Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Una piattaforma client-server universale per Aggregate Computing

Relatore:
Chiar.mo Mirko Viroli
Correlatori:
Danilo Pianini
Roberto Casadei

Presentata da:
Loris Cangini

Sessione: Marzo 2020
Anno Accademico: 2019/2020

Abstract

Negli scorsi anni sono stati svolti studi e sviluppati diversi linguaggi e framework per diffondere e rendere utilizzabile l'aggregate computing. L'aggregate computing è un approccio emergente utilizzato per progettare sistemi di coordinazione complessi e distribuiti. Il tutto si basa sul *Field Calculus*, un modello di programmazione universale utile per specificare computazioni aggregate tramite composizione di comportamenti [10].

Tra i linguaggi ed i framework sviluppati figurano Protelis e Scafi, i quali, utilizzando approcci diversi tra di loro, forniscono implementazioni concrete del Field Calculus, permettendo di definire programmi aggregati e di mandarli in esecuzione. Avendo a disposizione diversi framework per specificare le computazioni da svolgere, si è rivelato necessario avere una piattaforma sulla quale poter eseguire tali programmi.

La tesi verte sulla progettazione e conseguente implementazione di un sistema basato su architettura client-server che permetta l'esecuzione di programmi aggregati su una rete logica di dispositivi. È prevista la possibilità di utilizzare dispositivi virtualizzati e/o reali per formare la rete che eseguirà il programma (non necessariamente limitandosi ad una tipologia per esperimento). È prevista anche la possibilità per un client di entrare in modalità "lightweight", nella quale non sarà più quest'ultimo ad eseguire, ma sarà il Server a farsi carico della sua parte, indicando al client solamente i risultati delle computazioni. Indipendentemente dalla modalità di esecuzione, il sistema è nativamente compatibile con Scafi e Protelis ed è aperto a nuove implementazioni, permettendo quindi l'esecuzione di programmi aggregati indipendentemente dal linguaggio o framework utilizzato per definirli ed eseguirli.

L'intero progetto è stato sviluppato adottando una metodologia *test-driven*. La compatibilità del sistema con Scafi e Protelis è stata verificata testandolo con alcuni programmi di esempio forniti dagli autori dei framework stessi. Tutti i test sono stati continuamente verificati attuando un processo di continuous integration, permettendo così di individuare facilmente eventuali problematiche durante lo sviluppo.

Indice

Elenco delle figure

Capitolo 1

Contesto

In seguito verranno introdotti alcuni concetti per meglio comprendere ciò che è stato fatto. Più precisamente, verranno trattati:

- **Aggregate Computing**: i passaggi storici avvenuti per arrivare al ***Field Calculus***, il modello base per poter specificare computazioni aggregate.
- **Protelis**: linguaggio per specificare computazioni aggregate, basato su Java.
- **Scafi**: framework per computazioni aggregate basato su Scala.
- **Programmazione aggregata**: architettura per supportare la creazione ed il mantenimento di sistemi complessi.

1.1 Aggregate Computing

La computazione aggregata è un approccio emergente, sviluppato a partire dai modelli di coordinazione, con l'idea di comporre funzionalmente comportamenti per ottenere comportamenti complessi e resilienti in reti dinamiche [10]. La computazione aggregata si basa sul *Field Calculus*, un modello per specificare e comporre i comportamenti. Al di sopra è stato costruito un modello a strati per permettere di progettare la coordinazione in sistemi distribuiti complessi.

Per arrivare alla computazione aggregata sono stati necessari alcuni passaggi storici, partendo dai modelli di coordinazione di sistemi paralleli, fino ad arrivare al Field Calculus (fig. 1.1).

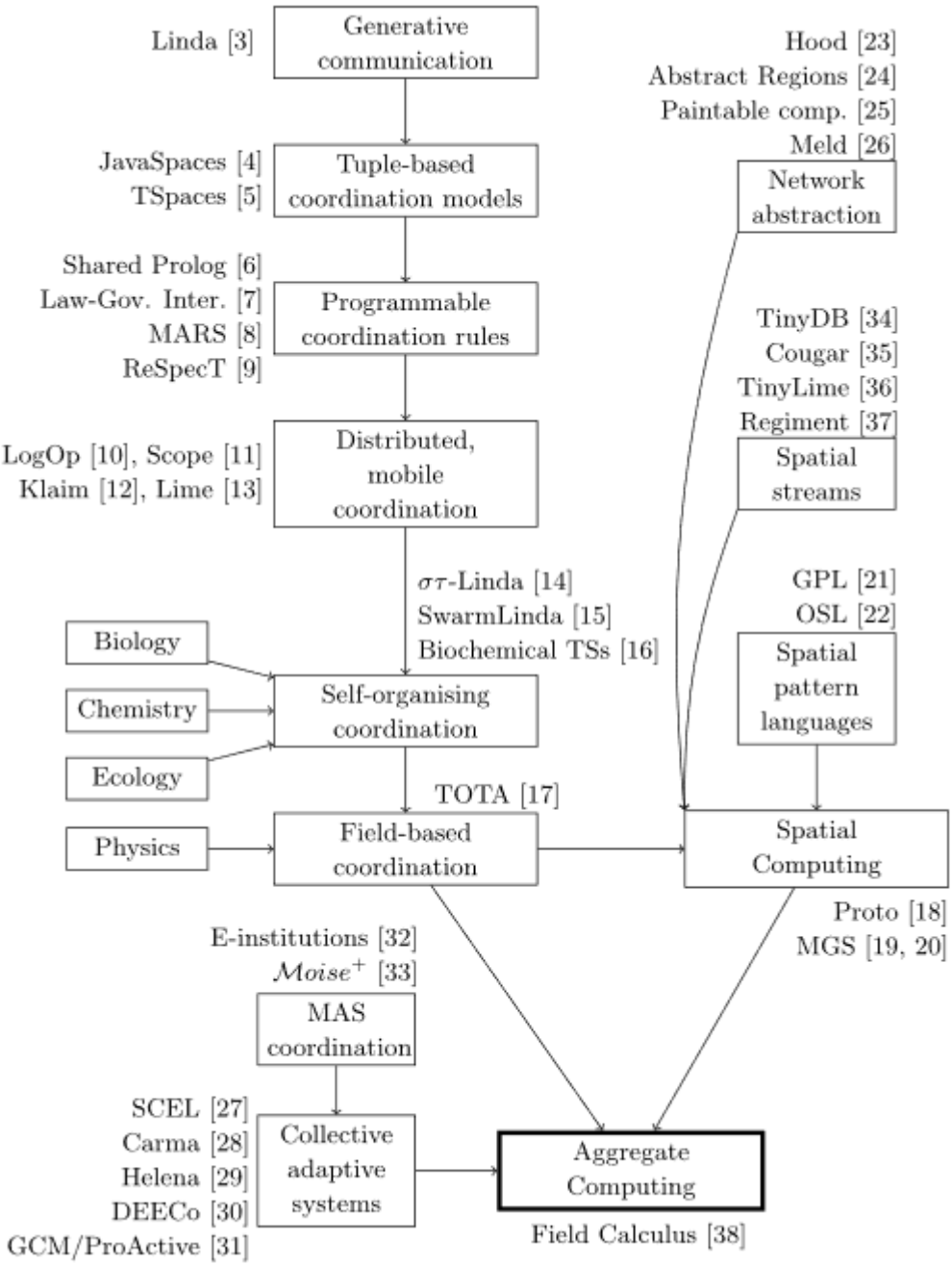


Figura 1.1: Da coordinazione a Field Calculus [10]

1.1.1 Comunicazione generativa

I modelli di coordinazione sono basati sull'idea che l'interazione tra diversi sistemi autonomi ed indipendenti possa essere concepita e progettata come uno spazio ortogonale alla computazione [10]. I processi di un sistema parallelo hanno a disposizione uno **spazio dei dati comune**, dove tutti possono leggere e scrivere informazioni.

1.1.2 Coordinazione basata su tuple

Quando i dati scambiati assumono la forma di collezioni di dati potenzialmente eterogenei, si parla di coordinazione basata su tuple [14]. Per ritrovare questi dati dallo spazio dei dati comune, detto **spazio delle tuple**, i processi eseguono interrogazioni fornendo un template da ricercare. Un template è una rappresentazione parziale della struttura dei dati da ricercare. Le conseguenze di questo approccio sono principalmente due:

- disaccoppiamento nella comunicazione: per comunicare non è necessaria alcuna informazione riguardo il mittente, allo spazio delle tuple ed al tempo di inserimento.

- è possibile coordinarsi anche in ambienti con informazioni scarse o imprecise, in quanto la ricerca viene effettuata tramite rappresentazioni parziali.

1.1.3 Regole di coordinazione programmabili

L'utilizzo di spazi di dati condivisi utilizzati per la coordinazione viene promossa maggiormente con la nascita di modelli a spazi di tuple logici. Gli agenti possono coordinarsi utilizzando tuple create tramite logica del primo ordine. Gli spazi di tuple possono essere programmati come teorie nella stessa logica. Si fornisce **intelligenza agli spazi condivisi**, cioè la capacità, tramite una logica applicativa, di manipolare i dati in tali spazi e cambiare le loro modalità di accesso. Diversi approcci permettono di programmare lo spazio delle tuple per definire regole di coordinazione.

1.1.4 Coordinazione distribuita

Tutti gli approcci visti finora non gestiscono esplicitamente la distribuzione.

Anche gli **spazi di tuple** possono essere **distribuiti** nell'ambiente, permettendo l'utilizzo di astrazioni di coordinazione distribuite [11]. Alcuni middleware gestiscono anche la mobilità e la locazione, permettendo anche l'utilizzo di topologie dinamiche, mettendo così le basi per modelli di coordinazione in sistemi pervasivi.

1.1.5 Coordinazione auto-organizzante

Vanno risolte diverse problematiche poter creare sistemi di coordinazione in sistemi complessi:

- apertura: l'ambiente non è prevedibile, così come le interazioni ed i fallimenti che avverranno
- scalabilità: potenzialmente, numeri enormi di agenti e di astrazioni di comunicazione andranno gestiti
- adattività: vanno captati gli eventi e bisogna reagire ad essi.

Per poter gestire i problemi sopracitati è necessario ricorrere ad approcci di coordinazione auto-organizzanti, nei quali le astrazioni di coordinazione gestiscono solamente le interazioni locali, così che **pattern di comunicazione globali** e robusti possano **emergere** [10, 2].

Tipicamente, modelli di coordinazione di questo tipo traggono ispirazione dai sistemi naturali (specialmente biologici), cercando di riutilizzarne i meccanismi alla base.

1.1.6 Sistemi Multi-Agente (MAS)

La coordinazione nei sistemi multi-agente ha un ruolo fondamentale, in quanto bisogna fare in modo che diversi agenti, per loro natura autonomi e potenzialmente con goal locali in conflitto tra di loro, cooperino per raggiungere un obiettivo globale.

La ricerca sui MAS ha anche riconosciuto l'importanza dell'organizzazione nel realizzare comportamenti di sistema [5].

1.1.7 Sistemi Adattivi Collettivi (CAS)

I concetti fondamentali in questo contesto sono la decentralizzazione del controllo, operazioni asincrone ed interazioni opportunistiche. In questa branca di ricerca, **gruppi di dispositivi** vengono considerati **astrazioni di primo ordine**, supportando interazioni con altri gruppi astraendosi dai dettagli di più basso livello [9].

1.1.8 Coordinazione basata sui campi

Traendo spunto dalla fisica, viene definito il **campo di coordinazione** come un astrazione dell'ambiente reale per gestire l'auto-organizzazione di agenti mobili in ambienti complessi. Gli agenti possono percepire il valore del campo dal punto in cui sono per navigare nell'ambiente.

1.1.9 Computazione spaziale

Lo scopo è quello di costruire sistemi distribuiti intelligenti attraverso astrazioni di più alto livello su sistemi adattivi collettivi spaziali. I vari approcci in questo campo possono essere raggruppati in:

- Semplificazione della programmazione astraendosi dai dispositivi singoli.
- Linguaggi e pattern spaziali.
- Strumenti per raccogliere informazioni su regioni dello spazio/tempo.
- modelli computazionali per manipolare dati distribuiti sia nello spazio che nel tempo.

Combinando tecniche provenienti dai vari approcci e generalizzando, è stato proposto il **Field Calculus** come modello fondamentale per la computazione aggregata.

1.1.10 Field Calculus

Proposto come un linguaggio con gli ingredienti chiave per fare uso di **campi computazionali** [10]. Si basa sull'idea di specificare il comportamento di un insieme di dispositivi, con relazioni di vicinanza che indicano con chi un certo individuo può comunicare direttamente. Un concetto fondamentale di questo approccio è che le specifiche possono essere interpretate sia localmente che globalmente [10].

Localmente può essere visto come la descrizione della computazione di un dispositivo. Un dispositivo computa iterativamente in round asincroni, ricevendo messaggi da vicini, percependo l'ambiente, salvando lo stato interno della computazione e mandando messaggi ai vicini. Globalmente, un'espressione field calculus specifica il campo computazionale che associa ogni round di ogni dispositivo al valore che l'espressione assume in quel dato momento.

Definizione del linguaggio Come mostrato dalla grammatica in figura 1.2, un programma P in Field Calculus è una sequenza di definizioni di funzioni, seguite dall'espressione principale e, che a sua volta può essere formata da:

- una variabile x
- un valore v
  - locale l
  - proveniente dai vicini ϕ
- una funzione $f(\overline{e})$

  - dichiarata dall'utente d
  - preesistente b

- una ramificazione $if(e)\{e\}\{e\}$
- $nbr\{e\}$, che costruisce un mapping tra i vicini ed il loro valore più recente della valutazione di e.
- $rep(e)\{(x) => e\}$, che modella l'evoluzione dello stato nel tempo.

P	::=	$\overline{\texttt{F}}$ e	program
F	::=	def d($\overline{\texttt{x}}$) {e}	function declaration
e	::=	x | v | f($\overline{\texttt{e}}$) | if(e){e}{e} | nbr{e} | rep(e){(x)=>e}	expression
f	::=	d | b	function name
v	::=	ℓ | ϕ	value
ℓ	::=	c($\overline{\ell}$)	local value
ϕ	::=	$\overline{\delta} \mapsto \overline{\ell}$	neighbouring field value

Figura 1.2: Grammatica del Field Calculus [10]

Nbr e *rep* sono costrutti speciali, che richiedono la scambi messaggi tra dispositivi e nel tempo. Si assume che tali costrutti siano supportati da un meccanismo di gestione dei messaggi detto allineamento, che garantisce il corretto scambio di messaggi per portare a termine le operazioni.

Proprietà

Alcune proprietà fondamentali sono state isolate e studiate. Tra di esse l' **auto-stabilizzazione** garantisce che la valutazione di un programma dato un input costante converge ad un valore limite in ogni dispositivo in tempo finito. Tale valore limite dipende unicamente dagli input [10]. Applicando questa proprietà ad un sistema dinamico si garantisce che, qualora l'input cambi, l'output reagisce di conseguenza senza essere influenzato da valori passati.

Diversi frammenti auto-stabilizzanti sono stati proposti nel tempo. Inizialmente si erano ottenuti tramite un operatore *spreading*, funzionante su valori aggiornati da una funzione di diffusione [10]. Questo approccio non permette l'utilizzo di espressioni *rep* e *nbr* esplicite.

Un frammento auto-stabilizzante più grande è stato introdotto successivamente limitando *rep* a funzionare su tre pattern. Questi pattern corrispondono a tre blocchi:

- **G**: stima di distanza
- **C**: aggregazione di valori
- **T**: evoluzione temporale

Nozioni di equivalenza è sostituibilità per programmi auto-stabilizzanti sono state introdotte, permettendo ottimizzazioni di programmi sostituendo funzionalità con altre equivalenti ma con migliori performance. Questa relazione di equivalenza introduce un terzo punto di vista ai due (locale e globale) introdotti precedentemente, astraendo dalle caratteristiche transitorie ed isolando le relazioni tra input ed output.

Un quarto punto di vista continuo può essere analizzato. Se la densità di dispositivi in un'area aumenta, gli output potrebbero convergere ad un limite continuo. Programmi con tale proprietà sono detti **consistenti** ed hanno un'interpretazione continua.

Higher-order Field Calculus

L'higher-order Field Calculus (HFC) è un estensione del Field Calculus con funzioni come astrazioni di primo ordine [10, 1].

Lo scopo è quello di permettere ai programmatori di utilizzare funzioni come qualsiasi altro valore, permettendo di muovere, inserire ed eseguire codice dinamicamente.

1.2 Protelis

Protelis[1] è un linguaggio utile a definire sistemi resilienti formati da dispositivi eterogenei [7, 6]. Include una sintassi concreta per definire programmi aggregati, un parser, un interprete, una macchina virtuale, astrazioni per i dispositivi, astrazioni per la comunicazione. Il parser traduce il codice Protelis in una rappresentazione HFC valida. Il programma tradotto, assieme al contesto di esecuzione, viene fornita alla macchina virtuale che esegue il programma tramite l'interprete [10].

Il contesto definisce le interfacce verso il sistema operativo, includendo le astrazione delle capacità dei dispositivi e del sistema di comunicazione.

L'intera infrastruttura è scritta in Java, quindi funzionante sulla Java Virtual Machine. La traduzione da sintassi Protelis ad Higher-Order Field Calculus è stata implementata usando Xtext.

[1] `https://protelis.github.io/`

La sintassi è stata progettata per essere quanto più simile possibile a C, Java e Phyton, riducendo così la curva di apprendimento necessaria alla maggior parte degli sviluppatori.

1.2.1 Utilizzo

Definizione delle dipendenze

Per poter utilizzare Protelis è necessario definire, utilizzando un qualsiasi build tool, la dipendenza al linguaggio.

Di seguito vengono mostrate le versioni per Gradle ed SBT:

```
compile 'org.protelis:protelis:13.1.1'
```

Listato 1.1: Dipendenza Gradle per Protelis [7]

```
libraryDependencies += "org.protelis" % "protelis" % "13.1.1"
```

Listato 1.2: Dipendenza SBT per Protelis

Definizione del programma aggregato

Per definire programmi, è necessario scrivere dei sorgenti utilizzando il linguaggio Protelis.

```
/*obbligatorio definire il nome del modulo*/
module leader

/*Eventuali import*/
import protelis:state:time

/*Il programma vero e proprio*/
let leader = env.has("leader");
if(leader){
    self.announce("Sono␣il␣leader");
    true;
} else {
    false;
}
```

```
if(anyHood(nbr(leader))) {
    self.announce("Sono␣il␣vicino␣del␣leader")
    true;
} else {
    false;
}
```

Listato 1.3: Esempio di un programma Protelis

Definizione del contesto

Per poter eseguire è poi necessario definire il contesto. Tipicamente viene creato a partire dalla classe astratta `AbstractExecutionContext`, contenente le implementazioni delle funzionalità più comuni.

```
class MyContext() extends AbstractExecutionContext<MyContext>(
    executionEnvironment,
    networkManager
) {
    /*funzioni da implementare per definire il contesto*/
    override fun instance(): MyContext = MyContext()
    override fun nextRandomDouble(): Double = /**/
    override fun getDeviceUID(): DeviceUID = /**/
    override fun getCurentTime(): Number = /**/

    /*Ulteriori funzioni da rendere disponibili*/
    fun announce(text: String) = println(text)
}
```

Listato 1.4: Definizione del contesto

Come si può notare dal listato 1.4, per definire il contesto sono necessari altri due oggetti:

- `ExecutionEnvironment`: utile per gestire le variabili di ambiente (come *leader* nell'esempio 1.3). È disponibile un implementazione basilare chiamata `SimpleExecutionEnvironment`.

- `NetworkManager`: gestore delle comunicazioni tra dispositivi appartenenti alla rete.

Esecuzione

```
/*Parsing del modulo Protelis*/
val program = ProtelisLoader.parse("leader")

/*Creazione del contesto*/
val context = MyContext()

/*Creazione della macchina virtuale*/
val vm = ProtelisVM(program, context)

/*Esecuzione*/
vm.runCycle()
```

Listato 1.5: Esecuzione in Protelis

L'esecuzione consiste in:

1. caricamento il programma aggregato attraverso `ProtelisLoader` (nel listato 1.5 il programma viene caricato un file chiamato leader.pt).
2. definizione del contesto.
3. definizione della macchina virtuale che sarà in grado di eseguire il programma aggregato ottenuto dal parsing del modulo Protelis.
4. esecuzione effettiva tramite `runCycle`.

1.3 Scafi

Scafi[2] (Scala Fields) è un framework che include un DSL per il Field Calculus interno al linguaggio Scala [3, 4, 12]. I costrutti del Field Calculus sono modellati tramite l'interfaccia Scala espressa nel listato 1.6.

I campi non emergono in quanto Scafi, rispetto ad HFC, fornisce una semantica leggermente differente, nella quale i campi dei vicini vengono sostituiti da operazioni di *folding* effettuate sull'insieme dei vicini [10].

[2] `https://scafi.github.io/`

```
trait Constructs {
  def rep[A](init: => A)(fun: A => A): A
  def share[A](init: => A)(fun: (A, () => A) => A): A
  def nbr[A](expr: => A): A
  def foldhood[A](init: => A)(acc: (A, A) => A)(expr: => A): A
  def branch[A](cond: => Boolean)(th: => A)(el: => A)

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

Listato 1.6: Costrutti Field Calculus in Scafi [3]

1.3.1 Utilizzo

Scafi è stato scritto e pensato per essere utilizzato attraverso il linguaggio di programmazione Scala.

Definizione delle dipendenze

Per poter utilizzare Scafi è necessario dichiarare le dipendenze necessarie utilizzando un build tool. Di seguito vengono considerati Gradle ed SBT:

```
dependencies {
    implementation("org.scala-lang:scala-library:2.12.2")
    implementation("it.unibo.apice.scafiteam:scafi-core_2.12:0.3.2")
}
```

Listato 1.7: Dipendenze Gradle per Scafi [3]

```
libraryDependencies += "it.unibo.apice.scafiteam" %% "scafi-core"
                                                  % "0.3.2"
```

Listato 1.8: Dipendenze SBT per Scafi [3]

L'unica differenza sostanziale è il fatto che, utilizzando Gradle, va definita esplicitamente anche la dipendenza dal linguaggio Scala. Usando SBT non è necessario in quanto è il linguaggio predefinito del build tool.

Definizione del programma aggregato

Per definire un programma è sufficiente estendere `AggregateProgram` ed implementare il metodo *main*. È necessario definire anche un incarnazione, tipicamente creando una classe che estenda da `BasicAbstractIncarnation`. Un incarnazione è la definizione di tutti gli elementi necessari all'esecuzione

```
class MyIncarnation extends BasicAbstractIncarnation {
    /*Eventuali definizioni utili all'esecuzione*/
}
...
import MyIncarnation._
...
class MyProgram extends AggregateProgram {
    override def main(): Any = /*main program*/
}
```

Listato 1.9: Definizione di un programma Scafi

Esecuzione

L'esecuzione di un programma Scafi è equivalente all'invocazione di una funzione che accetta come unico parametro il contesto, formato da:

- letture dei sensori, sia di chi sta attualmente eseguendo che dei suoi vicini.
- risultati di esecuzioni passate, sia di chi sta attualmente eseguendo che dei suoi vicini.

Il risultato viene restituito sotto forma di Export.

```
val program = new MyProgram()

val results = /*Lettura dei risultati passati*/
val sensors = /*Lettura dei valori dei sensori*/
val nbSensors = /*Lettura dei valori dei sensori dei vicini*/

val context = factory.context(id, results, sensors, nbSensors)

val export = program(context)
```

Listato 1.10: Esecuzione Scafi

1.4 Programmazione aggregata

La programmazione aggregata fornisce un'architettura a strati (fig. 1.3) per semplificare drasticamente la pianificazione ed il mantenimento di sistemi distribuiti complessi.

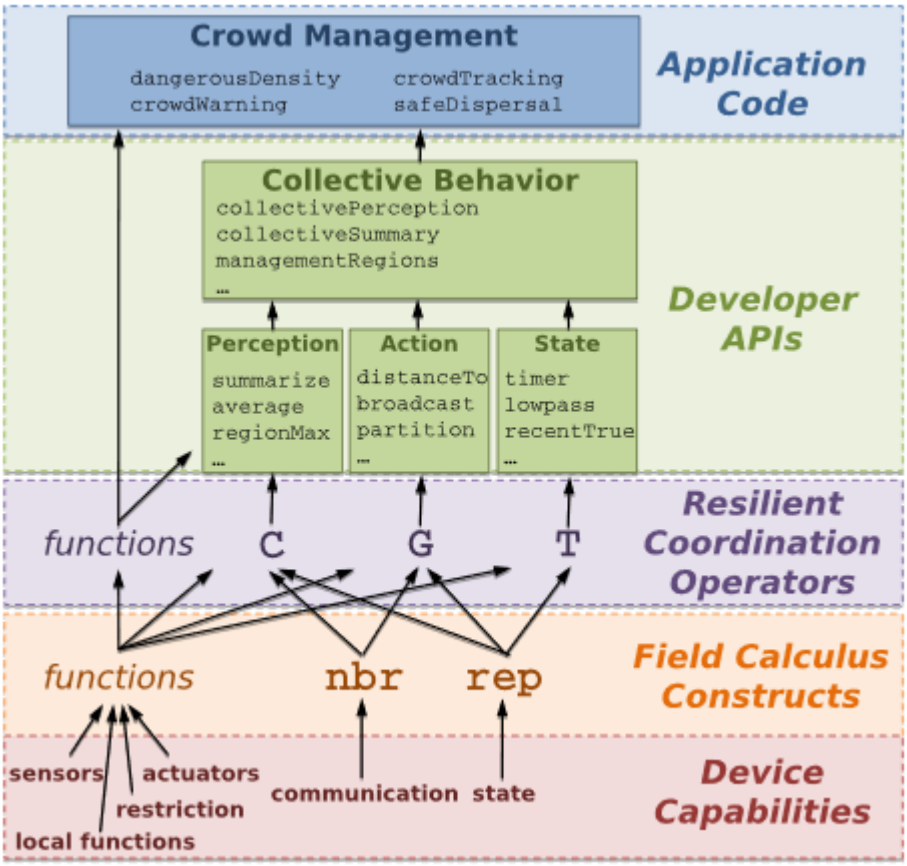


Figura 1.3: Architettura della programmazione aggregata [10]

L'approccio è motivato dalle seguenti osservazioni:

- la composizione di moduli deve essere semplice e trasparente
- sottosistemi diversi necessitano meccanismi di coordinazione diversi
- la coordinazione deve essere nascosta, così che i programmatori non debbano interagire con i dettagli implementativi.

Le varie incarnazioni del Field Calculus forniscono meccanismi per i primi due punti. La programmazione aggregata aggiunge due ulteriori strati per risolvere anche l'ultimo dei tre punti:

- **Lo strato di resilienza** (Resilient Coordination Operators), cruciale per nascondere la complessità e nel supportare la creazione di sistemi di coordinazione distribuiti.
  - L'operatore G (Gradient) è un operazione atta a spargere informazioni all'esterno usando una nozione di distanza.
  - L'operatore C (Collect) è l'operazione complementare di G.
  - L'operatore T (Time) è un operazione di evoluzione nel tempo.
- **API per i programmatori**, posto al di sopra dello strato di resilienza con lo scopo di catturare i pattern più comuni e fornire API più facili da usare per i programmatori.

Capitolo 2

Analisi del problema

La tesi consiste nella progettazione e sviluppo di una piattaforma per permettere l'esecuzione di computazioni aggregate. La piattaforma dovrà essere composta da due sotto-sistemi, in quanto è stata adottata l'architettura client-server. Il Server è il punto di centralizzazione e sincronizzazione dei vari dispositivi distribuiti, mentre i Client sono i vari dispositivi distribuiti che si collegano al Server ed eseguono il programma aggregato.

Definito un programma aggregato da eseguire, la piattaforma deve permettere a dei Client di collegarsi ad un Server, così da entrare a far parte della rete che eseguirà il programma. Il Server si occupa della gestione della rete e delle comunicazioni, mentre i Client eseguono il programma aggregato.

2.1 Requisiti

2.1.1 Indipendenza dall'implementazione di Field Calculus

L'implementazione effettiva di Field Calculus usata non deve incidere sull'architettura del sistema.

2.1.2 Apertura verso future implementazioni

Il sistema deve mantenersi aperto a nuove implementazioni di Field Calculus, quindi deve garantire la possibilità di supportare nuovi framework senza dover modificare l'architettura.

2.1.3 Compatibilità con Scafi e Protelis

Il sistema deve essere nativamente compatibile con Scafi e Protelis. Deve essere in grado di accettare programmi scritti in Protelis e/o con il DSL fornito da Scafi e deve essere in grado di eseguirli su una rete di dispositivi. Come già specificato in 2.1.1, il fatto di dover garantire compatibilità con queste due implementazioni non deve incidere sull'architettura.

2.1.4 Diverse modalità di esecuzione

Indipendentemente dal linguaggio usato per definire il programma aggregato, il sistema deve poter eseguire in una delle seguenti modalità:

- **Modalità virtuale**: il programma aggregato viene eseguito interamente all'interno del Server con Client emulati.
- **Modalità locale**: il programma aggregato viene eseguito interamente all'interno del Server, inviando solamente i risultati ai Client distribuiti.
- **Modalità remota**: l'esecuzione viene portata a termine dai dispositivi distribuiti.
- **Modalità ibrida**, alcuni dispositivi dichiarano di non poter eseguire; il Server deve farsi carico delle loro esecuzioni.

2.1.5 Topologia non nota ai dispositivi

La topologia di rete non deve essere nota a priori dai vari dispositivi distribuiti. Solamente il Server conosce le relazioni di vicinato, quindi, per potersi scambiare messaggi, i vari Client devono contattare il Server, che fungerà da broker.

2.1.6 Programma noto al Server

Indipendentemente dalla modalità di esecuzione, il Server conosce sempre e comunque il programma aggregato in esecuzione.

Capitolo 3

Progettazione

In questo capitolo verranno illustrate tutte le scelte architetturali effettuate prima di iniziare lo sviluppo del sistema. Verranno mostrati i componenti principali necessari a garantire il funzionamento del sistema rispettando i requisiti individuati e le loro interazioni.

3.1 Terminologia

Per evitare ambiguità, sono stati adottati i seguenti termini fin da inizio progettazione.

Supporto Parte del Server funzionante da broker di messaggi .

Piattaforma di esecuzione Esecutore del programma aggregato. Può essere il modello di un dispositivo remoto o virtualizzato.

Adattatore Parte atta a far interagire il sistema con le varie implementazioni di Field Calculus.

3.2 Architettura di sistema

L'architettura generale della piattaforma nel suo insieme può essere schematizzata come da figura 3.1.

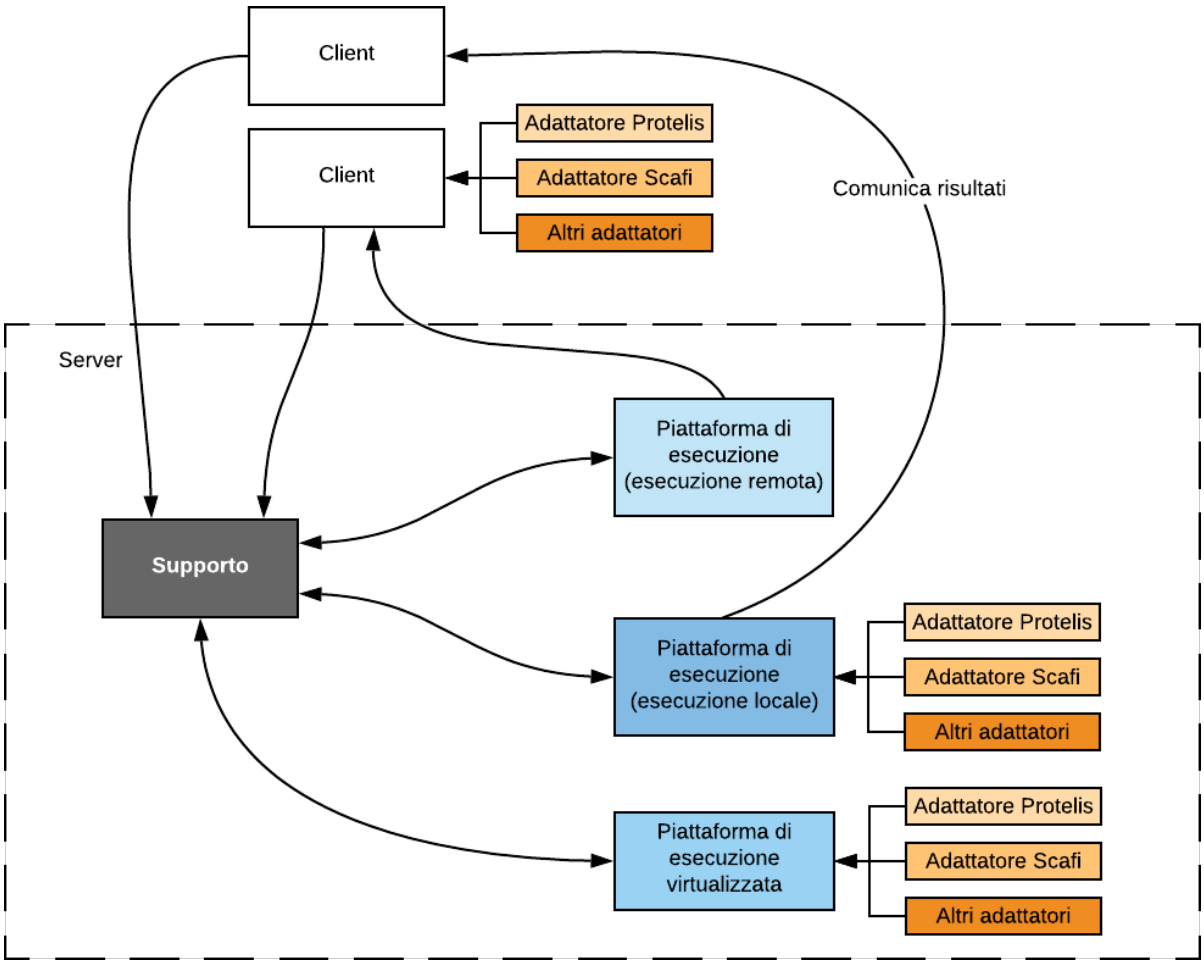


Figura 3.1: Architettura generale

3.2.1 Supporto

Il Supporto è il punto di incontro tra le varie piattaforme di esecuzione. Chiunque necessiti di comunicare qualcosa con un vicino deve obbligatoriamente contattare il Supporto, in quanto è l'unico punto dell'intera piattaforma a conoscere la topologia di rete. Il Supporto è anche l'unico punto di ingresso per i Client. Qualsiasi comunicazione che vogliano inviare passa attraverso il Supporto, che funziona quindi anche da broker di messaggi tra Client (sia fisici che emulati).

3.2.2 Adattatore

L'adattatore è il collante tra la piattaforma e le varie implementazioni di Field Calculus. Qualunque parte del sistema che debba eseguire un programma aggregato necessita obbligatoriamente di un adattatore. Un adattatore ha il compito di tradurre gli input ricevuti in comandi specifici dell'implementazione utilizzata in modo che la piattaforma di esecuzione sia in grado di eseguire il programma. Inoltre, un adattatore deve anche interpretare i risultati delle varie esecuzioni in modo che il sistema possa gestirli.

Indipendenza dall'implementazione

Come espresso anche da requisiti, l'architettura del sistema deve essere indipendente dall'implementazione effettiva utilizzata per esprimere programmi aggregati e per eseguirli. Inoltre, qualora fosse necessario, dovrà essere possibile aggiungere compatibilità per altre implementazioni non prese in considerazione in questo specifico progetto, persino per implementazioni ancora non esistenti.

L'utilizzo di adattatori è utile anche per soddisfare questi requisiti. Relegando tutta la logica relativa ad una specifica implementazione all'interno di un adattatore è possibile mantenere la struttura interna del sistema invariata, indipendentemente dall'adattatore che si sta usando.

Il concetto di adattatore come oggetto inglobante tutta la logica relativa ad una particolare implementazione è stato ispirato dal simulatore ***Alchemist***[1]. In quest'ultimo, la compatibilità con diversi framework (tra i quali sono presenti anche Scafi e Protelis) è stata fornita utilizzando oggetti chiamati incarnazioni, con lo scopo di fare da collante tra i vari sistemi ed il simulatore stesso [8, 13].

3.2.3 Piattaforme di esecuzione

Sono state previste diverse tipologie di piattaforme di esecuzione per soddisfare le diverse modalità di esecuzione previste da requisiti. Una piattaforma di esecuzione è il modello di un dispositivo in grado di eseguire un programma aggregato, sia esso un Client reale o completamente emulato.

Piattaforma di esecuzione virtualizzata

Tipologia di piattaforma utilizzata quando si vuole emulare completamente un dispositivo. Necessita di un adattatore per poter eseguire il programma aggregato.

[1] `http://alchemistsimulator.github.io/`

Piattaforma di esecuzione remota

Piattaforma utilizzata quando ad eseguire il programma aggregato deve essere il Client remoto. Non servono direttamente adattatori in quanto non è la piattaforma stessa ad eseguire, ma necessita di comunicare con il vero e proprio Client per impartire comandi.

Piattaforma di esecuzione locale

Utilizzata nel caso di modalità di modalità ibrida (2.1.4). Necessita sia di un adattatore, in quanto deve poter eseguire, che di comunicare con il Client associato per fornire i risultati dell'esecuzione.

3.3 Design di dettaglio

3.3.1 Tipologie di messaggi

Data la diversa natura delle interazioni tra componenti, sono utili diverse tipologie di messaggi che le entità del sistema possono utilizzare per comunicare tra di loro. Le comunicazioni avvengono principalmente in 3 direzioni:

- da Client a Supporto
  - un Client deve necessariamente comunicare con il supporto tutte le volte che serve inviare una qualsiasi informazione ad un vicino.
  - un Client deve contattare il Supporto per entrare nella rete.
  - un Client deve contattare il Supporto qualora voglia entrare o uscire dalla modalità di esecuzione locale.
- da Supporto a Piattaforma di esecuzione
  - il Supporto deve contattare le piattaforme di esecuzione per consegnare loro messaggi provenienti da altre piattaforme o Client.
  - deve anche contattare le piattaforme per dir loro di eseguire il programma aggregato.
- da Piattaforma di esecuzione a Client
  - in base alla modalità di esecuzione, una piattaforma di esecuzione deve inviare al Client varie informazioni.

Individuati i precedenti flussi di comunicazione si possono identificare le seguenti tipologie di messaggi (figura 3.2):

- **Richiesta di ingresso in rete**: utilizzato dai Client per esprimere al Supporto la volontà di entrare nella rete.
- **Da inviare ai vicini**: utilizzato per comunicare al Supporto che il messaggio che sta inviando va inoltrato ai suoi vicini. Da ricordare il fatto che che il Supporto è l'unico a conoscere le relazioni di vicinanza, quindi l'utilizzo di questa tipologia di messaggio è l'unico modo disponibile ai Client ed alle piattaforme di esecuzione per comunicare con i loro vicini.
- **Risultato**: utilizzato per comunicare un risultato di una computazione ai vicini. Tipicamente il messaggio effettivo da inviare ai vicini sarà di questa tipologia. Per questa ragione il Supporto lo utilizza per comunicare alle piattaforme di esecuzione i risultati da inoltrare.

  Se si sta utilizzando la modalità di esecuzione locale, la stessa tipologia di messaggio viene usata anche dalle piattaforme di esecuzione per comunicare i risultati della computazione ai Client associati.
- **Esecuzione**: messaggio utilizzato principalmente per indicare alle piattaforme di esecuzione di iniziare ad eseguire il programma aggregato. Se il messaggio viene ricevuto da una piattaforma di esecuzione remota verrà inoltrato al Client.
- **Entrata in di esecuzione locale**: utilizzato da un Client per comunicare che da quel momento non potrà più eseguire, lasciando al Server il compito di eseguire la sua parte.
- **Uscita da di esecuzione locale**: tipologia di messaggio complementare della precedente tipologia di messaggio. Utilizzato per uscire dalla modalità di esecuzione locale ed esprimere il fatto che da quel momento il Client può eseguire autonomamente.

3.3.2 Comunicazione tra Client e Server

A livello di architettura non è stato prevista alcuna metodologia specifica di comunicazione tra Client e Server. Tale comunicazione può essere implementata secondo diverse modalità, purché venga mantenuta la possibilità di scambiarsi le diverse tipologie di messaggi precedentemente individuate.

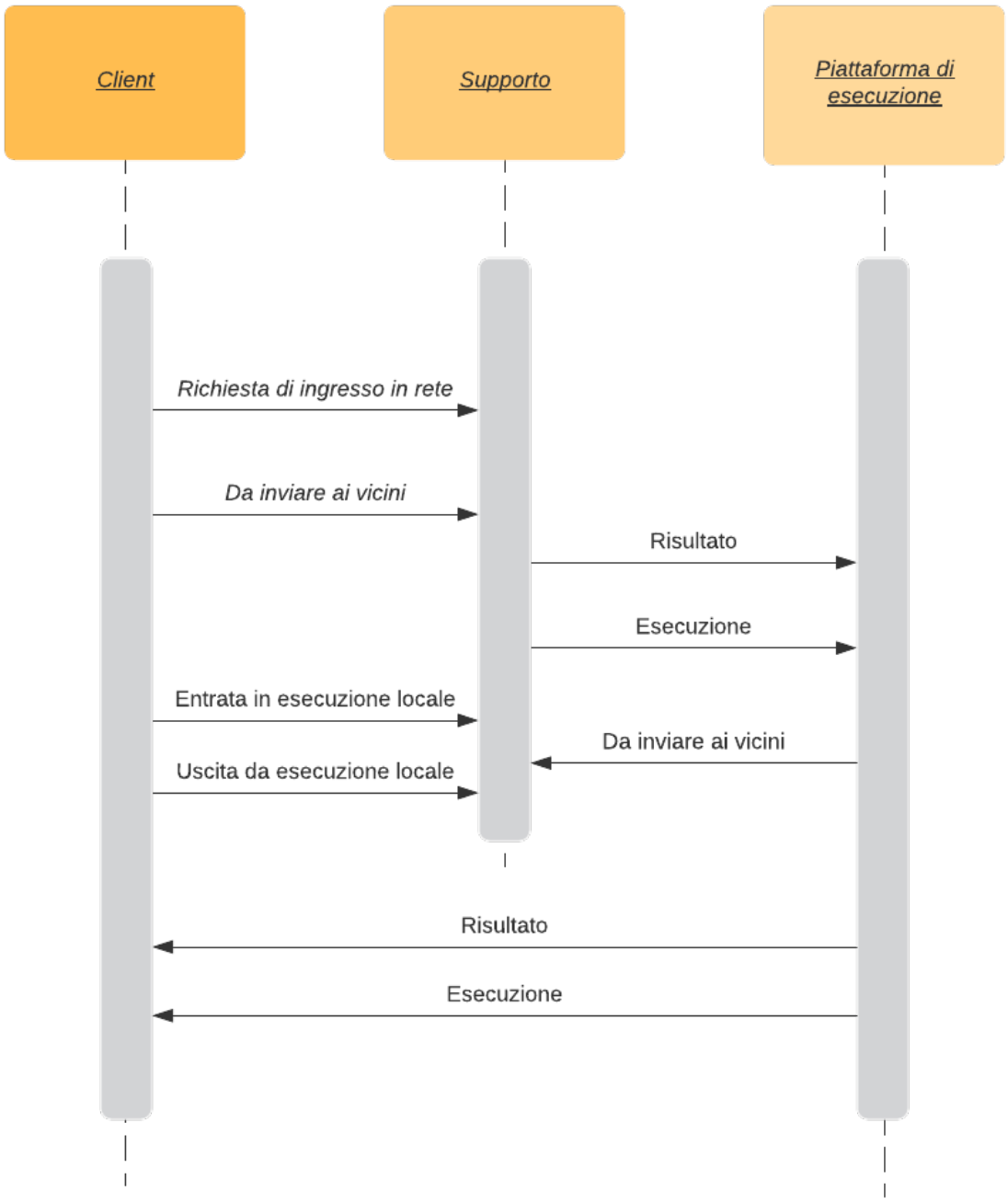


Figura 3.2: Tipologie di messaggi

Capitolo 4

Implementazione

Di seguito verranno illustrate le varie implementazioni facenti seguito alle scelte fatte in fase di progettazione. Verranno mostrate le implementazioni effettive di tutti i componenti realizzati, sia per quanto riguarda la parte relative al server (4.1) che ai client (4.2).

4.1 Sotto-sistema Server

Il Server è stato sviluppata perlopiù utilizzando Kotlin come linguaggio di programmazione. L'Integrazione con Scafi ha richiesto l'utilizzo di alcune parti scritte in Scala, rendendo quindi necessaria una parte di adattamento del codice scritto in Kotlin per poter essere usato agevolmente anche da Scala e viceversa.

È stato anche creato un repository[1] per mantenere il codice relativo alla parte Server della piattaforma.

4.1.1 Suddivisione in package

Per meglio organizzare il codice e semplificare successive fasi di manutenzione che si renderanno sicuramente necessarie nel tempo, i sorgenti sono stati suddivisi in package, raggruppando parti atte a raggiungere il medesimo scopo o dedicate a risolvere una specifico problema.

[1] `https://github.com/LorisCangini/Aggregate-Computing-Backend`

I package utilizzati sono:

- **devices**: contenente la definizione di tutte le tipologie di piattaforme di esecuzione.

  Maggiori dettagli nella sezione 4.1.2.

- **server**: contenente il Supporto e la gestione della topologia di rete.

  Maggiori dettagli nella sezione 4.1.3.

- **adapters**: contenente la parte relativa agli adattatori. Al suo interno sono contenuti gli adattatori Scafi e Protelis. Eventuali adattatori futuri andranno inseriti in questo package.

  Maggiori dettagli nella sezione 4.1.4.

- **communication**: contenente la parte necessaria a stabilire una connessione tra piattaforme di esecuzione, Client e Supporto.

  Maggiori dettagli nella sezione 4.1.6.

4.1.2 Dispositivi

Per dar seguito alle tipologie di piattaforme di esecuzione individuate in fase di progettazione, è stata creata una gerarchia di dispositivi (figura 4.1).

Device Inizialmente è stata creata la generalizzazione di più alto livello, raccogliendo le proprietà e le funzionalità comuni a tutte le tipologie di piattaforme di esecuzione. Tali proprietà includono `id` (un identificatore univoco che ogni dispositivo ha per essere distinto facilmente dagli altri), `name` (un nome umanamente leggibile per rendere il dispositivo riconoscibile anche ad un primo sguardo) e `status` (lo stato interno del dispositivo. Contiene l'insieme dei messaggi ricevuti dai vicini e delle computazioni portate a termine).

Inoltre, indipendentemente dalla tipologia di piattaforma, un dispositivo deve poter eseguire il programma aggregato (funzione `execute`), ricevere un messaggio (funzione `tell`) e mostrare i risultati della computazione (funzione `showResult`).

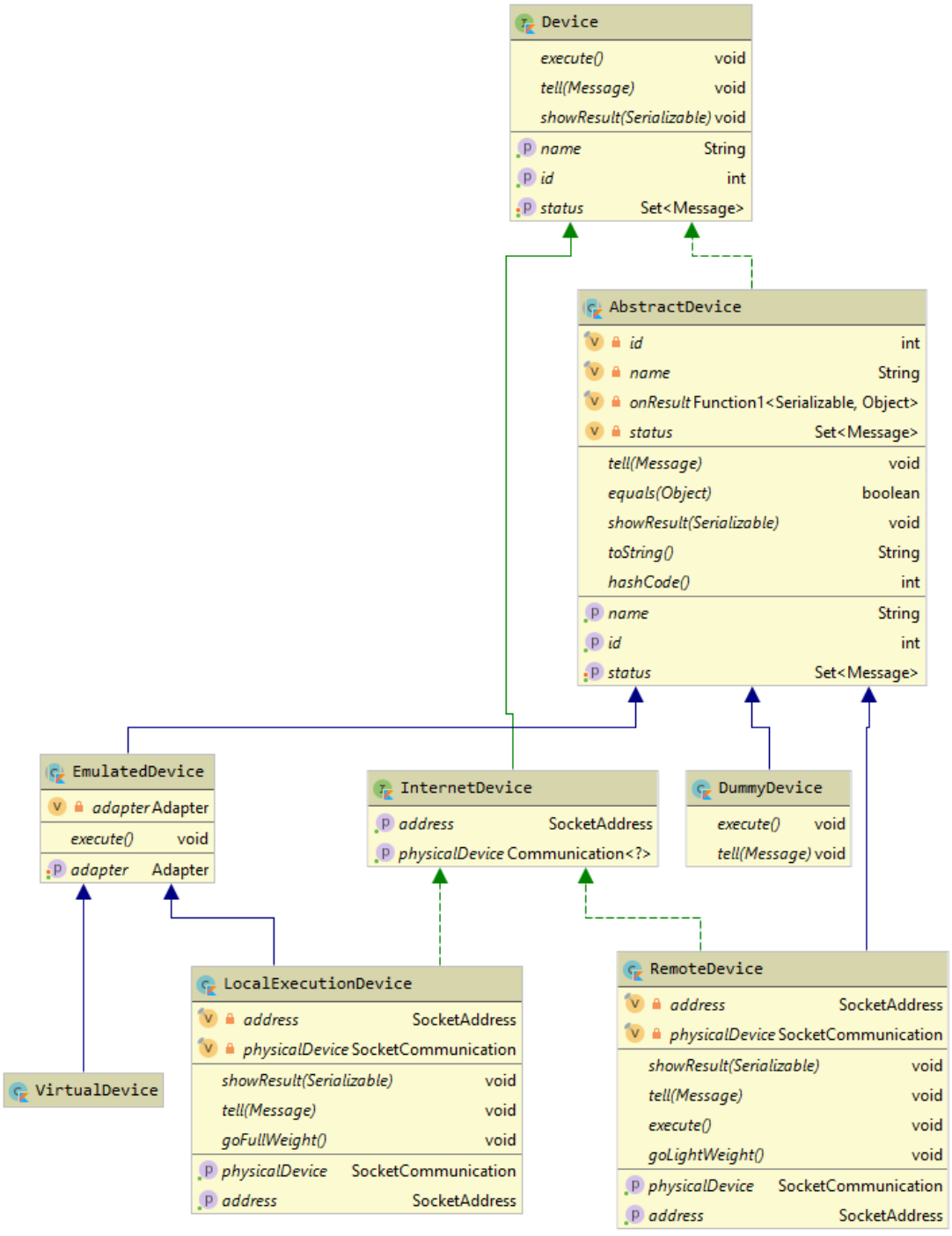


Figura 4.1: Gerarchia delle tipologie di dispositivi

AbstractDevice Implementazione basilare dei campi e dei metodi previsti da `Device`.

Tramite `toString` il nome è stato reso opzionale, se resta vuoto verrà usato l'identificativo al suo posto.

In seguito alla ricezione di un messaggio, in base alla sua tipologia, vengono attuate diverse azioni. Se si tratta di un risultato di una computazione, sia esso derivante da un esecuzione dello stesso dispositivo o proveniente da un vicino, viene aggiornato lo stato interno (variabile `status`). Se si tratta di un messaggio di inizio esecuzione, provieniente quindi dal Supporto, viene richiamato il metodo `execute`, ancora non definito in questo livello.

Sono stati implementati anche i metodi `equals` e `hashCode` per far sì che due dispositivi vengano considerati uguali semplicemente se il loro id è uguale.

InternetDevice Ulteriore interfaccia che aggiunge due ulteriori campi rispetto a quelli definiti da `Device`. Lo scopo di questo livello di astrazione è rappresentare quei dispositivi che necessitano di comunicare con un Client remoto.

I campi aggiunti sono `address` (l'indirizzo formato da IP e porta del client remoto col quale comunicare) e `physicalDevice` (la metodologia di comunicazione da adottare per comunicare con il Client). Maggiori dettagli sulle metodologie di comunicazione sono disponibili in 4.1.6.

EmulatedDevice Questa astrazione rappresenta quelle piattaforme di esecuzione che necessitano di avere un adattatore per poter eseguire. È una specificazione ulteriore di `AbstractDevice`, mantenendosi però ancora astratta come definizione.

In questa astrazione viene definito il campo `adapter`, (l'adattatore che il dispositivo utilizzerà quando necessario) e si implementa `execute` reindirizzando il comando dalla piattaforma all'adattatore utilizzato.

DummyDevice Prima piattaforma di esecuzione non astratta. Rappresenta una tipologia di dispositivo da utilizzare unicamente con lo scopo di testare altre funzionalità. Questa piattaforma non può eseguire e non attua alcuna azione a fronte di messaggi ricevuti.

VirtualDevice Rappresenta una piattaforma di esecuzione completamente virtualizzata, senza avere un corrispettivo Client fisico. Non necessita di definire ulteriori campi o funzioni rispetto a quelle definite da `EmulatedDevice`.

RemoteDevice Rappresenta la piattaforma di esecuzione nella quale è il Client remoto ad eseguire il programma aggregato. Per questo motivo, non necessita di un adattatore, in quanto non è la piattaforma ad eseguire.

È stato necessario ridefinire alcune funzioni rispetto a quanto fatto per `AbstractDevice`. `showResult` è stata definita in modo da inviare i risultati al Client associato. Da notare che questa funzionalità resta inutilizzata in quanto è direttamente il Client a computare i risultati, quindi la piattaforma di esecuzione all'interno del Server non riceverà mai i risultati e non dovrà visualizzarli.

`tell` è stata ridefinita in modo tale da inviare al Client associato qualsiasi messaggio ricevuto dalla piattaforma. L'unica eccezione è data dal messaggio inviato dal Client nel caso di volontà di smettere di eseguire autonomamente. In tal caso, questa piattaforma viene trasformata in un `LocalExecutionDevice` tramite la funzione appositamente prevista. Similmente, anche `execute` è stata definita in modo tale da inviare al Client un messaggio di inizio esecuzione.

È stata prevista anche la funzione `goLightWeight` per trasformare questa piattaforma in un `LocalExecutionDevice`, in caso il Client comunichi di non poter più eseguire autonomamente.

LocalExecutionDevice Rappresenta la piattaforma di esecuzione in modalità locale, cioè il caso in cui un Client non possa eseguire autonomamente. In questa situazione è necessario avere sia un adattatore, per poter eseguire allo stesso modo di un `VirtualDevice`, sia un riferimento al Client, per poter inviare i risultati allo stesso modo di un `RemoteDevice`.

Similmente a quanto era accaduto nella precedente tipologia di dispositivo, è stata definita `goFullWeight` per trasformare questa piattaforma in un `RemoteDevice`, nel caso in cui un Client manifesti la volontà di ritornare ad eseguire.

4.1.3 Supporto, Esecuzione e Topologia

Supporto Il principale obiettivo del Supporto è consentire la comunicazione tra i dispositivi della rete. Essendo il Supporto l'unica entità del sistema dove la topologia ed i membri della rete sono noti, è necessariamente da questo punto che l'esecuzione da parte della rete del programma aggregato deve partire.

Il supporto è stato modellato come un `AbstractDevice` ed un `InternetDevice`, in quanto le operazioni che deve rendere disponibili non sono molto diverse da quelle di un dispositivo che comunica con un Client.

Rispetto ai dispositivi descritti in precedenza, `address` rappresenta l'indirizzo e la porta dove il Supporto resterà in ascolto di comunicazioni da parte di eventuali Client.

Il Supporto non ha uno stato interno contenente i messaggi ricevuti, in quanto non rappresenta un dispositivo. È presente l'insieme dei dispositivi (campo `devices`) che compongono la rete, con funzionalità di gestione della topologia di rete. `execute` non esegue direttamente, ma vengono richiamati gli `execute` di tutte le piattaforme di esecuzione presenti nella rete.

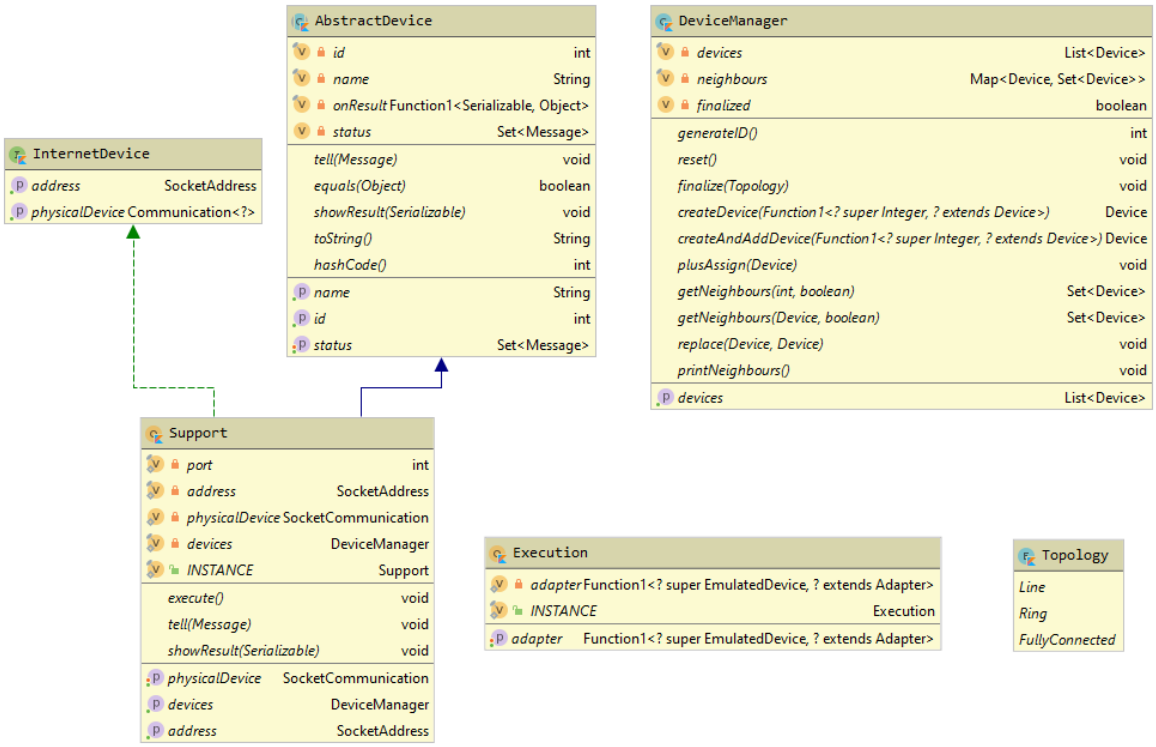


Figura 4.2: Supporto, Esecuzione e `DeviceManager`

Le tipologie di messaggi in arrivo supportate, e le relative risposte, sono differenti rispetto a quelli previsti da dispositivi comuni.

- **Messaggio di entrata nella rete**: viene ricevuto da un Client che vuole entrare a far parte della rete.

  Nel caso, si crea una nuova piattaforma di esecuzione remota (`RemoteDevice`) associata al Client che ha inviato il messaggio e si aggiunge alla rete.

  Successivamente si risponde al Client comunicandogli l'id assegnato alla piattaforma di esecuzione appena creata, così che il Client possa utilizzarlo nelle successive comunicazioni per farsi riconoscere facilmente all'interno della rete.

- **Messaggio da inoltrare ai vicini**: il messaggio ricevuto viene inoltrato (tramite la funzione `tell`) ai vicini del mittente. Il campo `devices` fornisce tutte le funzionalità necessarie.

- **Messaggio di entrata in/uscita da esecuzione locale**: il messaggio viene inoltrato alla piattaforma di esecuzione corrispondente al Client che ha inviato il messaggio.

Gestione della topologia Per gestire la topologia di rete, è stato creato un componente ad-hoc, il ***DeviceManager***, il quale ha come compiti principali il permettere e semplificare l'entrata di nuove piattaforme di esecuzione nella rete egestire le relazioni di vicinanza esistenti tra i dispositivi nella rete. Per ottemperare al primo dei compiti sopracitati, vengono predisposte alcune funzionalità:

- `generateID`, con lo scopo di trovare e restituire il primo identificativo non utilizzato. Questa è una funzionalità ad uso interno, che tornerà utile per le funzioni descritte di seguito.
- `createDevice`, che sfruttando `generateID` descritta in precedenza, genera un nuovo dispositivo utilizzando la strategia di creazione fornita come parametro, assegnando come ID quello generato.

  Il dispositivo creato in questa maniera **non** viene aggiunto alla rete.
- `createAndAddDevice`, che si comporta come la precedente, aggiungendo però il dispositivo appena alla rete.
- `plusAssign`, la definizione dell'operatore `+=` per la classe `DeviceManager`.

  Qualora si voglia aggiungere un dispositivo alla rete, sarà sufficiente utilizzare questo operatore su un istanza di `DeviceManager`, rendendo così immediata la comprensione del codice necessario a definire la rete che dovrà poi eseguire il programma aggregato.
- `replace`, che sostituisce un dispositivo con un altro, ereditando eventuali relazioni di vicinanza.

  Questa funzionalità è utile specialmente nei casi di entrata ed uscita da modalità di esecuzione locale, in quanto un `RemoteDevice` va trasformato in un `LocalExecutionDevice` e viceversa.

Per quanto riguarda la gestione della topologia di rete, quindi delle relazioni di vicinanza tra dispositivi, sono state previste altre funzionalità:

- `getNeighbours`, che, dato un dispositivo, o anche solamente il suo identificativo, vengono trovati e restituiti tutti i suoi vicini.
- `printNeighbours`, utilizzata per stampare a video tutte le relazioni di vicinanza attualmente esistenti. Creata principalmente per scopi di testing.
- `finalize`, la funzione che crea le relazioni di vicinanza.

  Dato un valore previsto dall'enumeratore `Topology`, che indica come dovranno essere create le varie relazioni di vicinanza, la funzione crea tutte queste relazioni, così che si possa iniziare ad eseguire.

Le topologie attualmente previste (figura 4.3) sono:

- `Line`: le relazioni vengono inizializzate supponendo i dispositivi in linea retta. Ogni dispositivo potrà comunicare direttamente con il suo predecessore ed il suo successore (se presenti).
- `Ring`: simile alla topologia *line*, ma l'ultimo dispositivo potrà comunicare con il primo e viceversa.
- `FullyConnected`: ogni dispositivo è connesso con tutti gli altri.

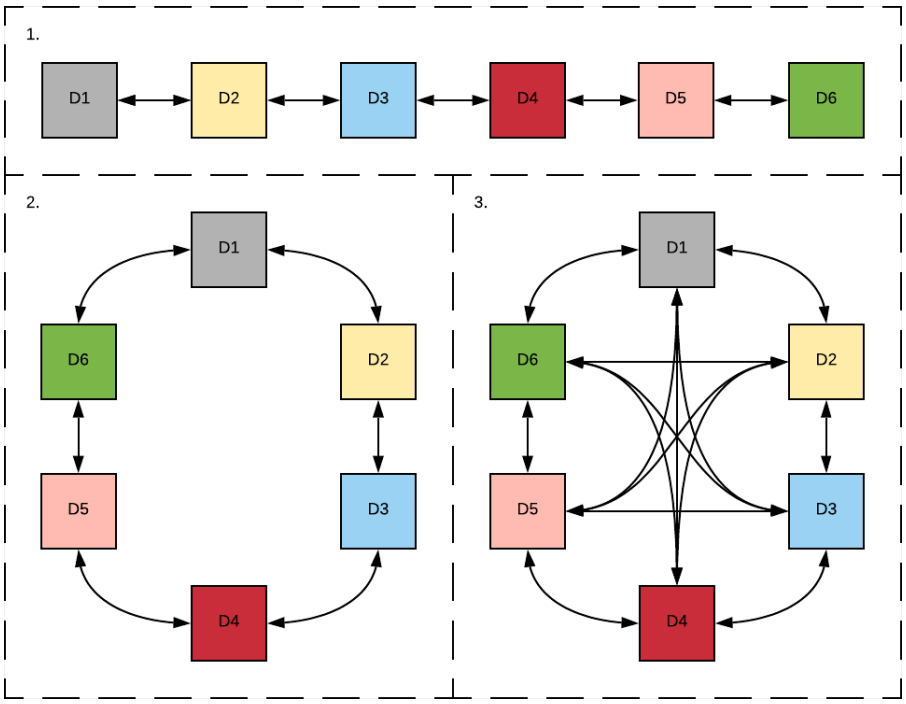


Figura 4.3: Topologie attualmente previste.
1. in linea 2. ad anello
3. completamente connessa

Nel caso serva aggiungerne di nuove, è sufficiente definire la nuova topologia all'interno di `Topology` e definire come le varie relazioni di vicinanza dovranno essere create all'interno di `finalize`.

- `reset`, utilizzato per cancellare tutti i dispositivi attualmente presenti, comprese relative relazioni di vicinanza.

Esecuzione L'oggetto `Execution` contiene le informazioni utili all'esperimento attuale per poter portare a termine tutte le operazioni. L'unico oggetto utile a questo scopo è la strategia di creazione dell'adattatore in grado di eseguire il programma aggregato Questo valore viene utilizzato tutte le volte che è necessario creare un nuovo adattatore e non ne viene specificato uno. L'utilizzo principale si ha quando una piattaforma di esecuzione remota deve entrare in modalità di esecuzione locale. In questo caso la piattaforma stessa deve trasformarsi, passando da `RemoteDevice` a `LocalExecutionDevice`, quindi con il requisito aggiuntivo di necessitare di un adattatore per poter eseguire.

4.1.4 Adattatori

Un adattatore è la parte che si frappone tra il sistema e le implementazioni effettive che si desidera utilizzare per specificare il programma aggregato da eseguire e per gestire l'esecuzione. Come espresso anche in fase di progettazione (3.2.2), l'utilizzo di adattatori deve rendere il sistema indipendente dalle implementazioni effettive dei vari framework per computazioni aggregate. Questa proprietà è stata garantita facendo uso di un interfaccia e di raffinamenti di essa (figura 4.4).

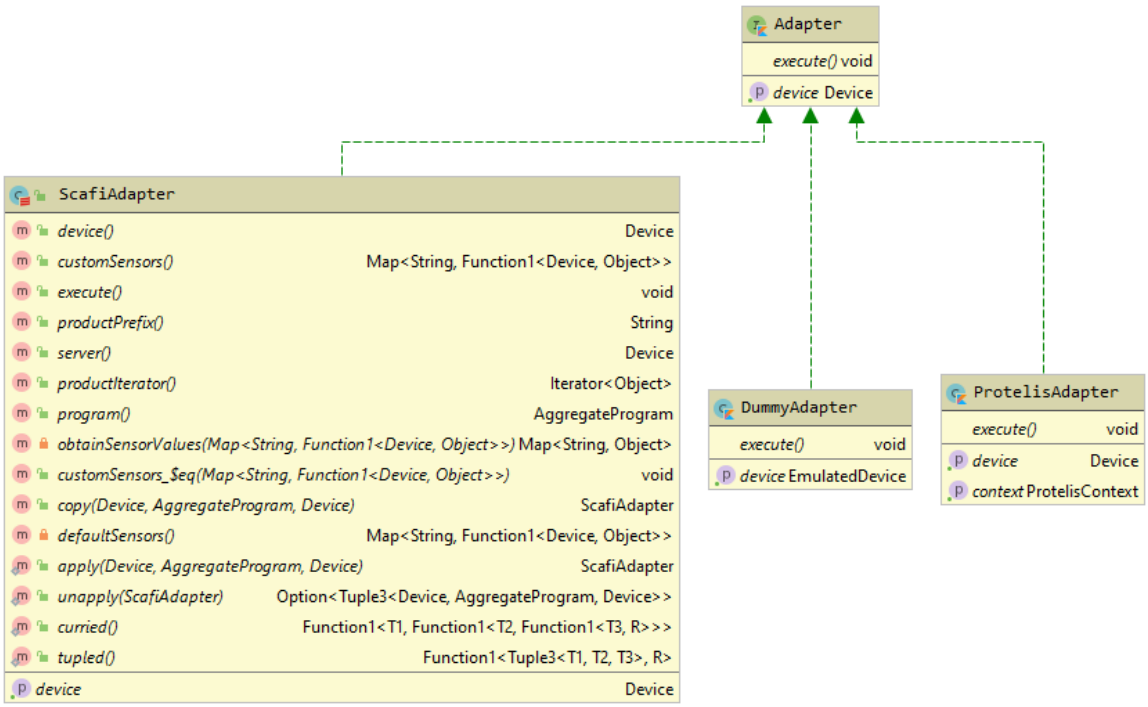


Figura 4.4: Gerarchia adattatori

Come già mostrato in 4.1.2, un dispositivo che necessita di eseguire (quindi un `EmulatedDevice`), richiede ed utilizza un oggetto di tipo `Adapter`, cioè l'interfaccia generica che raccoglie i tratti comuni di tutti gli adattatori. Questo rende il funzionamento del sistema indipendente dai vari framework in quanto le piattaforme di esecuzione non conoscono nulla delle loro implementazioni, ma agiscono solamente tramite l'interfaccia più generica, che nasconde loro i dettagli implementativi.

Per lo stesso motivo, il sistema è aperto a supportare nuove implementazioni, in quanto sarà sufficiente definire i corrispettivi adattatori estendendo `Adapter`. Dato che il resto del sistema lavora solamente tramite quest'ultima interfaccia generica, la nuova implementazione sarà automaticamente supportata dall'intera infrastruttura.

Interfaccia `Adapter` L'interfaccia `Adapter` contiene solamente il campo `device`, che è il riferimento ad dispositivo contenente questo adattatore, e la funzione `execute`, che verrà implementata dai veri e propri adattatori specifici per eseguire il programma aggregato. Questi due elementi sono sufficienti all'intero sistema per fare tutte le operazioni necessarie all'esecuzione.

Adattatore Dummy Il `DummyAdapter` è un adattatore fittizio utilizzato principalmente per scopi di testing. Non è in grado di eseguire alcun programma aggregato.

Adattatore Protelis Per scrivere l'adattatore per il linguaggio Protelis è stato necessario definire alcuni componenti previsti dall'architettura di Protelis stesso (figura 4.5). *Protelis* per poter funzionare si aspetta di avere un **contesto**, definito da un oggetto che implementa l'interfaccia `ExecutionContext`. Nel caso specifico, il contesto è stato modellato tramite la classe astratta `ProtelisContext`, a sua volta definita a partire da un'altra implementazione del contesto chiamata `AbstractExecutionContext` (non rappresentata in figura 4.5 per motivi di spazio). Tutti i tratti più comuni del contesto atteso da Protelis sono stati già inseriti in `AbstractExecutionContext`, quindi, per è sufficiente definire le poche funzioni rimaste senza implementazione. Successivamente è stata creata l'implementazione finale `SimpleProtelisContext`, che sarà l'effettiva classe usata per gli esperimenti più comuni. Nel caso in cui per un dato esperimento sia necessario definire un contesto più elaborato, è sufficiente definirlo a partire da `ProtelisContext`.

Oltre al contesto, è necessario avere anche un **gestore delle comunicazioni** di rete, definito da un oggetto che implementa l'interfaccia `NetworkManager`. Il `NetworkManager` è la parte che si occupa di trasferire le informazioni tra dispositivi. Un dispositivo deve poter accedere allo stato più recente dei vicini, e deve poter aggiornare il proprio stato da esportare per i vicini. Per implementare queste funzionalità è stata creata la classe `ProtelisNetworkManager`.

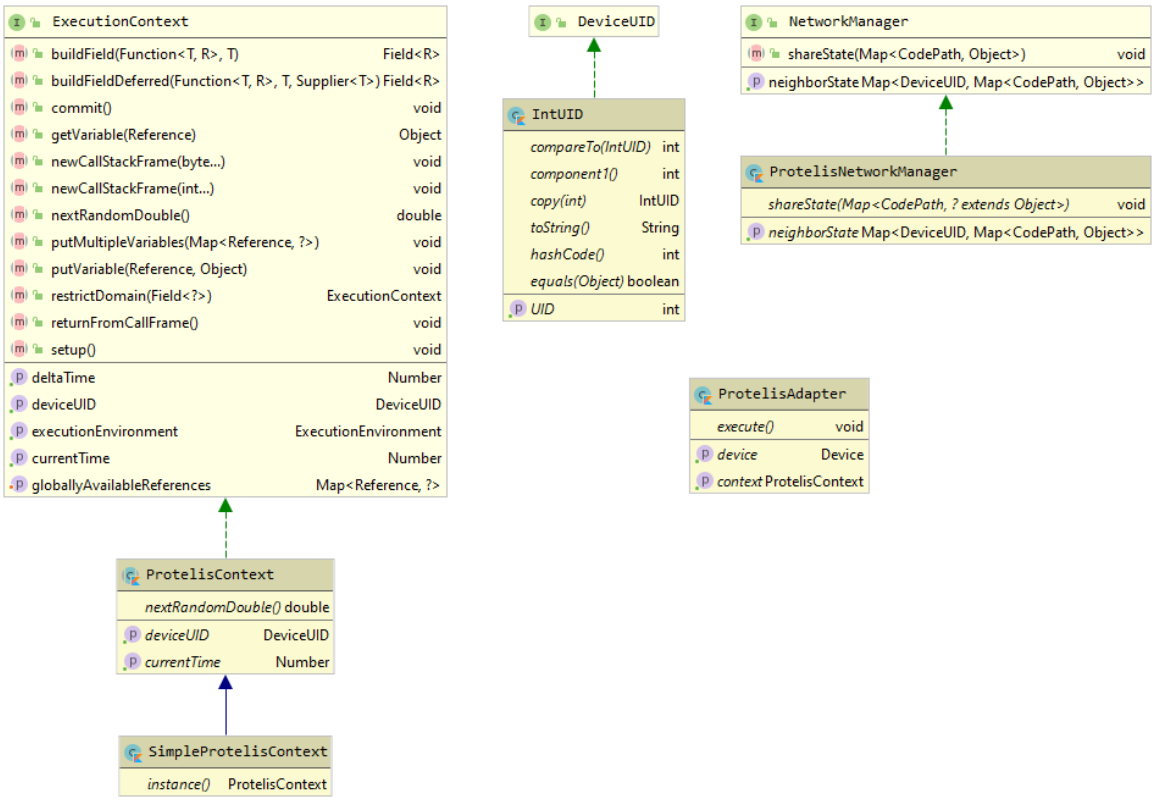


Figura 4.5: Componenti per l'adattatore Protelis

È stato necessario implementare le seguenti due funzioni per svolgere il compito preposto da `NetworkManager`:

- `getNeighbourState`: utilizzata per collezionare lo stato più recente dei vicini.

  Ogni dispositivo ha al suo interno la variabile `status`, contenente il proprio stato, ma anche quello inviatogli dai vicini. Per questo motivo, è stato sufficiente leggere tale variabile e trasformarla nel formato atteso dall'interfaccia `NetworkManager`.

- `shareState`: utilizzata per inviare il proprio stato aggiornato ai vicini. Per implementare questa funzionalità, si invia semplicemente un messaggio al Supporto, indicando di inviare ai vicini un altro messaggio contenente lo stato aggiornato.

Protelis si aspetta anche di ricevere gli `id` dei dispositivi espressi tramite un oggetto che implementi `DeviceUID`. È stato semplicemente definito `IntUID`, che funziona da involucro per gli id dei dispositivi già menzionati in 4.1.2.

Forniti tutti questi componenti, per poter utilizzare l'adattatore Protelis è necessario fornire il programma aggregato da eseguire.

Un esecuzione in Protelis non è altro che un invocazione sulla macchina virtuale creata precedentemente (1.2.1). Lo stato interno dei vari dispositivi appartenenti alla rete viene automaticamente aggiornato dal `NetworkManager`.

Adattatore Scafi L'adattatore Scafi è stato definito utilizzando il linguaggio Scala. Il framework stesso è stato scritto con lo stesso linguaggio, quindi le API fornite sono utilizzabili agevolmente solamente attraverso Scala.

L'approccio di Scafi è funzionale, al contrario di quello basato su oggetti di Protelis, quindi per definire l'adattatore Scafi non è stato necessario creare diversi oggetti che estendessero interfacce predefinite. L'unica cosa che va definita è il programma aggregato, creando un oggetto che estenda `AggregateProgram`, definendo nella funzione `main` il corpo del programma da eseguire.

Per poter definire un programma aggregato è necessario specificare anche un incarnazione. La definizione di `AggregateProgram` è interna a quella di una qualsiasi incarnazione, quindi per poter utilizzare, e quindi anche estendere, la classe `AggregateProgram` serve definire prima l'incarnazione che si vuole utilizzare.

Definito il programma aggregato, l'adattatore ha tutto il necessario per poter eseguire. Un esecuzione in Scafi è l'applicazione del programma aggregato, visto come funzione, passandogli come parametro il **contesto** di esecuzione, formato da:

- i risultati delle esecuzioni precedenti dello stesso dispositivo e dei suoi vicini. Tutti questi valori sono disponibili sfruttando la variabile `status` dei dispositivi.

- i valori più recenti dei sensori, sia del dispositivo in esecuzione che dei suoi vicini. Per quanto riguarda i sensori previsti di default, il loro valore viene ottenuto invocando funzioni predefinite. È possibile specificare anche sensori personalizzati, in tal caso si specifica anche come ottenere i loro valori tramite funzioni che verranno invocate prima dell'esecuzione del programma aggregato.

Il risultato dell'esecuzione è detto **Export**, che verrà utilizzato per aggiornare manualmente lo stato dei dispositivi. L'Export viene inviato prima al dispositivo proprietario dell'adattatore in questione, poi a tutti i suoi vicini (passando come già indicato dal Supporto, che provvederà a consegnare i messaggi sfruttando il `DeviceManager`).

4.1.5 Integrazione Scala-Kotlin

Dato l'utilizzo di Scala e Kotlin nello stesso progetto, considerando il fatto che le porzioni scritte nei due linguaggi devono interagire tra di loro nel loro funzionamento, è stata necessaria una fase di adattamento del codice per poter utilizzare codice scritto in un linguaggio dall'altro e viceversa.

Traduzione in bytecode

È possibile utilizzare Scala e Kotlin e farli interagire tra di loro in quanto basati entrambi sulla Java Virtual Machine (JVM). Il codice sorgente scritto in un qualsiasi linguaggio basato sulla JVM viene tradotto in bytecode. I risultati di questa traduzione non sono sempre interoperabili, specialmente quando si usano funzionalità specifiche ed avanzate dei diversi linguaggi.

Limiti riscontrati Nel caso specifico, l'utilizzo intensivo di funzioni higher-order in più parti del progetto e all'interno di Scafi e Protelis ha reso l'interoperabilità tra i due linguaggi piuttosto complessa, in quanto le traduzioni risultanti da funzioni higher-order utilizzate in Kotlin è differente da quello generato dalle equivalenti funzioni utilizzate in Scala.

Per risolvere la situazione sono state create alcune classi di supporto (package *util*) per convertire funzioni Scala in funzioni Kotlin e viceversa. Questo è stato sufficiente per poter utilizzare il sistema piuttosto agevolmente da entrambi i linguaggi.

Inoltre, sono stati creati anche ulteriori wrapper Scala per semplificare l'utilizzo del sistema attraverso quest'ultimo linguaggio. `ScalaSupport` è stato creato per semplificare l'utilizzo di `Support`, `ScalaExecution` per semplificare l'utilizzo di `Execution` e `DeviceFactory` per semplificare la creazione di piattaforme di esecuzione. La loro esistenza non è strettamente necessaria per poter utilizzare il sistema tramite Scala, ma rendono l'utilizzo da quest'ultimo linguaggio molto più semplice ed immediato.

4.1.6 Comunicazione

Nel package *communication* è contenuta la porzione dedita a rendere possibile la comunicazione tra piattaforme di esecuzione e relativi Client.

La comunicazione è stata modellata tramite l'interfaccia `Communication`, utilizzata da tutti i dispositivi che necessitano di comunicare con un Client. `Communication` si astrae dall'implementazione che verrà effettivamente utilizzata per comunicare ed esprime le funzionalità che ogni metodologia di comunicazione deve fornire.

Il campo `device` indica il dispositivo relativo ad una particolare istanza di `Communication`. Ogni dispositivo che necessita di comunicare ha il proprio oggetto addetto alla comunicazione.

La funzione `startServer` è stata inserita per restare in ascolto di potenziali messaggi in ingresso. Utilizzata principalmente dal Supporto, per attendere messaggi da parte di eventuali Client. Il parametro indica quale azione bisogna intraprendere a fronte di un messaggio in arrivo. Nel caso in cui ad usare questa funzionalità sia il Supporto, è stata predisposta la `serverCallback`, cioè le azioni previste di default che il Supporto deve eseguire a fronte di un messaggio. Questo riduce e semplifica il codice necessario ad inizializzare una rete che resti in ascolto di messaggi da parte di Client esterni, evitando

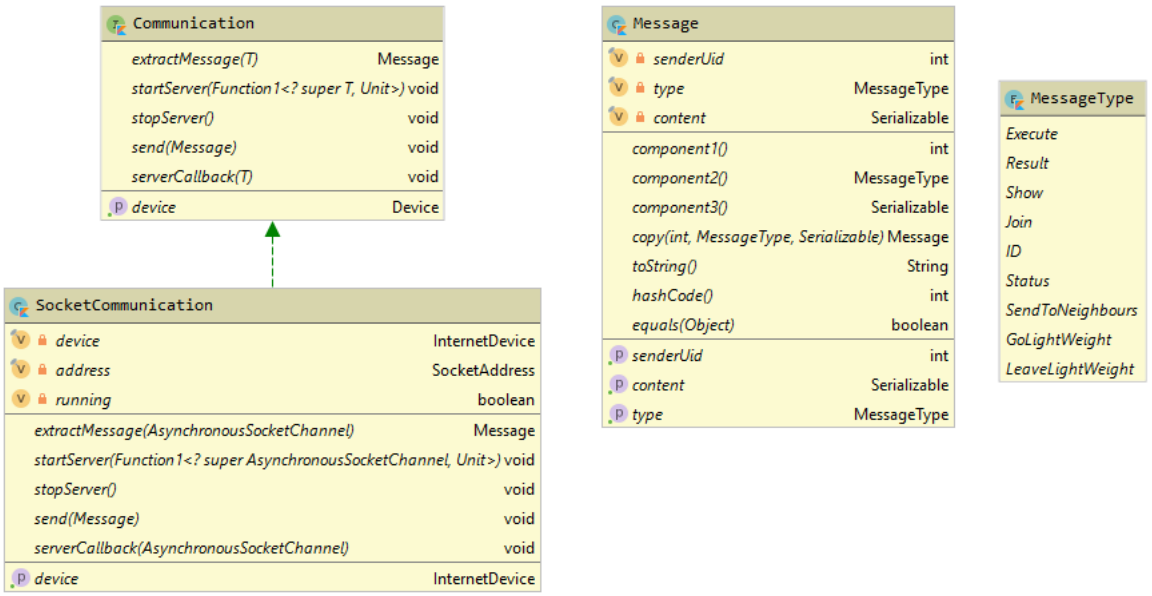


Figura 4.6: Package communication

di specificare la callback per il Supporto, in quanto è molto complesso che le azioni da attuare varino rispetto a quelle di default. `stopServer` è la funzione complementare alla precedente, che chiude il canale di comunicazione.

La funzione `extractMessage` serve ad estrarre il messaggio ricevuto dal canale di comunicazione effettivamente utilizzato. La funzione `send` invia un messaggio al Client associato al dispositivo indicato dal campo `device`.

Comunicazione tramite Socket

È stata prevista un'implementazione che fa uso di Socket per permettere la comunicazione. In questo caso il canale di comunicazione è un `AsynchronousSocketChannel`. La definizione di `serverCallback` indica che, se il messaggio rappresenta la volontà di entrare in rete da parte di un client, viene estratto l'indirizzo di partenza della comunicazione (l'indirizzo del client) e viene reindirizzata la richiesta al Supporto, che la gestirà come detto in precedenza. In tutti gli altri casi, il messaggio viene reindirizzato direttamente al Supporto e gestito come specificato in 4.1.3.

Messaggi

Gli oggetti scambiati nelle comunicazioni tra le varie entità sono stati modellati tramite la classe `Message`. Un messaggio contiene al suo interno l'id del mittente, la tipologia

di messaggio ed il contenuto del medesimo.

Per evitare ogni tipo di problematica dovuta alle potenziali diverse rappresentazioni delle tipologie di messaggi nei vari membri della rete, quelle disponibili sono state definite a priori tramite un enumeratore. In base alla tipologia di messaggio varia anche il significato del contenuto.

- `Execute`: messaggio che rappresenta un ordine di esecuzione.

  Viene inviato dal Supporto quando le piattaforme devono iniziare ad eseguire il programma aggregato. A loro volta, le piattaforme di esecuzione remota inoltrano il messaggio al Client per far sì che questi ultimi possano iniziare ad eseguire.

  Il contenuto è nullo in quanto tutte le informazioni relative all'esecuzione sono già presenti all'interno delle varie piattaforme di esecuzione.

- `Result`: messaggio contenente il risultato di una computazione Scafi.

  Utilizzato da tutti i dispositivi per condividere il loro risultato con i vicini, aggiornando di conseguenza il loro stato.

- `Status`: messaggio contenente un aggiornamento di stato, utilizzato dal gestore delle comunicazioni Protelis.

  Il contenuto è lo stato aggiornato da condividere.

- `Show`: messaggio contenente un oggetto da far visualizzare al destinatario.

  Utilizzato principalmente dalle piattaforme di esecuzione locale per indicare ai corrispettivi Client di visualizzare i risultati ottenuti. Questa tipologia è stata separata dalle precedenti in quanto è possibile che il risultato da visualizzare non sia esattamente quello utilizzato per aggiornare lo stato delle piattaforme. Così facendo si mantiene una maggiore flessibilità, anche in vista di potenziali nuovi adattatori che potranno essere aggiunti in futuro.

- `Join`: messaggio inviato da un Client per entrare a far parte della rete di dispositivi.

  Il contenuto del messaggio è il numero di porta dove inviare le comunicazioni dirette al Client. L'indirizzo IP viene estrapolato dalla connessione stessa, quindi non è necessario inserirlo nel contenuto del messaggio.

- `ID`: messaggio di risposta al precedente messaggio, inviato dal Supporto al Client. Indica a quest'ultimo l'identificativo assegnato alla sua piattaforma di esecuzione all'interno della rete.

- `SendToNeighbours`: messaggio da inviare ai vicini del mittente.

  Il destinatario è sempre il Supporto, che inoltrerà il contenuto, che è un altro messaggio, ai destinatari finali.

- `GoLightWeight`: messaggio utilizzato da un Client per entrare in modalità di esecuzione locale.

  Il contenuto è nullo in quanto il Server conosce già tutto il necessario per trasformare la piattaforma di esecuzione da remota a locale (4.1.3).

- `LeaveLightWeight`: messaggio utilizzato da un Client per uscire dalla modalità di esecuzione locale.

  Il contenuto è ancora una volta nullo, in quanto il Client conosce già tutto il necessario per ricominciare ad eseguire.

Tipologia di messaggio	**Contenuto**
Execute	-
Result	il risultato effettivo
Show	l'oggetto da visualizzare
Status	lo stato aggiornato
Join	la porta da utilizzare
ID	l'id assegnato
SendToNeighbours	il messaggio da inoltrare
GoLightWeight	-
LeaveLightWeight	-

Tabella 4.1: Tipologie di messaggi previste

Serializzazione

Dovendo comunicare anche attraverso oggetti complessi tramite Socket, è fondamentale che tali oggetti siano serializzabili per poter essere inviati e ricevuti in modo corretto. Nella maggior parte dei casi questo non ha causato problemi in quanto gli oggetti da inviare sono risultati serializzabili per loro natura. L'unico caso problematico è dato dagli **Export** generati dall'esecuzione Scafi. Un export deve necessariamente mantenere riferimenti a diverse classi interne di Scafi, ed alcune di esse non sono serializzabili.

È possibile definire una serializzazione personalizzata per un oggetto Java facendolo estendere dall'interfaccia `Serializable` e definendo al suo interno le funzioni `readObject` e `writeObject`. `writeObject` specifica come serializzare l'oggetto in questione scrivendo i valori necessari alla sua ricostruzione in uno Stream. `readObject` è la funzione complementare, che specifica come deserializzare un oggetto estraendo i valori da uno Stream.

Il problema dell'approccio appena descritto è la necessità di poter modificare il sorgente dell'oggetto da serializzare. Nel caso specifico, l'oggetto problematico è `Export`, definito all'interno di Scafi, quindi si è impossibilitati a modificare il suo codice.

Per risolvere il problema è stato necessario definire un ulteriore oggetto, chiamato `ExportWrapper`, funzionante da involucro di export, all'interno del quale specificare la serializzazione di Export attraverso i campi e le funzioni messe a disposizione.

La parte da serializzare di un export (figura 4.7) contiene un insieme di coppie formate da percorso (`Path`) e valore. Un percorso è formato da un insieme (anche vuoto) di slot. Uno slot può essere di una di cinque tipologie: `Nbr, Rep, Scope, FoldHood` e *FunCall*.

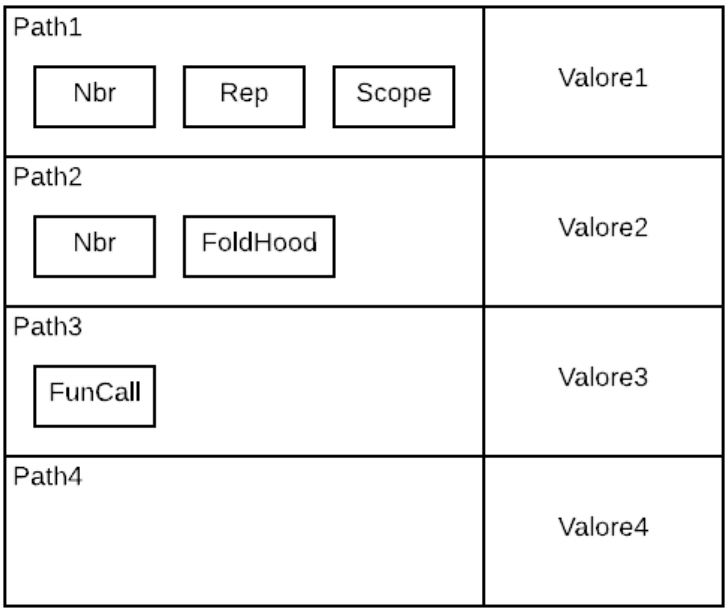


Figura 4.7: Struttura di un Export

La serializzazione è stata definita navigando nella struttura dell'Export, prima scorrendo le `path`, poi gli slot di ogni percorso, salvando di man in mano i valori necessari alla deserializzazione dell'oggetto, La deserializzazione, computata dal ricevente, avviene in maniera complementare alla serializzazione, leggendo i valori salvati in nello stesso ordine con il quale erano stati scritti.

4.1.7 Utilizzo del sistema (senza Client)

Di seguito verrà illustrato l'utilizzo tipo del sistema attraverso dispositivi virtualizzati (l'utilizzo con Client verrà mostrato in 4.2.3). I passi da seguire sono:

1. creazione delle piattaforme di esecuzione, in questo caso dei `VirtualDevice`, e dei relativi adattatori.

2. finalizzazione della topologia, scegliendo quale "forma" dare alla rete.

3. eseguire il programma aggregato richiamando il metodo `execute` di `Support`.

Definizione dell'adattatore Protelis

Per creare un adattatore Protelis bisogna definire un contesto ed il programma Protelis da eseguire. Il contesto, come già espresso in 4.1.4, può essere definito estendendo `ProtelisContext`, definendo tutte le funzionalità che si vogliono rendere disponibili in fase di esecuzione. Se non serve alcuna funzionalità particolare, si può anche utilizzare direttamente `SimpleProtelisContext` come contesto, evitando di definirne uno.

Il programma aggregato va definito utilizzando il linguaggio Protelis, salvato in un file di testo con estensione *.pt* ed inserito tra le risorse del progetto. È possibile anche specificare il programma come stringa ed utilizzarla al posto del nome del modulo, quindi senza utilizzare un file esterno, ma non è una pratica consigliata, se non altro per la qualità del codice che ne risulterebbe.

In seguito, è sufficiente fornire al costruttore dell'adattatore Protelis il nome del modulo da caricare ed il contesto da utilizzare.

```
// nome del modulo Protelis da caricare
val module = "moduleOne"
/* strategia di creazione del contesto
 * specifica come creare il contesto utilizzando il riferimento al
 * dispositivo ed il NetworkManager utilizzato */
val context = ::SimpleProtelisContext
/* strategia di creazione del NetworkManager
 * specifica come creare il NetworkManager utilizzando
 * il riferimento al dispositivo */
val netManager = { ProtelisNetworkManager(it) }

//adattatore da utilizzare in questo esperimento
Execution.adapter = {
    // it = riferimento al dispositivo
    /* Se context non viene specificato viene utilizzato
     * SimpleProtelisContext */
    /* Se netManager non viene specificato viene
     * utilizzato ProtelisNetworkManager */
    ProtelisAdapter(it, module, context, netManager)
}

// Creazione ed inserimento nella rete attraverso createAndAddDevice.
/* Viene automaticamente utilizzato Execution.adapter
 * se non specificato diversamente */
Support.devices.createAndAddDevice(::VirtualDevice)
```

```
/*Creazione di ulteriori piattaforme di esecuzione*/
...
/*Decisione della topologia di rete*/
Support.devices.finalize(Topology.Ring)
/*Esecuzione*/
Support.execute()
```

Listato 4.1: Utilizzo di `VirtualDevice` con adattatori Protelis

Definizione dell'adattatore Scafi

Per utilizzare un adattatore Scafi è sufficiente definire il programma aggregato, creando un oggetto che estenda da `AggregateProgram`, e fornirlo al costruttore dell'adattatore. L'API Scala fornita da `ScalaExecution` velocizza il settaggio dell'adattatore Scafi nell'oggetto `Execution`.

```
// Definizione del programma
// utilizzando ScafiIncarnation come incarnazione
class Program extends ScafiIncarnation.AggregateProgram {
    override def main() = mid()
}

//adattatore da utilizzare in questo esperimento
ScalaExecution.setAdapter(ScafiAdapter(_, new Program()))

ScalaSupport.createAndAddDevice(id =>
    //creazione di un VirtualDevice attraverso la factory Scala
    DeviceFactory.virtual(
        id, // id auto-generato
        "Device" + id, // nome da assegnare al dispositivo
    )
)

// Esecuzione
Support.INSTANCE.execute()
```

Listato 4.2: Utilizzo di `VirtualDevice` con adattatori Scafi

4.2 Sotto-sistema Client

Per quanto riguarda la parte Client, è stato concordato di svilupparla su supporto Android, utilizzando Kotlin come linguaggio.

Come accaduto per la parte relativa al server, è stato creato un repository[2] dedicato anche alla parte client.

Avendo modellato i componenti della parte Server tenendo in considerazione del fatto che poi avrebbero dovuto essere utilizzati anche dai Client ha reso il codice necessario a questi ultimi per funzionare minimale.

Oltre al codice specifico di Android per inizializzare le viste e per scrivere a video i risultati delle computazioni, i componenti necessari all'utilizzo di un Client sono il **modello del Client** (la piattaforma di esecuzione che eseguirà il programma), il **modello del Server** (utile ad indicare dove inviare le comunicazioni quando necessario) ed il **mezzo di comunicazione** da utilizzare.

Modello del Client La piattaforma di esecuzione è modellata come un `VirtualDevice` (4.1.2). Il comportamento che un client deve assumere a fronte di un messaggio di esecuzione è esattamente quello di un dispositivo completamente emulato all'interno del Server.

Modello del Server In maniera similare, il Server viene visto come un comune `RemoteDevice`, al quale è possibile inviare messaggi semplicemente tramite la funzione `tell`.

Comunicazione con il Server Il mezzo di comunicazione è stato rappresentato attraverso l'oggetto `ClientCommunication`. Le azioni che ogni mezzo di comunicazione deve rendere disponibili sono `listen` (un Client deve poter restare in ascolto di messaggi in arrivo dal Server), `stop` (funzione complementare alla precedente, un client può chiudere il canale di comunicazione) e `subscribeToServer` (un client deve poter contattare un Server esprimendo la volontà di entrare nella rete dei dispositivi).

Similmente a quanto era avvenuto nel Server (4.1.6), l'implementazione fornita nella classe `SocketClientCommunication` fa uso di Socket. Al suo interno è stato fortemente riutilizzato il codice già scritto in `SocketCommunication` per mettere il Client in ascolto di messaggi. L'unica porzione ad essere stata definita ex-novo è stata la callback da eseguire a fronte di messaggi in arrivo. In caso di messaggi di tipologia ID, viene istanziato il modello del Client, in tutti gli altri casi, il messaggio viene reindirizzato direttamente al modello del Client, che verrà gestito con le stesse modalità espresse in 4.1.2.

[2] `https://github.com/LorisCangini/Aggregate-Computing-Client`

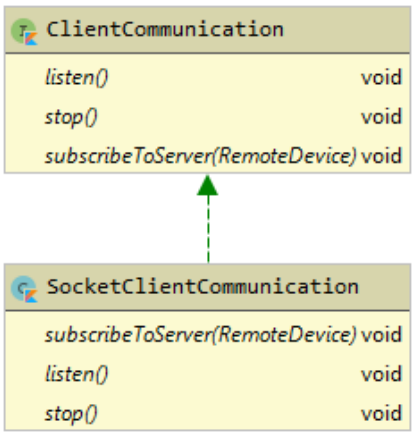


Figura 4.8: Comunicazione Client-Server

4.2.1 Adattatore Protelis

L'utilizzo di adattatori Protelis in ambiente Android ha evidenziato problemi di compatibilità tra alcuni framework utilizzati (*log4j* e *Guice*) e l'SDK stesso.

Logging for Java (log4j)

Come specificato in 1.2, il parsing del programma Protelis è stato effettuato utilizzando il framework **XText**. Tra le dipendenze di XText è presente **log4j**, una libreria largamente utilizzata in programmi Java per effettuare il logging, cioè per assicurarsi di salvare informazioni sulle esecuzioni effettuate, così da poter sempre essere in grado di ricostruire l'avvenuto.

Log4j utilizza al suo interno anche oggetti provenienti dal package `java.beans`. Tale package è completamente disponibile solo nel JDK utilizzabile in ambienti desktop, mentre, per quanto riguarda l'SDK Android, `java.beans` è presente a livello di package, ma quasi tutti gli oggetti al suo interno non sono stati inseriti. Questo rende di fatto inutilizzabili tutte le librerie facenti uso di oggetti interni al suddetto package su Android, log4j incluso.

Simple Logging Facade for Java (slf4j)

slf4j[3] è una libreria che fornisce astrazioni verso i framework di logging più utilizzati, tra i quali è compreso *log4j*. Più precisamente, tramite dei `Bridge`, reindirizza le chiamate

[3] `http://www.slf4j.org/`

dei vari framework ad oggetti slf4j che svolgono il lavoro al loro posto, tutto ciò senza cambiare il codice facente uso delle API fornite dei vari framework di logging.

Bridge log4j-over-slf4j

Tra i vari Bridge disponibili, *log4j-over-slf4j*[4] è quello utilizzato per reindirizzare chiamate da *log4j* a *slf4j*.

Utilizzando questa dipendenza in ambiente Android è possibile quindi sfruttare codice utilizzante *log4j*, evitando però le chiamate interne a *java.beans*, in quanto ad eseguire il vero e proprio logging è *slf4j*.

Per poter sfruttare il Bridge *slf4j* e non il framework *log4j* utilizzato nativamente da XText è necessario specificare alcune opzioni tramite Gradle. Le azioni da attuare sono due:

- va esclusa la dipendenza transitiva di Protelis da *log4j*.
- va indicato il Bridge *log4j-over-slf4j* come dipendenza del Client.

```
implementation('org.protelis:protelis:13.0.3'){ dep ->
    ['log4j'].each{ group -> dep.exclude group: group }
}
implementation('org.slf4j:log4j-over-slf4j:1.7.30')
```

Listato 4.3: Utilizzo di log4j-over-slf4j tramite Gradle

Essendo i Bridge *slf4j* pensati per funzionare senza modificare il codice, questo è sufficiente per eliminare tutte le incompatibilità di *log4j* con Android. Tutte le chiamate fatte da XText a *log4j* vengono automaticamente gestite da *slf4j*, eliminando ogni problema precedentemente riscontrato.

Incompatibilità Guice

Risolto il problema relativo a *log4j* si è presentata un'altra incompatibilità, questa volta relativa a ***Guice***[5], un framework di *dependency injection* utilizzato anch'esso all'interno di XText.

La versione standard di Guice fa utilizzo di *Aspect Oriented Programming (AOP)*[6], caratteristica però non supportata da nessuna piattaforma mobile (Android compreso).

[4] http://www.slf4j.org/legacy.html#log4j-over-slf4j

[5] https://github.com/google/guice

[6] https://github.com/google/guice/wiki/AOP#aspect-oriented-programming

AOP opzionale Guice fornisce anche una versione della propria libreria senza AOP, quindi utilizzabile anche in ambienti mobile.

Il procedimento per poter utilizzare la versione di Guice senza supporto per AOP è lo stesso utilizzato per utilizzare *slf4j* al posto di *log4j*:

- si esclude la dipendenza transitiva di Protelis da Guice.
- si specifica la versione di Guice senza AOP come dipendenza del Client Android.

```
implementation('org.protelis:protelis:13.0.3'){ dep ->
    ['com.google.inject'].each{ group -> dep.exclude group: group }
}
implementation 'com.google.inject:guice:4.0-beta:no_aop'
```

Listato 4.4: Utilizzo di Guice senza AOP tramite Gradle

Questo rende possibile l'utilizzo di XText anche in ambienti Android. Nel caso specifico, queste azioni hanno reso possibile il parsing di programmi Protelis, e quindi la loro esecuzione, effettuata sempre come illustrato in 1.2.1 e 4.1.7.

4.2.2 Adattatore Scafi

La definizione di un adattatore Scafi in ambiente Android è risultata problematica per la natura stessa di Scafi.

Incompatibilità Scala-Kotlin su Android

Come già detto in 4.1.4, sia Scafi che il suo adattatore sono stati definiti utilizzando Scala, quindi anche le API fornite sono state pensate per essere utilizzate tramite lo stesso linguaggio di programmazione. Questo è stato un problema di rilevante importanza, in quanto l'utilizzo simultaneo di Kotlin e Scala in ambiente Android non è supportato in alcun modo da alcun plugin.

Più precisamente, Kotlin è diventato lo standard di fatto come linguaggio di sviluppo di applicazioni Android, in quanto permette di creare applicazioni più velocemente e con meno codice rispetto al corrispettivo (e nativo) codice Java. Altro standard di fatto in ambiente Android è il build tool **Gradle**, utilizzato da praticamente tutte le applicazioni per gestire le dipendenze da includere. Scala, al contrario, non viene praticamente utilizzato in ambiente Android, se non per piccoli esperimenti. I tentativi di rendere il linguaggio utilizzabile in ambiente Android sono pochi e non aggiornati, soprattutto per quanto riguarda il supporto fornito tramite Gradle (Scala è spesso utilizzato con sbt, un build tool scritto specificatamente per tale linguaggio).

Per risolvere la situazione ed essere in grado di definire adattatori Scafi anche su Android si è pensato di costruire un API Kotlin per permettere l'utilizzo di Scafi anche da quest'ultimo linguaggio.

Utilizzo di Scafi da Kotlin

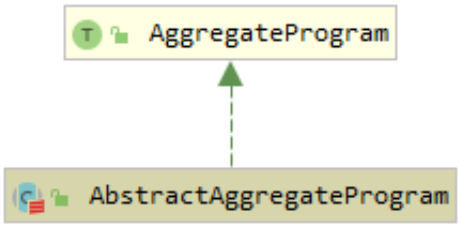
Il problema principale riguardo l'utilizzo di Scafi da Kotlin è la definizione del programma aggregato. Come già spiegato in 4.1.4, un programma Scafi viene definito estendendo l'interfaccia `AggregateProgram`. Fare ciò direttamente da Kotlin non si è dimostrato fattibile, in quanto la traduzione in bytecode (4.1.5) del codice Scafi richiede, alla parte scritta in Kotlin, di specificare implementazioni di svariate funzioni auto-generate, alcune delle quali mostrate nel listato 4.5.

```
class Program : Incarnation.AggregateProgram {
    override fun main(): Any  = /*main program body*/

    override fun vm(): Semantics.RoundVM = ???
    override fun `vm_$eq`(vm: Semantics.RoundVM?)  = ???
    override fun `apply$mcDD$sp`(p0: Double): Double = ???
    override fun `apply$mcJD$sp`(p0: Double): Long = ???
    override fun `apply$mcIF$sp`(p0: Float): Int = ???
    override fun `apply$mcFJ$sp`(p0: Long): Float = ???
    override fun `apply$mcZF$sp`(p0: Float): Boolean = ???
    override fun `apply$mcZI$sp`(p0: Int): Boolean = ???
    override fun `apply$mcVF$sp`(p0: Float): Unit = ???
    ...
}
```

Listato 4.5: implementazioni richieste da `AggregateProgram`

Per risolvere il problema, è stato necessario frapporre un ulteriore livello di astrazione tra l'utilizzo in Kotlin e le API in Scala fornite da Scafi.



È stata definita, sempre utilizzando Scala, la classe astratta `AbstractAggregateProgram`, che estende da `AggregateProgram` senza specificare alcunché. La parziale implementazione in Scala tramite la classe astratta permette alla parte Kotlin di riconoscere, quindi non richiedere, le implementazioni delle funzioni mostrate nel listato 4.5, in quanto generate anch'esse dalla traduzione in bytecode di `AbstractAggregateProgram`. Utilizzando Kotlin ed estendendo da questa nuova classe, invece che da `AggregateProgram`, è possibile definire programmi aggregati Scafi semplicemente specificando, come previsto, solamente la funzione `main`.

La classe appena definita, assieme alle classi di supporto spiegate in 4.1.5, ha permesso di utilizzare Scafi direttamente tramite Kotlin, quindi senza utilizzare Scala. Questo ha reso possibile definire ed utilizzare adattatori Scafi anche su Android, permettendo così l'interazione tra Client e Server così come era stata prevista in fase di progettazione.

```
// creazione del programma aggregato
// utilizzando AbstractAggregateProgram
class Program : AbstractAggregateProgram() {
    override fun main(): Any = mid()
}
```

Listato 4.6: Creazione di un programma Scafi attarverso Kotlin

In questo caso non è necessario esplicitare quale incarnazione utilizzare in quanto è stata indicata in `AbstractAggregateProgram`. Le parti di creazione dell'adattatore, di scelta della topologia e di esecuzione non cambiano rispetto a quanto mostrato nel listato 4.1, con l'unica differenza che invece di creare un `ProtelisAdapter` bisogna creare uno `ScafiAdapter`.

4.2.3 Utilizzo del sistema (con Client)

Per quanto riguarda il server, per poter utilizzare il sistema sfruttando anche piattaforme di esecuzione remota, i passi da seguire sono:

1. mettere in ascolto il Supporto, in attesa di messaggi da parte di eventuali Client.
2. finalizzare la rete
3. lanciare l'esecuzione

Mentre per quanto riguarda il Client:

1. contattare il Server per entrare in rete
2. mettersi in ascolto di eventuali messaggi da parte del Server

Server Per quanto riguarda la parte relativa al Server, le funzionalità necessarie sono state introdotte in 4.1.3 e 4.1.6.

```
//il Supporto viene messo in ascolto di messaggi
//utilizzando la callback di default
Support.physicalDevice.startServer()

//fase di attesa dei Client... senza fare nulla

//una volta che i Client desiderati hanno aderito alla rete
//si sceglie la topologia
Support.devices.finalize(Topology.FullyConnected)

//e si esegue allo stesso modo
Support.execute()
```

Listato 4.7: Utilizzo con Client (parte Server)

È possibile anche utilizzare differenti tipologie di piattaforme di esecuzione con lo stesso programma aggregato. L'utilizzo non cambia, è sufficiente definire tutti i dispositivi che si vogliono utilizzare prima di finalizzare la rete. Nell'esempio precedente, se si volessero utilizzare anche delle piattaforme virtualizzate assieme al Client, sarebbe stato sufficiente creare dei `VirtualDevice` come mostrato nel listato 4.1 prima dell'istruzione `finalize`. Tutto il resto viene automaticamente gestito dal sistema in autonomia.

Client Per quanto riguarda il Client, le funzionalità necessarie sono già state introdotte in 4.2.

```
// definizione del server come un RemoteDevice
// serverAddress sono indirizzo IP e porta da contattare
/* serverId e serverName sono id e nome da assegnare al server,
 * ma non sono utilizzati in alcun modo */
val server = RemoteDevice(serverId, serverAddress, serverName)

//definizione del Client come VirtualDevice
//viene inizializzato dopo aver ricevuto l'ID dal Server
private lateinit var client: VirtualDevice

Execution.adapter = { /*adattatore da utilizzare*/ }

//definizione del mezzo di comunicazione
val listener = SocketClientCommunication(
    myAddress, //indirizzo del Client
    //callback da utilizzare in caso di messaggio ID
    onID = { id ->
        //inizializzazione di client
        client = VirtualDevice(id, "Name", Execution.adapter,
            { result -> /*mostra il risultato*/ }
        )
    },
    //callback per tutti gli altri messaggi
    onMessage = { client.tell(it) }
)

//invio di un messaggio di tipo Join al Server
listener.subscribeToServer(server)
//messa in ascolto di messaggi da parte del Server
listener.listen()
```

Listato 4.8: Utilizzo con Client (parte Client)

Il modello del client, trattato come `VirtualDevice`, si occupa di gestire tutti i messaggi in ingresso, compresa l'esecuzione.

Il server viene visto semplicemente come un `RemoteDevice`, collegato al Supporto. Qualora sia necessario comunicare con il Server è quindi sufficiente utilizzare la funzione

`tell` del modello appena citato.

`listener` è l'oggetto di comunicazione tra il client e il server, in questo caso definito come `SocketClientCommunication`. Come già esplicitato in 4.2. le uniche parti da definire sono le reazioni da eseguire a fronte della ricezione di un messaggio.

La callback `onID` viene eseguita alla ricezione di un messaggio di tipo ID, che viene utilizzato solamente in fase di inizializzazione della rete per comunicare al client il suo identificativo. Nel caso specifico, ricevuto l'id viene solamente definito il modello del client.

`onMessage` viene eseguita a fronte di qualsiasi altra tipologia di messaggio, il quale viene semplicemente reindirizzato al modello del client.

Capitolo 5

Validazione

Nel seguente capitolo verrà verificato il soddisfacimento dei requisiti alla luce di quanto implementato. Verranno inoltre specificati quali test sono stati effettuati, su quali parti per progetto, e le modalità di esecuzione di questi ultimi.

5.1 Verifica dei requisiti

Indipendenza dall'implementazione di Field Calculus L'indipendenza dall'implementazione effettiva di *Field Calculus* è stata garantita relegando tutta la parte relativa ai framework ed ai linguaggi da utilizzare all'interno di **adattatori**. Le varie piattaforme di esecuzione utilizzano solamente l'astrazione di più alto livello di tali adattatori, garantendo così totale indipendenza dalle varie implementazioni.

Apertura verso future implementazioni Per lo stesso motivo, il sistema è aperto a nuove implementazioni, in quanto è sufficiente definire nuovi adattatori per rendere tutta l'infrastruttura compatibile con nuovi framework e linguaggi.

Compatibilità con Scafi e Protelis Sia Scafi che Protelis sono stati supportati sempre utilizzando degli adattatori, fornendo in ognuno il codice necessario per far interagire le due implementazioni con il sistema sviluppato.

Topologia non nota ai dispositivi La topologia è nota solamente al Supporto attraverso l'oggetto `DeviceManager`. Qualora una piattaforma o un Client necessiti di comunicare con un vicino, prima viene inviato al Supporto un messaggio di tipo `SendToNeighbours` con il vero e proprio messaggio da inviare come oggetto. Il supporto, ad avvenuta ricezione del messaggio, estrae il contenuto e lo inoltra ai vicini del mittente.

Diverse modalità di esecuzione Le diverse modalità di esecuzione previste (esecuzione locale, esecuzione remota e virtualizzata) sono state previste creando per ognuna una **piattaforma di esecuzione**.

Una piattaforma di esecuzione è il modello di un dispositivo in grado di eseguire un programma aggregato. Le tre piattaforme corrispondenti alle modalità sopracitate sono:

Piattaforma	Modalità
`LocalExecutionDevice`	Esecuzione locale
`RemoteDevice`	Esecuzione remota
`VirtualDevice`	Esecuzione virtualizzata

Tabella 5.1: Piattaforme di esecuzione

Programma aggregato noto al Server Il Server conosce sempre il programma aggregato attraverso l'oggetto `Execution`. Tale oggetto contiene la strategia di creazione dell'adattatore da utilizzare qualora necessario, che viene specificata all'inizio di ogni esperimento (4.1.7). Ogni adattatore, per poter essere creato, richiede, tra le altre cose, il programma da eseguire, rendendo così noto al Server il programma da eseguire.

5.2 Test effettuati

L'intera fase di sviluppo è stata portata avanti seguendo un approccio test-driven. Per ogni componente o funzionalità, prima di procedere all'implementazione, sono stati scritti alcuni test che esprimessero il comportamento desiderato. L'implementazione può dirsi corretta quando tutti i test scritti in precedenza vengono valutati senza errori.

Tutti i test sono stati scritti utilizzando il framework **JUnit**[1], un sistema di testing ideato inizialmente per testare programmi Java, ma utilizzabile senza problemi anche per verificare implementazioni in Kotlin e Scala.

La maggior parte dei test sono stati scritti in Kotlin, in quanto il sistema è stato sviluppato prevalentemente in quel linguaggio, ma sono presenti anche alcuni test definiti tramite Scala, utilizzati principalmente per controllare la parte relativa all'adattatore Scafi. I test sono stati raggruppati seguendo la stessa divisione in package utilizzata per lo sviluppo (4.1.1).

[1] `https://junit.org/junit5/`

Package `devices` In questo package sono stati inseriti i test relativi al funzionamento delle piattaforme di esecuzione (non degli adattatori utilizzati al loro interno).

Più precisamente, i test inseriti sono stati 2:

- `RemoteToLocalTest`: verifica della parte relativa al passaggio di modalità da parte di una piattaforma di esecuzione da remota a locale. Simulando l'arrivo di un messaggio da parte di un Client, è stata verificata la trasformazione della piattaforma da `RemoteDevice` a `LocalExecutionDevice`. La nuova piattaforma di esecuzione deve mantenere lo stato le relazioni di vicinanza della vecchia. Inoltre le piattaforme vicine a quella che sta attuando la trasformazione devono considerare quella appena creata come loro vicino, non più quella ad esecuzione remota.

- `LocalToRemoteTest`: similmente a quanto verificato col precedente test, questo è stato utilizzato per controllare il passaggio da dispositivo da esecuzione locale a remota. Le proprietà da verificare e le modalità di svolgimento del test sono invariate rispetto al precedente.

Package `server` Qua sono stati inseriti i test relativi al Supporto ed al gestore della topologia.

- `DeviceManagerTest`: utilizzato per verificare il funzionamento del gestore della topologia. Più precisamente, sono state verificate le finalizzazioni della rete utilizzando le varie topologie previste, controllando le relazioni di vicinanza che venivano create di volta in volta. Sono stati considerati anche casi limite, con 1 o 2 dispositivi totali nella rete, ed il comportamento è stato quello voluto.

- `ScalaSupportTest`: verifica del funzionamento dell'API fornita per l'utilizzo del Supporto e da Scala. È stata verificata principalmente la possibilità di inserire dispositivi in rete, di finalizzare la rete stessa e di ottenere le relazioni di vicinanza.

Package `adapters` Verifiche del funzionamento dei vari adattatori utilizzati o di loro porzioni.

- `ProtelisAdapterTest`: è stato verificato il funzionamento dell'adattatore Protelis. Simulando un utilizzo tipo con piattaforme di esecuzione virtualizzate, come indicato in 4.1.7, è stata verificata la possibilità di eseguire il programma aggregato specificato. Come già specificato in 4.1.3, per iniziare ad eseguire il programma aggregato è sufficiente invocare la funzione *execute* del Supporto. La verifica effettuata dal test è la capacità del sistema di portare a termine l'esecuzione utilizzando i dispositivi attualmente in rete con la topologia indicata in fase di finalizzazione.

- `ScafiAdapterTest`: similmente al precedente, è stato verificato anche l'adattatore Scafi alla stessa maniera. L'unica differenza sostanziale è il fatto che questo test è stato scritto in Scala per meglio utilizzare le API Scafi.

- `ScafiFromKotlinTest`: verifica della possibilità di utilizzare adattatori Scafi anche tramite Kotlin. La correttezza delle azioni attuate per consentire l'utilizzo di Scafi da Kotlin (4.2.2) è stata verificata procedendo come nei test descritti in precedenza. Definendo un esperimento con piattaforme virtualizzate, e definendo il programma aggregato utilizzando `AbstractAggregateProgram`, è stata verificata la capacità del sistema di portare a termine l'esecuzione.

- `ExportWrapperTest`: verifica della serializzazione personalizzata degli Export Scafi. Per verificare la corretta serializzazione e deserializzazione degli Export, ne è stato prima scritto uno (con quanti più valori e situazioni particolari possibile) in un file esterno, quindi serializzandolo. Il file è stato poi successivamente letto, deserializzando l'Export scritto in precedenza. Completati i due passaggi, è stato verificata l'equivalenza dell'oggetto deserializzato dal file esterno con quello presente prima della serializzazione.

Package *communication* Verifica del corretto funzionamento della comunicazione tra Server e Client. Entrambi i test inclusi seguono la stessa struttura. Utilizzando piattaforme di esecuzione remota, come mostrato in 4.2.3, si è verificato il corretto funzionamento della comunicazione da utilizzare prima e durante l'esecuzione del programma aggregato.

- `ScafiRemoteTest`: utilizzato per verificare le interazioni tra piattaforme di esecuzione e Client intenti ad eseguire un programma Scafi.

- `ProtelisRemoteTest`: come il precedente, ma viene effettuata l'esecuzione di un programma Protelis.

Questi test sono stati utilizzati anche per verificare la possibilità per un client di entrare ed uscire effettivamente dalla modalità di esecuzione locale (non in maniera simulata come era avvenuto nei test `RemoteToLocalTest` e `LocalToRemoteTest`).

I due test appena descritti non possono essere automatizzati per via della necessaria interazione da parte di un Client esterno. Simulando tale interazione si introdurrebbero condizioni non rappresentative della situazione reale.

5.2.1 Continuous Integration

Per automatizzare l'esecuzione dei test preparati per verificare le varie funzionalità è stato utilizzato lo strumento di Continuous Integration **Travis CI**[2]. Tale strumento si integra con GitHub ed esegue automaticamente i task specificati ad ogni aggiornamento del repository.

Nel caso specifico, Travis è stato utilizzato per eseguire automaticamente tutti i test (escludendo i due atti a verificare la connessione con i Client) ad ogni aggiornamento del codice. Questo ha permesso di aggiungere funzionalità in maniera incrementale, verificando automaticamente la compatibilità delle parti nuove con il codice preesistente.

[2] `https://travis-ci.org/`

Capitolo 6

Conclusioni

Al termine del progetto è stato possibile raggiungere tutti gli obiettivi desiderati. È stato possibile sviluppare sia la parte relativa al server (4.1) che al client (4.2) inserendo tutte le funzionalità previste in fase di progettazione. Tutte le modalità previste da requisiti sono state inserite e funzionano come previsto.

6.1 Discussione

Durante la fase di progettazione è stato immaginato il sistema nel suo complesso, identificando i componenti e le interazioni necessarie al sistema per funzionare e soddisfare i requisiti prima di iniziare l'effettiva implementazione. Questo ha permesso di avere ben chiari, durante tutta la fase di sviluppo, quali dovessero essere le interazioni tra i componenti e quali funzionalità inserire in ognuno.

L'analogia iniziale con *Alchemist* ha permesso di individuare la giusta strategia per garantire l'indipendenza del sistema dalle effettive implementazioni da utilizzare in seguito, lasciando aperta la possibilità di aggiungere ulteriori adattatori per altri framework in un secondo momento.

Nella fase di sviluppo si sono dovute affrontare anche problematiche di non previste in fase di progettazione, come l'incompatibilità di Android con alcune parti di Protelis e Scafi e la necessità di avere una serializzazione personalizzata degli Export per poter comunicare risultati tra Server e Client. Nonostante la mancata previsione di tali problemi, è stato possibile risolverli facendo ricorso a nozioni e tecniche apprese durante il corso di laurea, sia triennale che magistrale.

Il fatto di aver adottato un processo di sviluppo *test-driven*, assieme al beneficio introdotto da Travis CI, ha permesso di aggiungere incrementalmente funzionalità, senza preoccuparsi ogni volta di verificare manualmente la compatibilità delle nuove porzioni

con il codice preesistente, riducendo i tempi complessivi di sviluppo e garantendo di man in mano la correttezza di ciò che si stava facendo.

6.2 Sviluppi futuri

Come da requisiti, il sistema è aperto nuove implementazioni di Field Calculus, quindi qualora fosse necessario supportare nuovi framework, il tutto sarebbe possibile semplicemente definendo i relativi adattatori. È inoltre possibile specificare nuove topologie di rete semplicemente definendo come creare le relazioni di vicinanza tra i dispositivi.

Una possibile direzione per sviluppi futuri potrebbe essere l'inserimento di un ulteriore modalità di esecuzione adattativa. In tale modalità i client non devono esplicitamente indicare al server la volontà di entrare o uscire da modalità "lightweight", ma tale decisione verrebbe presa in autonomia valutando le condizioni attuali del dispositivo. Per realizzare tale funzionalità, l'unica differenza rispetto al sistema attuale sarebbe la definizione i criteri secondo i quali effettuare il passaggio. Una volta valutati tali criteri, se soddisfatti, sarebbe sufficiente inviare al Server le stesse tipologie di messaggi utilizzati attualmente.

Una altra evoluzione potrebbe essere il passaggio da un architettura client-server ad una peer-to-peer. Con il sistema attuale sarebbe possibile definire il Server anche su una piattaforma mobile, come Android, ma rimarrebbe comunque un'architettura client-server. Per effettuare il passaggio ad un'architettura peer-to-peer sarebbe necessario rivalutare alcuni dei requisiti, in quanto non ci sarebbe più un unico punto nel quale la topologia di rete è nota. Inoltre tutti i dispositivi dovrebbero essere in grado di inviare messaggi ai loro vicini senza supporto esterno.

Bibliografia

[1] G. Audrito, M. Viroli, F. Damiani, D. Pianini, and J. Beal, *A higher-order calculus of computational fields*, ACM Trans. Comput. Log., 20 (2019).

[2] R. Casadei, D. Pianini, M. Viroli, and A. Natali, *Self-organising coordination regions: A pattern for edge computing*, in Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings, H. R. Nielson and E. Tuosto, eds., vol. 11533 of Lecture Notes in Computer Science, Springer, 2019, pp. 182–199.

[3] R. Casadei and M. Viroli, *Sito web scafi*.

[4] ——, *Towards aggregate programming in scala*, in First Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2016, Rome, Italy, July 17, 2016, ACM, 2016, p. 5.

[5] E. Nardini, A. Omicini, and M. Viroli, *General-purpose coordination abstractions for managing interaction in MAS*, in Proceedings of the 2009 IEEE/WIC/ACM International Conference on Web Intelligence and International Conference on Intelligent Agent Technology - Workshops, Milan, Italy, 15-18 September 2009, IEEE Computer Society, 2009, pp. 501–506.

[6] D. Pianini, J. Beal, and M. Viroli, *Practical aggregate programming with protelis*, in 2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017, IEEE Computer Society, 2017, pp. 391–392.

[7] D. Pianini and M. Francia, *Sito web protelis*.

[8] D. Pianini, S. Montagna, and M. Viroli, *Chemical-oriented simulation of computational systems with ALCHEMIST*, J. Simulation, 7 (2013), pp. 202–215.

[9] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini, *Engineering resilient collective adaptive systems by self-stabilisation*, ACM Trans. Model. Comput. Simul., 28 (2018).

[10] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, *From distributed coordination to field calculus and aggregate computing*, J. Log. Algebraic Methods Program., 109 (2019).

[11] M. Viroli, M. Casadei, and L. Gardelli, *A self-organising solution to the collective sort problem in distributed tuple spaces*, in Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007, Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, and Y. W. Koo, eds., ACM, 2007, pp. 354–359.

[12] M. Viroli, R. Casadei, and D. Pianini, *On execution platforms for large-scale aggregate computing*, in Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp Adjunct 2016, Heidelberg, Germany, September 12-16, 2016, P. Lukowicz, A. Krüger, A. Bulling, Y. Lim, and S. N. Patel, eds., ACM, 2016, pp. 1321–1326.

[13] ——, *Simulating large-scale aggregate mass with alchemist and scala*, in Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016, M. Ganzha, L. A. Maciaszek, and M. Paprzycki, eds., vol. 8 of Annals of Computer Science and Information Systems, IEEE, 2016, pp. 1495–1504.

[14] M. Viroli and A. Ricci, *Tuple-based coordination models in event-based scenarios*, in 22nd International Conference on Distributed Computing Systems, Workshops (ICDCSW '02) July 2-5, 2002, Vienna, Austria, Proceedings, IEEE Computer Society, 2002, pp. 595–601.

Ringraziamenti

Ringrazio tutti coloro che mi hanno supportato e sopportato durante tutto l'arco degli studi!