Alma Mater Studiorum — Università di Bologna
Campus di Cesena

Dipartimento di Informatica — Scienza e Ingegneria
Corso di Laurea magistrale in Ingegneria e Scienze Informatiche

Progettazione di una piattaforma web per la simulazione di programmi aggregati

Tesi in Pervasive Computing

Presentata da:
Niccolò Maltoni
Matricola 840825

Relatore:
Prof. Mirko Viroli

Correlatore:
Prof. Danilo Pianini

III sessione di laurea
Anno Accademico 2019–2020

Parole chiave

Aggregate computing

Aggregate programming

Protelis

Applicazione web

Simulazione

A tutti gli amici e i familiari che mi hanno sostenuto.
A chi non ha potuto assistere anche a questo mio traguardo.

Sommario

La *programmazione aggregata* è un approccio innovativo nato in tempi recenti per far fronte alla necessità di un punto di vista nuovo nella programmazione di sistemi distribuiti. In particolare, basandosi sull'impianto teorico del *field calculus*, negli ultimi anni sono stati realizzati, da parte dell'Università di Bologna, linguaggi e framework innovativi per la sua applicazione in contesti d'uso reale: *Protelis* e *ScaFi*.

La principale criticità che mina la diffusione di questo tipo di linguaggi è legata alla configurazione del sistema per l'esecuzione: è infatti necessario avere a disposizione una rete, reale o fisica, di dispositivi per l'esecuzione del codice e, soprattutto in contesto didattico, la necessità di dispiegare un certo numero di dispositivi o configurare un simulatore può costituire un ulteriore gradino di complessità.

Lo scopo di questa tesi è progettare una piattaforma web che permetta di realizzare semplici programmi aggregati senza configurazione alcuna. È stato realizzato un sistema composto da un server esecutore, che si avvale del simulatore Alchemist per eseguire il codice Protelis, e da un'applicazione web in React che permetta la scrittura del codice e il *monitoring* dell'esecuzione.

Indice

Introduzione **1**

I. Background 3

1. Programmazione aggregata **4**

1.1. Field calculus e Building Blocks 6
1.2. Protelis 9
1.3. ScaFi 10

2. Progettazione di sistemi web **12**

2.1. Architetture & paradigmi 12
2.2. Linguaggi ad uso web 13
2.2.1. JavaScript e ECMAScript 13
2.2.2. TypeScript 14
2.2.3. Scala.js 15
2.2.4. Kotlin/Multiplatform e Kotlin/JS 16

3. Motivazioni e Stato dell'arte **18**

3.1. Accessibilità allo sviluppo in Protelis 18
3.2. Ambienti di sviluppo online 20

II. Contributo: WebProtelis 21

4. Requisiti e Analisi **22**

4.1. Requisiti della piattaforma 22
4.1.1. Requisiti funzionali 22
4.1.2. Requisiti non funzionali 23
4.2. Analisi dei requisiti e vincoli di fattibilità 23
4.2.1. Requisiti del client 24
4.2.2. Requisiti del server 24
4.3. Architettura logica 25

5. Progettazione **26**
5.1. Design dell'applicazione . . . 26
5.1.1. Mockup dell'interfaccia . . . 26
5.1.2. Design di riferimento . . . 27
5.2. Architettura del client . . . 28
5.2.1. Framework di sviluppo . . . 28
5.2.2. Pattern di gestione dello stato . . . 29
5.3. Architettura del server . . . 32
5.3.1. Pattern reactor . . . 32
5.3.2. Progettazione dei verticle . . . 33
5.3.3. Simulatore scelto: Alchemist . . . 33
5.4. Interazioni . . . 36
5.4.1. Scelta del modello di comunicazione e del protocollo . . . 36
5.4.2. Comportamento . . . 37

6. Implementazione **40**
6.1. Tecnologie utilizzate . . . 40
6.1.1. Linguaggi di programmazione . . . 40
6.1.2. Strumenti per lo sviluppo e il controllo del software . . . 42
6.1.3. Controllo di versione e CI/CD . . . 44
6.1.4. Deployment . . . 45
6.2. Dettagli implementativi: Frontend . . . 45
6.3. Dettagli implementativi: Backend . . . 47

III. Conclusioni **49**

7. Valutazione dei risultati **50**
7.1. Considerazioni sul contributo . . . 50
7.2. Valutazione dell'interfaccia . . . 51
7.2.1. Performance . . . 51
7.2.2. Accessibilità e Best Practices . . . 52
7.2.3. SEO . . . 52

8. Considerazioni finali e lavori futuri **53**

Appendice **55**

A. Dockerfile del server **56**

B. YAML di configurazione per Alchemist **57**

Ringraziamenti **58**

Bibliografia **59**

Introduzione

Nel corso degli ultimi anni i sistemi informatici hanno avuto uno sviluppo notevole. Grazie all'incremento della potenza computazionale disponibile a basso costo, alla riduzione delle dimensioni delle unità di calcolo e alla diffusione delle reti wireless, ormai molti degli ambienti in cui vive la maggior parte della popolazione sono pervasi di sensori e di dispositivi "smart", tanto che si parla sempre più spesso di *IoT* [21], ovvero Internet delle Cose.

Questo ha portato alla necessità di programmare sistemi distribuiti composti da numerosi dispositivi che devono potersi coordinare tra loro per poter portare a termine la computazione.

La *programmazione aggregata* è un approccio promettente per lo sviluppo di sistemi di questo tipo. Tale paradigma è basato sull'impianto teorico del *field calculus* e ha visto negli ultimi anni la realizzazione, da parte dell'Università di Bologna, di linguaggi e framework innovativi per la sua applicazione in contesti d'uso reale: *Protelis* e *ScaFi*.

La principale criticità legata a questo tipo di linguaggi, soprattutto in contesto didattico, è correlata alla configurazione del sistema per l'esecuzione. Infatti, affinché la programmazione aggregata risulti significativa, è richiesta la presenza di diversi dispositivi sui quali il codice possa essere eseguito. Poiché avere a disposizione una rete di dispositivi è costoso, molto spesso si decide di appoggiarsi a simulatori compatibili, la cui configurazione in molti casi non è banale e può coinvolgere anche la realizzazione di un progetto strutturato e completo per poter realizzare anche solo un prototipo minimale, aggiungendo un ulteriore gradino di complessità rispetto ai linguaggi tradizionali.

In tempi recenti, anche il web ha visto una rapida evoluzione a livello tecnologico, la quale ha permesso di realizzare applicazioni utilizzabili tramite browser con livelli di complessità comparabili alle controparti desktop, senza il carico aggiuntivo, dal punto di vista dell'utente, dell'installazione e della configurazione. Inoltre, servizi complessi possono non dipendere esclusivamente dalle risorse computazionali dei dispositivi dell'utente, bensì sfruttarle solo quando necessario, appoggiandosi alla potenza computazionale di un server di backend per le operazioni più onerose.

Si è dunque ritenuto utile realizzare un sistema web semplice e immediato da usare che permetta di abbozzare esempi di codice aggregato (Protelis, nel prototipo implementato per questa tesi) e poterlo eseguire senza la necessità di configurare una rete di dispositivi o un simulatore. In particolare, si è deciso di propendere per l'implementazione di un backend reattivo su piattaforma JVM che esegue il codice inviato dal client su una rete di dispositivi simulata tramite il simulatore Alchemist [31]. Il client è una *Single-Page Application* statica che

permette lo sviluppo di codice direttamente dal browser e comunica con il server tramite un bus di eventi.

Il documento è suddiviso in tre Parti principali.

Nella Parte I viene fatta una disamina del contesto nel quale l'elaborato di tesi va a inserirsi. In particolare, nel Capitolo 1 del documento viene fatta un'introduzione alla programmazione aggregata, con particolare attenzione ai linguaggi ad essa collegati. Nel Capitolo 2 viene fatta riportato il risultato della fase di studio dello stato dell'arte in merito allo sviluppo di sistemi web, con particolare attenzione ai linguaggi che permettono la realizzazione di applicazioni frontend in esecuzione sui browser degli utenti. Nel Capitolo 3 vengono infine esaminate più nel dettaglio le ragioni per le quali il progetto è stato realizzato, con particolare attenzione allo stato dell'arte e a possibili soluzioni a problematiche simili.

Nella Parte II viene analizzato il processo di progettazione del sistema a cui questo elaborato di tesi si riferisce e di sviluppo del prototipo. Ciascuna fase del processo è descritta in un Capitolo dedicato.

Infine, nella Parte III vengono raccolti i risultati del processo di sviluppo (Capitolo 7), presentate le prospettive future del sistema prodotto ed enunciate brevemente le considerazioni finali (Capitolo 8) a conclusione di questa tesi.

Parte I.

Background

1. Programmazione aggregata

La crescita esponenziale di dispositivi informatici di varia natura inseriti in contesti quotidiani ha avuto un impatto globale notevole. Questo insieme di entità connesse (Figura 1.1) ha dato luogo a ciò che viene definito *Internet of Things* (*IoT*) [21]: sistemi costituiti da reti di oggetti fisici, tipicamente embedded, che interagiscono mediante la rete Internet.

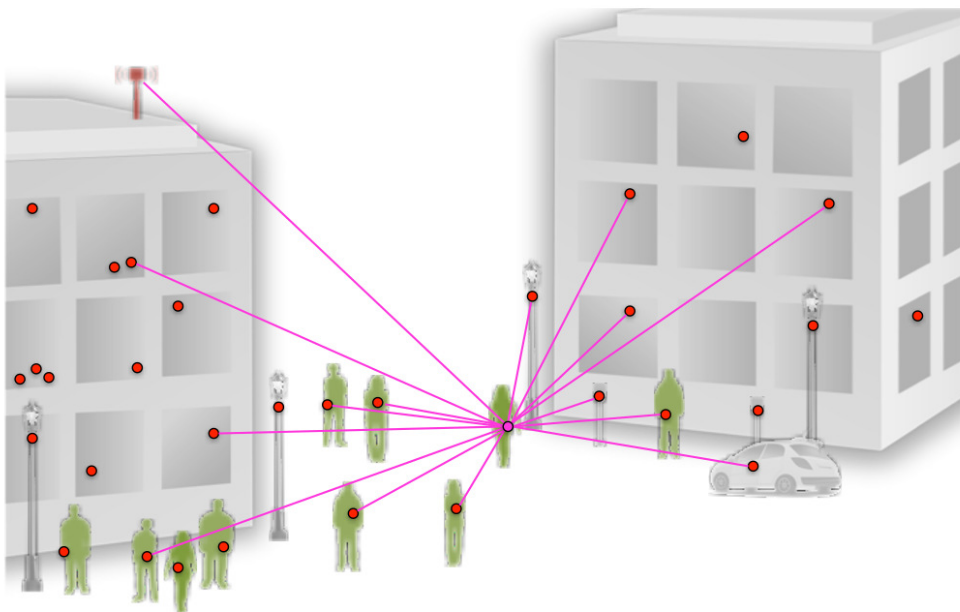
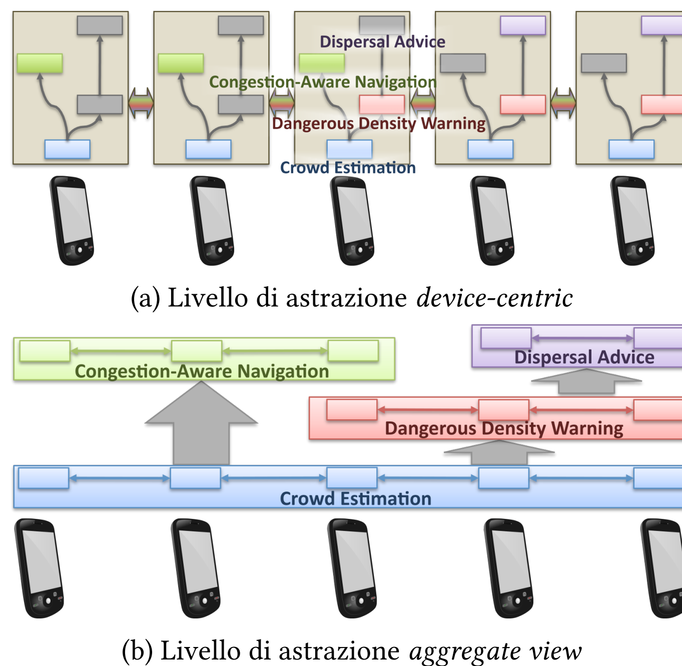


Figura 1.1.: Possibile scenario di rete in un contesto urbano.
Figura ripresa da [34].

L'approccio tradizionale per la realizzazione di sistemi in questo contesto è sempre stato dal punto di vista del singolo dispositivo, il quale è preso come unità fondamentale, connessa con il mondo fisico e con gli altri device. In questo punto di vista, l'insieme di tutti i comportamenti individuali delle unità determina il funzionamento del sistema. Tale approccio, per quanto valido, può risultare limitante in sistemi distribuiti eterogenei, nei quali possono presentarsi diversi problemi legati all'organizzazione della rete e alla sua gestione a causa delle dimensioni e delle differenze tra i dispositivi. Tali problematiche, generalmente, vorrebbero essere gestite ad un livello di astrazione più elevato.

L'*aggregate programming*, o *programmazione aggregata*, costituisce un'alternativa all'approccio "classico" volta a semplificare la progettazione, creazione e manutenzione di sistemi distribuiti complessi. La programmazione aggregata, infatti, ragiona su larga scala, cercando di spostare l'attenzione sul *collettivo di dispositivi* che collaborano, astraendo dai dettagli relativi

al singolo device [34] per quanto possibile. L'idea alla base è dunque di definire una modalità di deduzione del comportamento locale al singolo dispositivo a partire dal comportamento globale, di più alto livello, effettuando un *mapping da globale a locale*. In contesti distribuiti, sono stati individuati principalmente due punti di vista a questo livello di astrazione [34, 41]: locale o globale. Il primo, detto anche *device-centric* (Figura 1.2a), fa riferimento alla computazione aggregata eseguita dal singolo dispositivo. Questo può essere considerato il punto di vista tradizionale. Il secondo punto di vista, detto *aggregate view* (Figura 1.2b), sposta invece l'attenzione sulla computazione svolta dal sistema aggregato come singola unità. Rispetto all'approccio tradizionale, tale punto di vista sposta maggiormente l'attenzione dal *come* il sistema possa funzionare al *cosa* effettivamente si desidera che faccia.



(a) Livello di astrazione *device-centric*

(b) Livello di astrazione *aggregate view*

Figura 1.2.: Le figure, riprese da [34], mostrano un esempio di sistema distribuito costituito da smartphone che monitorano la pericolosità di un luogo affollato. In (a), il programmatore deve focalizzarsi sia sul protocollo di interazione, sia sulle modalità con le quali le interazioni locali possono definire il comportamento globale desiderato. In (b), invece, egli deve ragionare solamente sul flusso di trasformazione delle strutture dati necessarie.

In letteratura, sono state adottate diverse strategie per la progettazione [34] ad un livello di astrazione così elevato. Ad esempio, l'approccio *TOTA* ("*Tuples On The Air*" [22]) prevede di rendere le interazioni tra i dispositivi implicite, mentre l'*Origami Shape Language* [12] si basa sulla composizione di costruzioni geometriche e topologiche attraverso operazioni di *folding*. Altri esempi sono le tecniche di sintesi dei dati provenienti da alcune regioni spazio-temporali per inviarli come stream ad altre regioni (delineate con *TinyDB* [13]) o le modalità con cui modelli di *computazione cloud* come *MapReduce* [19] dividono la computazione in modo automatico tra i nodi. Infine, linguaggi come *Protelis* [50], di cui tratteremo meglio in Sezione 1.2,

forniscono costrutti generalizzabili per la computazione spazio-temporale che si prestano ad un uso in ambiente IoT.

Lo studio dei suddetti approcci ha permesso di fare considerazioni sulle modalità di programmazione dei sistemi situati di larga scala; innanzitutto, laddove non sia richiesto al programmatore di gestire i meccanismi di interazione, le modalità di coordinazione dovrebbero essere robuste e nascoste "*under the hood*" [34]. Poi, il framework di programmazione dovrebbe essere modulare, e tali moduli dovrebbero essere componibili in modo semplice e trasparente, come avviene nella programmazione funzionale. Infine, dovrebbe essere possibile fornire meccanismi di coordinazione differenti per diverse parti del sistema collocate in regioni dello spazio e tempi distinti.

In letteratura [34], sono stati individuati tre principi per rispondere a queste esigenze:

1. la "macchina" programmata non è il singolo dispositivo, bensì il loro agglomerato;

2. il "programma" è specificato come manipolazione di strutture dati distribuite che evolvono nel tempo dette *computational fields*;

3. tali manipolazioni vengono eseguite dai dispositivi inseriti all'interno della regione, tramite l'uso di meccanismi di coordinazione resilienti e di interazioni basate sulla prossimità.

In questo modo, i meccanismi di coordinazione, spesso complessi, vengono nascosti facilitando la costruzione e favorendo la modularità. In particolare, il paradigma si struttura su più livelli di astrazione, come è possibile vedere in Figura 1.3.

Nelle Sezioni successive analizzeremo più nel dettaglio alcuni di questi.

1.1. Field calculus e Building Blocks

Al livello più basso della struttura rappresentata in Figura 1.3 si trova il *field calculus* [54]. Esso è un *core calculus*, ovvero un modello teorico di programmazione che riassume la semantica operazionale minimale necessaria alla progettazione di un sistema aggregato.

Il field calculus si basa sul concetto di *campo computazionale* [32, 54]: il termine riprende la nozione di campo in fisica [14], esteso al concetto di computazione e inteso come «*una proiezione di ciascun dispositivo computazionale nello spazio verso un oggetto computazionale arbitrario*», ossia l'applicazione di una funzione che, in un dato momento nel tempo, mappa ogni punto dello spazio (un dispositivo o un nodo), verso un oggetto computazionale (un valore) che rappresenta il risultato della computazione su quel device. I *campi* sono strutture dati distribuite che si adattano ai cambiamenti della topologia sottostante e alle interazioni con l'ambiente.

I campi vengono generati e manipolati attraverso cinque costrutti fondamentali [54]:

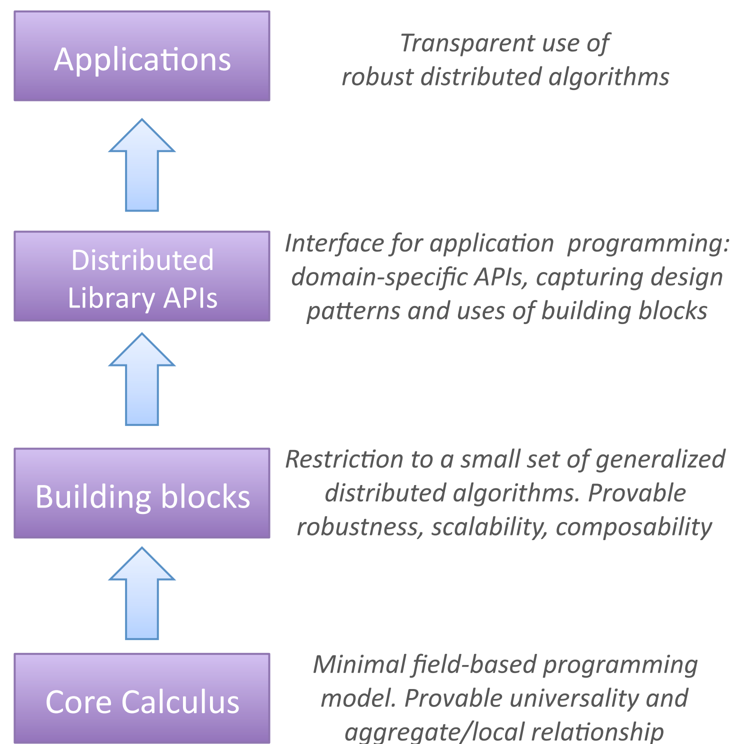


Figura 1.3.: Principali livelli dell'*Aggregate Programming*.
Figura ripresa da [38].

Operatori built-in `(b (`$e_1 \dots e_n$`))`

Un operatore *built-in* `b` modella in modo uniforme operazioni basate su valori puntuali, cioè che non coinvolgono né lo stato, né la comunicazione. Esso determina il valore del campo in output all'evento `m` (un punto nello spazio-tempo) solo dai valori dello spazio `e` e dei campi in input $e_1 \dots e_n$. Possono essere funzioni *stateless* matematiche, logiche o algoritmiche, ma anche sensori, attuatori, funzioni di libreria, ecc.

Definizione e chiamata di funzione `(def f (`$x_1 \dots x_n$`) e)`

Astrazione e ricorsione sono supportate attraverso la definizione di funzione: una funzione `f` con parametri formali $x_1 \dots x_n$ e corpo `e` può essere invocata con `(f (`$e_1 \dots e_n$`))`.

Evoluzione nel tempo `(rep x` e_0 `e)`

Il costrutto di ripetizione supporta l'evoluzione dinamica dei campi, assumendo che ciascun dispositivo computi il proprio programma ripetutamente in *round* asincroni. La variabile di stato `x` è inizializzata con il risultato della valutazione dell'espressione e_0 e aggiornato ad ogni step computando `e` in relazione al precedente valore di `x`.

Valori di vicinato `(nbr e)`

L'interazione diretta tra i dispositivi è incapsulata nel costrutto `nbr`; con esso, si ottiene il *neighbouring field*, ossia una mappa da ciascun vicino al proprio valore corrente di `s`. Funzioni "*hood*" *built-in* possono poi riassumere queste mappe.

Restrizione di dominio (`if` e_0 e_1 e_2)
La ramificazione distribuita è implementata dal costrutto `if`, che permette di suddividere la rete in due regioni: una nella quale l'espressione e_0 è vera, nel quale e_1 viene computato, e una nella quale è falsa, che invece computerà e_2. Tali suddivisioni sono incapsulate e non possono avere effetti al di fuori dei relativi sottospazi.

Questi costrutti permettono al *field calculus* di essere universale [52], supportando ogni computazione spazio-temporale causale e approssimabile. Tramite questi operatori, inoltre, sono garantite *portabilità*, *indipendenza* dall'infrastruttura e *integrazione* con servizi non aggregati.

Il livello di astrazione successivo è costituito dagli operatori "*building block*" [33]. Questo layer consiste di un insieme di operatori generici e di più alto livello, che offrono allo sviluppatore un ambiente di programmazione più espressivo, contribuendo in particolare alla cosiddetta *self-stabilization*, ossia la capacità di raggiungere uno stato atteso in un numero finito di passi, indipendentemente dallo stato di partenza. Qualora il sistema sia auto-stabilizzante, tale proprietà è garantita anche per tutti i campi ottenuti tramite composizione funzionale di questi operatori [33].

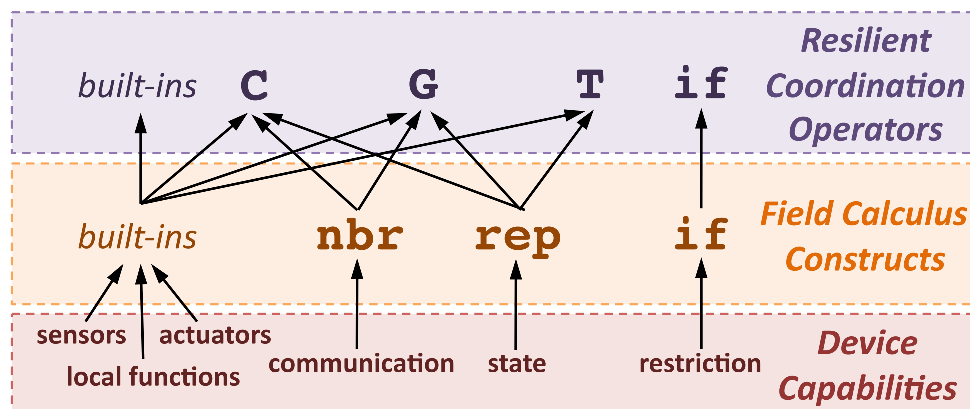


Figura 1.4.: Dettaglio dei livelli più bassi dell'*aggregate programming*.

Come riportato in Figura 1.4, i *building block* individuati in aggiunta al costrutto `if` del *field calculus* sono tre operatori di coordinazione [34, 33]:

Diffusione dell'informazione nello spazio Letteralmente «diffusione dell'informazione nello spazio», quest'operatore generalizza operazioni molto comuni come la stima della distanza e i messaggi broadcast. È definito come:

`G(source, init, metric, accumulate)`

Raccoglimento di informazione attraverso lo spazio Letteralmente «raccoglimento di informazione attraverso lo spazio», quest'operatore aggrega le informazioni verso la sorgente attraverso il gradiente di un campo specificato. È definito come:

`C(potential, accumulate, local, null)`

Riassunto dell'informazione nel tempo Letteralmente «riassunto dell'informazione nel tempo», quest'operatore generalizza un timer il cui rateo di aggiornamento può variare nel tempo. È definito come:

```
T(init, floor, decay)
```

Questi operatori sono sufficientemente espressivi da poter coprire, da soli o combinati tra loro, molti dei pattern di coordinazione usati nei sistemi a larga scala.

Come livello di astrazione ulteriore (il secondo dall'alto nella Figura 1.3), volto a semplificare la composizione dei *building blocks*, si aggiungono le *API general-purpose* [48]. Esse possono essere usate e composte tra loro per scrivere applicazioni distribuite senza preoccuparsi dei meccanismi di coordinazione, la cui robustezza è garantita dagli operatori descritti sopra.

1.2. Protelis

Field calculus è un impianto teorico sul quale devono essere costruiti linguaggi "pratici". Vista la necessità di un'architettura portabile in grado di gestire gli aspetti di comunicazione, esecuzione e interfacciamento con hardware e sistema operativo, è stato realizzato Protelis.

Protelis [50] è un linguaggio di programmazione basato sul paradigma aggregato fortemente influenzato da *Proto* [17]. Il linguaggio incorpora le principali funzionalità di computazione spaziale di field calculus in una sintassi più simile ai linguaggi strutturati tradizionali come C o Java.

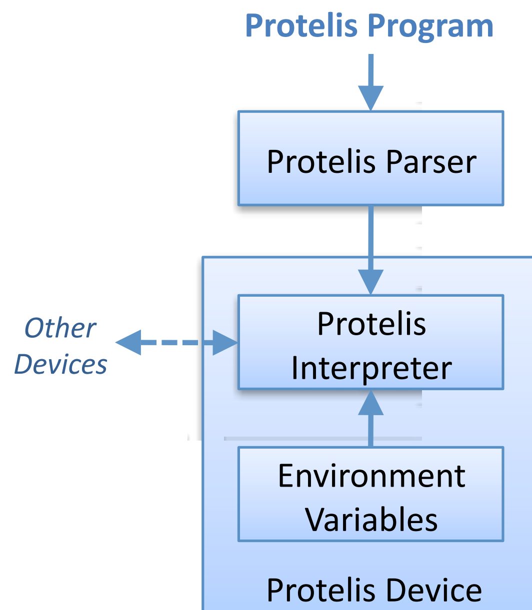


Figura 1.5.: Architettura di Protelis.
Figura ripresa da [38].

Un nodo Protelis è costituito da un *parser* che traduce il programma in codice eseguibile, il quale è poi eseguito a intervalli regolari da un *interprete*, che si fa carico degli aspetti di interazione con il vicinato e con l'ambiente. Ogni esecuzione è chiamata *computational round*.

Il linguaggio e l'interprete sono basati su Java e possono essere inseriti in contesti virtuali [38] o reali indifferentemente [46]. Questo offre, da un lato, la portabilità e il supporto alle differenti piattaforme che la JVM (*Java Virtual Machine*) mette a disposizione, dall'altro l'estendibilità che l'ecosistema di librerie Java può offrire.

Nel mondo scientifico, il linguaggio è già stato utilizzato per la realizzazione di diversi algoritmi aggregati. Di seguito sono riportati alcuni esempi.

Algoritmi legati all'affollamento Tramite Protelis, è stato possibile [34] stimare la pericolosità di una data zona nell'ambiente basandosi sulla densità dei dispositivi presenti e definire modalità di dispersione efficaci.

In un altro progetto [55] è stato definito un algoritmo di *rendezvous* in grado di evitare le zone ad alta densità nel contesto di un evento di massa, permettendo l'incontro di due individui in un punto intermedio.

Gestione di reti di servizi Un altro utilizzo significativo è stato per la realizzazione di un sistema di gestione di servizi in una rete. Tali servizi, talvolta datati, possono avere molte dipendenze tra loro e scarse capacità di coordinazione. Per evitare stati di inconsistenza, spesso l'ordine di arresto dei server è strettamente legato alle dipendenze e rende difficile l'automazione.

Utilizzando Protelis è stato possibile [35] realizzare un sistema in grado di organizzarsi per riavviare lo stack. In particolare, sono state definite entità chiamate *daemon* che monitorano ciascuna uno specifico servizio e comunicano con le altre al fine di garantire l'ordine necessario.

Coordinazione di droni Il linguaggio Protelis è stato utilizzato anche per la realizzazione di un sistema di coordinazione per una rete di sensori aerotrasportati [39, 40] tramite droni.

Integrazione con servizi di realtà aumentata La programmazione aggregata è stata testata anche nell'ambito dell'AR (*Augmented Reality*) [37].

Ad esempio, è stato possibile utilizzare visori di realtà aumentata per visualizzare nell'ambiente i campi computazionali o, viceversa, modellare i dati raccolti da sensori AR come campi (detti *augmented fields*).

1.3. ScaFi

Una tecnologia analoga è rappresentata da *ScaFi* (Sca*la with computational* Fi*elds*) [41]: si tratta di un framework in Scala per la realizzazione di programmi aggregati attraverso un set compatto di primitive, presentato come implementazione del field calculus alternativa a Protelis.

Il framework è composto principalmente da due parti:

Aggregate programming support La prima parte è un *internal DSL* (*Domain Specific Language*) di Scala che fornisce la sintassi e la semantica per i costrutti base del field calculus.

Aggregate platform support La seconda parte è una piattaforma distribuita basata sul modello ad attori di Akka che permette la configurazione e l'esecuzione di sistemi aggregati. Essa può essere utilizzata in modalità decentralizzata (*peer-to-peer*) o in modalità centralizzata (*server-based*).

Nel caso centralizzato, è la piattaforma a mantenere le posizioni spaziali dei dispositivi e a gestire il vicinato, secondo il modello tradizionale client-server.

Dato che si appoggia a Scala come linguaggio ospite, è in grado di interoperare con Java e gli altri linguaggi in grado si eseguire in ambiente JVM, mantenendo il solido *type-system* messo a disposizione da Scala e i suoi costrutti funzionali.

2. Progettazione di sistemi web

Il World Wide Web [24] ha assunto un ruolo sempre più centrale nella quotidianità delle persone e nelle dinamiche di business. In particolare, il modello di comunicazione è sempre più virato verso scenari distribuiti, nei quali piattaforme eterogenee riescono a comunicare tra loro condividendo informazioni di diverso tipo attraverso la rete Internet.

Anche i pattern di progettazione e le tecnologie implementative sono cresciute altrettanto velocemente negli ultimi anni, cambiando anche radicalmente gli approcci di interazione possibili. Risulta dunque importante prestare attenzione allo stato dell'arte in tal senso, chiarendo quali siano i pattern più adatti e moderni per il contesto d'uso di questa tesi.

2.1. Architetture & paradigmi

Con *sistema web* si intende genericamente un sistema software distribuito che coinvolge una o più entità server che espongono in rete API di varia natura, con le quali entità client possono comunicare per usufruire dei servizi. Generalmente, in contesto web i client sono costituiti da pagine web aperte nei browser degli utenti.

Le possibilità di progettazione di un'applicazione web possono essere molto differenti e nel tempo si è vista una vera e propria evoluzione in tal senso.

Nel periodo iniziale del web, ciascuna pagina era inviata al client come documento statico; in questo caso, i server si occupavano della computazione di quanto richiesto dall'utente (anche appoggiandosi a servizi esterni tramite *Common Gateway Interface* [16]) e della composizione del documento. Nel corso degli anni, sono stati sviluppati linguaggi di scripting (come JavaScript e Macromedia Flash) in grado di aumentare le possibilità di interazione lato client, riducendo la quantità di dati trasferiti tra il server web e il browser: non viene infatti più scaricata una nuova pagina ad ogni azione dell'utente, bensì solo i dati richiesti tramite chiamate *AJAX* (*Asynchronous JavaScript And XML*); il documento viene poi manipolato localmente inserendo i dati ricevuti.

Con l'avvento di HTML 5 [20], viene introdotto il supporto nativo ai contenuti multimediali ed arricchita la semantica del documento; inoltre JavaScript riceve un supporto di prima classe e la necessità di plugin esterni come Flash diminuisce.

Vengono sviluppati numerosi framework per la realizzazione di applicazioni costituite da una singola pagina (*SPA*, *Single-Page Application*), in grado di offrire l'esperienza d'uso di un applicativo desktop. Un applicazione web moderna normalmente è di questo tipo.

2.2. Linguaggi ad uso web

In questa Sezione verranno analizzati i principali linguaggi di programmazione utilizzati recentemente per lo sviluppo della componente frontend delle applicazioni web. In particolare, verranno presi in considerazione i due linguaggi più popolari in questo contesto, ossia *JavaScript* e il suo super-set *TypeScript*, e due linguaggi non nativi della piattaforma che possono offrire una valida alternativa: *Scala.js* e *Kotlin/JS*.

2.2.1. JavaScript e ECMAScript

JavaScript è un linguaggio di scripting debolmente tipizzato multi-paradigma, con supporto agli stili di programmazione funzionale, ad eventi e orientato agli oggetti. Sviluppato originariamente nel 1995 da Brendan Eich per Netscape Communications (inizialmente con il nome di *Mocha* e poi *LiveScript*), esso è stato concepito con lo scopo di avere un linguaggio di scripting per il browser Netscape Navigator più semplice da apprendere rispetto a quelli esistenti. JavaScript è stato standardizzato per la prima volta nel 1997 dalla ECMA con il nome di *ECMAScript* [9, 10] e l'attuale versione è la undicesima.

Il linguaggio è attualmente il più popolare per uso web, in quanto l'unico ad essere supportato da tutti i browser moderni, almeno nelle sue feature principali. Diversi linguaggi, come quelli che vedremo in seguito, vengono transcompilati in una versione sufficientemente supportata di JS da poter essere eseguiti nei browser.

Analizzato dal punto di vista tecnico, esso presenta i seguenti aspetti strutturali:

Imperativo e strutturato Il linguaggio si presenta con una sintassi di programmazione strutturata standard, con il supporto a tutte le strutture di controllo tradizionali. Una parziale differenza era presente nella gestione della visibilità delle variabili (*scope*): inizialmente, JavaScript garantiva solo la visibilità a livello di funzione (*function scope*) tramite la parola chiave `var`; con ECMAScript 6 è stato aggiunto il supporto alla visibilità a livello di blocco (*block scope*).

Tipizzazione dinamica JavaScript è un linguaggio debolmente tipizzato: alle variabili non sono associati dei tipi di dato, ma solo dei valori, che possono dinamicamente cambiare tipo durante il ciclo di vita della variabile. La tipizzazione dinamica consente lo stile di tipizzazione chiamato *duck typing* [29] o *structural subtyping* [42] a seconda delle definizioni: è possibile determinare la semantica di un oggetto in base ai metodi ed alle proprietà che esso possiede, non in base al suo tipo.

Orientamento agli oggetti Prototype-based A differenza di altri linguaggi orientati agli oggetti (come Java o C++), JavaScript non fornisce un'implementazione del concetto di *classe* [6]: la stessa keyword `class`, introdotta con ES6, non è altro che zucchero sintattico per migliorare l'interazione da parte degli sviluppatori con il prototipo dell'oggetto.

In termini di ereditarietà, infatti, JS prevede un solo costrutto: gli *oggetti*. Ogni oggetto ha un collegamento interno ad un altro oggetto chiamato *prototype*. Questo oggetto prototipo ha a sua volta un suo prototype, e così via finché si raggiunge un oggetto con la proprietà prototipo settata a `null`. Per definizione, `null` non ha un prototype ed agisce come anello finale nella *catena dei prototipi*.

Quasi tutti gli oggetti in JavaScript sono istanze di `Object`, che risiede in cima alla catena dei prototipi.

First-class function JavaScript offre un supporto di prima classe alle funzioni, che sono considerate oggetti; come tali, esse possono avere delle proprietà, come ad esempio `.bind()` e `.call()`. Ciascuna funzione costituisce una *chiusura lessicale*.

All'interno degli oggetti, le proprietà di tipo funzione vengono utilizzate come costruttori e come metodi. Inoltre, le funzioni possono essere utilizzate in combinazione alle classi per anche per l'implementazione di *pattern di ruolo* come i *tratti* [26] e i *mixin*.

Web APIs Con ECMAScript si standardizza la componente *core* del linguaggio, che può eseguire in ambienti browser come su interpreti non legati strettamente al web (come ad esempio Node.js [25]). Ciascuno degli ambienti nei quali il linguaggio viene interpretato mettono a disposizione delle API di libreria specifiche per l'interazione con la piattaforma; quando si fa riferimento al browser, tali supporti sono chiamati "*Web APIs*".

Attraverso di esse, lo sviluppatore può avere accesso agli elementi del DOM (Document Object Model) di HTML, potendo manipolare la pagina visualizzata e reagire ad eventi su di essa.

Asincronismo JavaScript supporta inoltre nativamente l'esecuzione di operazioni in modo asincrono: inizialmente la soluzione di riferimento era il *Continuation-passing style* tramite funzioni di callback, mentre con ES6 è stato introdotto il costrutto della *promise*. Un oggetto built-in `Promise` implementa quest'ultimo pattern, andando a costituire un *proxy* per un valore non necessariamente noto quando la promise viene creata. È poi possibile gestire il valore ottenuto con costrutti "*thenable*" o con il *pattern async/await*.

2.2.2. TypeScript

TypeScript è un linguaggio di programmazione open-source sviluppato da Microsoft. Esso è un super-set di JavaScript ES6 nato con l'obiettivo principale di offrire un'esperienza di sviluppo più robusta e moderna.

Innanzitutto, il linguaggio prevede l'introduzione di funzionalità *cutting-edge* (talvolta anche nei primi stage di approvazione) di ECMAScript, rese disponibili transcompilando il sorgente TypeScript in codice JavaScript meno recente ma completamente compatibile con l'ambiente di esecuzione scelto (ad esempio ES5 per i browser), spesso avvalendosi di *polyfill* per estenderne la compatibilità. Tra queste funzionalità, ci sono ad esempio:

Decoratori Definiti in una proposta ECMAScript[1] tutt'ora pendente (stage 2 al momento della scrittura), i decoratori sono dichiarazioni (definite tramite carattere "@") associate a una dichiarazione di classe, un metodo, una proprietà o un parametro che permettono la valutazione a runtime di una espressione specificata. Il framework Angular, ad esempio, utilizza estensivamente questa funzionalità per effettuare *dependency injection.*

Operatore di coalescenza del null & safe navigation In JavaScript, sono definiti "*falsy*" valori come `NaN`, `0` e la stringa vuota che, pur non essendo `null` o `undefined`, vengono trattati come valori "non presenti" e dunque come `false` dagli operatori booleani. Per ridurre la possibilità di bug dovuto a questo tipo di valori, sono stati definiti gli operatori di *nullish coalescing* "`??`" e di *safe navigation* "`.?`". Sono stati introdotti con la versione 3.7 di TypeScript sulla base di quanto definito in una precedente proposta ECMAScript[2], solo più tardi integrata in ES11.

Inoltre, il linguaggio introduce un *type system statico* opzionale espresso tramite *sintassi postfissa*; ale sistema di tipi viene presentato come *class-based* (in contrapposizione al *prototype-based* tipico di JavaScript, soprattutto prima dell'introduzione dello zucchero sintattico per le classi), ma mantiene le caratteristiche di structural subtyping citate precedentemente. Avvalendosi di ciò, il linguaggio permette la definizione di tipi complessi: è infatti garantito il supporto ai generici, esteso da un'algebra dei tipi molto flessibile che permette unione, intersezione e condizionalità, oltre alla possibilità di accesso ai tipi delle singole proprietà.

La generazione dei metadati per la descrizione dei tipi permette un'integrazione migliore di TypeScript e JavaScript con gli ambienti di sviluppo; inoltre, negli anni è diventato lo standard *de-facto* per la documentazione dei moduli per il linguaggio, sostituendo le soluzioni basate sul parsing dei commenti come JSDoc.

2.2.3. Scala.js

Scala.js è un compilatore per il linguaggio di programmazione Scala che genera codice JavaScript invece di bytecode per la JVM. I principali vantaggi dell'impiego di un linguaggio come Scala all'interno del contesto web sono i seguenti:

Struttura più solida Come detto, JavaScript è un linguaggio debolmente tipizzato; se questo aggiunge notevole flessibilità durante lo sviluppo, d'altro canto aggiunge una maggiore possibilità di bug. Anche TypeScript, che offre un sistema di tipi molto più completo, risulta talvolta anomalo per via della natura strutturale della tipizzazione implementata.

Scala mette a disposizione un type system allo stesso tempo più espressivo e più rigoroso, di conseguenza può risultare più *user-friendly* sia durante lo sviluppo che in fase di debug.

[1] `https://tc39.es/proposal-decorators`
[2] `https://tc39.es/proposal-optional-chaining`

Prestazioni Secondo benchmark [53] realizzati tramite una suite estensiva già utilizzata in letteratura [43], per quanto Scala.js risulti più lento della controparte JVM, riesce ad essere fino al 33 % più veloce di un programma equivalente scritto in JavaScript.

Inoltre, il compilatore risulta molto efficiente in termini di dimensioni finali del *bundle*: secondo il sito ufficiale, generalmente si parte dai 45 kB per un'intera applicazione compressa in gzip. Questo permette di avere buone performance al primo caricamento dell'applicazione web.

Interoperabilità con JavaScript Scala.js è dichiaratamente compatibile con qualsiasi modulo JavaScript, purché sia dotato del codice *façade* necessario per garantirne il type checking. Il team che mantiene il progetto ha effettuato il wrapping di numerose librerie di uso comune (50 al momento della scrittura) e mette a disposizione uno strumento di conversione automatico per le definizioni di tipo generate dal compilatore di TypeScript. Data la popolarità di quest'ultimo, la copertura può essere dunque considerata elevata.

Riuso Un'altra compatibilità utile nella realizzazione di sistemi web di grandi dimensioni è quella con codice Scala impiegato anche nel backend. Questo permette di riutilizzare il codice condiviso tra server e client, riducendo i costi di manutenzione del medesimo codice in linguaggi differenti e la possibile *friction* originata dall'etereogeneità dei linguaggi.

Non è possibile riutilizzare codice che dipende strettamente da funzionalità della JVM (*reflection*, eccezioni native di Java), in quanto porta a comportamenti indefiniti.

Scala inoltre ha un'integrazione molto buona con ambienti di sviluppo comunemente utilizzati, che può portare a una rilevazione degli errori più veloce e ad un autocompletamento più preciso.

2.2.4. Kotlin/Multiplatform e Kotlin/JS

Con il rilascio della versione 1.1 di Kotlin nel 2017, JetBrains ha annunciato *Kotlin/JS* [47], un target del compilatore in grado di generare codice JavaScript ES5 invece che bytecode per la JVM, in modo simile a quanto fatto da Scala.js. Il progetto si colloca in un progetto di più ampio respiro, ossia *Kotlin/Multiplatform* [49], presentato al termine dello stesso anno in concomitanza con Kotlin 1.2, che mira a integrare anche il supporto ad altri target, tra cui ad esempio LLVM.

L'intento è molto simile a quello di Scala.js: fornire un linguaggio solido per la realizzazione di componenti web (ad esempio un frontend) con lo stesso codice in esecuzione sul server con cui dialogare, in modo da coprire l'intero sistema nel modo più omogeneo possibile.

Kotlin, però, a differenza di quanto avviene in Scala.js, prevede l'utilizzo di sorgenti separati per la parte comune (*common source set*), che modella il sistema in Kotlin puro, e per la parte specifica di ciascun particolare target: in questo modo, differenti piattaforme possono avere implementazioni più vicine alle rispettive native, avvalendosi di librerie specifiche fornite dal sistema o da terze parti.

Anche in questo caso, le criticità sono simili a quelle citate nella Sezione precedente: in primo luogo, per quanto l'intera libreria standard sia stata portata da JetBrains sulle diverse piattaforme nel modo più trasparente possibile, le piattaforme di esecuzione rimangono differenti. Di conseguenza, alcuni costrutti molto specifici strettamente legati alla JVM (come ad esempio la reflection) non sono disponibili o hanno limitazioni sul target JS.

Inoltre, per garantire il supporto corretto ai tipi, è necessaria la presenza di codice *wrapper* che li modelli. JetBrains ha realizzato gli adattamenti solo per poche librerie, principalmente legate a React, e ha messo a disposizione alcuni strumenti per la conversione delle definizioni di tipo di TypeScript. Purtroppo, la prima soluzione, `ts2kt`, è stata deprecata solo dopo qualche anno e il sostituto, `dukat`, al momento della scrittura non è ancora considerato stabile. In assenza dei wrapper, è comunque possibile lavorare con moduli JavaScript attraverso il costrutto `dynamic`, ma si perde il controllo sui tipi.

JetBrains pare supportare attivamente l'utilizzo di Kotlin/JS per la realizzazione di codice frontend (in particolare con la libreria React), ma attualmente non viene ritenuto ancora sufficientemente stabile per un uso reale. Sul sito ufficiale sono invece specificati numerosi framework per la realizzazione di backend che offrono supporto di prima classe al linguaggio.

3. Motivazioni e Stato dell'arte

Come detto nell'Introduzione, il rapido aumento di dispositivi capaci di computare e distribuiti nell'ambiente ha reso necessario ideare nuovi paradigmi di programmazione. Uno di questi è senza dubbio l'*aggregate programming*, di cui si è trattato nel dettaglio nel Capitolo 1. Prima di partire con la progettazione del sistema, è stato necessario valutare lo stato dell'arte nel quale il software va ad inserirsi. In particolare, si è preso in considerazione la procedura di configurazione di un progetto realizzato in un linguaggio aggregato, confrontandolo con l'esperienza di sviluppo ottenuta con altri linguaggi in altri contesti.

In questo Capitolo verrà analizzato lo stato dell'arte relativamente alla creazione di un programma aggregato, mettendo in evidenza le ragioni per la quale è stato realizzato questo progetto di tesi.

3.1. Accessibilità allo sviluppo in Protelis

I due principali linguaggi di programmazione aggregata che implementano il modello teorico del field calculus sono Protelis e ScaFi. ScaFi è un DSL interno a Scala e, come tale, richiede di essere utilizzato all'interno di progetti basati su tale linguaggio; poiché sono già state realizzate tesi [45, 51] incentrate sull'esecuzione di ScaFi, è stato richiesto di concentrarsi, in questa tesi, solamente su Protelis.

Protelis è un linguaggio *Java-hosted* e, come tale, deve essere caricato in un progetto JVM che esegue un'istanza del suo interprete. Di seguito sono riportate le possibili modalità di esecuzione documentate nello stato dell'arte.

Alchemist [38] Una prima possibilità di utilizzo è tramite il simulatore Alchemist [31]. Esso mette infatti a disposizione all'interno del proprio meta-modello un'incarnazione specifica per Protelis. Esso permette la simulazione di reti con pattern anche complessi con semplici file di configurazione. In Figura 3.1 sono riportate le entità in gioco.

Alchemist può essere utilizzato all'interno di altro software in ambiente Java come libreria di virtualizzazione, oppure in modo indipendente tramite linea di comando[1]. In questo secondo caso, è possibile definire il codice Protelis insieme a file di configurazione in formato YAML e specificarli al lancio del file jar eseguibile di Alchemist. Il simulatore

[1] `https://github.com/Protelis/Protelis-Alchemist-tutorial`

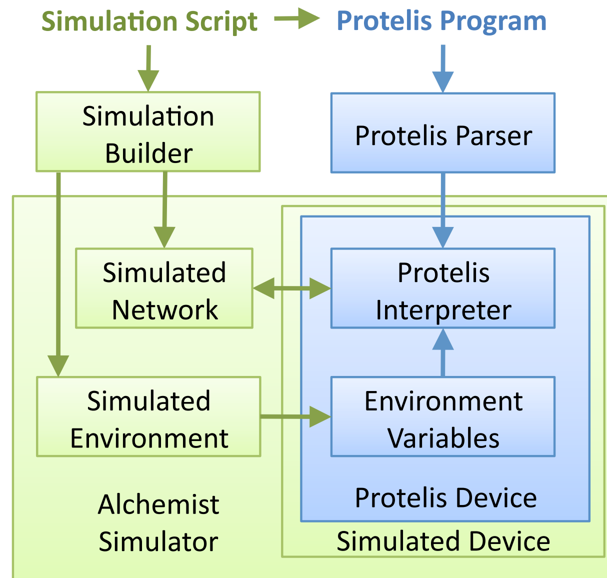


Figura 3.1.: Implementazione dell'architettura di Protelis in Alchemist.
Figura ripresa da [38].

mette a disposizione un'interfaccia grafica minimale per la visualizzazione e il controllo della simulazione.

ProtelisVM [56] Un'altra possibilità è tramite l'importazione del framework di Protelis nel *classpath* di un progetto Java, ad esempio prelevandolo come dipendenza Maven. Il framework mette a disposizione una classe denominata `ProtelisVM` (Figura 3.2), che si occupa di eseguire un programma Protelis in un `ExecutionContext`, ossia un'astrazione del dispositivo esecutore che si frappone tra la macchina virtuale e l'ambiente fisico.

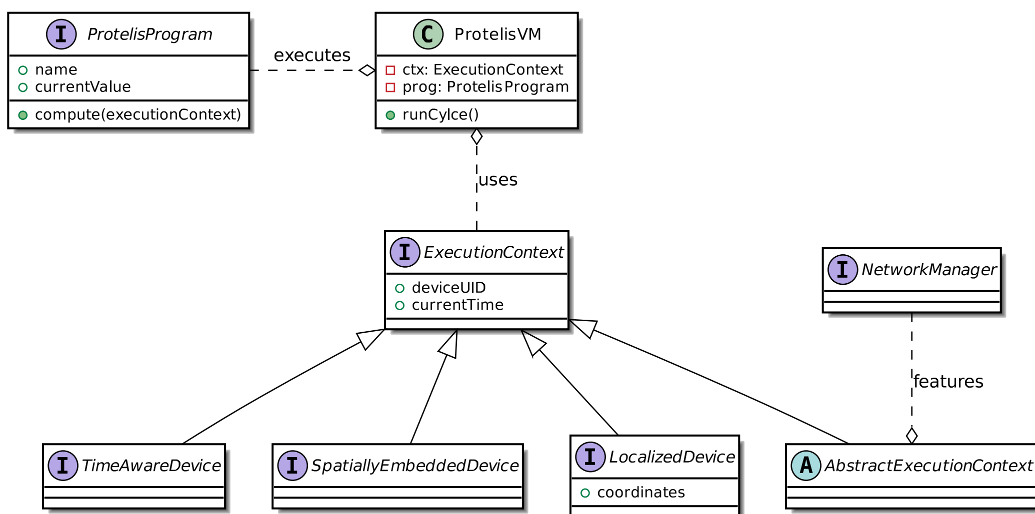


Figura 3.2.: UML delle principali entità del framework di Protelis

Attraverso la libreria è possibile definire dispositivi virtuali o, potenzialmente, collegare

implementazioni fisiche. Su GitHub[2] sono riportati alcuni esempi con diverse opzioni di integrazione.

NASA WorldWind [18] Un'ultima possibilità vede invece l'utilizzo del framework di visualizzazione open-source WorldWind, sviluppato dalla NASA. Esso è stato utilizzato per dimostrare come Protelis possa essere uno strumento che permette di controllare anche dispositivi reali come uno sciame di droni. Il codice della demo è pubblico su GitHub[3].

In ciascuno di questi esempi risulta evidente che la configurazione di un progetto per Protelis, anche estremamente minimale, coinvolge strumenti esterni la cui complessità può non essere banale. Sarebbe interessante avere a disposizione un ambiente di sviluppo che non richieda configurazione e permetta di provare prototipi di codice durante l'apprendimento del linguaggio.

3.2. Ambienti di sviluppo online

Con il progredire delle capacità delle applicazioni JavaScript per browser e della popolarità del linguaggio stesso, sono nate numerose implementazioni di IDE (*Integrated Development Environment*) in grado di eseguire all'interno di una pagina web, comunicando al più con un *language server* remoto.

Inizialmente sono stati appannaggio di ambienti per la prototipazione di codice JavaScript, in quanto si avvalevano del motore nativamente integrato nel browser per l'esecuzione; attualmente molti linguaggi offrono un ambiente di questo tipo, spesso chiamato *playground*, in cui sperimentare (ad esempio Kotlin, TypeScript o Scala). Alcuni ambienti di questo tipo, addirittura, sono in grado di offrire un'esperienza utente tanto immediata e completa che talvolta vengono preferite da alcuni a installazioni desktop tradizionali. È questo il caso, ad esempio, di Overleaf.

Overleaf[4] è un editor per LaTeX completamente online che permette all'utente di scrivere documenti tramite browser; il sorgente del markup viene compilato in modo trasparente all'utente, il quale deve preoccuparsi unicamente del contenuto che sta scrivendo. Questo risparmia agli utenti inesperti la fase di installazione e configurazione di una distribuzione LaTeX e la scelta di un editor tra i tanti disponibili.

Potrebbe essere interessante offrire all'utente novizio di Protelis un'esperienza simile: la rete di dispositivi (reale o simulata) viene configurata nel server, insieme all'interprete per il linguaggio. L'utente avrebbe dunque a disposizione, tramite il proprio browser, solo un semplice editor per scrivere il codice e metterlo in esecuzione.

[2] https://github.com/Protelis/Protelis-Demo
[3] https://github.com/Protelis/Protelis-Demo-Visualized
[4] https://www.overleaf.com

Parte II.

Contributo: WebProtelis

4. Requisiti e Analisi

In questo Capitolo saranno enunciati ed analizzati i requisiti del progetto realizzato per questa tesi; ne sarà poi valutata la fattibilità e verrà descritta una prima architettura logica.

4.1. Requisiti della piattaforma

Partendo dalle considerazioni fatte analizzando lo stato dell'arte nel Capitolo 3, sono stati identificati diversi requisiti per la piattaforma che si è deciso di denominare WebProtelis. In particolare, l'obiettivo principale di questa tesi è progettare un sistema che permetta all'utente di iniziare a utilizzare un linguaggio aggregato come Protelis richiedendo meno configurazioni possibile. La componente che deve interfacciarsi con l'utente, il quale si assume essere inesperto della piattaforma, dovrebbe astrarre la maggior parte della complessità e modellare esclusivamente le funzionalità che l'utente potrà utilizzare.

4.1.1. Requisiti funzionali

Di seguito sono riportati i requisiti funzionali del sistema, che descrivono quali funzionalità devono essere offerte dal sistema.

Nessun setup Essendo orientato alla sperimentazione con il linguaggio, l'esperienza d'uso deve essere il più semplice possibile. In particolare, nel prototipo non deve essere necessario configurare una rete di alcun tipo per poter realizzare codice aggregato ed eseguirlo.

Protelis Dei linguaggi analizzati precedentemente, il prototipo del sistema da realizzare per questa tesi deve supportare Protelis.

Modificare il programma Per quanto possa essere utile avere codice di esempio già inserito nel campo di testo, l'applicazione web deve permettere all'utente di sperimentare con i costrutti del linguaggio, avendo la possibilità di scrivere il proprio codice nell'editor. Tale editor deve offrire per quanto possibile un'esperienza di scrittura che ricordi quella di un editor di codice desktop.

Lanciare l'esecuzione L'applicazione deve permettere di lanciare il codice scritto dall'utente su una rete predeterminata di dispositivi. Tale rete deve essere trasparente all'utente.

Visualizzare l'esecuzione L'applicazione deve permettere di osservare graficamente il progresso dell'esecuzione del codice scritto dall'utente.

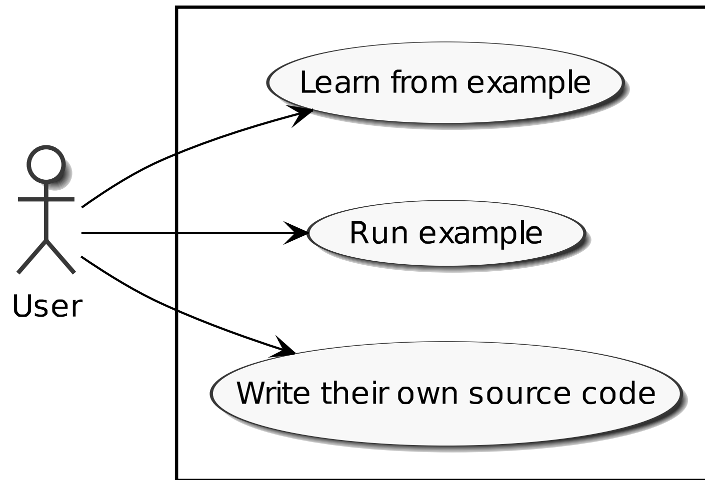


Figura 4.1.: Il diagramma UML rappresenta i casi d'uso principali dell'interfaccia

4.1.2. Requisiti non funzionali

Di seguito sono riportati i requisiti non funzionali che descrivono le proprietà non comportamentali che il sistema deve possedere.

Single-Page Application Dal punto di vista dell'utente, l'applicativo web deve presentarsi come una singola pagina, permettendogli di gestire tutte le interazioni in un unico luogo.

Efficienza L'esecuzione non deve appesantire il dispositivo client su cui esegue. Deve sfruttare in modo efficace le risorse messe a disposizione dalla macchina dell'utente.

Responsive, ma desktop-first L'applicazione ha come destinazione d'uso il desktop, dunque non è necessaria un'interfaccia *mobile-first*. È comunque necessario che il layout della pagina sia *responsive*, ossia possa adattarsi a schermi di differenti misure e proporzioni.

Compatibilità con i browser Il sistema dovrebbe essere supportato da quante più piattaforme browser possibile, con particolare attenzione a quelli più usati.

4.2. Analisi dei requisiti e vincoli di fattibilità

L'applicazione web delineata dai requisiti non richiede strettamente la presenza di un server di appoggio: potenzialmente, come è stato visto anche nelle Sezioni 2.1 e 3.2, le tecnologie sono sufficientemente mature da permettere la realizzazione di un ambiente in grado di modificare ed eseguire codice interamente all'interno della sandbox di un browser. In questo caso specifico,

però, si è ritenuta imprescindibile la presenza di un esecutore remoto esterno. Infatti, come è stato più volte sottolineato nella Parte I, il linguaggio Protelis si appoggia alla piattaforma JVM per la sua esecuzione. Al momento della scrittura, l'unico supporto da parte delle tecnologie browser per Java ritenuto stabile era dato dal plugin per le Applet API. Tale plugin è stato deprecato da alcuni anni [44], in concomitanza con il rilascio di Java 9; la quasi totalità dei browser ne ha ormai dismesso la compatibilità o lo farà a breve. Poiché il supporto ai browser più recenti e utilizzati è un requisito fondamentale, realizzare un esecutore per Protelis client-side è, al momento, impraticabile.

Risulta dunque necessario distinguere i requisiti per il server esecutore dai requisiti dell'applicazione front-end che svolgerà il ruolo di client.

4.2.1. Requisiti del client

La componente client mantiene i requisiti funzionali già delineati ed analizzati nella Sezione 4.1.1. Anche i requisiti non funzionali delineati nella Sezione 4.1.2 rimangono validi, ma viene aggiunto un ulteriore requisito.

Agnostico rispetto al backend La rete dispositivi a cui si collega per l'esecuzione deve poter essere reale o virtuale senza che questo influenzi l'esperienza utente con il frontend. Le tecnologie utilizzate per l'implementazione del backend devono essere trasparenti al client.

4.2.2. Requisiti del server

La componente server di questo progetto non deve interfacciarsi direttamente con l'utente, bensì fornire delle API generiche per l'esecuzione di codice Protelis proveniente dal client.

Per questa entità, sono stati individuati i seguenti requisiti:

Esecuzione di codice Protelis Il server deve poter eseguire codice Protelis ricevuto tramite le proprie API esposte in rete. In particolare, il server deve essere in grado di generare reti simulate di dispositivi su cui eseguire il codice *on-demand*.

Supporto a più esecuzioni contemporanee Il server deve permettere a più utenti di lavorare con il sistema contemporaneamente. In particolare, deve essere in grado di gestire più simulazioni, ciascuna distinta dalle altre e associata al client che l'ha richiesta.

Mantenimento della connessione Il server deve essere in grado di mantenere una connessione bidirezionale stabile con ciascun client, in modo da permettere al client di ottenere aggiornamenti sullo stato dell'esecuzione.

Per quanto riguarda i requisiti non funzionali, sono state delineate le seguenti proprietà:

Scalabilità Il sistema deve essere aperto alla possibilità di essere scalato efficacemente. In particolare, non è necessario che il prototipo sia in grado di scalare autonomamente, ma deve permettere l'introduzione di un eventuale orchestratore senza particolare difficoltà.

Protocollo di connessione efficiente Il server deve esporre le proprie API tramite un protocollo efficiente dal punto di vista delle performance, non andando a limitare in modo sensibile la velocità con cui il frontend viene informato dei progressi dell'esecuzione. Inoltre, tali API dovrebbero permettere di sostituire lo standard di comunicazione senza la necessità di effettuare modifiche notevoli nel motore.

Compatibilità con i browser Il protocollo di comunicazione utilizzato dal prototipo dovrebbe essere supportato da quante più piattaforme possibile (sia dal punto di vista dei browser che del server).

4.3. Architettura logica

Una volta terminata l'analisi dei requisiti e del problema, è possibile delineare un'architettura logica che possa essere un punto di partenza per la fase di progettazione, descritta nel Capitolo 5. In particolare, la piattaforma da realizzare appare come un sistema software distribuito con struttura client-server, come è possibile vedere in Figura 4.2: un'interfaccia Single-Page accessibile tramite browser (il client) permette all'utente di avere accesso alle API di un server, le quali nascondono completamente tutta la complessità di configurazione della rete (reale o simulata), permettendo l'esecuzione del codice e il monitoring remoto.

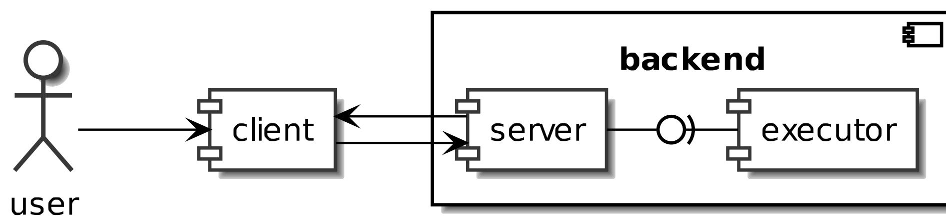


Figura 4.2.: Il diagramma UML riporta l'architettura di massima dei componenti del sistema

5. Progettazione

Nelle Sezioni successive di questo Capitolo si intende descrivere la progettazione di dettaglio del sistema: in particolare, si intende partire con lo studio dell'interfaccia grafica, per poi prendere in considerazione l'architettura logica riportata in Figura 4.2, focalizzandosi sulla progettazione dell'architettura di dettaglio di ciascun componente.

5.1. Design dell'applicazione

5.1.1. Mockup dell'interfaccia

Una volta chiariti i requisiti e le possibili fonti di ispirazione per la struttura della UI da realizzare, sono stati disegnati dei mockup che potessero rappresentare una linea guida per l'implementazione concreta dell'interfaccia.

Come detto anche nella Sezione 3.2, la struttura grafica dell'applicazione dovrebbe ispirarsi a quella di altri ambienti di sviluppo online, come ad esempio Overleaf (Figura 5.1).

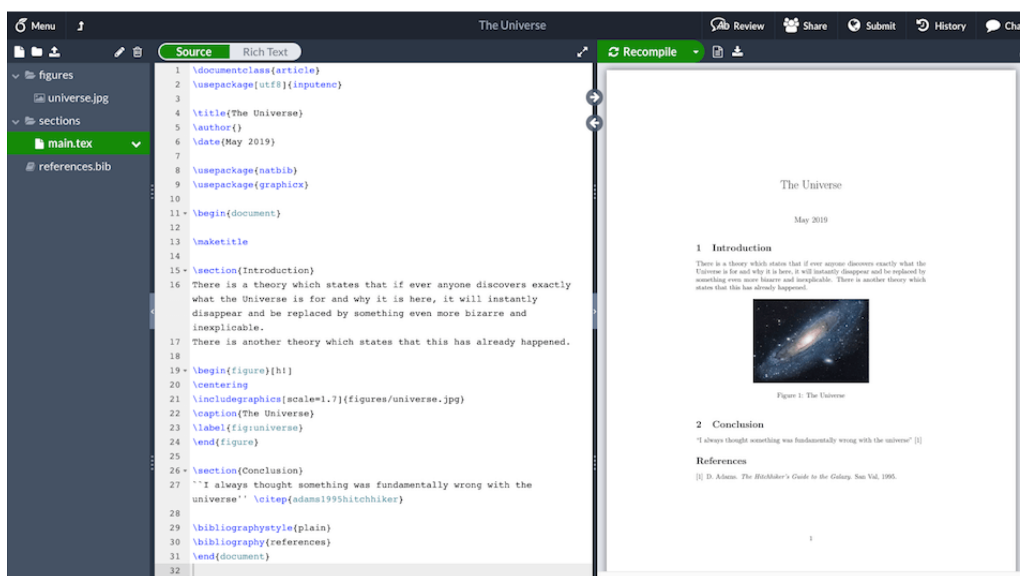


Figura 5.1.: Screenshot prelevato dalla pagina principale della web app Overleaf

Tali applicazioni hanno generalmente una struttura bipartita: nella parte sinistra è solitamente presente un editor che ricorda quello disponibile in diverse IDE desktop, mentre nella parte destra viene generalmente inserita una visualizzazione dell'output. Ad esempio, in Overleaf è possibile, alternativamente, visualizzare il log degli errori o il documento compilato.

Nel mockup finale, riportato in Figura 5.2, ci si è ispirati a questo tipo di struttura. L'interfaccia dovrebbe infatti essere costituita dalle parti seguenti:

- Una *barra superiore*, nella quale è riportato il nome e il logo del progetto, insieme a un selettore per il backend. Nei primi mockup, tale selettore era posizionato nella sezione principale della pagina, ma successivamente si è preferito spostarlo per sfruttare al meglio lo spazio a disposizione.
- Una *sezione di sinistra*, che costituisce la parte con cui l'utente può interagire per lavorare sul codice. Il componente principale è appunto l'editor, un campo di testo avanzato che permette di visualizzare il codice Protelis di esempio e modificarlo. Sotto di esso sono presenti i bottoni di controllo per interagire con l'esecuzione.
- Una *sezione di destra*, che ospita un canvas in cui l'esecuzione viene rappresentata. Al suo interno verranno visualizzati i nodi su cui il codice sta eseguendo.

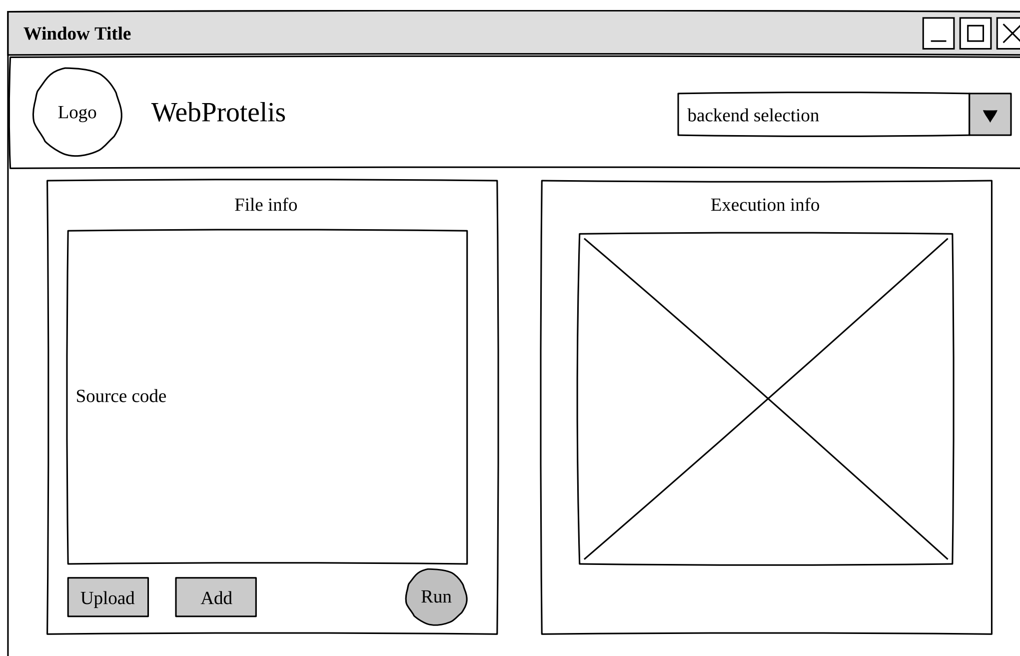


Figura 5.2.: Mockup dell'interfaccia che dovrà presentare la pagina web

5.1.2. Design di riferimento

Come è stato già sottolineato, l'applicazione vede come utilizzo principale quello dell'utente inesperto del linguaggio. L'interfaccia non doveva dunque essere solo semplice, ma anche moderna, gradevole e intuitiva. Era dunque necessario scegliere uno stile grafico familiare, moderno e facilmente adattabile a quella che sarebbe essere la nuova interfaccia che si stava progettando.

Prendendo come esempio l'interfaccia di Overleaf (Figura 5.1), è possibile notare come il design di base abbia uno stile di tipo *flat*; si è deciso dunque di valutare tra i principali design possibili quali fosse più adeguato per la UX che si aveva intenzione di progettare.

La scelta è infine ricaduta sul Material Design sviluppato da Google: dal suo annuncio nel giugno del 2014 al Google I/O 2014 Keynote esso è stato almeno parzialmente adottato in molte applicazioni web, mobile e desktop e ben si si presta all'implementazione di un'interfaccia semplice e minimale.

Per offrire un'esperienza coerente, si è deciso di utilizzare le icone e le direttive in merito a dimensioni e variazioni nella palette di colori fornite da Google[1]. Il colore base utilizzato per generare la palette è stato ricavato dall'icona ufficiale di Protelis.

5.2. Architettura del client

L'applicazione web che svolge il ruolo di client è a tutti gli effetti un'applicazione indipendente dotata di interfaccia grafica. Sono numerosi i pattern di modellazione documentati in letteratura. La caratteristica maggiormente ricercata durante la progettazione è la *reattività*: il sistema dovrebbe aggiornarsi rapidamente sia quando l'utente lo richiede, interagendo via browser, sia quando il server manda un aggiornamento.

5.2.1. Framework di sviluppo

Sono disponibili numerosi framework per lo sviluppo di applicazioni web *single-page*, ciascuno dei quali ottimizzato per determinati pattern di progettazione. Essendo un requisito la realizzazione di una SPA, la scelta di quale framework impiegare è fondamentale già in fase di progetto, in quando può notevolmente condizionare il piano di lavoro.

Per l'implementazione di questo prototipo è stato scelto il framework React, sviluppato da Facebook e compatibile, ufficialmente o meno, con numerosi linguaggi. Tecnicamente React, senza prendere in considerazione gli strumenti sviluppati intorno ad esso, sarebbe una libreria per la costruzione di pagine web reattive e *data-driven*; esso potrebbe essere considerato, riduttivamente, il *view layer* dei pattern architetturali *MV** (*Model View Anything*). React non è però vincolato al pattern MVC come AngularJS o a MVVM [28] come Angular dalla versione 2 in poi.

La divisione principale che determina la struttura è quella tra *componenti*. In React, un componente è un'astrazione che incapsula i dati, la loro manipolazione e la logica di rappresentazione e va a definire il più piccolo elemento costitutivo dell'applicazione. Esso rimuove la necessità del *data-binding* tra modello e vista, tipico dei pattern MV*, e mantiene la logica applicativa all'interno di ciò a cui fa riferimento.

[1] `https://material.io`

Un componente definisce insomma cosa deve essere renderizzato; il sistema, autonomamente, determina in modo reattivo quando una delle dipendenze è cambiata e il componente può essere singolarmente ridisegnato.

In questo modo, è possibile costruire applicazioni componendo tra loro questi elementi in una struttura simile a un albero, delegando la logica di gestione al motore di React, che se ne occuperà in modo efficiente.

La progettazione dell'architettura deve dunque spostarsi sulla gestione dello stato.

5.2.2. Pattern di gestione dello stato

Durante la fase di progettazione di un sistema software, è fondamentale definire la struttura con la quale le singole componenti saranno organizzate.

Nel contesto applicativo di software orientati all'interazione con l'utente tramite interfaccia grafica, come ad esempio un'applicazione web, è spesso consigliato definire chiaramente la gestione dello stato e del flusso informativo, quando questi non sono imposti da eventuali framework utilizzati.

Per esempio, come accennato nella Sezione 5.2.1, React prevede che la struttura dei componenti sia definita da un albero: lo stato di questi può fluire solo verso il basso, dunque si consiglia di mantenerlo il più alto possibile nella gerarchia; si parla infatti di "*state lift-up*". Questo tipo di soluzione può risultare limitante in caso di un numero elevato di componenti, dunque sono stati elaborati pattern specifici per la gestione dello stato.

Flux architecture

Come alternativa al più classico *Model-View-Controller* (MVC) [15], Facebook propone per le applicazioni React un nuovo pattern architetturale denominato Flux [36]. Esso sottolinea la natura unidirezionale del flusso di dati in un'applicazione React (di qui il nome) e può essere di fatto considerato una variante del *pattern Observer* [7] applicato all'architettura del sistema.

Come è possibile vedere in Figura 5.3, la struttura è composta da quattro entità principali:

Store Rappresenta il "contenitore" che incapsula lo *stato* (del dominio applicativo e/o dell'interfaccia grafica). Ciascuno store agisce come *single source of truth* (SSOT) e non permette di modificare direttamente i valori dello stato, ma solo tramite *azioni* passate con un *dispatcher*.

Dispatcher Singolo oggetto che invia in broadcast come eventi le azioni agli store; gli store devono essere registrati per gli eventi che sono in grado di gestire all'avvio dell'applicazione.

View Rappresenta la componente di interazione con l'utente; essa osserva gli store per aggiornarsi al variare dello stato e genera azioni sulla base delle richieste dell'utente. Sono possibili due tipi di view:

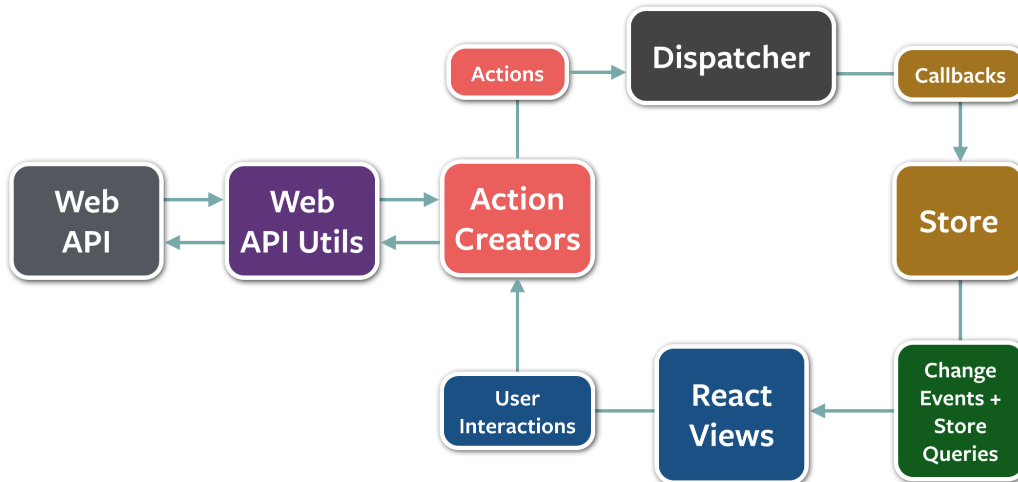


Figura 5.3.: Rappresentazione del flusso unidirezionale dei dati in un'applicazione React con Flux. Figura ripresa dalla documentazione ufficiale.

Presentation view Non si collega né al dispatcher, né agli store, bensì comunica tramite *proprietà* definite alla costruzione.

Container view Si collega al dispatcher e/o agli store, reagendo agli eventi e/o generandoli.

Action Oggetto semplice e immutabile che contiene tutte le informazioni necessarie per modellare un'interazione con lo stato. Possono essere generate dalla view o da API web esterne, ma sempre attraverso un *action creator*.

Il flusso informativo avviene dunque in una singola direzione e tramite *callback*. Questo garantisce le seguenti proprietà:

Asincronismo Il motore che interpreta JavaScript gestisce ciascuna callback eseguendola al ciclo successivo, non bloccando l'esecuzione attuale.

Consistenza di rappresentazione Essendo ciascuno store centralizzato per l'intera applicazione, lo stato rappresentato dovrebbe essere univoco indipendentemente dalla pagina attuale.

Disaccoppiamento Essendo lo store esterno ai componenti grafici, essi risultano maggiormente disaccoppiati rispetto al dominio e più riusabili.

Determinismo Essendo lo stato aggiornabile solo tramite azioni, non si hanno effetti secondari e l'applicazione risulta più facile da rappresentare come una sequenza finita di stati, agevolando anche la fase di testing.

Redux pattern

Il pattern Redux è una delle più popolari varianti del pattern Flux.

Redux[2] è il nome della libreria JavaScript per la gestione dello stato che per prima ne ha definito un'implementazione. Creata da Dan Abramov e Andrew Clark nel 2015 come implementazione alternativa a quella ufficiale del pattern Flux, essa combina idee provenienti dal pattern *Command* [7] e dall'*architettura Elm* teorizzata con l'omonimo linguaggio [30]: lo stato dell'applicazione è descritto da un singolo POJO (*Plain Old JavaScript Object*) all'interno dello store e gli aggiornamenti avvengono tramite *reducer*, funzioni pure di un'azione e dello stato corrente che generano un nuovo stato.

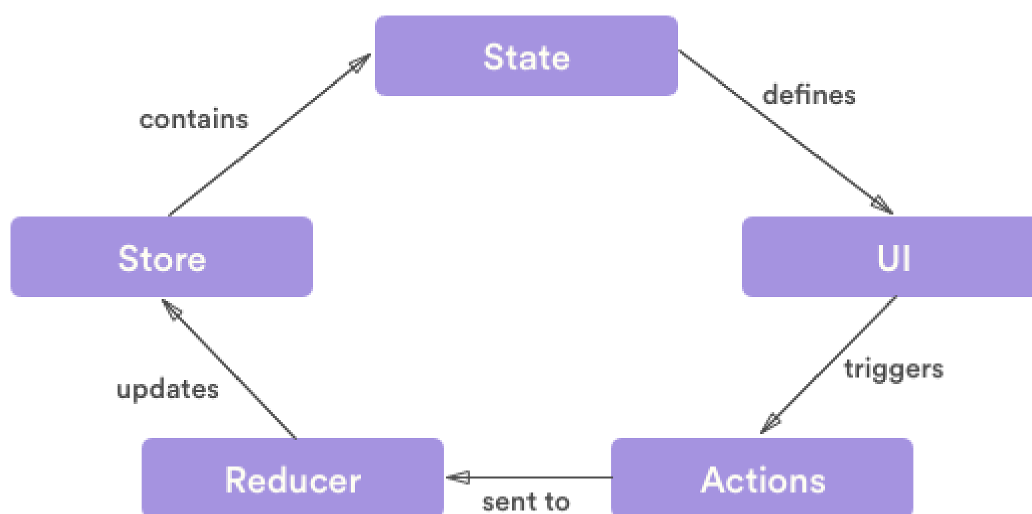


Figura 5.4.: Rappresentazione del flusso unidirezionale dei dati in Redux. Figura ripresa dalla documentazione ufficiale.

Secondo la documentazione ufficiale, la completa centralizzazione dello stato permette al sistema di essere completamente predicibile, permettendo il "*time-travel debug*" tramite strumentazione integrata nel browser. Inoltre, il non essere strettamente legato a un framework di rappresentazione e la possibilità di caricare middleware lo rendono estremamente flessibile. Nel caso di questo progetto, tale soluzione è stata ritenuta ottimale per il tipo di architettura che si intende realizzare.

Prendendo dunque in considerazione il mockup delineato alla Sezione 5.1.1, lo store sarebbe costituito dalle seguenti parti (dette *slice*):

Editor In `editorSlice` saranno inserite tutte le informazioni relative allo stato dei file. In particolare, è possibile definire al suo interno una struttura ad albero riguardante i file e lo stato di apertura degli stessi.

[2] `http://redux.js.org`

Esecuzione In `execSlice` saranno invece inseriti i dati relativi all'esecuzione; saranno dunque presenti lo stato della connessione, l'ID e lo stato della simulazione e i dati dei nodi da rappresentare.

5.3. Architettura del server

Il server costituisce l'entità del sistema che si occupa dell'esecuzione del codice Protelis; è un esecutore remoto a tutti gli effetti.

Innanzitutto, è stato necessario chiarire se l'architettura dovesse essere monolitica o separata in microservizi. In tempi recenti, l'approccio a microservizi viene preferito a causa di diversi vantaggi: in primo luogo, un sistema composto da microservizi indipendenti è solitamente più semplice da scalare, in quanto è possibile replicare ciascun servizio in modo indipendente dagli altri, a seconda delle esigenze. Inoltre, l'approccio a microservizi risulta generalmente più semplice da manutenere, in quanto disaccoppia i servizi tra loro, rendendo chiare le dipendenze condivise e permettendo lo sviluppo indipendente delle componenti. Infine, un sistema di questo tipo offre un'integrazione migliore con orchestratori cloud e permette di impiegare tecnologie di *continuous deployment* (CD) in modo più efficiente.

Tali vantaggi sono però maggiormente evidenti quante più sono le funzionalità che devono essere offerte. Nel caso di questo progetto, di contro, il sistema deve essere in grado di gestire in modo efficiente un solo tipo di servizio, ossia l'esecuzione di codice su una rete simulata.

In questo caso, dunque, è stato ritenuto più adeguato scegliere un'architettura monolitica, favorendo la semplicità di progettazione e delegando la gestione dello scaling al livello di piattaforma di deploy.

5.3.1. Pattern reactor

Il giusto livello di reattività ed efficienza è stato trovato nell'approccio *event-driven* con *event-loop*. Tramite questo modello di concorrenza, denominato *pattern Reactor* [8], il server gestisce le richieste dei client attraverso una coda: uno o più cicli si occupano di gestire gli eventi nella coda in modo sincrono. In particolare, si è deciso di adottare il modello *multi-reactor* fornito da Vert.x.

Vert.x è un framework applicativo event-driven che esegue su JVM (nonostante offra un supporto poliglotta a diversi linguaggi). Del modello architetturale messo a disposizione dal framework, è stato considerato interessante il concetto di *Verticle*: esso è un'astrazione, simile al pattern ad attori [1] ma non considerato pienamente aderente al modello teorico dalla stessa documentazione ufficiale[3], che incapsula un event-loop insieme al suo stato e interagisce tramite gli eventi provenienti da un EventBus.

[3] `https://vertx.io/docs/vertx-core/kotlin/#_verticles`

Per questo progetto, il modello è stato considerato adatto, in quanto in grado di garantire il giusto livello di astrazione e i criteri di reattività richiesti.

5.3.2. Progettazione dei verticle

Il sistema progettato è composto da due componenti principali, che possono essere facilmente mappati su altrettante entità verticle secondo il lessico di Vert.x.

Il primo, chiamato `BridgeVerticle`, è dedicato alla gestione delle API per la comunicazione da e verso l'esterno. In particolare, esso implementa il pattern *bridge* relativamente alle connessioni verso l'esterno, trasformando le chiamate HTTP eseguite dai client in eventi espliciti dell'EventBus. Gestendo le comunicazioni con l'esterno, esso astrae l'intero processo di gestione del protocollo di comunicazione dalle altre componenti dell'applicazione.

Il secondo è invece chiamato `AlchemistVerticle` e costituisce l'entità che si interfaccia con un motore di esecuzione esterno, a cui ci si è riferiti già nell'architettura logica delineata al termine della fase di analisi del problema (Figura 4.2). In particolare, per eseguire il codice si è scelto di utilizzare il simulatore Alchemist, che verrà analizzato più nel dettaglio nella Sezione 5.3.3.

Oltre a questi, è stato anche progettato l'uso di un verticle principale `MainVerticle`, che viene lanciato dall'avviatore di Vert.x e che coordina l'avvio dei due verticle descritti sopra.

5.3.3. Simulatore scelto: Alchemist

Alchemist [31] è un meta-simulatore estendibile completamente *open-source* che esegue su *Java Virtual Machine*, nato all'interno dell'Università di Bologna.

Simulazione

In generale, una *simulazione* [5] è una riproduzione del modo di operare di un sistema o un processo del mondo reale nel tempo. L'imitazione del processo del mondo reale è detta *modello*; esso risulta essere una riproduzione più o meno semplificata del mondo reale, che viene aggiornata ad ogni passo di esecuzione della simulazione.

Alchemist rientra nell'archetipo dei simulatori ad eventi discreti (DES) [23, 4]: gli eventi sono strettamente ordinati e vengono eseguiti uno alla volta, determinando il passare del tempo. L'idea dietro al progetto è quello di riuscire ad avere un framework di simulazione il più possibile generico, in grado di simulare sistemi di tipologia e complessità diverse, mantenendo le prestazioni dei simulatori non generici (come ad esempio quelli impiegati in ambito chimico [2]).

Per perseguire questo obiettivo, la progettazione dell'algoritmo è partita dallo studio del lavoro di Gillespie del 1977 [3] e di altri scienziati nell'ambito della simulazione chimica. Nonostante siano presenti algoritmi in grado di eseguire un numero di reazioni addirittura in tempo costante, la scelta dell'algoritmo è infine ricaduta su una versione migliorata dell'algoritmo SSA di Gillespie, il Next Reaction Method [11] di Gibson e Bruck: ad ogni passo di simulazione,

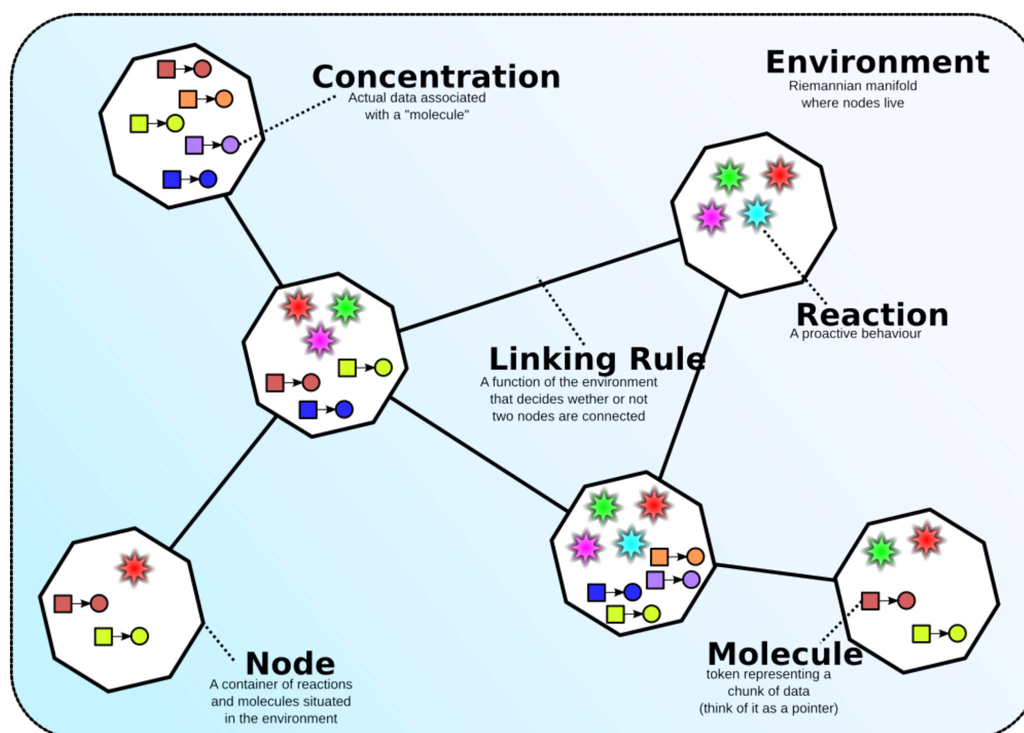


Figura 5.5.: Rappresentazione grafica delle diverse entità di Alchemist.
Figura ripresa dal sito ufficiale[4].

esso è in grado di selezionare la reazione successiva in tempo costante e richiede un tempo logaritmico per aggiornare le strutture dati interne al termine dell'esecuzione dell'evento.

Astrazioni e modello

Il modello di astrazione di Alchemist è ispirato dal lavoro della comunità scientifica nell'ambito dei simulatori a scopo di ricerca chimica e ne riprende dunque la nomenclatura. Le entità (visibili in Figura 5.5) su cui lavora sono le seguenti:

Molecola Una *molecola* rappresenta il nome assegnato ad un particolare dato all'interno di un *nodo*, del quale ne astrae parte dello stato.

Un parallelismo con la programmazione imperativa vedrebbe la *molecola* come un'astrazione del nome di una variabile.

Concentrazione La *concentrazione* di una *molecola* è il valore associato alla proprietà rappresentata dalla *molecola*.

Mantenendo il parallelismo con la programmazione imperativa, la *concentrazione* rappresenterebbe il valore della variabile.

Nodo Il *nodo* è un contenitore di *molecole* e *reazioni* che risiede all'interno di un *ambiente* e che astrae una singola entità.

[4]http://alchemistsimulator.github.io

Ambiente L'*ambiente* è l'astrazione che rappresenta lo spazio nella simulazione ed è l'entità che contiene i *nodi*.

Esso è in grado di fornire informazioni in merito alla posizione dei *nodi* nello spazio, alla distanza tra loro e al loro vicinato; opzionalmente, l'*ambiente* può offrire il supporto allo spostamento dei *nodi*.

Regola di collegamento La *regola di collegamento* è una funzione dello stato dell'*ambiente* che associa ad ogni *nodo* un *vicinato*.

Vicinato Un *vicinato* è un'entità costituita da un *nodo* detto "centro" e da un insieme di altri *nodi* (i "vicini").

L'astrazione dovrebbe avere un'accezione il più possibile generale e flessibile, in modo da poter modellare qualsiasi tipo di legame di vicinato, non solo spaziale.

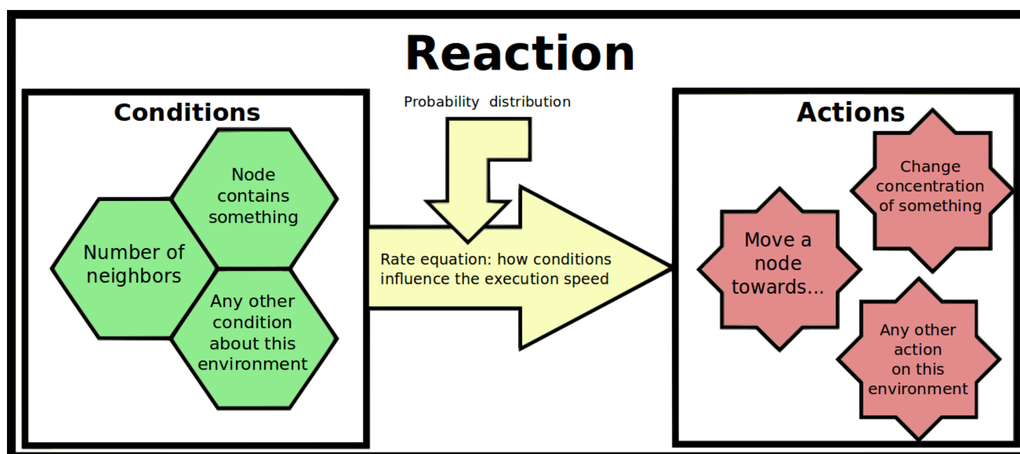


Figura 5.6.: Rappresentazione grafica della *Reazione*.

Reazione Il concetto di *reazione* è da considerarsi molto più elaborato di quello utilizzato in chimica: in questo caso, si può considerare come un insieme di *condizioni* sullo stato del sistema, che qualora dovessero risultare vere innescherebbero l'esecuzione di un insieme di *azioni*.

Una *reazione* (di cui si ha una rappresentazione grafica in Figura 5.6) è dunque un qualsiasi evento che può cambiare lo stato dell'*ambiente* e si compone di un insieme di *condizioni*, una o più *azioni* e una distribuzione temporale.

La frequenza di accadimento può dipendere da:

- Un tasso statico;
- Il valore di ciascuna *condizione*;
- Una equazione che combina il tasso statico e il valore delle *condizioni*, restituendo un "tasso istantaneo";
- Una distribuzione temporale.

Ogni *nodo* è costituito da un insieme (anche vuoto) di *reazioni*.

Condizione Una *condizione* è una funzione che associa un valore numerico e un valore booleano allo stato corrente di un *ambiente*.

Azione Un'*azione* è una procedura che provoca una modifica allo stato dell'*ambiente*.

Per quanto la terminologia sia ripresa dalla chimica, il meta-modello del simulatore è estendibile, adottando interpretazioni più o meno lasche dei termini "molecola" e "concentrazione". In particolare, in Alchemist esiste il concetto di *incarnazione*, che definisce l'istanza concreta del meta-modello, delineando le modalità con le quali le astrazioni vengono implementate.

Incarnazione Protelis

Alchemist fornisce l'implementazione di diverse incarnazioni; per lo scopo di questa tesi, ci si propone di utilizzare l'incarnazione Protelis. In essa, la molecola identifica il nome di un sensore, mentre la sua concentrazione è il valore misurato.

Attraverso la configurazione di Alchemist, è possibile definire il posizionamento dei nodi e le modalità di collegamento, nonché la presenza di specifiche molecole. In questo modo, è possibile definire una molecola che conterrà il codice Protelis che ciascun nodo deve eseguire; il sistema può così caricare dinamicamente il codice ottenendo la relativa concentrazione. Un esempio di configurazione è riportato in Appendice B.

5.4. Interazioni

Una volta analizzato il comportamento delle due entità in gioco, è necessario delineare anche la loro interazione remota.

5.4.1. Scelta del modello di comunicazione e del protocollo

Come detto nella Sezione 5.3.1, per la progettazione del server di backend si è scelto di utilizzare un modello a event-loop multipli comunicanti tramite EventBus; inoltre, anche il pattern di gestione dello stato scelto per il funzionamento del client (Sezione 5.2.2) è event-driven. È risultato dunque naturale strutturare anche la comunicazione tra client e server utilizzando un modello a eventi.

In particolare, l'EventBus di Vert.x supporta l'utilizzo di *bridge* per la comunicazione remota attraverso numerosi protocolli. Tra questi, *SockJS* è un protocollo pensato per realizzare una comunicazione *WebSocket-like* sul maggior numero di piattaforme possibili. Esso gestisce in modo autonomo la verifica del supporto del protocollo WebSocket [27] da parte del client e del server, migrando verso una soluzione a *polling* tramite HTTP standard in caso assenza. Tramite un protocollo di questo tipo, è possibile realizzare un canale di comunicazione bidirezionale

veloce, adatto per il trasferimento di un elevato numero di eventi come nel caso di questo progetto di tesi.

Dunque, il `BridgeVerticle` esporrà tramite API (al percorso "`/eventbus`" relativamente all'*host* principale) l'accesso all'EventBus per i messaggi previsti. L'applicazione web si dovrà connettere attraverso un client generando azioni sullo store interno.

5.4.2. Comportamento

Una volta chiarite le modalità di trasferimento delle informazioni, viene progettato il comportamento che permette al sistema di essere reattivo. In Figura 5.8 viene rappresentata la sequenza di operazioni svolte dal server sulla base degli eventi inoltrati sull'EventBus di Vert.x; in Figura 5.7, invece, il diagramma UML riassume la sequenza di azioni che permutano lo stato di Redux con il procedere dell'esecuzione. Di seguito, invece, sono riassunti i passaggi nel loro complesso.

1. Il primo passo consiste nell'instaurare la connessione. La pressione di un bottone genera un'azione sullo store che abilita la connessione SockJS verso il backend. L'utente viene notificato del risultato dell'operazione attraverso la generazione, da parte del sistema, di azioni con le relative permutazioni dello stato.

2. Una volta che il codice Protelis è pronto per essere lanciato, l'utente utilizzerà il bottone dedicato per eseguirlo. Questo causa la creazione di un'azione di richiesta di upload del codice; esso è già presente nello store, dal quale viene prelevato e inoltrato tramite socket al server.

3. Il verticle riceve l'evento tramite EventBus; procede dunque costruendo un simulatore, al quale viene assegnato un identificativo univoco e un componente osservatore. Dopodiché, tramite EventBus viene inviato l'identificativo.

4. Il client riceve questo ID tramite socket e un'azione di Redux viene generata. La vista viene aggiornata di conseguenza, informando l'utente dell'avvenuta configurazione e dell'imminente avvio dell'esecuzione.

5. Quando il motore di simulazione esegue uno step, il componente osservatore viene notificato. Viene dunque generato un evento sul bus degli eventi diretto verso l'esterno, su un indirizzo legato all'identificativo iniziale e alla tipologia di variazione avvenuta nella simulazione (step iniziale, *round* d'esecuzione, terminazione).

6. Quando l'esecuzione viene terminata sul simulatore, il client viene notificato dell'evento nello stesso modo con cui ha ricevuto i vari aggiornamenti. Sul server, l'ID verrà classificato come terminato e il simulatore potrà essere riutilizzato o distrutto a seconda delle necessità.

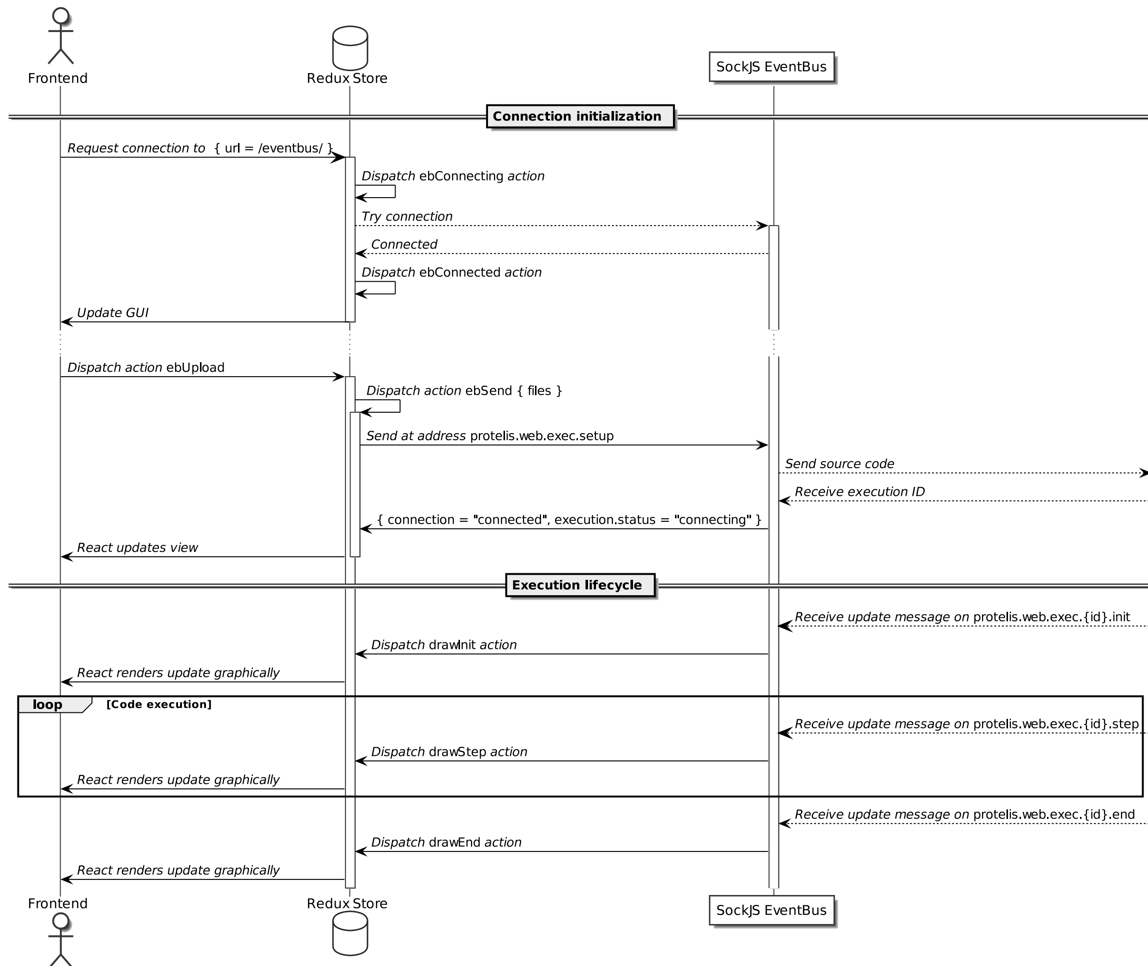


Figura 5.7.: Diagramma UML di sequenza rappresentante il flusso delle azioni sullo store di Redux.

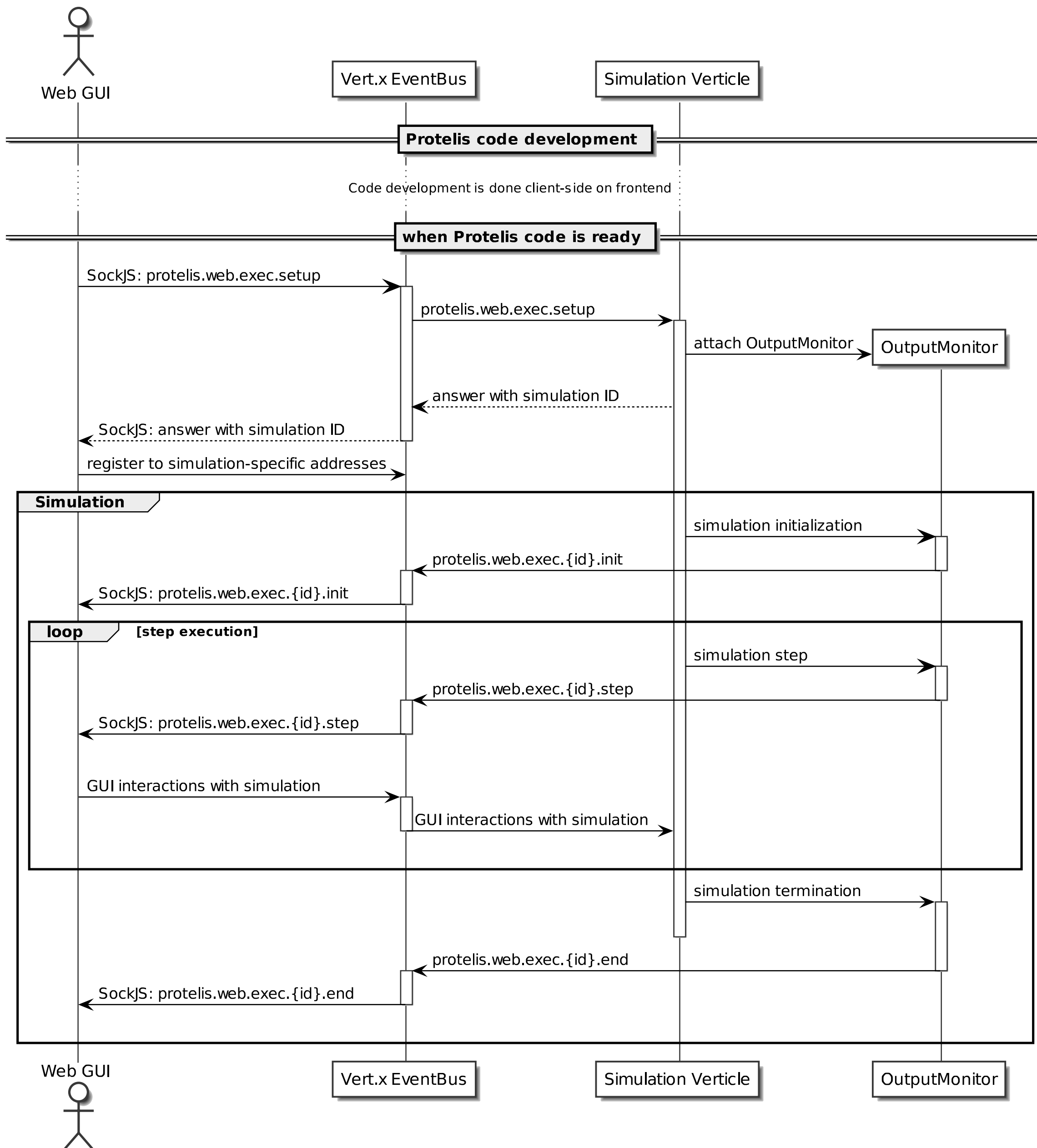


Figura 5.8.: Diagramma UML di sequenza che rappresenta la gestione degli eventi sul bus di Vert.x.

6. Implementazione

In questo Capitolo sono esposti tutti i dettagli legati al processo di sviluppo e all'implementazione. In particolare, verranno motivate le scelte tecnologiche che in fase di progettazione non erano state considerate rilevanti. Inoltre, verranno analizzati alcuni dettagli implementativi ritenuti importanti alla descrizione del funzionamento del progetto.

6.1. Tecnologie utilizzate

6.1.1. Linguaggi di programmazione

Il sistema è stato realizzato in due differenti linguaggi. Di seguito sono espresse le motivazioni della scelta di ciascuno di essi.

Linguaggio per il backend: Kotlin

Per la realizzazione del backend, come è stato specificato nel Capitolo 4, era necessario l'impiego di un linguaggio in grado di eseguire su piattaforma JVM. Il framework scelto, Vert.x, è presentato come poliglotta (supporta Java, JavaScript, Groovy, Ruby, Ceylon, Scala e Kotlin), non vincolando dunque la scelta.

Si è deciso di procedere con Kotlin per le seguenti motivazioni:

Brevità ed espressività Kotlin si presenta come molto meno verboso di Java, supportando un maggior numero di costrutti che permettono di ridurre il codice "*boilerplate*".

Robustezza Una delle funzionalità che è stata pubblicizzata maggiormente alla presentazione di Kotlin è la *null-safety*. Kotlin permette di evitare le comuni eccezioni di tipo `NullPointerException` gestendo il valore `null` in modo maggiormente chiaro tramite notazione opzionale e controllo da parte del compilatore.

Interoperabilità Kotlin nasce per essere affiancato a Java e sostituirlo gradualmente. Risulta dunque intercompatibile con quasi qualsiasi piattaforma o libreria sia pensata per Java, non limitando eventualmente l'affiancamento futuro ad un altro linguaggio.

Inoltre, il compilatore di Kotlin supporta diversi altri target, come browser, piattaforme mobile e binario nativo X86 e ARM, lasciando dunque aperte ulteriori possibilità di espansione future.

Approccio funzionale Kotlin è principalmente un linguaggio orientato agli oggetti, ma supporta molti costrutti tipici della programmazione funzionale, come espressioni *lambda*, *function types* e le funzioni di ordine superiore.

Pragmatismo Kotlin presenta, secondo la documentazione, l'intento di essere "pragmatico", ossia focalizzato sull'esperienza d'uso dello sviluppatore. Esso offre una buona integrazione con gli strumenti esistenti, sia per la costruzione (come Gradle e Maven) che per lo sviluppo (come Eclipse, Visual Studio e gli IDE di JetBrains).

Vert.x offre inoltre una estensione della propria libreria *core* per sfruttare al meglio la sintassi di Kotlin.

Linguaggio per il frontend: TypeScript

Per la realizzazione del frontend era necessario l'impiego di un linguaggio in grado di eseguire su browser, quindi che fornisse come target di compilazione JavaScript o WebAssembly. A differenza del backend, il framework scelto, React, vincolava la scelta a un numero limitato di linguaggi in grado di generare codice JS. In particolare, sono stati presi in considerazione i quattro linguaggi introdotti nella Sezione 2.2, ma si è preferito impiegare TypeScript per i seguenti motivi:

Supporto ufficiale Per quanto React venga presentato anche nella documentazione ufficiale come non strettamente dipendente dal linguaggio, il supporto disponibile per ciascuno di questi può essere differente. In particolare, Facebook, dichiara il supporto diretto solo per JavaScript e TypeScript; linguaggi come Kotlin, Scala, Reason e F# sono dichiaratamente compatibili, ma la loro integrazione non è gestita direttamente dalla società.

Kotlin (attraverso JetBrains) e Scala (attraverso il progetto Scala.js) riescono ad offrire un supporto comunitario sufficiente, ma comunque non comparabile a quello offerto da Facebook e Microsoft.

Type Checking JavaScript supporta unicamente la tipizzazione dinamica. Se questo aspetto da un lato aggiunge flessibilità al linguaggio, dall'altro aumenta la possibilità di bug e situazioni non previste. TypeScript offre un sistema di tipi completo e flessibile, che permette di definire una struttura chiara e flessibile, rendendo il processo di sviluppo più sicuro.

Facebook supporta ufficialmente anche Flow e la libreria `prop-types` come alternativa all'utilizzo di un diverso linguaggio. Flow non offre però la medesima espressività di TypeScript e non può vantare un supporto altrettanto vasto da parte della comunità per la tipizzazione delle librerie JavaScript. La libreria `prop-types` invece offre un controllo dei tipi unicamente a tempo di esecuzione, richiedendo la definizione del tipo atteso manualmente. Questo non permette di adottare un approccio "*fail-fast*" come quello vantato da TypeScript.

Integrazione con gli IDE La possibilità di avere informazione sui tipi a livello di compilatore permettono a IDE come Visual Studio Code e WebStorm di essere molto più precisi ad evidenziare codice errato rispetto all'approccio basato su commenti di JavaScript.

Estensione della sintassi JSX e TSX Una delle maggiori particolarità del framework React è la sintassi *JSX*. L'acronimo sta per J*avaScript* e*X*tension e identifica un particolare zucchero sintattico non presente in JavaScript "standard". Esso permette la definizione di "`element`", ossia di componenti direttamente rappresentabili, attraverso una sintassi dichiarativa di markup che ricorda HTML.

Essendo in React la logica rappresentativa strettamente legata agli elementi non grafici del componente, essere in grado di applicare una corretta "*separation of concerns*" all'interno dello stesso componente e file senza doversi affidare a markup esterno è un vantaggio notevole.

TypeScript supporta questa estensione della sintassi, denominandola *TSX*.

L'unica alternativa ritenuta valida al posto di TypeScript era Kotlin. L'impiego di Kotlin per tutta l'applicazione sarebbe stato infatti molto interessante per utilizzare un unico linguaggio in tutto il sistema, migliorando l'integrazione e il supporto a dipendenze condivise.

Purtroppo, per quanto JetBrains supporti ufficialmente React per Kotlin/JS, l'integrazione all'inizio dei lavori di questa tesi era ancora troppo instabile per un uso reale. Si sono riscontrati problemi[1] con lo strumento di generazione dei *wrapper* per i tipi `dukat` e l'output del compilatore genera codice JS datato e pesante. Fintanto che la versione 1.4 di Kotlin non sarà rilasciata in pianta stabile, si ritiene che il linguaggio non sia ancora pronto per un uso frontend su browser.

6.1.2. Strumenti per lo sviluppo e il controllo del software

Per gestire le dipendenze in sistemi moderni, l'uso di applicazioni per la *build-automation* dotati di risoluzione delle dipendenze è l'approccio più comune.

L'utilizzo di strumenti che controllino la qualità del codice e diano la possibilità di testarlo in modo immediato è fondamentale per la realizzazione di un sistema complesso. Essi permettono infatti di revisionare il codice in modo sistematico, così da evitare errori che a volte possono verificarsi, senza bisogno che il programma venga realmente eseguito: analizzano il codice sorgente per individuare potenziali bug e spesso indicano possibili miglioramenti e ottimizzazioni.

Essendo le due componenti basate su ecosistemi completamente diversi, verranno trattate separatamente.

[1] `https://github.com/Kotlin/dukat/issues/120#issuecomment-560423099`

Backend

Il progetto di backend utilizza Gradle per la risoluzione delle dipendenze, la costruzione del software e la verifica della qualità. *Gradle* è un sistema per l'automazione dello sviluppo, nato per includere tutte le caratteristiche provenienti da Apache Ant, Maven e Ivy attraverso la definizione di *buildscript* originariamente in Groovy e più recentemente anche in Kotlin. Pensato per i linguaggi che compilano per JVM, questo sistema permette di scaricare le dipendenze da diversi repository Maven durante la fase di compilazione. Nello script di costruzione sono stati utilizzati i seguenti plugin:

Gradle Shadow Il plugin, sviluppato da John Engelman, permette la generazione di "shadow jar", ossia pacchetti jar contenenti anche tutte le dipendenze oltre al codice compilato.

Vert.x Gradle Plugin Il plugin, sviluppato da Julien Ponge e adottato quasi ufficialmente dai manutentori del progetto Vert.x, fornisce una configurazione del progetto ottimizzata per progetti Vert.x. Rende possibile il lancio dei verticle in modo indipendente e con la ricompilazione automatica, molto comodo in sede di sviluppo e debug.

Kotlin Il plugin, realizzato in modo ufficiale da JetBrains, configura Gradle per la compilazione di codice Kotlin e la configurazione delle opzioni per la generazione del bytecode.

Per motivi di compatibilità con il simulatore Alchemist, è stata scelta come versione target del bytecode Java 11.

Ktlint Gradle Il plugin, realizzato da Jonathan Leitschuh, è un wrapper dello strumento di analisi statica di codice Kotlin *ktlint*, realizzato da Pinterest. Si è scelto di utilizzare ktlint anziché *detekt*, altro strumento simile, in quanto non richiede una configurazione personalizzata, bensì impone un insieme standard di regole, comunemente studiate e accettate dalla community.

RefreshVersions Il plugin, sviluppato da Jean-Michel Fayard, permette una migliore gestione delle dipendenze automatizzandone gli aggiornamenti.

JUnit *JUnit* è uno dei più noti framework di unit testing per Java e linguaggi derivati. Il plugin, integrato ufficialmente in Gradle, permette l'esecuzione di test automatizzati con tale framework.

In questo progetto è stata utilizzata la versione 5 di JUnit, utilizzando il motore di esecuzione *jupiter*.

JaCoCo *JaCoCo* è uno strumento per la misura della copertura del codice da parte dei test realizzati. Si integra con JUnit ed è pensato per Java e linguaggi derivati. Il plugin, integrato ufficialmente in Gradle, offre una dettagliata configurazione e permette la generazione di report in diversi formati.

Per lo sviluppo del codice è stato utilizzato l'ambiente di sviluppo integrato *JetBrains Intellij IDEA* in versione *Ultimate 2019.3.3*.

Frontend

Il progetto frontend è un modulo Node.js generato tramite lo strumento ufficiale `create-react-app` (spesso abbreviato in *CRA*) fornito da Facebook. Si seguito vengono riportate tutte le tecnologie utilizzate durante lo sviluppo:

Yarn Come *package manager* per la gestione delle dipendenze è stato scelto *Yarn* anziché NPM. Esso, oltre a essere la scelta consigliata da React, presenta diversi vantaggi rispetto alla controparte. Ad esempio, la cartella delle dipendenze risulta generalmente più compatta in quanto utilizza collegamenti simbolici per le dipendenze condivise dai moduli JS risolti. Inoltre, la velocità di risoluzione delle dipendenze è generalmente più elevata, in quanto sfrutta una strategia di caching più efficiente.

React Scripts Lo strumento CRA astrae la complessità di configurazione di un progetto attraverso l'utilizzo di un pacchetto denominato `react-scripts`. Essi incapsulano strumenti quali WebPack, Babel, ESLint e Jest, mettendo a disposizione semplici script che si appoggiano alle configurazioni ufficiali di riferimento per la compilazione, l'esecuzione e la verifica del codice realizzato.

ESLint & stile Airbnb *ESLint* è uno strumento estendibile per l'analisi statica del codice, che supporta nativamente JavaScript e TypeScript. Il pacchetto `react-scripts` fornisce una configurazione minimale, ma è consigliato di adottare uno stile (insieme di regole) tra quelli più famosi. In questo caso è stato scelto quello fornito da Airbnb.

Jest *Jest* è un framework di test per JavaScript sviluppato da Facebook che fornisce tutti gli strumenti per la scrittura e l'esecuzione dei test e per la raccolta della loro copertura sul codice.

Per lo sviluppo del codice è stato utilizzato l'ambiente di sviluppo integrato *JetBrains WebStorm* in versione *2019.3.3*.

6.1.3. Controllo di versione e CI/CD

Il controllo di versione utilizzato per questo progetto è affidato al DVCS (D*istributed* V*ersion* C*ontrol* S*ystem*) Git, utilizzato con flusso di lavoro di tipo *Git flow*.

Il codice è disponibile su GitHub nei repository Protelis-Web[2] e protelis-web-frontend[3].

Attraverso GitHub sono allacciate ai repository diverse soluzioni di *continuous integration* (CI):

[2] `https://github.com/NiccoMlt/Protelis-Web`
[3] `https://github.com/NiccoMlt/protelis-web-frontend`

Travis CI *Travis CI* è un sistema di integrazione continua distribuito, utilizzato per la compilazione e il test di progetti caricati su repository GitHub. Ad ogni operazione di push sul repository, il codice viene testato su tutte le maggiori piattaforme (Linux, Windows, MacOS) e le differenti versioni degli SDK (NodeJS LTS, AdoptOpenJDK con OpenJDK HotSpot e AdoptOpenJDK con Eclipse OpenJ9). Viene inoltre fatto un controllo sulla qualità del codice.

Codecov Tramite *Codecov* vengono raccolti i report di copertura del codice generati da JaCoCo e Jest, permettendo la visualizzazione online di grafici di dettaglio. Al termine di ogni esecuzione di una *pipeline* in Travis CI, i report sulla piattaforma di riferimento vengono pubblicati qui.

Codacy *Codacy* è uno strumento online per l'automazione della *code review* che aggrega diversi strumenti di analisi del codice per numerosi linguaggi, generando report dettagliati. Ad ogni operazione di push sul repository, il codice viene verificato.

6.1.4. Deployment

Vert.x non si basa su *servlet* e può essere eseguito su qualsiasi piattaforma permetta il lancio di un jar, senza la necessità di un hosting specifico per Java EE.

Per motivi di test, il server è in esecuzione su due differenti piattaforme:

- Al termine della pipeline di Travis CI per il branch principale, l'eseguibile viene automaticamente caricato e lanciato su Heroku. *Heroku* è un servizio online di tipo PaaS (P*latform as a* S*ervice*) di proprietà di SalesForce e compatibile con diversi linguaggi. Informazioni sulla VM utilizzata non sono disponibili.
- Per performance migliori, viene utilizzato anche un server fornito dall'Università di Bologna. Su di esso viene effettuato il deploy tramite SSH avvalendosi del `docker-engine` in esecuzione all'interno di Linux. *Docker* è uno strumento per la virtualizzazione a livello di sistema operativo (*container*) molto utilizzato per la pacchettizzazione delle applicazioni. Il Dockerfile per la costruzione dell'immagine utilizzata, basato su AdoptOpenJDK con Eclipse OpenJ9, è allegato a questo documento in Appendice A.

Il client React è compilato come un bundle JavaScript importato da una pagina HTML. Esso non viene servito dal server Vert.x, bensì attraverso la piattaforma di hosting statico *ZEIT Now*, che ne esegue il *continuous deployment* direttamente da GitHub.

6.2. Dettagli implementativi: Frontend

Per l'implementazione dei componenti di base del frontend, si è fatto largo uso della libreria Material-UI. Essa mette a disposizione una grande quantità di componenti React di base (come

elementi per il layout e elementi di controllo) aderenti al design Material scelto in fase di progettazione. Inoltre, tramite la libreria è stato possibile definire un tema specifico per il progetto Protelis on Web nelle due varianti chiara e scura, caricate a seconda delle preferenze del browser dell'utente.

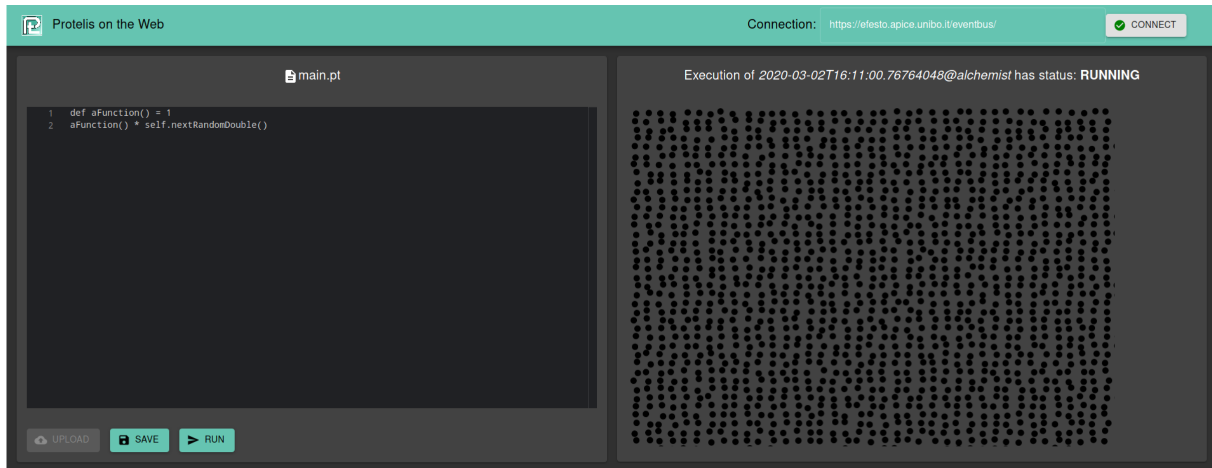


Figura 6.1.: Screenshot della schermata intera durante l'esecuzione

La struttura del frontend, visibile in Figura 6.1, è composta da tre elementi principali:

- la `AppBar` superiore, nella quale è posizionato il selettore con il quale è possibile indicare quale server di backend utilizzare.
- la `Card` di sinistra ospita tutti i controlli principali. In essa è presente un editor realizzato con l'ausilio della libreria Monaco Editor, realizzata da Microsoft e utilizzata, ad esempio, per Visual Studio Code; Essa supporta buona parte dei plugin disponibili per tale IDE e lascia spazio, in futuro, all'integrazione di *language server* dedicati all'autocompletamento.

  Sotto l'editor sono presenti i bottoni dedicati al salvataggio del codice e all'avvio dell'esecuzione.
- la `Card` di destra ospita invece principalmente un canvas, realizzato tramite la libreria Konva. Essa permette di realizzare canvas performanti modellando ciascuna operazione di disegno come componenti React.

  È risultato dunque semplice delegare le operazioni di disegno al motore di React, attraverso un'operazione di binding dei componenti di Konva con lo stato interno del sistema.

La gestione dello stato, come deciso in fase di progettazione, è delegata alla libreria Redux; in particolare, si è scelto di utilizzare lo strumento ufficiale `redux-toolkit`, che permette una configurazione dello store ottimale riducendo il codice *boilerplate*.

Per memorizzare le informazioni sullo stato del sistema, le due *slice* illustrate alla Sezione 5.2.2 sono state sufficienti.

Per quanto riguarda la gestione del protocollo di comunicazione, invece, è stato necessario realizzare un *middleware* che incapsulasse la comunicazione.

In Redux, un *middleware* è una funzione che si frappone tra il *dispatcher* e il *reducer* e permette l'elaborazione dell'azione prima che essa venga gestita. Attraverso un middleware, è possibile definire comportamenti asincroni senza rallentare la gestione delle permutazioni sullo stato: esso può infatti "catturare" azioni provenienti da componenti di controllo nella pagina richiedenti operazioni bloccanti e, una volta che il processo asincrono è terminato, generare un'azione per aggiornare lo stato.

Sono stati sviluppati numerosi middleware per la gestione delle websocket, ma non ne sono stati trovati in grado di supportare SockJS. Si è deciso dunque di definirne uno che nascondesse l'intera procedura di connessione, gestendo le azioni che permettono la connessione, la disconnessione e l'invio di messaggi e aggiornando lo stato quando necessario.

Poiché l'intera logica di connessione è nascosta, in futuro sarà possibile, se necessario, sostituire il protocollo di comunicazione senza che siano necessarie altre modifiche al sistema.

6.3. Dettagli implementativi: Backend

Come progettato nella Sezione 5.3, il server è stato realizzato con due verticle di Vert.x che modellano la logica applicativa e uno dedicato all'avvio del sistema. Tali verticle comunicano tra loro e con l'esterno tramite EventBus.

L'EventBus viene esposto verso l'esterno tramite la libreria ufficiale di Vert.x per la realizzazione di bridge con SockJS. In particolare, la classe `BridgeVerticle` si occupa di costruire un router HTTP e HTTPS sul quale viene montato il bridge.

La gestione della simulazione avviene, come detto, appoggiandosi ad Alchemist (il diagramma UML in Figura 6.2 mostra le principali classi coinvolte).

Come è possibile vedere, Alchemist non viene utilizzato direttamente, bensì incapsulato all'interno di un `SimulatedProtelisEngine`. Tale classe implementa l'interfaccia `ProtelisEngine`, che astrae il concetto di esecutore per codice Protelis; in questo modo, il sistema non è vincolato all'implementazione simulata, né essa è legata strettamente ad Alchemist.

Anche l'astrazione di Alchemist per l'osservazione del motore di esecuzione è stata incapsulata, definendo una classe di *boundary*, `ProtelisOutputMonitor`, che agisce da adattatore.

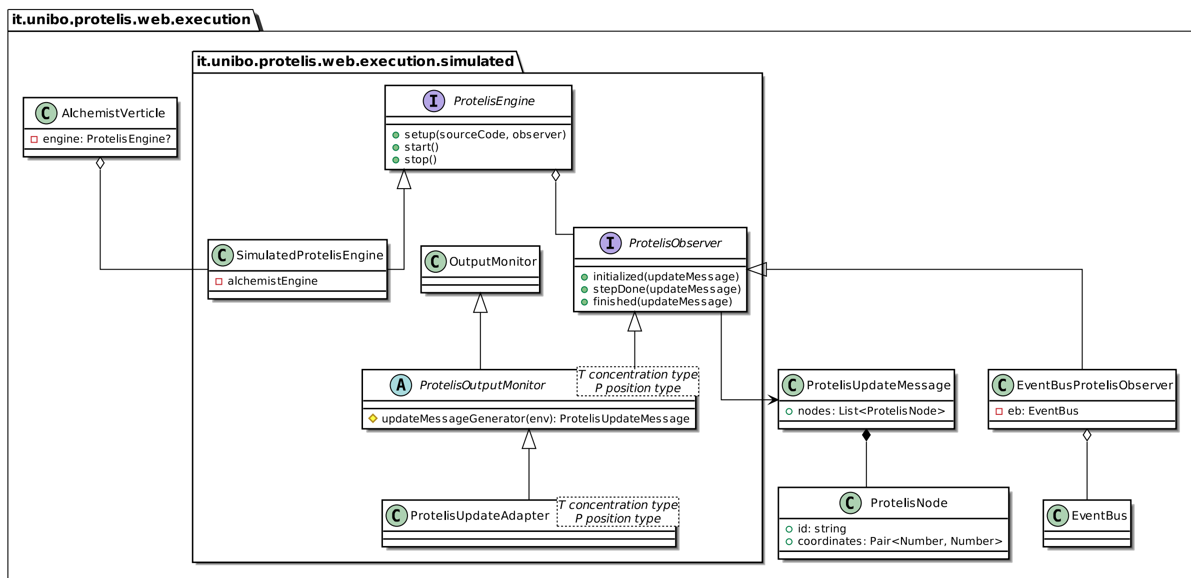


Figura 6.2.: Rappresentazione UML delle classi coinvolte nella simulazione

Parte III.

Conclusioni

7. Valutazione dei risultati

7.1. Considerazioni sul contributo

Il problema delineato nei Capitoli 4 e 5 prevedeva la realizzazione di un sistema per l'esecuzione di codice Protelis attraverso una piattaforma web, appoggiandosi ad una rete di dispositivi simulata. I requisiti funzionali sono stati tutti soddisfatti: la Single-Page Application realizzata permette di visualizzare e modificare il codice Protelis all'interno del campo editor e fornisce una rappresentazione dei risultati attraverso un canvas che disegna l'output. L'utente non deve configurare impostazioni complesse: l'unica configurazione messa a disposizione, oltre ovviamente al codice da eseguire, è la selezione del server di backend. Anche il server soddisfa i propri requisiti funzionali: l'esecuzione del codice è delegato a singole istanze del simulatore Alchemist, le quali generano ciascuna una rete virtuale, in modo da poter servire ciascuno degli utenti del sistema in contemporanea.

Il client soddisfa appieno anche i requisiti non funzionali: l'esperienza utente è immediata, con un'interfaccia orizzontale responsiva strutturata su una singola pagina. La configurazione dei polyfill permette il supporto a tutti i browser con percentuali di utilizzo superiori allo 0,2 % e che non abbiano terminato il supporto ufficiale da più di 24 mesi; lo strumento `browserlist` stima una copertura del 91,67 % delle piattaforme correntemente utilizzate. Anche il protocollo impiegato per la comunicazione con il server, SockJS, risulta compatibile con la maggior parte delle piattaforme in uso, utilizzando però soluzioni di trasporto meno efficienti su browser più datati. Infine, il server soddisfa il requisito di scalabilità, garantito principalmente dalle possibilità fornite dal framework Vert.x.

Il sistema inoltre è stato progettato con diverse possibilità di espansione in mente, principalmente per quanto riguarda il supporto ad altri linguaggi (come ScaFi) e ad altri motori di esecuzione. Si tratterà meglio questo aspetto nel Capitolo 8.

L'unica problematica individuata riguarda l'efficienza: la costruzione di una nuova simulazione Alchemist per ogni utente che richieda un'esecuzione comporta tempi di costruzione e impiego di memoria non banali (soprattutto su piattaforme cloud gratuite come Heroku *Free*). Questo problema è strettamente legato alla necessità dei linguaggi aggregati di eseguire su una rete e può essere solamente aggirato tramite meccanismi di caching o scalando il sistema aumentando le risorse a disposizione.

7.2. Valutazione dell'interfaccia

Per quanto riguarda la valutazione della qualità dell'interfaccia web realizzata, è stata presa in considerazione principalmente l'esperienza finale dal punto di vista dell'utente web. Per avere una misura quantitativa di quanto l'applicazione web si presenti adeguata, si è deciso di utilizzare lo strumento Lighthouse messo a disposizione da Google.

Lighthouse è uno strumento automatizzato open-source, fornito inizialmente con la suite di strumenti *Chrome DevTools*, che permette l'*auditing* di pagine web secondo gli standard premiati dal motore di ricerca di Google. Esso verifica le prestazioni di caricamento e navigazione, l'accessibilità, l'ottimizzazione per i motori di ricerca, e in generale le buone pratiche di programmazione. Supporta inoltre controlli aggiuntivi per le PWA (P*rogressive* W*eb* A*pp*) se abilitati.

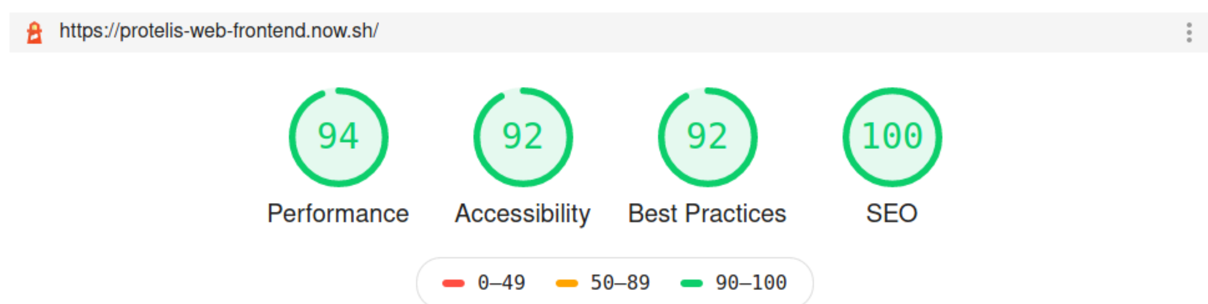


Figura 7.1.: Punteggio ottenuto con Google Lighthouse

Come è possibile vedere in Figura 7.1, un'analisi dell'applicazione web realizzata attraverso il plugin ufficiale per FireFox ha ottenuto un buon punteggio generale. Di seguito verranno analizzati i singoli valori separatamente.

7.2.1. Performance

Lighthouse ad ogni analisi assegna un punteggio denominato *performance* basato su diverse metriche in relazione ai diversi stadi caricamento della pagina; i valori grezzi ottenuti per ciascuna metrica vengono mappati su una distribuzione log-normale derivata dalle metriche prestazionali ricavate da HTTPArchive[1]. 0 è il minor valore possibile e generalmente indica un errore interno a Lighthouse, mentre 100 è il valore migliore, che posiziona la pagina analizzata nel 98-esimo percentile dei siti più performanti.

Secondo lo strumento, le performance del frontend di WebProtelis sono buone, con tempi di primo disegno inferiori al secondo e un ritardo prima di avere possibilità di interazione intorno a 1,6 s. A pesare un poco sul punteggio vi è l'assenza di gestione della cache delle richieste, funzionalità non ritenuta importante in questo progetto.

[1]https://httparchive.org/

7.2.2. Accessibilità e Best Practices

Il punteggio denominato *accessibility* è dato da una media ponderata dei numerosi audit di accessibilità effettuati; a differenza di quanto avveniva per le performance, ciascun audit non ha valori parziali, bensì solo 0 o 100. Anche il punteggio di *best practices* è composto da diverse caratteristiche, delle quali viene ricavata una media non pesata.

I punteggi ottenuti da questo progetto sono strettamente legati dalle librerie impiegate, sia in positivo che in negativo: il valore elevato deriva da ottimizzazioni della libreria stessa sui componenti che fornisce, cercando di aderire quanto possibile agli standard più moderni; le criticità che non portano al 100 % sono legate a limiti difficilmente aggirabili se non configurazioni complesse e fuori tema rispetto all'obiettivo di questa tesi.

7.2.3. SEO

Infine, con *SEO* si intende S*earch* E*ngine* O*ptimization*, ovvero quanto la pagina web è ottimizzata per la ricerca tramite motori come Google. Il punteggio è determinato dalla presenza dei corretti tag nel documento, che permettono alla pagina di essere analizzata e descritta in modo accurato. Nel caso di questo progetto, ottimizzando manualmente la configurazione base generata da React seguendo le specifiche documentate da Google, si è riuscito ad ottenere il massimo punteggio, che dovrebbe garantire un buon ranking nelle ricerche.

8. Considerazioni finali e lavori futuri

L'obiettivo della tesi era quello di progettare un sistema che permettesse, senza particolari configurazioni, di scrivere codice aggregato ed eseguirlo su una rete di esempio. Esso doveva essere facilmente accessibile e immediatamente utilizzabile, dunque si è scelto di indirizzarsi verso tecnologie web.

Per la realizzazione del prototipo, ci si è focalizzati su un solo linguaggio di programmazione aggregata: Protelis. Inoltre, per motivi di realizzabilità concreta, si è deciso di simulare la rete di dispositivi su cui eseguire il codice, anziché utilizzarne una fisica.

La realizzazione del sistema nel suo complesso ha comportato la progettazione di due applicazioni su differenti piattaforme e con diverse tecnologie e linguaggi: la componente server è stata realizzata avvalendosi del framework Vert.x nel linguaggio Kotlin, mentre l'interfaccia web è stata implementata come Single-Page Application avvalendosi di React in TypeScript.

Per la soddisfazione dei requisiti è stato dunque richiesto uno studio approfondito, anche a causa della profonda diversità di questi due sistemi. La decisione di appoggiarsi a soluzioni per la programmazione web moderne è stata però fondamentale per ottenere un sistema davvero di uso immediato come richiesto. Si ritiene dunque che il lavoro svolto abbia raggiunto gli obiettivi prefissati, diventando un'occasione formativa notevole e portando alla realizzazione di un sistema potenzialmente molto utile per la divulgazione della programmazione aggregata.

Il lavoro realizzato per questa tesi, per quanto sia un prototipo, si pone come punto di partenza per diverse possibili modifiche. In primo luogo, l'implementazione attuale si focalizza sulla *simulazione* di una rete di dispositivi, ma rimane aperto ad altre alternative. Ad esempio, è possibile rimpiazzare Alchemist con un utilizzo di più basso livello degli strumenti offerti dal framework di Protelis, andando a definire una differente implementazione dell'interfaccia `ProtelisEngine`.

Oppure, il sistema potrebbe essere riadattato facilmente per essere impiegato come interfaccia di monitoring per reti reali: utilizzando infatti un verticle come *bridge*, sarebbe possibile allacciare al sistema una rete di dispositivi fisici, eventualmente necessitando che uno di questi funga da entry-point.

Un altro aspetto su cui sarebbe interessante porre l'attenzione in futuro sarebbe la scalabilità: come detto nella Sezione 5.3, il prototipo è stato realizzato con un architettura considerabile monolitica, ma il framework offre molte libertà. Vert.x permette infatti l'esecuzione di singoli verticle (ed eventuali dipendenze) in modo indipendente, realizzando, di fatto, dei microservizi.

Esso offre inoltre il supporto a diversi strumenti per l'integrazione con tecnologie per il *service discovery*, per lo scambio di messaggi e per il bilanciamento del carico.

Una soluzione interessante di deploy potrebbe vedere, ad esempio, diversi verticle pacchettizzati come container Docker ed eseguiti in una piattaforma basata su Kubernetes o OpenShift, delegando a quest'ultimo livello PaaS il deploy di repliche per l'incremento delle performance on-demand.

Spostando l'attenzione sul client, potrebbe essere utile aumentare le possibilità di interazione con l'esecuzione. Potrebbe, ad esempio, risultare utile la possibilità di interagire con i nodi rappresentati, spostandoli e vedendo così l'esecuzione adattarsi alla perturbazione. Funzionalità di questo tipo possono essere inserite in modo abbastanza semplice tramite la realizzazione di eventi specifici, generati dai componenti React e inoltrati tramite SockJS verso il server.

Infine, un ultimo approccio di miglioramento potrebbe coinvolgere il cambio di parte delle tecnologie impiegate. Per la realizzazione di questo prototipo si è ritenuto l'uso di TypeScript ottimale per le ragioni espresse nelle Sezioni 2.2.2, 2.2.4 e 6.1.1, ma, potenzialmente, Kotlin potrebbe essere una soluzione molto interessante una volta che avrà raggiunto una stabilità accettabile per il target JS. Tale migrazione permettere una condivisione più efficiente delle componenti di modello condivise e potenzialmente delle dipendenze. Inoltre, collaborando con il team che mantiene il progetto Protelis, sarebbe potenzialmente possibile realizzare un'implementazione dell'interprete locale al client. In questo modo, il ruolo del server potrebbe diventare non più necessario per piccoli progetti a scopo educativo come quelli a cui questo progetto ha fatto riferimento fin dall'inizio.

Appendice

A. Dockerfile del server

Di seguito è riportato il codice del Dockerfile che permette il deploy del server come container Docker. L'immagine costruita durante lo sviluppo è anche disponibile su Docker Hub con il nome `niccomlt/protelis-web`.

```
FROM adoptopenjdk:11-jre-openj9

ENV VERTICLE_FILE protelis-on-web-all.jar
ENV VERTICLE_HOME /usr/verticles

EXPOSE 8080

COPY build/libs/$VERTICLE_FILE $VERTICLE_HOME/

WORKDIR $VERTICLE_HOME
ENTRYPOINT ["sh", "-c"]
CMD ["exec java -jar $VERTICLE_FILE"]
```

B. YAML di configurazione per Alchemist

Di seguito è riportata la configurazione in formato YAML che permette ad Alchemist di costruire la rete simulata sulla quale viene eseguito il codice.

```yaml
incarnation: protelis

environment:
  type: Continuous2DEnvironment
  parameters: []

network-model:
  type: ConnectWithinDistance
  parameters: [5]

displacements:
  - in:
      type: Grid
      parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]
    contents:
      - molecule: _to_exec
        concentration: ""
    programs:
      - - program: 'eval(env.get("_to_exec"))'
        - program: send
```

Ringraziamenti

Sono ormai giunto al termine di questo mio percorso universitario e per questo devo ringraziare tutte le persone che mi hanno accompagnato lungo questi cinque anni.

Ringrazio i miei genitori, che mi hanno materialmente permesso di compiere questo percorso e mi hanno supportato nei momenti di difficoltà. Ringrazio anche gli altri miei familiari che mi hanno sostenuto negli anni.

Ringrazio poi tutti gli amici che mi sono stati vicino in questo percorso: coloro con cui ho condiviso i progetti ingestibili e le infinite sessioni di studio in biblioteca o al campus, coloro con cui da sempre condivido serate di spensieratezza, coloro che ho trovato o *ri*trovato con l'università.

Infine ringrazio il professor Mirko Viroli e il professor Danilo Pianini per la bella opportunità che mi hanno offerto, per l'aiuto datomi per realizzare questo progetto e per le competenze che mi hanno permesso di sviluppare.

Grazie a tutti, di cuore.

Bibliografia

[1] C. Hewitt, P. B. Bishop e R. Steiger, «A Universal Modular ACTOR Formalism for Artificial Intelligence,» in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, N. J. Nilsson, cur., William Kaufmann, 1973, pp. 235–245. indirizzo: `http://ijcai.org/Proceedings/73/Papers/027B.pdf`.

[2] D. T. Gillespie, «A general method for numerically simulating the stochastic time evolution of coupled chemical reactions,» *Journal of Computational Physics*, vol. 22, n. 4, pp. 403–434, 1976, ISSN: 0021-9991. DOI: `10.1016/0021-9991(76)90041-3`. indirizzo: `http://www.sciencedirect.com/science/article/pii/0021999176900413`.

[3] ——, «Exact stochastic simulation of coupled chemical reactions,» *The Journal of Physical Chemistry*, vol. 81, n. 25, pp. 2340–2361, 1977. DOI: `10.1021/j100540a008`. eprint: `http://dx.doi.org/10.1021/j100540a008`.

[4] G. S. Fishman, *Principles of discrete event simulation (Wiley series on systems engineering and analysis)*. New York, NY, USA: John Wiley & Sons, Inc., 1978, ISBN: 9780471043959.

[5] J. Banks e J. S. Carson, «Introduction to discrete-event simulation,» in *Proceedings of the 18th conference on Winter simulation - WSC '86*, ser. WSC '86, Washington, D.C., USA: ACM Press, 1986, pp. 17–23, ISBN: 9780911801118. DOI: `10.1145/318242.318253`. indirizzo: `http://doi.acm.org/10.1145/318242.318253`.

[6] D. Ungar, C. Chambers, B.-W. Chang e U. Hölzle, «Organizing programs without classes,» *Lisp and Symbolic Computation*, vol. 4, n. 3, pp. 223–242, 1991, ISSN: 1573-0557. DOI: `10.1007/bf01806107`. indirizzo: `https://doi.org/10.1007/BF01806107`.

[7] E. Gamma, R. Helm, R. E. Johnson e J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1 dic. 1995, ISBN: 0201633612. indirizzo: `https://dl.acm.org/doi/book/10.5555/186897`.

[8] D. C. Schmidt, «Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events,» 1995.

[9] ISO Central Secretary, «Information technology — ECMAScript language specification,» en, International Organization for Standardization, Geneva, CH, Standard ISO/IEC 16262:1998, 1998, p. 98. indirizzo: https://www.iso.org/standard/29696.html.

[10] ECMA, *ECMA-262: ECMAScript Language Specification*, Third. pub-ECMA:adr: ECMA (European Association for Standardizing Information e Communication Systems), dic. 1999. indirizzo: http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM.

[11] M. A. Gibson e J. Bruck, «Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels,» *The Journal of Physical Chemistry A*, vol. 104, n. 9, pp. 1876–1889, 2000. DOI: 10.1021/jp993732q. eprint: http://dx.doi.org/10.1021/jp993732q. indirizzo: http://dx.doi.org/10.1021/jp993732q.

[12] R. Nagpal, «Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics,» tesi di dott., Massachusetts Institute of Technology, Cambridge, MA, USA, 2001. indirizzo: http://hdl.handle.net/1721.1/86667.

[13] S. Madden, R. Szewczyk, M. J. Franklin e D. E. Culler, «Supporting aggregate queries over ad-hoc wireless sensor networks,» in *Proceedings Fourth IEEE Workshop on Mobile Computing Systems and Applications*, IEEE Comput. Soc, 2002, pp. 49–58. DOI: 10.1109/mcsa.2002.1017485.

[14] E. McMullin, «The origins of the field concept in physics,» *Physics in Perspective*, vol. 4, n. 1, pp. 13–39, 2002. DOI: 10.1007/s00016-002-8357-5.

[15] T. Reenskaug, «The Model-View-Controller (MVC) Its Past and Present,» 2003. indirizzo: http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/HM1A93.html.

[16] K. A. L. Coar e D. Robinson, «The Common Gateway Interface (CGI) Version 1.1,» RFC Editor, rapp. tecn. 3875, 2004, 36 pp. DOI: 10.17487/rfc3875. indirizzo: https://rfc-editor.org/rfc/rfc3875.txt.

[17] J. Beal e J. Bachrach, «Infrastructure for Engineered Emergence on Sensor/Actuator Networks,» *IEEE Intelligent Systems*, vol. 21, n. 2, pp. 10–19, 2006. DOI: 10.1109/mis.2006.29.

[18] D. G. Bell, F. Kuehnel, C. Maxwell, R. Kim, K. Kasraie, T. Gaskins, P. Hogan e J. Coughlan, «NASA World Wind: Opensource GIS for Mission Operations,» in *2007 IEEE Aerospace Conference*, IEEE, 2007, pp. 1–9. DOI: 10.1109/aero.2007.352954.

[19] J. Dean e S. Ghemawat, «MapReduce: Simplified Data Processing on Large Clusters,» *Communications of the ACM*, vol. 51, n. 1, pp. 107–113, 2008, ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. indirizzo: https://doi.org/10.1145/1327452.1327492.

[20] M. Smith, «HTML 5 Publication Notes,» W3C, W3C Note, giu. 2008. indirizzo: http://www.w3.org/TR/2008/NOTE-html5-pubnotes-20080610/.

[21] K. Ashton, «That 'Internet of Things' thing,» *RFID journal*, vol. 22, n. 7, pp. 97–114, 2009. indirizzo: https://www.rfidjournal.com/articles/view?4986.

[22] M. Mamei e F. Zambonelli, «Programming Pervasive and Mobile Computing Applications: The TOTA Approach,» *ACM Transactions on Software Engineering and Methodology*, vol. 18, n. 4, pp. 1–56, 2009, ISSN: 1049-331X. DOI: 10.1145/1538942.1538945. indirizzo: https://doi.org/10.1145/1538942.1538945.

[23] E. Babulak e M. Wang, «Discrete Event Simulation: State of the Art,» *International Journal of Online Engineering (iJOE)*, vol. 4, n. 2, pp. 60–63, 2010. DOI: 10.5772/9894. indirizzo: https://doi.org/10.5772/9894.

[24] T. Berners-Lee, R. Cailliau, J.-F. Groff e B. Pollermann, «World-wide web: the information universe,» *Internet Research*, vol. 20, n. 4, pp. 461–471, 2010. DOI: 10.1108/10662241011059471.

[25] S. Tilkov e S. Vinoski, «Node.js: Using JavaScript to Build High-Performance Network Programs,» *IEEE Internet Computing*, vol. 14, n. 6, pp. 80–83, 2010, ISSN: 1941-0131. DOI: 10.1109/mic.2010.145.

[26] T. V. Cutsem e M. S. Miller, «Traits.Js: Robust Object Composition and High-Integrity Objects for Ecmascript 5,» in *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients - PLASTIC '11*, ser. PLASTIC '11, Portland, Oregon, USA: ACM Press, 2011, pp. 1–8, ISBN: 9781450311717. DOI: 10.1145/2093328.2093330. indirizzo: https://doi.org/10.1145/2093328.2093330.

[27] A. Melnikov e I. Fette, «The WebSocket Protocol,» RFC Editor, rapp. tecn. 6455, 2011, 71 pp. DOI: 10.17487/rfc6455. indirizzo: https://rfc-editor.org/rfc/rfc6455.txt.

[28] C. Anderson, «The Model-View-ViewModel (MVVM) Design Pattern,» in *Pro Business Applications with Silverlight 5*, Berkeley, CA: Springer-Verlag GmbH, 8 giu. 2012, pp. 461–499, ISBN: 9781430235019. DOI: 10.1007/978-1-4302-3501-9_13. indirizzo: https://doi.org/10.1007/978-1-4302-3501-9_13.

[29] R. Chugh, P. M. Rondon e R. Jhala, «Nested refinements: a logic for duck typing,» *ACM SIGPLAN Notices*, vol. 47, n. 1, pp. 231–244, 2012, ISSN: 0362-1340. DOI: 10.1145/2103621.2103686. indirizzo: https://doi.org/10.1145/2103621.2103686.

[30] E. Czaplicki, «Elm: Concurrent FRP for Functional GUIs,» Senior thesis, Harvard University, 30 mar. 2012. indirizzo: https://elm-lang.org/assets/papers/concurrent-frp.pdf.

[31] D. Pianini, S. Montagna e M. Viroli, «Chemical-oriented simulation of computational systems with ALCHEMIST,» *Journal of Simulation*, vol. 7, n. 3, pp. 202–215, 2013, ISSN: 1747-7778. DOI: 10.1057/jos.2012.27. eprint: https://doi.org/10.1057/jos.2012.27. indirizzo: http://link.springer.com/10.1057/jos.2012.27.

[32] M. Viroli, F. Damiani e J. Beal, «A Calculus of Computational Fields,» in *Advances in Service-Oriented and Cloud Computing*, ser. Communications in Computer and Information Science, C. Canal e M. Villari, cur., vol. 393, Pre-proceedings available at: http://foclasa.lcc.uma.es/documents/foclasa2013-preproceedings.pdf, Malaga, Spain: Springer Berlin Heidelberg, 2013, cap. 10, pp. 114–128. DOI: 10.1007/978-3-642-45364-9_11. indirizzo: http://link.springer.com/book/10.1007/978-3-642-45364-9.

[33] J. Beal e 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*, IEEE, 2014, pp. 8–13. DOI: 10.1109/sasow.2014.6.

[34] J. Beal, D. Pianini e M. Viroli, «Aggregate Programming for the Internet of Things,» *Computer*, vol. 48, n. 9, pp. 22–30, set. 2015, ISSN: 0018-9162. DOI: 10.1109/mc.2015.261.

[35] S. S. Clark, J. Beal e P. Pal, «Distributed Recovery for Enterprise Services,» in *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*, IEEE, set. 2015, pp. 111–120. DOI: 10.1109/saso.2015.19.

[36] B. Fisher, «Flux: A Unidirectional Data Flow Architecture for React Apps,» in *Applicative 2015*, ser. Applicative 2015, New York, NY, USA: Association for Computing Machinery, 2015, ISBN: 9781450335270. DOI: 10.1145/2742580.2742818. indirizzo: https://doi.org/10.1145/2742580.2742818.

[37] D. Pianini, A. Croatti, A. Ricci e M. Viroli, «Computational Fields Meet Augmented Reality: Perspectives and Challenges,» in *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, IEEE, 2015, pp. 80–85. DOI: 10.1109/sasow.2015.18.

[38] R. L. Wainwright, J. M. Corchado, A. Bechini e J. Hong, cur., *Protelis: practical aggregate programming*, SAC '15, Salamanca, Spain: Association for Computing Machinery, 2015, pp. 1846–1853, ISBN: 978-1-4503-3196-8. DOI: 10.1145/2695664.2695913. indirizzo: https://doi.org/10.1145/2695664.2695913.

[39] J. Beal, K. Usbeck, J. Loyall e J. Metzler, «Opportunistic Sharing of Airborne Sensors,» in *2016 International Conference on Distributed Computing in Sensor Systems (DCOSS)*, IEEE, 2016, pp. 25–32. DOI: 10.1109/dcoss.2016.43.

[40] J. Beal, K. Usbeck, J. Loyall, M. Rowe e J. Metzler, «Adaptive Task Reallocation for Airborne Sensor Sharing,» in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, IEEE, 2016, pp. 168–173. DOI: 10.1109/fas-w.2016.46.

[41] R. Casadei e M. Viroli, «Towards Aggregate Programming in Scala,» in *First Workshop on Programming Models and Languages for Distributed Computing on - PMLDC '16*, Rome, Italy: ACM Press, lug. 2016, 5:1–5:7, ISBN: 978-1-4503-4775-4. DOI: 10.1145/2957319.2957372. indirizzo: http://doi.acm.org/10.1145/2957319.2957372.

[42] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip e Y. Choi, «Type inference for static compilation of JavaScript,» *ACM SIGPLAN Notices*, vol. 51, n. 10, pp. 410–429, 2016, ISSN: 0362-1340. DOI: 10.1145/3022671.2984017. indirizzo: https://doi.org/10.1145/3022671.2984017.

[43] S. Marr, B. Daloze e H. Mössenböck, «Cross-language compiler benchmarking: are we fast yet?» *ACM SIGPLAN Notices*, vol. 52, n. 2, pp. 120–131, 2016, ISSN: 0362-1340. DOI: 10.1145/3093334.2989232. indirizzo: https://doi.org/10.1145/3093334.2989232.

[44] K. Rushforth, A. Herrick, D. Dehaven, K. Rushforth e S. Marks, *JEP 289: Deprecate the Applet API*, Oracle, cur., http://openjdk.java.net/jeps/289, 9 feb. 2016.

[45] C. Varini, «Sviluppo di un simulatore per la piattaforma Scafi,» Tesi di Laurea, Università di Bologna, 2016. indirizzo: http://amslaurea.unibo.it/12188/.

[46] M. Viroli, R. Casadei e D. Pianini, «On execution platforms for large-scale aggregate computing,» in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing Adjunct - UbiComp '16*, P. Lukowicz, A. Krüger, A. Bulling, Y. Lim e S. N. Patel, cur., ser. UbiComp '16, Heidelberg, Germany: ACM Press, 2016, pp. 1321–1326, ISBN: 978-1-4503-4462-3. DOI: 10.1145/2968219.2979129. indirizzo: http://doi.acm.org/10.1145/2968219.2979129.

[47] R. Belov. (1 mar. 2017). Kotlin 1.1 Released with JavaScript Support, Coroutines and more. J. K. Blog, cur., indirizzo: https://blog.jetbrains.com/kotlin/2017/03/kotlin-1-1/.

[48] M. Francia, D. Pianini, J. Beal e M. Viroli, «Towards a Foundational API for Resilient Distributed Systems Design,» in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, IEEE Computer Society, 2017, pp. 27–32. DOI: 10.1109/fas-w.2017.116. indirizzo: http://doi.ieeecomputersociety.org/10.1109/FAS-W.2017.116.

[49] D. Jemerov. (28 nov. 2017). Kotlin 1.2 Released: Sharing Code between Platforms. J. K. Blog, cur., indirizzo: https://blog.jetbrains.com/kotlin/2017/11/kotlin-1-2-released/.

[50] D. Pianini, J. Beal e M. Viroli, «Practical Aggregate Programming with Protelis,» in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, IEEE, 2017, pp. 391–392. DOI: 10.1109/fas-w.2017.186.

[51] G. Aguzzi, «Sviluppo di un front-end di simulazione per applicazioni aggregate nel framework Scafi,» Tesi di Laurea, Università di Bologna, 2018. indirizzo: http://amslaurea.unibo.it/16824/.

[52] G. Audrito, J. Beal, F. Damiani e M. Viroli, «Space-Time Universality of Field Calculus,» in *Coordination Models and Languages*, G. Di Marzo Serugendo e M. Loreti, cur., Cham: Springer International Publishing, 2018, pp. 1–20, ISBN: 978-3-319-92408-3.

[53] S. J. R. Doeraene, «Cross-Platform Language Design,» eng, tesi di dott., IINFCOM, Lausanne, 2018, p. 184. DOI: 10.5075/EPFL-THESIS-8733. indirizzo: http://infoscience.epfl.ch/record/256862.

[54] G. Audrito, M. Viroli, F. Damiani, D. Pianini e J. Beal, «A Higher-Order Calculus of Computational Fields,» *ACM Transactions on Computational Logic*, vol. 20, n. 1, 5:1–5:55, 2019. DOI: 10.1145/3285956. indirizzo: https://doi.org/10.1145/3285956.

[55] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio e M. Viroli, «Modelling and simulation of Opportunistic IoT Services with Aggregate Computing,» *Future Generation Computer Systems*, vol. 91, pp. 252–262, feb. 2019. DOI: 10.1016/j.future.2018.09.005. indirizzo: https://doi.org/10.1016/j.future.2018.09.005.

[56] F. Nardini, «Sviluppo di piattaforme per il linguaggio Protelis in Kotlin e Java,» Tesi di Laurea, Università di Bologna, 2019. indirizzo: http://amslaurea.unibo.it/19778/.