ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Triennale in Ingegneria e Scienze Informatiche

Da MIT Proto a Collektive: Trasposizione di esempi di programmazione aggregata in un DSL in Kotlin.

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Danilo Pianini

Candidato
Andrea Cecchini

Correlatore
Dott. Angela Cortecchia

III Sessione di Laurea
Anno Accademico 2024-2025

Sommario

La rapida diffusione di dispositivi intelligenti ha reso sempre più centrale la programmazione di sistemi distribuiti su larga scala. I paradigmi tradizionali, incentrati sul singolo dispositivo, evidenziano limiti in termini di modularità, scalabilità e resilienza, rendendo complessa la progettazione di applicazioni collettive. La programmazione aggregata affronta queste criticità adottando una prospettiva globale: il comportamento dell'intera rete viene descritto tramite un unico programma aggregato, fondato su formalismi come Field Calculus e le sue estensioni.

Nel corso degli anni sono stati sviluppati diversi linguaggi e framework per supportare questo paradigma, ciascuno con punti di forza e debolezze. MIT Proto ha rappresentato una pietra miliare storica, ma soffre di limitazioni legate ad ergonomia e portabilità. Collektive, un Domain-Specific Language interno a Kotlin, si propone come soluzione moderna, offrendo una sintassi espressiva, gestione trasparente dell'allineamento e supporto multipiattaforma.

Questo lavoro presenta la trasposizione di una selezione di programmi aggregati, originariamente sviluppati in Proto, nell'ecosistema Collektive. L'obiettivo è fornire una guida pratica alla migrazione di sistemi esistenti verso soluzioni più moderne e robuste, facilitando l'adozione delle nuove tecnologie.

Alla mia famiglia, per il supporto incondizionato e l'incoraggiamento costante. A Maria Chiara e ai miei amici, per la pazienza e la comprensione dimostrate durante questo percorso.

Indice

Capitolo 1

Introduzione

1.1 Contesto

Le attuali tendenze tecnologiche, in particolare la rapida espansione dell'Internet of Things (IoT), stanno favorendo una crescente integrazione dei dispositivi computazionali nell'ambiente circostante, aprendo nuove prospettive per servizi e applicazioni innovative. Tra gli esempi più rilevanti si annoverano sistemi di monitoraggio ambientale, reti di sensori intelligenti, infrastrutture urbane connesse e robotica collaborativa.

I paradigmi di programmazione tradizionali, che individuano l'unità programmabile nel singolo dispositivo, risultano ormai inadatti allo sviluppo di applicazioni distribuite su larga scala. Questa prospettiva individuale evidenzia le complessità legate alla gestione della comunicazione, della sincronizzazione e del coordinamento tra nodi, che diventano parti integranti del codice applicativo del sistema distribuito. Con l'aumentare della complessità, tali sistemi manifestano limiti di design, come scarsa modularità e riusabilità, difficoltà nel deployment e nel testing. Di conseguenza, la progettazione di sistemi distribuiti viene spesso percepita come una sfida ardua e rischiosa, generando timore e resistenza invece che stimolare la valorizzazione delle potenzialità offerte da un ecosistema di dispositivi cooperanti.

1.1.1 Aggregate Programming

Aggregate Programming (AP) [BPV15] è un paradigma di macroprogrammazione [Cas22] ideato per lo sviluppo di Sistemi Collettivi Adattivi (CAS). A differenza degli approcci tradizionali, in cui il comportamento globale di un sistema emerge dall'interazione dei comportamenti locali dei singoli dispositivi (logica bottom-up), AP adotta una prospettiva globale (top-down): il comportamento desiderato dell'intera rete viene specificato attraverso un unico programma aggregato, in cui l'unità programmabile non è più il singolo dispositivo, bensì l'intero collettivo. Saranno poi il linguaggio e gli strumenti sottostanti a derivare automaticamente i comportamenti locali necessari affinché il sistema si auto-organizzi per soddisfare le specifiche globali.

Dal punto di vista formale, AP si fonda su Field Calculus (FC) [BPV15] (Sezione 1.1.2), un linguaggio minimale progettato per specificare il comportamento aggregato di un sistema distribuito. L'astrazione centrale è quella del *campo computazionale* [BPV15] (Sezione 1.1.2): una mappa, potenzialmente dinamica, che associa a ciascun dispositivo della rete un valore. Da una prospettiva globale, un programma aggregato implementa una serie di trasformazioni su queste strutture dati distribuite. L'approccio a campi consente di creare algoritmi distribuiti riusabili tramite funzioni componibili — da campo a campo — organizzabili in librerie di complessità crescente per costruire interi servizi applicativi.

Proprietà I sistemi sviluppati con AP possiedono intrinsecamente proprietà di adattabilità e resilienza, che si manifestano in risposta a variazioni dell'ambiente operativo, come cambiamenti nella topologia della rete, guasti dei dispositivi o fluttuazioni delle risorse disponibili. Inoltre, tali sistemi sono in grado di gestire efficientemente un numero crescente di dispositivi, mantenendo basso l'overhead comunicativo e l'efficienza computazionale.

Architettura a livelli AP semplifica la complessità della coordinazione distribuita grazie a una struttura a più livelli di astrazione (Figura 1.1), costruita a partire da FC [BPV15].

La scelta di basarsi su un calcolo minimale e matematico permette di dimostrare formalmente le proprietà dei costrutti fondamentali e di definire operatori

generali con caratteristiche verificabili. Questi operatori possono essere utilizzati per realizzare librerie di alto livello, preservando nelle composizioni le proprietà dimostrate a livello dei singoli componenti.

1.1.2 Field Calculus e Higher-Order Field Calculus

FC è un linguaggio funzionale minimale, progettato per offrire i costrutti fondamentali necessari alla manipolazione dei campi [AVD+19]. In questo formalismo, l'astrazione unificante è quella del *campo computazionale*, attorno al quale si organizzano i costrutti del linguaggio che permettono di operare su tali strutture.

Campo computazionale Un campo computazionale è una struttura dati distribuita nello spazio e nel tempo che associa ad ogni dispositivo un valore. A titolo esemplificativo, si consideri una rete di sensori di temperatura distribuiti su un'area geografica: tale configurazione può essere interpretata come un campo computazionale discreto, in cui ogni sensore rappresenta un punto del campo e il valore a esso associato corrisponde alla temperatura rilevata in quel istante.

Modello di Esecuzione Una computazione può essere osservata da due prospettive diverse: locale e globale [AVD+19].

Dal punto di vista aggregato (globale), è possibile specificare il comportamento collettivo del sistema attraverso una serie di trasformazioni su campi computazionali. Ad esempio, l'input di una computazione può essere un campo di temperature, in cui a ciascun dispositivo viene associato il valore rilevato dal proprio sensore; l'output, invece, può essere un campo booleano che indica quali dispositivi superano una determinata soglia di temperatura.

Dal punto di vista locale, la computazione è vista dal singolo dispositivo δ_i, che esegue periodicamente un programma `P` in round di esecuzione asincroni. In questa visione, il dispositivo δ_i non può percepire l'intero campo globale, ma solo una proiezione limitata ai dispositivi entro il proprio raggio di comunicazione, definita come *campo di vicinato*, ovvero una mappa $\phi : \delta \to v$ che associa a ciascun vicino δ_j il valore v_j più recente. In ogni round, il dispositivo: i) raccoglie i valori dal proprio campo di vicinato ϕ_i; ii) ottiene le informazioni dall'ambiente (tramite

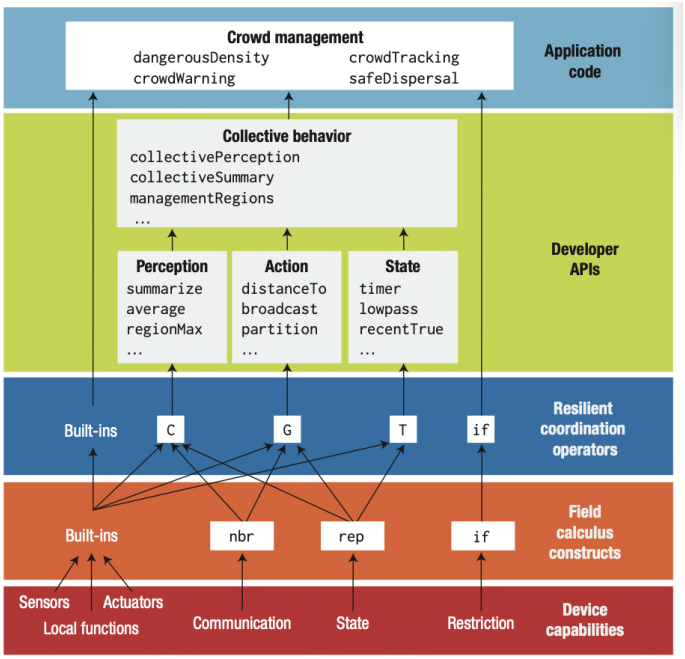

Figura 1.1: Architettura a livelli della programmazione aggregata [BPV15]. Le capacità del dispositivo sono sfruttate per implementare i costrutti di FC, che a loro volta costituiscono la base per la creazione di un limitato numero di operatori aggregati di cui si può provare la resilienza. Tali costrutti sono così generali da poter essere utilizzati per costruire librerie di alto livello per la programmazione aggregata.

sensori); iii) recupera lo stato interno; iv) valuta il programma `P` utilizzando queste informazioni; v) aggiorna il proprio stato e invia il valore risultante ai vicini; vi) attende il successivo round di esecuzione [AVD+19].

Sintassi I costrutti fondamentali di FC [BPV15] sono:

- Chiamate a funzione: `b`$(e_1, \ldots, e_n)$ applica la funzione `b` agli argomenti $e_1, \ldots, e_n$, restituendo un nuovo campo computazionale.
- Dinamica Temporale: `rep`(e_0) {`(x) => e`} è fondamentale per modellare l'evoluzione nel tempo di un campo. Inizializza una variabile di stato locale `x` con il valore iniziale e_0 e, ad ogni round di esecuzione, aggiorna `x` con il risultato dell'espressione `e`. L'espressione `e` può fare riferimento al valore precedente di `x`, consentendo la definizione di processi iterativi.
- Interazione Spaziale: `nbr`{`e`} modella l'evoluzione spaziale del campo, consentendo a un dispositivo di accedere ai valori calcolati dai vicini nell'ultimo round di esecuzione. Restituisce un campo di vicinato (ϕ) che mappa l'identificatore di ogni vicino all'espressione `e` valutata in quel dispositivo. I campi di vicinato così ottenuti possono essere manipolati tramite funzioni di aggregazione spaziale, come `min-hood`, `max-hood`, `sum-hood` e altre.
- Restrizione del dominio: `if` (e_0) $\{e_1\}$ `else` $\{e_2\}$ partiziona la rete in due sottoinsiemi D_{true} e D_{false} in base alla valutazione di e_0. Se un dispositivo δ appartiene a D_{true}, valuta esclusivamente l'espressione e_1, ignorando e_2; viceversa, se appartiene a D_{false}, valuta solo e_2. In entrambi i casi, le informazioni di stato e i valori scambiati con i vicini non vengono condivisi tra i due rami: quando un dispositivo passa da un ramo all'altro, le variabili di stato vengono reinizializzate e i valori precedenti vengono persi. Se è necessario condividere informazioni tra i rami, si può utilizzare il costrutto `mux`(e_0, e_1, e_2), che valuta entrambe le espressioni e_1 ed e_2, prima del partizionamento, restituendo il valore corrispondente a e_0.
- Share: la combinazione dei costrutti `rep` e `nbr` permette di modellare l'evoluzione sia spaziale che temporale di un campo. Tuttavia, come evidenziato in

[ABDV18], la loro integrazione introduce un ritardo implicito nel sistema che può causare la perdita di round. Per risolvere questo problema, l'approccio presentato in [ABD+19] introduce il costrutto `share`:

$$\texttt{share}(e_1)\ \{\texttt{(x) => } e_2\}$$

Nonostante la sintassi sia identica a quella di `rep`, i due si distinguono radicalmente per l'interpretazione della variabile `x` ad ogni round. Nel caso di `share`, `x` non rappresenta un valore locale (come in `rep`), bensì un campo di vicinato (ϕ) che contiene i valori condivisi dai vicini nel round precedente. L'espressione e_2 ha quindi il compito di elaborare questo campo di vicinato per produrre un nuovo valore locale, che sarà poi utilizzato come e_1 nel round successivo e condiviso con i vicini.

Allineamento Ogni nodo deve comunicare esclusivamente con i dispositivi che si trovano nello stesso contesto di esecuzione, ossia che hanno seguito lo stesso percorso di valutazione del programma: hanno invocato le stesse funzioni e preso le stesse decisioni. Tali dispositivi si definiscono *allineati*.

Si consideri l'esempio in listing 1.1, in cui sono definite due funzioni, `f1` e `f2`, entrambe contenenti il costrutto `nbr` applicato alla stessa espressione `e1`. Quando un dispositivo esegue `nbr {e1}` all'interno di `f1`, deve recuperare esclusivamente i valori dei vicini che hanno valutato l'espressione nel medesimo contesto. Di conseguenza, i valori prodotti dai vicini per `nbr {e1}` all'interno di `f2` risultano irrilevanti.

Listing 1.1: Allineamento tra le esecuzioni di `nbr` in contesti diversi

```
1 def f1() { nbr {e1} }
2 def f2() { nbr {e1} }
3
4 f1()
5 f2()
```

Questo aspetto risulta particolarmente evidente nel costrutto `if`, che non si limita a determinare quale ramo eseguire, bensì suddivide il dominio in due gruppi distinti, ciascuno con il proprio contesto di esecuzione. In questi casi, l'allineamen-

to diventa fondamentale per garantire che la comunicazione avvenga solo tra nodi che hanno scelto lo stesso ramo.

Poiché non esiste una memoria condivisa per tracciare lo stato computazionale, è necessario introdurre meccanismi di allineamento capaci di monitorare ogni nodo e determinare quali risultino effettivamente allineati. In un linguaggio di programmazione ideale, questi dettagli di basso livello dovrebbero rimanere trasparenti all'utente finale.

Higher-Order Field Calculus FC è stato esteso in [AVD+19] con l'introduzione di Higher-Order Field Calculus (HFC), che consente di trattare le funzioni come valori. Questa estensione abilita capacità fondamentali quali: i) accettare funzioni come argomenti e restituirle come risultati; ii) creare funzioni dinamicamente; iii) trasferirle tra dispositivi (tramite `nbr`) e memorizzarle o modificarle nel tempo (tramite `rep`).

1.1.3 XC

eXchange Calculus (XC) è un calcolo formale che definisce una semantica operazionale per la programmazione aggregata, generalizzando ed estendendo le capacità di FC [ACD+24]. È progettato per supportare lo sviluppo di comportamenti collettivi adattivi astraendo la gestione di aspetti di basso livello quali concorrenza, esecuzione asincrona, comunicazione e gestione dei guasti.

Modello di Esecuzione Ogni dispositivo, indicato con δ_i, comunica con i propri vicini tramite uno scambio di messaggi. Il funzionamento dei nodi si articola in round di esecuzione asincroni. In ogni round, ciascun dispositivo esegue un programma XC, elabora i messaggi ricevuti e produce nuovi messaggi da inviare al vicinato (Figura 1.2).

Data la natura asincrona del sistema, può verificarsi che un dispositivo completi più round prima che un vicino ne termini uno. In questi casi, il vicino considererà solo l'ultimo messaggio ricevuto, sovrascrivendo i precedenti.

Tipi di dato Nel formalismo FC, si distinguono due categorie di valori: i valori locali ℓ e i campi di vicinato ϕ. Quest'ultimi sono mappe che associano un valore

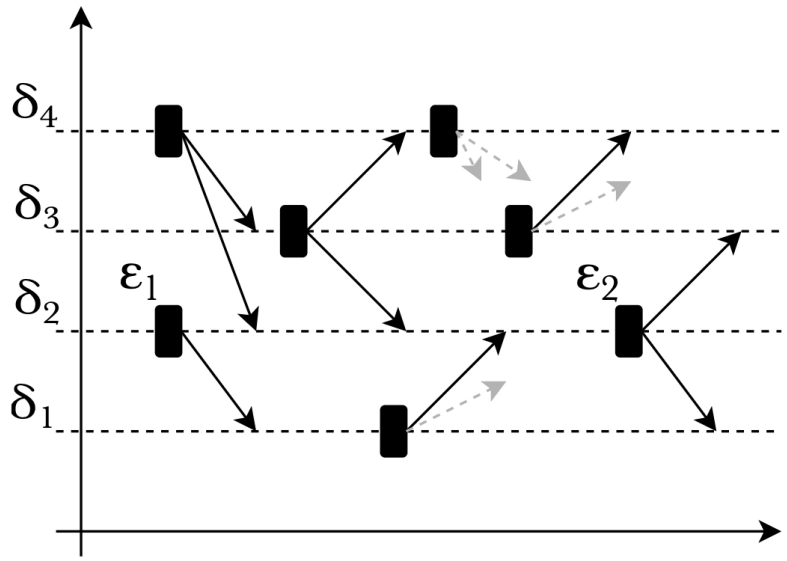

Figura 1.2: Modello di esecuzione di XC: il dispositivo δ_i esegue il programma XC nei round ε_n e invia i messaggi ai vicini al termine di ciascuno [ACD+24].

locale a ciascun identificatore di dispositivo e vengono impiegati per rappresentare i messaggi ricevuti dai vicini. Tuttavia, FC impone che solo i valori locali ℓ possano essere inviati. Ciò implica che ogni nodo trasmetta necessariamente lo stesso messaggio a tutti i suoi vicini (comunicazione *isotropica*), limitando l'espressività del linguaggio impedendo l'invio di messaggi differenziati.

XC risolve questa limitazione unificando le due categorie in una singola classe di valori denominata `nvalues` (v), consentendo a ciascun dispositivo di inviare messaggi diversi a vicini differenti (comunicazione *anisotropica*).

Un `nvalue` è una struttura dati che associa un valore locale ℓ_i a ciascun identificatore di dispositivo δ_i specificato e include un valore di default ℓ:

$$v = \ell[\delta_1 \to \ell_1, \ldots, \delta_n \to \ell_n]$$

Tale notazione significa che v assume il valore di default ℓ per tutti i dispositivi, ad eccezione di quelli esplicitamente specificati $(\delta_1, \ldots, \delta_n)$, ai quali sono associati i valori $\ell_1, \ldots, \ell_n$. Un valore locale ℓ può essere automaticamente convertito in un `nvalue` con ℓ come valore di default e senza eccezioni ($\ell[]$).

Questa unificazione semplifica notevolmente il sistema di tipi rispetto a FC e, grazie alla flessibilità degli `nvalues`, XC risulta più espressivo e versatile nella gestione della comunicazione eterogenea tra dispositivi.

exchange La comunicazione in XC è costruita attorno ad un'unica primitiva chiamata `exchange` [ACD+24], che incapsula in unico costrutto computazione, comunicazione e gestione dello stato.

La sintassi è la seguente:

$$\texttt{exchange(}e_i\texttt{, (n)) => return } e_r \texttt{ send } e_s$$

Dove:

- $\mathbf{e_i}$: È l'espressione che determina il valore locale iniziale. Tale valore funge da default per l'`nvalue` dei messaggi nel primo round o quando non sono ancora stati ricevuti messaggi dai vicini.

- **n**: È una variabile che, ad ogni round di esecuzione, viene automaticamente sostituita con l'`nvalue` che incapsula i messaggi ricevuti dai vicini.
- $\mathbf{e_r}$: È l'espressione che definisce il valore locale di ritorno del costrutto, che rappresenta il risultato del round corrente del dispositivo.
- $\mathbf{e_s}$: È l'espressione che definisce l'`nvalue` dei messaggi che il dispositivo invierà ai vicini al termine del round corrente.

Ogni costrutto di FC può essere espresso in termini di `exchange`, come illustrato in [ACD+24]. Il fatto che ogni programma FC possa essere codificato in XC, permette a quest'ultimo di ereditare tutte le proprietà di resilienza e universalità dimostrate per FC [ABDV18].

1.2 Motivazione

Il panorama della programmazione aggregata è caratterizzato da diversi linguaggi e framework che implementano i concetti fondamentali di FC, ognuno con specifici vantaggi e svantaggi. Tra le soluzioni più note e consolidate si annoverano MIT Proto [BB06] (Sezione 1.2.1), Protelis [PVB15] (Sezione 1.2.2), ScaFi [CVAP22] (Sezione 1.2.3) e FCPP [Aud20] (Sezione 1.2.4).

In questo scenario, l'ultima proposta è Collektive (Sezione 1.2.5), un Domain-Specific Language (DSL) sviluppato in Kotlin. Collektive si pone l'obiettivo di sintetizzare i punti di forza dei linguaggi esistenti, superandone al contempo le limitazioni strutturali, rappresentando così un passo in avanti nel contesto della programmazione aggregata.

1.2.1 MIT Proto

Proto [BB06], riconosciuto come il precursore della programmazione aggregata, fu concepito originariamente da Jonathan Bachrach e Jacob Beal e si basa su *Amorphous Computing* [Bea04].

È un linguaggio puramente funzionale, con una sintassi ispirata a Lisp (Listing 1.2).

Listing 1.2: Dimostrazione della sintassi di Proto

```
(def distance-to (src)
   (letfed ((n infinity (mux src 0 (min-hood (+ (nbr n) (nbr-range))))))
      n))
```

A supporto delle sue funzionalità, veniva fornito un simulatore integrato che consentiva di eseguire e testare i programmi in ambienti simulati. Successivamente fu sviluppato anche WebProto, un ambiente di prototipazione web che permetteva di scrivere e testare codice aggregato direttamente dal browser.

Lo sviluppo del linguaggio è stato interrotto nel 2016 e Proto è stato ufficialmente dismesso in favore di Protelis [PVB15] (Sezione 1.2.2), che ne ha raccolto l'eredità concettuale e operativa.

La sintassi e i costrutti del linguaggio verranno esaminati in dettaglio nel capitolo 2.

1.2.2 Protelis

Protelis [PVB15] è un linguaggio di programmazione funzionale che implementa i concetti di HFC (Sezione 1.1.2). La sua sintassi è ispirata al linguaggio C, il che lo rende più accessibile rispetto al suo predecessore. È inoltre caratterizzato dall'essere debolmente tipizzato (Listing 1.3).

Protelis è un linguaggio stand-alone per la Java Virtual Machine (JVM), sviluppato utilizzando Xtext, un framework specializzato nella creazione di DSL. L'esecuzione dei programmi avviene tramite un interprete che valuta periodicamente il codice aggregato. Questo gestisce in modo automatico la comunicazione tra dispositivi e l'interazione con l'ambiente. Poiché l'interprete è stato scritto da zero, i meccanismi di basso livello, come l'allineamento, sono gestiti in modo trasparente per l'utente finale, senza richiedere interventi espliciti nel codice applicativo. L'integrazione con la JVM garantisce portabilità tra sistemi e dispositivi, oltre a facilitare l'importazione di numerose librerie e API tramite i meccanismi di reflection di Java. Tuttavia, la necessità di una JVM rappresenta una limitazione per l'impiego in ambienti con risorse estremamente ristrette, come quelli tipici dell'IoT, riducendo l'eterogeneità dei dispositivi utilizzabili.

Attualmente il linguaggio è attivamente mantenuto, ma lo sviluppo di nuove funzionalità è stato congelato.

Listing 1.3: Dimostrazione della sintassi di Protelis

```
def distanceTo(source) {
  share (distance <- POSITIVE_INFINITY) {
    mux (source) {
      0
    } else {
      foldMin(POSITIVE_INFINITY, distance + self.nbrRange())
    }
  }
}
```

1.2.3 ScaFi

ScaFi [CVAP22] (Scala Fields) è un framework per la programmazione aggregata, realizzato come DSL interno al linguaggio Scala e basato su FC (1.1.2), in particolare sulla variante FScaFi [CVAD20].

Offre un ecosistema completo per lo sviluppo di programmi aggregati, comprendente librerie, strumenti di simulazione e la possibilità di prototipazione rapida via browser grazie a ScaFi Web [ACM+21].

L'integrazione come DSL interno consente a ScaFi di beneficiare direttamente della sintassi (Listing 1.4), del compilatore e degli strumenti di Scala, risultando così più semplice da mantenere, immediatamente familiare per chi già conosce il linguaggio ospite e in grado di sfruttare appieno il potente sistema di tipi di Scala.

Questa scelta, tuttavia, comporta anche alcune limitazioni: la sintassi è vincolata ai costrutti validi di Scala e, aspetto cruciale per la programmazione aggregata, possono emergere involontariamente dettagli di basso livello che non dovrebbero essere esposti all'utente, come la gestione dell'allineamento. Inoltre, essendo pensato principalmente per l'esecuzione sulla JVM, ScaFi condivide le stesse limitazioni di applicabilità già riscontrate per Protelis (1.2.2).

Listing 1.4: Dimostrazione della sintassi di ScaFi

```
def distanceTo(source: Boolean): Double =
  rep(Double.PositiveInfinity) (d => {
    mux (source) { 0.0 } {
      foldHoodPlus(Double.PositiveInfinity)(Math.min) {
        nbr(d) + nbrRange
      }
    }
  })
```

1.2.4 FCPP

FCPP [Aud20] rappresenta la prima implementazione nativa di AP in C++.

L'utilizzo di C++ come linguaggio di base è un elemento chiave, poiché consente l'esecuzione diretta del codice sui dispositivi, senza la necessità di una macchina virtuale. Questo approccio offre due vantaggi principali: da un lato, risulta particolarmente adatto a dispositivi con risorse limitate, come sistemi di microcontrollori e dispositivi embedded tipici dell'IoT, superando così le restrizioni imposte dalla dipendenza dalla JVM che caratterizzano soluzioni come Protelis e ScaFi; dall'altro, permette di ottenere prestazioni elevate nell'implementazione del paradigma aggregato.

Tuttavia, la scelta di C++ comporta anche alcune criticità. La complessità della sintassi (Listing 1.5) e la limitata ergonomia rispetto ad altre soluzioni rendono difficile l'adozione da parte di un ampio pubblico di sviluppatori. Inoltre, la gestione dell'allineamento non è trasparente, ma richiede interventi manuali tramite macro. Sebbene FCPP sia pensato come una piattaforma flessibile ed estensibile, il numero di funzionalità attualmente disponibili risulta inferiore rispetto alle alternative.

Listing 1.5: Dimostrazione della sintassi di FCPP

```
DEF() double distance_to(ARGS, bool source) { CODE
  return nbr(CALL, INF, [&] (field<double> d) {
    double v = source ? 0.0 : INF;
    return min_hood(CALL, d + node.nbr_dist(), v);
  });
}
```

1.2.5 Collektive

Collektive[1] nasce dall'esigenza di disporre di un linguaggio che i) garantisca una gestione dell'allineamento robusta e completamente trasparente; ii) sia multipiattaforma e quindi adatto a operare su reti di dispositivi eterogenei; iii) offra una sintassi moderna, espressiva ed ergonomica; iv) possa contare su un ecosistema maturo e ricco di funzionalità grazie al linguaggio host.

I linguaggi analizzati in precedenza non riescono a soddisfare tutti questi requisiti contemporaneamente: Protelis offre una buona gestione dell'allineamento ma dipende dalla JVM; ScaFi è anch'esso vincolato alla JVM e presenta un allineamento debole; FCPP è nativo e performante ma manca di ergonomia, semplicità d'uso e trasparenza.

Analogamente a ScaFi, Collektive è un DSL interno; tuttavia, si basa su Kotlin, la cui natura multi-piattaforma, resa possibile dal progetto Kotlin Multiplatform (KMP)[2], consente l'operatività su reti di dispositivi eterogenei, che spaziano da server con elevate capacità di calcolo (tramite JVM), a dispositivi mobili (An-

[1] https://github.com/Collektive
[2] https://kotlinlang.org/docs/multiplatform.html

droid e iOS), offrendo inoltre la possibilità di eseguire nativamente applicazioni su piattaforme quali Windows, MacOS e Linux, sia su architetture x86 che ARM.

A differenza di ScaFi, il compilatore di Kotlin è stato esteso tramite un apposito plugin per gestire automaticamente l'allineamento del codice aggregato.

Se si volesse gestire manualmente l'allineamento, sarebbe sufficiente adottare un semplice DSL; tuttavia, il codice risulterebbe esposto a meccanismi di basso livello, rendendo la scrittura dei programmi aggregati più complessa e soggetta ad errori (Listing 1.6).

Listing 1.6: Come apparirebbe un programma aggregato in Collektive se l'allineamento fosse gestito manualmente dallo sviluppatore invece che automaticamente dal compilatore.

```
fun <ID: Any> Aggregate<ID>.distanceTo(source: Boolean, metric: Field<ID, Double>)
    = alignedOn("Aggregate.distanceTo(Boolean)") { // Avoid clashing with other
        functions with a similar structure
        share(Double.POSITIVE_INFINITY) { distances ->
            alignedOn("share(Boolean)") { // We need to manually align again on
                share operator.
                val actualMetrics = project(metric) // Fields need projection
                val throughNeighbor = distances
                    .alignedMapValues(actualMetrics, Double::plus)
                when {
                    source -> alignedOn(true) { 0.0 } // Align on true
                    else -> alignedOn(false) { throughNeighbor } // Align on false
                }
            }
        }
    }
```

Il plugin del compilatore si occupa di iniettare automaticamente il codice necessario per gestire l'allineamento e la proiezione dei campi di vicinato, permettendo agli sviluppatori di scrivere programmi aggregati in Kotlin in modo naturale.

Listing 1.7: Dimostrazione della sintassi di Collektive: il compilatore si occupa automaticamente di gestire l'allineamento per conto dello sviluppatore.

```
fun <ID: Any> Aggregate<ID>.distanceTo(source: Boolean, metric: Field<ID, Double>)
    = share(Double.POSITIVE_INFINITY) { distances ->
        val throughNeighbor = distances.alignedMapValues(metric, Double::plus)
        if (source) 0.0 else throughNeighbor
    }
```

1.3 Obiettivo

Quando emerge un nuovo linguaggio o framework, è fondamentale che gli sviluppatori possano sfruttare le conoscenze acquisite con le tecnologie precedenti. I sistemi moderni, quindi, devono offrire una guida chiara su come realizzare soluzioni già presenti. Senza questa guida e senza esempi pratici, l'adozione di nuove tecnologie può essere rallentata dalla curva di apprendimento e dalla scarsa familiarità.

Fornire esempi concreti di trasposizione di sistemi legacy a nuove soluzioni dimostra l'espressività del nuovo sistema rispetto ai precedenti, riduce significativamente il tempo di apprendimento e accresce la fiducia nella nuova tecnologia.

Questo lavoro si propone di trasporre una serie di esempi di programmazione aggregata (Capitolo 4) realizzati originariamente in Proto (Capitolo 2) al linguaggio Collektive (Capitolo 3).

Capitolo 2

MIT-Proto

In questo capitolo viene presentata la sintassi e i costrutti principali di Proto, al fine di fornire le basi necessarie per comprendere il capitolo 4.

Per ulteriori dettagli sul linguaggio, si rimanda alla documentazione ufficiale[1], da cui sono tratte le sezioni seguenti.

2.1 Introduzione alla sintassi

Valutazione delle espressioni Proto è un linguaggio puramente funzionale caratterizzato dall'utilizzo di s-expression (sexp) (Listing 2.1), ovvero espressioni in notazione polacca racchiuse tra parentesi, adottando una sintassi molto simile a quella di Scheme[2].

Listing 2.1: Le espressioni in Proto sono scritte in notazione polacca, specificando l'operatore seguito dagli operandi.

```
1 (+ (* 1 2) 2) ; => 1 * 2 + 2
```

La valutazione di un'espressione Proto produce un programma rappresentato sotto forma di *dataflow graph*.

[1] https://github.com/jakebeal/MIT-Proto/blob/master/proto/man/proto-language-reference.pdf

[2] https://www.scheme.org/

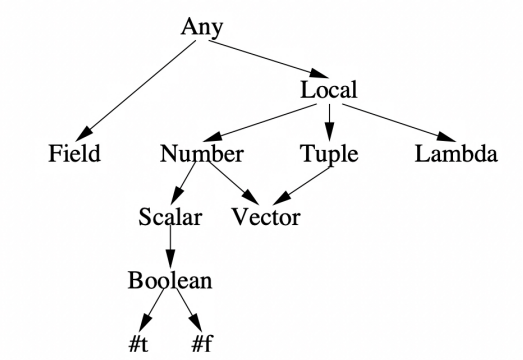

Figura 2.1: Gerarchia dei tipi di dato in Proto.

Tale grafo, quando valutato rispetto a uno spazio di dispositivi, genera un campo computazionale in evoluzione, associando a ogni punto nello spazio una serie di valori che si modificano nel tempo.

Questa architettura consente di esprimere il comportamento collettivo del sistema attraverso trasformazioni funzionali composte, in cui il risultato di un'espressione rappresenta un campo che può essere utilizzato come input per successive trasformazioni.

Tipi di dato Tutte le espressioni producono campi, categorizzati nel sistema di tipi in fig. 2.1:

- `Any`: Radice della gerarchia dei tipi.
- `Field`: Rappresenta un campo di vicinato ϕ, associando a ogni dispositivo un valore locale (`Local`).
- `Local`: Rappresenta qualsiasi valore che non sia un `Field`. I sottotipi di `Local` includono:
  - `Tuple`: Insieme ordinato di valori locali (`Local`).
  - `Scalar`: Numero a virgola mobile, inclusi i valori speciali `nan`, `inf` e `-inf`.
  - `Vector`: Tupla (`Tuple`) di valori scalari (`Scalar`).
  - `Number`: Valore scalare (`Scalar`) o vettoriale (`Vector`).
  - `Boolean`: Valore booleano true (`#t`) o false (`#f`).
  - `Lambda`: Funzione anonima.

Nel seguito, i tipi di dato saranno indicati con la prima lettera maiuscola: ad esempio, una variabile `x` di tipo `Local` sarà denotata come `x|L`, mentre `y` di tipo `Field` come `y|F`.

Dichiarazione di funzioni e variabili Le funzioni vengono dichiarate utilizzando la forma `(def` *nome* `(`*parametri*`) (`*corpo*`))`, dove *nome* è il nome della funzione, *parametri* è una lista di parametri formali e *corpo* è l'espressione che definisce il comportamento della funzione. Il tipo di ritorno della funzione è dedotto automaticamente in base al tipo dell'espressione *corpo*.

Le variabili vengono dichiarate tramite la parola chiave `let`, seguendo la sintassi `(let (`*nome* `(`*valore*`)))`, dove *nome* è il nome della variabile, *valore* è l'espressione che ne determina il valore.

Se si desidera dichiarare più variabili contemporaneamente, si possono racchiudere più coppie `(`*nome* `(`*valore*`))` dentro il corpo di `let`. Se per definire le variabili successive si fa riferimento a quelle precedentemente dichiarate, dando importanza all'ordine di dichiarazione, si utilizza la parola chiave `let*`.

2.2 Operatori

Proto fornisce un insieme di operatori che corrispondono ai costrutti fondamentali di FC (sezione 1.1.2).

Dinamica temporale Per modellare lo stato di un dispositivo si sfrutta l'operatore `rep`, che segue la sintassi:

Listing 2.2: Sintassi dell'operatore `rep` in Proto.

```
(rep .var ,init|L ,evolve|L) -> L
; example
(rep t 0 (+ t (1))) ; incrementing t each round.
```

- `.var` è il nome della variabile di stato locale.
- `init` è l'espressione che inizializza il valore della variabile di stato.
- `evolve` è l'espressione che definisce come il valore della variabile di stato evolve nel tempo.
- Il tipo di ritorno dell'operatore è lo stesso tipo dell'espressione `init` ed `evolve`.

L'esempio in listing 2.2 illustra l'uso di `rep` per creare un contatore che incrementa il proprio valore ad ogni round di esecuzione.

Interazione Spaziale L'operatore `nbr` consente a un dispositivo di accedere ai valori calcolati dai vicini nell'ultimo round di esecuzione. La sintassi è la seguente:

Listing 2.3: Sintassi dell'operatore `nbr` in Proto.

```
(nbr ,expr|L) -> F
;example
nbr 1 ; map each neighboor to 1
```

- `expr` è l'espressione il cui valore locale viene raccolto dai vicini.
- Il tipo di ritorno dell'operatore è un `Field` che mappa l'identificatore di ogni vicino al valore locale risultante dalla valutazione di `expr` in quel dispositivo.

In listing 2.3 viene mostrato un esempio di utilizzo di `nbr` nel quale si costruice un campo di vicinato che associa 1 ad ogni vicino.

Proto fornisce varianti dell'operatore `nbr` che permettono di raccogliere specifici valori dai vicini come `nbr-range`, per ottenere la distanza fisica tra il dispositivo corrente e ciascun vicino, e `nbr-vec`, per ottenere un vettore che indica la direzione verso ciascun vicino rispetto al dispositivo corrente.

Vengono fornite delle funzioni di aggregazione che permettono di ridurre un `Field` a un singolo valore locale (`Local`) (Listing 2.4).

Listing 2.4: Esempi di funzioni di aggregazione spaziale in Proto.

```
; returns the lower limit of values in the range of expr.
(min-hood ,expr|F_n) -> N
; returns the upper limit of values in the range of expr.
(max-hood ,expr|F_n) -> N
; returns false if the range of expr includes false; otherwise returns true.
(all-hood ,expr|F_b) -> B
; returns true if the range of expr includes true; otherwise returns false.
(any-hood ,expr|F_b) -> B
; returns the sum of expr over all devices in the neighborhood.
(sum-hood ,expr|F_n) -> N
; returns the integral of expr over the neighborhood.
(int-hood ,expr|F_n) -> N
```

Queste funzioni si basano su `fold-hood` (Listing 2.5), un operatore più generale che consente di specificare una funzione di combinazione personalizzata.

Listing 2.5: Sintassi dell'operatore `fold-hood` in Proto.

```
(fold-hood ,fold|Lambda ,base|L ,value|L) -> L
; example
(fold-hood + 0 1) ;counts the number of neighbors. Equals to (sum-hood (nbr 1))
```

Si noti che `fold-hood` non richiede in ingresso un `Field`, bensì `value` è un valore locale (`Local`). Internamente, `fold-hood` utilizza l'operatore `nbr` per raccogliere i valori dai vicini in modo trasparente al programmatore e applica la funzione di combinazione specificata per aggregarli.

Restrizione del dominio L'operatore `if` partiziona la rete dei dispositivi in due sottoinsiemi in base alla valutazione di una condizione. La sintassi è la seguente (Listing 2.6):

Listing 2.6: Sintassi dell'operatore `if` in Proto.

```
(if ,test|B ,true ,false) -> A
; example
; counts the neighbhors which have followed the same path.
(if (mod mid 2) (sum-hood (nbr 1)) (sum-hood (nbr 1)))
```

- `test` è l'espressione booleana che determina la partizione dello spazio.
- `true` è l'espressione che viene valutata nei dispositivi per cui `test` è vera.
- `false` è l'espressione che viene valutata nei dispositivi per cui `test` è falsa.
- Il tipo di ritorno dell'operatore è lo stesso di `true` e `false`.

Anche se `true` e `false` dovessero risultare la stessa espressione, il loro valore potrebbe differire, in accordo a quanto detto in sezione 1.1.2.

È possibile utilizzare l'operatore `mux` come alternativa a `if` quando si desidera che le due espressioni `true` e `false` vengano valutate prima del partizionamento, garantendo la partecipazione di tutti i dispositivi alla computazione (Sezione 1.1.2).

2.3 Funzioni di libreria

Proto è equipaggiato con una libreria che fornisce diverse funzioni comuni durante lo sviluppo di programmi aggregati.

- `(distance-to ,source|B) -> S`: calcola la distanza minima dalla sorgente per ogni dispositivo. I nodi in cui `source` è `#t` fungono da origine, mentre gli altri ricevono la distanza dalla sorgente più vicina. `gradient` è un alias di questa funzione.
- `(broadcast ,source|B ,value|L) -> L`: propaga il valore locale `value` dai dispositivi sorgente verso l'intera rete. Ogni dispositivo riceve il valore dalla sorgente più vicina.
- `(dilate ,source|B ,d|S) -> B`: Verifica quali dispositivi si trovano entro una distanza `d` dalla sorgente più vicina.

- `(distance ,r1|B ,r2|B) -> S`: calcola la distanza tra le due regioni `r1` e `r2` e la propaga in tutta la rete.
- `(timer) -> S`: restituisce il tempo trascorso durante la valutazione dell'espressione nel dispositivo corrente.

Inoltre sono fornite funzioni per operare con i tipi di dato elencati in fig. 2.1, come operazioni aritmetiche, logiche, di confronto, manipolazione di tuple e di vettori, nonché funzioni per la generazione di numeri casuali.

Listing 2.7: Funzioni di built-in in Proto.

```
; Arithmetic Operations
(+ ,x|N ,y|N ++) -> N          ; Adds two or more numbers
(- ,x|N ,y|N) -> N             ; Subtracts y from x
(* ,x|S ++ ,y|N) -> N          ; Multiplies numbers together
(/ ,x|S ,y|S) -> S             ; Divides x by y
(mod ,num|S ,divisor|S) -> S   ; Returns remainder
(max ,x|N ,y|N) -> N           ; Returns maximum value
(min ,x|N ,y|N) -> N           ; Returns minimum value
(sqrt ,n|S) -> S               ; Returns square root
(abs  ,n|S) -> S               ; Returns absolute value

; Random Operations
(rnd ,min|S ,max|S) -> S       ; Random number between min and max

; Vector Operations
(vdot ,a|V ,b|V) -> S          ; Dot product of vectors
(vlen ,v|V) -> S               ; Length of vector
(normalize ,v|V) -> V          ; Normalizes vector to length 1

; Tuple Operations
(tuple ,v|L ++) -> T           ; Creates tuple from arguments, tup is an alias
(1st ,tuple|T) -> L            ; Returns first element
(2nd ,tuple|T) -> L            ; Returns second element
(3rd ,tuple|T) -> L            ; Returns third element
```

Nel simulatore di Proto, ciascun nodo è dotato di sensori e sonde per interagire con l'ambiente. Le istruzioni `sense` e `probe` permettono rispettivamente la lettura dai sensori e la scrittura sulle sonde. In particolare, `(probe (e1) idx)` invia il valore di `e1` alla sonda `idx` e lo restituisce come risultato.

Capitolo 3

Collektive

Collektive si articola in tre componenti indipendenti: il DSL (Sezione 3.2), che definisce sintassi e semantica del linguaggio; una libreria standard (Sezione 3.3), che fornisce funzionalità comuni per la programmazione aggregata; un plugin per il compilatore, che gestisce automaticamente l'allineamento e la comunicazione.

Progettato per essere multipiattaforma grazie al supporto del progetto KMP, il linguaggio consente di eseguire il codice su diverse piattaforme (JVM, JavaScript e nativo) con modifiche minime. L'intero processo di build è gestito tramite Gradle[1].

Collektive è inoltre integrato con Alchemist [PMV13], un simulatore che permette di sviluppare, testare e validare programmi aggregati in ambienti controllati, prima del loro deployment sui dispositivi reali.

Il linguaggio supporta XC (Sezione 1.1.3), consentendo agli sviluppatori di sfruttare l'operatore `exchange` per modellare comunicazioni anisotropiche tra dispositivi.

[1] `https://gradle.com/`

3.1 Field

In Collektive, un `Field` è una struttura dati che associa a ciascun dispositivo del vicinato un valore. A differenza del campo di vicinato (sezione 1.1.2), tiene traccia anche del dispositivo locale.

Ad esempio, per un dispositivo con identificativo 0 e vicini 1, 2 e 3, che assegna il valore 1 a ciascuno (Listing 3.1), la rappresentazione è:

$$\phi(\texttt{localId=0, localValue=1, neighbors=\{1=1, 2=1, 3=1\}})$$

dove `localId` è l'identificatore del dispositivo locale; `localValue` è il valore associato al dispositivo corrente; `neighbors` è una mappa che associa gli identificatori dei vicini ai loro rispettivi valori.

Listing 3.1: Creazione di un campo di vicinato che associa il valore 1 a ogni dispositivo.

```
// phi(localId=0, localValue=1, neighbors={1=1, 2=1, 3=1})
val f: Field<Int, Int> = mapNeighborhood { 1 }
// A collapsing view over the field entries
// that includes only neighbors.
val onlyNeighbors = f.neighbors
// A collapsing view over the field entries
// that includes the local entry and all neighbors.
val all = f.all
```

Le proprietà `all` e `neighbors` (Listing 3.1) permettono di ottenere una vista sulle entry del `Field`: la prima include il dispositivo locale, mentre la seconda restituisce solo i vicini. Forniscono inoltre metodi per convertire la vista in diverse strutture dati native di Kotlin, come `list`, `set`, `sequence` e `map` (3.1).

```
with(mapNeighborhood { 1 }.all) {
  val l = list // to a Kotlin's list of field entries (deviceId, value)
  val s = set // to a Kotlin' set of field entries (deviceId, value)
  val seq = sequence // to a Kotlin's sequence of field entries (deviceId, value)
  val map = toMap() // to a Kotlin's map, where each entry is (deviceId -> value)
}
```

I `Field` possono essere manipolati sfruttando `map/mapValues` e combinati tramite `alignedMap/alignedMapValues` (Listing 3.2).

Listing 3.2: Esempi di manipolazione e combinazione di `Field` in Collektive.

```
val f = mapNeighborhood { 1 }
val f1 = mapNeighborhood { 2 }
val f2 = mapNeighborhood { 3 }

/* Manipulation */
// phi(localId=0, localValue=10, neighbors={1=2, 2=10, 3=2})
f.map { (id, value) -> if (id % 2 == 0) value * 10 else value + 1 }
// phi(localId=0, localValue=2, neighbors={1=2, 2=2, 3=2})
f.mapValues { value -> value + 1 }

/* Combination */
// phi(localId=0, localValue=5, neighbors={1=6, 2=5, 3=6})
f2.alignedMap(f3) { (id, v1), (_, v2) -> if (id % 2 == 0) v1 + v2 else v1 * v2 }
// phi(localId=0, localValue=5, neighbors={1=5, 2=5, 3=5})
f2.alignedMapValues(f3) { v1, v2 -> v1 + v2 }
```

Data la verbosità di queste operazioni, Collektive fornisce operatori più concisi per la loro combinazione (Listing 3.3).

Listing 3.3: Operatori per la combinazione di `Field` in Collektive.

```
val f2 = mapNeighborhood { 2 }
val f3 = mapNeighborhood { 3 }
// phi(localId=0, localValue=5, neighbors={1=5, 2=5, 3=5})
f2 + f3
// phi(localId=0, localValue=-1, neighbors={1=-1, 2=-1, 3=-1})
f2 - f3
// phi(localId=0, localValue=6, neighbors={1=6, 2=6, 3=6})
f2 * f3
```

I `Field` supportano anche operazioni di riduzione, come `fold`, e possono accedere alle funzioni di aggregazione standard di Kotlin una volta convertiti in strutture dati native (Listing 3.4).

Listing 3.4: Esempi di riduzione di `Field` in Collektive.

```
val f = mapNeighborhood { 1 }
// Counting the number of devices in the neighboorhood (including self)
val count = f
  .all
  .fold(0) { acc, _ -> acc + 1 }
```

3.2 Operatori

Collektive mette a disposizione l'interfaccia `Aggregate`, che definisce il set minimo di operazioni fondamentali per la programmazione aggregata. Tali operazioni sono accessibili sia come metodi dell'interfaccia sia tramite extension functions. L'interfaccia mantiene inoltre l'identificativo locale (`localId`) del dispositivo che esegue il programma aggregato. Per utilizzare i costrutti di programmazione aggregata, i programmi devono estendere `Aggregate`.

I nomi degli operatori sono stati scelti privilegiando una sintassi più adatta a Kotlin, piuttosto che i termini usati in letteratura. In particolare, `rep` è stato rinominato in `evolve` e `nbr` in `neighboring`.

Exchange L'operatore `exchange` (Listing 3.5) è il cuore del DSL: consente a ciascun dispositivo di inviare valori differenti ai propri vicini, permettendo così interazioni personalizzate e modellando l'evoluzione spazio-temporale del sistema tramite una comunicazione anisotropica.

Listing 3.5: Firma dell'operatore `exchange` in Collektive.

```
inline fun <ID : Any, reified Shared> Aggregate<ID>.exchange(
    initial: Shared,
    noinline body: (Field<ID, Shared>) -> Field<ID, Shared>,
): Field<ID, Shared>
```

A partire da un valore iniziale (`initial`), che funge da default quando non sono ancora stati ricevuti messaggi dai vicini, `exchange` costruisce un campo contenente i messaggi ricevuti nell'ultimo round. Su questo viene applicata una funzione (`body`) che restituisce un nuovo campo, i cui valori vengono inviati come messaggi personalizzati ai rispettivi vicini al termine del round corrente. Questo costrutto, derivato da XC (Sezione 1.1.3), permette di esprimere tutti gli altri operatori di FC. Tuttavia, non tutti i costrutti ne richiedono l'utilizzo: ad esempio, `evolve` si occupa esclusivamente dell'evoluzione temporale locale, senza comunicazione con i vicini, e per questo non viene implementato tramite `exchange`, poiché risulterebbe inefficiente.

Un esempio di utilizzo dell'operatore `exchange` è riportato in listing 3.6: partendo dal valore 1, ogni dispositivo riceve i valori dai vicini e, in base al loro

identificativo, invia a ciascuno il valore ricevuto incrementato di 1 oppure il suo doppio. Il valore locale viene aggiornato secondo la stessa logica, tenendo presente che il dispositivo corrente è incluso in `field`.

Listing 3.6: Esempio di utilizzo dell'operatore `exchange` in Collektive.

```
// when the device ID is even, send value + 1; when odd, send value * 2
exchange(1) { field ->
    field.map { id, value ->
        if (id % 2 == 0) value + 1 else value * 2
    }
}
```

Share L'operatore `share` (Listing 3.7) modella l'evoluzione spazio-temporale del dispositivo, consentendogli di condividere la stessa informazione con tutti i vicini.

Listing 3.7: Firma dell'operatore `share` in Collektive.

```
inline fun <ID : Any, reified Shared> Aggregate<ID>.share(
    initial: Shared,
    noinline body: (Field<ID, Shared>) -> Shared,
): Shared
```

A partire da un valore iniziale (`initial`), `share` costruisce un `Field` di valori condivisi che viene elaborato dalla funzione `body`. Il risultato viene memorizzato internamente e inviato a tutti i vicini; ad ogni round, l'operatore restituisce lo stato aggiornato in base ai messaggi ricevuti. Internamente, è implementato tramite `exchange` (Sezione 3.2).

Un esempio di utilizzo di `share` è riportato in listing 4.8, dove ogni dispositivo calcola la propria temperatura in base a quella dei vicini e, una volta ottenuto il risultato, lo condivide con tutti i nodi adiacenti.

Neighboring l'operatore `neighboring` (Listing 3.8), corrispondente al costrutto `nbr` di FC (Sezione 1.1.2), consente a un dispositivo di accedere ai valori calcolati dai vicini nell'ultimo round di esecuzione.

Listing 3.8: Firma dell'operatore `neighboring` in Collektive.

```
inline fun <ID : Any, reified Shared> Aggregate<ID>.neighboring(local: Shared):
    Field<ID, Shared>
```

Restituisce un `Field` che mappa ogni vicino al valore locale `local`.

Per minimizzare la comunicazione, si usano le varianti `neighborhood` e `mapNeighborhood`. La prima restituisce un `Field` con i soli identificatori dei vicini, mentre la seconda consente di mappare ciascun vicino ad un valore calcolato dal dispositivo corrente.

In listing 3.9 si illustra la differenza tra i tre operatori, considerando un dispositivo con tre vicini identificati da 1, 2 e 3. L'operatore `neighboring` crea un `Field` in cui a ciascun nodo δ_i è associato il proprio valore locale $\mathtt{x}_i$. Con `neighborhood` si ottiene, invece, un `Field` che associa a ogni vicino il valore 0, utile per identificare i dispositivi adiacenti senza scambiare informazioni aggiuntive. Infine, `mapNeighborhood` produce un `Field` che associa a ciascun vicino δ_i il valore $\mathtt{x}_0$, cioè il valore locale del dispositivo corrente.

Listing 3.9: Varianti dell'operatore `neighboring` in Collektive.

```
val x = localComputation()
// phi(localId=0, localValue=x_0, neighbors={1=x_1, 2=x_2, 3=x_3})
val f = neighboring(x);
// phi(localId=0, localValue=0, neighbors={1=0, 2=0, 3=0})
val f1 = neighborhood();
// phi(localId=0, localValue=x_0, neighbors={1=x_0, 2=x_0, 3=x_0})
val f2 = mapNeighborhood { x };
```

Evolve L'operatore `evolve` (Listing 3.10) corrisponde al costrutto `rep` di FC (Sezione 1.1.2) e consente di modellare l'evoluzione temporale dello stato di un dispositivo.

Listing 3.10: Firma dell'operatore `evolve` in Collektive.

```
fun <Stored> evolve(initial: Stored, transform: (Stored) -> Stored): Stored
```

A partire da un valore iniziale (`initial`), a ogni round di esecuzione ciascun dispositivo applica la funzione `transform` al valore corrente per determinare il valore del round successivo.

In listing 3.11 si mostra come utilizzare l'operatore `evolve` per creare un campo che associa ad ogni dispositivo un contatore che incrementa ad ogni round.

Listing 3.11: Esempio di utilizzo dell'operatore `evolve` in Collektive.

```
val round = evolve(0) { it + 1 }
```

3.3 Standard Library

Collektive dispone di una libreria standard, sviluppata a partire dai costrutti del DSL, che mette a disposizione funzionalità comuni utili alla realizzazione di programmi aggregati.

In questa sezione verranno presentate solamente le funzionalità utilizzate nel capitolo 4; per un elenco completo si rimanda alla documentazione ufficiale[2].

Spreading Questo package offre funzioni per la propagazione di informazioni all'interno della rete di dispositivi.

Tra le principali, troviamo `gradientCast`, `multiGradientCast` e `distanceTo`. La funzione `gradientCast` (Listing 3.12) permette di diffondere un valore locale da una sorgente a tutta la rete, assegnando a ciascun dispositivo il valore proveniente dalla sorgente più vicina. `multiGradientCast` (Listing 3.13) amplia questa funzionalità, consentendo a ogni dispositivo di ottenere una mappa dei valori propagati da tutte le sorgenti. Infine, `distanceTo` (Listing 3.14) calcola la distanza minima dalla sorgente più vicina.

Listing 3.12: Firma della funzione `gradientCast` dalla libreria standard di Collektive

```
/**
 * Computes a fast, self-healing gradient broadcast of local values
 * from all source nodes, always retaining the
 * data from the nearest source.
 */
inline fun <reified ID : Any, reified Type> Aggregate<ID>.gradientCast(
    source: Boolean,
    local: Type,
    metric: Field<ID, Double>,
    maxDiameter: Int = Int.MAX_VALUE,
    noinline accumulateData: (fromSource: Double, toNeighbor: Double, data: Type)
        -> Type = { _, _, data -> data },
    crossinline accumulateDistance: Reducer<Double> = Double::plus,
): Type
```

[2] https://javadoc.io/doc/it.unibo.collektive/collektive-stdlib/latest/index.html

Listing 3.13: Firma della funzione `multiGradientCast` dalla libreria standard di Collektive

```
/**
 * For each ID in sources, propagates data from that source using
 * a fast-repair integer gradient, and collects the results in a map
 * from source ID to propagated value.
 */
inline fun <reified ID : Any, reified Value> Aggregate<ID>.multiGradientCast(
    sources: Iterable<ID>,
    local: Value,
    metric: Field<ID, Double>,
    maxDiameter: Int = Int.MAX_VALUE,
    noinline accumulateData: (fromSource: Double, toNeighbor: Double, data: Value)
        -> Value = { _, _, data -> data },
): Map<ID, Value>
```

Listing 3.14: Firma della funzione `distanceTo` dalla libreria standard di Collektive

```
/**
 * For each ID in sources, propagates data from that source using
 * a fast-repair integer gradient, and collects the results in a map
 * from source ID to propagated value.
 */
inline fun <reified ID : Any, reified Value> Aggregate<ID>.multiGradientCast(
    sources: Iterable<ID>,
    local: Value,
    metric: Field<ID, Double>,
    maxDiameter: Int = Int.MAX_VALUE,
    noinline accumulateData: (fromSource: Double, toNeighbor: Double, data: Value)
        -> Value = { _, _, data -> data },
): Map<ID, Value>
```

Accumulation Questo package offre funzioni per l'accumulo di valori su specifici nodi della rete.

La funzione `convergeCast` (Listing 3.15) permette di aggregare un campo computazionale verso una destinazione, seguendo la discesa del gradiente di un campo potenziale fornito e applicando una funzione di accumulo a ciascun nodo attraversato. La funzione `countDevices` (Listing 3.16), basata su `convergeCast`, conta il numero di dispositivi presenti nella rete, aggregando i valori verso il nodo destinazione (`sink`).

Listing 3.15: Firma della funzione `convergeCast` dalla libreria standard di Collektive

```
/**
 * Aggregates a field of local values along a spanning tree built by descending the provided potential field.
 * The parent of the current node is selected by picking the minimum as provided by the [selectParent] comparator,
 * which by default selects the parent with the lowest potential. Data is accumulated using the [accumulateData] function.
 */
inline fun <reified ID : Any, reified Data, reified Potential : Comparable<Potential>> Aggregate<ID>.convergeCast(
    local: Data,
    potential: Potential,
    selectParent: Comparator<FieldEntry<ID, Potential>> = defaultComparator(),
    crossinline accumulateData: (Data, Data) -> Data,
): Data

/**
 * Aggregate a field of local into the closest sink
 * along a spanning tree built using hopDistanceTo.
 * Data is accumulated using the accumulateData function.
 */
inline fun <ID : Any, Data> Aggregate<ID>.convergeCast(
    local: Data,
    sink: Boolean,
    crossinline accumulateData: (Data, Data) -> Data
): Data
```

Listing 3.16: Firma della funzione `countDevices` dalla libreria standard di Collektive

```
/**
 * Counts the number of devices in the network
 * by aggregating ‘1‘ from each device towards the sink.
 */
inline fun <ID : Any> Aggregate<ID>.countDevices(sink: Boolean): Int
```

Collapse Questo package mette a disposizione funzioni di riduzione per ottenere un singolo valore a partire da un `Field`.

`all`, `any` e `countMatching` (Listing 3.17) valutano un predicato su tutti i valori del `Field`: `all` restituisce `true` se tutti i valori lo soddisfano, `any` restituisce `true` se almeno uno lo soddisfa, mentre `countMatching` restituisce il numero di valori che soddisfano il predicato.

`fold` e `reduce` (Listing 3.18) riducono un `Field` a un singolo valore sfruttando una funzione di accumulo; la differenza è che `fold` consente di specificare un valore iniziale, mentre `reduce` utilizza il primo elemento come punto di partenza e non considera il dispositivo locale.

Infine, sono disponibili un insieme di varianti delle funzioni `min` e `max` per ottenere rispettivamente il valore minimo e massimo all'interno di un `Field`.

Listing 3.17: Firme delle funzioni `all`, `any` e `countMatching` dalla libreria standard di Collektive

```
/**
 * Returns true if all elements in the collapsed field
 * satisfy the given predicate.
 */
inline fun <T> Collapse<T>.all(
    crossinline predicate: Predicate<T>
): Boolean

/**
 * Returns true if any element in the collapsed field
 * satisfies the given predicate.
 */
inline fun <T> Collapse<T>.any(
    crossinline predicate: Predicate<T>
): Boolean

/**
 * Counts how many elements in the collapsed field
 * satisfy the given predicate.
 */
inline fun <T> Collapse<T>.countMatching(
    crossinline predicate: Predicate<T>
): Int
```

Listing 3.18: Firme delle funzioni `fold` e `reduce` dalla libreria standard di Collektive

```
/**
 * Folds the collapsed field into a single value,
 * starting from initial and combining elements
 * with the provided accumulator.
 */
inline fun <Destination, T> Collapse<T>.fold(
    initial: Destination,
    crossinline accumulator: Accumulator<Destination, T>
): Destination

/**
 * Reduces the elements in this collapse
 * (which excludes the local element, i.e., only peers)
 * into a single value by repeatedly applying reducer.
 */
inline fun <T : Any> CollapseNeighbors<T>.reduce(
    crossinline reducer: Reducer<T>
): T?
```

Capitolo 4

Esempi Trasposti in Collektive

In questo capitolo si presenta la trasposizione di una selezione di programmi aggregati, originariamente sviluppati in Proto (Capitolo 2), nel linguaggio Collektive (Capitolo 3). Per ciascun esempio viene illustrato il codice sorgente originale, una descrizione dettagliata del funzionamento, la relativa implementazione in Collektive e le immagini delle simulazioni realizzate.

Gli esempi selezionati comprendono: calcolo della media locale (Sezione 4.1), verifica dell'appartenenza a una regione circolare (Sezione 4.2), identificazione dei dispositivi lungo un anello (Sezione 4.3), diffusione della temperatura (Sezione 4.4), navigazione lungo la discesa del gradiente (Sezione 4.5), partizionamento di Voronoi (Sezione 4.6), connessione lungo l'albero dei cammini minimi (Sezione 4.7), tracciamento di un dispositivo (Sezione 4.8) e dinamica di uno stormo (Sezione 4.9).

Il codice sviluppato per questo lavoro è reso disponibile pubblicamente nel repository ufficiale degli esempi di Collektive[1].

[1] `https://github.com/Collektive/collektive-examples`

4.1 Media Locale

Obiettivo Calcolare la media dei vettori nel vicinato.

Codice originale La funzione `local-average` (Listing 4.1) accetta come parametro il vettore locale `v` del dispositivo, con $\texttt{v} \in \mathbb{R}^3$.

Listing 4.1: Codice della funzione `local-average` in Proto.

```
(def local-average (v)
  (* (/ 1 (sum-hood 1)) (fold-hood + (tup 0 0 0) v)))
```

L'espressione complessiva è definita come il prodotto (`*`) di due termini distinti. Il termine di destra, `(fold-hood + (tup 0 0 0) v)`, calcola la somma dei vettori locali presenti nel vicinato. Il termine di sinistra, `(/ 1 (sum-hood 1))`, si occupa invece del conteggio dei vicini, incluso se stesso, e del calcolo del suo inverso. Il risultato del programma è quindi formalizzato come

$$\texttt{local-average(v)} = \frac{1}{\sum_{d \in N} 1} \cdot \sum_{d \in N} v(d)$$

, dove N rappresenta l'insieme dei dispositivi nel vicinato.

Trasposizione in Collektive (Listing 4.2) La raccolta dei vettori nel vicinato avviene tramite il costrutto `neighboring` (3.8); la loro somma viene calcolata con la funzione `fold` (3.18) e, infine, la media si ottiene dividendo tale somma per il numero di dispositivi, determinato tramite la proprietà `size`.

Listing 4.2: Trasposizione di `local-average` in Collektive.

```
/**
 * Computes the local average of vectors in the neighborhood of the current device
 */
fun Aggregate<Int>.localAverage(v: Point3D): Point3D = with(neighboring(v).all) {
    fold(vectorZero) { acc, nbr -> acc + nbr.value } / size.toDouble()
}
```

In fig. 4.1 si illustra la simulazione dell'esperimento, dove ogni nodo è etichettato con la norma euclidea del vettore risultante.

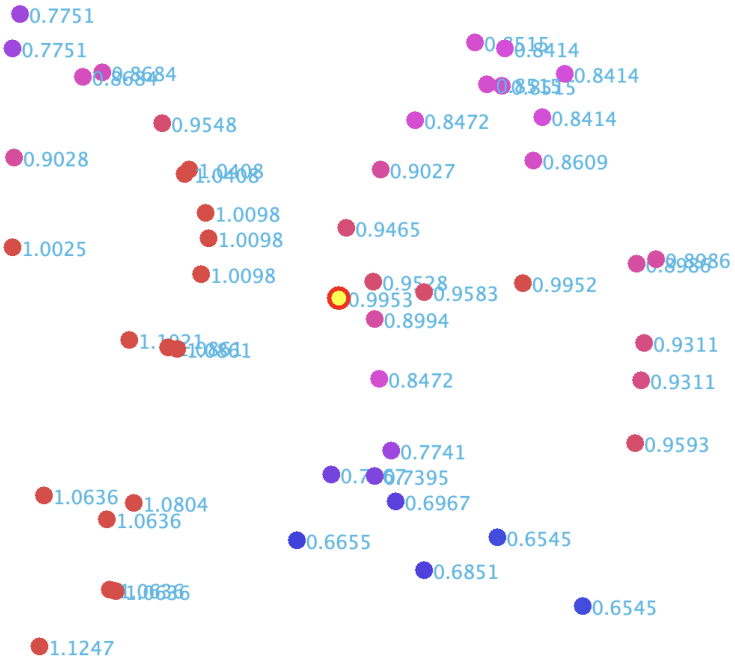

Figura 4.1: Simulazione dell'esperimento in listing 4.2. I dispositivi inizializzano `v` casualmente e convergono verso il valore medio locale. Il colore dei nodi dipende dalla norma euclidea del vettore calcolato, la cui intensità è indicata a fianco.

4.2 Nel Cerchio

Obiettivo Determinare se un dispositivo appartiene a una regione circolare, dati il centro e il raggio del cerchio.

Codice originale La funzione `in-circle` (Listing 4.3) richiede in ingresso le coordinate del centro (`o`) e il raggio (`r`) del cerchio.

Listing 4.3: Codice della funzione `in-circle` in Proto.

```
(def in-circle (o r)
  (let ((dv (- (probe (coord) 1) o)))
    (< (probe (vdot dv dv) 0) (* r r))))
```

L'istruzione `let ((dv (- (probe (coord) 1) o)))` calcola la differenza tra le coordinate del dispositivo corrente e quelle del nodo centrale, depositando il risultato nella variabile `dv`. Successivamente, l'espressione `(< (probe (vdot dv dv) 0) (* r r))` verifica che il prodotto scalare di `dv` con sè stesso sia inferiore al quadrato del raggio `r`, restituendo un valore booleano che indica l'appartenenza al cerchio.

Trasposizione in Collektive (Listing 4.4) La propagazione delle coordinate del nodo centrale è realizzata tramite la funzione `gradientCast` (Listing 3.12). In seguito, si verifica che il quadrato della distanza tra il dispositivo corrente e il centro sia inferiore al quadrato del raggio.

In fig. 4.2 è mostrata la simulazione dell'esperimento: il centro `o` è evidenziato in verde, i dispositivi all'interno del cerchio sono colorati in rosso, mentre i dispositivi esterni al cerchio sono rappresentati in nero.

Listing 4.4: Trasposizione della funzione `in-circle` in Collektive.

```
/**
 * Determines if the current device (located in [location])
 * is within a circle of a specified radius from a [center] point.
 */
fun Aggregate<Int>.inCircle(center: Boolean, location: Point2D, metric: () ->
    Field<Int, Double>): Boolean = with(location) {
        val centerPos = gradientCast(center, this, metric())
        distanceToSquared(centerPos) <= RADIUS.pow(2)
    }
```

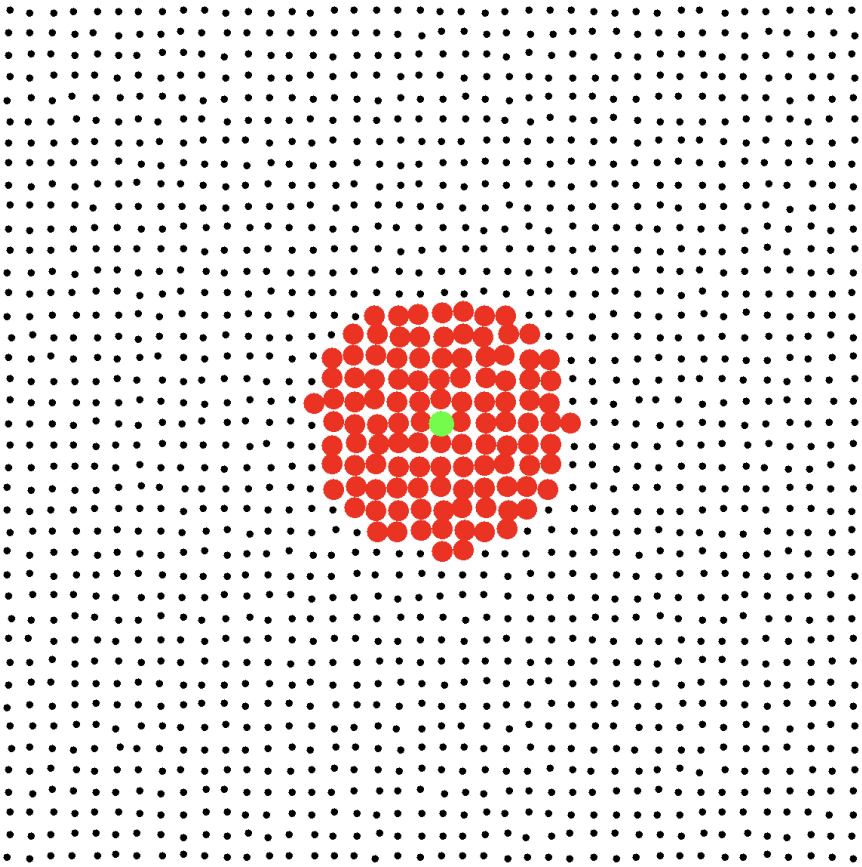

Figura 4.2: Simulazione dell'esperimento in listing 4.4. Il centro `o` è evidenziato in verde, i dispositivi all'interno del cerchio sono colorati in rosso, mentre i dispositivi esterni al cerchio sono rappresentati in nero.

4.3 Anello

Obiettivo Identificare i dispositivi disposti lungo un anello di raggio variabile nel tempo.

Codice originale La funzione `ring` (Listing 4.5) non prevede argomenti in ingresso, ma si avvale della funzione `sense` per individuare i dispositivi che fungono da sorgente del segnale.

Listing 4.5: Codice della funzione `ring` in Proto.

```
(def ring ()
  (let ((t (probe (broadcast (sense 1) (if (sense 1) (timer) 0)) 0))
        (d (probe (gradient (sense 1)) 1)))
    (if (< (abs (- (* 5 t) d)) 4)
      (green 1)
      (blue 1))))
```

L'espressione `(let (...))` si articola in due componenti principali. La prima,

```
((t (probe (broadcast (sense 1) (if (sense 1) (timer) 0)) 0)))
```

, propaga il valore del `timer` dalle sorgenti del segnale: ogni dispositivo riceve il valore dalla sorgente più vicina e lo memorizza nella variabile `t`. La seconda componente, `(d (probe (gradient (sense 1)) 1))`, calcola la distanza dalla sorgente più prossima, assegnandola alla variabile `d`. Infine, l'espressione `(if (< (abs (- (* 5 t) d)) 4) (green 1) (blue 1))` verifica se la distanza `d` si discosta di meno di 4 unità dalla posizione dell'anello, che avanza con una velocità di 5 unità al secondo. Se la condizione è soddisfatta, il dispositivo viene colorato di verde; altrimenti, di blu.

Trasposizione in Collektive (Listing 4.6) La propagazione del valore del timer dalle sorgenti avviene tramite la funzione `gradientCast` (Listing 3.12), mentre la funzione `distanceTo` (Listing 3.14) permette a ciascun dispositivo di calcolare la distanza dalla sorgente più vicina.

Come illustrato in fig. 4.3, l'anello si propaga come un'onda, originata dalla sorgente. Per evitare che la simulazione si concluda subito dopo la prima propaga-

zione, è stata introdotta una dinamica periodica che permette all'anello di formarsi e dissolversi ciclicamente nel tempo.

Listing 4.6: Trasposizione della funzione `ring` in Collektive.

```
/**
 * Create a ring wave pattern originating from the [center] node.
 * The wave propagates outward from the center,
 * with a speed defined by [WAVE_SPEED],
 * a thickness defined by [WAVE_THICKNESS],
 * and a period defined by [WAVE_PERIOD].
 * It returns a boolean field indicating whether the ring
 * is active at each node.
 */
private fun Aggregate<Int>.ring(center: Boolean, currentTime: () -> Double, metric
    : () -> Field<Int, Double>): Boolean =
    run {
        val waveTime = broadcastTime(center, currentTime, metric)
        val distance = distanceTo(center, metric = metric())
        isRingActive(waveTime, distance)
    }

/**
 * Broadcast the current time from the [center] to all
 * other nodes in the network,
 * based on the given [metric].
 */
private fun Aggregate<Int>.broadcastTime(
    center: Boolean,
    currentTime: () -> Double,
    metric: () -> Field<Int, Double>,
): Double = gradientCast(
    source = center,
    local = currentTime(),
    metric = metric(),
)

/**
 * Check if the ring is active at the given [waveTime] and [distance]
 * from the center.
 * A ring is active if the distance from the center
 * is within the wave thickness.
 */
private fun isRingActive(waveTime: Double, distance: Double): Boolean =
    abs(WAVE_SPEED * (waveTime % WAVE_PERIOD) - distance) < WAVE_THICKNESS
```

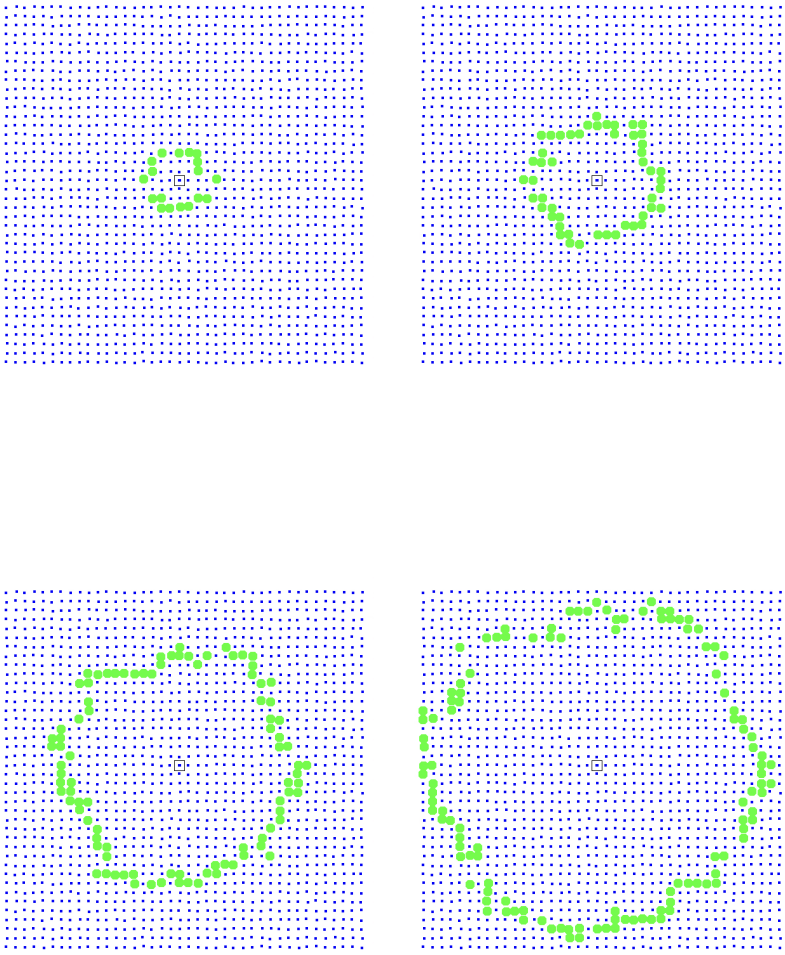

Figura 4.3: Simulazione dell'esperimento in listing 4.6. I dispositivi lungo l'anello sono colorati in verde, mentre gli altri sono colorati in blu. La simulazione è mostrata in quattro istanti temporali distinti, evidenziando il movimento dell'anello nel tempo.

4.4 Temperature

Obiettivo Simulare la diffusione del calore in un'area, considerando la presenza simultanea di sorgenti di calore e di freddo.

Codice originale La funzione `temperature` (Listing 4.7) non accetta argomenti. Un dispositivo è una sorgente di calore se il suo identificatore `mid` è minore di 6 ed è pari, mentre è una sorgente di freddo se `mid` è minore di 6 ma dispari.

Listing 4.7: Codice della funzione `temperature` in Proto.

```
(def temperature ()
  (let ((hot (and (< (mid) 6) (mod (mid) 2)))
  (cold (and (< (mid) 6) (not (mod (mid) 2)))))
    (rep temp 25 (mux hot 30 (mux cold 20 (/ (sum-hood (nbr temp)) (sum-hood 1))))
        )))
```

L'istruzione `( rep temp 25 ( mux hot 30 ( mux cold 20 (/ (sum-hood(nbr temp))(sum-hood 1)))))` modella l'evoluzione del campo di temperatura nello spazio e nel tempo. Inizialmente, ogni dispositivo imposta la propria temperatura a 25 gradi (`rep temp 25 ...`). Tale valore viene aggiornato ad ogni round in base al ruolo del dispositivo: se il nodo agisce come sorgente di calore, viene impostato a 30 gradi (`mux hot 30 ...`); se invece funge da sorgente di freddo, viene fissato a 20 gradi (`mux cold 20 ...`); in tutti gli altri casi, si assume la media delle temperature rilevate nel vicinato (`(/ (sum-hood(nbr temp))(sum-hood 1))`).

Trasposizione in Collektive (Listing 4.8) L'evoluzione spazio-temporale della temperatura è modellata tramite l'operatore `share` (Sezione 3.2).

In fig. 4.4 è riportata una simulazione dell'esperimento, in cui i dispositivi che agiscono come sorgenti di calore sono evidenziati da un rettangolo rosso, mentre quelli che fungono da sorgenti di freddo sono evidenziati da un rettangolo blu. Il colore di ciascun dispositivo rappresenta la temperatura locale, secondo una scala cromatica che va dal nero (freddo) al rosso (caldo).

Listing 4.8: Trasposizione della funzione `temperature` in Collektive.

```
/**
 * Represents an evolving pseudo-temperature field where
 * each device updates its temperature over time.
 * Some devices act as fixed heat sources ([heatSource])
 * at [HEAT_SOURCE_TEMPERATURE], others as fixed cold sources
 * ([coldSource]) at [COLD_SOURCE_TEMPERATURE].
 * All other devices calculate their temperature dynamically as
 * the average temperature of their neighboring devices.
 */
fun Aggregate<Int>.temperature(heatSource: Boolean, coldSource: Boolean): Double =
    share(INITIAL_TEMPERATURE) { previousTemperatures ->
        val averageTemperature = previousTemperatures.all
            .values
            .sequence
            .average().takeIf { it.isFinite() } ?: INITIAL_TEMPERATURE
        when {
            heatSource -> HEAT_SOURCE_TEMPERATURE
            coldSource -> COLD_SOURCE_TEMPERATURE
            else -> averageTemperature
        }
    }
```

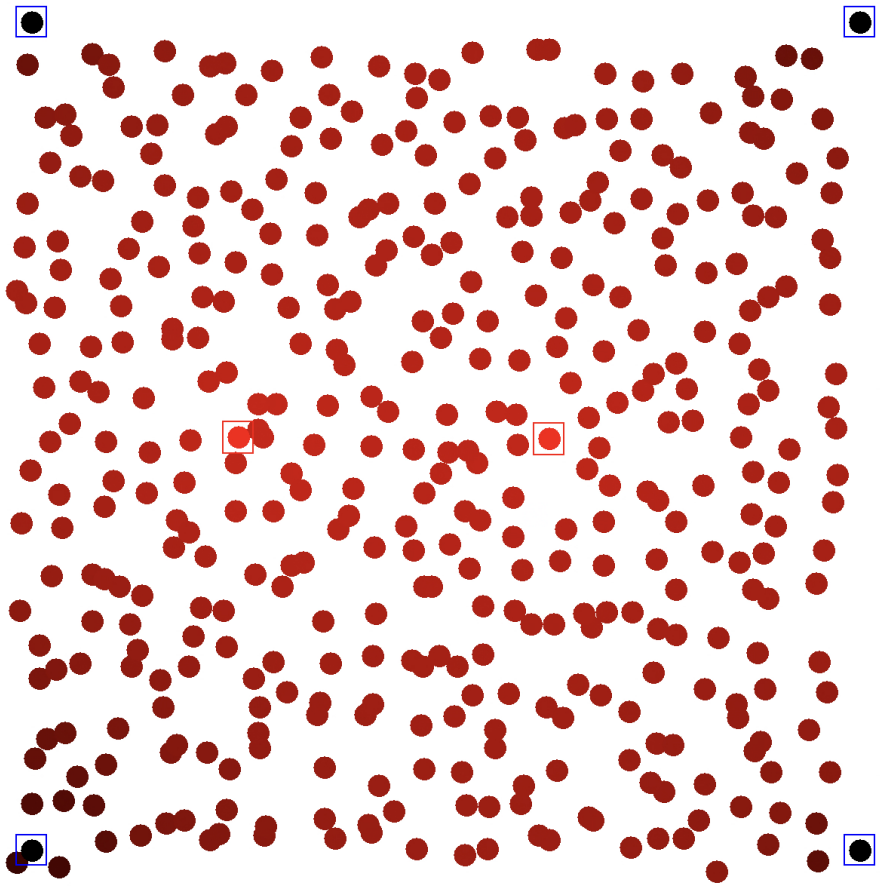

Figura 4.4: Simulazione dell'esperimento in listing 4.8. I dispositivi che agiscono come sorgenti di calore sono evidenziati da un rettangolo rosso, mentre quelli che agiscono come sorgenti di freddo sono evidenziati da un rettangolo blu. Il colore dei dispositivi rappresenta la propria temperatura, con una scala che va dal nero (freddo) al rosso (caldo).

4.5 Navigazione del Gradiente

Obiettivo Guidare i nodi mobili verso una fonte di interesse all'interno della rete, muovendosi lungo la discesa del gradiente del campo delle distanze.

Codice originale Il programma (Listing 4.9) è composto da tre funzioni: `grad`, `share-distance-to` e `nav-grad`.

Listing 4.9: Codice Proto per l'esperimento di navigazione del gradiente, comprendente le funzioni `grad`, `share-distance-to` e `nav-grad`.

```
(def grad (v)
  (* (/ 1 (int-hood 1)) ; normalize over neighborhood
     (int-hood (if (or (= (nbr-range) 0) (not (< (abs (- v (nbr v))) (inf))))
                   (tup 0 0 0) ; ignore singularity
                   (* (/ (- v (nbr v)) (nbr-range))
                      (normalize (nbr-vec)))))))

(def share-distance-to (is-calculating source)
  (let ((base (if is-calculating (distance-to source) (inf))))
    (green (< base (inf)))
    (mux is-calculating base (min-hood (+ (nbr-range) (nbr base))))))

(def nav-grad (is-mover source)
  (let ((g (grad (share-distance-to (not is-mover) source))))
    (mux (and is-mover (> (len g) 0)) (normalize g) (tup 0 0))))
```

`grad` calcola il gradiente di un campo di valori scalari `v`, restituendo un vettore che indica la direzione del massimo decremento di quel campo. L'espressione più esterna consiste in una moltiplicazione (`*`) tra un fattore di normalizzazione e l'integrale che aggrega i contributi dei vicini al gradiente. Il fattore di normalizzazione, definito come `(/ 1 (int-hood 1))`, calcola il reciproco del numero di vicini (pesati) considerati nell'integrale. Il contributo di ciascun nodo viene determinato mediante l'espressione `int-hood (if ...)`. Per ogni membro del vicinato, viene valutata la condizione `(or (= (nbr-range) 0) (not (< (abs (- v (nbr v))) (inf))))`. Se la distanza è nulla (indicando il dispositivo stesso) oppure se la differenza tra il valore locale `v` e quello del vicino è indefinita, il contributo viene azzerato per evitare singolarità.Altrimenti, si calcola mediante l'espressione `(* (/ (- v (nbr v)) (nbr-range)) (normalize (nbr-vec))))`, che corrisponde al pro-

dotto tra la differenza dei valori del campo, normalizzata rispetto alla distanza tra i due dispositivi, e il vettore normalizzato direzionato verso il vicino.

`share-distance-to` calcola e condivide una stima della distanza dalla sorgente all'interno della rete. Essa richiede due argomenti booleani: `is-calculating`, che indica se il dispositivo è abilitato al calcolo della distanza, e `source`, che identifica i dispositivi di cui si vuole conoscere la distanza. I nodi abilitati al calcolo sfruttano la funzione `distance-to` per stimare la distanza dalla sorgente, memorizzando il risultato nella variabile `base`. I dispositivi per cui `base` risulta finito vengono illuminati di verde. La funzione restituisce il risultato dell'espressione `(mux is-calculating base (min-hood (+ (nbr-range) (nbr base))))`.
Se `is-calculating` è vero, la funzione restituisce `base`; altrimenti, restituisce il minimo tra le somme delle distanze dai vicini e le rispettive stime di distanza dalla sorgente.

La funzione `nav-grad` combina le funzionalità di `share-distance-to` e `grad` per calcolare il vettore di navigazione verso la sorgente. Richiede in ingresso `is-mover`, il quale indica la mobilità del dispositivo, e `source`, che identifica i dispositivi sorgente del campo di distanza. Il gradiente del campo viene calcolato tramite l'espressione `(grad (share-distance-to (not is-mover) source))` e il risultato viene memorizzato nella variabile `g`. Si restituisce il risultato di `(mux (and is-mover (> (len g) 0)) (normalize g) (tup 0 0)`. Se il dispositivo è mobile e `g` ha lunghezza positiva, l'esito è il vettore normalizzato `g`; altrimenti, si ottiene il vettore nullo.

Trasposizione in Collektive La funzione `grad` (Listing 4.10) è stata tradotta come segue : si calcolano le differenze (`differences`) tra il valore locale e quelli dei vicini tramite l'espressione `mapNeighborhood { v } - neighboring(v)` (Listing 3.9), quindi si determinano distanze e direzioni rispetto ai vicini e si combinano questi campi di vicinato (`alignedMapValues`) per ottenere il gradiente.

La funzione `shareDistanceTo` (Listing 4.11) utilizza la funzione `distanceTo` (Listing 3.14) per calcolare le distanze dalla sorgente; i dispositivi incapaci di calcolarla determinano il minimo tra la somma delle distanze dai vicini e le rispettive stime (`neighborDistances() + neighboring(myDist)`), grazie all'espressione `potentialDist.all.valueOfMinBy(...)`.

In Listing 4.12 si mostra la trasposizione della funzione `nav-grad` che, come l'originale, combina le funzionalità delle due funzioni precedenti per calcolare il vettore di navigazione verso la sorgente.

Listing 4.10: Trasposizione della funzione `grad` in Collektive.

```
/**
 * Compute the gradient of a scalar field [v].
 */
fun Aggregate<Int>.grad(
    v: Double,
    neighborDistances: () -> Field<Int, Double>,
    neighborDirectionVectors: () -> Field<Int, Vector2D>,
): Vector2D {
    val differences = mapNeighborhood { v } - neighboring(v)
    val directions = neighborDirectionVectors()
    val distances = neighborDistances()
    return distances.alignedMapValues(
        differences,
        directions,
        { dist, diff, dir ->
            when {
                dist == 0.0 || !(abs(diff) < Double.POSITIVE_INFINITY) ->
                    vectorZero
                else -> dir.normalize() * (diff / dist)
            }
        }).all.run {
            fold(vectorZero) { acc, (_, value) -> acc + value } / size.toDouble()
        }
}
```

Listing 4.11: Trasposizione della funzione `share-distance-to` in Collektive.

```
/**
 * Share the distance from the [isCalculating] or [source]
 * to all non [isCalculating] nodes.
 */
fun Aggregate<Int>.shareDistanceTo(
    isCalculating: Boolean,
    source: Boolean,
    neighborDistances: () -> Field<Int, Double>,
): Double {
    val toSource = distanceTo(source, neighborDistances())
    val myDist = if (isCalculating) toSource else Double.POSITIVE_INFINITY
    val potentialDist = neighborDistances() + neighboring(myDist)
    val minDistance = potentialDist.all.valueOfMinBy { (_, value) -> value }
    return if (isCalculating) myDist else minDistance
}
```

Listing 4.12: Trasposizione della funzione `nav-grad` (4.9) in Collektive.

```
/**
 * Computes the navigation gradient vector for a node,
 * determining its direction of movement.
 */
fun Aggregate<Int>.navGrad(
    mover: Boolean,
    source: Boolean,
    neighborDistances: () -> Field<Int, Double>,
    neighborDirectionVectors: () -> Field<Int, Vector2D>,
): Vector2D = shareDistanceTo(!mover, source, neighborDistances).let { distance ->
    val g = grad(distance, neighborDistances, neighborDirectionVectors)
    when {
        mover && g.magnitude() > 0.0 -> g.normalize()
        else -> vectorZero
    }
}
```

In fig. 4.5 è riportata la simulazione dell'esperimento, in cui ogni nodo è colorato in base alla distanza dalla sorgente (evidenziata in rosso), mentre i dispositivi mobili sono evidenziati in blu. La figura mostra come i nodi mobili seguano la discesa del gradiente del campo delle distanze per raggiungere la sorgente.

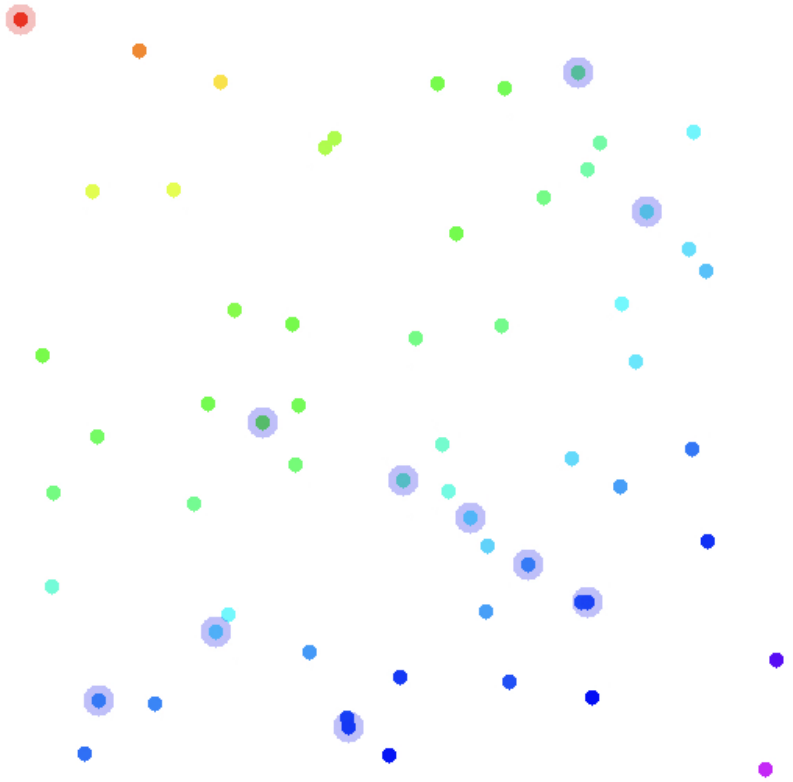

Figura 4.5: Simulazione dell'esperimento in listing 4.12. Ogni nodo è colorato rispetto alla distanza dalla sorgente (nodo evidenziato in rosso). I dispositivi mobili sono evidenziati da un cerchio blu.

4.6 Partizionamento di Voronoi

Obiettivo Suddividere lo spazio della rete tramite partizionamento di Voronoi. Un insieme di dispositivi è designato come sorgenti. Ogni altro nodo determina a quale sorgente è più vicino, suddividendo di fatto lo spazio della rete in celle. Il dispositivo è classificato in base alla sua posizione: in una cella, su un bordo (tra due celle) o come vertice (all'intersezione di almeno tre celle). La colorazione dei nodi riflette questa classificazione.

Codice originale Il programma (Listing 4.13) richiede in ingresso due valori booleani: `src`, che definisce l'insieme dei dispositivi generatori delle celle, e `showedge`, un flag che abilita la visualizzazione dei bordi.

Listing 4.13: Codice della funzione `voronoi` in Proto.

```
(def voronoi (src showedge)
  (let* ((closest-src (broadcast src (mid)))
     (edge (any-hood (not (= (nbr closest-src) closest-src))))
   (vertex
    (and edge
     (let ((max-nbr (max-hood (if (not (= closest-src (nbr closest-src)))
               (nbr closest-src)
               -1)))
     (min-nbr (min-hood (if (not (= closest-src (nbr closest-src)))
               (nbr closest-src)
               (inf)))))
       (not (= max-nbr min-nbr))))))
    (if (and edge showedge)
      (if vertex
  (rgb (tup 1 0 0))
  (rgb (tup 0 0 0.5)))
      (uid2rgb closest-src 10))
    (tup closest-src edge)))
```

Tramite il costrutto `let*`, si definiscono tre variabili in modo sequenziale:

- `closest-src`: identificatore del dispositivo sorgente più vicino, sfruttando la funzione `broardcast src (mid)`.
- `edge`: indica se il dispositivo corrente si trovi su un bordo tra due celle, ossia se esista almeno un vicino (`any-hood`) la cui sorgente più prossima (`nbr closest-src`) sia diversa dalla propria (`closest-src`).

- `vertex`: indica se il dispositivo corrente sia un vertice, ossia se si trovi su un bordo (`and edge`) e, tra i vicini appartenenti a celle diverse (`max-nbr`, `min-nbr`), vi siano almeno due identificativi di sorgente distinti (`(not (= max-nbr min-nbr))`).

Se il flag `showedge` è attivo, i dispositivi vengono colorati in base alla loro classificazione: i vertici assumono il colore `rgb (tup 1 0 0)`, i bordi `rgb (tup 0 0 0.5)`, mentre gli altri dispositivi adottano il colore associato alla propria cella di appartenenza (`uid2rgb closest-src 10`). La funzione restituisce la coppia `(tup closest-src edge)`.

Trasposizione in Collektive (Listing 4.14) L'identificatore della sorgente più vicina è determinato dalla funzione `gradientCast` (Listing 3.12), la cui natura assicura che ogni dispositivo ottenga il valore da quella più prossima.

I vicini condividono il valore calcolato tramite `neighboring` (Listing 3.8), permettendo a ogni nodo di contare il numero di sorgenti diverse nel vicinato, determinando così la propria classificazione. Se `distinctSources >= 3`, il dispositivo è un vertice; se `distinctSources == 2`, è su un bordo.

In questa traduzione, il programma non restituisce la coppia `(closest-src edge)`, bensì si limita a restituire il colore (Figura 4.6) associato al dispositivo.

Listing 4.14: Trasposizione della funzione `voronoi` in Collektive.

```
/**
 * Computes the Voronoi tessellation (https://en.wikipedia.org/wiki/Voronoi_diagram)
 * based on a set of [source]s, producing a field of integers which identify
 * the region each device belongs to.
 * A device can take one of the following roles:
 * - **Vertex**: it is at the junction of three or more Voronoi cells.
 *   Its color will be [VERTEX_COLOR].
 * - **Border**: it is at the junction of two Voronoi cells.
 *   Its color will be [BORDER_COLOR].
 * - **Cell Member**: it is neither a vertex nor a border. Its color
 *   is calculated based on the ID of the closest source.
 */
fun Aggregate<Int>.voronoi(source: Boolean, metric: () -> Field<Int, Double>): Int {
    val closestSource = closestSource(source, metric)
    val neighborClosestSources = neighboring(closestSource)
    val distinctSources = neighborClosestSources.all
        .sequence
        .map { it.value }
        .toSet()
        .count()
    val isVertex = distinctSources >= 3
    val isBorder = distinctSources == 2
    return when {
        isVertex -> VERTEX_COLOR
        isBorder -> BORDER_COLOR
        else -> closestSource.toColor() // toColor() maps an ID to a color value
    }
}

/**
 * Find the closest source by computing a multi-gradient from all sources.
 * If there are no sources, return 0.
 */
private fun Aggregate<Int>.closestSource(source: Boolean, metric: () -> Field<Int, Double>): Int = gradientCast(
    source = source,
    local = localId,
    metric = metric(),
)
```

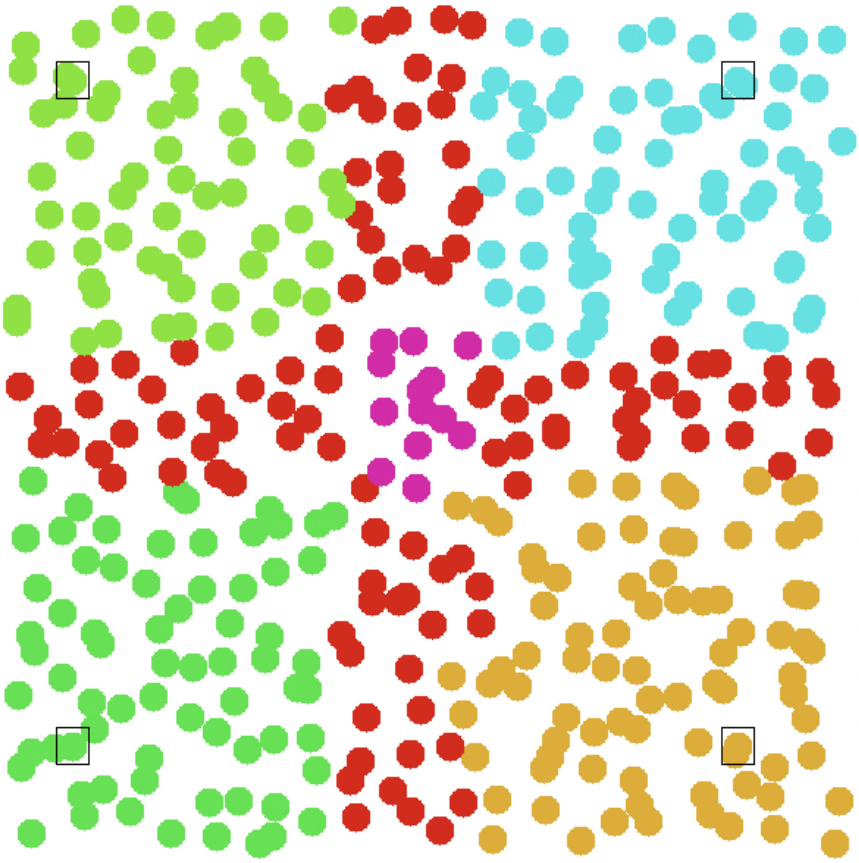

Figura 4.6: Simulazione dell'esperimento in listing 4.14. I dispositivi sono colorati in base alla cella di appartenenza: i dispositivi sui bordi sono colorati in rosso, mentre i vertici sono colorati in magenta. Le sorgenti sono evidenziate da un rettangolo nero.

4.7 Connessione lungo l'albero dei cammini minimi

Obiettivo Stabilire un collegamento tra due dispositivi, identificando e attraversando i nodi che compongono l'albero dei cammini minimi che li collega.

Codice originale Il programma è costituito da tre funzioni: `spath`, `connect` e `wire`.

La funzione `spath` (Listing 4.15) determina l'albero dei cammini minimi tra sorgente (`src`) e destinazione (`dest`), sfruttando la distanza `d` di ciascun dispositivo rispetto alla destinazione.

In fase iniziale, viene definita la variabile `min-id`, la quale memorizza l'identificatore del vicino che presenta la distanza minima `d`, come espresso da `(2nd (min-hood (tup (nbr d) (nbr (mid)))))`.

Successivamente, l'espressione `(rep ispath ...)` tiene traccia dei dispositivi che appartengono all'albero. La variabile `ispath` è inizializzata a `0` (`false`) e viene aggiornata a ogni round mediante l'espressione:

```
(mux src 1 (any-hood (muxand (= (mid) (nbr min-id) (nbr ispath)))))
```

Questa logica opera come segue:

- Se il dispositivo corrente è la sorgente (`src`), `ispath` è impostato a `1` (`true`).
- Altrimenti, il dispositivo verifica se almeno un vicino (`any-hood`) soddisfa congiuntamente due condizioni: che il proprio identificatore (`(mid)`) sia il `min-id` del vicino (`= (mid) (nbr min-id)`), e che tale vicino faccia già parte dell'albero (`(nbr ispath)`).

La funzione restituisce `ispath`.

Listing 4.15: Codice della funzione `spath` in Proto.

```
(def spath (src dest d)
  (let* ((min-id (2nd (min-hood (tup (nbr d) (nbr (mid)))))))
    (rep ispath
     0
     (mux src 1 (any-hood (muxand (= (mid) (nbr min-id)) (nbr ispath)))))))
```

`connect` (Listing 4.16) stabilisce il collegamento logico tra sorgente (`src`) e destinazione (`dest`), computando un vettore di connessione che indica il prossimo dispositivo lungo l'albero.

Ogni dispositivo calcola innanzitutto la propria distanza `d` dalla destinazione mediante la funzione `distance-to`. Successivamente, tutti i dispositivi verificano la propria appartenenza all'albero, invocando `spath`.

Attraverso l'espressione `if thepath ...`, i dispositivi che risultano parte dell'albero collaborano per calcolare il vettore di connessione. Questo calcolo è basato sulla ricerca della distanza minima rispetto alla destinazione tra i nodi vicini (`(min-hood (nbr d))`). Vengono poi sommati i contributi di quei vicini per cui la distanza `(nbr d)` coincida con tale distanza (`= (nbr d) min-d`).

Se tale condizione è verificata, l'espressione `mux ...` restituisce il vettore che punta verso il vicino in questione (`(nbr-vec)`); in caso contrario, restituisce il vettore nullo.

Listing 4.16: Codice della funzione `connect` in Proto.

```
(def connect (src dest)
  (let* ((d (distance-to dest))
   (thepath (spath src dest d)))
  (probe d 0)
    (if thepath
       (let ((min-d (min-hood (nbr d))))
   (sum-hood
    (mux (= (nbr d) min-d)
      (nbr-vec)
      (tup 0 0 0))))
      (tup 0 0 0)))
  )
```

La funzione `wire` (listing 4.17) invoca `connect` solo per i nodi che non hanno un ostacolo vicino (`near-obst`) oppure che siano la sorgente o la destinazione (`(and (not src) (not dest))`).

Listing 4.17: Codice della funzione `wire` in Proto.

```
(def wire (src dest near-obst)
    (if (and near-obst (and (not src)(not dest)))
      (tup 0 0 0)
      (connect src dest))
  )
```

Trasposizione in Collektive In Listing 4.18 è mostrata la trasposizione della funzione `spath`. Si utilizza il costrutto `share` (Listing 3.7) per mantenere lo stato di appartenenza all'albero, condividendolo allo stesso tempo con i nodi adiacenti e ottenendo così una vista sul loro stato (`nbrIsPath`). L'identificatore del vicino con distanza minima dalla destinazione si ottiene costruendo un `Field` che associa a ciascun nodo del vicinato la sua distanza (`neighboring(toDestination)` (Listing 3.8)), e selezionando l'identificatore corrispondente al valore minimo tramite `.minBy { (_, value) -> value }.id`. Successivamente, si costruisce un `Field` di `minId`, che viene trasformato in un campo di valori booleani: il valore è `true` per i nodi il cui `minId` coincide con il proprio identificatore. Questo campo viene quindi combinato con `nbrIsPath` tramite un'operazione logica `and`; se almeno un nodo adiacente restituisce `true`, il dispositivo viene considerato parte dell'albero.

La corrispondente implementazione di `connect` è riportata in Listing 4.19. Ogni nodo calcola la distanza dalla destinazione tramite `distanceTo` (Listing 3.14) e verifica la propria appartenenza all'albero di percorso minimo invocando la funzione `shortestPath`. Per calcolare il vettore di connessione, si combinano le distanze dei nodi adiacenti dalla destinazione (`neighborDistances`) con i rispettivi vettori direzionali (`neighborVectors`), considerando esclusivamente il contributo del nodo la cui distanza corrisponde al valore minimo rilevato nel vicinato.

La trasposizione della funzione `wire` è mostrata in Listing 4.20. La presenza di ostacoli nei dintorni viene rilevata costruendo un `Field` booleano tramite `neighboring(obstacle)` (Listing 3.8) e verificando, con la funzione `any(...)`, se almeno uno dei nodi adiacenti rappresenta un ostacolo.

In fig. 4.7 è mostrata la simulazione dell'esperimento.

Listing 4.18: Trasposizione della funzione `spath` (4.15) in Collektive.

```
/**
 * Check whenever the current node is on the path from [source] to destination.
 * [toDestination] is the distance to the destination.
 */
fun Aggregate<Int>.shortestPath(source: Boolean, toDestination: Double): Boolean =
     share(false) { nbrIsPath ->
    val minId = neighboring(toDestination).all.minBy { (_, value) -> value }.id
    val isOnShortestPath = neighboring(minId)
        .mapValues { it == localId }.and(nbrIsPath)
        .all
        .any { (_, value) -> value }
    when {
        source -> true
        else -> isOnShortestPath
    }
}
```

Listing 4.19: Trasposizione della funzione `connect` (4.16) in Collektive.

```
/**
 * Connect [source] to [destination] using the given [metric] to measure distances
 * and [neighborDirectionVectors] to get the direction to each neighbor.
 */
fun Aggregate<Int>.connect(
    source: Boolean,
    destination: Boolean,
    metric: () -> Field<Int, Double>,
    neighborDirectionVectors: () -> Field<Int, Vector2D>,
): Vector2D {
    val toDestination = distanceTo(destination, metric())
    val isOnShortestPath = shortestPath(source, toDestination)
    return when {
        isOnShortestPath -> {
            val neighborDistances = neighboring(toDestination)
            val minNeighborhoodDistance = neighborDistances.all.valueOfMinBy { (_,
                 dist) -> dist }
            neighborDirectionVectors()
                .alignedMapValues(neighborDistances) { dir, dist ->
                    if (dist == minNeighborhoodDistance) dir else vectorZero
                }
                .all
                .fold(vectorZero) { acc, (_, v) -> acc + v }
        }
        else -> vectorZero
    }
}
```

Listing 4.20: Trasposizione della funzione `wire` (4.17) in Collektive.

```
/**
 * Wire the source and the destination with connections to the next hop on the
     shortest path, avoiding obstacles.
 */
fun Aggregate<Int>.wire(collektiveDevice: CollektiveDevice<*>, env:
    EnvironmentVariables): Unit =
    with(collektiveDevice) {
        val source: Boolean = env["src"]
        val destination: Boolean = env["dest"]
        val obstacle: Boolean = env["obstacle"]
        val hasObstacleInNeighborhood = neighboring(obstacle).all.any { it.value }
        val position = coordinates()
        val connectionDir = when {
            hasObstacleInNeighborhood && (!source && !destination) -> vectorZero
            else -> connect(
                source = source,
                destination = destination,
                metric = { distances() },
                neighborDirectionVectors = {
                    neighboring(position).alignedMapValues(mapNeighborhood {
                        position }) { p, new0 -> p - new0 }
                },
            )
        }
        pointTo(connectionDir)
    }
```

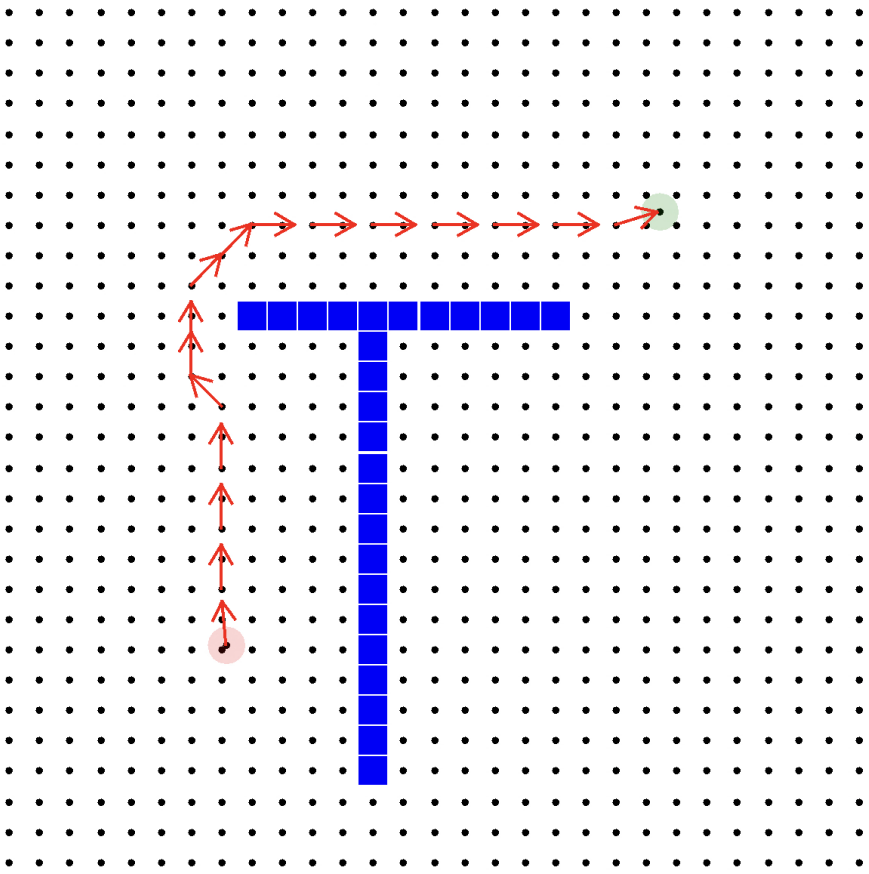

Figura 4.7: Simulazione dell'esperimento in listing 4.20. La sorgente è evidenziata in rosso mentre la destinazione in verde. Gli ostacoli sono rappresentati attraverso quadrati blu. I dispositivi che appartengono all'albero puntano verso il prossimo nodo del cammino, formando un collegamento che unisce sorgente e destinazione.

4.8 Tracciamento

Obiettivo Progettare un sistema di tracciamento in cui un dispositivo (destinazione) ne segua un altro (target), sfruttando un canale di comunicazione dedicato.

Codice originale Il programma (Listing 4.21) si compone di due funzioni: `channel` e `track`.

La funzione `channel` ha lo scopo di creare un "canale" di dispositivi che collega la sorgente (`src`) alla destinazione (`dest`), restituendo `true` per tutti i nodi che fanno parte di questo percorso e `false` per gli altri. In particolare, la funzione calcola la distanza tra sorgente e destinazione (`(distance src dst)`) e considera appartenenti al canale tutti i dispositivi per cui la somma delle distanze dalla sorgente e dalla destinazione risulta minore o uguale alla distanza tra le due, incrementata di un margine di tolleranza (`(<= (+ (gradient src) (gradient dst)) (+ d 1))`). Infine, la funzione restituisce `true` solo per i dispositivi effettivamente raggiungibili sia dalla sorgente sia dalla destinazione e che rientrino nella larghezza specificata dal canale, ampliata tramite la funzione `dilate`.

La funzione `track` consente alla destinazione di seguire il target sfruttando il canale creato da `channel`: la posizione del target viene propagata lungo il canale e, una volta ricevuta, la destinazione ne calcola la differenza rispetto alle proprie coordinate, determinando così la direzione verso cui puntare (`(- (broadcast target coord) coord)`).

Listing 4.21: Codice Proto per l'esperimento di navigazione del gradiente, comprendente le funzioni `channel` e `track`.

```
(def channel (src dst width)
  (let* ((d (distance src dst))
         (trail (<= (+ (gradient src) (gradient dst)) (+ d 1)))) ;; slop
    (and (< d (inf)) (dilate trail width))))

(def track (target dst coord)
  (if (channel target dst 10)
    (all (blue 1)
    (mux dst (- (broadcast target coord) coord) (tup 0 0 0)))
    (tup 0 0 0)))
```

Trasposizione in Collektive Per l'implementazione di `channel` è stata adottata la versione già utilizzata nella simulazione ChannelWithObstacle[2] (Listing 4.23).

La propagazione delle coordinate del nodo target nella funzione `track` (Listing 4.22) avviene tramite la funzione `gradientCast` (Listing 3.12), che diffonde le informazioni esclusivamente tra i nodi appartenenti al canale, grazie al partizionamento derivato dal costrutto `when`.

[2] `https://github.com/Collektive/collektive-examples/blob/fc511935fa0cce9116cd6bb8e00088443c2cd506/simulation/src/main/kotlin/it/unibo/collektive/examples/channel/ChannelWithObstacles.kt#L28C1-L44`

Listing 4.22: Trasposizione della funzione `track` (4.21) in Collektive.

```
/**
 * Computes the direction to the [target], relative to [destination].
 * If the device is not in the [channel], it returns a zero vector.
 * If the device is in the [channel] but not [destination], it returns a zero vector.
 * If the device is [destination], it returns the vector pointing to the [target].
 */
fun Aggregate<Int>.track(
    target: Boolean,
    destination: Boolean,
    channel: Boolean,
    coordinates: Point2D,
    metric: () -> Field<Int, Double>,
): Vector2D = when {
    channel -> {
        // Broadcast the target's coordinates through the channel
        val targetCoordinates = gradientCast(
            source = target,
            local = coordinates,
            metric = metric(),
        )
        if (destination) targetCoordinates - coordinates else vectorZero
    }
    else -> vectorZero
}
```

Listing 4.23: Funzione `channel` tratta dall'esperimento ChannelWithObstacles.

```
/**
 * Compute the channel between the [source] and the [destination]
 * with a specific [channelWidth].
 */
fun Aggregate<Int>.channel(
    collektiveDevice: CollektiveDevice<*>,
    source: Boolean,
    destination: Boolean,
    channelWidth: Double,
): Boolean = with(collektiveDevice) {
    require(channelWidth.isFinite() && channelWidth > 0)
    val distances = distances()
    val toSource = distanceTo(source, metric = distances)
    val toDestination = distanceTo(destination, metric = distances)
    val sourceToDestination = broadcast(distances = distances, from = source,
        payload = toDestination)
    val channel = toSource + toDestination - sourceToDestination
    return if (channel.isFinite()) channel <= channelWidth else false
}
```

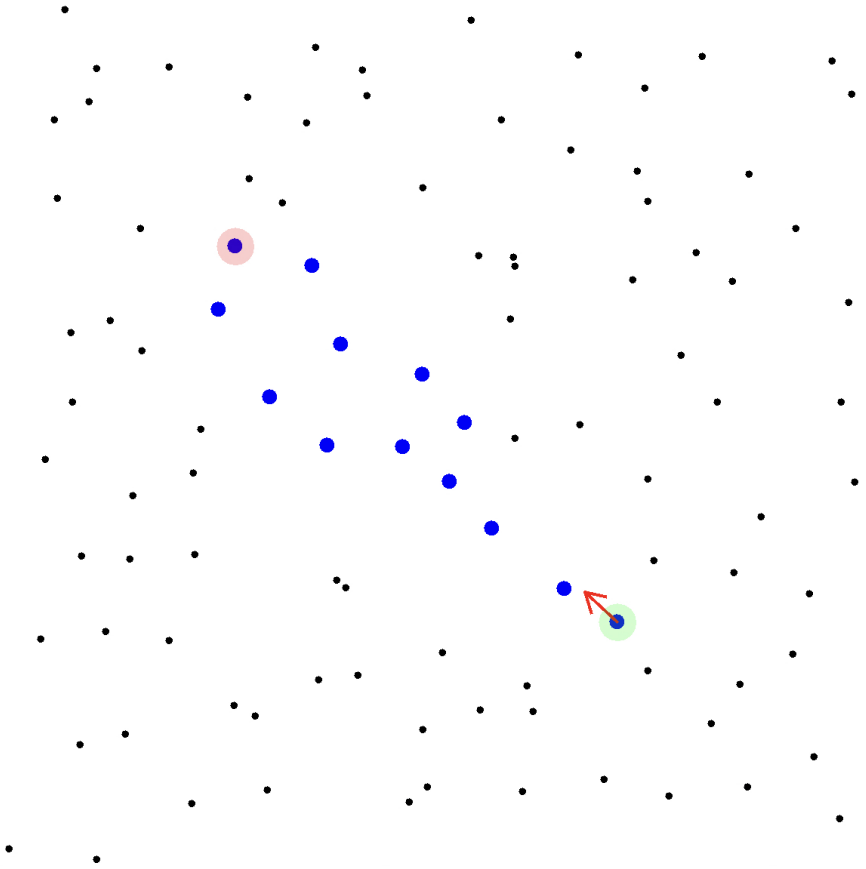

Figura 4.8: Simulazione dell'esperimento in listing 4.22. Il dispositivo target è evidenziato in rosso, mentre la destinazione in verde. I nodi che compongono il canale di comunicazione sono colorati in blu. La destinazione segue il target sfruttando le informazioni propagate lungo il canale.

4.9 Dinamica di uno stormo

Obiettivo Simulare la dinamica di uno stormo.

Ciascun dispositivo calcola la propria velocità in funzione della direzione iniziale, della distanza dai nodi vicini e delle loro rispettive direzioni di movimento. La direzione iniziale è determinata dal ruolo del nodo: se il dispositivo è un "leader", viene orientato verso l'origine del sistema di coordinate; in caso contrario, la direzione iniziale è nulla. La velocità, che evolve dinamicamente nel tempo, viene aggiornata secondo logiche di repulsione, coesione e allineamento, in base alla distanza dai vicini. La direzione risultante è data dalla somma ponderata dei contributi di ciascun vicino, insieme alla direzione iniziale.

Codice originale La funzione `flock` (Listing 4.24) richiede in ingresso la direzione di partenza (`dir`) del dispositivo.

Listing 4.24: Codice della funzione `flock` in Proto.

```
(def flock (dir)
  (rep v
   (tup 0 0 0)
   (let ((d
    (normalize
     (int-hood
      (if (< (nbr-range) 5)
        (* -1 (normalize (nbr-vec)))
        (if (> (nbr-range) 10)
    (* 0.2 (normalize (nbr-vec)))
    (normalize (nbr v))))))))
     (normalize
      (+ dir (mux (> (vdot d d) 0) d v))))))
```

L'evoluzione temporale della velocità dei nodi è gestita dall'istruzione `(rep v (tup 0 0 0) ...)`. Ad ogni round, viene determinato il vettore risultante `d` come somma dei contributi di repulsione, coesione e allineamento, provenienti dai dispositivi nel vicinato. I contributi dei vicini vengono calcolati e aggregati tramite la funzione `int-hood ...`, seguendo queste regole:

- Se la distanza da un vicino è inferiore a 5 unità (`(if (< (nbr-range) 5) ...)`), il contributo è un vettore che allontana dall'entità vicina. Si

calcola come il versore diretto verso il vicino, moltiplicato per -1 (`(* -1 (normalize (nbr-vec)))`).

- Se la distanza è superiore a 10 unità (`(if (> (nbr-range) 10) ...)`), si esercita un'attrazione. Il contributo è il versore orientato verso il vicino, scalato di 0.2 (`* 0.2 (normalize (nbr-vec))`).
- Se la distanza è compresa tra 5 e 10 unità, l'entità si allinea alla direzione attuale del vicino. Il contributo è dato dalla velocità attuale del vicino normalizzata (`normalize (nbr v)`).

Infine, la nuova velocità è determinata tramite l'espressione `(+ dir (mux (> (vdot d d) 0) d v))`: questa istruzione somma la direzione iniziale `dir` al vettore risultante `d` se la sua norma è positiva; in caso contrario, mantiene la velocità precedente `v`.

Trasposizione in Collektive La traduzione in Collektive è riportata in listing 4.25.

L'evoluzione della velocità dei nodi è gestita tramite l'operatore `share` (Listing 3.7), poiché la dinamica dipende sia dallo stato dei vicini sia dallo stato calcolato nel round precedente.

La necessità di una migliore approssimazione contigua del campo ha portato all'introduzione delle funzioni ausiliarie `sumWeightedNeighbors` e `spatialWeight` (Listing 4.26), che calcolano rispettivamente la somma pesata dei contributi dei vicini e il peso attribuito a ciascun nodo adiacente. Il peso assegnato si ottiene dividendo l'area del cerchio, il cui raggio è pari a quello di comunicazione, per il numero di dispositivi presenti nel vicinato.

La simulazione dell'esperimento (Figura 4.9) mostra i nodi coordinatori evidenziati in rosso. Si osserva come i dispositivi si organizzino spontaneamente in tre stormi, ciascuno dei quali si dirige verso l'origine grazie alla presenza di un numero sufficiente di leader che guidano il movimento collettivo.

Listing 4.25: Trasposizione della funzione `flock` (4.24) in Collektive.

```
/**
 * Implements flocking behavior.
 * The behavior is defined as follows:
 * - If a neighbor is closer than [CLOSE_NEIGHBOR_THRESHOLD] units, steer away from it.
 * - If a neighbor is farther than [FAR_NEIGHBOR_THRESHOLD] units, steer slightly towards it.
 * - Otherwise, align with the neighbor.
 * The resulting direction is normalized and combined with the current direction.
 */
fun Aggregate<Int>.flock(
    initialDirection: Vector2D,
    neighborDistances: () -> Field<Int, Double>,
    neighborDirectionVectors: () -> Field<Int, Vector2D>,
): Vector2D = share(vectorZero) { neighborVelocities ->
    val weights = neighboring(spatialWeight(CONNECTIVITY_RADIUS))
    val direction = neighborVelocities.alignedMapValues(
        neighborDistances(),
        neighborDirectionVectors(),
    ) { vel, dist, dir ->
        when {
            // steer away if too close
            dist > 0.0 && dist <= CLOSE_NEIGHBOR_THRESHOLD -> dir.normalize() * -1.0
            // steer slightly towards if too far
            dist > FAR_NEIGHBOR_THRESHOLD -> dir.normalize() * FAR_NEIGHBOR_ATTRACTION_WEIGHT
            // align if at a good distance
            else -> vel.normalize()
        }
    }.sumWeightedNeighbors(weights).normalize()
    (initialDirection + if (direction vdot direction > 0) direction else neighborVelocities.local.value).normalize()
}
```

Listing 4.26: Supporto per la simulazione di `int-hood` in Collektive.

```
/**
 * Computes integral (weighted sum) of neighbor vectors in the field.
 * This function combines the directions from all neighbors by applying their respective [weights] and summing the resulting vectors.
 */
fun Field<Int, Vector2D>.sumWeightedNeighbors(weights: Field<Int, Double>): Vector2D =
    alignedMapValues(weights) { point, weight -> point * weight }.all
        .fold(Vector2D(0.0 to 0.0)) { acc, entry -> acc + entry.value }


```

```
10 /**
11  * Computes the spatial weight of a device given a [radius].
12  */
13 fun Aggregate<Int>.spatialWeight(radius: Double): Double {
14     val neighborsSize = neighborhood().all.size
15     val totalArea = PI * radius * radius
16     return totalArea / neighborsSize
17 }
```

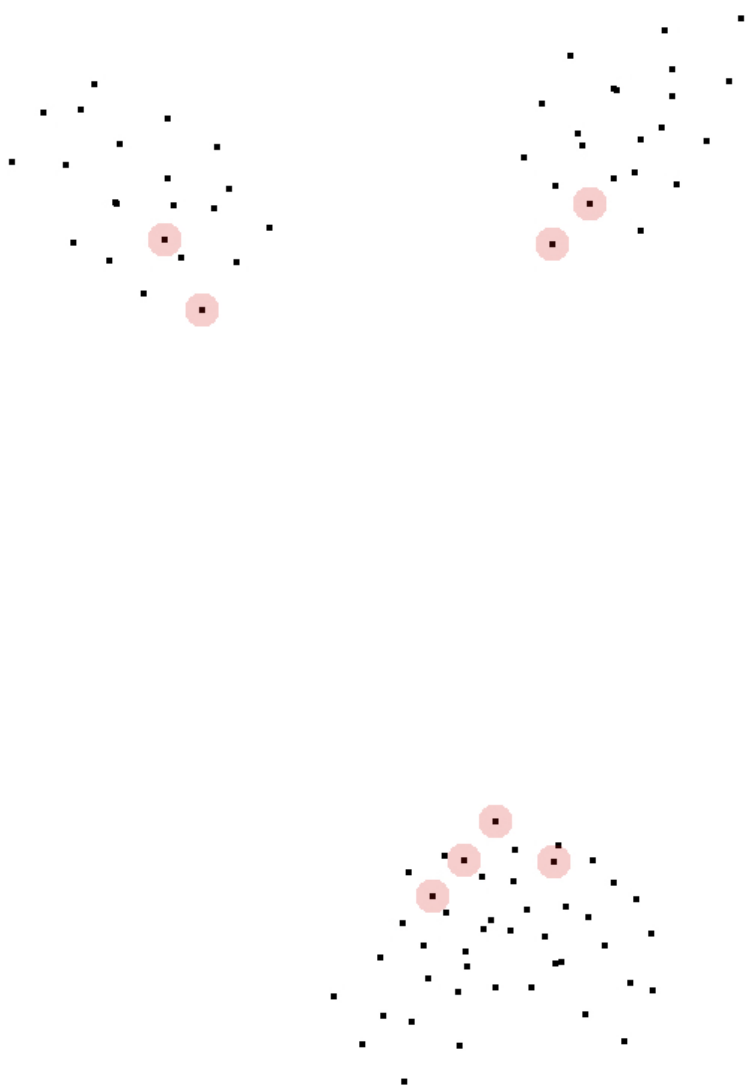

Figura 4.9: Simulazione dell'esperimento in listing 4.25. I dispositivi leader sono colorati in rosso, mentre gli altri sono colorati in blu. Si osserva come i nodi si organizzino spontaneamente in tre stormi, ciascuno dei quali si dirige verso l'origine grazie alla presenza di un numero sufficiente di leader che guidano il movimento collettivo.

Capitolo 5

Conclusioni

L'obiettivo di questa tesi è stato trasporre una selezione di programmi aggregati, originariamente sviluppati in Proto, nel linguaggio Collektive, con l'intento di offrire una guida pratica alla migrazione di sistemi legacy verso soluzioni più moderne e accessibili.

Il percorso intrapreso per raggiungere questo obiettivo ha richiesto un'analisi approfondita di AP e della sua evoluzione storica. Partendo dai concetti fondamentali (Capitolo 1), sono state esaminate le basi teoriche di FC e XC, per poi tracciare l'evoluzione dei linguaggi che hanno implementato questi principi. Particolare attenzione è stata dedicata a Proto (Capitolo 2), linguaggio pionieristico che ha posto le fondamenta di AP, analizzandone sintassi e costrutti. L'analisi di Collektive (Capitolo 3) ha poi rivelato come questo linguaggio non solo erediti i principi teorici di FC e XC, ma li integri in un contesto moderno e multipiattaforma.

Attraverso la trasposizione di una serie di programmi rappresentativi (Capitolo 4), si è dimostrato che Collektive è in grado di rappresentare efficacemente le logiche dei programmi originali, migliorandone la leggibilità e semplificandone lo sviluppo grazie a una libreria standard ricca e versatile. Tale libreria fornisce funzioni auto-stabilizzanti che consentono di sviluppare programmi con caratteristiche di resilienza e adattabilità. Un ulteriore vantaggio significativo è la presenza nativa dell'operatore `share`, che consente di modellare elegantemente l'evoluzione spazio-temporale dei campi computazionali in un unico costrutto, evitando la perdita di round di esecuzione presente nell'approccio di Proto basato sulla combinazione

esplicita di `rep` e `nbr`.

Questo lavoro apre diverse possibilità per rafforzare ulteriormente il ruolo di Collektive nell'ecosistema della programmazione aggregata. Innanzitutto, un confronto sistematico tra Collektive ed altri linguaggi come Protelis, ScaFi e FCPP fornirebbe agli sviluppatori una valutazione oggettiva riguardo a prestazioni, espressività ed ergonomia. Trasponendo gli stessi esempi in tutti questi linguaggi, sarebbe possibile offrire una guida preziosa per la scelta dello strumento più adatto. Inoltre, lo sviluppo di un playground web, sul modello di WebProto e ScaFi Web, permetterebbe agli sviluppatori di sperimentare il codice aggregato direttamente nel browser, accelerando l'adozione del framework e facilitandone l'apprendimento.

In conclusione, l'obiettivo è stato raggiunto, dimostrando che Collektive rappresenta un'evoluzione matura nel campo della programmazione aggregata. Fornendo un riferimento pratico per la traduzione da Proto, non solo si valida l'espressività del nuovo linguaggio, ma si offre anche una risorsa concreta alla comunità, abbassando le barriere d'ingresso e promuovendo la fiducia verso questa tecnologia. Il codice prodotto è stato integrato nel repository ufficiale degli esempi di Collektive, contribuendo direttamente all'ecosistema del progetto e rendendo disponibili i risultati a futuri sviluppatori.

Elenco delle figure

List of Listings

Bibliografia

[ABD+19] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. The share operator for field-based coordination. In Hanne Riis Nielson and Emilio Tuosto, editors, *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11533 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2019.

[ABDV18] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. Space-time universality of field calculus. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings*, volume 10852 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2018.

[ACD+24] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli. The exchange calculus (xc): A functional programming language design for distributed collective systems. *Journal of Systems and Software*, 210:111976, 2024.

[ACM+21] Gianluca Aguzzi, Roberto Casadei, Niccolò Maltoni, Danilo Pianini, and Mirko Viroli. Scafi-web: A web-based application for field-based coordination programming. In Ferruccio Damiani and Ornela Dardha,

editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 285–299. Springer, 2021.

[Aud20] Giorgio Audrito. Fcpp: an efficient and extensible field calculus framework. *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 153–159, 2020.

[AVD+19] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *ACM Trans. Comput. Log.*, 20(1):5:1–5:55, 2019.

[BB06] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[Bea04] Jacob Beal. Programming an amorphous computational medium. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms, International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers*, volume 3566 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2004.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[Cas22] Roberto Casadei. Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling, 2022.

[CVAD20] Roberto Casadei, Mirko Viroli, Giorgio Audrito, and Ferruccio Damiani. *FScaFi : A Core Calculus for Collective Adaptive Systems Programming*, pages 344–360. 10 2020.

[CVAP22] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala dsl and toolkit for aggregate programming. *SoftwareX*, 20:101248, 2022.

[PMV13] D Pianini, S Montagna, and M Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *Journal of Simulation*, 7(3):202–215, August 2013.

[PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1846–1853. ACM, 2015.

Acknowledgements

Desidero esprimere la mia sincera gratitudine al mio relatore, Prof. Danilo Pianini, e alla mia correlatrice, Dott.ssa Angela Cortecchia, per la guida e il supporto costante durante tutto il percorso di tesi, con preziosi feedback e consigli.

Un ringraziamento speciale va a Maria Chiara, alla sua famiglia e ai miei amici, per la pazienza, la comprensione e l'incoraggiamento continuo che mi hanno aiutato a superare i momenti più difficili. Finalmente, alla domanda "Cecco, ma quando ti laurei?" posso rispondere con un sorriso: "Tra poco!".

Infine, ringrazio la mia famiglia per l'amore e il sostegno incondizionato che mi hanno sempre dimostrato, rendendo possibile questo traguardo. Spero di rendervi orgogliosi.