ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

SVILUPPO DI PIATTAFORME PER IL LINGUAGGIO PROTELIS IN KOTLIN E JAVA

Elaborato in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. DANILO PIANINI

Presentata da
FILIPPO NARDINI

Correlatore
Prof. MIRKO VIROLI

Anno Accademico 2018 – 2019

Alla mia famiglia.

Sommario

L'ambiente da cui siamo circondati tutti i giorni è pervaso da dispositivi in grado di effettuare computazioni e comunicare. L'eterogeneità di questi dispositivi rende difficile sviluppare applicazioni distribuite resilienti e affidabili utilizzando le tecniche dell'ingegneria del software classica. La programmazione aggregata si propone come paradigma di sviluppo che fornisce meccanismi di comunicazione flessibili e robusti tra questi dispositivi, in particolare vengono presi in considerazione il *field calculus* e una sua implementazione pratica: Protelis, un linguaggio di programmazione che supporta tale paradigma di programmazione.

In questo lavoro vengono analizzate le caratteristiche di Protelis, in particolare la sua architettura e i suoi livelli di astrazione. In seguito vengono modellati nuovi concetti, per la realizzazione di un modello ad oggetti riusabile, che possa essere un potenziale punto di partenza per l'implementazione di un'infrastruttura facilmente estendibile, riusabile, e manutenibile. A supporto della flessibilità del modello, sono presentate tre realizzazioni dell'architettura descritta, che eseguono lo stesso programma aggregato, ma finalizzate a descrivere tre scenari d'uso distinti: la prima rappresenta un micro-simulatore, che emula un contesto distribuito, ma sfrutta la memoria condivisa per permettere lo scambio di messaggi tra i nodi; la seconda realizza un'applicazione distribuita utilizzando le socket TCP per la comunicazione; la terza si serve di un broker di messaggistica centrale e sfrutta il protocollo MQTT, un modello di scambio di messaggi utilizzato nell'Internet-of-Things, per implementare la comunicazione. Per provare l'interoperabilità dell'architettura con i linguaggi eseguiti sulla Java Virtual Machine, questi tre scenari sono stati implementati sia in Java che in Kotlin.

Indice

Capitolo 1

Contesto e motivazioni

1.1 Scenario

Nell'ambiente in cui viviamo siamo circondati da sempre più entità computazionali eterogenee tra loro. Device come smartphone, smartwatch, fitness tracker, display pubblici, droni, insegne digitali, sensori di ogni tipo stanno sempre più pervadendo il nostro ambiente quotidiano. L'interazione tra questi dispositivi gioca un ruolo fondamentale in settori emergenti come Internet-of-Things (IoT), Smart City, reti di sensori o più in generale in sistemi collettivi adattivi. Quando si parla di sistemi collettivi si deve tenere in considerazione l'elevato numero di dispositivi diversi di cui essi sono composti e l'eterogeneità che ne deriva in termini di piattaforme, tecnologie, paradigmi di programmazione, protocolli di comunicazione, eccetera. Specifiche come efficienza, organizzazione e la necessità di coordinare i dispositivi, influenzano pesantemente le scelte di design del sistema, che può risultare rigido, costoso da manutenere, estendere, ed eventualmente scalare in dimensione.

Sorge quindi la necessità di un modello di programmazione più ad alto livello, che consenta di astrarre i dettagli di un sistema e possa occuparsi di tutti i requisiti non funzionali come le proprietà di auto-organizzazione e auto-adattività, delegando allo strato sottostante tutte le questioni di più basso livello. Possono essere elencate tre caratteristiche chiave che questo strato dovrebbe presentare [7]:

- i meccanismi di comunicazione dovrebbero essere nascosti e i gli sviluppatori non dovrebbero essere tenuti a interagire con essi;

- la composizione di moduli o sottosistemi dovrebbe essere semplice e trasparente;

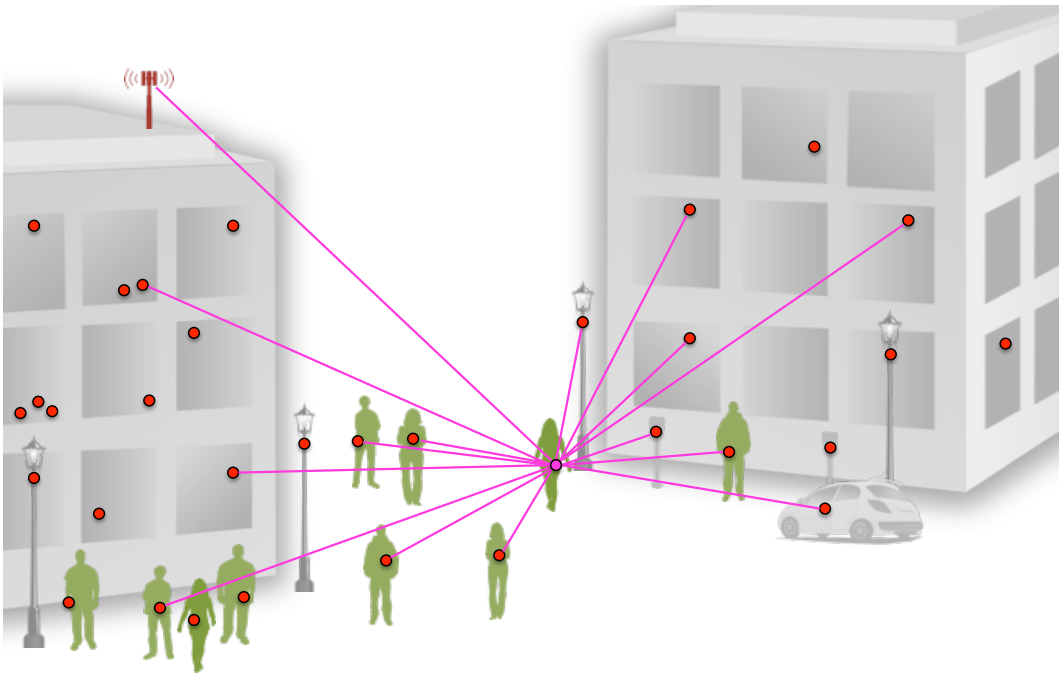


Figura 1.1: Lo spazio che ci circonda è denso di oggetti in grado di effettuare calcoli e capaci di comunicare. Alcuni di questi sono oggetti infrastrutturali, ma la maggior parte sono oggetti personali, in continuo movimento. Immagine tratta da [20].

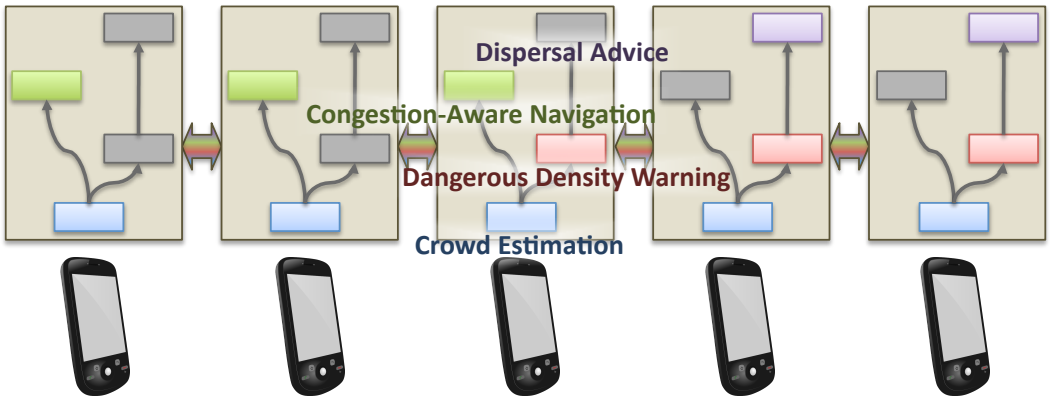
- sottosistemi distinti necessitano di meccanismi di coordinazione per distinte regioni dello spazio-tempo.

1.2 Aggregate computing

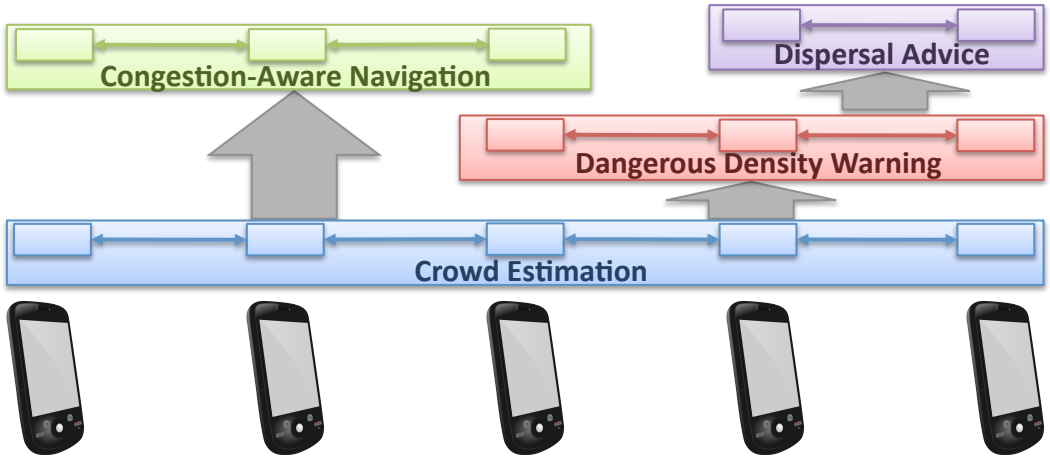
Spesso la modellazione del comportamento collettivo di un sistema interessa più dei singoli dispositivi da cui è composto, ma i linguaggi di programmazione classici, orientati al singolo dispositivo, forzano lo sviluppatore a porre l'attenzione sui singoli device e all'interazione tra di essi. L'*aggregate computing* è un paradigma che si propone di risolvere le precedenti questioni tramite i seguenti principi [7]:

1. il dispositivo programmato è una regione dell'ambiente computazionale e prescinde dagli specifici dettagli;

2. il programma è specificato come manipolazione funzionale di strutture dati distribuite nello spazio-tempo di interesse;



(a) Programmazione device-centric



(b) Programmazione aggregata

Figura 1.2: Differenze tra programmazione orientata al singolo dispositivo e programmazione aggregata. Immagini tratte da [20].

3. queste manipolazioni sono effettivamente eseguite dai singoli dispositivi nella regione, usando meccanismi di coordinazione resilienti e interazioni basate sulla prossimità.

Un esempio di utilizzo nel mondo reale potrebbe essere un servizio che, utilizzando interazioni tra gli smartphone per stimare la densità di popolazione presente in un area o ad un certo evento, possa fornire indicazioni relative a zone che possono essere considerate pericolose perché troppo dense e che percorso seguire eventualmente per disperderle; più in generale, possono essere erogate indicazioni per raggiungere un punto scelto evitando le zone più pericolose.

La Figura 1.2a mostra come con un linguaggio di programmazione orientato al singolo device il programmatore sia costretto a porre la propria attenzione sull'interazione tra i dispositivi, mentre si occupa di modellare localmente un comportamento che dovrà produrre un certo effetto globale. Invece, con l'utilizzo della programmazione aggregata (Figura 1.2b), si è liberi di pensare in maniera più naturale tramite strutture dati *continuum-like* e servizi che possono essere composti in maniera modulare. Nello specifico il servizio di *crowd estimation* produce in output una struttura dati distribuita — un campo computazionale — che associa ad ogni posizione una densità di popolazione. Questo è l'input dei servizi che poi si vogliono costruire: navigazione, segnalazione delle zone di pericolo ed istruzioni per un'eventuale evacuazione.

La capacità di separare la logica dei servizi dall'implementazione della comunicazione e dai protocolli utilizzati per questa, orientano verso lo sviluppo di applicazioni distribuite più complesse, ma allo stesso tempo modulari, riusabili e facilmente estendibili. L'obiettivo della programmazione aggregata è nascondere la complessità di coordinare un sistema distribuito utilizzando diversi livelli di astrazione.

Nel tempo sono stati sviluppati diversi modelli di programmazione aggregata, ma la maggior parte di questi si sono rivelati troppo specializzati per singole istanze di problemi e non sufficientemente generici da risolvere intere classi di questi [6]. Un'eccezione è il *field calculus*: un insieme di costrutti fondamentali, derivati dagli elementi comuni degli altri metodi, che modellano la computazione e l'interazione tra un largo numero di dispositivi sparsi nello spazio.

1.2.1 Field calculus

Utilizzando diversi approcci di programmazione aggregata emergono pattern ripetitivi. Il *field calculus* [25] ne riassume le caratteristiche in una semantica operazionale minimale che da origine ad un linguaggio universale [1].

L'elemento fondamentale del *field calculus* è il campo computazionale, ispirato a concetti fisici come i campi magnetici, che associa ad ogni dispositivo

in rete un valore locale. Ogni valore, funzione o variabile è un campo: per esempio una collezione di sensori di temperatura produce un campo di temperature dell'ambiente, una collezione di accelerometri di smartphone produce un campo di direzioni, una notifica di un'applicazione produce un campo di messaggi.

I campi sono generati e manipolati utilizzando quattro costrutti fondamentali:

- **Funzione**: `b(`$e_1, \ldots, e_n$`)` applica la funzione `b` agli argomenti $e_1, \ldots, e_n$. Queste sono funzioni matematiche, logiche o algoritmiche stateless, oppure sensori, attuatori, funzioni definite dall'utente o importate da librerie.

- **Dinamica e stato**: `rep(x<-v)` $\{s_1; \ldots; s_n\}$ definisce una variabile locale di stato `x` inizializzata con il valore v e aggiornata periodicamente con il risultato dell'esecuzione gli statement $s_1; \ldots; s_n$ che compongono il suo corpo. In questo modo viene definito un campo che cambia nel tempo.

- **Interazione**: `nbr(`s`)` acquisisce un campo che associa ad ogni device tra tutti i dispositivi vicini (incluso se stesso), il loro ultimo valore di s. Questo campo può essere poi processato da funzioni built-in chiamate *hood*, che permettono di ridurre il campo ad un valore, per esempio il minimo.

- **Restrizione del dominio**: `if(`e`)` $\{s_1; \ldots; s_n\}$ `else` $\{s'_1; \ldots; s'_m\}$ partiziona la rete in due regioni disgiunte: dove la e è vero viene eseguito $s_1; \ldots; s_n$, nell'altra parte, invece, è eseguito $s'_1; \ldots; s'_n$. È importante menzionare il fatto che le due diramazioni sono incapsulate e non possono avere effetti al di fuori del loro sottospazio.

Questi costrutti permettono portabilità, indipendenza dall'infrastruttura e modularità.

1.2.2 Building blocks

Il successivo livello di astrazione serve ad aggiungere resilienza. È identificata una collezione di operatori "building block" generici finalizzata a una coordinazione robusta delle applicazioni. I meccanismi di questo livello (quello in mezzo nella Figura 1.3), sono *auto-stabilizzanti*, cioé raggiungono lo stato corretto indipendentemente dallo stato di partenza in un numero finito di passi, caratteristica che preservano quando sono composti tra loro [4].

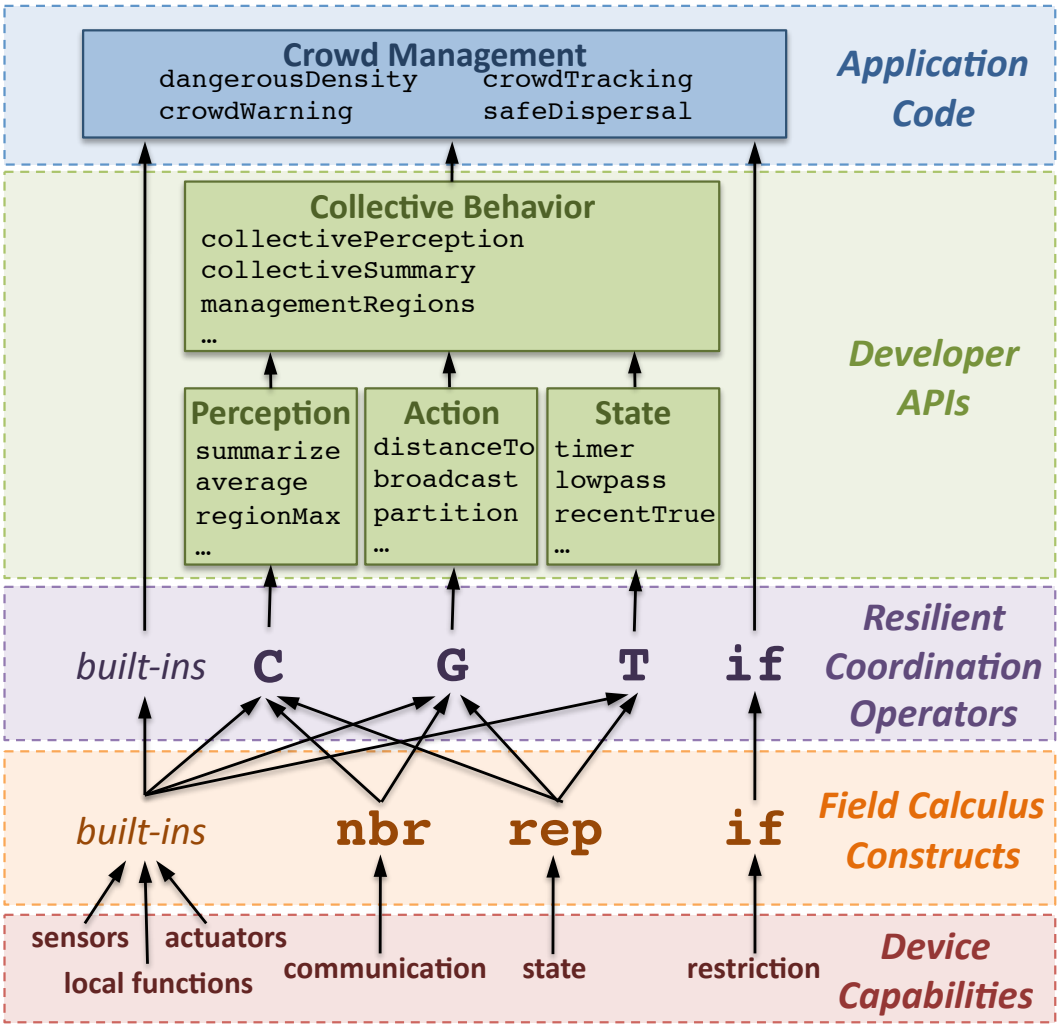


Figura 1.3: Livelli di astrazione dell'aggregate computing. A basso livello le capacità software e hardware vengono astratte e composte per creare un livello comune: il *field calculus*, che è la base di partenza dell'architettura delle API. Immagine tratta da [20].

La collezione è composta da tre operatori di coordinazione generali, che mascherano gli elementi di basso livello del linguaggio, e dal costrutto `if`. I tre operatori sono:

- `G(source, init, metric, accumulate)`: distribuisce un informazione nello spazio. Questo operatore generalizza operazioni molto comuni come la stima della distanza e messaggi broadcast. Per farlo esegue due azioni: computa un campo di distanze minime da una regione `source` utilizzando la `metric` scelta; in seguito propaga i valori attraverso i nodi iniziando da `init` e modificandolo con la funzione `accumulate`.

- `C(potential, accumulate, local, null)`: aggrega un'informazione alla `source` attraversando il gradiente di un campo `potential`. Viene effettuata una riduzione che, iniziando con il valore `null` e sfruttando una funzione associativa `accumulate`, risale il gradiente verso la `sorgente` e combina i valori in `local`.

- `T(init, floor, decay)`: generalizza un timer il cui rateo di aggiornamento può variare nel tempo. La funzione `decay` diminuisce il valore del proprio input successivo, che parte da `initial` e si ferma a `floor`.

1.2.3 General-purpose API

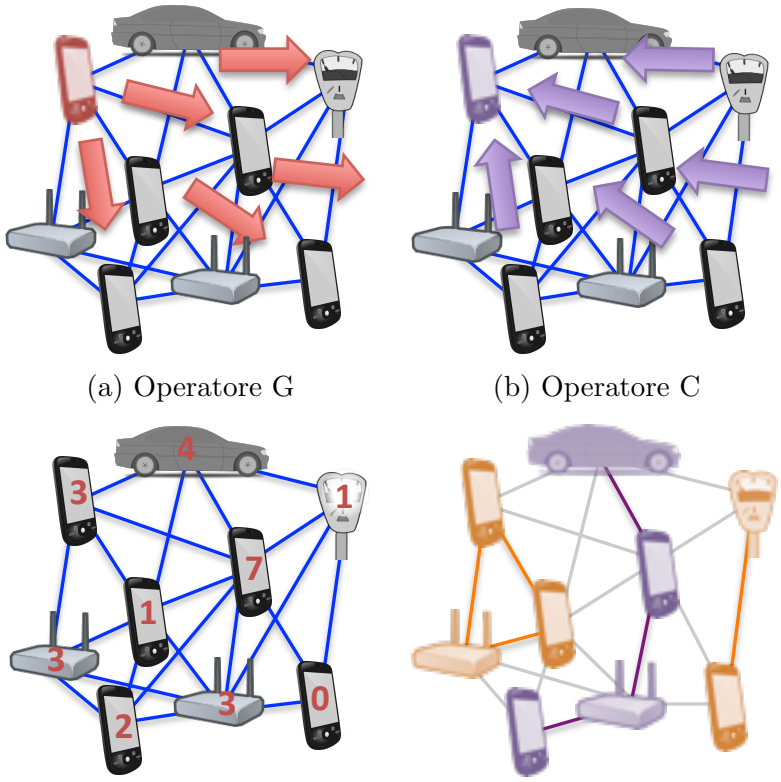
È un ulteriore livello di astrazione (il secondo dall'alto nella Figura 1.3) che mira a semplificare la composizione dei building-blocks, fornendo delle API di alto livello [15]: funzioni come `distanceTo`, che permette di ottenere la distanza tra nodi utilizzando una data metrica, o `broadcast`, che consente di diffondere un messaggio nella rete.

Queste API possono essere usate e composte tra loro per scrivere applicazioni distribuite senza proccuparsi di meccanismi di coordinazione. Infatti sono costruite tramite la composizione degli operatori descritti alla Sezione 1.2.2, dai quali eredita le caratteristiche di robustezza.

Inoltre sono esposti da questo livello dei meccanismi di coordinazione non auto-stabilizzanti, utili in determinati ambiti applicativi.

1.3 Protelis

Il *field calculus* è un fondamento teoretico importante, ma manca di uno strumento pratico che fornisca tool, librerie e possibilità di integrazione con piattaforme esistenti. Protelis [20] è un linguaggio di programmazione che si



(a) Operatore G (b) Operatore C

(c) Operatore T (d) Operatore if

Figura 1.4: "Building block" per computazioni aggregate: questi quattro operatori sono la base fondante di tutte le API auto-stabilizzanti fornite al livello superiore. Immagini tratte da [20].

pone come obiettivo quello di fornire un'implementazione pratica dell'*higher-order field calculus* [2] con la capacità di interfacciarsi con hardware, sistemi operativi e piattaforme esistenti. Poiché la maggior parte dei sistemi che saranno implementati tramite questo linguaggio saranno distribuiti, è di fondamentale importanza che l'implementazione sia facilmente portabile tra un ambiente simulato, nel quale un'applicazione può essere testata, e un ambiente reale, dove questa potrà essere eseguita in produzione.

La sintassi del linguaggio è di facile apprendimento in quanto si ispira a quella dei linguaggi C-like come Java.

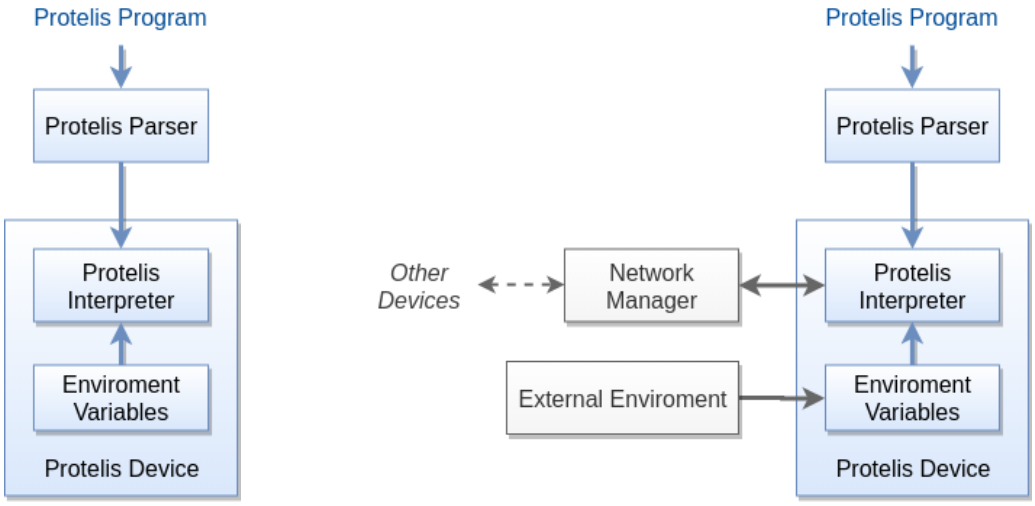
1.3.1 Protelis nel mondo scientifico

Protelis è in grado di esprimere algoritmi distribuiti complessi in poche righe di codice. È infatti stato sfruttato per implementare diversi algoritmi distribuiti. Tra questi i più rilevanti sono: *rendezvous* ad una riunione di massa [20], algoritmo che consente a due persone che partecipano ad un evento di massa di incontrarsi in un punto intermedio, evitando zone di alta densità; stima di pericolosità di una zona per sovrappopolazione e notifica di pericolo [4], che consente di stimare la pericolosità di una zona, basandosi sulla densità di dispositivi presenti in essa, e come disperdere l'eventuale massa di persone in maniera efficiente.

Un altro ambito di utilzzo è la gestione dei servizi in una rete: alcuni servizi di rete sono spesso *legacy* e possono non essere in grado di gestire un errore di altri servizi da cui dipendono. Quindi potrebbero continuare a comportarsi in maniera non definita, modificando in maniera inconsistente il proprio stato. Reagire in maniera coerente ad un errore del genere richiede un sistema di coordinazione tra i servizi coinvolti per riavviare l'intero stack. La programmazione aggregata è stata usata per risolvere il problema in [13].

La realtà aumentata, AR, è un ambito nel quale sono stati fatti tentativi di integrazione con la programmazione aggregata, poiché sono settori di ricerca che hanno in comune l'interazione con l'ambiente reale. In [18] viene introdotto il concetto di campi aumentati, una combinazione di campi computazionali e realtà aumentata. Le possibili applicazioni dei campi aumentati sono la visualizzazione dei campi computazionali tramite visori, o l'interazione dei campi con dati prodotti AR.

Altri contesti in cui Protelis è stato utilizzato sono coordinazione di droni [3], sensor sharing [8] e task allocation [17].



(a) Funzionamento di Protelis

(b) Estendibilità di Protelis

Figura 1.5: Architettura astratta di Protelis (a) e meccanismo di estendibilità previsto dal linguaggio (b). Immagini tratte da [20].

1.3.2 Architettura di Protelis

Protelis è un linguaggio di programmazione aggregato, fortemente influenzato da Proto [5], la cui esecuzione avviene all'interno di una macchina virtuale [20]. Inizialmente un parser traduce un programma Protelis in una rappresentazione comprensibile all'interprete, che può eseguirlo, in seguito, a intervalli regolari. L'interprete può interagire con l'ambiente circostante, implementato tramite coppie *(chiave, valore)* (Figura 1.5a).

La duplice possibilità di poter eseguire un interprete Protelis in due contesti diversi, quali sono un ambiente simulato e quello reale, evidenziano la necessità dell'introduzione di un componente middleware, che gestisca la comunicazione tra le macchine virtuali in maniera completamente trasparente a queste ultime (Figura 1.5b). Sfruttando la caratteristica del polimorfismo dei linguaggi ad oggetti, questo modello può essere facilmente specializzato in entrambi i contesti precedentemente citati.

La piattaforma di supporto scelta per l'implementazione è Java, scelta per la sua portabilità, il meccanismo interno della reflection, e il sempre crescente numero di dispositivi embedded a basso costo che supportano Java. Un altro fattore determinante è l'efficienza in termini di costo di risorse che le implementazioni Java hanno raggiunto, rendendolo competitivo con linguaggi di basso livello come C o C++.

Protelis è interoperabile con Java, e indirettamente con Kotlin, di

conseguenza può essere integrato con un vastissimo ecosistema di librerie.

Capitolo 2

Architettura riusabile per backend di Protelis

2.1 Piattaforme di simulazione

L'architettura utilizzata per Protelis promette di essere flessibile, adatta alla portabilità su diversi sistemi reali [14] e simulati quali Alchemist [19] e NASA World Wind [9].

Alla base dell'elevata portabilità vi è un'architettura a due livelli, che mantiene separati interprete e infrastruttura, e che richiede di implementare determinate interfacce al fine di difinire i meccanismi di comunicazione tra i dispositivi.

2.1.1 Alchemist

Tra le piattaforme esistenti l'esempio più rilevante è Alchemist [19]: un simulatore per computazione pervasiva, aggregata e ispirata alla natura. Esso utilizza al proprio interno un meta-modello nel quale dispositivi vicini e interconnessi comunicano seguendo un insieme di leggi ispirate al mondo della chimica.

Alchemist prevede la possibilità di utilizzare incarnazioni, ovvero implementazioni concrete del proprio meta-modello per modellare uno specifico concetto di interesse. Una di queste incarnazioni è quella che consente al simulatore di eseguire un programma Protelis all'interno della propria infrastruttura, che permette di creare e gestire una rete di nodi simulati (Figura 2.1).

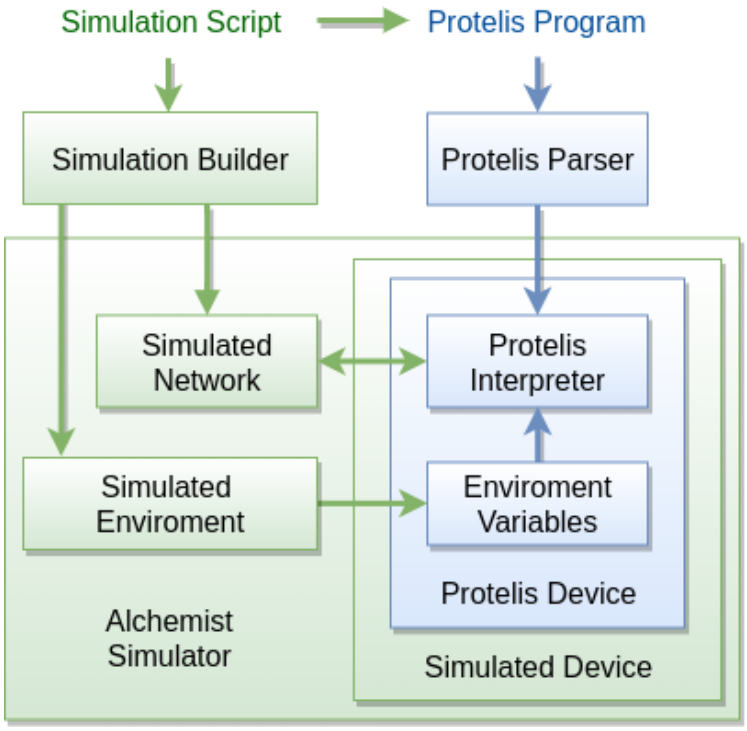


Figura 2.1: Implementazione di Protelis utilizzata in Alchemist.

2.1.2 NASA World Wind

NASA World Wind [9] è un progetto open-source cross-platform sviluppato dalla NASA, che offre un'interfaccia di programmazione per creare, in maniera rapida, delle visualizzazioni 3D interattive di un globo virtuale. Si differenzia da altri software simili come Google Earth[1] perché non è una semplice applicazione, piuttosto un SDK che può essere utilizzata come base per costruire una propria applicazione.

È stato utilizzato per dimostrare come Protelis possa essere uno strumento che permette di controllare anche dispositivi reali come uno sciame di droni. In questa simulazione[2] 25 quadricotteri, disposti a griglia, volano a qualche centinaio di metri da terra. Essi fanno uso di comunicazione a corto raggio, 500 metri, per parlare con i dispositivi adiacenti.

[1] https://www.google.it/intl/it/earth/
[2] https://github.com/Protelis/Protelis-Demo-Visualized

2.2 Sistemi distribuiti

Un sistema distribuito è una collezione di computer indipendenti che appare all'utente finale come un unico sistema coerente [24]. Il concetto di indipendenza implica che i dispositivi appartenenti al sistema non abbiano risorse condivise, in particolare la memoria; allo stesso tempo questi devono tenere un comportamento tra loro armonico. Deve essere quindi prevista una modalità in cui questi componenti possano comunicare, per scambiarsi informazioni e coordinarsi.

I costrutti fondanti della programmazione aggregata e i building block possono semplificare notevolmente il design e lo sviluppo di applicazioni in un contesto di Internet-of-Things [7]. Essa infatti consente di costruire, tramite composizione delle proprie API, delle applicazioni distribuite robuste e affidabili. Al fine di eseguire macchine virtuali Protelis in un contesto distribuito è necessario implementare un modello di scambio di messaggi tra queste (Figura 2.6b). L'architettura di Protelis (Sezione 1.3.2) è stata ideata avendo questo problema ben chiaro, pertanto è possibile estenderla facilmente, tramite l'implementazione di uno strato middleware.

2.3 Modellazione di un'architettura riusabile

L'obiettivo di questa sezione è di modellare una architettura del backend di Protelis flessibile, che consenta di incorporare la macchina virtuale all'interno di un dispositivo, così che attraverso questo sia possibile eseguirne le iterazioni. Concretamente si effettuerà un mapping delle caratteristiche di un dispositivo (livello in basso nella Figura 1.3), così che queste possano essere utilizzate per implementare i costrutti del *field calculus*. Un obiettivo importante di questo modello sarà circoscrivere in un'entità ben definita il concetto di strategia di comunicazione. In questo modo il device sarà in grado di funzionare indipendentemente dal metodo specifico scelto per realizzarla.

2.3.1 API di Protelis

Le API del backend di Protelis sono scritte in Java, ma sono facilmente integrabili anche con altri linguaggi eseguiti sulla Java Virtual Machine, ed espongono un insieme di astrazioni volte a modellare il dispositivo.

ExecutionContext

Interfaccia che si pone fra una macchina virtuale Protelis e l'ambiente da cui essa è circondata. I suoi compiti sono tre:

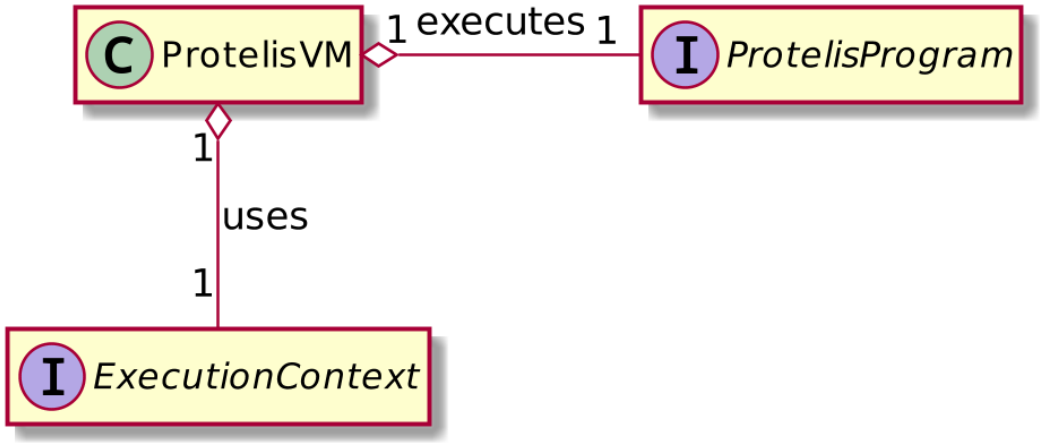


Figura 2.2: Relazione tra `ProtelisVM`, `ExecutionContext` e `ProtelisProgram`.

- tenere traccia dello stato persistente attraverso le iterazioni successive del prorgamma;
- tenere traccia dell'ultimo stato condiviso dai dispositivi vicini;
- tenere traccia dell'interazione del dispositivo con l'ambiente esterno (tempo, spazio, sensori, attuatori, ecc).

ProtelisVM

La macchina virtuale Protelis è il nucleo centrale dell'architettura: contiene l'interprete del linguaggio che al proprio interno implementa gli operatori del *field calculus*. Accetta in input un `ProtelisProgram` e utilizza un `ExecutionContext` per mantenere il proprio stato e interfacciarsi con l'esterno. L'interfaccia permette di azionarne i cicli di esecuzione.

AbstractExecutionContext

L'interfaccia `ExecutionContext` contiene la definizione di molti metodi che dovrebbero essere comuni a tutte le implementazioni di Protelis. Questa entità astratta realizza quelle procedure delegando a una nuova entità, il `NetworkManager`, il compito di stabilire il metodo di comunicazione.

NetworkManager

Astrazione di rete utilizzata dalla macchina virtuale Protelis: ad ogni iterazione, la macchina virtuale ha bisogno di accedere all'ultimo stato ricevuto

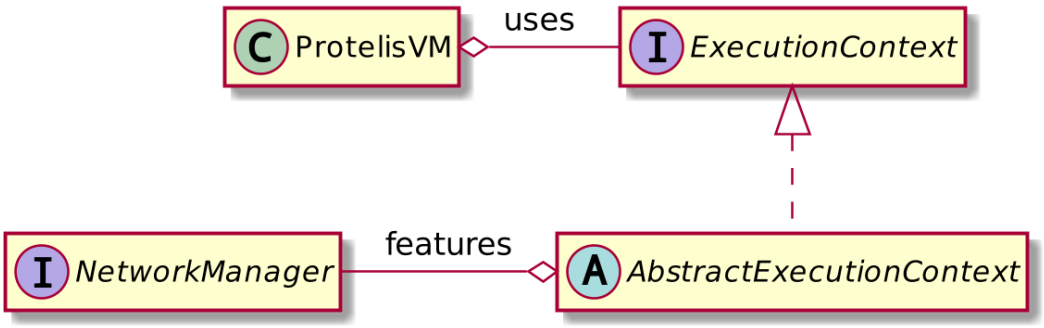


Figura 2.3: Introduzione di `AbstractExecutionContext` e `NetworkManager`.

dai vicini e di aggiornare lo stato che verrà inviato ad essi. L'implementazione di questa interfaccia è cruciale, infatti definisce le modalità di comunicazione tra i nodi della rete (Figura 2.2).

Una considerazione importante da fare è che la documentazione ufficiale specifica che non è necessario che lo stato venga ricevuto o inviato ad ogni iterazione. La scelta della frequenza di aggiornamento dello stato è demandata alla specifica implementazione di questa interfaccia, per trovare il rapporto giusto tra efficienza e condivisione dello stato.

CodePath

Rappresenta un percorso, che parte dalla radice dell'albero di esecuzione della macchina virtuale Protelis e termina in uno dei nodi. Supportando la serializzazione, viene utilizzato per confrontare l'esecuzione locale a un nodo con quella dei propri vicini e permettere l'allineamento del codice.

La specifica implementazione di questa classe è critica per la dimensione dei pacchetti generati dalla comunicazione tra i nodi. Sarà necessario, quindi, porre l'attenzione su questo aspetto quando la comunicazione sarà basata su protocolli di rete.

All'interno delle API di Protelis sono già presenti due implementazioni di questa interfaccia:

- `DefaultTimeEfficientCodePath`: il cui algoritmo di generazione è orientato alla velocità di esecuzione. L'output non è ottimizzato in termini di spazio. Questa implementazione non dovrebbe essere utilizzata quindi per l'utilizzo in una rete reale, perché potrebbe generare delle stringhe molto lunghe, con conseguenti problemi di performance;

- `HashingCodePath`: il cui output dipende dall'algoritmo di hashing utilizzato. L'algoritmo di generazione è orientato all'ottimizzazione dello

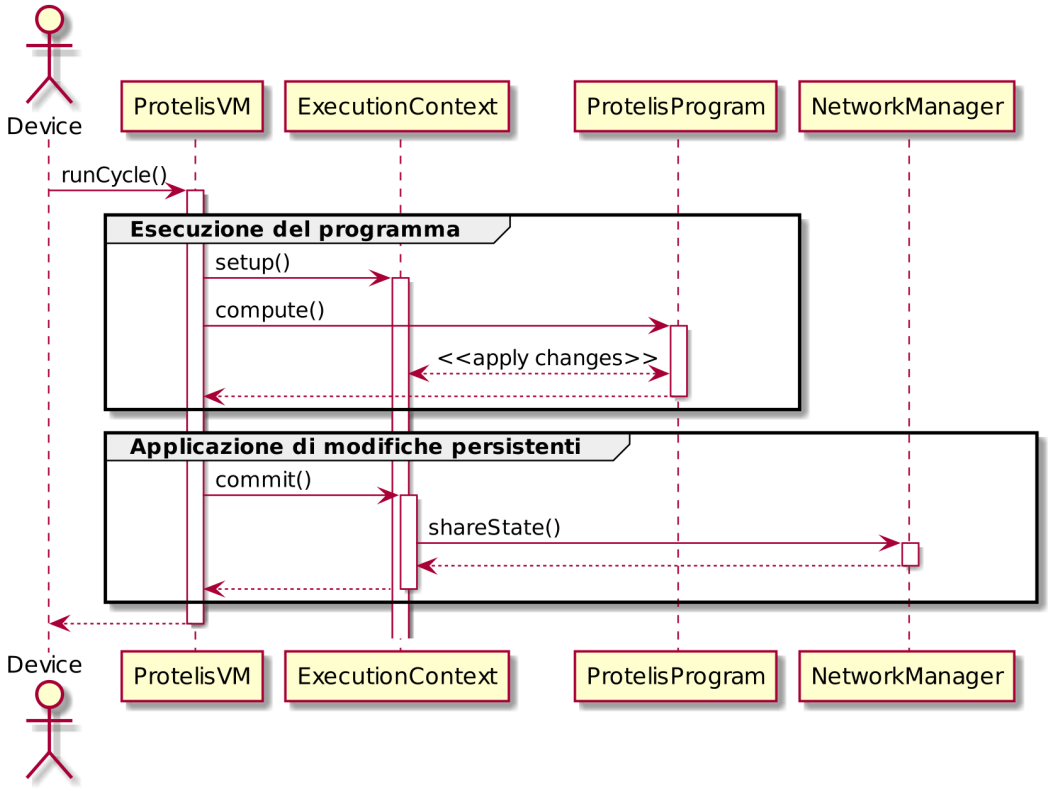


Figura 2.4: Rappresentazione attraverso un diagramma di sequenza UML di un ciclo di esecuzione della macchina virtuale Protelis. Come si vede l'utilizzo di diversi `NetworkManager` specializza il comportamento di tutto il sistema.

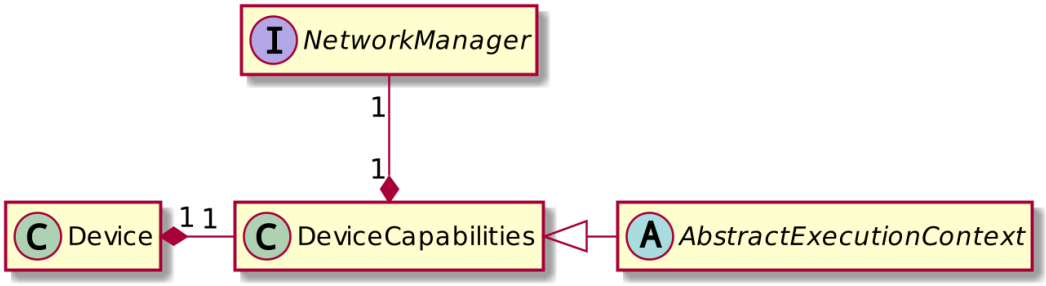


Figura 2.5: Introduzione di `DeviceCapabilities` e `Device`.

spazio, in quanto produce un output di lunghezza predicibile. Quindi è l'implementazione consigliata per l'utilizzo in una rete reale. Questa versione eredita dall'utilizzo delle funzioni di hashing il rischio di collisioni, che diminuisce con l'utilizzo di funzioni più complesse. Spetta al progettista la scelta della funzione di hashing che minimizzi le probabilità di collisione, massimizzando l'efficienza in termini computazionali.

2.3.2 Modellazione di un dispositivo e le sue capacità

Protelis non si cura di modellare il concetto di dispositivo. Le entità che offre non sono progettate per fornire un'architettura riusabile. Il contributo di questa tesi è di presentare un modello che consenta di definire esplicitamente le capacità di un dispositivo in una singola entità, utilizzata in seguito da un dispositivo. Questo modello offre i vantaggi della programmazione ad oggetti, come la semplicità di estendere una classe. Questo modello è adatto a rappresentare dispositivi sia in un ambiente reale che in uno simulato, e rappresenta dunque una possibile base di partenza riusabile per la realizzazione di un backend di Protelis.

Capacità di un dispositivo

L'obiettivo di `DeviceCapabilities` è di realizzare un `ExecutionContext`, traendo vantaggio delle funzioni già implementate in `AbstractExecutionContext`. Implementa le seguenti caratteristiche di un dispositivo:

- **Mantenere uno stato nel tempo**: ovvero essere in grado di mantenere un insieme di variabili locali e aggiornarle in caso di necessità.
- **Comunicare con altri dispositivi**: capacità e protocolli, regole utilizzate per effettuare uno scambio di informazioni con altri dispositivi

vicini. Per fare questo viene riutilizzato il concetto di `NetworkManager` esistente in Protelis.

- **Interagire con l'ambiente**: possibilità di interagire con il contesto da cui esso è circondato. In un contesto Internet-of-Things, per esempio, questo potrebbe concretizzarsi con l'utilizzo di sensori o attuatori.
- **Eseguire funzioni definite dall'utente**: facoltà di definire funzioni, comportamenti che possono essere richiamate all'occorrente durante l'esecuzione del programma.

Device

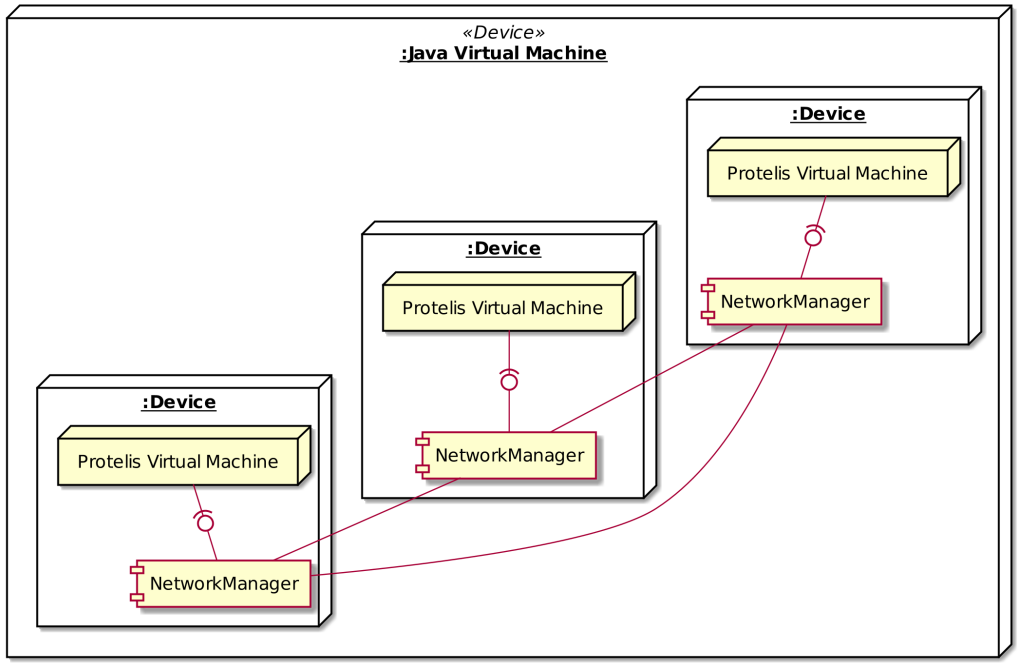
Un `Device` è un concetto che rappresenta un singolo dispositivo in una rete di nodi. Ciascuno di essi incorpora al proprio interno una macchina virtuale Protelis ed espone le proprie capacità (2.3.2), come la capacità di interagire con l'ambiente esterno o quella di mantenere lo stato di una variabile d'ambiente.

Un dispositivo per funzionare ha bisogno di un'implementazione concreta di `NetworkManager` (2.3.1), che viene specificata durante la creazione di uno specifico oggetto, seguendo il pattern strategy [16]. In questo modo il concetto di dispositivo non è vincolato a uno specifico metodo di comunicazione. Nel capitolo successivo è presente una dimostrazione della sua riusabilità.

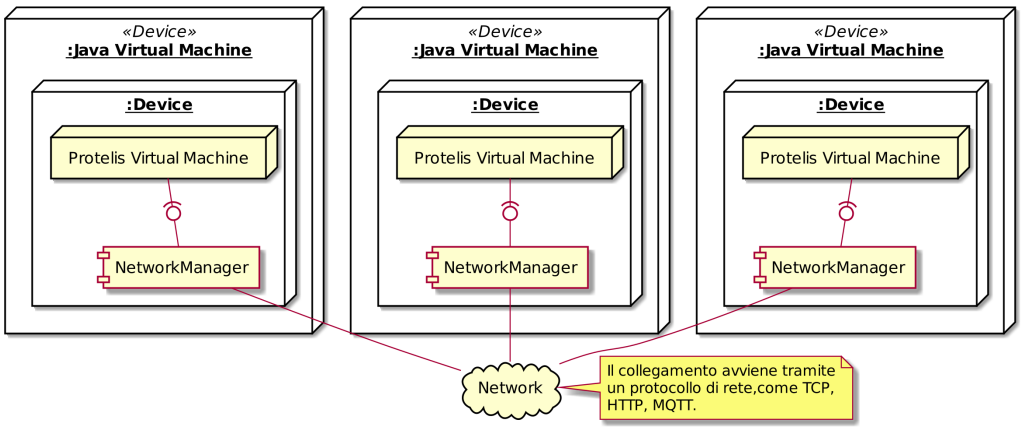
Speaker

Implementazione d'esempio di una *capability*, rappresenta la capacità di un `Device` di pubblicare un messaggio, in un modo non definito dall'interfaccia, un messaggio.

La sua implementazione più elementare è `ConsoleSpeaker`, che prevede che il messaggio venga stampato sullo standard output.



(a) Deployment in ambiente simulato



(b) Deployment in ambiente distribuito

Figura 2.6: Rappresentazione attraverso un diagramma UML di deployment delle due differenti modalità di comunicazione utilizzate dal `NetworkManager` in contesti distribuiti e simulati. Nella prima i dispositivi appartengono a una memoria condivisa, possono usare questa per comunicare. Nella seconda ciascun dispositivo appartiene a una diversa Java Virtual Machine, quindi devono utilizzare un altro metodo per comunicare, per esempio le socket.

Capitolo 3

Esempi applicativi

In questo capitolo viene provata la flessibilità del modello definito nella sezione 2.3, offrendo tre diversi esempi di realizzazione. In queste dimostrazioni sono presenti numerose assunzioni finalizzate a semplificare lo sviluppo. Una di queste è che la simulazione inizia quando la rete ha già effettuato il meccanismo di discovery. Questo consente di fornire a ciascun dispositivo l'elenco dei propri vicini e concentrare l'attenzione sul `NetworkManager`, vero protagonista di questa fase.

Le demo sono state sviluppate sia in Java che in Kotlin, utilizzando per entrambi un framework di testing, relativamente *Junit* e *Kotlintest*. In seguito i test sono stati agganciati ad una pipeline di continuous integration basata su TravisCI[1].

Sono stati realizzati un totale di tre esempi applicativi, usando rispettivamente:

- un micro-simulatore;
- un esempio con comunicazione via socket TCP;
- un esempio con comunicazione via protocollo MQTT.

Tutti e tre gli esempi eseguono il medesimo programma aggregato, cambiando solo la piattaforma di esecuzione. Per ciascun esempio, si fornisce una implementazione in Java e una in Kotlin. In questo modo si può notare come l'implementazione del backend sia completamente trasparente al comportamento finale del sistema. Le simulazioni sono state eseguite su una rete di cinque dispositivi, disposti secondo topologia ad anello: ciascun dispositivo è quindi in grado di comunicare solo con quello a sé adiacente.

[1] https://travis-ci.org/

```
// Declare the name of this Protelis module
// Just like declaring a Java package, except it ends with this
    file's name (sans extension)
module hello

import protelis:state:time

// Get a variable from the environment
let leader = env.has("leader");

if(leader) {
   self.announce("The leader is at "+self.getDeviceUID().getUID());
   self.announce("The leader's count is: "+countDownWithDecay(4,1));
} else {
   // Otherwise, stay silent
   false;
};

// Check if any neighbor is the leader
if (anyHood(nbr(leader))) {
   // if so, speak!
   self.announce("Hello from the leader to its neighbor at
       "+self.getDeviceUID().getUID());
} else {
   // Otherwise, stay silent
   //self.announce("No one is leader here");
   false;
};
```

Listato 3.1: Hello, World! in Protelis.

Il programma Protelis eseguito è un semplice HelloWorld (Listato 3.1). Nella prima parte il leader effettua un conteggio a ritroso. Nella seconda parte i dispositivi adiacenti al leader vengono salutati da questo.

3.1 Micro-simulatore

La prima implementazione che è stata realizzata è un simulatore. L'implementazione di Protelis già esistente in Alchemist, quella che utilizza la libreria NASA World Wind, e quella già esistente nel vecchio repository di Protelis sono state la linea guida per lo sviluppo di questa versione, in quanto le strategie di comunicazione sono pressoché le stesse.

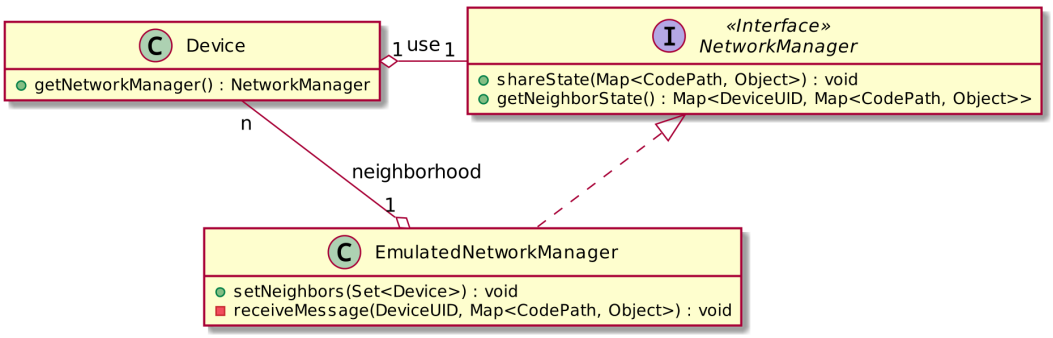


Figura 3.1: Rappresentazione tramite diagramma UML delle classi della classe `EmulatedNetworkManager`. Essa contiente un insieme di `Device` vicini al `Device` a cui appartiene. Utilizza questi riferimenti per inviare loro messaggi.

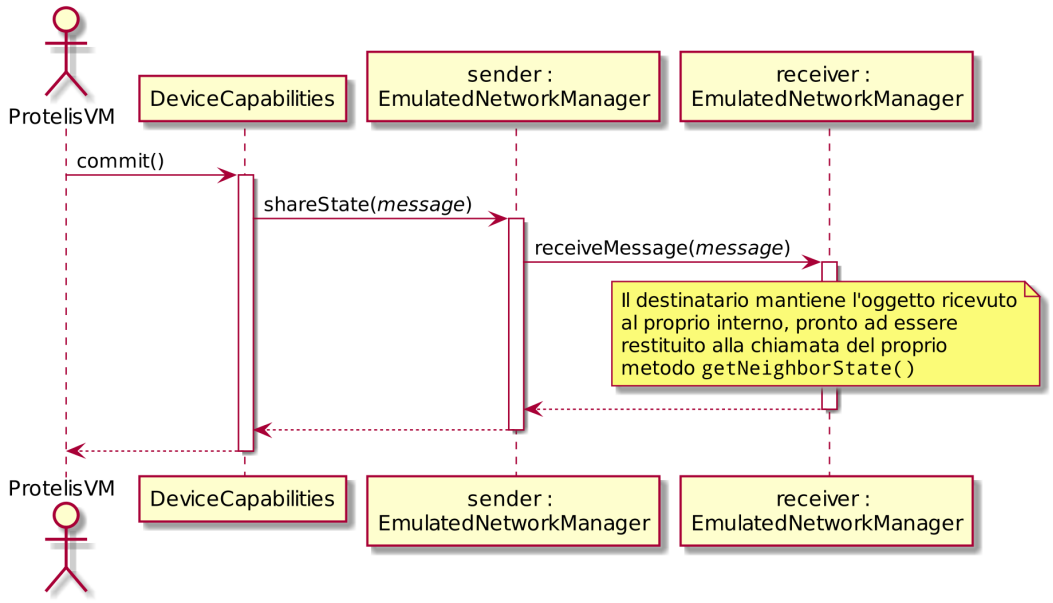


Figura 3.2: Rappresentazione attraverso un diagramma UML di sequenza l'uso dell'entità `EmulatedNetworkManager`.

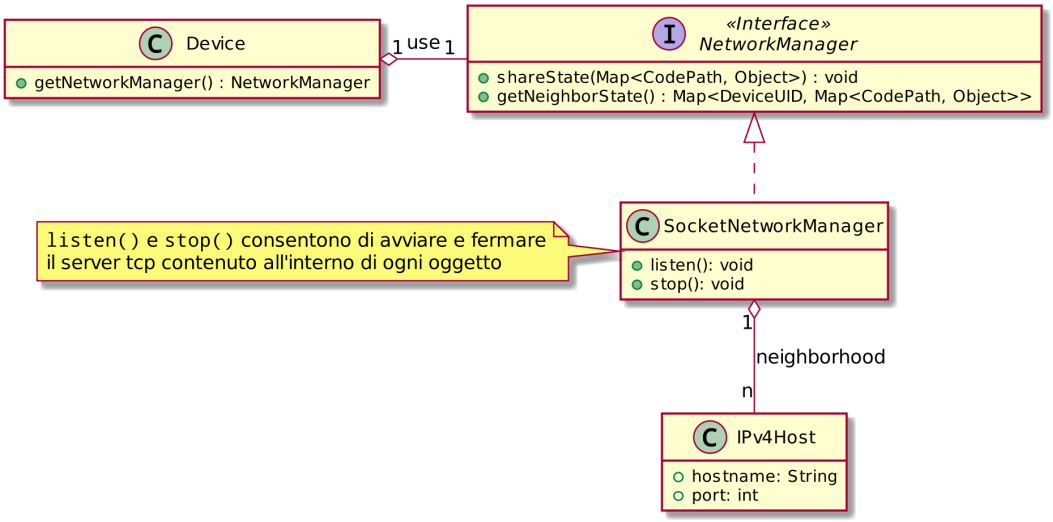


Figura 3.3: Rappresentazione tramite diagramma UML delle classi dell'entità `SocketNetworkManager`.

In questa modalità i `Device` sono semplici oggetti eseguiti all'interno della stessa Java Virtual Machine (Figura 2.6a). Ne segue che ciascuno ha visibilità dell'altro in memoria e la comunicazione avviene, come di consueto nella programmazione ad oggetti, tramite le chiamate ai metodi; il contenuto dei messaggi è passato come argomento di questi ultimi.

Nello specifico è stata realizzata la classe `EmulatedNetworkManager`, che simula il comportamento che il `NetworkManager` avrebbe in una rete reale. Ciascun `EmulatedNetworkManager` contiene una lista di `Device` vicini(Figura 3.5). Il meccanismo utilizzato per l'allineamento del codice in fase di esecuzione prevede lo scambio di un oggetto di tipo `Map<CodePath, Object>`. Questo viene passato attraverso un apposito metodo per ricevere un messaggio dall'esterno, utilizzato nel momento in cui ad un dispositivo viene richiesto di scambiare lo stato con altri dispositivi (Figura 3.6). L'implementazione del `CodePath` scelta per questa versione è `DefaultTimeEfficientCodePath`, in quanto la dimensione di questo oggetto non è rilevante, poiché questo viene semplicemente referenziato in memoria, senza necessità di effettuarne copie.

3.2 Comunicazione via socket TCP

Questa versione realizza una versione distribuita del backend di Protelis. È stata fatta però una semplificazione rispetto a quanto descritto dal diagramma presente nella Figura 2.6b): i `Device` non sono eseguiti in macchine virtuali Java distinte. Per simulare il fatto che lo siano essi non possono accedere a

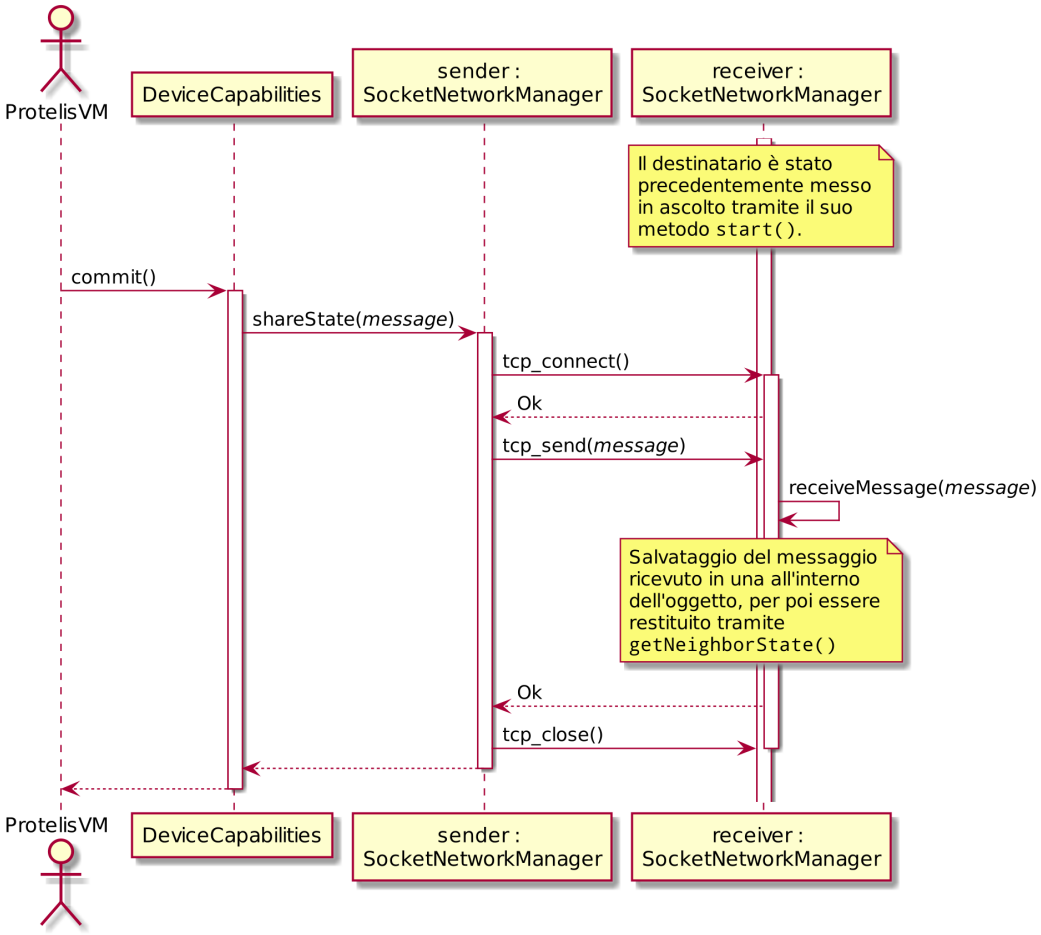


Figura 3.4: Rappresentazione tramite diagramma UML di sequenza del comportamento della classe `SocketNetworkManager`.

zone di memoria condivise. Per comunicare è necessario che essi utilizzino metodi alternativi, per esempio che utilizzino la rete IP come tramite.

In questo esempio il passaggio dell'oggetto `Map<CodePath, Object>` avviene tramite l'utilizzo di socket TCP. Il protocollo TCP [21] è un protocollo di rete di livello trasporto, connesso, che utilizza un'architettura client-server e che garantisce un canale affidabile di comunicazione tra due applicazioni su host distinti. La comunicazione avviene nel seguente modo: un server si mette in ascolto su una determinata porta; un client invia al server una richiesta di connessione su quella porta. A questo punto il server accetta la connessione e viene stabilito un canale di comunicazione affidabile, che può essere utilizzato per lo scambio in entrambe le direzioni di flussi di dati. La connessione può essere terminata in qualsiasi momento da uno degli host.

La classe che utilizza le socket è denominata `SocketNetworkManager`. In questa versione ciascun `SocketNetworkManager` contiene al proprio interno una lista di tuple nella forma *(hostname, porta)*, che rappresentano i vicini del dispositivo che lo contiene. Ciascuno di essi svolge un duplice compito di server e client: il primo consiste nel rimanere in ascolto di connessioni entranti per ricevere messaggi; il secondo comporta stabilire connessioni con altri `SocketNetworkManager` quando deve inviare un messaggio. Una volta che la connessione è stabilita il mittente invia l'oggetto `Map<CodePath, Object>` al destinatario, che poi procede a memorizzarlo.

Per permettere il passaggio di un oggetto tramite un protocollo di rete è necessario effettuarne una serializzazione, ovvero una conversione del suo stato in una serie di byte, cosicché questa possa essere memorizzata o, in questo caso, trasferita tramite la rete sotto forma di stream di dati. Una volta giunta a destinazione questa viene de-serializzata, in modo da poter ricostruire un oggetto replica dell'originale. I principali metodi utilizzabili per la serializzazione di oggetti sono: Protocol Buffers[2], JSON [12], YAML [11], Kryo[3], Elsa[4], etc.. In questa demo, per motivi di semplificazione, è stata utilizzato il meccanismo di serializzazione già presente in Java: l'interfaccia `Serializable`, che, in quanto non sono presenti particolari richieste in termini di spazio o tempo, soddisfa pienamente le necessità del progetto.

L'implementazione di `CodePath` utilizzata in questa demo è `HashingCodePath`, in quanto l'oggetto necessita di essere serializzato e trasferito successivamente tramite la rete. È conveniente dunque che la dimensione finale di questo sia predicibile a priori.

[2]https://developers.google.com/protocol-buffers
[3]https://github.com/EsotericSoftware/kryo
[4]https://github.com/jankotek/elsa

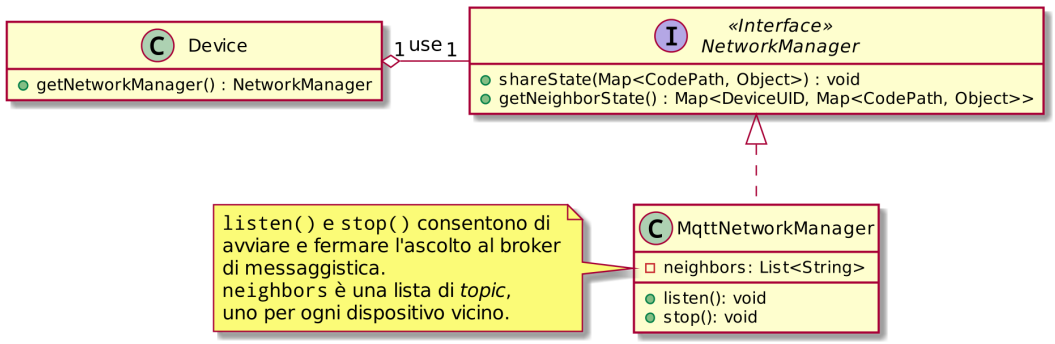


Figura 3.5: Rappresentazione tramite diagramma UML delle classi dell'entità `MqttNetworkManager`.

3.3 Protocolli orientati all'Internet-of-Things

Come già ampiamente discusso, le caratteristiche della programmazione aggregata la rendono adatta a scenari tipici di ambiti come IoT e reti di sensori. È importante che l'architettura definita possa supportare le tecnologie e i protocolli utilizzati in questi ambiti, come MQTT [23], Stomp[5], CoAP [22], eccetera. In questa sezione viene mostrata un'implementazione `NetworkManager` che sfrutta il protocollo MQTT, per permettere al proprio dispositivo di comunicare con gli altri nodi della rete.

Il protocollo MQTT è un protocollo nato per l'utilizzo nell'ambito Internet-of-Things e comunicazione *machine-to-machine*. Utilizza il modello *publish/-subscribe*. Questo prevede la presenza di un broker di messaggistica, un nodo centrale attraverso il quale i client possono registrarsi per essere notificati di determinati *topic* (argomenti). In qualsiasi momento un nodo può registrarsi a un nuovo argomento di interesse o inviare un messaggio a uno di questi. Nel momento in cui il broker riceve un nuovo messaggio relativo ad un argomento, notifica tutti i client che si erano registrati ad esso in precedenza. Questo approccio consente di disaccoppiare la produzione del messaggio dalla sua accettazione, rendendo la comunicazione asincrona.

La classe che utilizza questo protocollo si chiama `MqttNetworkManager`. Come la classe `SocketNetworkManager` descritta in precedenza, anche questa ha una duplice funzione. La prima è quella di essere a disposizione per ricevere messaggi: questo avviene mediante la registrazione, presso un broker centrale, relativa ai messaggi destinati a sé stesso. La seconda è quella di inviare messaggi agli altri nodi. Per fare questo ciascun `MqttNetworkManager` contiene al proprio interno una lista di argomenti a cui inviare messaggi, uno per ogni vi-

[5] https://stomp.github.io/

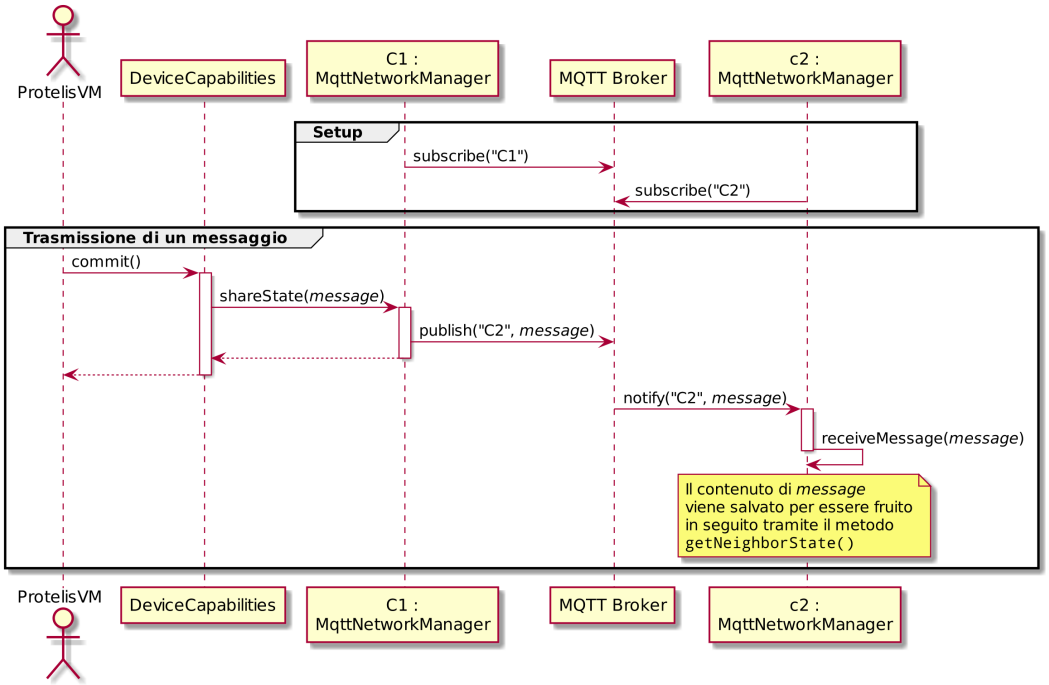


Figura 3.6: Rappresentazione attraverso un diagramma UML di sequenza dell'uso della classe `MqttNetworkManager`. La fase di trasmissione di un messaggio è ripetuta per ogni dispositivo vicino.

.

cino; nel momento in cui gli viene richiesto di inviare l'oggetto `Map<CodePath, Object>` agli altri nodi esso lo invia, una volta per ogni argomento, dopo opportuna serializzazione, al broker di messaggistica, che provvederà a diffondere il messaggio ai rispettivi destinatari.

Anche in questa versione sono state fatte molte delle scelte effettuate per quella precedente: i `Device` sono eseguiti all'interno della stessa Java Virtual Machine, ma si comportano come se non lo fossero; il serializzatore scelto è quello incluso nelle librerie di Java; l'implementazione di `CodePath` utilizzata è `HashingCodePath` a causa della necessità di trasferire questo oggetto tramite la rete.

Capitolo 4

Conclusioni e lavori futuri

4.1 Conclusioni

La programmazione aggregata è un approccio promettente alla risoluzione di numerosi problemi relativi ai sistemi distribuiti come IoT, reti di sensori, eccetera.

Partendo dalle API esistenti di Protelis, è stato definito un modello riusabile che consente di eseguire una macchina virtuale Protelis su diverse infrastrutture, che possono essere reti reali, come nell'Internet-of-Things oppure simulate (come nel caso di Alchemist). Per fare ciò si è dovuto introdurre uno strato middleware che si occupasse della comunicazione tra i nodi in maniera completamente trasparente ad essi, ovvero in modo che se l'implementazione di questo cambiasse, il comportamento finale del sistema rimarrebbe lo stesso. Quindi, per isolare il concetto di comunicazione tra dispositivi dalle entità esistenti, si è fatto uso del concetto già definito dalle API di Protelis di `NetworkManager`, che fornisce una maniera molto semplice di realizzare lo scambio di messaggi.

A verifica della flessibilità del modello proposto, sono stati portati tre diversi esempi che rappresentano tre diverse realtà in cui il linguaggio Protelis può essere utilizzato. In particolare, il primo mira a creare una simulazione di una rete reale i cui nodi eseguono una macchina virtuale Protelis; il secondo mostra come l'architettura di Protelis possa essere facilmente estesa ad un sistema distribuito, supportando la comunicazione attraverso la rete IP; infine il terzo implementa l'uso di uno dei protocolli più utilizzati in ambito IoT.

Gli elaborati prodotti sono stati integrati come esempi al repository ufficiale di Protelis[1] e rappresentano un possibile punto di partenza per chiunque voglia implementare un nuovo backend del linguaggio.

[1] https://github.com/Protelis/Protelis-Demo

4.2 Lavori futuri

Gli elaborati proposti in questa tesi sono facilmente estendibili. Di seguito vengono elencati alcuni suggerimenti, per evolvere le demo esistenti o per svilupparne di nuove. Migliorare la demo è molto facile: è sufficiente fare un *fork*[2] del repository ufficiale della demo di Protelis, effettuare delle modifiche e proporle al repository originale tramite una *pull-request*[3].

4.2.1 Introduzione di nuove capacità

Il programma eseguito dai nodi in questi esempi è un semplice Hello, World! seguito da un conteggio a ritroso. Questo perché la sua finalità non è mostrare le capacità di Protelis di sfruttare le capacità di un dispositivo reale, bensì di mostrare come esso si possa adattare a tecniche di comunicazione diverse. Al fine di ampliare le dimostrazioni esistenti si potrebbe integrare alle capacità di un dispositivo: la facoltà di interagire con lo spazio che lo circonda, la possibilità di interagire con il tempo. In questo modo ciascun dispositivo acquisirebbe la consapevolezza della propria posizione nello spazio-tempo, garantendo la possibilità di fruire di nuove tipologie di algoritmi, come il gradiente. Una possibile e affascinante applicazione di questa nuova tipologia di dispositivi è il controllo di uno sciame di droni. Infatti, garantendo ciascuno il controllo della propria posizione nel tempo, ma descrivendone il comportamento in maniera collettiva, è relativamente semplice prevenire qualsiasi tipo di collisione, fornendo la possibilità di creare comportamenti o coreografie in maniera efficace [3]. Protelis fornisce supporto per l'impiego delle funzioni relative allo spazio e al tempo tramite le interfacce `SpatiallyEmbeddedDevice` e `TimeAwareDevice`.

4.2.2 Refactoring della build

L'elaborato prodotto utilizza Gradle[4], uno strumento di build automation, per la gestione di sei sottoprogetti, che consente di controllare in maniera centralizzata l'esecuzione di alcune attività relative ad essi, come l'esecuzione dei test o lanciare controlli della qualità del codice.

Nella configurazione attuale, in entrambi i linguaggi utilizzati, l'architettura riusabile modellata nel corso della tesi è definita solo nel sottoprogetto che

[2]https://help.github.com/en/github/getting-started-with-github/fork-a-repo

[3]https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests

[4]https://gradle.org/

descrive il primo scenario di esempio. Per poterne fare uso, gli altri due sottoprogetti, tramite una funzionalità di Gradle, importano direttamente i file sorgenti dal primo. Questa configurazione può essere migliorata introducendo un nuovo sottoprogetto, per ciascun linguaggio, nel quale possano essere isolate le parti comuni del modello, in modo che queste possano essere importate da tutti gli altri sottoprogetti.

4.2.3 Produzione di ulteriori template

Si è visto che l'architettura prodotta, per mezzo dell'entità `NetworkManager`, si presta all'utilizzo di svariate tecniche e protocolli di comunicazione. Un possibile miglioramento può essere l'implementazione di nuovi `NetworkManager`, che possano utilizzare tecniche diverse da quelle già utilizzate. In questo modo il repository ufficiale verrebbe arricchito di nuovi esempi, che possono essere un punto di riferimento per chi si approccia al linguaggio. Alcune possibili tecniche di comunicazione che possono essere utilizzate sono:

- **STOMP** (Simple Text-orientated Messaging Protocol): protocollo che mira ad offrire un canale di comunicazione basato su messaggi di testo, molto leggibile per l'uomo, meno efficiente in termini di banda.
- **COaP** (Constrained Application Protocol) [22]: protocollo sviluppato per l'interazione tra macchine. Si basa su un model REST, esattamente come HTTP, per offrire i propri servizi. È molto competitivo in ambito Internet-of-Things a causa della sua capacità di essere eseguito anche in dispositivi con pochissime risorse.
- **HTTP** (HyperText Transfer Protocol) [10]: protocollo destinato allo scambio di informazioni, come ipertesto, risorse in file di testo o streaming video.

4.2.4 Protelis su dispositivi mobili

Protelis si basa sull'infrastruttura esistente di Java, infatti viene eseguito all'interno di una Java Virtual Machine. Questa sua caratteristica può essere sfruttata per l'implementazione di una versione di Protelis in grado di essere eseguita all'interno di un dispositivo Android. Infatti le applicazioni per questi dispositivi sono scritte nativamente in Java e Kotlin, due linguaggi con cui le API di Protelis si possono interfacciare molto semplicemente. Inoltre i sensori presenti in un dispositivo mobile, come uno smartphone, allargano notevolmente il panorama delle possibili applicazioni.

Bibliografia

[1] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. *Space-Time Universality of Field Calculus*, pages 1–20. 01 2018.

[2] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *ACM Trans. Comput. Logic*, 20(1):5:1–5:55, January 2019.

[3] J. Beal, K. Usbeck, J. Loyall, M. Rowe, and J. Metzler. Adaptive task reallocation for airborne sensor sharing. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 168–173, Sep. 2016.

[4] J. Beal and M. Viroli. Building blocks for aggregate programming of self-organising applications. In *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 8–13, Sep. 2014.

[5] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *Intelligent Systems, IEEE*, 21:10 – 19, 04 2006.

[6] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, 02 2012.

[7] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9):22–30, 2015.

[8] Jacob Beal, Kyle Usbeck, Joseph Loyall, Mason Rowe, and James Metzler. Adaptive opportunistic airborne sensor sharing. *ACM Trans. Auton. Adapt. Syst.*, 13(1):6:1–6:29, April 2018.

[9] David G. Bell, Frank Kuehnel, Chris Maxwell, Randy Kim, Kushyar Kasraie, Tom Gaskins, Patrick Hogan, and Joe Coughlan. NASA world

wind: Opensource GIS for mission operations. In *2007 IEEE Aerospace Conference*. IEEE, 2007.

[10] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015.

[11] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. Yaml ain't markup language, version 1.2. *Available on: http://yaml. org/spec/1.2/spec. html*, 2009.

[12] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, March 2014.

[13] S. S. Clark, J. Beal, and P. Pal. Distributed recovery for enterprise services. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 111–120, Sep. 2015.

[14] Shane Clark, Jacob Beal, and Partha Pal. Distributed recovery for enterprise services. pages 111–120, 09 2015.

[15] Matteo Francia. *A Foundational Library for Aggregate Programming*. PhD thesis.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] A. Paulos, S. Dasgupta, J. Beal, Y. Mo, K. Hoang, L. J. Bryan, P. Pal, R. Schantz, J. Schewe, R. Sitaraman, A. Wald, C. Wayllace, and W. Yeoh. A framework for self-adaptive dispersal of computing services. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 98–103, June 2019.

[18] D. Pianini, A. Croatti, A. Ricci, and M. Viroli. Computational fields meet augmented reality: Perspectives and challenges. In *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 80–85, Sep. 2015.

[19] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7, 01 2013.

[20] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: Practical aggregate programming. *Proceedings of the ACM Symposium on Applied Computing*, pages 1846–1853, 01 2015.

[21] J. Postel. Transmission control protocol. Technical report, September 1981.

[22] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014.

[23] Shubhangi A Shinde, Pooja A Nimkar, Shubhangi P Singh, Vrushali D Salpe, and Yogesh R Jadhav. Mqtt-message queuing telemetry transport protocol. *International Journal of Research*, 3(3):240–244, 2016.

[24] Andrew Tanenbaum and Maarten van Steen. *Chapter 1 of Distributed Systems - Principles and Paradigms*. 03 2016.

[25] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg, 2013.