livello) al sensore locale `sensorName`
- `nbrvar(sensorName)`: permette di accedere (ad alto livello) ai sensori dei nodi vicini, simile ad `nbr` ma valido per sensori forniti dalla piattaforma, questi sensori forniscono un valore per ciascun vicino

In ScaFi, i campi non sono reificati esplicitamente ma esistono solo a livello semantico, questo significa che un'espressione Scala non viene gestita come un'espressione di campo finché non viene passata all'interprete ScaFi.

ScaFi fornisce alcuni operatori, detti anche *blocchi*, di alto livello già implementati tipici dei sistemi aggregati:

- **Sparse choice (definizione di un leader)**: Blocco

  ```
  S(grain: Double):Boolean
  ```

  produce un campo booleano auto organizzato che definisce a `true` un insieme sparso di dispositivi distanziati tra di loro di almeno `grain`

- **Gradient-cast (propagazione distribuita)**: Blocco

  ```
  G[T](source: Boolean, value: T, acc: T=>T): T
  ```

  utilizzato per propagare un valore `value` da una sorgente `source` a tutti i nodi della rete, in ordine di distanza crescente, seguendo un gradiente che modifica i valori di ogni nodo secondo una funzione di accumulo `acc`

- **Collect-cast (collezione distribuita)**: Blocco

  ```
  C[T](sink: Boolean, value: T, acc(T,T)=>T): T
  ```

  utilizzata per riassumere le informazioni distribuite in un unico dispositivo `sink`, i valori `values` prodotti dai dispositivi vengono "raccolti" dal gradiente che si muove in direzione di `sink` e vengono accumulati secondo la funzione di accumulo `acc`

2.3 Macroswarm

Macroswarm è un framework per la programmazione di sistemi di robotica distribuita basato su ScaFi e che estende lo stesso ScaFi. Macroswarm nasce con l'avanzare delle tecnologie ed il crescente interesse verso il controllo di flotte di dispositivi come droni, robot, veicoli o folle di persone definite da dispositivi portatili/indossabili in modo. Si è voluto, quindi, creare un framework che permettesse di programmare sistemi distribuiti di robot in modo semplice ed intuitivo

secondo degli *swarm behavior*[6], permettendo di concentrarsi solo sulla logica di alto livello del sistema. Questo framework è stato scelto per il progetto in esamina. Il paradigma AP e di conseguenza ScaFi sono state scelte naturali per la progettazione di tale framework (fig. 2.4).

Fondamentalmente, Macroswarm è un programma ScaFi il quale esegue un certo programma aggregato su una certa piattaforma. L'idea è quella di creare una rappresentazione astratta di una flotta di dispositivi attraverso campi di calcolo, tipicamente vettori. All'intero del Framework troviamo le API per progettazione dei sistemi aggregati su sistemi fisici (i rettangoli bianchi in fig. 2.5 rappresentano i moduli principali).

È rilevante notare che in Macroswarm la computazione della attuazione da eseguire su un certo dispositivo e la reale attuazione nel mondo reale sono disaccoppiate. Il motivo di questa scelta è dovuto al fatto che ad ogni roundsection 2.1.3 il programma aggregato può variare un qualche parametro per completare il task. È, comunque, possibile scegliere quale modalità di computazione-attuazione scegliere poiché, in base al contesto, una può essere più adatta dell'altra. Le modalità sono:

- **round-based**(fig. 2.6): è possibile eseguire l'attuazione al termine del prossimo round
- **long-standing**(fig. 2.7): l'attuazione che si vuole eseguire è valida finché non viene revisionata o ritirata

Questo modello di attuazione è modellato come una funzione purasection 2.1.1 quindi priva di effetti collaterali. Se ci si trova in un contesto in cui l'esecuzione dell'attuazione richiede un certo tempo allora il progettista del sistema aggregato può andare a modellare il sistema in modo tale da tenere in considerazione questo aspetto.

Come è possibile vedere in fig. 2.5 il framework è composto da diversi moduli che a loro volta contengono diversi blocchi, ognuno con un compito specifico, solo alcuni di questi blocchi sono sono stati utilizzati per la realizzazione del progetto e saranno descritti nei prossimi paragrafi.

[6]Comportamenti di sciame/flotta

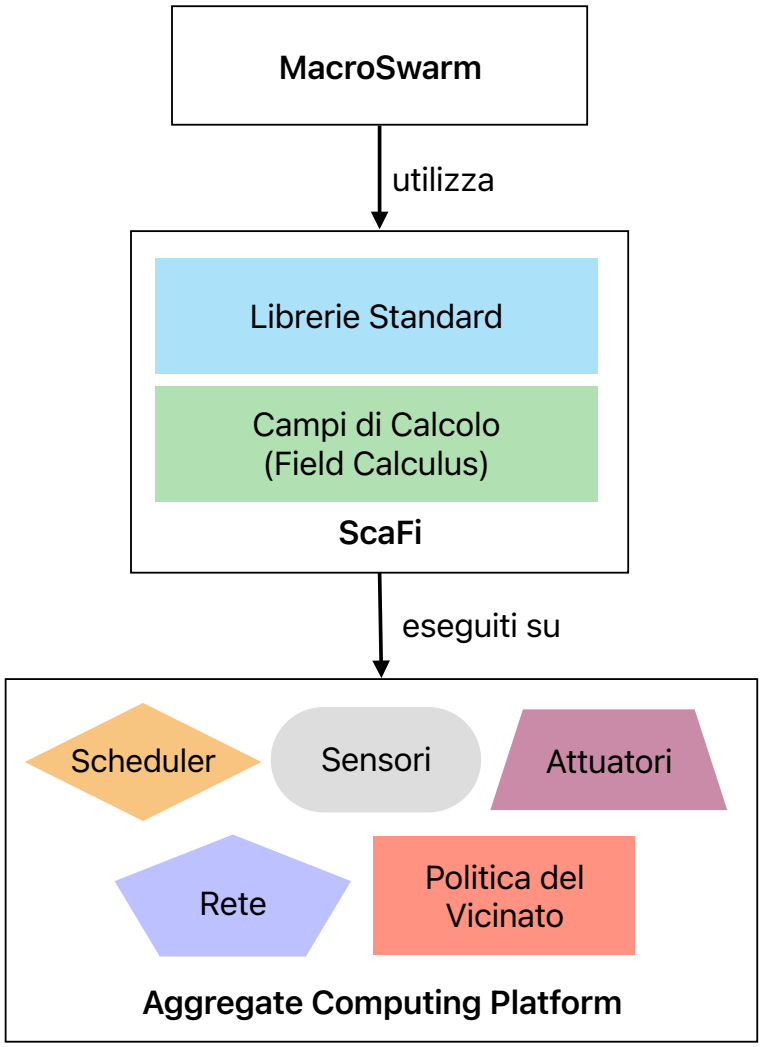

Figure 2.4: Architettura di Macroswarm

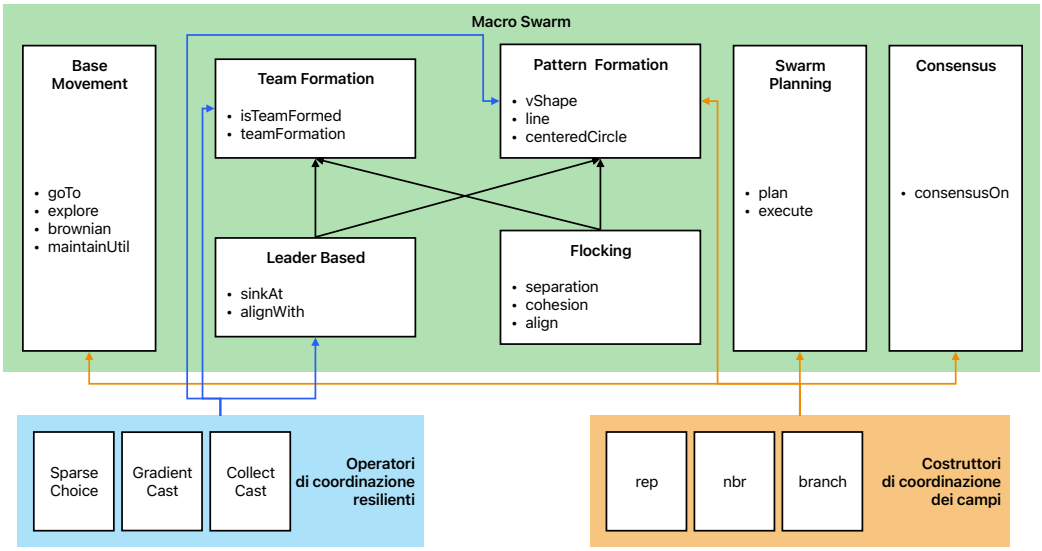

Figure 2.5: Struttura di Macroswarm e moduli interni

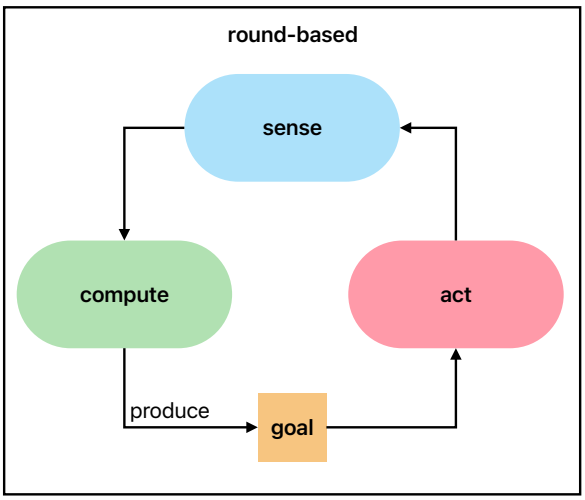

Figure 2.6: Modalità Round Based

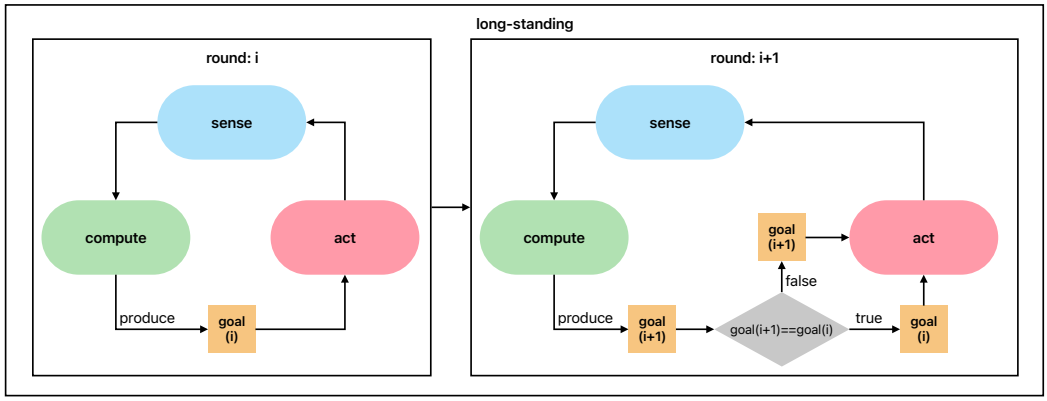

Figure 2.7: Modalità Long Standing

Movements blocks Questi blocchi si occupano di controllare i movimenti di ogni agente (dispositivo) della flotta. Il Movimento più semplice è descritto da un vettore `Vector(x,y,z)` che rappresenta la direzione e la velocità del movimento, ad esempio

```
1 Vector(3.6, 0, 0)
```

rappresenta un movimento di $3.6\frac{m}{s}$ lungo l'asse x. Una volta definito il blocco di movimento per un certo agente, i valori x, y, z devono essere accuratamente mappati sui motori del dispositivo per ottenere il movimento desiderato. Carattere di fondamentale importanza quando si lavora su un sistema eterogeneo e quindi formato da dispositivi differenti equipaggiati da motori differenti.

Un'altro blocco "semplice" è `brownian` che genera un vettore di movimento casuale ad ogni round, è possibile definire uno scalare (`scale`) che rappresenta la velocità massima del movimento casuale.

Sono disponibili anche blocchi più complessi (**??**), alcuni di questi simulati in fig. 2.8:

- `goTo`: permette di definire un target (posizione assoluta) alla quale un agente deve arrivare
- `explore`: definito uno spazio rettangolare con i valori `minBound` e `maxBound`, il blocco produce vettori da assegnare all'agente per esplorare l'intero spazio

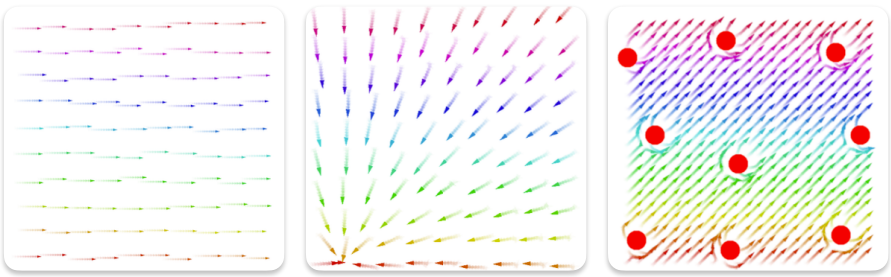

Figure 2.8: Simulazioni dei blocchi di movimento eseguita su Alchemist.

- `maintainTrajectory`: permette di mantenere un certo vettore di movimento per un certo periodo di tempo
- `maintainUntil`: permette di mantenere un certo vettore di movimento fino a quando non viene raggiunta una certa condizione
- `obstacleAvoidance`: genera vettori di movimento da assegnare agli agenti per evitare ostacoli

In figura fig. 2.8 sono state simulate le funzioni `goTo`, `explore` e `obstacleAvoidance` su Alchemist, un simulatore di sistemi aggregati.

```
1   // movimento casuale
2   def brownian(scale: Double): Vector
3
4   // posizioni assolute
5   def goTo(target: Point3D): Vector
6   def explore(minBound: Point3D, maxBound: Point3D): Vector
7
8   // condizioni temporali
9   def maintainTrajectory(trajectory: => Vector)(time: FiniteDuration):Vector
10
11  // condizioni booleane
12  def maintainUntil(direction: Vector)(condition: Boolean): Vector
13
14  // elusione ostacoli
15  def obstacleAvoidance(obstacles: List[Vector]): Vector
```

Questi blocchi possono essere assemblati per costruirne di nuovi più complessi, ad esempio, per creare un comportamento “muovi fino a destinazione evitando gli

ostacoli":

```
def moveToTargetAvoidingObstacles(target: Point3D, obstacles: List[Vector]):
    Vector = {
    val targetMovement = goTo(target)
    val obstacleAvoidanceMovement = obstacleAvoidance(obstacles)
    (targetMovement + obstacleAvoidanceMovement).normalize
}
```

La composizione di questi blocchi permette di creare comportamenti complessi in modo semplice e intuitivo andando a sommare comportamenti più semplici. È importante andare a *normalizzare* il risultato del comportamento complesso per produrre un singolo vettore nel caso in cui il movimento sia composto da più vettori.

Leader-based blocks Per gestire un insieme di agenti in modo coordinato per raggiungere un obiettivo si è valutato che ci fosse la necessità di avere un nodo leader che si occupasse di coordinare gli altri nodi. Il leader può essere scelto secondo un certo criterio (ad esempio per le sue caratteristiche speciali oppure per la sua posizione all'interno del gruppo) oppure casualmente ma è anche possibile andare a creare un leader *virtuale*, che quindi non è realmente presente nel ambiente. Allo stato attuale sono presenti solo due blocchi ritenuti essenziali:

- `alignWithLeader`: permette di allineare la velocita degli agenti con quella del leader
- `sinkAt`: permette di far convergere gli agenti verso il leader

Nel caso si necessiti di avere un comportamento più complesso che porti alla nascita di sotto-team si possono utilizzare i blocchi presenti in **Team formation blocks** che non approfondiremo in questo contesto.

Pattern formation blocks Per creare comportamenti coordinati per un team al fine di raggiungere un task si possono usare i blocchi presenti in **Leader-based blocks** e **Team formation blocks**, nel caso invece si è interessati soltanto alla forma geometrica che il team deve assumere si possono usare i blocchi presenti in **Pattern formation blocks**. La seconda tipologia di blocchi è una derivazione della prima poiché necessita della identificazione di un leader, il leader si occupa di

Figure 2.9: Simulazioni delle formazioni eseguite su Alchemist, in ordine: linea, formazione a V, cerchio.

collezionare le distanze dei propri "sottoposti" per poi assegnare loro una certa direzione per raggiungere la formazione desiderata. Il leader utilizza il Gradient-cast ed i Collect-cast per raccogliere le informazioni e utilizza, nuovamente, Gradient-cast per assegnare le direzioni.

Allo stato attuale del framework è possibile costruire in modo semplice le seguenti figure geometriche:

- formazione a "V", come quella dei volatili
- linea
- cerchio

Tutte e tre le formazioni fig. 2.9 si possono muovere nello spazio e necessitano di un leader che può anche variare velocità, in quel caso i nodi vicini si adatteranno alla velocità del leader. Questo tipo di formazioni sono capaci di ricostruirsi anche nel caso cambi il numero di nodi oppure un fenomeno esterno vada a disturbarne la struttura.

2.4 Thymio e tdmclient

> "Thymio è un robot educativo open-source progettato da ricercatori dell'EPFL (Politecnico federale di Losanna), in collaborazione con l'ECAL (Università d'arte e design di Losanna), e prodotto da

Figure 2.10: Robot Thymio

> Mobsya, un'associazione no-profit la cui missione è quella di offrire percorsi STEAM completi e coinvolgenti a studenti di tutte le età."
>
> – *Mobsya*

Thymio è un robot programmabile con un'ampia varietà di sensori e attuatori, tra cui:

- 9 sensori a infrarossi (portata circa 10 cm)
- 5 pulsanti a sfioramento (tecnologia capacitiva)
- 1 accelerometro a tre assi
- 1 termometro
- 1 microfono
- 1 ricevitore a infrarossi per il telecomando
- 39 LED controllabili

- 2 motori DC collegati alle ruote
- 1 altoparlante

Il deploy del codice sul robot Thymio avviene tramite un radio dongle USB.

Thymio Network : La connessione wireless dei robot è basata sul protocollo 802.15.4 in banda con frequenza 2.4 GHz. Questo protocollo permette di gestire una rete con molti nodi a discapito del rispetto delle seguenti condizioni:

- tutti i nodi devono trovarsi tutti nello stesso canale radio (sono disponibili 3 reti: [0, 1, 2])
- tutti i dispositivi devono avere lo stesso Identificativo di rete (PAN ID)
- tutti devono avere un nodo Identificativo univo (nodeID)

Questo protocollo è stato scelto per la sua bassa potenza e la sua affidabilità. Il protocollo è stato implementato in modo da permettere la comunicazione tra i robot e con il computer tramite il dongle USB.

Il dispositivo è stato scelto per essere implementato nella demo per la notte dei ricercatori in quanto è un robot molto versatile e adatto a molteplici applicazioni. Inoltre, il robot è molto diffuso nelle scuole e nei laboratori di robotica educativa, quindi è un'ottima scelta per mostrare il potenziale di ScaFi in un contesto educativo.

Questo piccolo robot nasce per essere programmato con linguaggi a blocchi come VPL, VPL3, Scratch, Blocky e non come Aseba, Python e Ros, tutti accessibili dalla suite dedicata *"Thymio Suite"*. Noi ci concentreremo sul uso di Python.

Python a differenza di altri linguaggi (ad esempio C/C++) è interpretato e non compilato. I linguaggi compilati prevedono la conversione del programma scritto in un linguaggio ad alto livello in linguaggio macchina per poi farlo eseguire dal processore, occorre quindi compilare il sorgente ad ogni nuova modifica per poi re-eseguirlo. Python, invece, è un linguaggio interpretato, il codice sorgente viene eseguito direttamente dal programma interprete, il quale esegue ogni comando riga per riga.

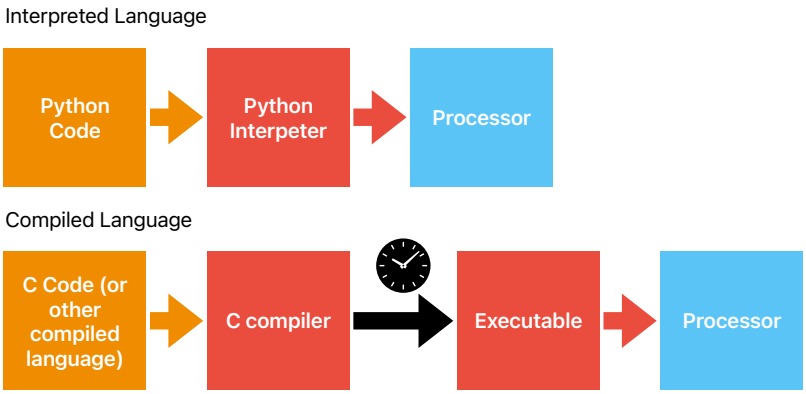

Figure 2.11: Interprete vs Compilatore

Nel caso del robot Thymio, non è presente un interprete nel suo microcontrollore. È presente, invece, una Aseba Virtual Machine che permette di eseguire programmi scritti in linguaggio Aseba.

Aseba è un linguaggio di programmazione ad alto livello basato su architettura ad eventi, il che significa che gli eventi sono eseguiti in modo asincrono. Gli eventi sono identificati da un Identificativo ed opzionalmente da un *payload* (dati aggiuntivi). Gli eventi possono essere di due tipi:

- **global events**: eventi generati dai nodi e condivisi con la rete
- **local events**: eventi generati da un nodo e non condivisi con la rete (ad esempio un evento generato da un sensore dello stesso nodo)

Un esempio di codice Aseba è il seguente listing 2.4.

Listing 2.4: Esempio di codice Aseba con eventi

```
var state

callsub init  # Inizializza il programma

sub init
    state = 0
    call leds.bottom.left(0,0,32)
    call leds.bottom.right(0,32,0)
    call leds.top(32,0,0)
```

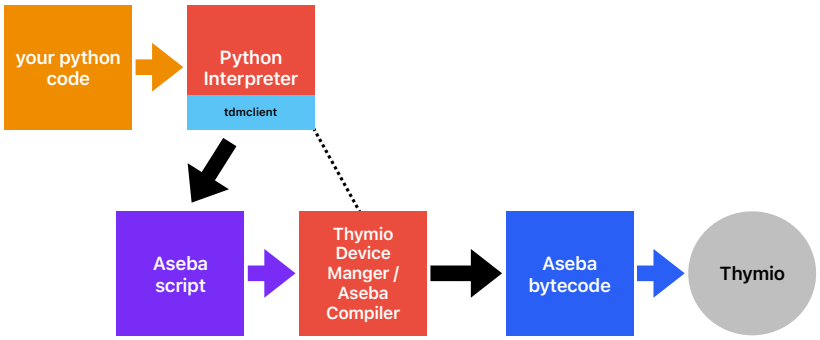

Figure 2.12: Tdmclient workflow

```
10
11      # Re-inizia quando il pulsante centrale viene premuto
12      onevent button.center
13          callsub init
```

Il **Thymio Device Manager** è una delle feature della Thymio Suite, si occupa di gestire la rete dei Thymio. Inoltre ha il compito di tradurre il sorgente, scritto dall'utente, da linguaggio Aseba ad Aseba bytecode per poi eseguire il *deploy* sul robot. Per l'utilizzo di Python, invece, è necessario l'utilizzo del modulo **tdmclient** che si occupa di far comunicare Python con il Thymio Device Manager (TDM) fig. 2.12. È necessario che la Thymio Suite sia in esecuzione per poter utilizzare il modulo **tdmclient**, il quale, a sua volta, necessita dell'utilizzo di **Python 3** come interprete.

Questo modulo permette di:

- accedere alle variabili, sensori, attuatori del robot
- convertire uno script Python in un script Aseba (transpiler)
- inviare codice Aseba al TDM che a sua volta invia il relativo bytecode al robot Thymio

Le possibilità offerte da questo modulo sono molteplici, ad esempio è possibile inviare comandi direttamente ad un robot Thymio da terminale:

Listing 2.5: Esempio di invio di comando (foo) ad un robot Thymio

```
python3 -m tdmclient sendevent --event foo --data 123
```

In listing 2.5 si assume che il robot accetti un evento di nome *foo* con un payload di tipo intero e che il robot sia in ascolto di eventi (*unlocked*). Per metter in ascolto un robot di un certo evento è necessario andare ad inserire una vista su quel determinato robot dalla Thymio Suite.

Listing 2.6: Esempio di programma interno a Thymio per la ricezione di eventi (foo)

```
var x

onevent foo
    x = event.args[0]
```

Il modulo permette di eseguire il transpile (conversione) di un programma Python in Aseba. Per farlo è necessario utilizzare il comando *transpile* del modulo **tdmclient**:

Listing 2.7: Esempio di transpile di un programma Python (print.py) in Aseba

```
python3 -m tdmclient transpile examples/print.py
```

Utilizzando il modulo aggiuntivo `ClientAsync` è possibile modificare le variabili dei robot Thymio attraverso chiamate asincrone in uno script **Python**. Si andrà ad esaminare ulteriormente questa feature di **tdmclient** nel capitolo chapter 4 e chapter 5 poiché è ciò che si è utilizzato per la progettazione del nostro sistema.

2.5 Aruco Tag

Gli Aruco marker sono tag quadrati che possono essere rilevati da una camera. Ogni tag è una matrice 4×4 ed ha un codice (intero) univoco associato che può essere rilevato da un algoritmo di visione artificialefig. 2.13. È rilevante notare che la matrice di ogni Aruco marker corrisponde ad uno stesso codice per ogni sua versione trasposta. Questi tag sono molto utili per la localizzazione e la navigazione di robot autonomi. Sarà questo strumento che verrà utilizzato per la demo della notte dei ricercatori per la localizzazione e calcolo della direzione dei robot Thymio.

Questo strumento è già presente nel sistema in questione.

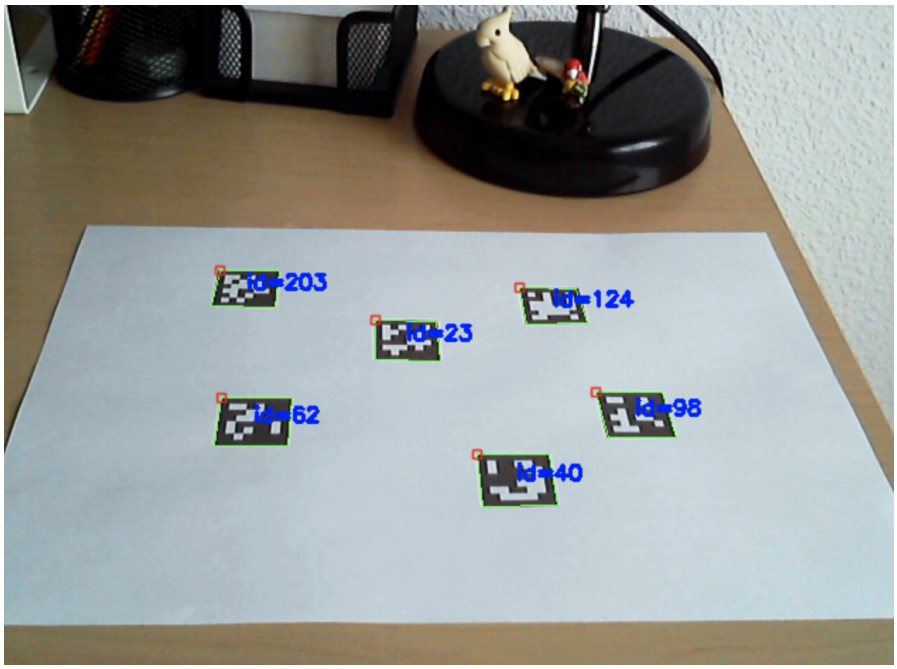

Figure 2.13: Aruco markers identificati da un algoritmo di visione artificiale

Chapter 3

Analisi

3.1 Obiettivi

Il progetto "researcher-night-demo" è nato per dimostrare, attraverso simulazioni in ambiente reale, le potenzialità di ScaFi. L'obiettivo di questo progetto è dimostrare l'eterogeneità di questo tipo di sistemi andando ad estendere ai robot già implementati il supporto per un nuovo modello di robot, il Thymio, e andando a gestire i vincoli di compatibilità del sistema Thymio.

Inizialmente, la demo, presentava soltanto la possibilità di utilizzare i robot Wave fig. 3.1. I Wave sono robot equipaggiati da 4 ruote motrici (4WD), scocca d'acciaio ed un modulo ESP32 (microcontrollore programmabile) già implementato di software (open-source) con il quale si può comunicare tramite protocollo WiFi o Bluetooth. Il progetto è stato esteso per includere il modello di robot Thymio fig. 2.10.

3.2 Requisiti

Requisiti funzionali:

- Il sistema è progettato per l'utilizzo di grandi quantità di dispositivi eterogenei quindi si ritiene necessario che vengano caricati attraverso file di configurazione invece che essere definiti direttamente nel codice. Questo permette di avere un sistema più flessibile e facilmente estendibile.

Figure 3.1: Robot Wave

- Integrare robot Thymio nel sistema già esistente.

Requisiti non funzionali:

- Refactoring del codice Java per il tracciamento dei robot.
- Velocità nelle comunicazioni con i nuovi robot integrati.

3.3 Dominio

Il progetto è stato sviluppato in linguaggio Scala ed è formato da tre componenti fisiche:

- webcam per il tracciamento dei ArucoTag posizionati sopra ai robot
- macchina che esegue il sistema aggregato e che comunica con i robot, solitamente un computer
- robot (uno o più)

La struttura del progetto si basa sul principio di Macroswarm, posizionandosi ad un livello di astrazione abbastanza alto da nascondere i dettagli implementativi dei dispositivi che si vuole andare a controllare. Di seguito un breve sommario delle componenti principali del progetto:

Tracciamento robot . Per il tracciamento dei robot attraverso la webcam è stato sviluppato un applicativo Java di visione artificiale che rileva i tag Aruco e ne calcola la posizione e l'orientamento.

Core . Il core del sistema è basato su ScaFi. Per costruire i sistemi aggregati si utilizza la classe `AggregateIncarnation` listing 3.1 che serve a poter definire i diversi programmi aggregati.

Listing 3.1: Classe AggregateIncarnation

```
import it.unibo.scafi.incarnations.BasicAbstractIncarnation

object AggregateIncarnation extends BasicAbstractIncarnation with
    BuildingBlocks
```

L'interfaccia `Environment`listing 3.2 rappresenta l'astrazione del mondo, essa rappresenta lo stato del mondo in un dato istante e va definita in base al contesto.

Listing 3.2: Interfaccia Enviroment

```
trait Environment[ID, Position, Info]:
    def nodes: Set[ID] // insieme di nodi definiti dagli identificatori ID
    def position(id: ID): Position // posizione di un certo nodo
    def sensing(id: ID): Info // le informazioni che un certo nodo espone
    def neighbors(id: ID): Set[ID] // vicinato di un certo nodo
```

La classe `AggregateOrchestrator` è la componente principale del sistema poiché si occupa di leggere lo stato del mondo e ritornare l'attuazione da eseguire (export) per ogni agente.

Demo In demo si trova la reale implementazione del sistema aggregato che produce le attuazioni nel mondo reale a partire dalla definizione del ambiente in cui si lavoralisting 3.3.

Listing 3.3: Definizione dell'ambiente

```
import it.unibo.core.DistanceEstimator.distance
import it.unibo.core.Environment
import it.unibo.demo.{ID, Info}
import it.unibo.utils.Position.{Position, *, given}

class DemoEnvironment(data: Map[ID, (Position, Info)], neighboursRadius:
    Double)
    extends Environment[ID, Position, Info]:

override def nodes: Set[ID] = data.keySet

override def position(id: ID): (Double, Double) = data(id)._1

override def sensing(id: ID): Info = data(id)._2
```

```
14
15      override def neighbors(id: ID): Set[ID] =
16          data.filter { case (k, v) => data(id)._1.distance(v._1) <=
                neighboursRadius }.keys.toSet
```

Mock Il progetto dispone anche di uno spazio per testare programmi aggregati in un ambiente simulato implementato utilizzando la libreria `JavaFX`. Questo ambiente permette di vedere i nodi della rete e le connessioni tra di essi.

Chapter 4

Design

A seguito di un analisi delle caratteristiche dei robot Thymio e dei protocolli di comunicazione che esso mette a disposizione si è optato per l'utilizzo del modulo **tdmclient** nel contesto di un server sviluppato utilizzando il framework Flask. Questo framework è stato sviluppato per la creazione di applicazioni web e, in particolare, per la creazione di *API RESTful*[1]. Questo framework è stato scelto per la sua semplicità, la sua flessibilità, scalabilità e indipendenza dalla tecnologia utilizzata.

Si è valutato anche l'utilizzo della libreria ScalaPy, che permette di sfruttare i moduli Python all'interno di un programma Scala, per incorporare le funzionalità di tdmclient direttamente all'interno del sistema aggregato ma data la "non stabilità" del progettato ScalaPy si è deciso di non impiegare questa soluzione.

4.1 Architettura server Flask

Nel nostro contesto si è sviluppato un server che comunica con due clientfig. 4.1:

- il client Thymio Device Manager (TDM) che comunica con i robot Thymio
- il client che ospita il programma aggregato

[1]Le API sono l'interfaccia che utilizzano due o più sistemi informatici per lo scambio sicuro di informazioni. REST è un'architettura software sulla quale si possono progettare le API

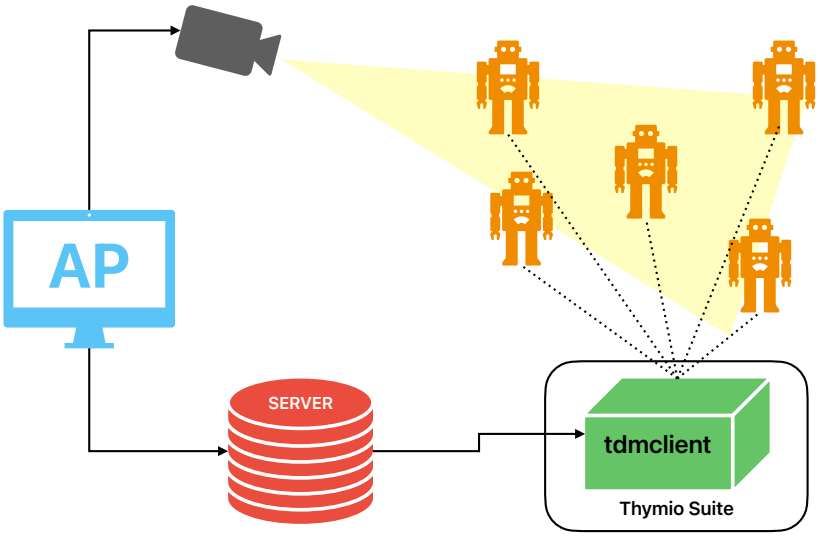

Figure 4.1: Architettura del server Flask

4.2 File di configurazione

Chapter 5

Implementazione

5.1 Implementazione del server Flask

5.2 Esempi di Algoritmi AP applicati ai Robot Wave e Thymio nello stesso ambiente

You may also put some code snippet (which is NOT float by default), eg: section 5.2.

5.3 Fancy formulas here

```
1 public class HelloWorld {
2   public static void main(String[] args) {
3     // Prints "Hello, World" to the terminal window.
4     System.out.println("Hello, World");
5   }
6 }
```

Chapter 6

Conclusione

Bibliography

Acknowledgements

Optional. Max 1 page.