ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

A Kotlin Multiplatform implementation of Aggregate Computing based on XC

Tesi di laurea in:
Laboratorio di Sistemi Software

Relatore
Prof. Pianini Danilo

Correlatore
Dott. Farabegoli Nicolas

Candidato
Cortecchia Angela

Quarta Sessione di Laurea
Anno Accademico 2022-2023

Abstract

The integration of technology in everyday activities is rising, with objects being increasingly equipped with computational capabilities and interconnected to form the Internet of Things, leading to the need for innovative cyber-physical services capable of creating a fast bridge between the real and virtual world.

The central idea of this thesis focuses on leveraging Kotlin Multiplatform to enhance aggregate computing based on XC principles, addressing challenges in developing versatile solutions across different environments, with the need of efficient and scalable applications operating from cloud to edge to mesh networks.

This thesis combines theoretical analysis, software development, and performance evaluation to assess the effectiveness of the objective, demonstrating versatility and efficiency. There is a notable improvement in performance, scalability, and adaptability across different network environments. With the proposed approach, the developed solution appears to be more efficient and effective in addressing complex challenges within systems rather than the current state of the art. The results demonstrate the transformative potential of this technology, suggesting that it can lead to more efficient and versatile service development.

In summary, this thesis shows the feasibility of using Kotlin Multiplatform to implement aggregate computing based on XC, demonstrating that the proposed approach is more efficient and scalable than the state of the art. Improved performance and scalability are emphasised through this approach, which opens doors for more efficient and adaptable solutions. This study sets the stage for future developments that could improve service efficiency and effectiveness.

To my family, a constant source of support,
who has always allowed me to choose
my own path with freedom and trust.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Context

Computing devices are becoming cheaper and *ubiquitous*, with objects being increasingly equipped with computational capabilities and interconnected to form the Internet of Thing (IoT), increasing the complexity of distributed systems. It is now common for individuals to own multiple computing devices of different types, resulting in technology becoming more integrated into daily activities. Inside those kinds of systems, different software components, and not necessarily one for each device, interact with each others building innovative services of cyber-physical nature, capable of creating a fast bridge between the real and the digital world.

Managing a single device in a distributed system can be challenging for several reasons, including: i) **scalability**, as the number of devices increases, managing each device individually becomes increasingly complex; ii) **resource constraints**, individual devices may have limited computational power, memory, and energy resources, making it challenging to perform complex tasks efficiently; iii) **heterogeneity**, devices in IoT systems often vary in terms of hardware capabilities, communication protocols, and software configurations, resulting in challenging management; iv) **context awareness**, single devices may have limited awareness of the broader context or environment in which they operate, leading to suboptimal decision-making and interactions.

Transitioning from a device-centric to an aggregate-centric approach can lead to several advantages: i) **distributed intelligence**, leveraging the collective capabilities of multiple devices enables distributed intelligence, where the system's intelligence emerges from interactions and collaboration among devices; ii) **resource pooling**, across multiple devices results in more efficient use of resources, such as computational power, memory, and energy; iii) **adaptability**, collective-centric approaches enable systems to adapt dynamically to changes in the environment or system requirements; devices can collaborate to self-organise, self-adapt, and self-optimise based on contextual information; iv) **robustness**, distributing computation and decision-making across multiple devices enhances system robustness and resilience to individual device failures or disruptions.

From an engineering perspective, the construction of IoT applications differs substantially from that of traditional software, especially when a particular service, to be realised, requires the coordination of devices of different nature. Hence, the need to engineer and coordinate the operations in such systems, one way is to program and operate in terms of *aggregates* of devices, or Collective Adaptive Systems (CAS), rather than manage each single device. In fact, the coordination of macroscopic behaviour in collective systems through a single program is a form of *macroprogramming*. However, this approach presents some primary challenges such as ensuring resilience, efficiency and privacy.

Collective Adaptive Systems CAS are systems composed of multiple autonomous entities, such as devices, sensors, and actuators, that interact to achieve a common goal [Fer15]. These systems are known for their ability to adjust to changes in their environment, system requirements, or operational conditions.

CAS are frequently used in IoT and Cyber-Physical Systems (CPS) applications, where devices collaborate to achieve a common goal, such as monitoring and controlling a physical environment or providing a service to users. The coordination of devices in CAS can be challenging due to device heterogeneity, resource constraints, and the dynamic environment.

Macroprogramming is used to promote collective behaviours, leveraging high-level abstractions and constructs to facilitate global coordination, decentralised control, and adaptability in complex systems.

Macroprogramming The term *macroprogramming* [Cas23] refers to the concept of expressing the macroscopic behaviour of a system through a single program, usually leveraging macro-level abstractions. This paradigm is driven by the need to capture *system-level behaviour* while abstracting the behaviour and interaction of the individual components involved. Macroprogramming approaches have been suggested to simplify the development of systems involving numerous interconnected sensors, actuators, and smart devices; they can be applied in contexts like IoT and CPS.

Macroprogramming abstractions can promote collective behaviour properties, such as self-organising or self-configuring in the context of CAS. By declaring tasks within a specific spatio-temporal region, systems can self-organise and effectively perform the task at hand, allowing for dynamic adaptation to the current deployment and spatial positions of the components involved.

Self-Organisation Coordination models are based on the notion that interactions among multiple independent and autonomous software systems can be designed as a space orthogonal to pure computation. This idea can be reified into a concept of shared data space, enabling so-called *generative communication*.

Over the course of time, different approaches have been created, such as *Linda* [Gel85] and *Mars* [CLZ00], suggesting innovative techniques for programming systems with devices of different nature focusing on the coordination of centralised local components, but not on the distribution of the systems. The main problems that can be encountered in distributed systems are dealing with *(i)* openness, as unexpected environment changes, *(ii)* large scale of agents and coordination abstractions to be managed, *(iii)* intrinsic adaptiveness, such as the ability to intercept relevant events and react to them, guaranteeing the resilience of the system.

The challenges necessitate a *self-organising coordination* approach, wherein coordination abstractions solely manage logical interactions. This ensures the emergence of global and robust patterns of correct coordination behaviour.

1.1.1 Computational Fields

Field-Based Coordination To facilitate self-organisation patterns of agents in complex environments, the concept of *coordination field* has been introduced. This abstraction serves as a navigational tool for agents over the actual environment.

In this context, the tuple-based middleware *TOTA* (Tuples On The Air) [MZ09] has been suggested to support field-based coordination for pervasive-computing applications. Initially, each field of the tuple in the system was assigned a name, along with a formula that supports the if-then-else construct and includes arithmetic and boolean operators to specify the field's behaviour over time. Secondly, it was introduced an operator in the tuple space responsible for applying formulas using contextual information.

Somewhat independently of the challenge of identifying appropriate coordination models for distributed and situated systems, several studies have tackled analogous issues in the broader endeavour of constructing distributed intelligent systems. This involves promoting higher abstractions of spatial collective adaptive systems.

Field Calculus Among the studies such as for managing space-time computing models for the manipulation of distributed data structures, the notion of *computational fields* were proposed [VBD$^+$19]. Consequently, the Field Calculus (FC) has been proposed as a foundational model for the coordination of computational devices spread in physical environments, also known as *aggregate computing.*

FC was introduced as minimal core calculus with the aim of capturing the fundaments that make use of computational fields: functions over and with fields, their evolution over time and the construction of field of values from neighbours.

The main concept of FC is to specify the aggregate system behaviour of a network of devices, where devices that can directly communicate with each other are indicated through a dynamic network relation. An example of its application is within a sensor network with a range of a broadcast communication.

The behaviour is applied through a functional composition of operators that manipulate the computational fields, called specification, that can be interpreted locally or globally. A local specification can describe a computation on an individual device executed in asynchronous "computation rounds", including sending or receiving messages from neighbours, getting information from sensors and computing the local value of the field. Each round is divided in three phases: i) *context building*, where the device collects information from the environment and from the messages received from neighbours, and aggregates them to build a local context; ii) *program execution*, each device executes a local program on the context; iii) *export sharing*, the device sends messages to its neighbours, containing the results of the computation.

In the global view, a field calculus expression specifies a mapping associating each computation round of each device to the value that it assumes at that space-time event.

This dual nature inherently facilitates the alignment of individual device behaviour with the overarching global behaviour of the entire network of devices.

Syntax of field calculus The *field calculus* is based on a minimal set of operators:

- *Stateful field evolution*: the expression $rep(e_1)\{(x) \rightarrow e_2\}$ describes a field evolving in time. e_1 represents the initial field value, the function $(x) \rightarrow e_2$ declares how the field changes at each execution;

- *Neighbour interaction*: the expression $nbr\{e\}$ builds a neighbouring field, a view of the field values in the surroundings of each device where neighbours are mapped to their evaluations of `e`;

- *Domain partitioning*: the expression $if(e_0)$ $\{e_1\}$ $\{e_2\}$ splits the computational field into two non-communicating zones hosting isolated computations: e_1 where the condition e_0 is `true` and e_2 where the condition e_0 is `false`.

Listing 1.1: The `rep` construct establishes an initial distance estimate **d** set to infinity, which then diminishes based on two conditions. If the source variable is **true**, indicating that the device is presently a **source**, its distance to itself is considered zero. Alternatively, if the device is not a **source**, the distance estimate is determined using the triangle inequality. This involves computing the minimum value obtained by adding the distance to each neighbour (using the `nbrRange` built-in function) to the neighbour field value (accessed via the `nbr{d}` notation). The `mux` function ensures that all arguments are evaluated prior to selection.

```
def mux(b, x, y) { if (b) {x} {y} }
def distanceTo(source) {
    rep(infinity){ (d) =>
        mux(source, 0, minHood(nbr{d} + nbrRange())
    }
}
```

In Listing 1.1 is shown a simple example of a field calculus program; it calculates the distance from a source node in a network. To implement more elaborated computations, it becomes necessary to combine the `rep` and `nbr` constructs. Their combination allows recreating a behaviour that evolves both in time and space: with `nbr` information is exchanged with neighbours, while `rep` takes care of the evolution of its own field. However, the joint use turned out that contained a hidden delay, identified and explained in [ABDV18].

To overcome the problem in the interaction between the two constructs, it has been proposed a new one: $share(e_1)\{(x) \rightarrow e_2\}$. This construct differs from `rep` and `nbr` combination in the way that the variable `x` is interpreted ad each round, that is identified as a *neighbouring field* rather than a value. In this case, e_2 is responsible for processing the neighbouring field into a local value shared with the neighbours at the end of the evaluation. Therefore, the `share` improves the communication speed.

The example proposed in Listing 1.1 can be rewritten using the `share` construct as in Listing 1.2.

Listing 1.2: The `share` construct is used to calculate the distance from a source node in a network.

```
def distanceTo(source) {
    share(infinity){ (d) =>
        mux(source, 0, minHood(d + nbrRange())
    }
}
```

Alignment The semantics of this language is defined as compositional and messages are exchanged thanks to `nbr`, with each message from a neighbour being automatically matched to a specific `nbr` construct, determined by a process called *alignment*. Each construct generates an "export", a data value intended to be sent to the neighbours, labelled with the coordinates of the node in the evaluation tree up to that construct. These exports are gathered into a message to broadcast to the neighbours that can be modelled as a value tree: an ordered tree of values obtained during the evaluation of each sub-expression of the program.

The *alignment* mechanism ensures that each construct of a device is matched with the corresponding construct of the neighbours, following an identical path in the evaluation tree.

1.1.2 Aggregate Computing

Aggregate programming elaborates a layered architecture that aims to simplify the design, creation and maintenance of distributed systems [CFP+19].

Through this methodology, the fundamental unit of computation shifts from an individual device to a collaborative ensemble of devices. It elaborates a layered architecture that aims to simplify the design, creation and maintenance of complex distributed systems.

Moreover, aggregate programming provides mechanisms for robust and adaptive coordination through simple programming APIs that implicitly guarantee safety and resilience. This framework is useful in large-scale scenarios, where there is insufficient fixed network infrastructure, as seen in situations like crowd management during large public events.

In various environments, interactions between wearable devices such as smartphones can support different kinds of services, including crowd detection, crowd-aware navigation or dispersal advice.

Aggregate Computing (AC) is a paradigm and engineering approach for the compositional development of self-adaptive IoT services from a global perspective [Cas23].

It has been developed with the core idea of functionality composing collective behaviours to achieve effective and resilient complex behaviours in dynamic networks. It views a given environment as a whole programmable entity whose parts collaboratively produce and consume services across space and time. AC is based on the principles of FC that is a functional programming model used to specify and compose collective behaviours with formally equivalent local and aggregate semantics.

The concept of *computational fields* can be viewed as a distributed data structure, with the aim of conceptually mapping each device to a value produced in a program, considering both space and time. Therefore, its structure supports the specification, analysis, simulation and runtime execution of *collective* or *aggregate* services, independently of the specific IoT architecture.

This paradigm has three key traits that characterise it: *(i) global stance with global-to-local mapping*, where the target of system design is the entire distributed IoT ecosystem, *(ii) behaviour compositionality*, whereby a rich collective service can be described in terms of the functional composition of simpler collective services, and *(iii) abstraction*, by which aggregate services enable adaptivity at different levels by abstracting from low-level issues and details. These attributes are crucial both in the design phase, where intricate solutions can frequently be articulated concisely and declaratively, and in the operational phase, where considerable flexibility is granted to the developers team and the platform regarding execution specifics and deployment strategies. The overall stack of the aggregate computing is shown in Figure 1.1.

Software platforms AC is designed to address many application scenarios, typically characterised by inherent distribution, heterogeneity, mobility and a lack of stable infrastructure. There are various strategies by which an AC system can run, contingent upon the selected and implemented communication methodologies.

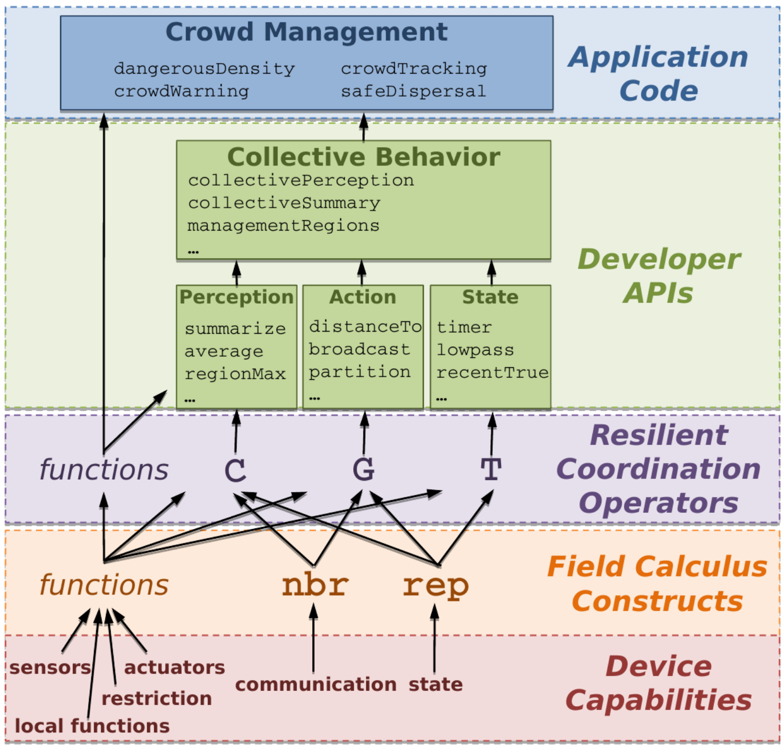

Figure 1.1: The aggregrete programming stack [BPV15].

Programs can be executed within a fully distributed peer-to-peer environment, where end-devices communicate directly with peer neighbours, and each independently runs its fragment of aggregate logic.

On the opposite side, there are entirely centralised solutions, where end-devices function solely as manager for sensors and actuators, forwarding perceptions upstream to one or more servers. These servers perform computations on behalf of the devices and subsequently transmit actuation data downstream. Thanks to the flexibility of its applications, AC has the potential to facilitate the creation of a more systematic spectrum for transitioning between cloud and distributed systems. This approach also embraces the emerging domains of Edge Computing [CV19] and Fog Computing [AWW18].

Aggregate computing in adaptive IoT Services Thanks to its features, *aggregate computing* is well suited for the development of *Opportunistic IoT Services*, supporting the properties:

- *Dynamicity*: the direct support for opportunistic service activation and evolution is achieved through code mobility and constructs that define dynamic domains of computations dependent on space and time;
- *Context-awareness*: aggregate programs utilise sensors, communication driven by neighborhood interactions, and iterative execution to consistently evolve the set of local contexts. This evolution forms the basis for the unfolding of computation and coordination logic;
- *Co-location*: as a natural method to define the concept of neighbourhood, relies on a physical space basis. AC inherently incorporates locality (both in space/time and purpose) to organise interactions and activities;
- *Transience*: AC provides constructs that directly supports the execution of distributed services based on time and context awareness.

Computational model There are some concepts and relationships included in the *aggregate computing* context:

- *Aggregate program*: an executable representation of specific aggregate logic that defines a collective behaviour;
- *Aggregate system*: a set of interconnected nodes or devices that support the collective execution of aggregate programs;
- *Aggregate application*: a specific aggregate logic operating on a designated aggregate system, aimed at solving particular problems in a specific context;
- *Node*: also referred to as *device*, it represents an individual AC-enabled entity, potentially equipped with sensors and actuators;
- *Neighbourhood*: the logical or physical set of nodes that can be directly contacted by a given node;
- *Global or local sensor*: a source for global or local information;
- *Global or local actuator*: a global or local actionable device for environment-directed actions;
- *Global or local computational environment*: anything that is detectable and subject to action through the use of global (or local) sensors and actuators This also includes the shared functionalities offered by the platform.

The networked devices inside an aggregate system compute and communicate at asynchronous rounds of execution. Each round, each device executes the global aggregate program according to the local semantics; then it updates its internal state and lastly generates the data for the external communication.

The round is performed by taking into account the computational context created by the previous state, sensor data and messages from neighbouring devices. After the execution of the round, result data are made available to neighbours and eventually instructed actuations are locally executed. The system can continuously react to changes by repeatedly executing rounds, allowing self-adaptation to contextual changes.

Models alignment Due to the high-level and platform-independent metamodels of IoT and AC systems, each with different goals or abstraction levels, it is necessary to align the two metamodels. This ensures that their unique focus is taken into account.

There are some main differences between the two metamodels:

- the concept of device is different, for AC is logical and is not the same as a device component of a Smart Object;
- although ensembles of Smart Objects are conceptualised as firs-class concept in AC, they are not explicitly represented in the IoT system metamodel;
- the concept of neighbourhood in AC regulates local, contextual communication among its devices. However, this concept cannot be explicitly mapped to IoT system concepts because device-to-device relationships are abstracted away from the metamodel.

1.1.3 XC

XC is an experimental programming language design to develop *homogeneous* distributed systems. Those kinds of systems consist of similar devices that communicate to neighbours and execute the same program. The aim of this experimental language is to push the abstraction boundaries further than actual existing approaches.

Many issues can arise in distributed systems, like concurrency, remote communication, asynchronous execution, message loss, and device failures. These kinds of problems must be taken into account when designing a programming language for such systems. Some of the possible applications of this approach may be crowd management by handled devices [BPV15], gossip-based data aggregation [JMB05], task allocation in robot swarms [BPP+14, VDPBMS05], and coordination of enterprise servers [CBP15].

The homogeneity in large-scale systems also arises when each device executes a program from a predefined set, reflecting a homogeneous configuration featuring a single program with an initial branch.

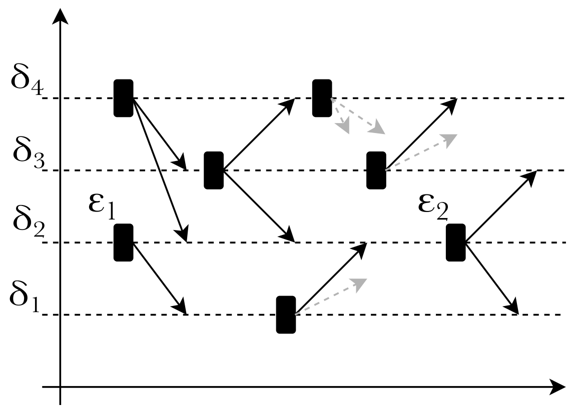

Figure 1.2: This image shows the devices' behaviour in **XC** where each δ_n is a device, ϵ_n is a computation round and the arrows represent the messages sent between devices. The device δ_2 executes two computation rounds ϵ_1 and ϵ_2. After the computation, δ_2 sends a message to δ_1 and δ_3. It can occur that a device δ_1 executes multiple rounds before δ_2 executes its own, in that case δ_2 will only see the last-received message from the device δ_1; newly received messages will overwrite the older ones. Grey arrows are messages lost never received.

System model Devices that can send or receive messages are called *neighbours*, and they can change dynamically to model network delays, failures or spatial movements.

Based on existing homogeneous systems, the device behaviour has been abstracted through a notion of *execution round*, in which a device independently executes an **XC** program and sends the resulting messages to its neighbours. Referring to *macroprogramming*, each device's behaviour in the network is developed as a single program, with no assumption of when an execution round will occur, meaning that they are entirely asynchronous.

Messages are handled queueing up in a buffer; when a device executes its **XC** program, it processes the received messages producing others to send to neighbours, that in turn will process their messages.

It can occur that a device executes multiple rounds before a neighbour executes its own, in that case the neighbour will only see the last received message from the first device; newly received messages will overwrite the older ones. Messages can persist across rounds; they are not removed from the buffer after they have been read unless they expire. The devices for which a message is available in a certain round are considered the *neighbours* for that round.

Data types **XC** features two kinds of values: *(i) local values (l)*, that include traditional types, and *(ii) neighbouring values (nvalues)*, a map from the device identifier to their local values, used to describe the set of values received from and sent to neighbours. In highly decoupled networks, it can occur that not all devices will produce a value, for this reason it will be used a default value.

Different types of operations can be applied to *nvalues*, where the function passed is repeatedly applied to neighbour's values in a field $\underline{w}$, excluding the self-value. A *local value* can be also converted to a *nvalue* using the default value for every device.

Communication in XC **XC** is based on a key communication primitive: $exchange(e_i,\ (\underline{n}) \rightarrow$ `return` e_r `send` $e_s)$ which is evaluated as follows: *(i)* the device computes the local value from the initial value e_i; *(ii)* the variable $\underline{n}$ gets substituted with the computed nvalues $\underline{w}$ of the received messages, eventually using the local value as default, the exchange returns the computed value v_r from e_r; *(iii)* e_s evaluates to a nvalue $\underline{w}_s$ consisting of local values to be sent to neighbours, that will take their corresponding value from $\underline{w}_s$ and use it for the execution of their next round.

A common pattern that can be used is to access neighbour's values through the use of the `exchange` function, as follows.
$nbr(e_i,\ e_s) = exchange(e_i, (\underline{n}) \rightarrow$ `return` $\underline{n}$ `send` $e_s)$, which means that the value of expression e_s is sent to neighbours and returns the received values gathered as $\underline{n}$ with its default, thus providing a view on neighbours' values e_s. Often the expressions e_r and e_s coincide, in this case, the `exchange` function is $exchange(e_i, (\underline{n}) \rightarrow$ `retsend` $e)$.

The crucial aspect of the **XC** expressivity is that `exchange` can send a different value to each neighbour, allowing custom interaction through them.

Conditionals **XC** supports conditional expressions, like $if\ (cond)\ \{e_1\}\ else\ \{e_2\}$. An exchange aligns only across the devices that take the same branch; thus, it evaluates only aligned sub-expressions. This means that a network can be split by a conditional expression into two non-communicating subnetworks, each evaluating a different branch without cross-communication.

Fault tolerance **XC** programs are resilient to failures: in case a node gets disconnected or messages get lost, the failed node won't show up among the neighbours of a given node in the next alignment. The logic of *exchange* is set to let neighbours' messages collectively operate, in order to make no assumptions on their number or identity. It is crucial to highlight that XC does not inherently offer assurances regarding fault tolerance. As a Turing-complete language, the capability to program non-resilient behaviour is inevitably present.

1.1.4 Collektive

Collektive is a minimal DSL for aggregate programming, designed to simplify the development of distributed systems by providing high-level abstractions for collective coordination and communication. It provides the means to specify the collective behaviour of a network of devices, where devices can directly communicate with each other and execute the same program. *Collektive* is based on the principles of AC, and it is designed to be used in the context of IoT applications. The language is designed to be easy to use, and to provide a high-level of expressiveness, while at the same time being efficient and scalable.

1.2 Motivations

In this section, the motivations for extending the existing DSL *Collektive* by applying the concepts of **XC** to aggregate programming constructs will be illustrated, the objectives that are intended to be achieved and the benefits that are expected to be obtained.

From an engineering perspective, the construction of IoT applications differs substantially from that of traditional software, especially when a particular service, to be realised, requires the coordination of devices of different nature.

Challenges and innovations Over the course of time, different approaches have been proposed to program these systems, such as TOTA, SAPERE [ZCF$^+$11], Aggregate Computing [BV16], Linda [Gel85], MARS [CLZ00] and others.

The most successful solution at the moment involves the use of Cloud Computing (Figure 1.3), which provides virtualized resources on a large scale, but presents limitations in terms of latency (due to the physical distance between machines) and scalability (due to centralisation) despite the significant improvement compared to traditional approaches. Cloud providers offer elastic scaling capabilities that allow organizations to dynamically adjust resources based on demand, minimizing the impact of scalability limitations and optimizing cost-efficiency.

Despite the scalability advantages of cloud computing, there are still scenarios where it may not be sufficient to meet the requirements of certain applications and workloads. For this reason, Edge Computing and Fog Computing have recently emerged, aiming to bring resources closer to the outer edge of the network, where interoperability problems between devices and complexity in managing distributed resources increase.

The design of applications capable of operating indistinctly on the cloud, on the edge, or even on a mesh network (in fact, on a computational continuum that goes from the edge to the cloud) can benefit from unconventional approaches, such as Aggregate Computing.

1.2.1 Heterogeneity limitations

One of the limitations of current collective programming approaches is the management of heterogeneity. In fact, the system assumes that there is a certain uniformity of capabilities along the continuum, or that it is possible to abstract it in some way (see, for example, the "pulverization" approach [CPP$^+$20]).

However, when the system includes devices whose nature is profoundly different (imagine, for example, a system where both well-equipped servers and tiny wearable or implanted devices with very moderate computing capabilities participate, as shown in Figure 1.4), finer mechanisms are needed. Such mechanisms can be the combination of adaptive strategies, resource management techniques, and the use of programming languages that can abstract the heterogeneity of devices.

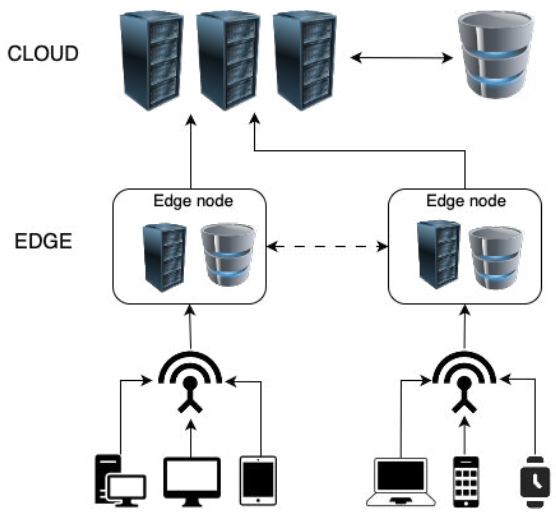

Figure 1.3: This figure shows the continuum of computing from the edge to the cloud, highlighting the edge-cloud verticality and the heterogeneity of devices.

A somewhat analogous problem appears in functional programming when modelling effects, and recent proposals aim to capture them in the form of capabilities.

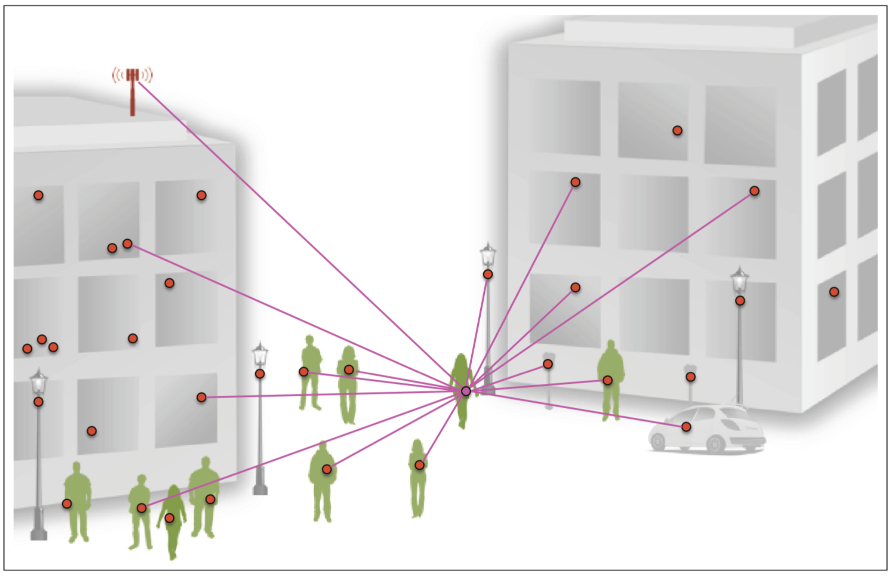

Figure 1.4: Our world is increasingly populated with a wide range of computing devices, embedded in our environment and with many opportunities for local and even location-independent interactions on fixed network infrastructures [BPV15].

1.2.2 Goal

The goal of this thesis is to extend the existing DSL *Collektive* by applying the concepts of **XC** to aggregate programming constructs, thus seeking to improve its performance in order to make this DSL competitive against existing approaches.

Furthermore, the aim is to create an integration of this DSL for the *Alchemist* simulator [Pia21], which serves as a meta-simulator primarily tailored for simulating intricate distributed systems across a diverse range of scenarios. These scenarios can include swarm robotics [ACV24], large-scale sensor networks [ACPV22], crowd simulation [BPV15], path planning, and even the morphogenesis of multicellular systems.

With these goals in mind, it is intended to create a programming language that can be executed on any device, regardless of its nature, and that can communicate with other devices in the network, thus simplifying the development of complex systems. To do this, it is necessary to rely on technologies and languages that can run on any device. Kotlin Multiplatform (KMP) will be used, which allows developers to write code that can be compiled for multiple targets, including JVM (ideal for server environments), Javascript (browser), Android, iOS (including watchOS and tvOS), and native versions (for Windows, MacOS, and Linux, both for x86 and ARM CPUs).

Finally, a performance comparison will be made between the extended language and existing solutions through benchmarking and simulations.

Impact The creation of a new programming language for aggregate computing is expected to have a substantial impact on the development of distributed systems, especially in the IoT and CPS domains. The new language will allow developers to write programs that can be executed on any device, regardless of its nature, and will be able to communicate with other devices in the network, thus simplifying the development of complex systems.

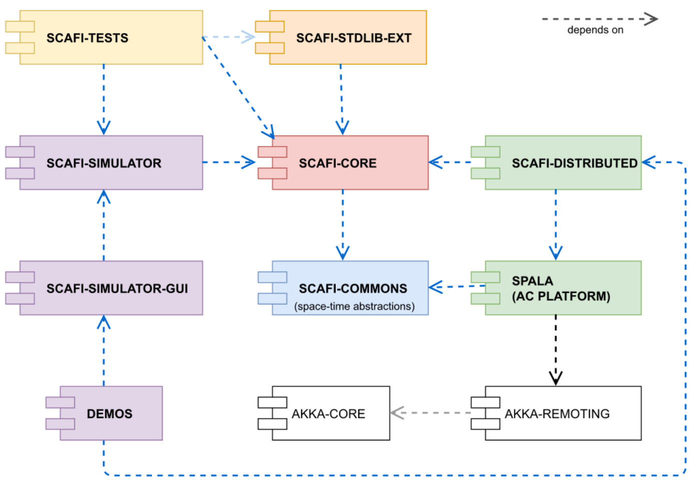

Figure 1.5: High-level architecture of the ScaFi toolkit.

1.3 State of Art

In this section, it will be presented the state of the art in the field of aggregate computing, focusing on the main existing frameworks and languages and their limitations.

Still, in terms of portability to heterogeneous systems, there are currently several software programs that implement the semantics of aggregate programming derived from field calculus, but they are not interoperable with each other.

1.3.1 ScaFi

ScaFi (Scala Field) is a Scala-based framework for aggregate programming [CVAP22]. It provides a DSL, libraries, a simulation environment with a GUI, integrated with the Alchemist simulator, and an actor-based runtime for the development of aggregate computing-based systems. *ScaFi*'s core is based on a variant of the *field calculus* called *FScaFi* [CVAD20], which peculiarity is to handle standard values to provide a simplified setting for DSL embedding.

This is achieved by introducing a notion of "computation against neighbours", meaning that is a computation whose output depends on the most recent values received from neighbours.

The architecture of *ScaFi* consists of various modules, as seen in Figure 1.5.

The DSL and standard library of reusable functions are implemented in the `scafi-core` module; meanwhile, the modules `scafi-simulator` and `scafi-simulator-gui` provide a simulation environment of aggregate systems with a graphical user interface.

The main applications of *ScaFi* are in the development of swarm systems [CAPV23], crowd management, wireless sensor network (WSN) [ACV22] and smart city applications.

ScaFi is capable of running aggregate programs on the Java Virtual Machine (JVM) and also on web browser thanks to the *ScaFi Web* tool [ACM+21], which allows the rapid prototyping of aggregate programs. On the other hand, it does not provide an aggregate standard library in the public version, neither supports to different advanced mechanisms. Furthermore, it is based on Scala 2, which means that it needs a review to be updated to Scala 3.

Since *ScaFi* only supports JVM and web-based applications, it is not well-suited for the development of complex distributed systems. devices and communication technologies, such as thin devices that do not support JVM, which limits the heterogeneity of devices that can be used in the system.

1.3.2 Protelis

Protelis [PVB15] is a functional programming language that implements a higher-order version of the *field calculus*, exposed through a C-like syntax, enabling the construction of widely reusable components of aggregate systems. It also provides various domain-specific APIs that are interoperable with Java, the *Protelis-Lang* [FPBV17]. *Protelis* has been developed since no foundational API for resilient, situated and distributed systems can be found in the Java ecosystem. It offers an implementation for the interoperability with the *Alchemist* simulator, where it can be seen its practical use in the development of aggregate systems.

As seen in the *Protelis* architecture (Figure 1.6), an interpreter executes a pre-parsed program at regular intervals, that communicates with other devices and draws contextual information from the environment. This is instantiated by specifying when the executions occur, how the devices communicate and how the environment is represented.

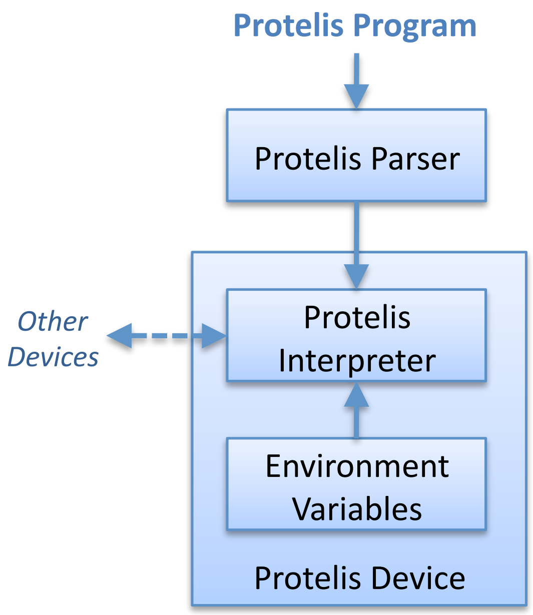

Figure 1.6: The abstract architecture of Protelis.

A key reason for this choice is that Java is highly portable across systems and devices. Another important reason is that Java's reflection mechanisms make it easy to import a large number of useful libraries and APIs for use in Protelis.

It is based on *Xtext*, a framework for the development of domain-specific languages, which provides a set of tools and libraries for the development. For this reason necessary requires a JVM to run, which limits the heterogeneity of devices that can be used in the system.

1.3.3 FCPP

FCPP [Aud20] is an implementation of the *Field Calculus* as a C++ library, with tools for simulations of distributed systems. This library is built as a component-based system Figure 1.7, with the aim of being easily extensible and reusable in different contexts.

It has a performance-oriented implementation based on compile-time optimisations, and it is designed to support simulated systems executed in parallel or self-organising cloud application.

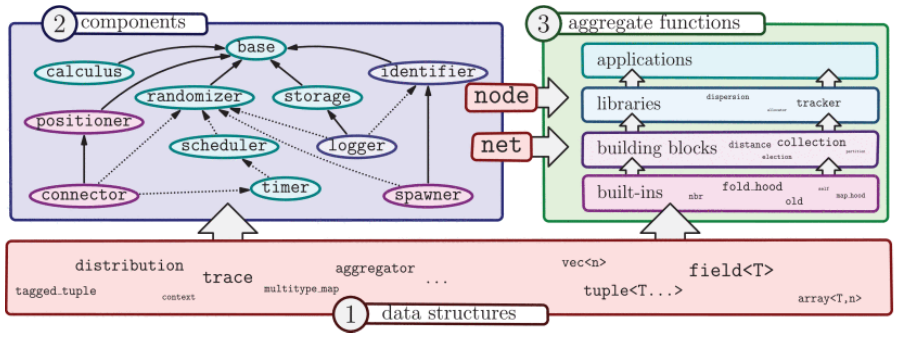

Figure 1.7: The Software architecture of FCPP represented as three main layers.

Being implemented in C++, allows *FCPP* to be used in a wide range of devices, including embedded systems and microcontrollers, for which a C++ compiler is available. Although *FCPP* is designed as a flexible and easily extensible platform, its currently supported range of features is more limited than *Protelis*' and *ScaFi*'s.

It allows running aggregate programs on low computational capacity devices. However, its syntax is less ergonomic compared to other languages, making it unlikely to reach a mainstream developer audience.

Final considerations Those three illustrated frameworks have more or less the same expressive power, with few notable differences. The main differences among them are between *ScaFi* and *FCPP*, that are *internal* DSL implemented in Scala and C++, and *Protelis* that is an *external dsl*, interpreted in the JVM.

Considering their syntax, *ScaFi* and *Protelis* partially imitate the abstraction level of the *field calculus*, being bound by the specific syntactic constraints of their host languages. Whereas, *Protelis*' syntax is more neat and specifically designed for field computation.

However, the more complex syntax of *ScaFi* and *FCPP* may prove easier to learn for programmers already familiar with Scala and *C++*.

Chapter 2

Contributions

2.1 Collektive

Collektive is a framework designed to simplify the definition of AC systems.

The main objective of this technology is to facilitate the development of aggregate programs that can be executed on a variety of computing systems, such as mobile and wearable devices, computers, and the cloud. This allows for interoperability and communication between these systems, despite their different nature.

To achieve this, *Collektive* uses the FC model to provide a straightforward and intuitive method for defining an aggregate program, without the need for low-level coding. In addition, *Collektive* has been developed to be multiplatform, so it can be executed on different systems thanks to the use of KMP.

As for the feature solution of alignment for the correct functioning of aggregate programming, it has been developed a compiler plugin with the purpose of annotating the functions that are aligned; those paths will be used for the actual alignment of the nodes.

Project Structure The project is subdivided into different submodules (as in Figure 2.1), each with a specific purpose:

1. **alchemist-incarnation-collektive**: contains the pieces for the Alchemist integration, in order to run the aggregate programs created with Collektive

on the simulator.

2. **dsl**: is the core of the project, contains the actual implementation of the logic and the AC operators and relative tests.

3. **compiler-plugin**: used to keep track of the stack at runtime, foreach aggregate program.

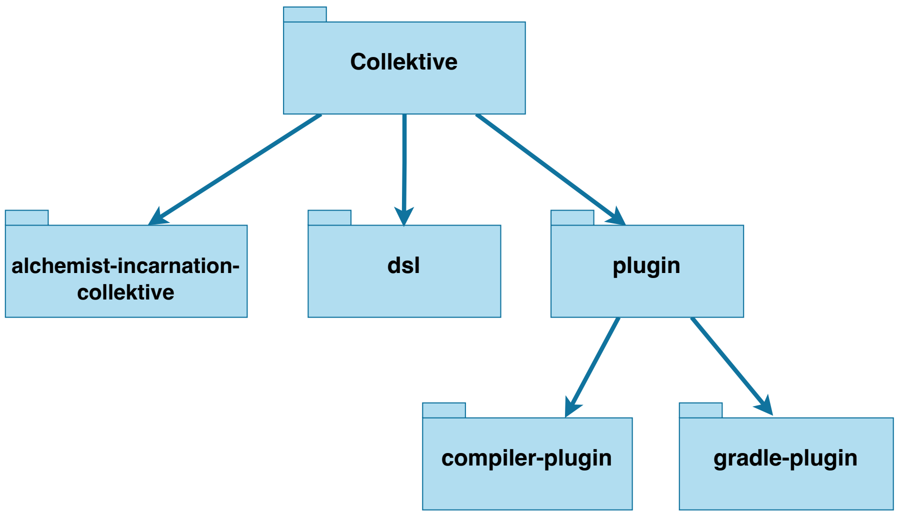

Figure 2.1: Packages diagram of the *Collektive* project.

Regarding the examples, there is a specific repository called **collektive-examples** that contains some samples of aggregate programs to show how to use the *Collektive* framework.

2.2 DSL

In this thesis, the original implementation of the DSL of *Collektive* will be modified to allow the use of **XC** and to improve its performance. The resultant DSL is available at a public GitHub repository[1].

A DSL is a specialised programming language or language framework tailored to address the requirements of a specific domain or problem area. In contrast

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

to general-purpose programming languages, which aim for versatility across various domains and problem categories, DSLs are crafted to cater precisely to the demands of a particular application or system. Consequently, general-purpose languages like Java typically exhibit greater complexity compared to DSLs.

Structure The *Collektive*'s DSL is composed of the following components:

- **Path**: represents a specific point in the Abstract Syntax Tree (AST) of an aggregate program;
- **State**: is an association between a path and a value, it is used by the compiler plugin to keep track of the computational state of the device in order to provide the correct alignment of the nodes;
- **Field**: represents the *computational field* used by aggregate constructs. It is a map of messages where the key is the identifier (ID) of the node, and the value is the associated message;
- **Message**: is an interface that represents the message exchanged between nodes, its concept will be explained in Section 2.2.2;
- **Network**: is the interface that represents the network used to manage the communication between devices;
- **Aggregate**: the actual core of the DSL. It is also used to handle all the data needed for the computation; for example, the *localId* and the *state* of the device. It contains the primary functions on which the language is extended, such as `exchange`, `exchanging`, `repeat` and `repeating`; To create an aggregate program, a function must extend this interface, in this way it will be possible to use the aggregate functions. It is also implemented the mechanism to handle the alignment of the functions used within the compiler plugin;
- **AggregateOperators**: contains the implementation of the functions created using the `exchange-exchanging` functions, such as `share`, `sharing` and `neighboringViaExchange`. Moreover, it contains the mechanism to manage the alignment of the fields used within the compiler plugin;

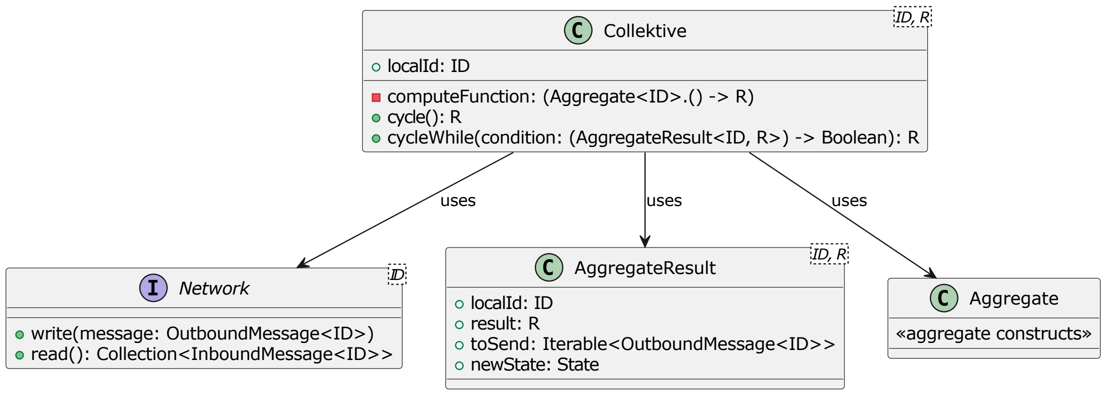
Collektive
ID, R
localId: ID
computeFunction: (Aggregate<ID>.() -> R)
cycle(): R
cycleWhile(condition: (AggregateResult<ID, R>) -> Boolean): R
uses
uses
uses
Network
ID
write(message: OutboundMessage<ID>)
read(): Collection<InboundMessage<ID>>
AggregateResult
ID, R
localId: ID
result: R
toSend: Iterable<OutboundMessage<ID>>
newState: State
Aggregate
«aggregate constructs»

Figure 2.2: Partial class diagram of the DSL, highlighting the entrypoint.

- **YieldSupport**: contains the *Yielding Context* and the *Yielding Result* for the "yielding" operations, which means that the function operates on an initial value but possibly returns a different value;

- **Collektive**: is the entrypoint for creating a "Collektive" device, it must have a specific *ID* and a *network* to manage the communication between devices. The effective aggregate program is identified by the *compute function*, which will be executed

- **AggregateResult**: is the result of one evaluation of the aggregate program, it contains the *localId* of the node, the effective *result* of the computation, the *messages to send* to other devices and the *new state* of the device;

In the class diagrams shown in Figure 2.3 and Figure 2.2, it is possible to see the main classes of the DSL introduced above and their relationships. The Interface *Aggregate* is implemented by the *AggregateContext* class, which is used to handle the logic and the computation of the main constructs of the DSL. This interface is then extended within *AggregateOperators* to handle the logic of the functions created using the `exchange-exchanging` functions.

2.2.1 XC in Collektive

Thanks to the design of **XC**, it is possible to implement the methods proposed by *field calculus* (Section 1.1.1) in terms of *exchange* (Section 1.1.3). The syntax of

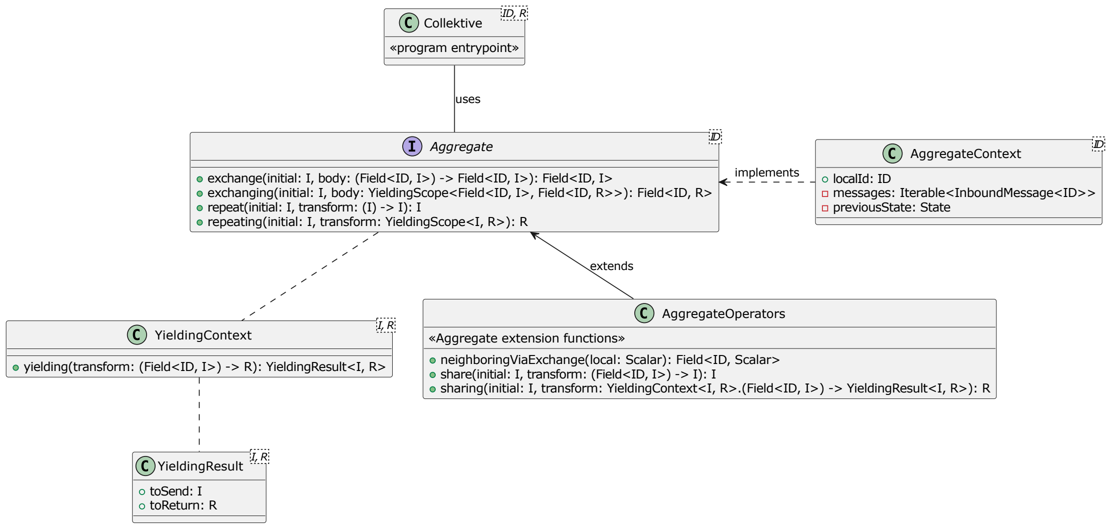
Collektive
ID, R
«program entrypoint»
uses
Aggregate
ID
exchange(initial: I, body: (Field<ID, I>) -> Field<ID, I>): Field<ID, I>
exchanging(initial: I, body: YieldingScope<Field<ID, I>, Field<ID, R>>): Field<ID, R>
repeat(initial: I, transform: (I) -> I): I
repeating(initial: I, transform: YieldingScope<I, R>): R
implements
AggregateContext
ID
localId: ID
messages: Iterable<InboundMessage<ID>>
previousState: State
extends
YieldingContext
I, R
yielding(transform: (Field<ID, I>) -> R): YieldingResult<I, R>
AggregateOperators
«Aggregate extension functions»
neighboringViaExchange(local: Scalar): Field<ID, Scalar>
share(initial: I, transform: (Field<ID, I>) -> I): I
sharing(initial: I, transform: YieldingContext<I, R>.(Field<ID, I>) -> YieldingResult<I, R>): R
YieldingResult
I, R
toSend: I
toReturn: R

Figure 2.3: Partial class diagram of the DSL structure, highlighting the *Aggregate* section.

XC allows for sending messages to specific nodes, enabling the implementation of *field calculus* operations through message exchange.

The *exchange* communication is based on *anisotropic* communications, meaning that it does not send the same properties or characteristics in all directions (Figure 2.4); therefore, messages have custom values sent to different neighbours.

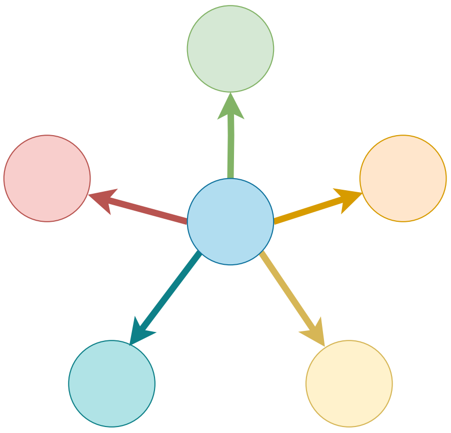

Figure 2.4: This image shows how anisotropic communication works, exchanging different information based on the neighbour.

This concept can be extended to the `share` function of field calculus, with the difference that the operation of `share` is based on *isotropic* communication, meaning that the information sent is uniform in all directions (Figure 2.5); therefore, messages have the same value sent to all neighbours.

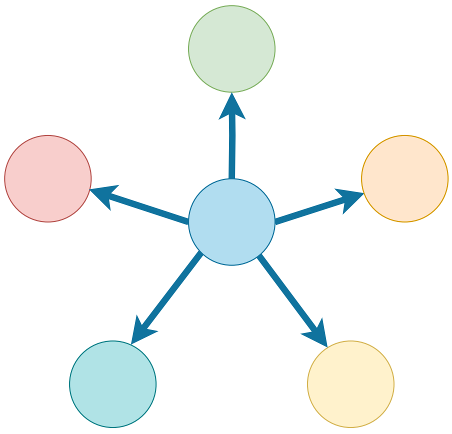

Figure 2.5: This image shows how isotropic communication works, exchanging the same information with all the neighbours.

All the DSL has been modified to use `exchange` for the implementation of the other constructs such as `share` and `nbr`, which is called `neighboring`. Only the `rep` construct has not been implemented in terms of `exchange`, as it is a function that allows iterating over oneself, it's better for neighbours not to receive messages of any kind, also for security and privacy reasons. As the original implementation, it supports the evaluation of *fields*.

Exchange The construct `exchange` provides i) access to neighbours' values, ii) persistence of information for subsequent executions, iii) communication with neighbours, and iv) compositional behaviour.

As seen in Section 1.1.3, the `exchange` function can send and return the same result, or it can send a message and return a different result; both cases have been implemented.

The `exchange` takes an *initial* value to use as a default, and a *body* that defines an aggregate function to be computed, as seen in the code snippet Listing 2.1, returning a *field*. When executing the aggregate program, it checks which messages have been received from the neighbours and applies the function, generating a custom value for each neighbour.

In the early stages of the design of this function, potential problems that could arise during execution were evaluated, such as the management of initialisation, i.e., the first round of message exchange. Consider a network consisting of n devices, all of which are neighbours to each other. The first device (d_1) to start within the network will certainly have a moment when it performs its first iteration ever.

The problem arises when the device d_1 has not yet received any messages, so it will not have neighbour values to perform the calculation, thus creating a deadlock situation where it does not know the neighbourhood and does not know who to actually send a message to. For this reason, it was necessary to implement a mechanism such that the device performs a first iteration on its initial value, and send the message to the network without a specific recipient, so that the future "neighbourhood" can receive it and start a communication with d_1. In this way, the next devices to "wake up" in the network will know that d_1 is present and will be able to communicate with it.

Listing 2.1: The signature of the `exchange` function.

```
fun <Initial> exchange(
    initial: Initial,
    body: (Field<ID, Initial>) -> Field<ID, Initial>,
): Field<ID, Initial>
```

Share The `share` construct captures the space-time nature of field computation through observation of neighbours' values, starting from an *initial* value, it reduces to a single local value given a *transform* function and updating and sharing to neighbours of a local variable, then returns the punctual value evaluated from the transformation (Listing 2.2).

Listing 2.2: The signature of the `share` function.

```
fun <ID : Any, Initial> Aggregate<ID>.share(
    initial: Initial,
    transform: (Field<ID, Initial>) -> Initial,
): Initial
```

As previously introduced, this construct can be expressed in terms of `exchange`. The `share` differs from the `exchange` because `share` does not differentiate the result of the computation based on the neighbour that sent the message; it sends it indiscriminately to all neighbours.

Neighboring The *field calculus* construct `nbr` is implemented as `neighboring`, more precisely as `neighboringViaExchange` due to its effective implementation based on the `exchanging` function. The `neighboringViaExchange` construct is used to access the values of the neighbours. It takes as parameter *local*, which can be an expression or a value, and returns a *field* of the same input type (Listing 2.3).

Listing 2.3: The signature of the `neighboringViaExchange` function.

```
fun <ID : Any, Scalar> Aggregate<ID>.neighboringViaExchange(
    local: Scalar,
): Field<ID, Scalar>
```

Repeat The `repeat` function is the equivalent of the `rep` construct in *field calculus*. It models the state evolution over time, the value of *initial* evolves at each execution depending on the *transform* function, and it returns the final value of the computation (Listing 2.4).

Listing 2.4: The signature of the `repeat` function.

```
fun <Initial> repeat(
    initial: Initial,
    transform: (Initial) -> Initial,
): Initial
```

As previously mentioned, this function is not implemented in terms of `exchange`, because iterating over oneself it is better not to send messages to neighbours, also for security and privacy reasons, its implementation will be explained in Section 2.5.4.

2.2.2 Messages

The accurate modelling of message functionality is crucial for device communication. Due to the different nature of messages supported, that is *anisotropic* and *isotropic* communication, it has been necessary to create specific classes for the handling of relative messages.

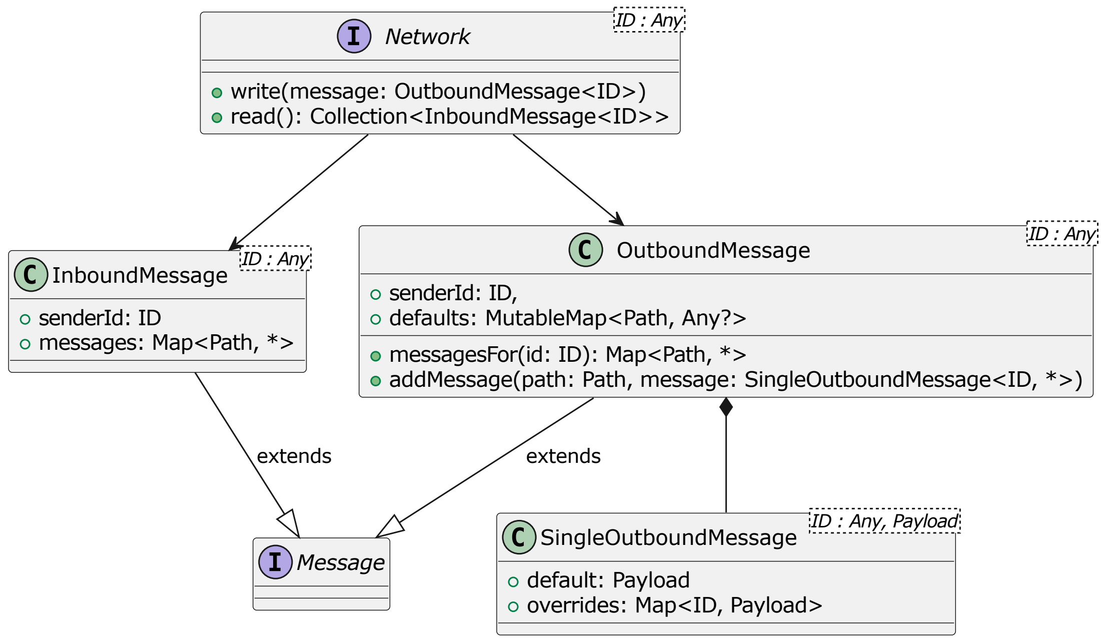
Network
ID : Any
write(message: OutboundMessage<ID>)
read(): Collection<InboundMessage<ID>>
InboundMessage
ID : Any
senderId: ID
messages: Map<Path, *>
OutboundMessage
ID : Any
senderId: ID,
defaults: MutableMap<Path, Any?>
messagesFor(id: ID): Map<Path, *>
addMessage(path: Path, message: SingleOutboundMessage<ID, *>)
extends
extends
Message
SingleOutboundMessage
ID : Any, Payload
default: Payload
overrides: Map<ID, Payload>

Figure 2.6: Class diagram of the network and messages stucture.

Previously, messages were sent to the network without considering the recipient, and the network was responsible for delivering the message to all the neighbours that were interconnected. Since the main feature of `exchange` is to send messages to specific neighbours with custom values, it is important to ensure that unintended recipients do not receive the message. Therefore, the old model of messages has been modified.

The Figure 2.6 shows the class diagram of the messages used in the DSL, highlighting their relationships and the inheritance of the *Message* interface. The concept and implementation of the messages will be explained in the following paragraphs.

OutboundMessage and SingleOutboundMessage It has been introduced the concept of *OutboundMessage* (Listing 2.5). Its goal is to associate to the sender of the message, the messages to send to the neighbours. With this implementation, it is possible to manage the messages for *isotropic* and *anisotropic* communication. In this way, a device sends to the network just one *OutboundMessage* that contains all the messages to send to the neighbours, and also unburdens the payload of the network.

Listing 2.5: Outbound message data class.

```
data class OutboundMessage<ID : Any>(
    expectedSize: Int,
    val senderId: ID,
) : Message
```

For what it concerns the *anisotropic* communication, it is necessary to keep track of the identifier of the recipient and the value associated with it, in order for the network to deliver the message to the correct neighbour. This has been implemented with a map of *overrides*, which associates to the recipient's identifier a specific value.

For the *isotropic* communication, it is just necessary to keep track of the default value to send to all the neighbours at a certain path of the computation. This is used when there are no overrides for the recipient that is reading the messages received. The network will be in charge to deliver the message to all the neighbours, so it is not necessary to keep track of the recipient's identifier.

Listing 2.6: Single outbound message data class.

```
data class SingleOutboundMessage<ID : Any, Payload>(
    val default: Payload,
    val overrides: Map<ID, Payload>,
)
```

In summary, when a program is executed and a device has computed its function, it has to generate the messages to send to the neighbours (so it's done inside the `exchange` function), it populates a *SingleOutboundMessage* (Listing 2.6), that will be used to populate the `overrides` and the `default` maps of the *OutboundMessage*. At the end it is added to a list of *OutboundMessage* and sent to the network.

InboundMessage The *InboundMessage* is a data class (Listing 2.7) modelled for the messages read from the network. When reading the *OutboundMessage*, there are some fields that are not needed, such as the receiver of the message, which is the device itself, and is necessary to keep the default value or the override value. To handle this, the *InboundMessage* has been modelled to keep only the identifier of the sender and a map of messages of *Path* and the associated value.

Listing 2.7: Inbound message data class.

```kotlin
data class InboundMessage<ID : Any>(
    val senderId: ID,
    val messages: Map<Path, *>,
) : Message
```

2.2.3 Network

The *Network* has the task of managing only the reading and writing of messages in the network for a single device. The actual implementation of the network is not managed by the *Collektive* DSL, but it is implemented in the *alchemist-incarnation-collektive* (Section 2.6) module, which is used to run the aggregate programs created with *Collektive* on the *Alchemist* simulator.

Inside the DSL, it is used at the beginning of each iteration of the aggregate program to read the messages received from other devices present in the network, and at the end of the iteration to write the messages to send to the other devices. The network actually supports only the specific types of messages introduced in Section 2.2.2, due to the concepts introduced by the study of **XC** and field calculus.

2.3 Plugin Extensions

In Kotlin, a plugin compiler extension refers to a mechanism that allows developers to extend or customise the behaviour of the Kotlin compiler. These extensions provide a way to hook into the compilation process and modify or enhance the way Kotlin code is compiled.

Developers can create their own compiler extensions by implementing the necessary interfaces or annotations defined by the Kotlin compiler API. These extensions can then be applied to Kotlin projects either globally or selectively, depending on the specific requirements of the project.

In KMP project development, the compiler undertakes a covert translation of Kotlin code into platform-specific code. Prior to this translation, the Kotlin code undergoes conversion into an Intermediate Representation (IR), creating an AST that is then transformed into platform-specific code. Notably, this approach eliminates the necessity to develop separate plugins for each platform, thus optimising the development process and fostering code reuse across varied environments.

Since the source code is fully accessible at compile time, it becomes feasible to analyse it to determine when the *alignment* of the functions is necessary; consequently, new code can be generated to ensure proper alignment. Furthermore, because the generation process is based on the intermediate representation, it enables interoperability across different project targets and devices running on the same platform. Modifications to the plugin are applied during compile time, minimising any notable impact on execution time. Lastly, users are shielded from the intricacies of alignment concerns, as the code generation process remains transparent to them.

Alignment The alignment in *Collektive* was initially implemented inspecting and annotating the body of the (ex)`aggregate` functions; none of the other functions were aligned.

In *Collektive*, the alignment of the functions is made by the compiler plugin, which keeps track of the sequence of functions called during the computation, using a custom stack. Since initially the entrypoint of the DSL was the `aggregate` function, the alignment was made by assuring that a reference to the *AggregateContext* was present, meaning that everything not related to the DSL did not require alignment.

By changing the main architecture of the DSL to use **XC**, it has been necessary to review the way the alignment was managed.

The alignment has been implemented in such a way that it also takes into account the functions from which an aggregate program is called. For example (Listing 2.8), we could have two different functions called `foo()` and `bar()` that both call the same aggregate program, but in fact they are two different functions and therefore must be aligned as two different programs.

Listing 2.8: Example of two unaligned functions.

```kotlin
val x = 0
fun foo(aggregate: Aggregate<Int>) = aggregate.neighboringViaExchange(x)
fun bar(aggregate: Aggregate<Int>) = aggregate.neighboringViaExchange(x)
acProgram { foo(this) } shouldNot alignWith { bar(this) }
```

The alignment process follows a specific strategy. Firstly, all function definitions are visited, and those involving aggregate computation are subject to alignment processing. Next, for each candidate function, the plugin visits all call sites in the function's body and checks if the call has an aggregate reference or is involved in an aggregate computation; meaning that it must have the *Aggregate* type as `extensionReceiver`, or `dispatchReceiver`, or one or more of the function's parameters. If so, the plugin aligns the expression call.

All the functions visited are so pushed into a stack. For each call to *Aggregate* also push the name of the function in the stack, appending a value that represents the occurrence of the function in the sequence of the computation. When the control flow exits that function, the function name is popped from the stack. The *Path* generated is a string that represents the sequence of the functions called during the computation, with the prepended calls of the non-Aggregate functions. The alignment is then made by comparing the *Path* of the two functions, and if they are equal, the two functions are aligned.

In an aggregate program, there may be a *branching* condition that divides the program into two non-communicating subprograms, each evaluating a different branch without cross-communication. Consequently, it is important for a subprogram to be aligned only with devices that follow the same sequence of computations and functions.

While visiting the function definition, the plugin only aligns branch conditions that involve aggregate computation. If a branch body does not involve aggregate computation, it will not be aligned. This default alignment ensures that all branches follow the branch semantics of AC. The alignment is then made by comparing the *Path* of the two functions, and if they are equal, the two functions are aligned.

2.4 Technologies

Kotlin Multiplatform *Kotlin* is a cross-platform, statically typed, general-purpose programming language with type inference. It is designed to interoperate fully with Java, and the JVM version of its standard library depends on the Java Class Library. However, *Kotlin* is also used to target JavaScript, and native code (via LLVM). The purpose of KMP is to streamline the development process of cross-platform projects by minimising the effort required to write and upkeep identical code for various platforms, such as Android, iOS, full-stack web applications, multiplatform libraries and its platform-specific implementations for JVM, JavaScript (JS), and native code.

KMP enables significant code reuse across multiple platforms, leading to faster development cycles, reduced maintenance efforts, and improved code consistency. KMP supports compilation targets for various platforms, including Kotlin/JVM for backend and desktop applications, Kotlin/JS for web applications, and Kotlin/-Native for Android, iOS, and other native platforms. Developers specify the compilation targets in the Gradle configuration to generate platform-specific binaries.

Kotest *Kotest* is a powerful testing framework for Kotlin that provides a flexible and expressive way to write tests for Kotlin projects. It offers a rich set of features and utilities to make testing easier, more concise, and more effective.

Kotest provides a wide range of built-in matchers and assertion functions for common test scenarios, such as equality checks, collection assertions, exception handling, and more. These utilities make it easy to write expressive and accurate assertions without boilerplate code. Some test examples will be shown in Section 3.1.

Gradle Gradle is a powerful build automation tool and dependency management system used primarily for Java, Kotlin, and Groovy projects. It is designed to be highly flexible, scalable, and efficient, making it a popular choice among developers and organisations for building and managing software projects.

Gradle uses a declarative DSL based on Groovy or Kotlin to define build scripts. This DSL allows developers to express build configurations, tasks, dependencies, and plugins in a concise and readable format. Gradle build scripts are typically named build.gradle and are written in Groovy or Kotlin.

Gradle organises the build process around tasks, which are units of work that perform specific actions, such as compiling source code, running tests, or generating documentation. Developers can define custom tasks and dependencies between tasks in the build script, allowing for fine-grained control over the build process.

Gradle executes tasks in parallel when possible, leveraging multi-threading to improve build performance.

2.5 Implementation

In this section, the actual implementation of the DSL's functionalities will be examined, with code examples and explanations regarding the implementation choices and the problems encountered during development.

2.5.1 Fields

About the manipulation of the *neighbouring values* introduced in Section 1.1.3, it has been implemented the concept of *Field* to handle the data structure to send between neighbours, and relatives functions to manipulate it.

The *Field* keeps track of the identifier of the device (`localId`) and the value associated with it (`localValues`) and eventually a map that associates the identifiers of the neighbours to their exchanged local values. In order to improve the performance of the system, a version of the *Field* has been implemented, called *ConstantField*. This is used to implement the mechanism of isotropic message, which is often used in the communications. The *ConstantField* keeps track of the identifiers of the neighbours with which to communicate in a list and the value to send only once, whilst the *Field* keeps track of the value to send for each neighbour as a map, which is more expensive in terms of performance.

2.5.2 Collektive entrypoint

The public entrypoint of the DSL is the *Collektive* class, which can be interpreted as a device that can execute an aggregate program.

As seen in the code snippet Listing 2.9, the *Collektive* class takes as parameters the *localId* of the device, the *network* used to manage the communication between devices, and the *computeFunction* that will be executed when the device is ready to compute the aggregate program.

Listing 2.9: The signature of the `Collektive` class.

```kotlin
class Collektive<ID : Any, R>(
    val localId: ID,
    private val network: Network<ID>,
    private val computeFunction: Aggregate<ID>.() -> R,
)
```

The aim of this class is to manage the logic and the effective execution of the rounds of the aggregate program passed as a parameter.

It provides two main functions that will be called in the incarnation to execute the program:

- `cycle`: applies once the *computeFunction* and returns the result of the computation;

- `cycleWhile`: applies the *computeFunction* while the condition passed as parameter is satisfied, and returns the result of the computation.

Both functions are implemented through a private function `executeRound`, which effectively calls the aggregate program, managing the result obtained and updating the internal state of the device.

There are two types of aggregate programs: one that uses the *Network* and one that does not, but directly takes a set of *InboundMessage*.

Listing 2.10: The signature of the `aggregate program`.

```kotlin
fun <ID : Any, R> aggregate(
        localId: ID,
        network: Network<ID>,
        previousState: State = emptyMap(),
        compute: Aggregate<ID>.() -> R,
    ): AggregateResult<ID, R> = with(AggregateContext(localId, network.read(), previousState)) {
        AggregateResult(localId, compute(), messagesToSend(), newState()).also {
            network.write(it.toSend)
        }
    }
```

The code seen in Listing 2.10 is the implementation of the `aggregate` function that relies on the *Network*. Taking the *AggregateContext* as context with the messages read from the network and the previous state of the device, it applies the *compute* function. The result of the computation is then returned as an *AggregateResult* (Listing 2.11), which contains the *localId* of the device, the effective *result* of the computation, the *messages to send* to other devices and the *new state* of the device. Finally, the messages to send are written to the network. This is the function called from the `executeRound` function.

Listing 2.11: The signature of the `aggregate result`.

```
data class AggregateResult<ID : Any, R>(
    val localId: ID,
    val result: R,
    val toSend: OutboundMessage<ID>,
    val newState: State,
)
```

2.5.3 Yielding Support

To operate on an initial value but possibly return a different value, it has been introduced the concept of *yielding*. The operations of yielding are used to operate on an initial value, which is usually exchanged with the neighbours, but possibly return a different value to the caller. This operation is executable on the `exchanging`, `sharing` and `repeating` methods, or it can be omitted in case the value obtained from the operation is to be returned. Those constructs will be explained in detail in the Section 2.5.4 and Section 2.5.5.

Listing 2.12: The signature of the `yielding context` class.

```
class YieldingContext<Initial, Return> {
    fun Initial.yielding(toReturn: () -> Return): YieldingResult<Initial, Return> =
        YieldingResult(this, toReturn())
}
```

2.5.4 Aggregate Context

The *AggregateContext* class is the class that effectively has the implementation of the basic constructs inside the DSL. Internally, it keeps track of the stack, the state, and the messages that need to be sent by that device.

Exchange There are two versions of this construct: one that has the same type of value of the field in output as the one in input (Listing 2.13), and one in which they differ (Listing 2.14). The `exchange` construct is the one that represents the communication in which e_r (the value to return) and e_s (the value to send) coincide, as illustrated in Section 1.1.3.

Listing 2.13: The implementation of the `exchange` function.

```kotlin
override fun <X> exchange(initial: X, body: (Field<ID, X>) -> Field<ID, X>): Field<ID, X> =
        exchanging(initial) { field -> body(field).run { yielding { this } } }
```

The `exchange` therefore simply calls the `exchanging`, but as the context of the yielding it passes the field on which the computation has been executed.

The real functioning of **XC** relies in the `exchanging` implementation, which is the one that effectively manages the actual logic of the computation and the messages to send to the neighbours.

In the Listing 2.14 snippet is shown how the `exchanging` function is implemented. First, it takes the current path from the stack kept inside the context and the messages received whose path is the same, so that it is possible to evaluate only aligned messages. Then it takes the state of the device at the current path, using the initial value passed as a parameter in case there is no previous state. A new field is then created on which to perform the computation, starting from the value of the state and the messages received at the current path. If there are no messages received at the current path, simply there will be an empty map of messages. The new field is then passed to the body passed in input, along with a new *yielding context*, on which the computation is effectively executed. The result obtained from the computation is then returned and, depending on what has been passed as a yielding context in the body, the value is sent to the neighbours, which can be either the result of the computation (in the `retsend` case) or a value of a different type.

A message of type *SingleOutboundMessage* is created, populated with the value obtained from the application to the field of the aggregate function put as the default value, while as an override it is evaluated what type of field has been obtained from the computation, and based on this the value is sent to the neighbours. If the field is of a constant type, the map of overrides will be empty, otherwise the value obtained from the computation is sent, excluding the value of the device itself. A check is made to verify that there are no messages aligned with the same path, as this could cause an alignment conflict. Finally, the message is added to the map of messages to be sent, and the state of the device is updated with the value obtained from the computation.

Listing 2.14: The implementation of the `exchanging` function.

```kotlin
override fun <Init, Ret> exchanging(
    initial: Init,
    body: YieldingScope<Field<ID, Init>, Field<ID, Ret>>,
): Field<ID, Ret> {
    val path = stack.currentPath()
    val messages = messagesAt<Init>(path)
    val previous = stateAt(path, initial)
    val subject = newField(previous, messages)
    val context = YieldingContext<Field<ID, Init>, Field<ID, Ret>>()
    return body(context, subject).also {
        val message = SingleOutboundMessage(
            it.toSend.localValue,
            when (it.toSend) {
                is ConstantField<ID, Init> -> emptyMap()
                else -> it.toSend.excludeSelf()
            },
        )
        check(!toBeSent.messages.containsKey(path)) {
            """
                Aggregate alignment clash by multiple aligned calls with the same path: $path.
                The most likely cause is an aggregate function call within a loop
            """.trimIndent()
        }
        toBeSent = toBeSent.copy(messages = toBeSent.messages + (path to message))
        state += path to it.toSend.localValue
    }.toReturn
}
```

Repeat The `repeat` construct is the one that models the state evolution over time. It is not implemented in terms of `exchange`, as it is a function that allows iterating over oneself, it can directly update its internal state without sending messages to neighbours.

The `repeat` construct takes as parameters the *initial* value and the *transform* function, and returns the final value of the computation of the same type as the initial. To return a different type, it has been implemented the `repeating` function, as for the `exchange` construct, using the *yielding support* (Listing 2.15).

Listing 2.15: The implementation of the `repeating` function.

```kotlin
     override fun <Initial, Return> repeating(
        initial: Initial,
        transform: YieldingScope<Initial, Return>,
    ): Return =
         transform(YieldingContext(), stateAt(stack.currentPath(), initial))
            .also { state += stack.currentPath() to it.toReturn }
            .toReturn
```

The `transform` function is applied to the state of the device, checking whether there are already values to use on the stack, or using the initial value passed as a parameter. After the computation, the stack is updated with the new value obtained from the computation, and the result is returned, depending on the yielding context passed in the body of the function.

2.5.5 Aggregate Operators

The functions implemented using the `exchange` mechanism are situated in a separate class called *AggregateOperators*.

Share The `share` is also implemented with the *yielding support* mechanism, using the same base concept as in the `exchange` and `repeat` functions. This means that it is possible to operate on an initial value, but possibly return a different value, as seen in the code snippet Listing 2.16.

Listing 2.16: The implementation of the `sharing` function.

```kotlin
fun <ID : Any, Initial, Return> Aggregate<ID>.sharing(
    initial: Initial,
    transform: YieldingContext<Initial, Return>.(Field<ID, Initial>) -> YieldingResult<Initial,
        Return>,
): Return = exchanging(initial) { field: Field<ID, Initial> ->
    with(YieldingContext<Initial, Return>()) {
        val result: YieldingResult<Initial, Return> = transform(field)
        field.map { result.toSend }.yielding {
            field.map { result.toReturn }
        }
    }
}.localValue
```

Given the *isotropic* nature of communication through `share`, the implementation via `exchange` is done in such a way as to send the same value to all neighbours, so there is no need to keep track of who sent the message, but only to send it to all neighbours. The value sent to the neighbours can be the value expressed thanks to the `yielding` function, or the value obtained from the computation, if the `share` function is used.

NeighboringViaExchange The `neighboringViaExchange` construct is used to access the values of the neighbours. It takes as parameter *local*, which can be an expression or a value, and returns a *field* of the same type (Listing 2.17).

Listing 2.17: The implementation of the `neighboringViaExchange` function.

```
fun <ID : Any, Scalar> Aggregate<ID>.neighboringViaExchange(local: Scalar): Field<ID, Scalar> =
    exchanging(local) { toYield ->
        toYield.mapToConstantField(local).yielding { toYield }
    }
```

As said in Section 1.1.3, the `nbr` construct is a peculiar case of `exchange`, in which the value of the expression evaluated is sent to neighbours and the values received from them are returned as a field. In this way, it provides a view of the values computed by the neighbours. Thus, its implementation consists of a call to the `exchanging` function, made in such a way as to send the value evaluated to the neighbours and return the values received from them as a field.

2.6 Alchemist Incarnation

Alchemist is a meta-simulator for pervasive computing and distributed systems, it is based on general abstractions that can be mapped to specific use cases. The meta-model consists of a set of entities (the *nodes*) that exist in an *environment* and interact with each other with relationship rules. The nodes contain a set of *molecules*, representing the variables, and *reactions* that represent the events that occur based on a set of conditions, then producing an effect described as an *action*. Thanks to this abstraction, the simulator is adaptable and flexible to different use cases and quantity of nodes, maintaining a consistent structure.

An "incarnation" serves as the interpreter enabling the *Alchemist Simulator* to comprehend and accurately execute a language. The *incarnation* models molecules and actions. Specifically designed for *Collektive*, this incarnation employs reflections to locate the aggregate entry point and dictates the methodology for each iteration of the aggregate program. Similar to other Alchemist incarnations like `alchemist-incarnation-scafi` and `alchemist-incarnation-protelis`, it has been implemented to launch simulations through Gradle tasks.

The goal is to ensure that the new implementation is still compatible with the simulator and that it can be used to run simulations without any issues.

To achieve this, it was necessary to modify the *Collektive Device* previously implemented, to make it compatible with the new implementation of the network and the related message exchange. Furthermore, the functioning of the *Distance Sensor* has been updated, and a new sensor has been added to detect the properties of the molecule of interest (*Local Sensing*). To run simulations starting from a YAML file with the appropriate *Alchemist* configuration, it was necessary to create a class that reads the configuration and sets up the simulation.

The *Alchemist* simulator is based on a system of actions and reactions, so to run an incarnation, it is necessary to define an action that represents the behaviour of the nodes. In this case, it is `RunCollektiveProgram` (Section 2.6).

The incarnation is therefore provided to manage the behaviour of the molecules or nodes, so overriding the methods provided by the `Alchemist` library.

In the Figure 2.7 is shown the structure of the *Alchemist Incarnation* module. It highlights the fact that the *CollektiveDevice* has to manage the specific sensors and actuators and implement the network from the device's perspective to manage messages sent and received from neighbours.

CollektiveDevice The *CollektiveDevice* is a representation of the device in the *Alchemist Simulator*. Its purpose is to manage specific sensors and actuators and implement the network from the device's perspective to manage messages sent and received from neighbours.

A device must have the *environment* in which it is located, the *node* property to be represented in the environment, a specific *ID* that is the same of the node, a *retention time* for messages which can be null if it is necessary to keep all messages. It must extend the eventual sensors and the network to manage the communication with the neighbours.

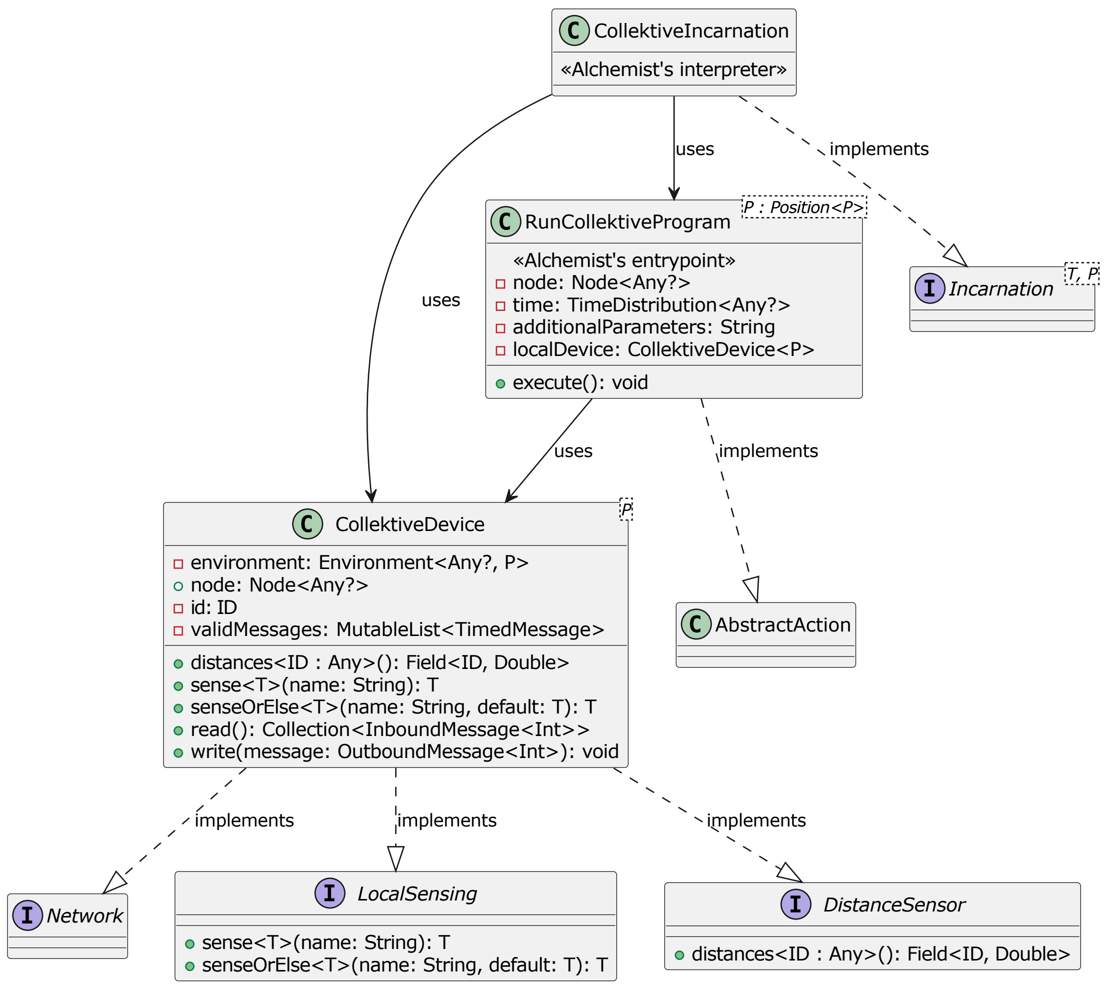
CollektiveIncarnation
«Alchemist's interpreter»
uses
implements
uses
RunCollektiveProgram
P : Position<P>
«Alchemist's entrypoint»
node: Node<Any?>
time: TimeDistribution<Any?>
additionalParameters: String
localDevice: CollektiveDevice<P>
execute(): void
Incarnation
T, P
uses
implements
CollektiveDevice
P
environment: Environment<Any?, P>
node: Node<Any?>
id: ID
validMessages: MutableList<TimedMessage>
distances<ID : Any>(): Field<ID, Double>
sense<T>(name: String): T
senseOrElse<T>(name: String, default: T): T
read(): Collection<InboundMessage<Int>>
write(message: OutboundMessage<Int>): void
AbstractAction
implements
implements
implements
Network
LocalSensing
sense<T>(name: String): T
senseOrElse<T>(name: String, default: T): T
DistanceSensor
distances<ID : Any>(): Field<ID, Double>

Figure 2.7: Class diagram of the *Alchemist Incarnation* module.

To keep track of the time when the messages arrived and thus to be able to discard them after a certain time, a private data class `TimedMessage` has been created that associates the time when each message arrived with the message itself. In this way, in the effective implementation of the `read` function of the network, it is possible to discard the messages that arrived before a certain time, and thus are no longer valid.

Listing 2.18: The implementation of the `read` function of the `Network`.

```kotlin
override fun read(): Collection<InboundMessage<Int>> =
    when {
        validMessages.isEmpty() -> emptySet()
        retainMessagesFor == null ->
            validMessages.map { it.payload }.also { validMessages.clear() }
        else -> {
            validMessages.retainAll { it.receivedAt + retainMessagesFor >= currentTime }
            validMessages.map { it.payload }
        }
    }
```

In the Listing 2.18 snippet is shown how the `read` function of the network is implemented. It checks if the *retainMessagesFor* property is null, meaning that it is necessary to keep all the messages, and in this case it returns the valid messages and clears the set, otherwise it retains only the messages that arrived after a certain time and returns them converted to a set of *InboundMessage*.

The `write` function is the one that effectively sends the messages to the network, it has to manage the messages taken as input from the *aggregate context*. It gets the neighbours of the device, and for each of them it creates an `InboundMessage` that the neighbour will read. It has been noticed that the implementation of the network's functions affects the performance of the aggregate programs, so it has been necessary to optimise the code to improve the performance.

If present, sensors and actuators are implemented within the *CollektiveDevice* class. For instance, in this particular implementation, the *CollektiveDevice* incorporates a *DistanceSensor* responsible for measuring the distance between the device and its neighbouring entities and a *LocalSensing* sensor for detecting the value of the molecule of interest. The method of the *DistanceSensor* is realised as an aggregate extension function, utilised for gauging the spatial separation between the device and its neighbours via the `neighboringViaExchange` construct. This method acquires the positions of neighbouring devices and computes the distances between them, as exemplified in code snippet Listing 2.19. The utility of this function extends to other aggregate programs by virtue of employing the `DistanceSensor` interface as a contextual receiver.

Listing 2.19: The implementation of the `distance` function.

```kotlin
override fun <ID : Any> Aggregate<ID>.distances(): Field<ID, Double> =
    environment.getPosition(node).let { nodePosition ->
        neighboringViaExchange(nodePosition).map { position -> nodePosition.distanceTo(position) }
    }
```

RunCollektiveProgram The *RunCollektiveProgram* is an action for the *Alchemist Simulator* that runs a *Collektive* program. An action in *Alchemist* models a change in the environment. It takes the *node* on which executes the action, the time distribution of the events and the *additional parameters* which is the path where the aggregate program to execute is located.

This action is designed to accept the program to be executed as a parameter of the YAML and instantiate it via reflection. Reflection stands as a potent feature within Kotlin, facilitating developers in the examination and manipulation of a program's structure during runtime. This capability enables access and modification of properties, methods, and types within a program, alongside dynamic invocation of functions and constructors. The reflection mechanism facilitates the retrieval of contexts passed to the aggregate program.

To instantiate the program, it is necessary to obtain any contexts passed through the context receivers or as parameters, instantiate them, and pass them to the program, along with the aggregated context required for its execution. This is where the behavior of the node is effectively defined when the action is executed, which calls the `cycle` method of `Collektive`.

Additionally, the implementation of a private cache has been essential for storing associated parameters of the aggregate program. This measure aims to circumvent the recurrent utilisation of the reflection mechanism during program execution, a practice known to substantially impact performance.

CollektiveIncarnation The *Collektive Incarnation* is an interpreter that enables the *Alchemist Simulator* to understand and execute the aggregate program written in the DSL. The incarnation is therefore provided to manage the behaviour of the molecules or nodes, so overriding the methods provided by the `Alchemist` library. It can evaluate runtime properties passed through the simulator, which are lambda functions. A cache system has been put in place to prevent the need for re-evaluating molecule's properties each time, activating only during system updates. This was necessary due to performance issues.

Running the simulations To execute the simulations outlined in the subsequent chapter (Section 3.2), the creation of a configuration file in YAML format, adhering to *Alchemist*'s YAML specifications, becomes imperative. Within the YAML configuration file, parameters defining the simulation are delineated, encompassing variables such as the node count, environmental characteristics, network model, and the pathway of the aggregate program slated for execution. Following the creation of the configuration file, simulation execution becomes feasible through Gradle tasks.

In the yaml configuration file, it is possible to specify the parameters of the simulation, such as the number of nodes, the type of environment, the network model, and the path of the aggregate program to run. Once the configuration file has been created, it is possible to run the simulation through Gradle tasks.

There are two types of Gradle tasks for each simulation: one is for testing purposes (`runTaskNameBatch`), and the other one is to run the simulation with the *Alchemist* GUI (`runTaskNameGraphic`). Tasks are configured in the `build.gradle.kts` file, and the parameters in which the two types of tasks differ are passed as arguments to the task.

Chapter 3

Validation

This chapter describes the validation process of the new DSL implementation and incarnation. Validating the hypotheses is a fundamental step for the correct evaluation of the work done. It is divided into testing and performance comparison, which are the two main aspects of this validation process. By comparing the performance of the new implementation with the original one, it is possible to understand if the introduction of the new features has led to an improvement in the performance of the system.

3.1 Tests

The testing phase is instrumental in affirming the integrity and functionality of the DSL codebase developed for this thesis. Tests serve as a critical mechanism for verifying the behaviour of the DSL across various scenarios, detecting potential bugs, and validating adherence to specifications.

Since the DSL is designed to be multiplatform, tests are written to ensure that the codebase is compatible with different platforms and the results across platforms must be consistent.

To achieve this, Kotlin offers *Multiplatform Extensions*, which allows the testing of the same codebase across different platforms, such as *JVM*, *JS*, and *Native*, just by adding needed extensions in the `build.gradle.kts` file.

For the testing phase, all the targets suppored by KMP have been used and are the following:

- Linux x64 and ARM64;
- Windows x64(MinGW);
- MacOS x64 and ARM64;
- IOS x64 and (simulator) ARM64;
- WatchOS x64 and (simulator) ARM64;
- TvOS x64 and (simulator) ARM64;

They cover the most common platforms and to ensure that the DSL is compatible with the most common devices, such as smartphones, tablets, and wearables. Test have been implemented using the *Kotest* framework, which is a flexible and comprehensive testing framework for Kotlin with multiplatform support.

Unit Tests Focused on individual units or components within a software system, unit testing serves to validate their functionality according to requirements. Typically conducted by developers as the initial testing phase, it involves automation and occurs each time modifications are made to the source code to prevent disruption of existing features. These tests are engineered to verify the smallest units of code, such as individual functions or methods, in isolation from the broader system context.

Integration Tests Integration testing is a software testing methodology used to evaluate the functionality of combined units of code. It serves to expose faults in the interaction between integrated units, ensuring that they function as expected. This type of testing is particularly useful in the context of the DSL as it allows the verification of the correct interactions of the different parts of the system.

3.1.1 Continuous Integration and Deployment

Continuous Integration (CI) and Continuous Deployment (CD) (CI/CD) are software development practices that aim to automate and streamline the process of delivering high-quality software. CI is a development practice where developers frequently integrate their code changes into a shared repository. Each integration triggers an automated build process, during which the code is compiled, tested, and verified against a set of predefined criteria.

The primary goals of CI are to detect integration errors early, ensure that the codebase remains functional, and promote collaboration among team members. CI key features include: i) automated builds, automatically triggered by committed code changes; ii) automated testing, run automatically during the build process; iii) immediate feedback, provided to developers regarding the status of their code changes, allowing to address issues promptly; iv) version control integration, enabling seamless integration with code repositories.

CD extends the principles of CI by automating the deployment process after successful integration and testing. It involves automatically deploying validated code changes to production or staging environments, eliminating manual intervention and reducing the time between code changes and their availability to users. CD helps streamline the release process, reduce deployment errors, and enable rapid and reliable software delivery. CD key features include: i) automated deployment, Deployments to production or staging environments are automated, ensuring consistency and reliability; ii) continuous monitoring, integrated into CD pipelines to track application performance and detect issues in real-time; iii) rollback mechanisms, in case of deployment failures; iv) environment provisioning, pipelines often include steps for provisioning and configuring target environments as part of the deployment process.

3.2 Alchemist Simulations

Another validation process regards the effective functioning of *Collektive Incarnation* for the *Alchemist* simulator. To see if the new incarnation is working as expected, examples of simulations have been implemented and run [1].

Using the Alchemist Simulator for validation showcases the capability of Collektive: implementing algorithms to specify system behavior is straightforward, and

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

execution is both swift and dependable.

Neighbour Counter The first example is a simple aggregate program in which the devices count the number of neighbours they have (Listing 3.1). The result is a map of the space in which each node has a value that represents the number of neighbours it has.

Listing 3.1: Neighbour counter code example

```
fun Aggregate<Int>.neighborCounter(): Int = neighboringViaExchange(0)
    .hood(0) { acc, _ -> acc + 1 }
```

The resultant simulation appears at first empty, because the nodes are coloured with a gradient that goes from white to blue based on the number of neighbours they have. Once the simulation starts and the nodes communicate with each other, the space is filled with colours, and the number of neighbours and connections is visible, as shown in Figure 3.1. The simulator also gives the opportunity to move the nodes around the environment, and the number of neighbours is updated in real-time.

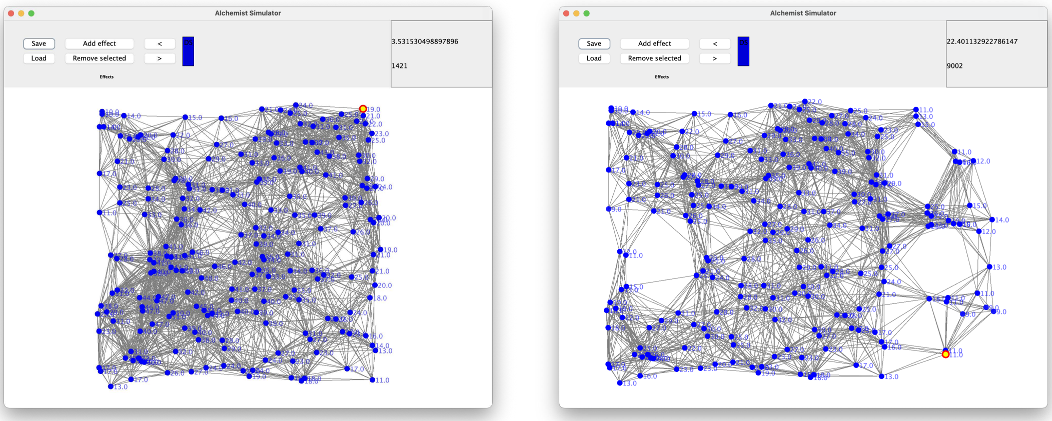

Figure 3.1: Neighbour counter simulation after some time ad after moving some nodes.

Gradient The second example is a simple gradient program in which the devices calculate the distance from a source node and communicate it to the other nodes

(Listing 3.2). The result is a map of the space in which each node has a value that represents the distance from the source node, changing the colour of the nodes based on the distance from the source (seen as a square in Figure 3.2). Also in this case, the simulator gives the opportunity to move the nodes around the environment, and the distance from the source is updated in real-time.

Listing 3.2: Gradient code example

```kotlin
context(LocalSensing,DistanceSensor)
fun Aggregate<Int>.gradientEntrypoint(): Double = gradient(sense("source"))

context(DistanceSensor)
fun Aggregate<Int>.gradient(source: Boolean): Double =
    share(POSITIVE_INFINITY) {
        val dist = distances()
        when {
            source -> 0.0
            else -> (it + dist).min(POSITIVE_INFINITY)
        }
    }
```

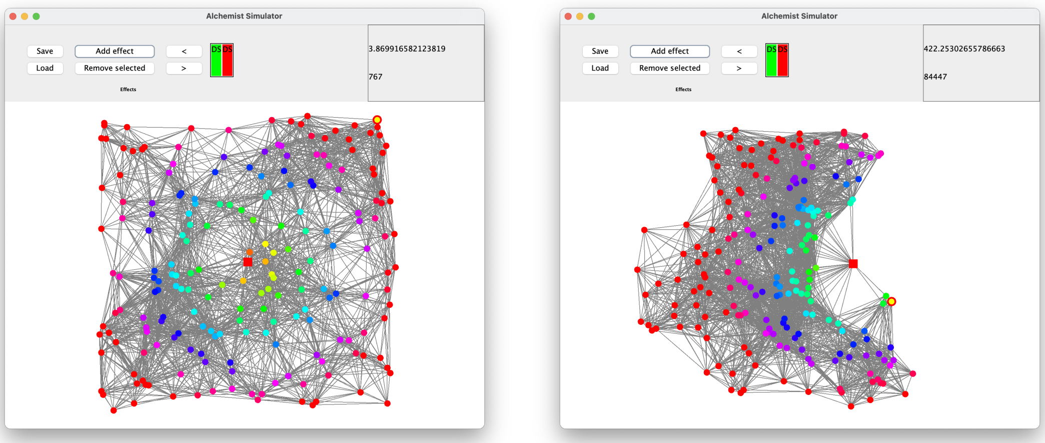

Figure 3.2: Gradient simulation after some time ad after moving some nodes including the root.

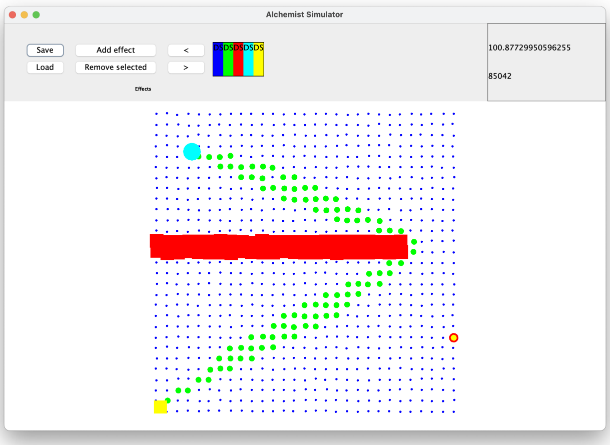

Figure 3.3: The resultant simulation of the channel with obstacles.

Channel with Obstacles The third example is a program a bit more complex, recreating a communication pathway (channel) within a distributed system where data transmission is impeded or affected by various obstacles. These obstacles could include network congestion, latency, limited bandwidth, or even physical barriers in certain distributed computing environments. In the context of aggregate computing, where computations are performed collectively by a network of interconnected devices or nodes, such obstacles can significantly impact the efficiency and effectiveness of communication and data exchange among the nodes.

From a source node to a target node, the goal is to find a minimum path that avoids obstacles and is the most efficient in terms of communication letting the information flow through the network (Listing 3.3).

As shown in Figure 3.3, the channel (in green) is gradually created from the source (in yellow) towards the target (in blue), and the obstacles (red) influence the trajectory of the channel, as expected.

Listing 3.3: Channel with Obstacles code example

```kotlin
context(LocalSensing, DistanceSensor)
fun Aggregate<Int>.channelWithObstacles(): Boolean =
    if (sense("obstacle")) {
        false
    } else {
        channel(sense("source"), sense("target"), channelWidth = 0.5)
    }

context(DistanceSensor)
fun Aggregate<Int>.channel(source: Boolean, target: Boolean, channelWidth: Double): Boolean {
    val sourceDist = gradient(source)
    val targetDist = gradient(target)
    val distBetween = distanceBetween(source, target)
    return !((sourceDist + targetDist).isInfinite() && distBetween.isInfinite()) &&
            sourceDist + targetDist <= distBetween + channelWidth
}

context(DistanceSensor)
fun Aggregate<Int>.distanceBetween(source: Boolean, target: Boolean): Double = broadcast(
     source, gradient(target))

context(DistanceSensor)
fun Aggregate<Int>.broadcast(source: Boolean, value: Double): Double = gradientCast(source,
     value) { it }

context(DistanceSensor)
fun Aggregate<Int>.gradientCast(source: Boolean, initial: Double, accumulate: (Double) ->
     Double): Double =
    share(POSITIVE_INFINITY to initial) { field ->
        val dist = distances()
        when {
            source -> 0.0 to initial
            else -> {
                val resultField = dist.alignedMap(field) { distField, (currentDist, value) ->
                    distField + currentDist to accumulate(value)
                }
                resultField.fold(POSITIVE_INFINITY to POSITIVE_INFINITY) { acc, value ->
                    if (value.first < acc.first) value else acc
                }
            }
        }
    }.second
```

3.3 Performance evaluation

To have a clear understanding of the performances of the new implementation, it is necessary to compare it with the *ScaFi* and *Protelis*' incarnations.

It is important to note that the performance discussed evaluates the execution speed of the language together with the relative incarnation. It is therefore possible that the competitors may have more performant results with a better implementation of the incarnation. In more complex experiments, the real effort falls on the language and the management of the fields, and not only on the incarnation.

The comparison is made by running the same simulations on each incarnation and comparing the results. The simulations [2] are run on the same machine to ensure that the comparison is fair, and that the differences are due to the implementation and not to the hardware.

The choice of scenarios for evaluating the performance of the new language in aggregate computing was based on several factors, each addressing different aspects of the language's capabilities and potential use cases. Here's a breakdown of the scenarios and the reasons for their selection:

1. Simple **state change**, involves simulating the dynamic evolution of device states over time intervals using the `repeat` construct. It was chosen to evaluate the language's efficiency in handling basic state changes, which are fundamental operations in aggregate computing systems;

2. A **counter of the neighbours**, implemented using the `neighbouring` construct, it allows for the effective calculation of spatial structures. This scenario was chosen to evaluate the language's ability to handle computations influenced by the density of the network, which affects the number of values present in a field and consequently the number of computations that a device has to perform. Assessing the language's performance in handling computations based on network density provides valuable insights into its efficiency in dynamically changing environments.

3. Simple **branching** operations, included to evaluate the language's performance in handling conditional execution paths. Since branching can be

[2] `https://github.com/angelacorte/collektive-benchmark`

resource-intensive, this scenario helps assess the efficiency of the language in managing multiple execution paths.

4. A **gradient**, which is a particular case of space-time variation implemented using the `share` construct, to associate each device in the system with its shortest distance to the nearest source. This algorithm is important because it is a fundamental building block for many other algorithms;

5. A **channel with obstacles**, involving computing a path between two points in a network, in this case a **source** and a **target**, avoiding obstacles and adapting to changes in network topology. It was chosen to evaluate the language's ability to handle dynamic network conditions and robustly adapt to changes, reflecting real-world scenarios where resilience and adaptability are crucial.

Overall, these scenarios collectively provide a comprehensive assessment of the new language's performance in various aspects relevant to aggregate computing, including state management, spatial computations, conditional branching, algorithmic building blocks, and adaptability to dynamic environments.

The first type of test is not influenced by the network density, as each node performs the operation on itself, independently of the number of neighbours, and there is no exchange of information between the nodes. In the other types of tests, the density of the network influences the computation, as the number of neighbours of a node affects the number of values present in a field and consequently the amount of computations that a device has to perform. Assessing the language's performance in handling computations based on network density provides valuable insights into its efficiency in dynamically changing environments.

Benchmark set up To evaluate the performance of the new language, a set of benchmarks has been designed to compare the performance of the new language with the existing ones. The tests have been run with the same parameters in the three different incarnations *Collektive*, *Protelis*, and *ScaFi*, within the *Alchemist* simulator. Each pair test-incarnation has been run ten times. The simulations have been performed on a network of medium density (around 30–40 neighbours per node), with a number of nodes equal to 200 and a communication radius of 7 for the first four types of tests, while for the channel test, the number of nodes has been increased to 800 and the communication radius to 10, lowering the density of the network to 10 neighbours per node. The termination condition of the simulation has been set to 1000 simulated seconds.

The results collected evaluate the execution time of the simulations in milliseconds, and the average execution time of the ten runs has been calculated. The results have been analysed and will be further discussed in this section.

Machine Specifications The results that will be presented have been obtained by running the simulations on a machine with the following specifications:

- **Processor**: Intel(R) Core(TM) i9-14900KF;
- **RAM**: 64GB 4800mhz;
- **OS**: Linux Manjaro;

Additionally, other test runs have been made on a different machine to ensure that the results are consistent across different hardware.

Field Evolution The first test has been implemented using the `repeat` construct, the results of the comparison are shown in Figure 3.4.

This construct entails a straightforward iteration over its respective field, incrementing the field value with each iteration for every node within the network. The `repeat` construct is used to manage the state of a system by performing state updates or transformations iteratively. This can be particularly useful in simulations, numerical computations, and real-time systems where state changes occur over time. Notably, in *Collektive*, this construct operates independently of neighbouring nodes, thereby ensuring enhanced performance by avoiding information exchange with neighbours.

The program simply increments the value of the field at each iteration; therefore, the results highlight that the difference of performance may be in the implementation of the field management inside the construct, showing that *Collektive* is faster than the other languages.

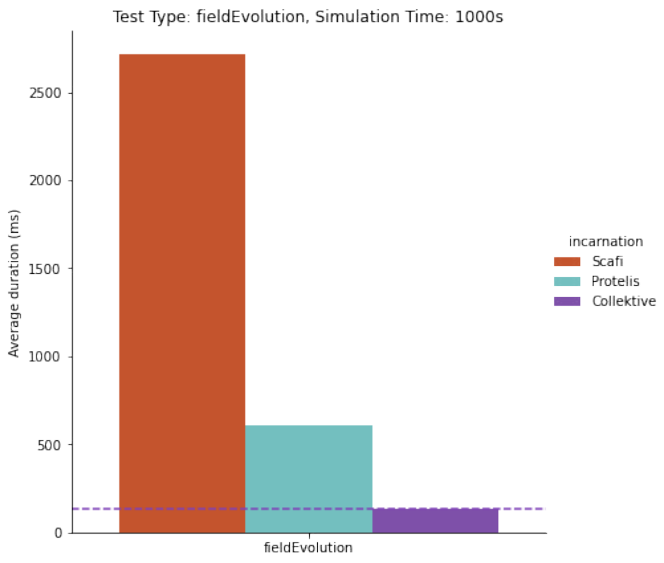

Figure 3.4: Graph of the results for the field evolution benchmark, showing that on average *Collektive* is 4.51 times faster than *Protelis* and 20.30 times faster than *ScaFi*.

Neighbour Counter The neighbour counter has been implemented using the `neighbouringViaExchange` construct, which is used to manage spatial structures and perform computations based on the information exchanged between the neighbours. The program is made in such a way to count the number of neighbours each node has, communicating with their neighbours to exchange information and increment the local value for each neighbour they have.

The network density influences this kind of operation, as the number node's neighbours affects the number of values present in a field and consequently the number of computations that a device has to perform. This means that the performances evaluated in this test are affected by the way the language manages the fields and the messages received from the neighbours.

From the results (Figure 3.5), it can be noticed that the difference with *Protelis* is minimal, while with *ScaFi* it is significant. This happens because *Protelis*, being an interpreted language, has advantages in the execution of less complex programs.

Branching "Branching" happens when a node has to make different evaluations based on a condition (as explained in Section 1.1.3). Comparison for this type of

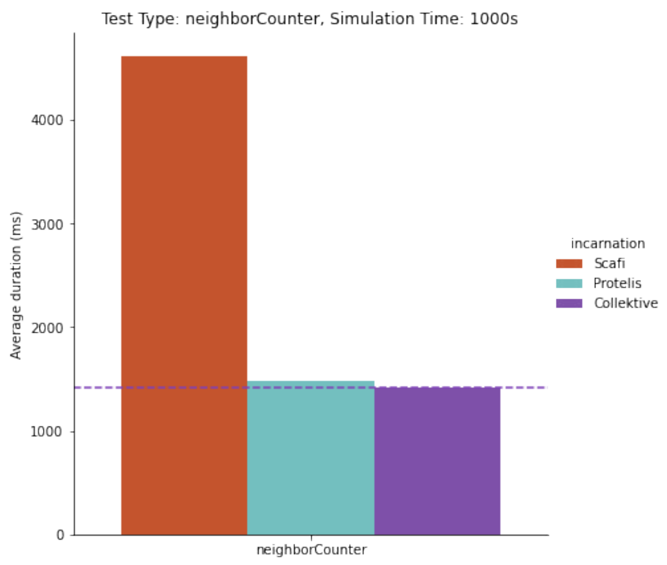

Figure 3.5: Graph of the results for the neighbour-counter benchmark, showing that on average *Collektive* is 1.04 times faster than *Protelis* and 3.25 times faster than *ScaFi*.

test is essential due to the observation that branching operations can be among the most resource-intensive tasks in terms of execution time, given that they create multiple subprograms and consequently multiple execution paths to be checked and managed. In *ScaFi*, branching is handled in a distinct manner, where exceptions are thrown to perform checks on the function to call, while in *Protelis* and *Collektive*, they follow a more conventional approach, aligning when a function is called.

Listing 3.4: Branching code example

```
fun Aggregate<Int>.branching() =
    if (sense("source")) {
        neighboringViaExchange(0).hood(0) { acc, _ -> acc + 1 }
    } else {
        0
    }
```

As shown in Listing 3.4, the program is made in such a way that if a condition is met, the node performs a simple neighbouring computation, that is the same as the one in the neighbour counter-test, otherwise it returns a value.

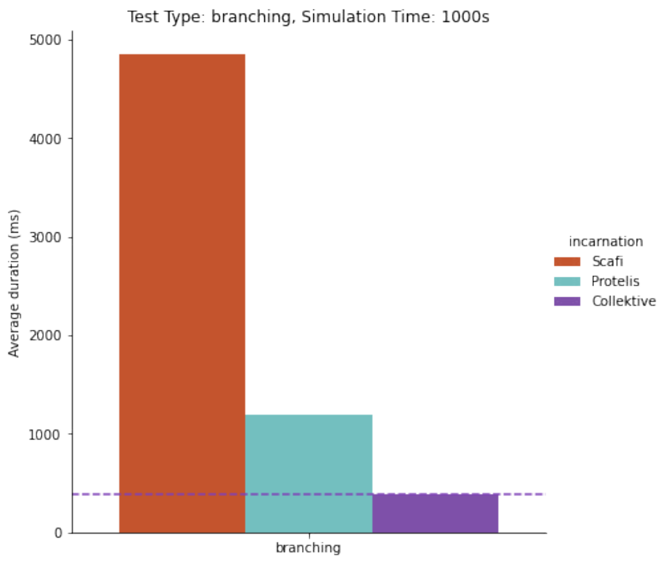

Figure 3.6: Graph of the results for the branching benchmark, showing that on average *Collektive* is 3.08 times faster than *Protelis* and 12.45 times faster than *ScaFi*.

The results depicted in Figure 3.6 illustrate the disparity in performance for this type of operation, underscoring the benefits of adopting a more traditional alignment approach, rather than the exception-based approach employed by *ScaFi*. Note that, being the same implementation as the neighbour counter, the results are influenced in the same way.

Gradient The gradient implemented with *Collektive* uses the `share` construct (based on the `exchange` construct) to calculate the distance from a source node and communicate it to the other nodes. For *Collektive* and *Protelis*, the apparent implementation is the same, while for *ScaFi* is different. This is because in *ScaFi* the `gradient` it is not implemented through the `share` construct, but it combines `rep` and `nbr` to achieve the same result. Apparently the implementation for *Collektive* and *Protelis* can be the same, but the execution is different, as *Collektive* leverage on the `exchange` construct, which manages the fields and the message exchange in a more efficient way.

The results in Figure 3.7 show that even with an increase in the amount of communication between nodes, *Collektive* does not lose performance compared to

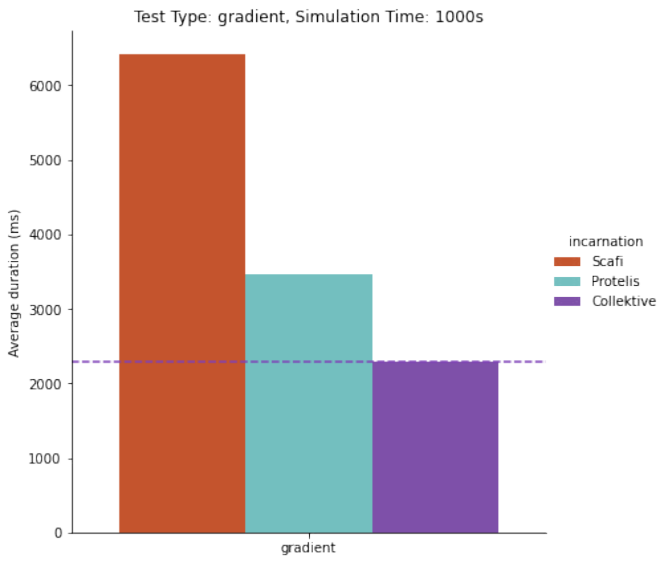

Figure 3.7: Graph of the results for the gradient benchmark, showing that on average *Collektive* is 1.51 times faster than *Protelis* and 2.80 times faster than *ScaFi*.

the other languages. This demonstrates how the management of the fields and the message exchange are the strength of *Collektive*.

Channel with Obstacles Among all the tests, the channel with obstacles is the most complex and the closest to a real-world scenario, because it simulates challenges encountered in actual deployments. It involves computing paths while considering obstacles, dynamically adapting to network changes, managing resources efficiently, and ensuring reliability despite obstacles. This scenario closely mirrors the complexities of real-world networks, making it a valuable test for evaluating aggregate computing systems in practical situations.

The program is implemented as in Listing 3.3, so it calculates many times the distances between nodes, using a bit more complex gradient implementation. This means that the performance of this test is influenced by the same factors that influence the gradient test.

The results in Figure 3.8 show that in complex programs, *Protelis* has a significant disadvantage compared to *Collektive*, while in the other examples they were closer. This is due to the fact that *Protelis* is an interpreted language and

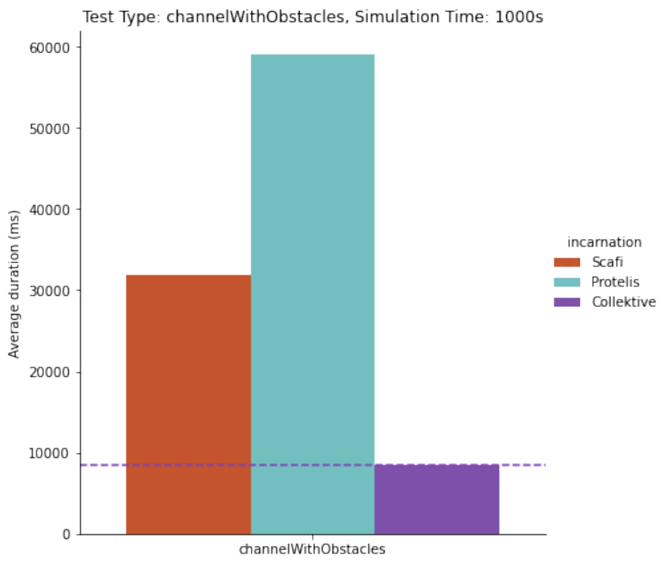

Figure 3.8: Graph of the results for the channel with obstacles benchmark, showing that on average *Collektive* is 6.91 times faster than *Protelis* and 3.73 times faster than *ScaFi*.

its execution of programs that need to go deeper in the execution tree is slower. This test emphasises the strength of *Collektive* in managing fields and message exchange, as the difference in performance is significant.

Changing the neighbourhood Performances may be related to the amount of communication between nodes, meaning that the more communication between nodes, the more time is spent in the execution of the program. For this reason, the experiments have been repeated with different network densities to see if the performance changes with the amount of communication, as the density increases, the number of neighbours a node has increases. Tests have been performed with low, medium, and high densities.

In the Figure 3.9, the speedup of the different incarnations with different network densities on the different tests is shown. The speedup is calculated as the ratio between the average execution time of the other incarnations and the average execution time of *Collektive*. The results show that *Collektive*, even with an increase in the amount of communication between nodes, is still faster than *Protelis* and *ScaFi*, with a significant difference in the most complex tests. This is a significant result, as it shows that the language is implemented in such way that can easily handle the increasing of the fields' size. This is a fundamental aspect, as the language is designed to be used in real-world scenarios, where the network density can change over time and can be high.

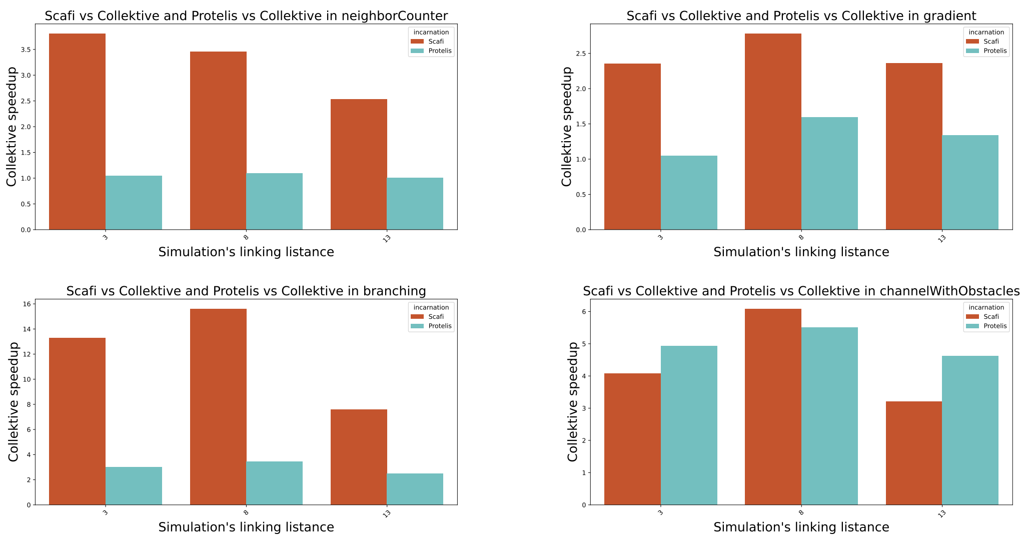

Figure 3.9: The speedup of the different incarnations with different network densities on the different tests. The speedup is calculated as the ratio between the average execution time of the other incarnations and the average execution time of *Collektive*.

Conclusions Keeping in mind that the performance of the language may be influenced by the implementation of the incarnation, the results show that *Collektive* actually is faster than *ScaFi* and *Protelis* overall, with a significant difference in the most complex programs, emphasizing the strength of *Collektive* in managing fields and message exchanges.

For the easier tests, the difference is minimal between *Collektive* and *Protelis*, while it is significant between *Collektive* and *ScaFi*. With the growth in complexity of the tests, and in the density of the network, the difference between *Collektive* and the other languages becomes higher, showing the strength of the combination of *Collektive*'s DSL and its incarnation.

Chapter 4

Conclusions

The primary objective of this thesis was to extend *Collektive* – a DSL for aggregate computing – by applying the principles of **XC** to enhance usability and offer a more expressive and flexible language for defining complex systems. For this purpose, it was necessary to explore and deepen the concepts of aggregate computing and **XC**, in such a way as to identify a solution able to guarantee the correctness of the language and its features, maintaining a good result in terms of performance.

First, an expanded version of the DSL was implemented, changing the syntax and the semantics of the language applying the principles of **XC**. By changing the semantics, it has been necessary to expand the alignment mechanism, to ensure the correct functioning of the language and its features. This was done maintaining the goal of having a multiplatform implementation with KMP, allowing the execution of the aggregate programs on different platforms.

Furthermore, an implementation of an incarnation for the *Alchemist* simulator has been provided, enabling the execution of aggregate programs, using the *Collektive* DSL, within its environment.

Finally, benchmarks have been implemented to compare the actual state of the art for aggregate computing – *ScaFi* and *Protelis* – and *Collektive*. The results showed that the combination of the DSL and incarnation developed in this thesis has an overall good performance.

4.1 Future Works

In the near future, the development of the DSL will continue, as it has been possible to win a research grant from the GARR organization. It will therefore be possible to further deepen the research and develop new features for the language.

As future works have been identified the followings:

- **Further optimisations of the DSL**: the DSL will be further optimised to improve its performance;
- **Alignment optimisation**: to improve performances and security looking forward to the execution of the aggregate program on different platforms and environments;
- **Development of a standard library**: to provide modules and functionalities to simplify the writing of the aggregate program. Such as self-stabilizing functions that could encompass a range of strategies commonly employed to attain adaptable and resilient decentralised behaviours; as the ones proposed by the *Protelis-Lang* library [FPBV17], a *Protelis*' API for resilient system design.
- **Creation of demonstrations**: to illustrate the possibility of running the same aggregate program on different platforms simultaneously, including server JVM, Android devices, iOS, and web browsers.

Bibliography

[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*, pages 54–71, Cham, 2019. Springer International Publishing.

[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*, pages 1–20, Cham, 2018. Springer International Publishing.

[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*, pages 285–299, Cham, 2021. Springer International Publishing.

[ACPV22] Gianluca Aguzzi, Roberto Casadei, Danilo Pianini, and Mirko Viroli. Dynamic decentralization domains for the internet of things. *IEEE Internet Computing*, 26(6):16–23, 2022.

[ACV22] Gianluca Aguzzi, Roberto Casadei, and Mirko Viroli. Addressing collective computations efficiency: Towards a platform-level reinforcement learning approach. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 11–20, 2022.

[ACV24] Gianluca Aguzzi, Roberto Casadei, and Mirko Viroli. Macroswarm: A field-based compositional framework for swarm programming, 2024.

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

[AWW18] Hany F. Atlam, Robert J. Walters, and Gary B. Wills. Fog computing and the internet of things: A review. *Big Data and Cognitive Computing*, 2(2), 2018.

[BPP$^+$14] Arne Brutschy, Giovanni Pini, Carlo Pinciroli, Mauro Birattari, and Marco Dorigo. Self-organized task allocation to sequentially interdependent tasks in swarm robotics. *Autonomous Agents and Multi-Agent Systems*, 28(1):101–125, 2014.

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

[BV16] Jacob Beal and Mirko Viroli. *Aggregate Programming: From Foundations to Applications*, pages 233–260. Springer International Publishing, Cham, 2016.

[CAPV23] Roberto Casadei, Gianluca Aguzzi, Danilo Pianini, and Mirko Viroli. Programming (and learning) self-adaptive e self-organising behaviour with scafi: for swarms, edge-cloud ecosystems, and more. In *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 33–34, 2023.

[Cas23] Roberto Casadei. Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. *ACM Comput. Surv.*, 55(13s), jul 2023.

[CBP15] Shane S. Clark, Jacob Beal, and Partha Pal. Distributed recovery for enterprise services. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 111–120, 2015.

[CFP+19] Roberto Casadei, Giancarlo Fortino, Danilo Pianini, Wilma Russo, Claudio Savaglio, and Mirko Viroli. Modelling and simulation of opportunistic iot services with aggregate computing. *Future Generation Computer Systems*, 91:252–262, 2019.

[CLZ00] G. Cabri, L. Leonardi, and F. Zambonelli. Mars: a programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[CPP+20] Roberto Casadei, Danilo Pianini, Andrea Placuzzi, Mirko Viroli, and Danny Weyns. Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment. *Future Internet*, 12(11), 2020.

[CV19] Roberto Casadei and Mirko Viroli. Coordinating computation at the edge: a decentralized, self-organizing, spatial approach. In *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 60–67, 2019.

[CVAD20] Roberto Casadei, Mirko Viroli, Giorgio Audrito, and Ferruccio Damiani. Fscafi : A core calculus for collective adaptive systems programming. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*, pages 344–360, Cham, 2020. Springer International Publishing.

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

[DPV13] S Montagna D Pianini and M Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, 2013.

[Fer15] Alois Ferscha. Collective adaptive systems. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*, UbiComp/ISWC'15 Adjunct, page 893–895, New York, NY, USA, 2015. Association for Computing Machinery.

[FPBV17] Matteo Francia, Danilo Pianini, Jacob Beal, and Mirko Viroli. Towards a foundational api for resilient distributed systems design. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 27–32, 2017.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, jan 1985.

[JMB05] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, aug 2005.

[MZ09] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4), jul 2009.

[Pia21] Danilo Pianini. Simulation of large scale computational ecosystems with alchemist: A tutorial. In Miguel Matos and Fabíola Greve, editors, *Distributed Applications and Interoperable Systems*, pages 145–161, Cham, 2021. Springer International Publishing.

[PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, page 1846–1853, 2015.

[VAB$^+$18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.*, 28(2), mar 2018.

[VBD+19] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109:100486, 2019.

[VDPBMS05] H. Van Dyke Parunak, S.A. Brueckner, R. Matthews, and J. Sauter. Pheromone learning for self-organizing agents. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 35(3):316–326, 2005.

[ZCF+11] Franco Zambonelli, Gabriella Castelli, Laura Ferrari, Marco Mamei, Alberto Rosi, Giovanna Di Marzo, Matteo Risoldi, Akla-Esso Tchao, Simon Dobson, Graeme Stevenson, Juan Ye, Elena Nardini, Andrea Omicini, Sara Montagna, Mirko Viroli, Alois Ferscha, Sascha Maschek, and Bernhard Wally. Self-aware pervasive service ecosystems. *Procedia Computer Science*, 7:197–199, 2011. Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

Acknowledgements

Benvenuti nell'unica sezione che molti di voi vorranno leggere, o per lo meno, in quella più facile da capire. Sì, non c'è di che, questa parte è in italiano. Parto chiedendo scudo perchè non sono molto sentimentale e faccio un po' schifo a scrivere queste cose, figuratevi se avessi scritto in inglese pure sta parte. Ma, in ogni caso, ci provo.

Vorrei innanzitutto ringraziare il mio relatore, il Prof. Danilo Pianini, per avermi dato l'opportunità di lavorare a questo progetto e di unirmi al suo gruppo di ricerca. Un grande grazie anche a Nico (aka dottor correlatore), che mi ha sOpportata e aiutata durante tutto il percorso di tesi.

Scrivere queste tre righe è stato veramente impegnativo. Ma comunque sia, ci tengo a ringraziare tutti i nuovi amichetti conosciuti durante quest'esperienza, dal *Lab4.0* con Marti, Nico, Ange (il gioco), Magnus, Ruslan, Giovanni^2, al *PsLab* con Gianlu, Dom e Samu, a tutti quelli di "Disagio in 4.0++". Grazie anche per tutte le risate fatte allo Shtop e alle varie serate assieme.

Grazie anche alle *Ragatte* e agli amiconi di una vita di Riolo: Arlessia, Smeo, Bananna, Sarabb, Diego, Rava, Fronz, Cri, Dave, Maicolle, Pulti e Santa; perdonate la mia trasformazione fantasma in questi ultimi mesi, rimedieremo asap.

Non ringrazio gli "Angeli" (*'s Angels) perchè sono loro che devono ringraziare me per la mia presenza nella loro vita. Dai scherzo (ma se volete non scherzo), grazie per i pranzi e le boiate (e le collab nei progetti) a Leo, Angelo, Spaolo, Fosco, Eddie, Davide e Filo. Prego per gli stickers e per i photoshoppini, è stato un piacere.

Grazie ai miei amiconi del *Quesito scientifico di importanza fondante* per la compagnia, le sessioni di confessionale, motivazionali e di sclero.

Onorevol mention anche a tutti gli amici conosciuti durante gli anni dell'uni

da *inform-erda* a *Notice me [...]*, e tutti gli altri, siete tanti e se vi nominassi tutti questa tesi diventerebbe un libro. Altra onorevol mention a tutti i miei amiconi cavallari del Delta e non.

Grazie a chi c'è sempre stato e a chi è stato solo di passaggio, siete stati tutti importanti.

Ultimi ma non per importanza, ringrazio tutta la mia famiglia, che mi ha sempre supportata e lasciata libera di scegliere il mio percorso con fiducia.

So che alcuni di voi si aspettavano qualche battuta "non convenzionale", ma penso sia stato più faticoso scrivere questi ringraziamenti che la tesi stessa (sono in burnout totale scusate). Andate in pace.