ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

MSc in Computer Science and Engineering

Design and development of a Rust-based execution platform for Aggregate Computing

Thesis in:
Pervasive Computing

Supervisor
Prof. Viroli Mirko

Co-supervisor
Dott. Aguzzi Gianluca

Candidate
Micelli Leonardo

IV Graduation Session
Academic Year 2022-2023

Abstract

The rapid expansion of the Internet of Things has led to the proliferation of computational resources in the physical world, which are now embedded in everyday objects and environments. The Aggregate Computing (AC) has emerged as a promising approach to tackle the complexity of designing and coordinating these systems, by shifting the focus from individual devices to programming the global behavior of whole computational collectives. There are several state-of-the-art implementations of this paradigm, one of them being Scala Fields (ScaFi), which targets the Java Virtual Machine (JVM). Concurrently, other implementations have been developed to bring AC also to resource-constrained, "thin" devices that cannot support the JVM, such as FCPP, which is based on the C++ programming language. The Rust Fields (RuFi) project aims to democratize the development of AC applications by exploiting the Rust programming language's features of performance, safety and expressiveness to provide a minimal functional core for AC that can be used on multiple platforms, including thin devices. In this paper, we will present the design and development of a module for the RuFi framework that will enable the distributed execution of RuFi-based aggregate programs.

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."

(Mark Weiser, 1991)

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Prof. Mirko Viroli, for his guidance and support throughout the development of this thesis. I also could not have undertaken this project without the support of my supervisor, Doct. Gianluca Aguzzi and the valuable advice and time he dedicated to me. I am also grateful to Doct. Nicolas Farabegoli and professors Roberto Casadei and Danilo Pianini for their help and support. A special thanks go to my group of friends first and colleagues later, the 'Angels'": Angelo, Paolo, Davide, Angela, Filippo, Eddie and Francesco. I would also like to thank my dear friend and fellow rustacean and gymbro Luca, as well as all the people who have been close to me during my years in Cesena: Massimo, Marco, Schiaro, Riccardo, Andrea and all the amazing people I met during this wonderful adventure. I'm also grateful to my amazing group of friends at home, my "Branco": Dona, Cristian, Riki, Claudio, Francesco, Simone, Aldo and Raffaele, we all met in childhood and have never been apart since. Last but not least, I would like to thank from the bottom of my heart my mother Simona and all of my family for being my best supporters and for always being there for me.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

The current trends from Internet of Things (IoT) are ushering in a new era for the interaction between humans and computing devices. The steady growth in the number of objects that are embedded with computational power and connection capabilities presents numerous opportunities and challenges. This environment has favored the birth of a new vision of the future of computing: Pervasive Computing (PC) [Sat01]. According to this vision, computational resources, which are now deeply intertwined with the physical world to the point of being almost invisible, operate and coordinate in an increasingly dynamic environment, where decentralized and peer-to-peer interactions are expected to be increasingly important. In this world, the AC paradigm offers an encouraging shift of focus on the design of the systems of the future, where the emphasis will now be on the global behavior of collections of devices, rather than on the individual devices themselves [BPV15]. The development of this new field has already led to the creation of very important and well-researched computational models, languages and frameworks. However, the current state of the art in AC frameworks are for the most part based on the JVM, exploiting its vast ecosystem. One important example of such a framework is ScaFi: a project that utilizes the JVM-based Scala language to deliver a powerful and flexible Domain Specific Language (DSL) and toolkit for AC. Nevertheless, relying on the JVM has some drawbacks, one of the most important being the fact that it comes with additional costs in terms of memory and computational resources, which we cannot assume are always available on every

device that could be part of the PC vision. It is in this perspective that the FCPP framework was born: to exploit C++'s performance and widespread support on many architectures to bring AC to thin devices. Within this varied context, the RuFi project aims to leverage the Rust programming language's features of memory safety, performance and expressiveness to provide a minimal functional core for AC that can be used on multiple platforms, including thin devices, and build high-level APIs on top of it. This thesis' work takes the core features developed in the RuFi project and builds up from them by testing and improving the API of the previous work and using them as a basis for the design and development of a Rust-based platform that will enable the distributed execution of aggregate programs written in Rust, taking a step towards the goal of bringing AC to thin devices. By the end of this thesis, it will be possible to write aggregate programs in Rust using a comprehensively tested core set of constructs and then execute them on a distributed network of devices.

Structure of the Thesis The structure of the thesis is designed to provide a basis for its context and objectives, and then use it as a foundation to fully describe the proposed solution. First, a comprehensive overview of important concepts will be given in the Background section (Chapter 2). Here, the reader will be introduced to the concepts of Field Calculus (FC), AC and the state of the art regarding FC implementations, as well as the main concepts and features of the Rust programming language. The main goal of the thesis and the requirements to achieve it will be presented in the Analysis and Requirements section (Chapter 3). Then, the thesis will describe the proposed solution in the Design section (Chapter 4). Here, the reader will be introduced to the RuFi framework, starting from the high-level architecture and then diving into the details and functioning of its modules. In particular, a thorough description of notable implementation challenges and choices made will be given in the Implementation section (Chapter 5). Finally, the Validation section (Chapter 6) will describe the validation process for the RuFi framework, which has been done on multiple axes, including unit testing, integration testing, user acceptance testing and memory profiling.

Chapter 2

Background

Presented in this chapter are some core concepts that serve as a knowledge base upon which this thesis is built. First, we will introduce the reader to the concept of AC. Then we will examine ScaFi, a state-of-the-art implementation of AC, that inspired this thesis's project. Finally, we will introduce the Rust programming language, which is the language of choice for the implementation of the project.

2.1 Aggregate Programming and Field Calculus

Aggregate programming [BV16] is a programming approach that aims to shift the focus on the individual device perspective that is typical of traditional programming approaches, which inevitably entangles the system's behavior design with aspects of distributed systems design (such as efficient and reliable communication, coordination and fault tolerance) to an approach that raises the abstraction level from individual devices to large aggregations of devices. It does so by exploiting the concepts of computational fields and FC [AVD+19, VDB13].

Within the FC, a *computational field* is a function mapping every computational device in a network, represented by a dynamic and reflexive neighboring relationship between devices, to a computational object. Depending on the computational object in question, there can be many examples of computational fields:

- **Scalar fields**: a field that maps every device to the value of some sensor

reading;

- **Vector fields**: a field that maps every location in the network to a set of the best routes to reach it;
- **Boolean fields**: a field that represents the area around an object of interest;

The FC main goal is to "capture a set of key ingredients of programming languages supporting the creation of computational fields: composition of fields, functions over fields, the evolution of fields over time, construction of fields of values from neighbors, and restriction of a field computation to a sub-region of the network [VDB13]".

This calculus is based on the idea of "expressing aggregate system behavior by a functional composition of operators that manipulate (evolve, combine, restrict) continuous fields [VDB13]"

A key concept of Field Calculus is that these aggregate-level specifications can also be interpreted as a local set of rules that define the iterative asynchronous execution of *computation rounds*. The local, round-based computational model for device δ is represented in the fig. 2.1

Computational Model

1. sleep for some time;
2. gather incoming messages from neighbors in the form of *neighboring fields* mapping neighbors identifiers to their shared computation values;
3. perceive contextual information through sensors;
4. retrieve stored information about the previous round execution;
5. evaluate the program P, manipulating the data values received by neighbors, perceived from the context or retrieved from local memory;
6. store some data to be used in the following round and emit a message to all neighbors with information about the computation outcome;
7. go back to sleep.

It is said that the device δ *fires* when performing the steps 2-6.

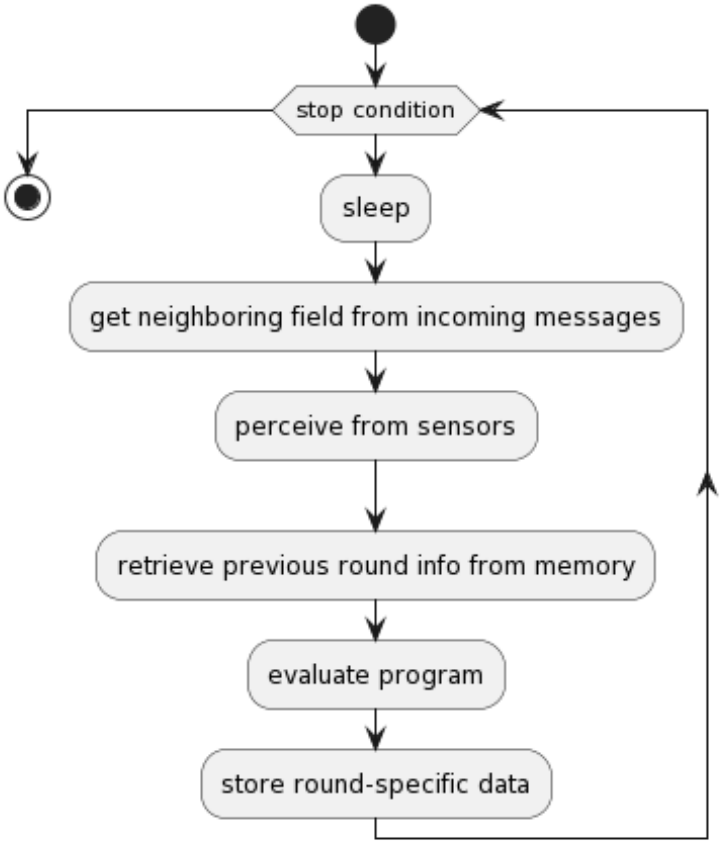

Figure 2.1: The local, round-based computational model for device δ

2.1.1 Field Calculus' Syntax

The core syntax of the FC is shown in fig. 2.2, where we can see the following elements:

- a program P, consisting of a sequence of function declarations and a main expression;
- function declarations F, which consists of the name of the function d, a list of variables $\bar{x}$ representing parameters, and a function body e;
- expressions e model an entire field evolution. A more detailed explanation of expressions is given in section 2.1.2;

$P ::= F\ e$	program
$F ::= def\ d(\bar{x})\{e\}$	function declaration
$e ::= x \mid v \mid (\bar{x}) \rightarrow e \mid if(e_0)\{e_1\}\{e_2\} \mid nbr\{e\} \mid rep(e_0)\{(x) \rightarrow e\}$	expression
$f ::= d \mid b \mid (\bar{x}) \rightarrow e$	function name
$v ::= l \mid \phi$	value
$l ::= c(\bar{l}) \mid f$	local value
$\phi ::= \bar{\delta} \mapsto \bar{l}$	neighbouring field value

Figure 2.2: Field Calculus syntax[AVD+19].

- a value v can be either a *neighboring field* ϕ or a *local value* l. When the device δ fires, l represents data produced by δ, while ϕ represents a field that maps neighbors of δ to their local values;
- in a higher-order extension of the model proposed in [AVD+19], l can be either a data value or a function value;

2.1.2 Informal Semantics

Hereby are presented the four core field manipulation expressions, previously mentioned in the section 2.1.1:

- $rep(e_0)\{(x) \rightarrow e\}$ is the "repeat" construct, representing *time evolution* and it is used to dynamically changing fields. At each computation round, the device δ yields the result of the application of the anonymous function $(x) \rightarrow e$ to the value of the rep expression at the previous round, then the same anonymous function is applied to the initialization expression e_0;
- $nbr\{e\}$ is the *neighboring field construction* expression, modeling device-to-neighbor interaction and mapping each neighbor of δ to the result of the expression e;
- $if(e_0)\{e_1\}\{e_2\}$ [1] represents *domain restriction*. It is a lazy-evaluating branch

[1] In some of the current implementations of FC, including the one presented in this thesis, this expression is often called "branch"

construct, computing e_1 on devices in the restricted domain D_t where e_0 is true and e_2 on devices in the restricted domain D_f where e_0 is false. Since the device *delta* does not compute the other branch of the expression, there are two important consequences:

  - any $nbr\{e\}$ expression in the opposite branch of the domain cannot communicate with the device δ since it never computes it;
  - if δ evaluated e_1 in its previous rounds, all rep-expressions in e_2 will start from scratch. Similarly, values stored for rep-expressions in e_1 will be lost so that also they will start from scratch in the next round.

  This means that the evaluation of e_1 and e_2 proceeds in complete isolation from one domain to the other;

- $e(e_1, ..., e_n)$, where $n >= 0$ and e evaluates to a field of function values, is the *function call* expression.

2.2 The ScaFi Framework

ScaFi is one of the most actively researched and maintained implementations of AC. It is hosted in the Scala language, a powerful and expressive JVM-based language that brings functional programming to the JVM ecosystem to achieve expressiveness, safety and scalability. Although ScaFi is not the only AC implementation, compared to other aggregate programming languages it "provides a more high-level platform that might support agile prototyping for research and easier integration with other tools and environments for distributed systems (cf. the Web and Android) [AVD+19]", representing a valuable tool for scientific research.

ScaFi Architecture

ScaFi, as a software artifact, consists of Scala DSL and API modules for writing, testing and running aggregate programs. The architecture of the ScaFi framework shown in figure 2.3 consists of the following components:

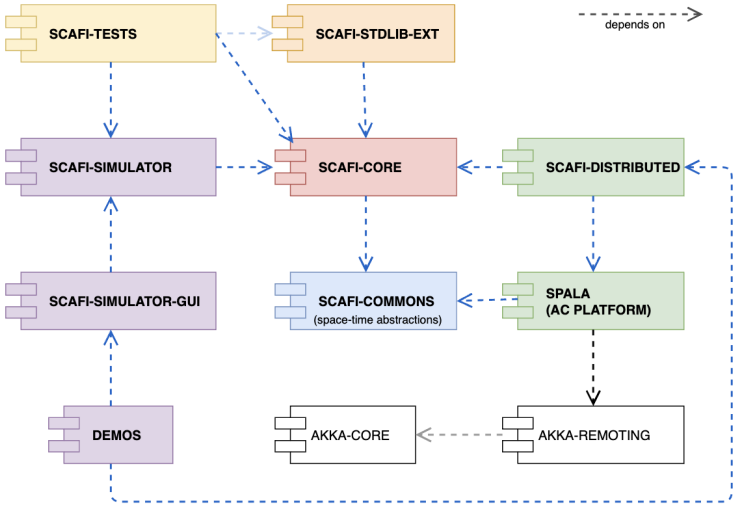

Figure 2.3: The ScaFi Architecture

- **scafi-commons**: provides basic abstractions and utilities such as temporal and spatial abstractions;
- **scafi-core**: provides the aggregate programming DSL, consisting of syntax, semantics and a virtual machine together with a standard library of functions;
- **scafi-stlib-ext**: provides extra functionalities that require external dependencies and hence are kept separate from the core;
- **scafi-simulator**: provides a basic support for simulating aggregate systems;
- **scafi-simulator-gui**: provides a graphical user interface for the simulator;
- **spala**: provides an actor-based aggregate computing middleware based on the Akka framework;
- **scafi-distributed**: provides an integration layer between Scafi and spala;

In particular, this thesis follows up on a project that will be later introduced which took inspiration from a subset of the scafi-core module.

2.3 FCPP

FCPP [Aud20] is an implementation of the FC model based on the C++ programming language. It has been designed and developed to bring the AC paradigm to resource-constrained devices that cannot support the JVM. It does so by providing an extensible C++ library and a performance-oriented simulator that allows the developer to speed up the development process of aggregate programs.

FCPP Architecture

The FCPP library consists on several header files, divided in three main conceptual layers, as shown in fig. 2.4

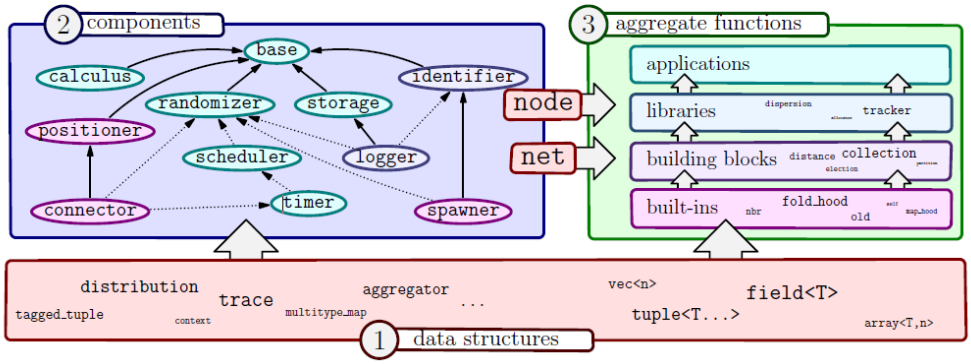

Figure 2.4: The FCPP Architecture

- **components**: this layer provides core abstractions such as *node* and *net*, that are obtained by combining a sequence of *components* in a mixin-like fashion;
- **aggregate functions**: this layer provides the actual implementation of the FC as templated functions that use *node* and *net* in their specification;
- **data structures**: this layer provides the implementation of the data structures used by the *components* and the *aggregate functions* layers.

2.4 The Rust Programming Language

The Rust Programming Language[KN] is a multi-paradigm, general-purpose programming language designed originally for systems-level development. It strives to achieve both execution speed and memory safety and efficiency while providing zero-cost abstractions and high-level features that are unusual for low-level programming languages such as C or C++. In this section, we will go through Rust's main features and asses whether this language is suitable to develop an AC implementation that can run on thin devices or not.

2.4.1 Rust's Basic Features

Variables and mutability

Like the majority of today's programming languages, Rust supports storing values inside variables for referencing them in various sections of the program.
The developer can store a value inside a variable through a *let* statement:

```
let x = 5;
```

In Rust, even if it is a statically typed language, the type of the variable can be omitted thanks to the type inference mechanism. This means that the compiler can figure out the type of the variable by looking at the value assigned to it. In this case, the type of x is *i32*, which is a 32-bit signed integer.

Another important feature of Rust variables is that they are immutable by default. This means that once a value is assigned to a variable, it cannot be changed. For example, the following code will not compile:

```
let x = 5;
x = 6; // error: cannot assign twice to immutable variable ‘x‘
```

Instead, the developer can opt out of the mutability by using the *mut* keyword:

```
let mut x = 5;
x = 6; // this code compiles
```

Data Types

The Rust language supports a wide range of data types that can be both found in low-level programs and in high-level designs. These data types can be divided into two main categories: scalar types and compound types. For scalar types, the following are supported:

- **Integers**: both signed and unsigned integers of different sizes. In particular, rust supports 8, 16, 32, 64, and 128-bit signed and unsigned integers;
- **Floating-point numbers**: both 32 and 64-bit floating-point numbers;
- **Booleans**: a boolean type that can be either *true* or *false*;
- **Characters**: the language's most primitive alphabetic type, represented by a single Unicode scalar value.

For compound types, the following are supported:

- **Tuples**: the simplest form of product type in Rust, represented by a collection of values of possibly different types;
- **Arrays**: a collection of values of the same type. Unlike other languages, Rust arrays have a fixed length.

In addition to these compound types, Rust offers several other collections; for example:

- **Vectors**: a collection of values of the same type. Unlike the arrays, Rust vectors have a dynamic length;
- **Strings**: a growable UTF-8 encoded string type;
- **Hash Maps**: a collection of key-value pairs, implemented as a hash table.

2.4.2 The Ownership System

Rust's Ownership System is its most unique feature and is a core part of how the language achieves memory safety without the need for a garbage collector. The term *ownership* refers to a set of rules that govern how a program's memory is managed and it is enforced by the compiler, meaning that if a program violates them, it won't compile. This means that none of these features will cause runtime overhead for the program.

Ownership Rules

The Rust's ownership rules are the following:

1. Each value in Rust has an owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

This means that a variable's validity (and presence in memory) is tied to the scope of the variable's owner: when the owner's scope is over, the compiler will automatically call the drop function on every owned variable, freeing the memory associated with it and making it so that the variable is no longer valid.

Moving and Copying

The ownership system has implications on what happens when a variable of a certain type is copied. For example in the following code:

```
1    let x = 5;
2    let y = x;
```

The value of x is copied into y. This means that there are now two variables on the stack both with the value of 5. This is possible because x is an integer-type variable, and integers have a fixed and known size at compile time, so they can be pushed cheaply onto the stack.

However, if we analyze the following code:

```
let s1 = String::from("hello");
let s2 = s1;
```

Since s1 is a String type, which does not have a known size at compile time, s1 will consist of a pointer in the stack, pointing to a heap-allocated memory that contains the actual string data, as shown in the fig. 2.5.

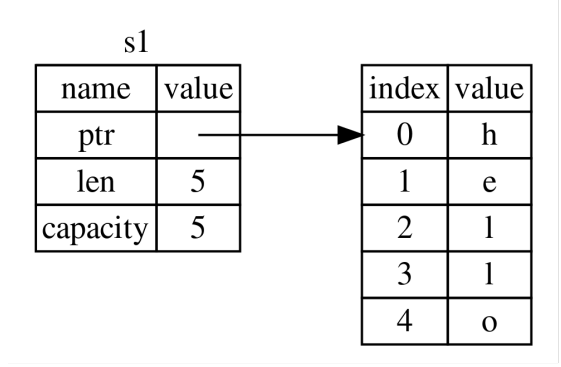

Figure 2.5: Representation of the memory layout of a string in Rust

When s1 gets copied into s2, only the pointer in the stack is copied, so that the memory layout of the program will look like the one in fig. 2.6.

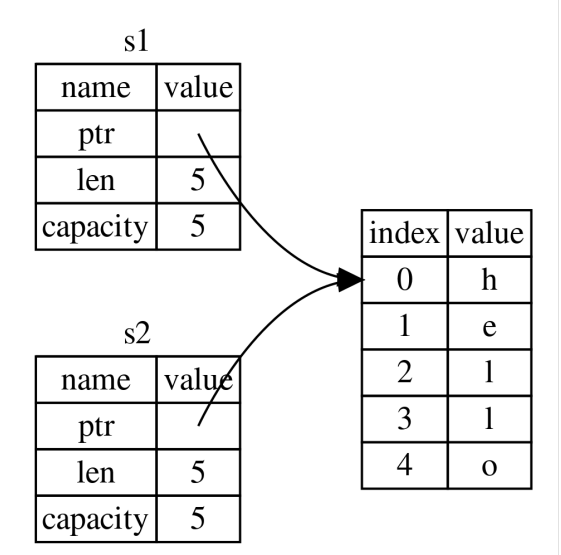

Figure 2.6: Representation of the memory layout of a string in Rust after the copy

According to the ownership rules, when s1 and s2 go out of scope, one may think that the memory will be freed twice, causing a double-free error. However, in reality, after the copy, the compiler will not consider s1 to be valid anymore, so when s1 goes out of scope, the memory will be freed only once, as shown in fig. 2.7.

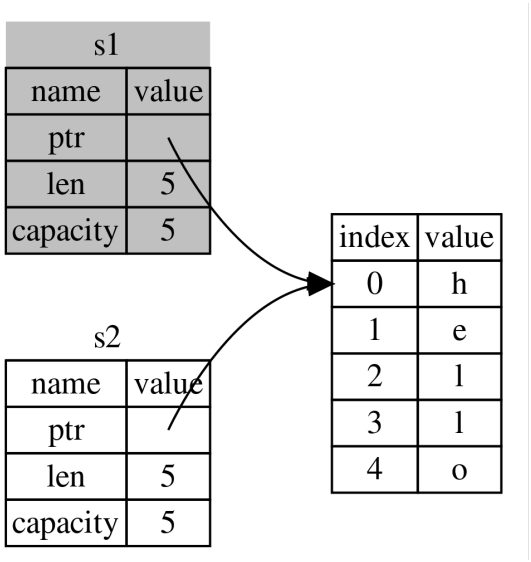

Figure 2.7: Representation of the memory layout of a string in Rust after the copy and the end of the scope of s1

In this case, it is said that the variable s1 has been *moved* into s2. This means that s1 is no longer valid and cannot be used anymore. This happens because, by default, Rust does not create deep copies of variables of types that don't have a known size at compile time. After all, creating a deep copy of such a variable would cause the allocation of a new memory block on the heap, an expensive operation both in terms of execution time and memory usage. If the developer needs to create deep copies of variables stored in the heap, they can explicitly use the *clone* method, which will create a new memory block on the heap and copy the data into it.

Ownership and Functions

Similarly to what happens during the variable assignment, passing a variable to a function will cause it, depending on its type, to be moved or copied, as shown in the listing 2.4.2.

```
fn main() {
    let s = String::from("hello");
    takes_ownership(s);
    let x = 5;
    makes_copy(x);

}

fn takes_ownership(some_string: String) {
    println!("{}", some_string);
}

fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
}
```

Returning values from functions will also cause ownership to be transferred, as shown in the listing 2.4.2.

```
fn main() {
    let s1 = gives_ownership();

    let s2 = String::from("hello")

    let s3 = takes_and_gives_back(s2);
}

fn gives_ownership() -> String {
    let some_string = String::from("yours");
    some_string
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String {
    a_string
}
```

References and Borrowing

Instead of taking ownership of a variable and then returning it to the caller, it is possible to pass a reference to the variable to the function, so that the function can

use the variable without taking ownership of it. For a function to take a reference to a variable, it is sufficient to prefix the type definition of the variable with an ampersand (&). By default, references are immutable, meaning that the function cannot modify the value of the variable. If the function needs to modify the value of the variable, it is possible to take a mutable reference to it by using the &mut keyword.

2.4.3 Functional Features of Rust

In this subsection, we will discuss some of the Functional Programming (FP)-adjacent features of Rust.

Product Types

In FP, product types are types that combine n values of possibly different types. In Rust, it is possible to define Product types by using the *struct* keyword. In particular, one can define a product type in two ways as shown in the listing 2.4.3.

```
1 // Simple tuple struct without named fields
2 struct Point(i32, i32);
3
4 // Struct with named fields
5 struct Person {
6     name: String,
7     age: u8,
8     address: String,
9 }
```

It is also possible to add functionality to the ADTs created by using the *impl* keyword, as shown in the listing 2.4.3.

```
1  struct Point(i32, i32);
2
3  impl Point {
4      pub fn new(x: i32, y: i32) -> Point {
5          Point(x, y)
6      }
7
8      pub fn x(&self) -> i32 {
9          self.0
10     }
11
12     pub fn y(&self) -> i32 {
```

```
        self.1
    }
}
```

Sum Types and Pattern Matching

In FP, a sum type represents a choice between some types. In Rust, we can define Sum types by using the *enum* keyword. It is also possible to perform pattern matching over a sum type, as it is shown in the listing 2.4.3.

```
// Here we define a simple sum type
enum List {
    Cons(i32, Box<List>),
    Nil,
}

// This function returns an Option that contains a reference to the first element
    of the list
fn head(list: &List) -> Option<&i32> {
    // In Rust, we can use pattern matching to destructure a sum type
    match list {
        List::Cons(value, _) => Some(value),
        List::Nil => None,
    }
}
```

Polymorphism

Rust supports polymorphism through traits. Rust's traits are similar to Haskell's typeclasses and they allow us to define a particular functionality that a particular type has. For example, we can implement Haskell's Show typeclass in Rust as shown in the listing 2.4.3.

```
// Here we define a trait for converting a type to a string.
trait Show {
    fn show(&self) -> String;
}

// Then we can implement it for some types.
impl Show for Point {
    fn show(&self) -> String {
        format!("({}, {})", self.x(), self.y())
    }
}
```

```
12
13 // Traits can also be automatically derived in some cases.
14 #[derive(Debug)]
15 struct Point3D(i32, i32, i32);
16
17 // Traits can also be used as bounds for generic types.
18 fn print_showable<T: Show>(s: T) {
19     println!("{}", s.show());
20 }
```

Lambdas and Closures

In FP, lambda functions or anonymous functions, are functions that are not bound to a name. Moreover, closures are lambda functions that can "capture" the environment in which they are defined. In Rust, both lambdas and closures are supported, though they are both called closures. Like in other languages, Rust closures can be assigned to variables and passed to functions. When defining a closure in Rust, it is important to reason about the ownership of the variables that are captured by it. The listing 2.4.3 shows some examples of closures in Rust.

```
1  // A simple closure that adds 5 to a number. Note that the type
2  // parameter of x can be omitted.
3  let add_five = |x: i32| x + 5;
4
5  // An example of a closure that captures its environment
6  fn add_prefix() -> impl Fn(&str) -> String {
7      let prefix = "Mr.";
8      // This closure needs to take ownership of prefix, so it isn't freed when the
       scope of the add_prefix function ends.
9      move |string: &str| format!{"{} {}", prefix, string}
10 }
```

Iterators

The Iterator pattern allows to traverse collection of elements in a particular manner, performing some task on each element in turn. The iterator is responsible for the traversal logic so that the developer does not need to reimplement it each time. In Rust, the Iterator pattern is implemented through the *Iterator* trait, which is implemented by the standard library's collections:

```
1      trait Iterator {
2          type Item;
```

```
3         fn next(&mut self) -> Option<Self::Item>;
4         //Other default methods omitted
5     }
```

The developer can also implement the Iterator trait for his custom types, enabling many functionalities that are divided into the following categories:

- **Consumer Adaptors**: these are methods that take ownership of the iterator because they traverse it using its next method, thus consuming it. Examples of consumer adaptors are reducing methods like *sum*;

- **Iterator Adaptors**: these are methods that don't take ownership of the iterator, since they take an iterator and return another, modified one. An example of an iterator adaptor is the *map* method, which applies a function to each element of the iterator;

Error Handling

Rust provides mechanisms for error handling that are equivalent of the ones we can find in most of the modern functional programming languages: Options and Results. The Option type models a value that can be absent and is implemented through an enum that can have two variants: Some, which contains a value, and None, which represents the absence of the value. This type offers many functions to manipulate its hypotetical value, such as *map*, *for_each* and *unwrap*, which attempts to get the value of the enum if present, panicking (the Rust equivalent of throwing an exception) if the value is not present. Another common mechanism for error handling in rust is the Result type, which represents a computation that may fail. This type is similar to the "Either" type of some functional languages like Haskell or Scala, and it is also implemented with an enum that can have two variants: Ok, which contains a value, and Err, which contains an error value. This type also offers many functions to operate with it, such as *unwrap*, *unwrap_or*, *map* and *map_err*. Since both of these types are enums, it is possible to deconstruct them via pattern matching.

2.4.4 Metaprogramming in Rust

In computer science, *metaprogramming* is the technique by which a programmer can write code that generates or manipulates other code. In Rust, this technique is enabled by its powerful *macro* system. There are two main families of macros in Rust: declarative macros and procedural macros.

Declarative Macros

Declarative macros are the most common type of macros in Rust. They are invoked similarly to functions, however, they can have a variable number of arguments and can be called with different types of parenthesis. At their core, declarative macros are similar to match expressions, but instead of matching against a value, they match against the Rust code that is passed to the macro, which can also include but it's not limited to expressions. The listing 2.4.4 shows an example of a declarative macro that implements the *vec!* macro.

```
#[macro_export] // export the macro to make it visible to other modules

// macro_rules! is a macro that defines a new macro with the name and body that follows the invocation
macro_rules! vec {
    // The macro's body is similar to a match expression and can contain several (...) => { ... } cases
    // If the code passed to the macro matches the pattern, the code in the block is expanded
    // In this case, this pattern matches a list of expressions separated by commas.
    // The $() syntax is used to define a variable. In this case, $x is of type expr, which means it can match any Rust expression.
    // There are several types of variables that can be used in a macro, such as expr, ident, block, etc.
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            // This special syntax allows to perform a repetition over the elements of the input
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

This thesis will not introduce the reader to procedural macros, as they constitute a deeply technical topic that is not relevant to the scope of this work.

2.4.5 Why Rust

As shown in the previous sections, Rust is a general-purpose programming language designed with a focus on performance and safety, while providing many of the high-level abstractions that are typical of modern programming languages. In particular, it has comparable performance to C and C++, while also granting a higher degree of robustness coming from the borrow checker, which ensures memory safety and a strong type system that enforces type safety. These features make Rust a good choice for developing software that could be run both on thin devices and in more performant systems, and thanks to its cross-compilation features, it simplifies the task of maintaining a codebase that can be run on different platforms. Moreover, Rust's powerful macro system allows for the possibility of writing a high-level DSL, making it a good candidate for implementing an AC framework that can standardize the development of aggregate programs.

2.5 Towards a Rust-based AC Implementation: the RustFields Project

The RustFields Project[CMPV23] was the first attempt to bring AC to thin devices by exploiting a modern programming language like Rust. The project aimed to reach its goals by developing two lines of research:

- A pure, Rust-based implementation of AC in the same vein as the ScaFi framework;
- A mixed approach that aims to mix ScaFi's highly expressive API with Rust's performance and efficiency;

2.5.1 RustFields Architecture

As stated in the official documentation of the project:
“ Since the project was meant as an exploration of different options for bringing aggregate programming into native contexts, we decided to explore both solutions. The resulting architecture reflects this choice: in fact, we decided to develop a standalone aggregate programming framework in the Rust language, while also experimenting with different ways to integrate it with the Scala ScaFi's ecosystem. ”

The resulting architecture reflects this choice and this is made evident in the diagram in fig. 2.8.

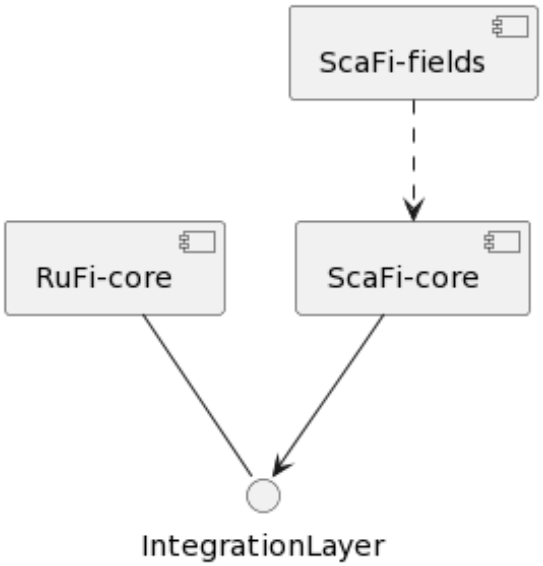

Figure 2.8: The RustFields Architecture

In the architecture diagram shown in fig. 2.8, the RuFi-core is responsible for implementing in Rust the core concepts of the new AC implementation, structured in a way that is somewhat similar to the ScaFi's core. Then, an *integration layer* was built on top of the Rust core to allow communication between Scala and Rust code. From then, the project development was divided into two main branches:

- The expansion of the RuFi core, which aimed to serve as a base for a fully-fledged AC implementation in Rust;

- The development of the integration layer, aimed to bring the best of both worlds by allowing the developer to use the expressive API of ScaFi with the performance and efficiency of Rust.

The design of the RuFi-core component is of particular interest since this thesis aims to expand it and further develop an AC implementation in Rust, and it will be discussed later in the Chapter 4.

Chapter 3

Analysis and Requirements

In this chapter, we will present the goal of this thesis and describe how it will be achieved. From here, an organized structure of requirements will be presented.

3.1 Thesis' Goal

This thesis is within the same scope and ambit as the RustFields project, i.e. enabling the execution of aggregate programs on thin devices that would not support the JVM. Specifically, this thesis aims to continue in this direction through the development of four objectives:

1. test, validate and possibly improve the design of the RuFi-core module;
2. develop a new module, RuFi-distributed, that will enable the distributed execution of aggregate programs within a network of devices;
3. develop and test an aggregate program that can be executed on a network of devices;
4. implement a demonstration of the whole system's functioning.

3.2 Requirements Breakdown Structure

From the thesis' goal and objectives, it is possible to devise a set of tasks that need to be accomplished. Each task can be further refined in sub-tasks that

constitute the requirements for the main task completion. Following this approach, a complete breakdown of the requirements emerges, as shown in this section.

1. **RuFi-core**: this requirement category is related to the objective 1: testing and improving the RuFi-core module.
   (a) expand the "by round" test suite;
   (b) implement functions to assert the equivalence of two aggregate programs;
   (c) expand the test suite with "by equivalence" tests;
   (d) investigate how to simplify the development of aggregate programs via Rust's macros;
2. **RuFi-distributed**: this requirement category is related to the objective 2: developing the RuFi-distributed module.
   (a) design an abstraction for networking operations;
   (b) implement a structure to be exchanged as a message between processes or machines;
   (c) implement the serialization logic for Export;
   (d) design an abstraction for a message queue;
   (e) design an abstraction for neighbor discovery;
   (f) implement the computational model shown in 2.1;
3. **Aggregate Program**: this requirement category is related to the objective 3: developing and testing an aggregate program.
   (a) **Implementation**: implement a simple gradient aggregate program using core constructs and builtins:
      i. implement the `mux` construct;
      ii. implement the `foldhood_plus` builtin;
      iii. implement the `gradient` program;
   (b) **Validation**: test and validate the aggregate program via unit testing;

4. **Demonstration**: this requirement category is related to the objective 4: implementing a demonstration of the whole system's functioning.
   (a) implement a simulation of the aggregate program execution locally on a single process;
   (b) implement a simulation of the aggregate program execution distributed across multiple processes hosted by the same machine;
   (c) implement a simulation of the aggregate program execution distributed across multiple machines;
   (d) **Reduce Memory Footprint**: this subset of requirements is an enabler for the main goal of bringing AC to thin devices:
      i. profile memory consumption;
      ii. reduce clone operations inside function implementations where possible;
      iii. change the core constructs and builtins to accept references to the Virtual Machine instead of owning it;

Chapter 4

Design

This chapter aims to give the reader a comprehensive view of this thesis' design. First, we will present the architectural design of the system, then we will shift the focus on the detailed design, where the system will also be described in terms of behavior and interaction.

4.1 Architectural Design

In this section, we will present and discuss RuFi's architectural design, shown in fig. 4.1.

In the figure, we can see the following components:

- **rf-core**: this component defines key abstractions, such as the fundamental aggregate operators, builtins and a virtual machine;
- **rf-distributed**: this component defines concepts related to the distributed execution of aggregate programs. In particular, it defines core abstractions related to networking and message passing, as well as an implementation of the computational model discussed in 2.1;
- **rf-distributed-impl**: this component exposes a standard implementation for the concepts defined in **rf-distributed**. The choice of separating the abstraction definitions and the implementations in two modules will be discussed in chapter 5;

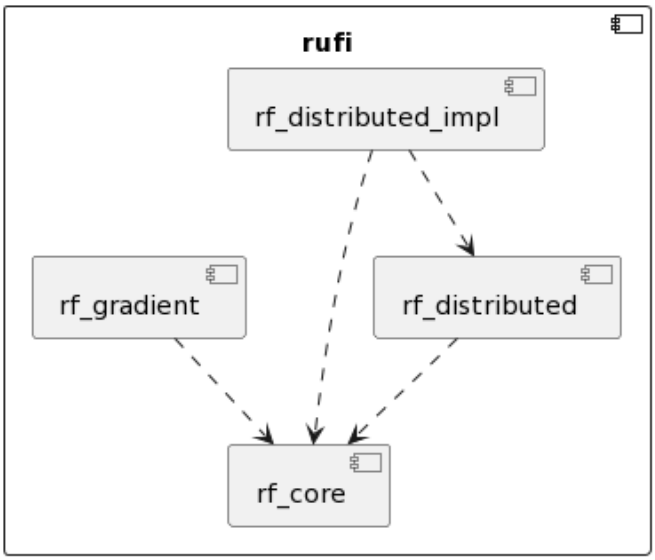

Figure 4.1: RuFi's architectural design

- **rf-gradient**: this component is a library exposing the gradient algorithm as an aggregate program.

4.1.1 RuFi Core

As mentioned in 4.1, the module RuFi Core contains all the fundamental abstractions needed to start writing and evaluating aggregate programs. Its structure is represented by the diagram in figure fig. 4.2.

- **round_vm**: This module defines a virtual machine for executing and evaluating aggregate programs and also other related concepts like the VM's status, the path, the execution context and the device exports, all of which will be discussed in more detail in section 4.2;

- **lang**: this module defines the fundamental aggregate operators, such as *rep*, *nbr*, *foldhood*, *branch* and some important builtin functions like *foldhood plus* and *mux*;

- **macros**: this module exports some declarative macros that aim to simplify the writing of aggregate programs;

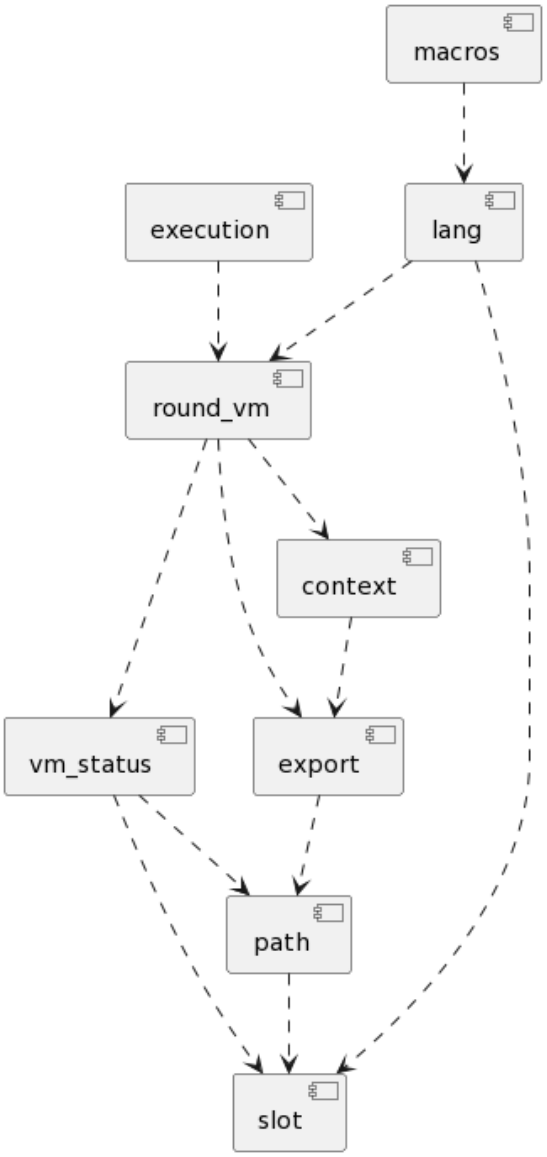

Figure 4.2: RuFi Core's architectural design

- **execution**: this module contains functions for evaluating aggregate programs alongside the virtual machine.

4.1.2 RuFi Distributed

This module, as described in 4.1, defines concepts related to the distributed execution of aggregate programs. Its structure is represented by the diagram in fig. 4.3.

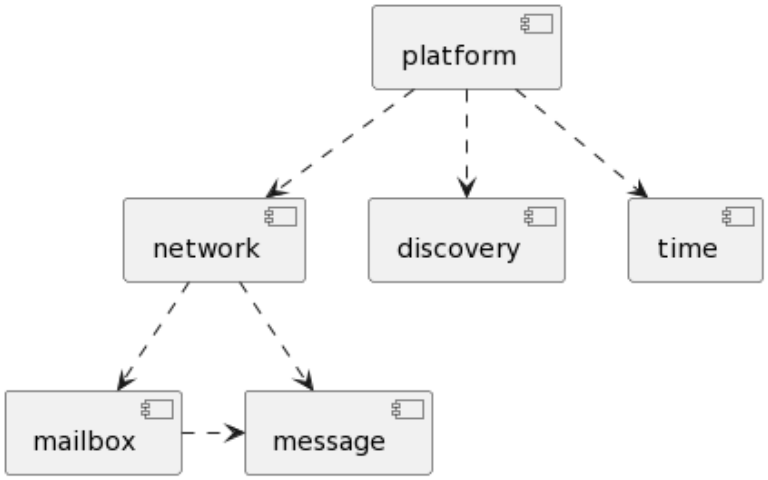

Figure 4.3: RuFi Distributed's architectural design

In the diagram, we see the following elements:

- **network**: this module defines abstractions for networking operations;
- **mailbox**: this module defines the logic of incoming message processing;
- **time**: this module defines some time-related operations that can be leveraged during the execution cycle;
- **discovery**: this module exposes traits for the discovery of possible neighbors and the logic for setting up neighboring sensors;
- **platform**: this module brings all the other functionalities together through the platform structure, which is responsible for managing the execution cycle of the aggregate programs;

4.2 Detailed Design

This subsection aims to explain in more detail some design choices that are important to understanding the whole system. After that, the system will be analyzed in terms of behavior and interaction.

4.2.1 RuFi Core

Round VM

The Round Virtual Machine is the operating heart of the RuFi framework. It is responsible for managing the state of the computation and generating the Abstract Syntax Tree (AST) of the aggregate program. Each core language construct leverages the VM functionalities in its implementation. In the class diagram in figure 4.4, we can see the elements that compose the VM.

The Slot is a representation of a core construct of the AC inside the AST. A Path is a sequence of Slots, denoting a chain of aggregate operations.
The Export represents the entire AST of the aggregate program, decorated with the value computed at each Path, and it is shared between neighbors during each computation round.
The VMStatus encapsulates the execution state: it has a representation of the execution stack and the current Path being evaluated.
The Context represents the execution context for the device and contains relevant information such as the device identifier and the device's sensors alongside the exports of all the neighbors. This means that during the computation process, each device has the AST of its neighbors. This aspect is crucial to implement neighboring operations.
The RoundVM is responsible for generating the AST and evaluate functions, and it does so with the *nest* 5.2.1 function, which gets executed whenever a core construct is called and causes the VM to push a new Slot into the current Path. A more detailed explanation of this function will be given in chapter 5.

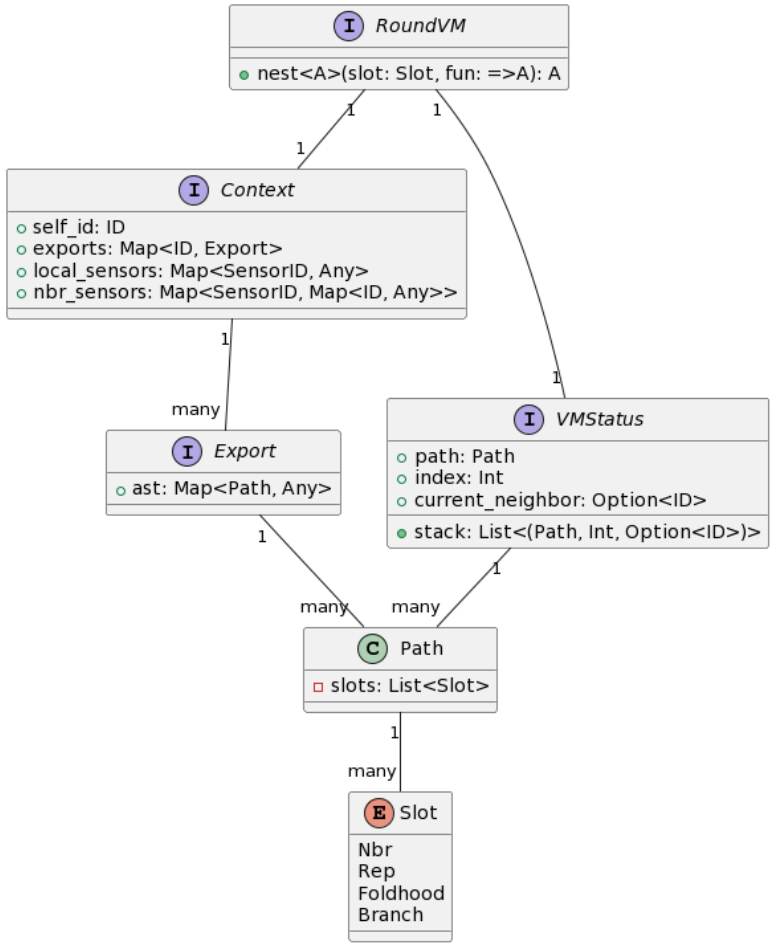

Figure 4.4: Round VM class diagram

Language

The language includes all the core constructs and built-in functions. Since each core construct call should grow the AST by one Slot, each language construct and built-in function has a dependency on the RoundVM, and so do the expressions that can be passed to them. This is reflected in their signatures, as shown in listing 4.2.1.

```
1 mod lang {
2     pub fn nbr<A, F>(vm: &mut RoundVM, expr: F) -> A
3     where
4         // F is a generic type bound that represents a function that takes a
              mutable reference to a RoundVM and returns a value of type A
5         F: Fn(&mut RoundVM) -> A,
```

```
    {
        //call vm.nest with the Nbr slot, passing also expr to it and then return the result
    }

    pub fn rep<A, F, G>(vm: &mut RoundVM, init: F, fun: G) -> A
    where
        F: Fn(&mut RoundVM) -> A,
        // G is a generic type bound that represents a function that takes a mutable reference to a RoundVM and a value of type A and returns a value of type A
        G: Fn(&mut RoundVM, A) -> A,
    {
        //...
    }

    pub fn foldhood<A, F, G, H>(vm: &mut RoundVM, init: F, aggr: G, expr: H) -> A
    where
        F: Fn(&mut RoundVM) -> A,
        // G is a generic type bound that represents an aggregator function that takes two values of type A and returns a value of type A
        G: Fn(A, A) -> A,
        H: Fn(&mut RoundVM) -> A,
    {
        //...
    }

    pub fn branch<A, B, TH, EL>(vm: &mut RoundVM, cond: B, thn: TH, els: EL) -> A
    where
        // B is a generic type bound that represents a function that takes no arguments and returns a boolean
        B: Fn() -> bool,
        // TH and EL are generic type bounds that represent functions that take a mutable reference to a RoundVM and return a value of type A.
        // TH is the type of the function that will be called if the condition is true, and EL is the type of the function that will be called if the condition is false.
        // We need two different type bounds here because in Rust, closures have unique types, even if they have the same signature.
        TH: Fn(&mut RoundVM) -> A,
        EL: Fn(&mut RoundVM) -> A,
    {
        //...
    }
}
```

4.2.2 RuFi Distributed

The Distributed module of RuFi has the goal of enabling the distributed execution of aggregate programs. A key concept defined in this module to reach this goal is the Platform. This structure is responsible for managing the execution cycle of aggregate programs, as well as the communication between neighbors, and it does so by leveraging the other concepts defined in the module. A representation of the main concepts in the module is shown in fig. 4.5.

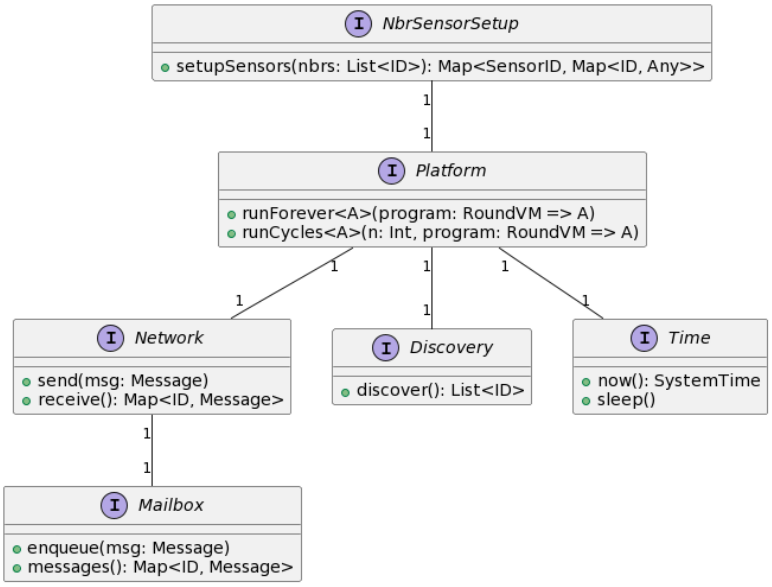

Figure 4.5: RuFi Distributed class diagram

A more detailed explanation of the module's components' functioning will be given in sections 4.2.3 and 4.2.4.

4.2.3 Behavior

In this section, we will showcase the behavior of a device when executing an aggregate program, which is represented by the activity diagram in fig. 4.6.

The activities represented in the diagram 4.6 closely resemble the computational model in 2.1. It is worth noting that the proposed behavior represents a synchronous execution cycle, where all the steps are performed in a sequential order. It is possible to extend this design to support the asynchronous execution of

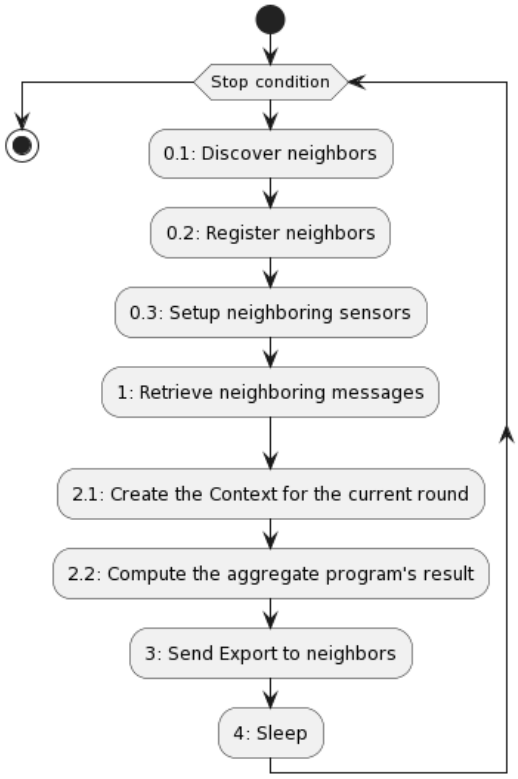

Figure 4.6: Device behavior

some of the steps like the discovery process, which could be done in parallel with the computation cycle. However, this solution would require the employment of an asynchronous runtime, which is a requirement that can be costly for certain device architectures. As such, the proposed design takes into account the synchronous case, leaving the asynchronous one to future developments.

4.2.4 Interaction

In this section, we will describe the interaction between devices during the distributed execution of an aggregate program, focusing in particular on the networking operations design. Although there is no strict requirement for the communication protocol between devices, the proposed design models the interaction between nodes via a publish-subscribe model akin to the one used in popular protocols like MQTT. As such, the following sequence diagrams will feature a Broker

participant, whose design falls out of this thesis' scope, who is responsible for forwarding and broadcasting messages between devices.

Network Creation

The behavior of the system when a device is added to the network is represented by the sequence diagram in fig. 4.7.

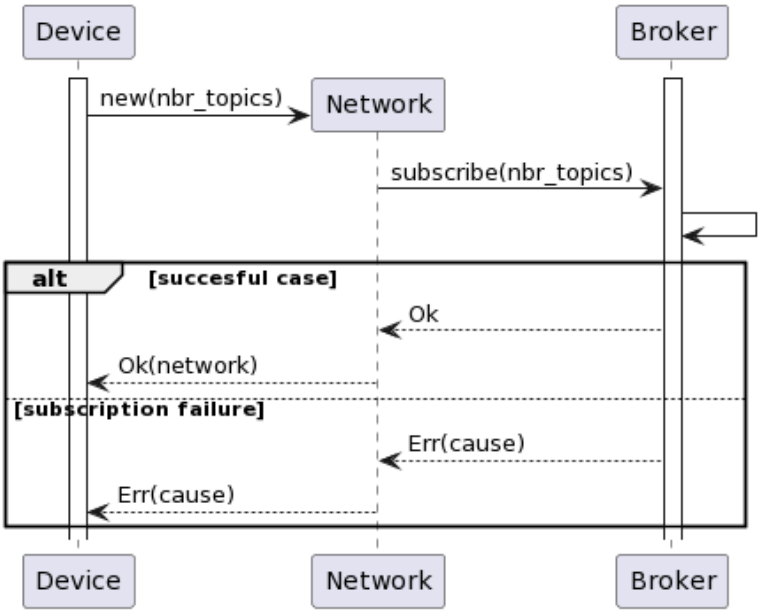

Figure 4.7: Network creation

1. The device first instantiates a new Network with an initial list of discovered neighbor topics;
2. the Network subscribes to the neighbors' topics via the Broker;
3. if the subscription is successful, a reference to the Network is returned to the device; if not, an `Err` is returned instead.

Message Passing

The behavior of the system when a device sends a message to its neighbors is represented by the sequence diagram in fig. 4.8.

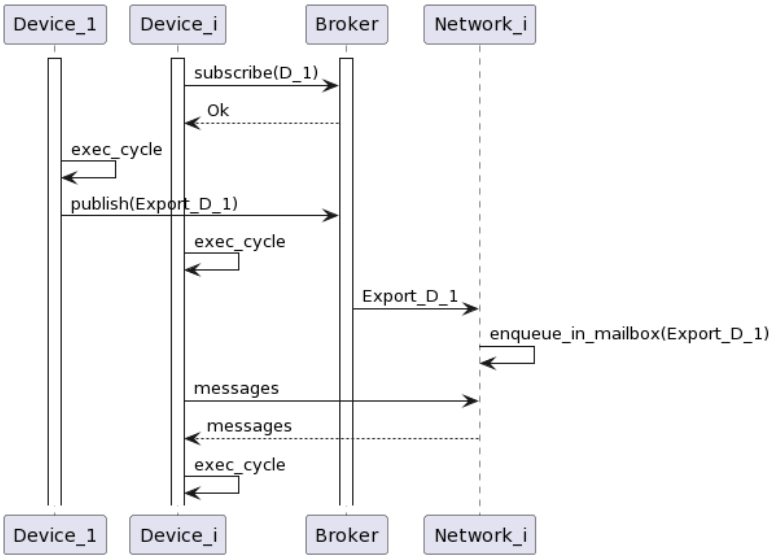

Figure 4.8: Message passing

1. The Device i subscribes to the topic of Device 1;
2. when Device 1 publishes a message, the Broker forwards it to Device i via its Network instance;
3. when Device i starts a new computation cycle, one of the first steps is to check the mailbox for incoming messages;
4. the Device i finds among other messages the one sent from Device 1, adding the Export it to its Context for the current computation cycle.

Chapter 5

Implementation

This chapter aims to provide an overview of important implementation choices, as well as highlight the technologies used to develop the RuFi framework.

5.1 Crate Structure

At the highest level, the framework consists of multiple *library crates*. In Rust, a crate is a standalone module that can be included as a dependency inside a project via the `cargo` package manager. There are three types of crates in Rust:

- *Binary crates* are crates that can be compiled into an executable. An example of a binary crate can be a program that runs on a device and utilizes the RuFi framework.
- *Library crates* are crates that can be used as a dependency in other projects.
- *Proc Macro crates* are library crates that expose procedural macros.

The development of RuFi followed a convention that is common in the Rust community for large projects, and that is the use of a *workspace*. A workspace is a directory that contains multiple Rust crates, and it is defined by a `Cargo.toml` file that lists the crates that are part of the workspace. Apart from this detail, each crate inside the workspace is a standalone Rust project with its specific dependency management and build configuration. In particular, there are five different library crates in the RuFi workspace:

- `rf-core` contains the RuFi core implementation.
- `rf-distributed` contains the RuFi distributed implementation.
- `rf-gradient` contains the implementation of the gradient aggregate program.
- `rf-distributed-impl` contains an implementation for the traits defined inside rf-distributed.
- `rufi` has a dependency on all the other crates and re-exports them under a common namespace. Thanks to Rust conditional compilation, it is possible to conditionally include or exclude entire modules from the dependency tree via the mechanism of `cargo features`, making this crate a convenient tool to access all the framework functionalities in a single, configurable dependency.

5.2 RuFi Core

5.2.1 RoundVM

The listing 5.2.1 shows the Round Virtual Machine, which is represented by a Rust struct that contains every dependency needed for executing its behavior.

```
struct RoundVM {
    context: Context,
    status: VMStatus,
    export_stack: Vec<Export>,
    isolated: bool,
}
```

One of the most important functions of the RoundVM is the *nest* function, which we can see in the listing 5.2.1.

```
impl RoundVM {
    // other RoundVM methods

    pub fn nest<A: Clone + 'static + FromStr, F>(
        &mut self,
        slot: Slot,
        write: bool,
```

```
        inc: bool,
        expr: F,
    ) -> A
    where
        F: Fn(&mut RoundVM) -> A,
    {
        self.status.push();
        self.status.nest(slot);
        let val = expr(self);
        let res = if write {
            let cloned_path = self.status.path().clone();
            self.export_data()
                .get::<A>(&cloned_path)
                .unwrap_or(
                    self.export_data()
                        .put_lazy_and_return(cloned_path, || val.clone()),
                )
                .clone()
        } else {
            val
        };
        if inc {
            self.status.pop();
            self.status.inc_index();
        } else {
            self.status.pop();
        }
        res
    }
}
```

The nest function takes as the parameters:

- `self`: a mutable reference to Self, the RoundVM;
- `slot`: the slot that should be written inside the Export;
- `expr`: an expression. In RuFi, expressions have the added parameter of a mutable reference to the RoundVM, since they may be language constructs, as will be later explained in this section;
- `write`: a boolean flag that determines if the value of the expression should be written in the Export.
- `inc`: a boolean flag that determines if the index of the AST's Slot in the VMStatus should be incremented.

The behavior of the function can be summarized as follows:

1. push the slot onto the current Path in VMStatus;
2. compute expr result;
3. if write is true, check if the export has a value for the current path, if not, write the result to the export;
4. if inc index is true, increment the index of the VMStatus (for ast navigation);
5. return the expr result.

5.2.2 Language

The first strategy that can come to mind for implementing the Language would be to have a Rust trait named "Language" that contains all the language constructs as methods, and then implement it for the RoundVM via an *impl Language for* block.

However, this approach has a major drawback: in fact, it clashes with Rust's borrowing rules. The listing 5.2.2 shows an erroneous attempt to implement the Language trait for the RoundVM.

```
trait Language {
    fn nbr<A, F> (&mut self, f: F) -> A
    where
        F: Fn() -> A + Copy,
        A: Clone + 'static + FromStr;

    // other constructs omitted for brevity
}

impl Language for RoundVM {
    fn nbr<A, F> (&mut self, f: F) -> A
    where
        F: Fn() -> A + Copy,
        A: Clone + 'static + FromStr,
    {
        self.nest(
            Nbr(self.index()),
            self.unless_folding_on_others(),
            true,
            || match self.neighbor() {
```

```
                Some(nbr) if nbr != self.self_id() => match self.neighbor_val::<A
                    >() {
                    Ok(val) => val,
                    _ => f(),
                },
                _ => f(),
            },
        )
    }
}
```

The code shown so far is valid Rust code, but when attempting to implement an aggregate program with this implementation, the flaws of this approach become evident, as shown in the listing 5.2.2.

```
fn main() {
    let ctx = Context::new(0, Default::default(), Default::default(), Default::
        default());
    let mut vm = RoundVM::new(ctx);
    let result: i32 = vm.nbr(|| 0); // This code compiles without errors and
        result is 0.

    let nested_result: i32 = vm.nbr(|| vm.nbr(|| 0)); // This code does not
        compile: ‘vm‘ doesn’t implement the Copy trait.
}
```

The problematic line of code is the last one when we try to pass an aggregate construct to the first nbr function. Since methods are not pure functions in Rust, we need to pass a closure that captures the outer vm variable, on which we can then call another aggregate construct. However, in order for the vm variable to be captured, it needs to implement the Copy trait, because the closure needs to have ownership of captured variables. Unfortunately, the RoundVM struct cannot implement the Copy trait, since it contains the Export struct, which contains a HashMap that has a non-Copy type as a value:

```
    pub struct Export {
        pub slots: HashMap<Path, Rc<Box<dyn Any>>>,
    }
```

Since the size of Any values cannot be known at compile-time, any structure that contains references to Any values cannot implement the Copy trait. It is however possible for them to implement the Clone trait, via the smart pointer “Rc”.

The code in the following listing will then compile:

```
let result = vm.nbr(|| vm.clone().nbr(|| 0));
```

However, this solution has a major problem: whenever a structure is cloned, the entire structure in memory is duplicated, meaning that every time the developer of an aggregate program passes an aggregate construct to another, the memory footprint of the program will increase.

The proposed implementation utilizes another approach: each aggregate construct and built-in function is a pure Rust function that takes a mutable reference to the RoundVM as a parameter. Each expression that can be passed to an aggregate construct would then need a mutable reference to the RoundVM as a parameter as well. In this way, we can combine aggregate constructs without the need to copy or clone the RoundVM, making the code more memory-efficient. The listing 5.2.2 shows the current implementation of the Language.

```
// The language is now a public module that exposes pure functions
pub mod lang {

    pub fn nbr<A, F>(vm: &mut RoundVM, expr: F) -> A
    where
        A : Clone + 'static + FromStr,
        F: Fn(&mut RoundVM) -> A,
    {
        vm.nest(
            Nbr(vm.index()),
            vm.unless_folding_on_others(),
            true,
            |vm| match vm.neighbor() {
                Some(nbr) if nbr != vm.self_id() => match vm.neighbor_val::<A>() {
                    Ok(val) => val,
                    _ => expr(vm),
                },
                _ => expr(vm),
            },
        )
    }

    // other constructs omitted
}
```

5.3 RuFi Distributed

The `rf-distributed` and `rf-distributed-impl` crates are responsible for realizing the distributed execution of RuFi programs. The first crate contains the definition of all the traits that are needed for this purpose, such as Network, Mailbox and Time, as well as an implementation of the Platform since it is generic in those traits. The second crate contains an implementation of the traits defined in the first crate. The choice of having two separate crates is because the implementation of some of the traits can be platform-specific. For example, the rf-distributed-impl crate utilizes a popular MQTT library named Rumqtt [Byt] to implement the Network trait. However, this library isn't compatible to all architectures. Other very popular and widely used, libraries like the de-facto standard asynchronous runtime Tokio [Nys] aren't fully compatible with the entirety of devices architectures. As such, the rf-distributed-impl crate is separated from the rf-distributed crate so it can be replaced with a different implementation if needed.

5.3.1 Networking

One of the key abstractions that are present in RuFi Distributed is the `Network` trait. This trait is responsible for providing the logic by which devices can send and receive messages through the network. The listing 5.3.1 shows the definition of the Network trait.

```
pub trait Network {
    fn send(&mut self, msg: Message) -> Result<(), Box<dyn Error>>;
    fn receive(&mut self) -> Result<HashMap<i32, Message>, Box<dyn Error>>;
}

#[derive(Debug, Serialize, Deserialize)]
struct Message {
    source: i32,
    msg: Export,
    timestamp: SystemTime,
}
```

The strategy chosen to implement this trait is to have a struct that contains a rumqttc-based MQTT Client and wraps it to adhere to the trait. Upon creation, the struct will setup the MQTT Client and spawn a new thread that will handle incoming messages, which will need to be "sent" back to the Network's thread.

In Rust, there are two ways for threads to communicate and share information, the first one of which is through the `message passing` paradigm. The main idea is represented by the slogan "Do not communicate by sharing memory; instead, share memory by communicating". To achieve this, Rust provides an implementation of the *channel* abstraction, which is a general programming concept by which data is sent from one thread to another. Channels have two halves:

- *Sender*: a cloneable type that can be used to send messages to the channel.
- *Receiver*: a cloneable type that receives messages from the channel.

The usage of the channel abstraction is shown in the listing 5.3.1.

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

So if we were to implement the communication between threads this way, the sender half of the channel would be passed to the thread that handles incoming messages (the "handler thread"), and the receiver half will remain inside the Network instance to be used by client code to retrieve the messages (the "client thread"). Whenever a packet is received from the handler thread, it will be sent to the receiver half of the channel.

However, this approach has some drawbacks:

1. the receiver half of the channel needs to be actively polled to retrieve the messages;
2. the channel is not bidirectional. This means that the handler thread cannot store the messages and send them all at once on demand.

These drawbacks combined mean that the client thread would need to busy wait for a single message from the network at every execution cycle, so the proposed implementation utilizes a second approach: `shared state concurrency`. This is a programming paradigm that allows multiple threads to access the same shared state, and in Rust, it is done via the *Mutex* abstraction. This way, the handler thread can directly and atomically push the incoming messages to the shared state upon arrival, and the client thread can request them all at once from the Network in a single call. Although programming concurrency through mutexes is generally avoided, especially in high-level languages, due to the complexity it can bring to the program, in this case, the mutex logic is confined to a relatively small portion of the code and is not exposed to the client, so it has been chosen as a valid option.

The implementation for the Network trait exposed in the rf-distributed-impl crate is the one in the listing 5.3.1.

```
struct SyncMQTTNetwork {
    client: rumqttc::Client,
    mb: Arc<Mutex<Vec<Bytes>>>,
}

impl SyncMQTTNetwork {
    pub fn new(
        options: MqttOptions,
        topics: Vec<i32>,
        mqtt_channel_cap: usize,
    ) -> Result<Self, Box<dyn Error>> {
        let (mut client, mut connection) = Client::new(options, mqtt_channel_cap);
        SyncMQTTNetwork::subscribe_to_topics(&mut client, topics)?;
        let mb: Arc<Mutex<Vec<Bytes>>> = Arc::new(Mutex::new(vec![]));

        let mb_clone = Arc::clone(&mb);
        thread::spawn(move || {
            loop {
                for (_i, notification) in connection.iter().enumerate() {
                    match notification {
                        Ok(Incoming(rumqttc::Packet::Publish(msg))) => {
                            if let Ok(mut mb) = mb_clone.lock() {
                                mb.push(msg.payload);
                            }
                        }
                        _ => {}
                    }
                }
            }
        });
```

```
31          Ok(Self { client, mb })
32      }
33
34      fn subscribe_to_topics(client: &mut Client, topics: Vec<i32>) -> NetworkResult
            <()> {
35          for nbr in topics.clone() {
36              if let Err(e) = client
37                  .subscribe(format!("nodes/{nbr}/subscriptions"), QoS::AtMostOnce)
38              {
39                  return Err(e.into());
40              }
41          }
42          Ok(())
43      }
44  }
45
46  impl Network for SyncMQTTNetwork {
47      fn send(&mut self, msg: Message) -> Result<(), Box<dyn<Error>> {
48          let source = msg.source;
49          let to_send = serde_json::to_vec(&msg)?;
50          self.client
51              .try_publish(
52                  format!("nodes/{source}/subscriptions"),
53                  QoS::AtMostOnce,
54                  false,
55                  to_send,
56              )
57              .map_err(|e| e.into())
58      }
59
60      fn receive(&mut self) -> Result<HashMap<i32,Message>, Box<dyn Error>> {
61          let mut mailbox = MemoryLessMailbox::new();
62
63          for u in self.mb.lock()?.iter() {
64              if let Ok(mex) = serde_json::from_slice::<Message>(u) {
65                  mailbox.enqueue(mex)
66              }
67          }
68
69          Ok(mailbox.messages())
70      }
71  }
```

5.4 RuFi Gradient

The RuFi Gradient crate contains an important example of what an aggregate program in RuFi can be and what it looks like. Since one of the core premises of FC is to provide key and reusable building blocks for aggregate computations, an aggregate program is nothing more than a function that combines these building blocks to achieve the desired behavior. The aggregate program could then be used as a building block in a larger aggregate program, and so on.

The listing 5.4 shows the implementation of the RuFi Gradient aggregate program:

```
pub fn gradient(vm: &mut RoundVM) -> f64 {
    fn is_source(vm: &mut RoundVM) -> bool {
        vm.local_sense::<bool>(&sensor("source")).unwrap().clone()
    }

    rep(
        vm,
        |_| 0.0,
        |vm1, d| {
            mux(
                vm1,
                is_source,
                |_vm| 0.0,
                |vm2| {
                    foldhood_plus(
                        vm2,
                        |_vm| f64::INFINITY,
                        |a, b| a.min(b),
                        |vm3| nbr(vm3, |_vm| d) + 1.0,
                    )
                },
            )
        },
    )
}
```

The core functions used in this program are:

- *rep*: the operator that denotes a dynamically changing field;
- *mux*: a branch variant that computes both the branches and returns the result of the branch that is selected by the condition. Since both branches of the operators are executed by the device, this construct does not restrict the

domain like the *branch* operator. Instead, it is used for simple conditional expressions;

- *foldhood plus*: a variant of the foldhood operator that excludes the device from its neighborhood.

The *is source* function calls the Virtual Machine and reads a sensor that establishes if the device is a source. The aggregate program itself is a rep operation that has an initial value for the distance d equal to 0.0. Then, inside the repetition, there is a *mux* call that returns a value of 0.0 if the device is a source or else an aggregation between neighboring values is performed via the *foldhood plus* builtin, resulting in the minimum distance $d + 1.0$ being kept as a result of the whole computation. In this way, the immediate neighbors of the source will compute a value of $0.0+1.0$, and the neighbors of the neighbors will compute $1.0+1.0$, and so on. For non-source devices that are not indirect neighbors of the source, the final result will be the starting value for the foldhood operator of $f64 :: INFINITY$.

5.5 Macro-based DSL

One of the objectives of the RuFi project is to provide a user-friendly and high-level DSL for writing aggregate programs. However, due to the Rust language's syntax, the current DSL isn't as user-friendly and easily readable as the ScaFi DSL: in fact, the developer of aggregate programs in RuFi needs to write a lot more boilerplate code, mainly due to the RoundVM dependency of the core constructs and the absence of high-level mechanisms like self-types and implicit parameters. The listing 5.4 highlights this issue, as there are many instances where we explicitly pass a RoundVM reference to an expression. To address this issue, we implemented a set of declarative macros that can be used instead of the core constructs and expands to a closure that takes a RoundVM reference as a parameter and passes it to a core construct like in the following listing:

```
1  #[macro_export]
2  macro_rules! rep {
3      ($init:expr, $fun:expr) => {{
4          |vm| rep(vm, $init, $fun)
5      }};
```

```
    }
```

The listing 5.5 shows how we can avoid some boilerplate code by using the macros instead of the core constructs when possible.

```
pub fn gradient() -> fn(&RoundVM) -> f64 {
    fn is_source(vm: &mut RoundVM) -> bool {
        vm.local_sense::<bool>(&sensor("source")).unwrap().clone()
    }

    rep!(|_| f64::INFINITY, |vm1, d| {
        mux(
            vm1,
            is_source,
            |_| 0.0,
            foldhood_plus!(|_| f64::INFINITY, |a, b| a.min(b), |vm2| {
                nbr(vm2, |_| d) + 1.0
            }),
        )
    })
}
```

Chapter 6

Validation

The validation for this thesis' work has been done on multiple axes, which will be described in this chapter.

6.1 Unit Testing

The first layer of testing is the "Unit Testing". In computer science, this term refers to the act of analyzing and scrutinizing the smallest units of software possible. This thesis adheres to Rust's unit testing practices: each public module that is part of the library has a corresponding and private testing module, annotated with the conditional compilation macro `#[cfg(test)]`, denoting this is a module that is only compiled when the test suite is run.
Inside this module, it is possible to write test functions by annotating them with the "#[test]" attribute. These functions can then be run with the "cargo test" command.
As such, each module in the RuFi library crates has a corresponding testing module containing unit test functions for them. For example, the listing 6.1 shows the unit tests for the vm_status module.

```
1 // inside vm_status.rs
2
3 #[cfg(test)]
4 mod test {
5     // this import directive lets us use the VMStatus struct and its methodss
6     use super::*;
```

```
7       use crate::path::Path;
8       use crate::slot::Slot::{Nbr, Rep};
9
10      #[test]
11      fn test_empty() {
12          let status = VMStatus::new();
13          assert_eq!(status.path, Path::new());
14          assert_eq!(status.index, 0);
15          assert_eq!(status.neighbour, None)
16      }
17
18      #[test]
19      fn test_fold_unfold() {
20          let mut status_1 = VMStatus::new();
21          let mut status_2 = VMStatus::new();
22          assert_eq!(status_1.neighbour, None);
23          assert!(!status_1.is_folding());
24          status_1.fold_into(Some(7));
25          status_2.fold_into(Some(8));
26          assert_eq!(status_1.neighbour, Some(7));
27          assert!(status_1.is_folding());
28          assert_eq!(status_2.neighbour, Some(8));
29          assert!(status_2.is_folding())
30      }
31
32      // other test functions...
33  }
```

6.2 Integration Testing

Unit testing is a crucial practice in the development of software artifacts, but testing each component in isolation is not sufficient to analyze every aspect of the software produced. Another important practice is the act of testing some or many components together, to ensure that their behavior when interacting is the one expected, which is called "Integration Testing". Again, this thesis adheres to Rust's integration testing practices: each library crate has a corresponding "tests" directory, where integration tests are written. These tests are run with the "cargo test" command, just like their unit counterpart.
Inside the tests directory, it is possible to create various source files for testing. Each file is isolated from one another and is external to the library since it is compiled as an individual crate: this means that the code inside these test files

utilizes the library via its public API just like any other client code.
The listing 6.2 shows an example of an integration test for the RuFi library that combines features coming from the RoundVM, Export and Language.

```
#[test]
fn export_should_compose() {
    fn ctx() -> Context {
        Context::new(
            0,
            HashMap::from([(sensor("sensor"), Rc::new(Box::new(5) as Box<dyn Any>)
                )]),
            Default::default(),
            Default::default(),
        )
    }

    let expr_1 = |_vm: &mut RoundVM| 1;
    let expr_2 = |vm: &mut RoundVM| rep(vm, |_vm1| 7, |_vm2, val| val + 1);
    let expr_3 = |vm: &mut RoundVM| {
        foldhood(
            vm,
            |_vm1| 0,
            |a, b| (a + b),
            |vm2| {
                nbr(vm2, |vm3| {
                    *vm3.local_sense::<i32>(&sensor("sensor")).unwrap()
                })
            },
        )
    };

    let mut vm = init_vm();
    let _ = round(&mut vm, combine(expr_1, expr_1.clone(), |a, b| a + b));
    assert_eq!(2, vm.export_data().root::<i32>().clone());
}
```

6.3 User Acceptance Testing

A third axes along which the thesis' work has been validated is the "User Acceptance Testing", which refers to the practice of testing the software in a real-world scenario. In particular, this involved the development of a demo project that exploits the RuFi framework to execute a gradient aggregate program within a network of 5 devices, each one represented by a process running on a machine. At an application level, the topology is linear, meaning each device i is a neighbor

of the devices $i + 1$ and $i - 1$, excluding 1 and 5 which are the extremes of the topology. At a deployment level, there are four processes each one simulating a device running on a Desktop PC, while the fifth runs on a Raspberry Pi 3.
Each process communicates with the others through the MQTT protocol via the public Mosquitto MQTT broker. A graphical representation of the network is shown in fig. 6.1.

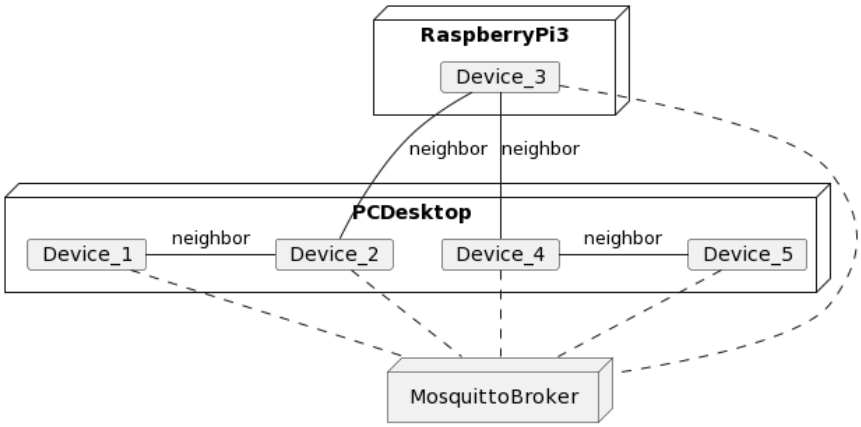

Figure 6.1: Network Topology

6.4 Memory Profiling

Another important aspect to consider while implementing a framework that aims to bring AC to thin devices is memory usage. Although a deep and comprehensive analysis of the memory usage of the RuFi framework is beyond the scope of this thesis, a simple memory profiling with a particular focus on memory allocation spikes has been done to ensure that the framework does not consume an excessive amount of memory. In particular, the profiling has been executed for three different execution cycles of the gradient aggregate program: 100, 300 and 500, at a frequency of 60mhz and for two devices: the device number 1 and the device number 3. These devices have been chosen because they have a different amount of neighbors, which means we can see how processing multiple neighboring messages can affect memory usage.

The results are shown in fig. 6.2. The first group of columns shows the memory usage for the device number 3, while the second group shows the memory usage for the device number 1.

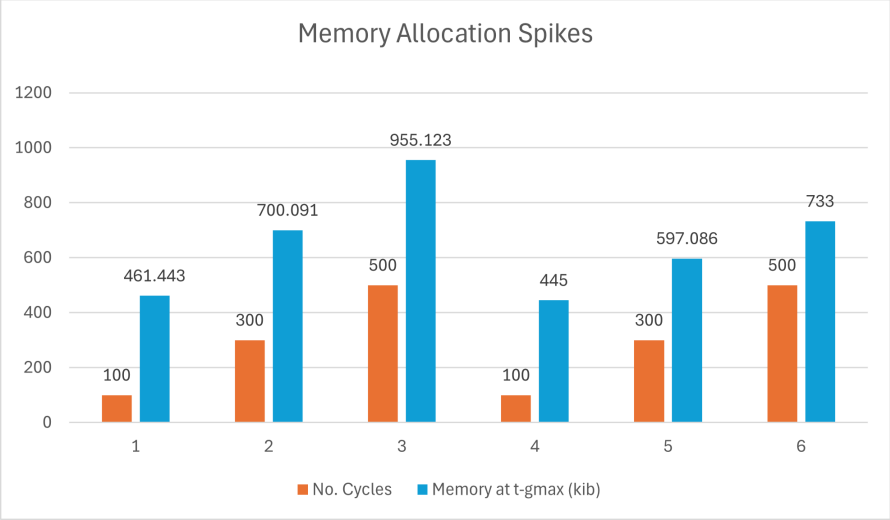

Figure 6.2: Memory profiling for the devices 3 and 1.

As we can see, the spike memory usage for the device with only one neighbor rises much more slowly than the one with two neighbors.

Chapter 7

Conclusion

The current landscape of Aggregate Programming frameworks highlights the opportunity to research and develop new solutions that can bring the paradigm to a wider range of devices while providing a high-level, functional API. It is with this objective in mind that we presented RuFi, a project that aims to leverage the Rust programming language's features of memory safety, performance and expressiveness to provide a minimal functional core for AC that can be used on multiple platforms, including thin devices, and build high-level APIs on top of it. In particular, this thesis tackled the problems of validating and improving the existing core of the RuFi framework, as well as the design and development of a new module that would enable the distributed execution of Rust-based aggregate programs. Starting by establishing a solid base regarding the context, paradigms and state-of-the-art for Aggregate Computing and a solid foundation of the Rust programming language concepts and idioms, we were able to identify a set of requirements and goals for this thesis project, which were then used as a guide during the design and development phases. Our analysis of the current state of the RustFields project has highlighted the need for validation and improvement of the current core library, as well as the need for a new module that would enable the distributed execution of Rust-based aggregate programs. With these considerations in mind, we started by thoroughly testing the core of the framework via unit testing and integration testing, as well as developing a set of macros that will help reduce the amount of boilerplate code one needs to write when defining an aggregate program. Then,

we proposed the design of the new RustFields framework, RuFi, highlighting first the architectural design of the project and its main components, and then delving into the detailed design of such components. These designs were then used as a guideline for the implementation phase, where we highlighted some important tactical choices that were made. We also started collecting experimental data on memory usage of the current RustFields framework, which will be useful for future research and improvements.

7.1 Current Limitations

As of now, although the main goal of the thesis of providing a distributed execution platform for the RustFields framework has been achieved, the higher-level objective of supporting RuFi on all thin devices is still not fully accomplished. Experiments on running the current RuFi framework in very resource-constrained devices like the Esp32 have shown that the current implementation is not yet suitable for such devices, as the memory usage is still too high, highlighting the need for further research and improvement on the memory footprint of such a framework.

7.2 Future Work

The previous analysis of the limitations of the current RuFi implementation has already suggested an important objective for future work. Nevertheless, there are also other interesting directions to consider, such as:

- support asynchronous network communication and execution: as mentioned before in section 5, the current solution is based on synchronous interfaces and execution cycle. This highlights the opportunity to implement an asynchronous version of the RuFi `Platform` that can manage and coordinate the processes of networked communication, neighbor discovery and program execution concurrently, allowing to take the most out of modern computer architectures;
- support reified fields: in the current RuFi implementation, the FC's concept of `Computational Field` is technically not represented. Instead, it is derived

by the `RoundVM` when executing a core construct. It would be interesting to explore the possibility of reifying the `Computational Field` as a first-class concept in the RuFi framework, allowing the constructs to explicitly manipulate fields. This solution could streamline the core of the framework as well as introduce the possibility of experimenting with other constructs and extensions of the FC model;

- improve further the RuFi API and DSL: the current improvements on the developer experience of the RuFi API and DSL are solely based on simple Rust declarative macros and although they eliminate some of the boilerplate code, they are still not as user-friendly as higher-level DSLs like ScaFi. A possible future line of research could be leveraging the more powerful Rust procedural macros, a subject that was not expanded in this thesis but could offer many opportunities to improve the RuFi DSL.

Bibliography

[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.

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

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

[BV14] Jacob Beal and Mirko Viroli. Building blocks for aggregate programming of self-organising applications. In *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 8–13, 2014.

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

[Byt] Bytebeamio. The mqtt ecosystem in rust. url = https://github.com/bytebeamio/rumqtt,.

[CMPV23] Angela Cortecchia, Leonardo Micelli, Paolo Penazzi, and Filippo Vissani. A port of the scafi framework in the rust language. url-https://rustfields.github.io,, 2023.

[CV16] Roberto Casadei and Mirko Viroli. Towards aggregate programming in scala. In *First Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '16, New York, NY, USA, 2016. Association for Computing Machinery.

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

[KN] Steve Klabnik and Carol Nichols. The rust programming language. url = https://doc.rust-lang.org/book/title-page.html,.

[LXZ15] Shancang Li, Li Da Xu, and Shanshan Zhao. The internet of things: a survey. *Information systems frontiers*, 17:243–259, 2015.

[Nys] Bob Nystrom. What color is your function? url = https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/,.

[REC15] Karen Rose, Scott Eldridge, and Lyman Chapin. The internet of things: An overview. *The internet society (ISOC)*, 80:1–50, 2015.

[Sat01] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal communications*, 8(4):10–17, 2001.

[VDB13] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In Carlos Canal and Massimo Villari, editors, *Advances in Service-Oriented and Cloud Computing*, pages 114–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.