

Corso di Laurea in Ingegneria e Scienze Informatiche

Eterogeneità dei sistemi di Aggregate Programming: un caso studio con WaveRobot e ThymioRobot

Tesi di laurea in:
OBJECTIVE ORIENTED PROGRAMMING

Relatore
Chiar.mo Prof. Mirko Viroli

Candidato
Elvis Perlika

Correlatore
Dott. Gianluca Aguzzi

Abstract

Il mondo IoT è in costante crescita e di conseguenza nasce la necessità di sviluppare sistemi aggregati, cioè formati da rilevanti quantità di nodi. L'Edge Computing è, ad oggi, l'approccio più utilizzato. Il motivo risiede nel buon compromesso tra capacità computazionale e velocità di comunicazione tra i nodi della rete. Risulta però non in grado di identificare una qualche astrazione di “Intelligenza Collettiva Emergente” per via della sua natura fortemente concentra sull’aspetto funzionale dei singoli dispositivi. La volontà di colmare questa lacuna e quindi di poter creare un’intelligenza distribuita in modo efficace, intuitivo e che permetta di astrarre dalla tecnologia dei dispositivi dell’insieme su cui si vuole definire la strategia operativa ha dato origine al mondo dell’Aggregate Computing. I contesti di applicazione di queste tecnologie sono molteplici, dal controllo di sensori IoT per smart city al controllo di droni in formazione per missioni di salvataggio. L’obiettivo di questa tesi progettuale è dimostrare come la macroprogrammazione e le tecnologie che ne derivano, come il framework ScaFi e la sua estensione MacroSwarm, risultino essere un paradigma di programmazione particolarmente efficace nella progettazione di sistemi aggregati eterogenei, cioè formati da dispositivi con caratteristiche e protocolli di comunicazione diversi. Si andrà a discutere il caso studio dell’integrazione dei robot Thymio in un programma aggregato.

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

Contents

Abstract	iii
1 Introduzione	1
2 Background	3
2.1 Paradigma Aggregate Programming	3
2.1.1 Scala	3
2.1.2 Sistemi collettivi ed eterogenei	5
2.1.3 Stato dell'arte	8
2.2 ScaFi	14
2.3 Macroswarm	19
2.4 Thymio e tdmclient	27
2.5 Aruco Tag	31
3 Analisi	33
3.1 Obiettivi	33
3.2 Requisiti	34
3.3 Dominio	34
4 Design	35
4.1 Architettura server Flask	35
5 Implementazione	37
5.1 Implementazione del server Flask	39
6 Validazione	43
6.1 File di configurazione	45
7 Conclusione	49

CONTENTS

51

Bibliography 51

List of Figures

2.1	Architettura del mondo IoT	6
2.2	Droni in formazione in ambiente notturno	8
2.3	Da sensori di temperatura $d_i \forall i \in \{\text{Insieme dei sensori.}\}$ a spazio continuo di temperatura $t(x, y, z)$	10
2.4	Manipolazioni diverse in base al luogo. In questo caso sensori più vicini al centro della stanza hanno un comportamento diverso rispetto a quelli più vicini alle pareti.	11
2.5	Sistema che si adatta alla perdita di nodi inserendo nuovi collegamenti.	11
2.6	Da dispositivi $d_i \forall i \in \{\text{Insieme dei dispositivi.}\}$ a valori $f(d_i)$	12
2.7	Fasi di un round di computazione in un sistema aggregato.	13
2.8	Architettura ad alto livello del framework ScaFi	15
2.9	Architettura del modulo scafi-core	16
2.10	Leader election	19
2.11	Gradient cast	19
2.12	Collect cast	19
2.13	Architettura di Macroswarm	21
2.14	Struttura di Macroswarm e moduli interni	22
2.15	Modalità Round Based	22
2.16	Modalità Long Standing	23
2.17	Simulazioni dei blocchi di movimento eseguita su Alchemist.	24
2.18	Simulazioni delle formazioni eseguite su Alchemist, in ordine: linea, formazione a V, cerchio.	26
2.19	Robot Thymio	27
2.20	Interprete vs Compilatore	29
2.21	Tdmclient workflow	30
2.22	Aruco markers identificati da un algoritmo di visione artificiale	32
3.1	Robot Wave	33
4.1	Architettura del server Flask	36

LIST OF FIGURES

5.1	Interfaccia web per il controllo dei robot Thymio	41
6.1	Simulazione del programma aggregato per far ruotare i robot del sistema seguendo la direzione del leader	44
6.2	Simulazione del programma aggregato per allineare i robot.	44
6.3	Simulazione del programma aggregato per allineare i robot dalla visuale del programma di tracking dei Aruco Tag (la linea rossa è stata inserita in post produzione).	45
6.4	Simulazione del programma aggregato per produrre un cerchio intorno ad un leader.	46

List of Listings

2.1	Interfaccia per la definizione di un campo di calcolo in Scala	17
2.2	Funzioni del modulo di movimento	24
2.3	Comportamento complesso	25
2.4	Esempio di codice Aseba con eventi	29
2.5	Esempio di invio di comando (foo) ad un robot Thymio	31
2.6	Esempio di programma interno a Thymio per la ricezione di eventi (foo)	31
2.7	Esempio di transpile di un programma Python (print.py) in Aseba .	31
5.1	Classe AggregateIncarnation	37
5.2	Interfaccia Enviroment	38
5.3	Definizione dell'ambiente	38
5.4	Funzione di setup per rilevare i Thymio nella rete	39
5.5	API per il controllo dei Thymio	39
5.6	Implementazione API per inviare un comando di movimento al Thymio	39
5.7	Classe per il controllo dei robot	40
6.1	Programma aggregato per far ruotare i robot del sistema seguendo la direzione del leader	43
6.2	Programma aggregato allineare i robot	44
6.3	File di configurazione per i robot Thymio	47
6.4	File di configurazione per i robot Wave	47
6.5	Lettura dei file di configurazione	47

LIST OF LISTINGS

Chapter 1

Introduzione

Negli ultimi anni, il numero di dispositivi interconnessi ha conosciuto una crescita esponenziale, soprattutto nel contesto dell'Internet of Things (IoT). Con un'ampia varietà di dispositivi eterogenei, dalle telecamere di sorveglianza ai sensori industriali, il mondo si sta popolando di sistemi intelligenti capaci di raccogliere, analizzare e scambiare dati in tempo reale. Questa proliferazione porta alla necessità di una tecnologia che possa coordinare il comportamento di gruppi di dispositivi in modo efficace, resiliente e scalabile, soprattutto in contesti dinamici e complessi. In questo scenario, tecniche di controllo collettivo e paradigmi di programmazione aggregata si rivelano fondamentali per garantire che i dispositivi possano operare in sinergia verso obiettivi comuni, gestendo autonomamente le interazioni locali e contribuendo a un comportamento globale emergente.

Nel tempo si è compreso che un approccio possibile potesse essere quello di rendere i dispositivi della rete una sorta di “superorganismo” in grado di gestirsi autonomamente e che necessiti, eventualmente, di comandi ad alto livello dai amministratori del sistema.

TODO: da finire

Struttura della tesi Nel capitolo chapter 2 si andranno ad approfondire tutte le tecnologie rilevanti per

Chapter 2

Background

2.1 Paradigma Aggregate Programming

2.1.1 Scala

Il difetto della programmazione ad oggetti è che richiede un particolare impegno nel gestire i sistemi da realizzare all'aumentare della loro complessità. Altri paradigmi, come la programmazione funzionale, sono stati sviluppati per risolvere questo problema.

Nel caso della OOP abbiamo riassunto il paradigma con la frase “*Everything is an Object*”, per la programmazione funzionale, invece, possiamo riassumerla con “*Everything is a Function*”. Quando si parla di funzioni nel ambito della Functional Programming (FP) si fa riferimento alle **funzioni pure**[Hun18], cioè funzioni “prive di effetti collaterali”. Per funzione “priva di effetti collaterali” si intende una funzione che non 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'è alcuna limitazione sulla complessità del software da costruire poiché questo paradigma non rappresenta reali estensioni strutturali al paradigma Objective Oriented Programming (OOP), bensì esprime un nuovo modo di pensare e scrivere il codice. È possibile immaginare tale paradigma come una filosofia nel mondo della programmazione.

Nel dettaglio, per **funzione pura** si intende una funzione $f : A \rightarrow B$, (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 ??.

Formalmente si può definire una funzione pura con il concetto di Referential Transparency (RT) [Hun18]:

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

Inoltre, la programmazione funzionale incoraggia l'uso di dati immutabili, il che significa che una volta creato un dato, non viene modificato. Questo riduce le possibilità di errori difficili da tracciare, perché le funzioni non alterano lo stato globale ma restituiscono nuovi valori. Nei progetti complessi, ciò rende più facile comprendere e prevedere il comportamento del codice. Grazie all'immutabilità e alle funzioni pure, la programmazione funzionale è particolarmente adatta per il calcolo parallelo e concorrente. L'assenza di modifiche condivise tra le funzioni evita il problema dei *race condition*, che si verifica quando due processi cercano di accedere o modificare contemporaneamente lo stesso dato. Questo è cruciale per la scalabilità nei sistemi complessi.

Queste proprietà rendono i programmi facili da testare e da correggere nel caso di bug. Di conseguenza rendono un programma progettato con approccio funzionale di essere maggiormente scalabile e mantenibile.

2.1.2 Sistemi collettivi ed eterogenei

Per comprendere il potenziale dell'Aggregate Programming (AP), che verrà esplorato nella sezione seguente, è fondamentale definire quale tipo di problema si vuole risolvere.

La crescita del mondo Internet of Thing (IoT) ha portato alla presenza di numerosi dispositivi connessi tra loro. Dispositivi di ogni tipologia 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 con l'obiettivo di gestire queste grandi quantità di dispositivi e dati. È necessario che questi sistemi siano capaci di:

- adattarsi a cambiamenti imprevedibili: i contesti in cui operano i dispositivi possono cambiare in modo imprevedibile, ad esempio a causa di guasti o malfunzionamenti e prevedere tutti gli scenari risulta essere altamente complesso.
- permettere l'implementazione di nuovi comportamenti: la nascita di nuovi bisogni, variazioni nel ambiente, evoluzione delle tecnologie, ottimizzazione delle risorse o modifiche delle normative richiedono la capacità di implementare nuovi comportamenti in modo rapido ed efficace.
- permettere una capacità di rilevazione e attuazione nel ambiente in modo distribuito e coordinato: la raccolta di informazioni coordinata è cruciale nello sviluppo di un'intelligenza avanzata e la capacità di attuare azioni in modo coordinato è fondamentale per il raggiungimento di obiettivi comuni evitando conflitti e sovrapposizioni.

Allo stato attuale della tecnologia è difficile e costoso progettare sistemi di questo tipo poiché non esistono framework che permettano di gestirne, in modo semplice e intuitivo, la complessità [BPV15] [Cas20].

Un'astrazione grafica del mondo IoT è presente in fig. 2.1 [TADT22]. È possibile comprendere come ci sia uno strato contenente i nodi (dispositivi)¹, uno strato

¹HMI: Human-Machine-Interaction.

2.1. PARADIGMA AGGREGATE PROGRAMMING

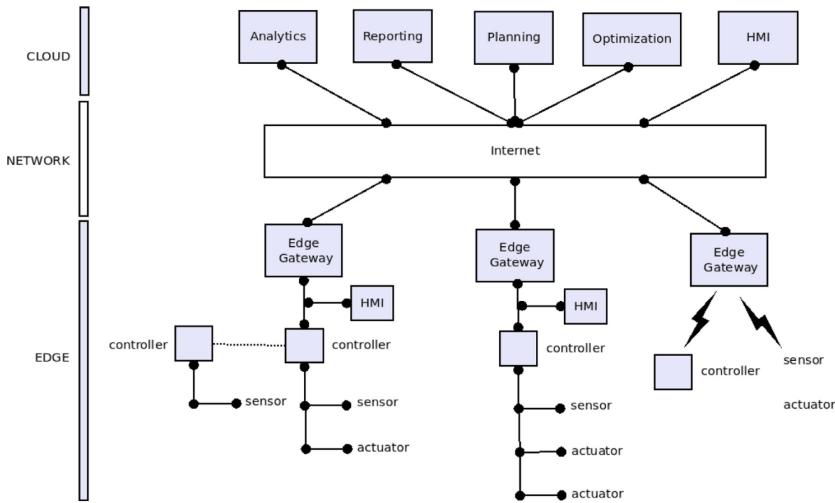


Figure 2.1: Architettura del mondo IoT

che rappresenta la rete (internet) mediante la quale comunicano² ed uno strato contenente 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 ritrovabile nelle problematiche nascenti dall'inviare ingenti quantità di dati da un numero elevato di nodi. Problema che può provocare effetti negativi sui costi, le disponibilità, la scalabilità della piattaforma e la latenza delle comunicazioni. Una delle soluzioni trovate è quella di spostare la computazione più vicino possibile ai nodi della rete, in modo da ridurre la quantità di dati da inviare e di conseguenza ridurre i problemi di latenza nelle comunicazioni. Questo approccio, chiamato *Edge Computing* consiste in quattro passaggi principali:

1. raccolta dati
2. pre-processamento locale dei dati sorgente
3. definita una strategia immediata

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

2.1. PARADIGMA AGGREGATE PROGRAMMING

4. condivisione della strategia con il Cloud o pre-processamento dei dati sul Cloud

Per quanto semplice, questo approccio va a mettere in una situazione di stress le organizzazioni di applicazioni IoT. Inoltre, non è sempre conveniente applicarlo, dipende dal contesto. Ad esempio non sarebbe adatto nel caso i nodi abbiano specifiche tecniche, come memoria e capacità di calcolo, deboli [TADT22].

Andando ad osservare nel dettaglio solo i dispositivi di questa architettura complessa è possibile notare come molto spesso, anche se in apparenza impercettibile, è presente un aspetto collettivo che merita di essere considerato. Preso un sistema in cui è presente una gestione dei dati distribuita, nella quale i nodi della rete condividono le proprie informazioni e quelle che hanno ricevuto da altri nodi al fine di raggiungere un obiettivo comune, se questo sistema è in grado di adattarsi ai cambiamenti e continuare nella realizzazione del task è possibile che esso vada a far emergere una forma di intelligenza collettiva autonoma.

La forza di un sistema del genere sta nella sua resilienza: ogni nodo può operare localmente con regole relativamente semplici, ma la somma delle interazioni può produrre una risposta globale complessa e adattabile. Tale struttura è particolarmente efficace in contesti in cui è richiesta una risposta flessibile e continua agli imprevisti, come nei casi di reti neurali distribuite, sistemi di swarm robotics, e ambienti IoT (Internet of Things) su larga scala.

Questa intelligenza distribuita può, quindi, essere considerata autonoma nella misura in cui non richiede una gestione centralizzata per adattarsi o evolversi, riuscendo a creare soluzioni nuove senza un controllo diretto. Questo fenomeno diventa particolarmente evidente in situazioni di problem-solving distribuito, dove i nodi, seguendo semplici protocolli di comunicazione e scambio, ottimizzano il risultato collettivo migliorando così la capacità del sistema di raggiungere obiettivi complessi.

Sono numerosi i casi di sistemi collettivi che necessitano di un controllo distribuito e coordinato. Un esempio pratico di questa intelligenza collettiva autonoma si può vedere nelle reti di sensori distribuiti, che monitorano in tempo reale fenomeni ambientali (come incendi, movimenti sismici o cambiamenti climatici). Ogni sensore condivide dati con quelli vicini e l'intero sistema è in grado



Figure 2.2: Droni in formazione in ambiente notturno

di rilevare e rispondere ad anomalie senza un controllo centrale. Questo comportamento emergente, basato su regole locali e interazioni dinamiche, rende questi sistemi più robusti, adattabili e affidabili in contesti complessi. Un altro 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 comune fig. 2.2.

È chiara la necessità di una tecnologia che permetta di costruire nuove applicazioni (o comportamenti) per il controllo di collettivi di dispositivi senza dover andare a definire i singoli comportamenti dei dispositivi, bensì trattando questi singoli dispositivi come un unico superorganismo.

2.1.3 Stato dell’arte

L’Aggregate Programming è un concetto che nasce dal paradigma della “macro programmazione”.

Con *paradigma di macro-programmazione* si intende la teoria (e la pratica) per la progettazione di singoli programmi che gestiscono il controllo di “macro” comportamenti, cioè comportamenti di collettivi di componenti. La definizione data in letteratura è la seguente:

Paradigma astratto per programmare il comportamento (macro) scopico di sistemi formati da entità computazionali. [Cas23]

2.1. PARADIGMA AGGREGATE PROGRAMMING

Questo paradigma nasce con lo scopo di astrarre il comportamento e le interazioni tra i singoli componenti del sistema e si basa su quattro principi fondamentali [Cas23]:

- *Distinzione micro-macro.* Per *micro* si intendono le singole entità del ambiente, mentre per *macro* si intende il comportamento del sistema, il suo stato, la sua struttura.
- *Prospettiva macroscopica.* Il programmatore si concentra principalmente sul comportamento macroscopico del sistema, piuttosto che sul comportamento dei singoli componenti, questo non esclude osservazioni a livello microscopico.
- *Macro-programma.* L'attività di sviluppo genera un macro-programma, ovvero un codice che specifica la logica di funzionamento macroscopica.
- *Mappatura macro a micro.* Anche conosciuta come *mappatura globale a locale*, è la logica che permette al macro-programma di compiere le specifiche attuazioni a livello microscopico per completare il task.

Questa teoria e la tecnologia che ne deriva mira a migliorare la progettazione, manutenzione e testing di programmi nell'ambito del controllo di dispositivi hardware di larga-scala.

L'Aggregate Computing (AC) si basa sulla mappatura di un mondo cyber-fisico in un nuovo modello logico. Da un punto di vista strutturale possiamo vedere un sistema **aggregato** come una rete di dispositivi, capaci di computare, equipaggiati con (o sui quali è possibile equipaggiare) sensori ed attuatori. Ogni dispositivo corrisponde ad un nodo e questi nodi sono connessi tra di 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 sono ciò che permettono al contesto locale e globale di influenzarsi reciprocamente.

La tecnica dell'Aggregate Computing si basa su tre principi fondamentali per la costruzione di sistemi robusti e resilienti:

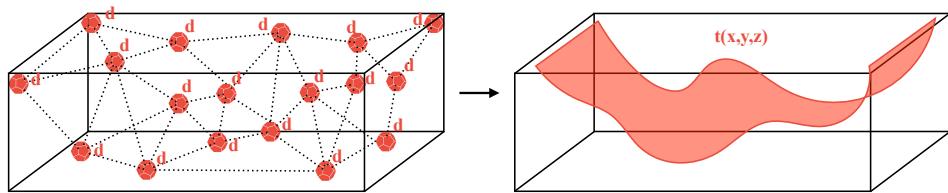


Figure 2.3: Da sensori di temperatura $d_i \forall i \in \{\text{Insieme dei sensori.}\}$ a spazio continuo di temperatura $t(x, y, z)$.

- **Aspetto strutturale.** I dettagli implementativi dei sistemi hardware che si vuole andare a manipolare devono essere nascosti così da permettere ai programmatore 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 pensare al dominio 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 fig. 2.3.
- **Aspetto comportamentale.** Il programma manipola le strutture dati della rete di dispositivi sia in funzione della loro estensione spaziale che di quella temporale. Approccio particolarmente utile in sistemi in cui l'informazione cambia in base al luogo fig. 2.4.
- **Aspetto interazionale.** 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 fig. 2.5.

Dalla tecnologia dell'Aggregate Computing nasce anche la volontà di definire un metodo per la programmazione di sistemi aggregati: l'Aggregate Programming (AP). 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

2.1. PARADIGMA AGGREGATE PROGRAMMING

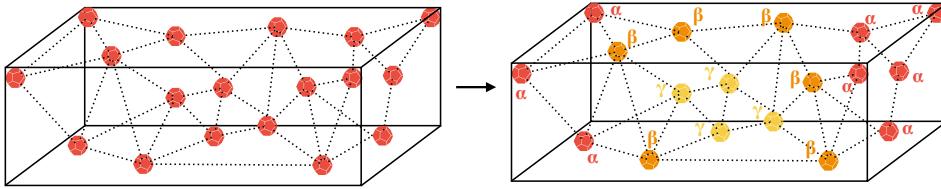


Figure 2.4: Manipolazioni diverse in base al luogo. In questo caso sensori più vicini al centro della stanza hanno un comportamento diverso rispetto a quelli più vicini alle pareti.

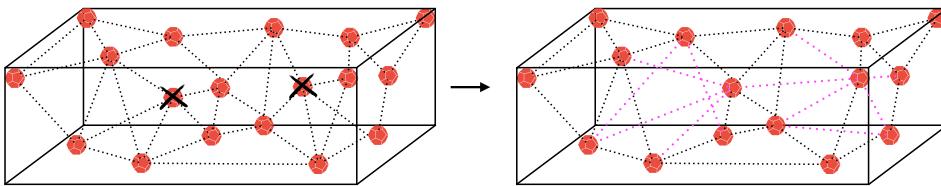


Figure 2.5: Sistema che si adatta alla perdita di nodi inserendo nuovi collegamenti.

approccio, seppur semplice, è poco scalabile e difficile da mantenere al crescere della complessità del sistema [PBV17].

Per *campi computazionali* o *computational fields* si intende la mappatura degli elementi del dominio in cui si opera in un insieme di valori fig. 2.6. I campi 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, come quello magnetico o quello gravitazionale. Ogni dispositivo della rete è considerato come un punto nello spazio ed il campo rappresenta un dato valore assegnato ad ognuno di questi punti. I campi, quindi, rappresentano un'astrazione ad alto livello dei dispositivi. L'insieme dei valori restituiti dalla mappatura rappresenta lo stato di un sistema in un dato istante [AVD⁺19].

Questo approccio è generalmente utilizzato per un controllo prolungato dei dispositivi della rete. È ideale per reti di dispositivi sulle quali si vuole eseguire task che necessitano un gran numero di rilevazioni (ad esempio utilizzando sensori), computazioni ed attuazioni nello spazio e nel tempo.

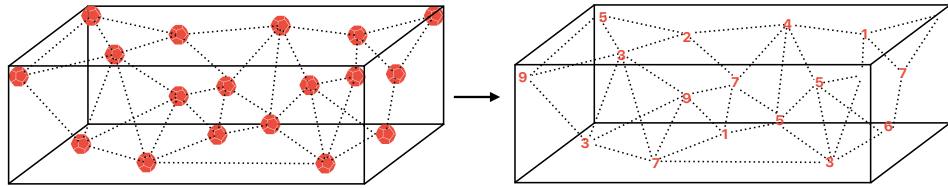


Figure 2.6: Da dispositivi $d_i \forall i \in \{\text{Insieme dei dispositivi.}\}$ a valori $f(d_i)$.

Ogni dispositivo del proprio sistema aggregato esegue una computazione in modo asincrono attraverso i **sense-compute-(inter)act rounds** [ACV24], ogni round è composto da tre fasi concettuali fig. 2.7 [CAV21]:

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 attuatori:** l'output del nodo viene utilizzato per l'esecuzione di uno o più attuazioni nel proprio ambiente

Posizionandoci in un livello più implementativo troviamo i seguenti costrutti per la manipolazione dei campi [PBV17]:

- **Functions:** funzioni

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

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

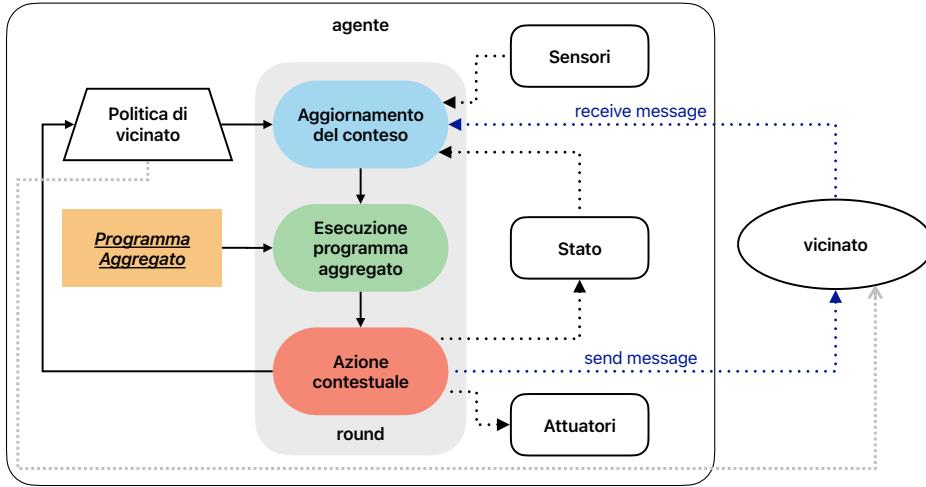


Figure 2.7: Fasi di un round di computazione in un sistema aggregato.

- **Dynamics:**

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

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

- **Interaction:**

$$nbr(s)$$

permette di fare una query di un certo valore rispetto al vicinato

- **Restrictions:**

```

if e
     $s_1; \dots; s_n;$ 
else
     $s'_1; \dots; s'_m;$ 

```

permette di andare a definire sotto spazio del campo principale 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.

Le tecnologie sviluppate con approccio AP non sono prive di difetti. Implementazioni di questo tipo sono difficili da integrare in sistemi già 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 lo sviluppo per lo sviluppo ad alto livello di sistemi aggregati costruito sulle basi dell'Aggregate Programming. Il termine ScaFi deriva dalla unione di **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 coinvolto 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* di cui abbiamo parlato nei capitoli precedenti.

L'Architettura di ScaFi [CVAP22] è riassumibile nella immagine fig. 2.8.

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 DSL³ per la progettazione di sistemi aggregati; con il supporto di librerie standard.
- **scafi-simulator**: fornisce un supporto per la simulazione dei sistemi di AP sviluppati
- **scafi-simulator-GUI**: fornisce un'interfaccia grafica per la simulazione

³Domain 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.

2.2. SCAFI

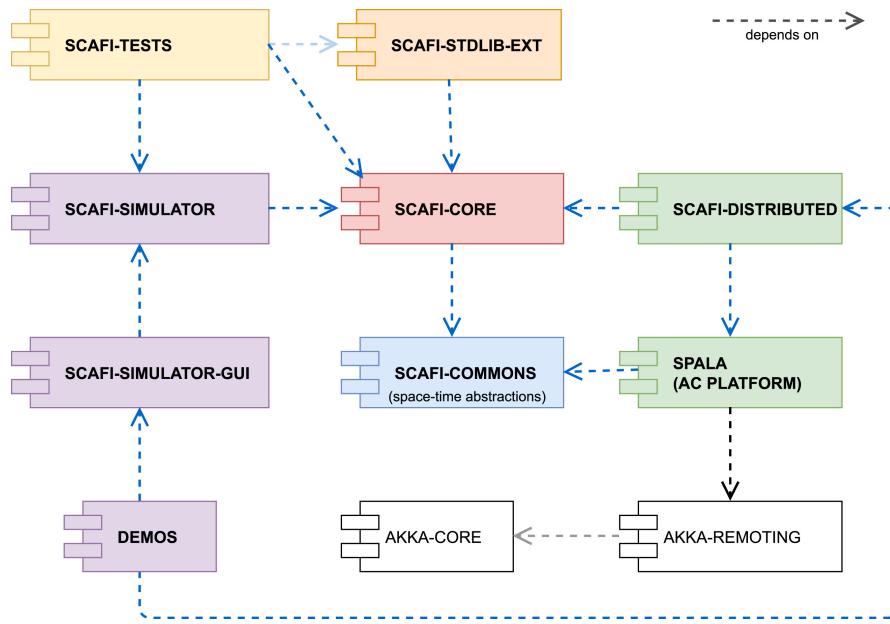


Figure 2.8: Architettura ad alto livello del framework ScaFi

- **spala**: (“spatial scala”) si tratta di un *middleware* che permette di eseguire programmi Aggregate Computing (AC) basati sugli attori ed è indipendente dal DSL di ScaFi
- **scafi-distributed**: layer integrativo per l'utilizzo di **spala** in ScaFi

scafi-core Questo modulofig. 2.9 espone l’incarnazione dell’interfaccia di un **AggregateProgram**, questo singolo programma si occupa di definire l’intera operatività del insieme dei dispositivi della rete. Andare a mischiare una sottoclassificazione di **AggregateProgram** con l’interfaccia **Gradients** permette di creare una

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

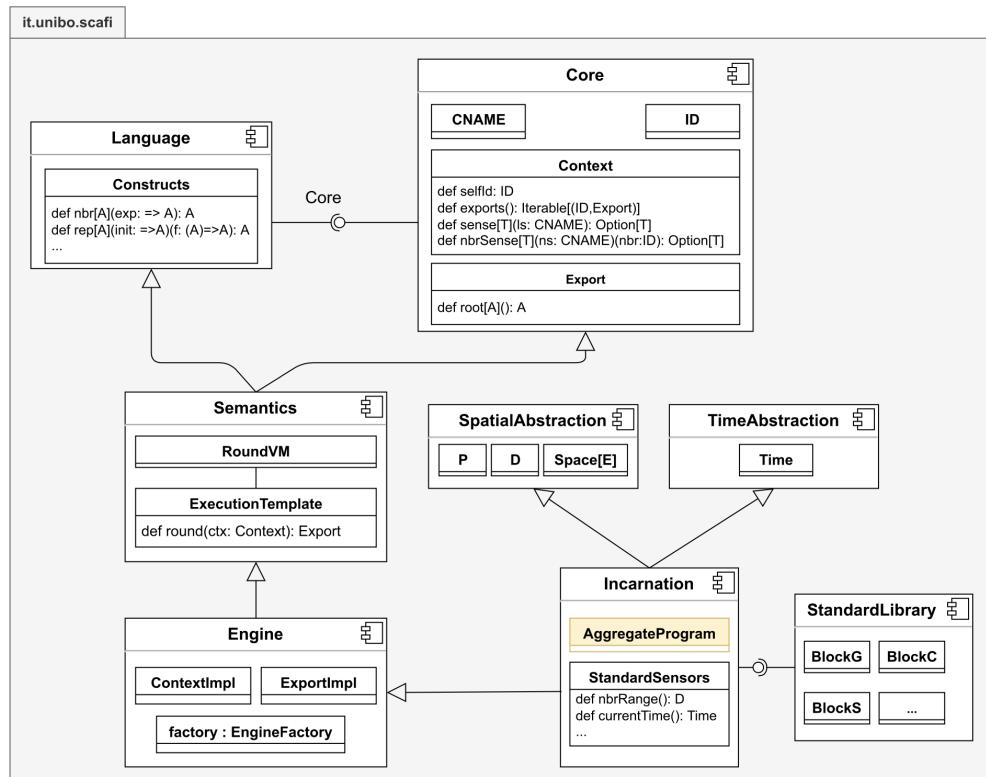


Figure 2.9: Architettura del modulo scafi-core

Listing 2.1: Interfaccia per la definizione di un campo di calcolo in Scala

```

1 trait Constructs {
2   def rep[A](init: => A)(fun: A => A): A
3   def nbr[A](expr: => A): A
4   def foldhood[A](init: => A)(acc: (A, A) => A)(expr: => A): A
5   def aggregate[A](f: => A): A
6
7   def branch[A](cond: => Boolean)(th: => A)(el: => A)
8   def share[A](init: => A)(fun: (A, () => A) => A): A
9
10  def mid: ID
11  def sense[A](sensorName: String): A
12  def nbrvar[A](name: CNAME): A
13 }
```

- 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 operatori 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 2.1):

Gli elementi principali di questa interfaccia sono [CVAP22]:

- **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 [Cas20].

ScaFi fornisce alcuni operatori, detti anche *blocchi*, di alto livello già implementati tipici dei sistemi aggregati [CVAP22] (le immagini sono tratte da [Cas20]):

- **Leader election (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** fig. 2.10.

- **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**. Un gradiente è una variazione continua di una quantità rispetto a una direzione. In ambito matematico e fisico, indica la direzione e la rapidità con cui una grandezza scalare come temperatura, pressione o colore in una mappa variano nello spazio fig. 2.11.

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

$C[T](\text{sink: Boolean, value: } T, \text{acc}(T,T) \Rightarrow T)$

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** fig. 2.12.

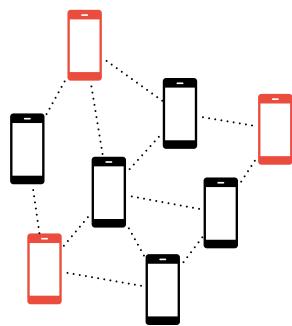


Figure 2.10: Leader election

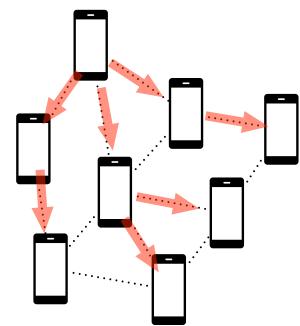


Figure 2.11: Gradient cast

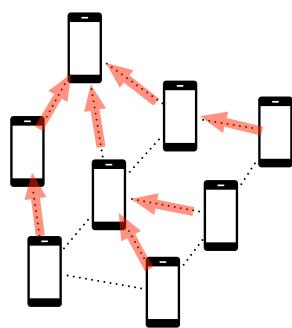


Figure 2.12: Collect cast

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*⁴, permettendo di concentrarsi solo sulla logica di

⁴Comportamenti di sciame/flotta

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. 2.13) [ACV24].

Macroswarm semplifica lo sviluppo di sistemi aggregati fornendo un insieme di blocchi di alto livello che permettono di definire comportamenti di “*stormo*” (come quello degli uccelli), comportamenti “*leader-follower*” (come nei branchi di elefanti), comportamenti di auto-organizzazione, cambiamento strutturale e formazione di *team* [ACV24]. I blocchi rappresentano le API per progettazione dei sistemi aggregati su sistemi fisici (i rettangoli bianchi in fig. 2.14 rappresentano i moduli principali). Questi blocchi rappresentano alcuni dei comportamenti essenziali, come vedremo nei prossimi paragrafi, per la ideazione di comportamenti più complessi la difficoltà rimane relativamente bassa poiché basterà comporre blocchi più semplici (come quelli già implementati) per ottenere il comportamento desiderato.

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 [ACV24]. Le modalità sono:

- **round-based**(fig. 2.15): è possibile eseguire l'attuazione definita al termine del prossimo round
- **long-standing**(fig. 2.16): 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. 2.14 [ACV24] il framework è composto da diversi moduli che a loro volta contengono diversi blocchi, ognuno con un compito

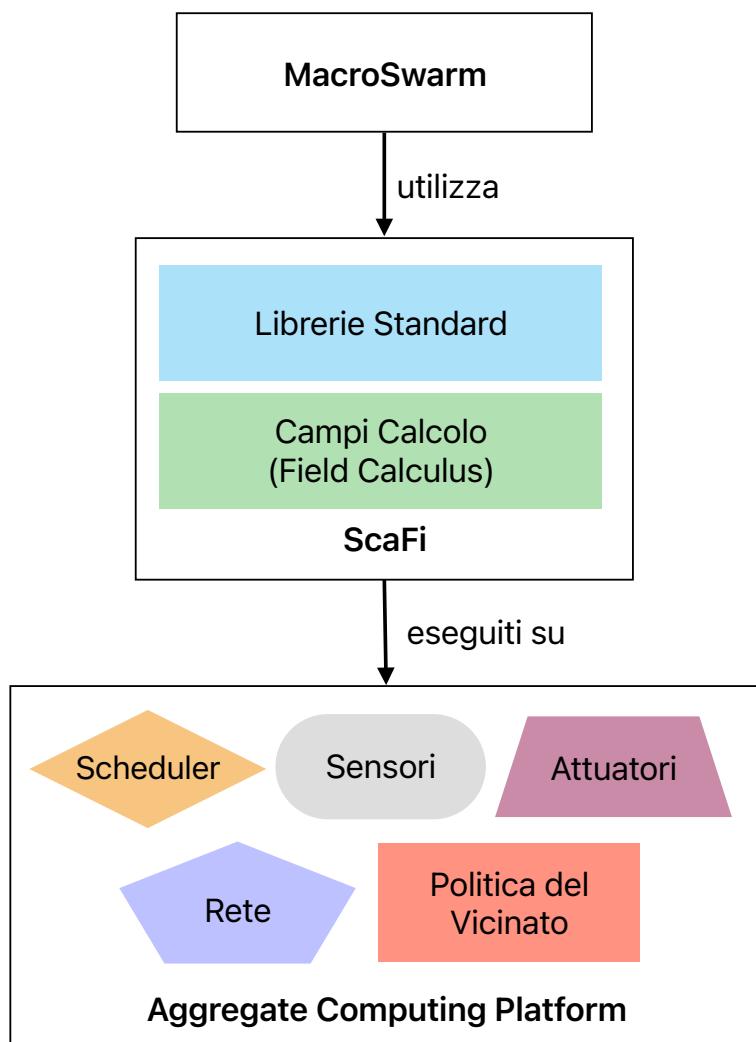


Figure 2.13: Architettura di Macroswarm

2.3. MACROSWARM

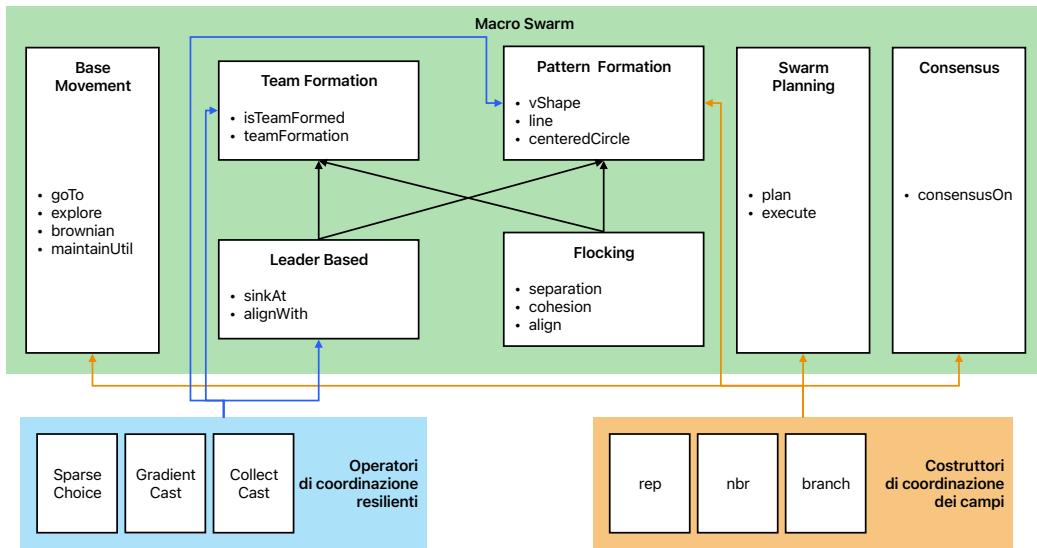


Figure 2.14: Struttura di Macroswarm e moduli interni

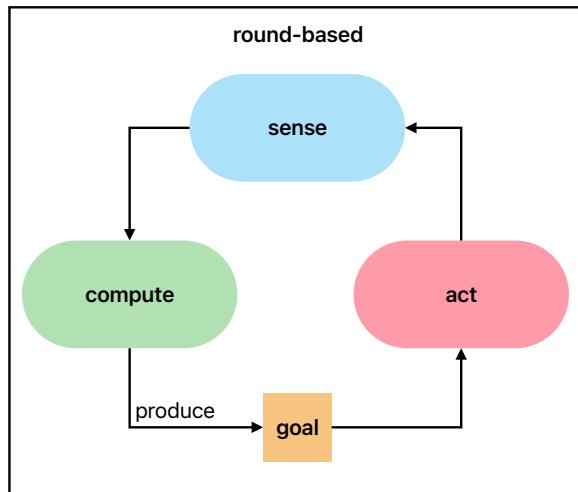


Figure 2.15: Modalità Round Based

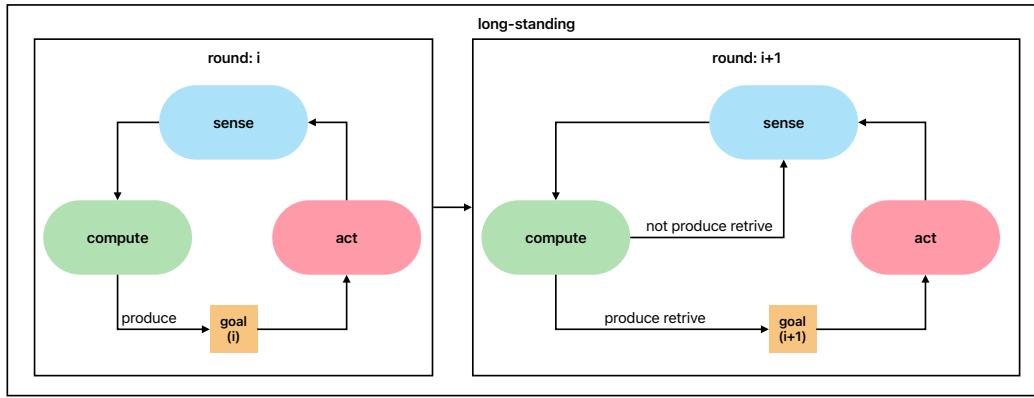


Figure 2.16: Modalità Long Standing

specifico, solo alcuni di questi blocchi sono stati utilizzati per la realizzazione del progetto e saranno descritti nei prossimi paragrafi.

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 (listing 2.2), alcuni di questi simulati in figura fig. 2.17:

- `goTo`: permette di definire un target (posizione assoluta) che gli agenti devono raggiungere

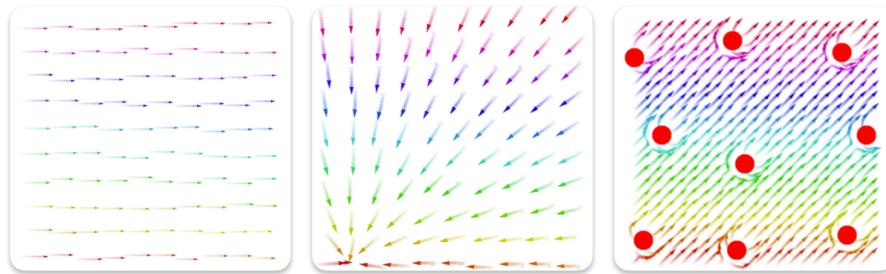


Figure 2.17: Simulazioni dei blocchi di movimento eseguita su Alchemist.

- **explore**: definito uno spazio rettangolare con i valori `minBound` e `maxBound`, il blocco produce vettori da assegnare agli agenti per esplorare l'intero spazio
- **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. 2.17 [ACV24] sono state simulate le funzioni `goTo`, `explore` e `obstacleAvoidance` su Alchemist, un simulatore di sistemi aggregati.

Listing 2.2: Funzioni del modulo di movimento

```

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

```

2.3. MACROSWARM

```
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” possiamo implementare un blocco nuovo nel seguente modo listing 2.3:

Listing 2.3: Comportamento complesso

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

La composizione di questi blocchi permette di creare comportamenti complessi in modo semplice e intuitivo andando semplicemente 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 al fine di raggiungere un obiettivo si è valutato che ci fosse la necessità di avere un nodo leader che si occupasse di coordinare gli altri nodi, proprio come succede nella natura nel caso dei branchi. 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 velocità 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.

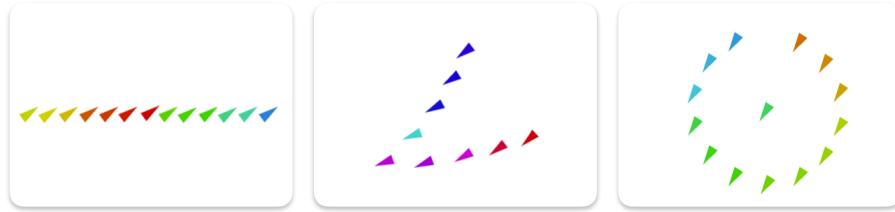


Figure 2.18: Simulazioni delle formazioni eseguite su Alchemist, in ordine: linea, formazione a V, cerchio.

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 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. 2.18 [ACV24] si possono muovere nello spazio e necessitano di un leader. Nel caso il leader vada a variare la sua velocità di movimento 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.



Figure 2.19: Robot Thymio

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 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 [Mobic] 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 [Moba]. Questo protocollo permette di gestire una rete con molti nodi a discapito 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)

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 2024 in quanto si tratta di 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*” [Mobic].

Noi ci concentreremo sull'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

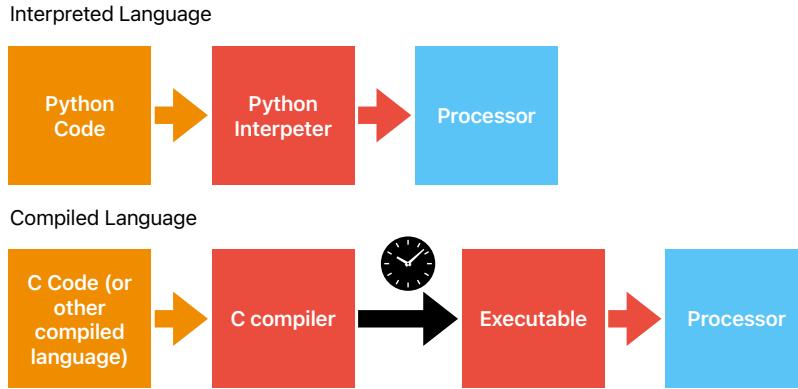


Figure 2.20: Interprete vs Compilatore

processore, occorre quindi compilare il sorgente ad ogni nuova modifica per poi re-eseguirlo. In Python, invece, il codice sorgente viene eseguito direttamente dal programma interprete, il quale esegue ogni comando riga per riga [Adv].

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

```

1 var state
2
3 callsub init # Inizializza il programma
4
5 sub init

```

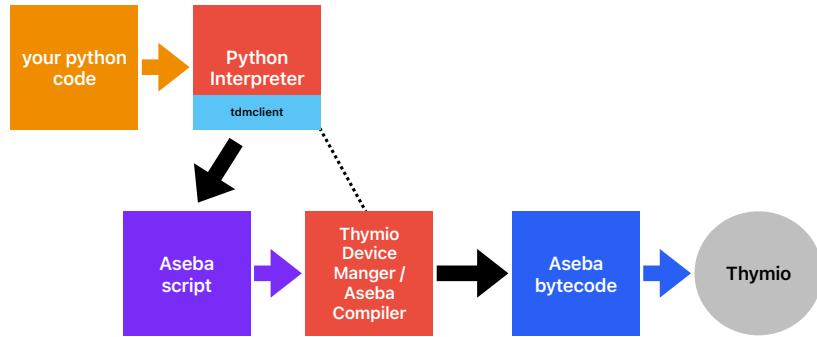


Figure 2.21: Tdmclient workflow

```

6      state = 0
7      call leds.bottom.left(0,0,32)
8      call leds.bottom.right(0,32,0)
9      call leds.top(32,0,0)
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. 2.21 [Mobb]. 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

```
1 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)

```
1 var x
2
3 onevent foo
4     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

```
1 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 artificiale fig. 2.22 [OPe]. 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

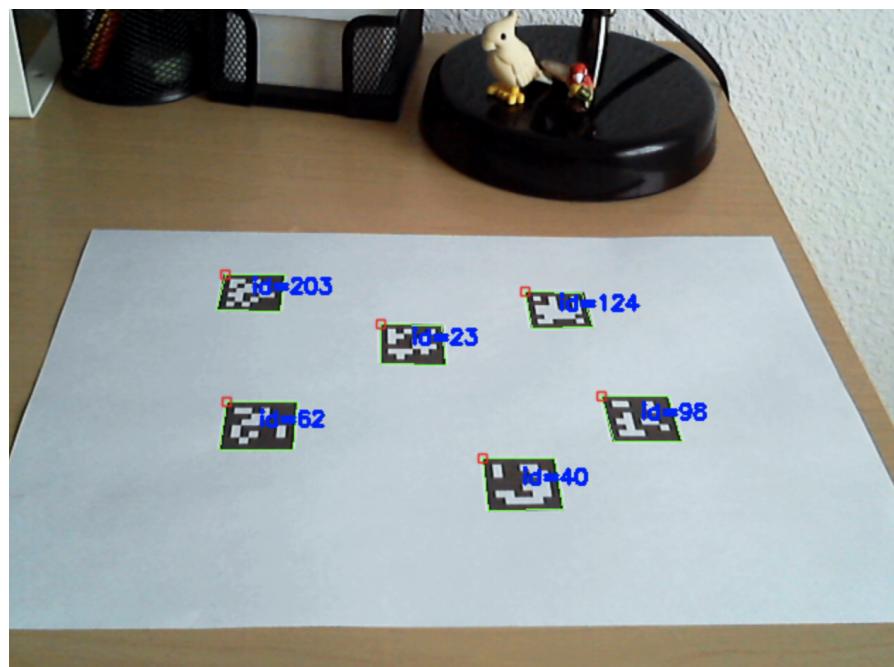


Figure 2.22: Aruco markers identificati da un algoritmo di visione artificiale

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.

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 (Thymio fig. 2.19.) e andando a gestire gli eventuali vincoli di compatibilità.

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) che ne permette il controllo mezzo protocollo WiFi o Bluetooth.



Figure 3.1: Robot Wave

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.
- Integrare robot Thymio nel sistema già esistente.

Requisiti non funzionali:

- Velocità nelle comunicazioni con i nuovi robot integrati.

3.3 Dominio

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.

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

Il server sviluppato si occupa di gestire la comunicazione con due clientfig. 4.1:

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

Come è possibile vedere in figura fig. 4.1 il programma aggregato utilizza una webcam per tracciare i robot, computate le posizioni e le direzioni dei robot, queste informazioni vengono elaborate dal programma aggregato ed al fine di completare

¹Le 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

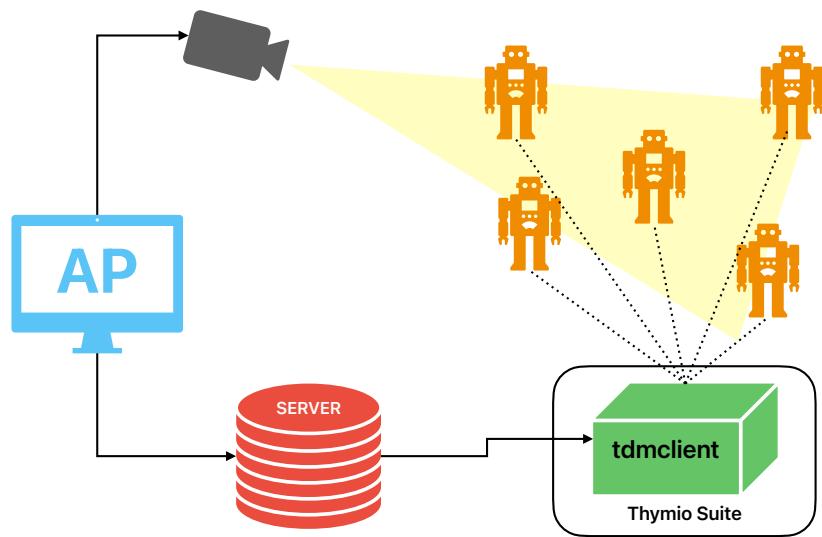


Figure 4.1: Architettura del server Flask

il task vengono inviati comandi per ogni Thymio al server Flask che, attraverso **tdmclient** si occupa di eseguire le attuazioni richieste.

Chapter 5

Implementazione

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

- webcam per il tracciamento dei Aruco Tag posizionati sopra ai robot
- macchina che esegue il sistema aggregato e che comunica con i robot, solitamente un computer
- collettivo di robot

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 una breve descrizione 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 5.1 che serve a poter definire i diversi programmi aggregati.

Listing 5.1: Classe AggregateIncarnation

```

1 import it.unibo.scafi.incarnations.BasicAbstractIncarnation
2
3 object AggregateIncarnation extends BasicAbstractIncarnation with
  BuildingBlocks

```

L’interfaccia `Environment` listing 5.2 rappresenta l’astrazione del mondo, essa rappresenta lo stato del mondo in un dato istante e va definita in base al contesto.

Listing 5.2: Interfaccia Environment

```

1 trait Environment[ID, Position, Info]:
2   def nodes: Set[ID] // insieme di nodi definiti dagli identificatori ID
3   def position(id: ID): Position // posizione di un certo nodo
4   def sensing(id: ID): Info // le informazioni che un certo nodo espone
5   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 lavora listing 5.3.

Listing 5.3: Definizione dell’ambiente

```

1 import it.unibo.core.DistanceEstimator.distance
2 import it.unibo.core.Environment
3 import it.unibo.demo.{ID, Info}
4 import it.unibo.utils.Position.{Position, *, given}
5
6 class DemoEnvironment(data: Map[ID, (Position, Info)], neighboursRadius:
7   Double)
8   extends Environment[ID, Position, Info]:
9
10  override def nodes: Set[ID] = data.keySet
11
12  override def position(id: ID): (Double, Double) = data(id)._1
13
14  override def sensing(id: ID): Info = data(id)._2
15
16  override def neighbors(id: ID): Set[ID] =
    data.filter { case (k, v) => data(id)._1.distance(v._1) <=
      neighboursRadius }.keys.toSet

```

5.1. IMPLEMENTAZIONE DEL SERVER FLASK

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.

5.1 Implementazione del server Flask

Lato Server, sono di rilevanza le implementazioni per rilevare i robot Thymio nella rete listing 5.4 e per ricevere, attraverso l'API listing 5.5 i comandi per controllarli listing 2.5.

Listing 5.4: Funzione di setup per rilevare i Thymio nella rete

```
1  async def setupThymios():
2      await notebook.start()
3      robots = await notebook.get_nodes()
4      global robotsMap
5      for i, robot in enumerate(robots):
6          physical_id = str(robot).replace("Node ", "")
7          robotsMap[physical_id] = robot
```

L'API è stata costruita sotto forma di richiesta HTTP (GET) con parametro un JSON contenente l'identificato fisico univoco del Thymio, la potenza da erogare nel motore sinistro e destro (nel caso la potenza sia negativa il motore andrà in senso inverso) listing 5.5.

Nel sorgente è possibile notare che il server è sviluppato per essere eseguito nella stessa macchina che ospita il programma aggregato, attuale è eseguito in hardcoded sulla porta 52000 di *localhost*.

Listing 5.5: API per il controllo dei Thymio

```
1
2 /thymio?json={"id": "44cdb758-cffc-42f3-ad5e-02262a80fcfc", "l": -50, "r": +50}
```

Listing 5.6: Implementazione API per inviare un comando di movimento al Thymio

```
1 @app.get('/thymio')
2 def getThymioParams():
3     assert request.method == 'GET'
4
5     # check that is only an params named json
6     if len(request.args) != 1 or 'json' not in request.args:
7         return "Error: only one parameter named json is allowed", 400
```

5.1. IMPLEMENTAZIONE DEL SERVER FLASK

```
8     encoded_json = request.args.get('json', default=None)
9     if encoded_json:
10         # decode the URL
11         decoded_json = unquote(encoded_json)
12         try:
13             command_data = json.loads(decoded_json)
14             physical_id_thymio = command_data.get("id")
15             left_motor = command_data.get("l")
16             right_motor = command_data.get("r")
17             moveThymio(robotsMap.get(physical_id_thymio), left_motor, right_motor)
18             )
19             return jsonify({"status": "ok", "message": "Data received", "ID":
20                             physical_id_thymio, "L": left_motor, "R": right_motor})
21         except json.JSONDecodeError:
22             return jsonify({"status": "error", "message": "Invalid JSON format"}),
23             400
24
25     def moveThymio(robot, left, right):
26         aw(robot.lock())
27         v = {
28             "motor.left.target": [int(left)],
29             "motor.right.target": [int(right)],
30         }
31         aw(robot.set_variables(v))
32         aw(robot.unlock())
```

Per completezza è stato sviluppata una semplice interfaccia web per poter inviare i comandi ai robot Thymio fig. 5.1 e verificare che tutti i robot siano correttamente connessi alla rete.

È presente anche un pulsante per fermare tutti i robot nella rete nel caso si verifichi un'anomalia.

Lato programma aggregato, invece, troviamo l'implementazione di una classe che nasconde ogni tipo di dettaglio implementativo per il controllo dei robot, sia Wave che Thymio listing 5.7. Nel codice è presente solo l'implementazione di `spinRight()` a fine dimostrativo ma gli altri comandi sono implementati in modo simile.

Listing 5.7: Classe per il controllo dei robot

```
1 package it.unibo.demo.robot
2
3 import java.net.URLEncoder
4 import upickle.default.ReadWriter
5 trait Robot:
```

5.1. IMPLEMENTAZIONE DEL SERVER FLASK

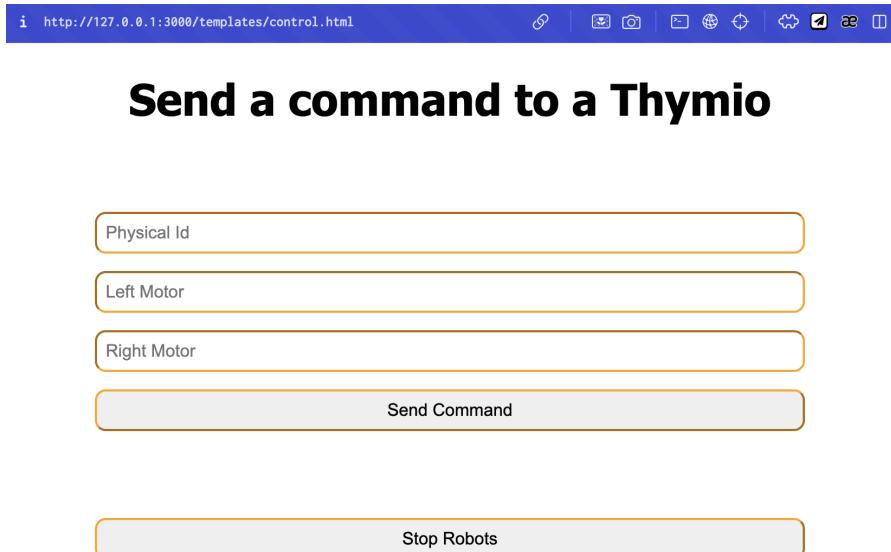


Figure 5.1: Interfaccia web per il controllo dei robot Thymio

```
6  def id: Int
7  def spinRight(): Unit
8  def spinLeft(): Unit
9  def forward(): Unit
10 def backward(): Unit
11 def nop(): Unit
12 def intensities(left: Double, right: Double): Unit
13
14 case class WaveRobot(ip: String, val id: Int) extends Robot derives ReadWriter:
15
16     private var lastCommandWasNoOp = false
17     def spinRight(): Unit =
18         requests.get(url = s"http://$ip/js?json=${Command(0.20, -0.20).toJson()}")
19         lastCommandWasNoOp = false
20     ...
21     override def nop(): Unit =
22         if !lastCommandWasNoOp then requests.get(url = s"http://$ip/js?json=${Command
23             (0, 0).toJson()}")
24         else ()
25         lastCommandWasNoOp = true
26     private class Command(left: Double, right: Double):
27         def toJson: String = URLEncoder.encode(s"""{"T":1, "L":$left, "R":$right}""")
28
29 /**
30 * @param physicalId is the physical Id
31 * @param id is the arUco Id
```

5.1. IMPLEMENTAZIONE DEL SERVER FLASK

```
32  /*
33   * case class ThymioRobot(val physicalId: String, val id: Int, val name: String)
34   * extends Robot derives ReadWriter:
35   * override def spinRight(): Unit =
36   *   requests.get(url = s"http://localhost:52000/thymio?json=${Command(physicalId,
37   *     +50, -50).toJson}")
38   * ...
39   * override def nop(): Unit =
40   *   requests.get(url = s"http://localhost:52000/thymio?json=${Command(physicalId,
41   *     0, 0).toJson}")
42   * override def intensities(left: Double, right: Double): Unit = {}
43
44   * private class Command(physicalId: String, left: Double, right: Double):
45   *   def toJson: String = URLEncoder.encode(s"""{"id": "$physicalId", "l": $left, "r": $right}""")
46
```

Chapter 6

Validazione

Di seguito si riportano alcuni esempi di algoritmi aggregati applicati ai robot Wave e Thymio nello stesso ambiente per dimostrare che i requisiti sono stati soddisfatti mantenendo flessibilità, autonomia e eterogeneità del sistema.

Copia rotazione del Leader Programma Aggregato per far ruotare i robot del sistema seguendo la direzione del leader listing 6.1 fig. 6.1.

Listing 6.1: Programma aggregato per far ruotare i robot del sistema seguendo la direzione del leader

```
1 class FollowTheLeaderRotating(private val leaderId: Int, private val timeInterval: Long = 10_000L) extends BaseDemo:
2   private val directions = List(
3     Actuation.Rotation(normalize(0.5, 0.5)),
4     Actuation.Rotation(normalize(0.5, -0.5)),
5     Actuation.Rotation(normalize(-0.5, -0.5)),
6     Actuation.Rotation(normalize(-0.5, 0.5)),
7   )
8
9   override def main(): Actuation =
10   executeOnEach(10_000L, (NoOp, 0)): (_, prevIndex) =>
11     (directions(prevIndex % directions.size), prevIndex + 1)
12     ._1
13
14   private def executeOnEach[T](deltaTime: Long, default: T)(logic: T => T): T =
15     val currentTime = System.currentTimeMillis()
16     val result = rep((currentTime, default)): (prevTime, prevValue) =>
17       if currentTime - prevTime > deltaTime then (currentTime, logic(prevValue))
18       else (prevTime, prevValue)
19     result._2
```

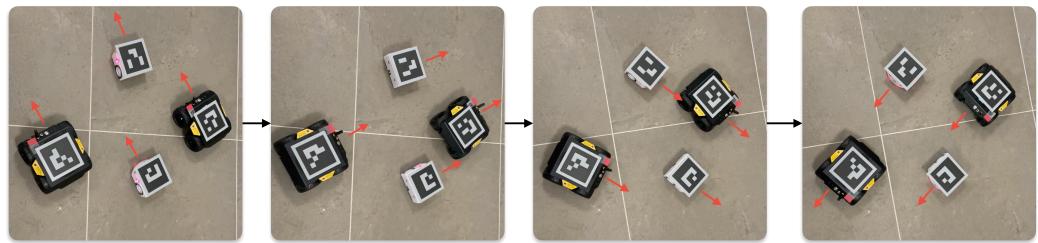


Figure 6.1: Simulazione del programma aggregato per far ruotare i robot del sistema seguendo la direzione del leader .



Figure 6.2: Simulazione del programma aggregato per allineare i robot.

Formazione a “linea” Programma aggregato che si occupa di allineare i robot del sistema al fine di creare una linea retta ?? fig. 6.3 fig. 6.2.

Formazione a “cerchio” Programma aggregato che si occupa di definire un leader e produrre una formazione circolare intorno ad esso listing 6.2. È possibile notare dalla immagine fig. 6.4 come il programma si comporti correttamente anche in situazioni anomale come la rimozione o l'inserimento di nuovo robot nell'ambiente.

Listing 6.2: Programma aggregato allineare i robot

```

1  class CircleFormation(radius: Double, leaderSelected: Int, stabilityThreshold:
2    Double)
3    extends ShapeFormation(leaderSelected, stabilityThreshold):
4    override def calculateSuggestion(ordered: List[(Int, (Double, Double))]): Map[
5      Int, (Double, Double)] =
6      val division = (math.Pi * 2) / ordered.size
7      val precomputedAngels = ordered.indices.map(i => division * (i + 1))
8      var availableDevices = ordered

```

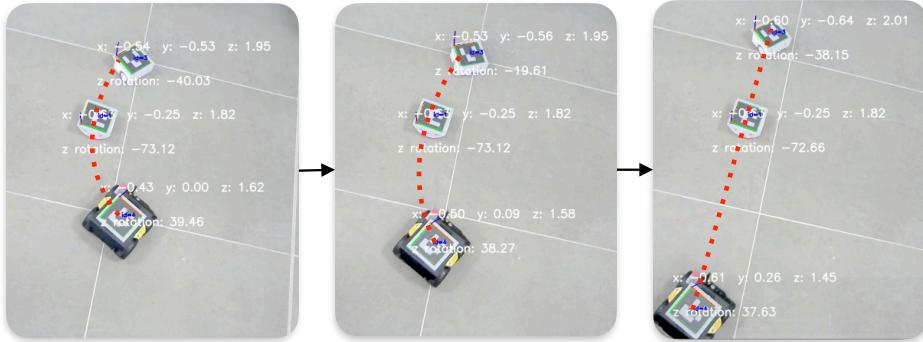


Figure 6.3: Simulazione del programma aggregato per allineare i robot dalla visualizzazione del programma di tracking dei Aruco Tag (la linea rossa è stata inserita in post produzione).

```

7   precomputedAngles
8     .map: angle =>
9       val candidate = availableDevices
10      .map:
11        case (id, (xPos, yPos)) =>
12          val newPosX @ (newXpos, newYpos) = (math.sin(angle) * radius + xPos,
13                                              math.cos(angle) * radius + yPos)
14          (id, math.sqrt((newXpos * newXpos) + (newYpos * newYpos)), newPos)
15          .minBy(_._2)
16          availableDevices = removeDeviceFromId(candidate._1, availableDevices)
17          candidate._1 -> candidate._3
           .toMap

```

6.1 File di configurazione

Per gestire l'inserimento o la rimozione di nuovi robot nel sistema in modo dinamico si è scelto di utilizzare i file di configurazione listing 6.3 listing 6.4.

Per la gestione dei file di configurazione all'interno del progetto Scala è stata utilizzata la libreria **uPickle**. È una libreria performante che permette un parsing veloce ed intuitivo, inoltre non ha problemi di dipendenze quindi può essere utilizzata in qualsiasi progetto Scala senza produrre conflitti [uPi].

6.1. FILE DI CONFIGURAZIONE

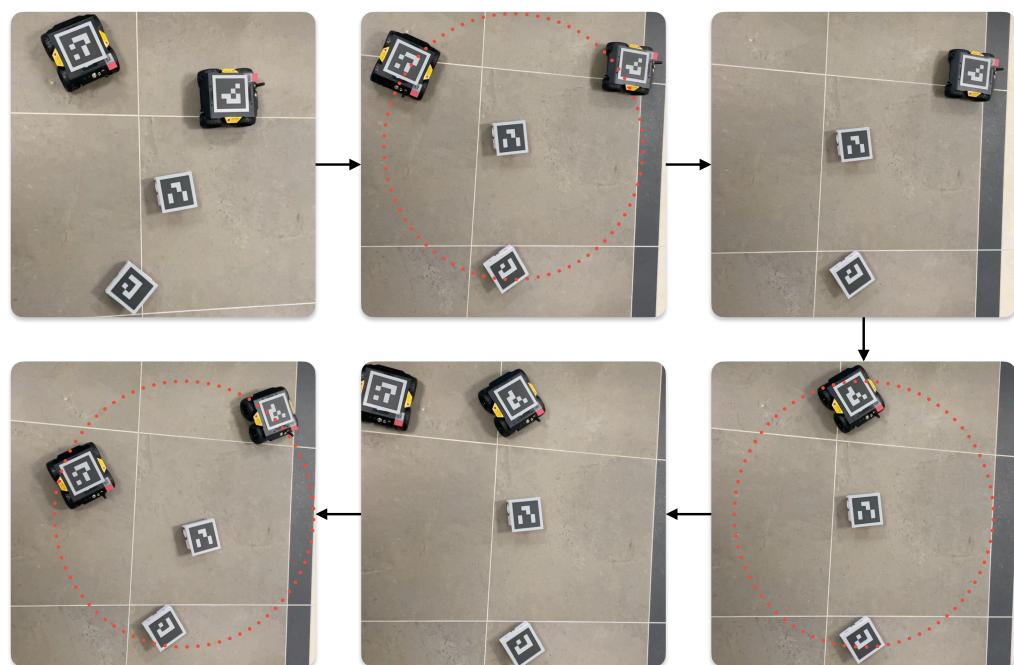


Figure 6.4: Simulazione del programma aggregato per produrre un cerchio intorno ad un leader.

6.1. FILE DI CONFIGURAZIONE

Listing 6.3: File di configurazione per i robot Thymio

```
1 [
2   {
3     "physicalId": "44cdb758-cffc-42f3-ad5e-02262a80fcfc",
4     "id": 1,
5     "name": "beer"
6   },
7   {
8     "physicalId": "2796a8f2-a495-4901-8402-d6f06f22d7cb",
9     "id": 3,
10    "name": "rosenblueth"
11  }
12 ]
```

Listing 6.4: File di configurazione per i robot Wave

```
1 [
2   {
3     "ip": "192.168.8.12",
4     "id": 4
5   }
6 ]
```

Questi file di configurazione vengono caricati utilizzando la libreria uPickle [uPi] listing 6.5.

Listing 6.5: Lettura dei file di configurazione

```
1 private val jsonString =
2   os.read(os.pwd / "src" / "main" / "scala" / "it" / "unibo" / "demo" / "
3     configuration" / "waveRobot.json")
4 private val waveData = ujson.read(jsonString)
5 private val robots = read[Seq[WaveRobot]](waveData).toList
6 private val waveList = robots.map(_.id)
7
7 private val thymioRobotJson =
8   os.read(os.pwd / "src" / "main" / "scala" / "it" / "unibo" / "demo" / "
9     configuration" / "thymioRobot.json")
10 private val thymioData = ujson.read(thymioRobotJson)
11 private val thymioRobots = read[Seq[ThymioRobot]](thymioData).toList
12 private val thymioList = thymioRobots.map(_.id)
13
13 private val list = thymioRobots ++ robots
```

Chapter 7

Conclusione

In questa tesi abbiamo dimostrato come sia possibile e semplice andare ad espandere il dominio del sistema che si vuole andare a controllare utilizzando un sistema aggregato. Nonostante la comunicazione con i robot Thymio sia stata gestita in linguaggio

Bibliography

- [ACV24] Gianluca Aguzzi, Roberto Casadei, and Mirko Viroli. Macroswarm: A field-based compositional framework for swarm programming, 2024.
- [Adv] Robot Advance. Using the educational robot Thymio with Python — robot-advance.com. <https://www.robot-advance.com/EN/actualite-python-with-thymio-complete-guide-228.htm>. [Accessed 01-11-2024].
- [AVD⁺19] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *ACM Transactions on Computational Logic*, 20(1):1–55, January 2019.
- [BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, September 2015.
- [Cas20] Roberto Casadei. Engineering self-adaptive collective processes for cyber-physical ecosystems. 2020.
- [Cas23] Roberto Casadei. Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. *ACM Computing Surveys*, 55(13s):1–37, July 2023.
- [CAV21] Roberto Casadei, Gianluca Aguzzi, and Mirko Viroli. A programming approach to collective autonomy. *Journal of Sensor and Actuator Networks*, 10(2):27, April 2021.
- [CVAP22] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafì: A scala dsl and toolkit for aggregate programming. *SoftwareX*, 20:101248, December 2022.

BIBLIOGRAPHY

- [Hun18] John Hunt. *Functional Programming in Scala*, page 229–240. Springer International Publishing, 2018.
- [Moba] Mobsya. Setting Wireless Thymio Network - Thymio & Aseba — aseba.wikidot.com. [http://aseba.wikidot.com/en:thymiosettingwireless#:~:text=How%20does%20a%20Wireless%20Thymio,it%20\(unlike%20e.g.%20Bluetooth\).](http://aseba.wikidot.com/en:thymiosettingwireless#:~:text=How%20does%20a%20Wireless%20Thymio,it%20(unlike%20e.g.%20Bluetooth).) [Accessed 07-11-2024].
- [Mobb] Mobsya. tdmclient — pypi.org. <https://pypi.org/project/tdmclient/>. [Accessed 07-11-2024].
- [Mobc] Mobsya. Thymio device manager 2014; aseba 1.7-alpha documentation — mobsya.github.io. <https://mobsya.github.io/aseba/thymio-device-manager.html>. [Accessed 20-10-2024].
- [OPe] OPenCV. Opencv: Detection of aruco markers — docs.opencv.org. https://docs.opencv.org/3.2.0/d5/dae/tutorial_aruco_detection.html. [Accessed 25-10-2024].
- [PBV17] Danilo Pianini, Jacob Beal, and Mirko Viroli. Practical aggregate programming with protelis. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, page 391–392. IEEE, September 2017.
- [TADT22] Lorenzo Testa, Giorgio Audrito, Ferruccio Damiani, and Gianluca Torta. Aggregate processes as distributed adaptive services for the industrial internet of things. *Pervasive and Mobile Computing*, 85:101658, September 2022.
- [uPi] uPickle. upickle 4.0.2 — com-lihaoyi.github.io. <https://com-lihaoyi.github.io/upickle/>. [Accessed 30-10-2024].

Acknowledgements

Optional. Max 1 page.