ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Department of Computer Science and Engineering

Second Cycle Degree in Computer Science and Engineering

Scala-based Cross-platform and Polyglot Systems: General Architecture and Incarnation for Aggregate Computing

Master thesis in
ADVANCED SOFTWARE MODELLING AND DESIGN

Supervisor
Prof. Mirko Viroli

Candidate
Luca Tassinari

Co-supervisors
Prof. Gianluca Aguzzi
Dott. Nicolas Farabegoli

December 2025 Graduation Session
Academic Year 2024 / 2025

Abstract

Developing software libraries that operate seamlessly across heterogeneous platforms and programming languages remains a desirable, yet complex challenge in modern Software Engineering. In distributed systems, where heterogeneous devices must coordinate across language boundaries, this challenge intensifies as components need to dynamically maintain consistent semantics while adapting to platform and language-specific requirements. Beyond these challenges, there remains the desire to express reusable, unified core logic in an idiomatic, concise, and type-safe language—the quintessential goal of software engineering. However, language mismatches and platform-specific constraints often hinder interoperability, forcing developers to compromise on their ideal choice for expressing core logic to meet deployment requirements or leverage specific language ecosystems.

This work addresses this challenge within Scala, a language renowned for its expressiveness, type safety, and rich language abstractions. A general architecture and implementation strategy are presented that enable cross-platform and polyglot software library development in Scala. The proposed approach allows core logic to be expressed idiomatically in Scala while interoperating with multiple target languages and platforms, facilitating broader adoption and reuse of Scala-based libraries across diverse software ecosystems.

To validate the approach, it is applied to the Aggregate Computing framework, a paradigm for coordinating large-scale distributed systems through functional field-based abstractions.

Contents

List of Figures

Listings

1 Introduction

When building a new system, Software Architects face critical decisions regarding programming language and target platform selection to meet system requirements. This decision is crucial, most of the time irreversible and mainly driven by two orthogonal factors: the technical requirements and expected use cases of the system, and the surrounding ecosystem each language and platform offers. The first concerns the constraints and capabilities of the target deployment environment where the software needs to operate. Consider edge computing and IoT: many of these systems must run on devices with diverse hardware characteristics—different processing architectures, memory hierarchies, and power constraints. On the other side, the ecosystem encompasses the available libraries, frameworks, tools, and community support that can significantly influence development speed, maintainability, and overall success of the project, making certain languages and platforms more appealing for specific domains or industries. Moreover, the choice can be influenced by external factors, such as team expertise and organizational preferences.

This often conflicts with software engineers' goal of encoding business logic and use cases once, using a high-level, expressive, type-safe programming language that supports concise, maintainable, and idiomatic code—an approach made difficult by the constraints described above.

These aspects become even more critical when developing libraries and frameworks for distributed systems, where software must serve diverse users with varying needs and constraints across multiple platforms and programming languages, while ensuring components can interoperate seamlessly and maintain consistent semantics during execution.

Of course, maintaining multiple, separate codebases for each target platform and language is not a viable solution, as it would lead to increased maintenance burden, code duplication, and inconsistencies across versions. Instead, modern programming languages increasingly support diverse compilation targets. Scala exemplifies this trend as a powerful, multi-paradigm language originally built for the Java Virtual Machine (JVM) and later expanded to compile to JavaScript and bare metal environments. Moreover, Scala has established itself as a leading language for building distributed systems applications and frameworks by combining functional programming principles with object-oriented programming, offering a powerful type system and concise syntax that facilitate the development of robust, scalable, and maintainable applications. Not by chance, notable distributed computing frameworks and libraries, such as Apache Spark [Zah+16] and Akka[1], are built using Scala. Therefore, Scala represents an ideal candidate for developing distributed systems libraries that can target multiple platforms. However, despite its cross-compilation capabilities, Scala's polyglot potential remains untapped due to the fragmentation of projects providing multi-platform support and the lack of a unified framework for achieving multi-language interoperability. This thesis addresses this gap by exploring strategies and techniques to develop cross-platform and polyglot distributed systems libraries in Scala, enabling seamless integration and interoperability across systems written in different programming languages and running on heterogeneous distributed platforms—all while maintaining a unified codebase. The main contribution is the presentation of a general architecture for achieving this goal, along with its concrete reification in the context of Aggregate Computing [BPV15], an emergent research paradigm for engineering large-scale distributed collective systems. In this domain, device heterogeneity and multi-language support are paramount for fostering real-world adoption and leveraging the diverse ecosystem of tools and libraries available across different programming languages and platforms.

More specifically, this thesis contributions include:

- the design of a Scala general serialization binding for expressing format-agnostic encoding and decoding capabilities as context bound in distribution-aware API signatures. This enables flexible serialization strategies by deferring the choice of concrete serialization format to the end-user, and facilitates polyglot interoperability while preserving type-safety and capturing serialization concerns at the type level;
- the conceptualization of an abstract Scala architecture for building cross-platform and polyglot distributed system libraries that clearly separates the core library software product—encompassing core agnostic business logic and platform-specific technological concerns—from the user-facing API layer, which does not add new functionalities but rather widens accessibility of the API from multiple programming languages;
- the proposal of a Scala-based Polyglot Abstraction Layer that interposes between the software product and the library users, constituting the user in-

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

terface. By employing language-agnostic Portable Types and corresponding Scala isomorphisms that abstract away language-specific details and semantic differences, this layer enables the definition and exposure of a consistent API across multiple programming languages, handling language mismatches like differences in memory management or execution model;

- the implementation of an automated integration testing framework that verifies exposed polyglot APIs across multiple target languages, validating the correctness and consistency of the API surface presented to end users;
- the reification of the proposed polyglot architecture and the implementation of cross-platform distribution support to ScaFi3—a novel Scala 3 reimplementation of Aggregate Computing—which thereby enables the core field-calculus library to be used in applications written in different programming languages and deployed on heterogeneous devices.

Thesis structure. This thesis is structured in chapters. Chapter 2 introduces the background concepts of software portability and language interoperability, providing a general overview of existing approaches and techniques to achieve them, as well as an insight into Scala's cross-compilation capabilities. Chapter 3 delves into the motivations inspiring this work and its application to the Aggregate Computing paradigm, discussing the main theoretical foundations. Chapter 4 and Chapter 5 present the main contributions of this thesis: the proposed architecture for cross-platform and polyglot distributed libraries in Scala, the main library-agnostic implementation details and its specific reification in the context of Aggregate Computing. Chapter 6 describes the employed validation strategies and assess the effectiveness and efficiency of the proposed solution through a real-world demonstration. Finally, Chapter 7 concludes the thesis by summarizing the key findings and outlining the directions for future research in this area.

2 Background and Related Work

This chapter provides the reader the background behind the concepts of software portability, multi-platform software development and interoperability between programming languages, unfolding how these concepts play a crucial role in the comprehensive software lifecycle and why they are relevant in modern software engineering. Finally, the main cross-platform polyglot languages are reviewed, with a particular focus on Scala.

2.1 The role of portability in Software Engineering

Software *portability* refers to the ability of a software artifact to be used in a diverse range of platforms and environments. In this context, the term *platform* primarily denotes the combination of hardware architecture and operating system, though it is often extended to include the execution runtime [El-+17]. The term *environment*, instead, represents the broader collection of external elements with which the software interacts, including system interfaces and libraries. In essence, a software artifact is portable, i.e., it exhibits portability, when the cost required to design and implement it for porting to multiple platforms does not exceed the cost of re-development for each of them [Moo]. Portability is, therefore, about engineering software products to maximize components reuse while ensuring consistent behavior across target platforms.

Portability spans over multiple levels [Moo]:

- *source portability* occurs when the software is adapted to the underlying platform by changing the source code, which is then recompiled for the target platform. This is the most common form of portability;
- *binary portability* involves porting software in its compiled binary format. This is the most advantageous form of portability, though it is also the most difficult to achieve and is limited to specific cases;
- *intermediate-level portability* is a middle-ground between source and binary portability and entails porting an intermediate representation of the software,

sitting between the source and binary code.

Portability has always been a relevant concern in software engineering since the early days of computing, when the landscape of hardware architectures and operating systems was extremely fragmented and heterogeneous, and software was tightly coupled to the underlying platform, requiring full rewrites when moving to a different one [Ran20]. Over time, the situation has profoundly changed thanks to a series of innovations and efforts: from the standardization of operating systems interfaces, such as POSIX, to the widespread introduction of increasingly higher-level programming languages and the diffusion of modern paradigms, like the World Wide Web, that inherently fostered portability.

From a programming language perspective, one influential shift addressing portability has been the move from compilation-based to virtual machine-based approaches, such as the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) for .NET. These emerged to overcome the limitations of traditional compilation-based approaches, like C and C++, that require developers to create and maintain separate binaries for each target platform, often with platform-specific code and complex build configurations—a costly and time-consuming process. Virtual machine approaches addressed these challenges through an additional abstraction layer, decoupling the software from the underlying platform by compiling programs into an Intermediate Representation (IR), like the Java bytecode, that is then interpreted or just-in-time compiled by a platform-specific runtime. This approach has found widespread adoption, with the JVM and CLR becoming the backbone of entire ecosystems of modern languages and frameworks and enabling the famous "write once, run anywhere" paradigm—the promise that applications would be portable across any platform supporting the respective runtime.

However, the technical reality proved to be more nuanced: different virtual machine implementations can exhibit subtle differences in behavior, and, in resource-constrained environments, the overhead imposed by the virtual machine may be prohibitive.

Contemporary portability solutions continue this abstraction progression, with modern languages supporting multi-target compilation to various platforms, including JVM, JavaScript, WebAssembly and native binaries using IRs.

However, despite the advancements in portability techniques and tools, achieving portability remains a complex challenge as it requires ensuring consistent behavior across diverse platforms and environments that often differ in their capabilities and constraints. To illustrate, consider the differences between platforms in terms of concurrency models. Nowadays, most applications are designed to take advantage of multicore architectures and heavily rely on concurrency to speed up computations or to handle multiple tasks simultaneously. Nevertheless, not all platforms support this feature equally. For instance, the Node JS platform, widely used for server-side and web applications, is based on a single-threaded event loop model where blocking operations cannot be performed, requiring a completely different paradigm based on Continuation-Passing Style and asynchronous programming.

Such fundamental differences in platform capabilities are concern permeating all stages of the software lifecycle, from design to release, requiring careful analysis.

Design and implementation. When designing software, architects must consider the targeted platforms' constraints and capabilities to minimize the *abstraction gap* between the domain problem and platform abstractions. The greater the abstraction gap is, the more challenging it becomes to implement the required functionalities in a clean, well-structured and maintainable way. In this context, the maturity of a platform's ecosystem, in terms of libraries and frameworks availability, significantly reduces the abstraction gap developers must bridge, influencing the platform choice. For example, if a software product requires advanced data analysis and machine learning capabilities, targeting a platform with a rich ecosystem in this domain, such as Python, would be a wise choice.

This is particularly relevant when targeting multiple platforms, as the abstraction gap may be significantly higher with respect to a single platform scenario. As Doeraene notes in [Doe18], portability is not enough: even if a language is portable, unless the ecosystem publishes portable libraries, it remains difficult to develop applications spanning multiple runtimes because of development costs.

Testing. Having a multi-platform stack significantly impacts the testing strategy, as artifacts need to be tested on each of the targeted platforms to ensure they behave correctly and consistently across them. Furthermore, different architectures may behave differently under the same platform (e.g., ARM vs. x86 in native execution), requiring additional testing efforts to cover all the supported architectures.

Release. Releasing a multi-platform software product introduces significant complexity, as each platform has its own conventions and requirements for packaging, distributing, versioning and deployment. For example, JVM artifacts are distributed as JARs via Maven Central, JavaScript applications via npm bundles such as Webpack, and Python packages as wheels via PyPI.

Despite the challenges, portability remains a crucial aspect of modern software engineering, driven by the need to reach heterogeneous infrastructures and reduce the burden of maintaining or re-implementing software for different platforms.

Without it, developers face fragmentation, inconsistencies between the different versions and increased maintenance costs.

2.2 Language interoperability

Portability is only one side of the coin. Orthogonally to the platform dimension, *language interoperability* is equally important, as it refers to the ability to communicate with other languages on each targeted platform, guaranteeing uniformity in semantics across them [Doe18]. Having portability without interoperability is of limited practical use: while portability make it possible to use software artifacts in different platforms, the lack of interoperability with their APIs and libraries would make the software product isolated and unable to interact with the surrounding ecosystem. For example, developing an application targeting the JavaScript platform without being able to access its specific functionalities, such as manipulating the DOM, handling user events or performing file system operations leveraging Node.js API, would make portability largely theoretical rather than practical.

Interoperability, however, extends beyond accessing native APIs: it entails the responsibility for library developers to expose the library interfaces in the target language. This enables the widest possible developers' audience to use the library and, most importantly, to let them integrate it with the full power of the underlying ecosystem. This is particularly relevant because platform ecosystems have been evolved in silos over the years; they often lack interoperability with each other, and each of them is focused and optimized for specific domains and use cases. Language interoperability therefore allows, together with portability, developers to choose the best platform and, consequently, the best library API for their specific needs, minimizing the risk of incurring in incompatibility issues.

Crafting a "good" language-interoperable layer is a complex task whose main difficulty lies in the mismatch between the run-time semantics and abstractions of the involved languages. This includes, among others, dealing with different memory management strategies (e.g., garbage collection vs manual memory management), type systems (e.g., static vs dynamic typing, strong vs weak typing), and error handling mechanisms (e.g., exceptions vs error codes). Sometimes, these differences cannot be fully bridged: some features of the source language can be dropped or limited on a case-by-case basis if they cannot be mapped to the target language and are not essential for the target ecosystem. For example, most of the runtime reflection capabilities available in JVM-based languages may not be fully supported when interoperating with other languages. However, the key point about interoperability is that the implemented features must be complete with respect to the semantics of the host language, ensuring compatibility with any library in its ecosystem [Doe18].

2.3 Approaches to multi-platform and polyglotism

While various categorizations of cross-platform approaches exist in literature [El+17], two fundamental mechanisms can be distinguished: *cross-compilation* and *wrapper-based* approaches.

In the cross-compilation approach, the source code is written in a sort of "superset" language that is automatically compiled and/or transpiled to different platform-specific targets thanks to a dedicated toolchain, generating platform-specific artifacts. In this approach the source language itself is designed to provide both portability and interoperability with each target platform and is referred to as *cross-platform language* [Doe18]. Despite having a single super language that can be cross-compiled to different targets, that does not mean that all the code can be shared across all the platforms. Thus, when embracing this approach, the main goal is to design the software product to maximize the reusability of its components across the different targeted platforms and fill the abstraction gap with each environment via minimal platform-specific code, leveraging, whenever possible, cross-platform libraries. The primary advantage of this approach is that it enables sharing a substantial portion of the codebase across different platforms. Indeed, when core business logic is designed to be technologically agnostic—as best practices prescribe—it can typically be, for the most part, if not completely, shared across all the platforms. In this respect, striving to maximize code sharing, though it may require more effort initially, pays off in the long run as it is synonymous with better design.

The other approach is the wrapper-based one, which consists in developing the software product on a main platform and language, and exposing its functionalities to others via a dedicated interoperability layer, typically implemented as a wrapper library through *Foreign Function Interfaces* (FFI). These interfaces require dedicated runtime support and often rely on C as a common interoperability *lingua franca*, due to its widespread use in interfacing with low-level operating system APIs and system libraries. For example, many interpreted languages, such as Python, include mechanisms for executing extensions modules written in lower-level languages like C or C++, which are dynamically loaded into the high-level language's virtual machine at runtime [Gri+18].

Over the years, automated binding generation tools have been developed to simplify the creation of interoperability layers. A well known example is the Simplified Wrapper and Interface Generator (SWIG) [Bea03], which can generate language bindings from C and C++ header files to a variety of high-level languages, including Python, Java, JavaScript, Ruby and others. Another notable example is Py4J[1], a library that enables Python programs to interact with Java objects running in a JVM through a bridge gateway. This tool is widely used in projects like PySpark[2] to allow Python developers to leverage the Apache Spark engine, written in Scala and Java.

Unlike the cross-compilation approach, in the absence of automated tools, wrapper modules must be implemented manually—a process that can be complex, error-prone and that does not scale well if the number of target languages increases.

2.3.1 The case of Scala

Scala is a modern multi-paradigm programming language seamlessly integrating both object-oriented and functional programming, whose powerful static type system and advanced language abstraction features make it suitable for crafting complex and maintainable software systems in a concise, elegant and expressive way.

Originally designed to run on the JVM and interoperate with Java, Scala has evolved to support also native and JavaScript platforms making it a cross-platform language. Despite other languages, Scala's cross-platform capabilities are not tied to the language itself but, rather, demanded by dedicated projects, each with its own compiler plugins and toolchains that extend the Scala compiler and ecosystem.

Scala.js

Scala.js[3] is the project, introduced in 2013 at EPFL, meant to compile Scala code to JavaScript with the goal of enabling developers to write rich web applications entirely in Scala and having them compiled to ready to run JavaScript code [Doe18]. Over the years Scala.js has matured into a stable project that allows to target both web browsers and the Node.js backend environment, making it suitable for developing full-stack applications entirely in Scala. More recently, an experimental support for WebAssembly has also been introduced[4].

Compilation to JavaScript is achieved via a pipeline of stages, shown in Figure 2.1, performed after the Scala frontend compiler have parsed and type-checked the source code:

[1]`https://www.py4j.org`
[2]`https://spark.apache.org/docs/latest/api/python/index.html`
[3]`https://www.scala-js.org`
[4]`https://www.scala-js.org/doc/project/webassembly.html`

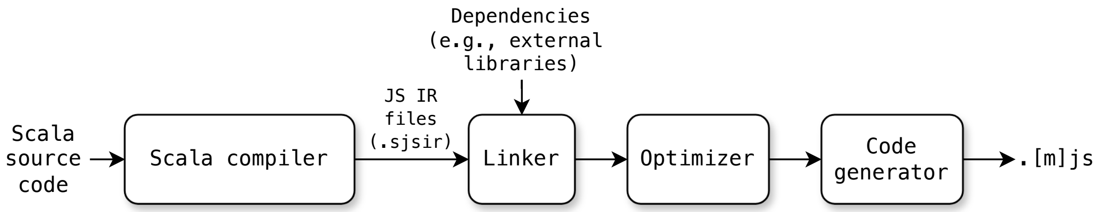


Figure 2.1: Scala.js simplified compiler pipeline.

- in the first stage the Scala.js compiler turns Scala Abstract Syntax Tree (AST) into an IR (`.sjsir`) specifically designed to provide both portability and interoperability;
- then, `.sjsir` files are linked together, type-checked and optimized. Optimizations include, among others, methods inlining at call sites and dead code elimination to make sure only exported API is kept, significantly reducing the final code size;
- finally, the optimized IR is compiled into a single `.[m]js` file containing the JavaScript code ready to be executed in the JavaScript environment.

The fact this compilation pipeline is attached to the standard Scala compiler allows supporting whole Scala language, including all advanced features and abstractions.

As for interoperability, Scala.js seamlessly integrates with existing JavaScript libraries and frameworks, thanks to a dedicated interoperability layer enabling both the consumption of JavaScript APIs and the exposure of Scala.js libraries to JavaScript code. Scala.js-specific code can therefore invoke JavaScript functions and interact with any library available in the JavaScript ecosystem, including the Node.js standard library, through typed facades mapping JavaScript APIs to Scala.js types and abstractions. These can be manually written or, more commonly, automatically generated using ScalablyTyped[5], a tool that converts TypeScript declaration files (`.d.ts`) into Scala.js facades via a Scala Build Tool (sbt) plugin. Conversely, software products APIs can be exposed to JavaScript code by annotating Scala.js classes and methods, instructing the compiler to generate the necessary bindings. Unfortunately, the project does not support automatic generation of TypeScript type definitions from Scala.js APIs, which would greatly facilitate the consumption of Scala.js libraries from TypeScript code (and there are no plans to support this in the future[6]).

[5]`https://scalablytyped.org/`

[6]`https://github.com/scala-js/scala-js/issues/3836#issuecomment-551273036`

To maximize code reuse across the JVM and JavaScript platforms, subsets of the Scala and Java standard libraries have been re-implemented to work natively in the JavaScript environment, providing almost equivalent semantics of their JVM counterparts: some differences are due to the inherent distinctions between the two platforms (e.g., the fact only `Doubles` are supported as numeric types in JavaScript) [Doe18]. A notable example of reusable standard library API is the Future-based concurrency model which, under the hood, impacts the JavaScript event-loop model while exposing the same API available on the JVM platform. Nevertheless, not all the JVM standard library can be ported; for instance, blocking I/O operations are incompatible with the JavaScript's asynchronous, non-blocking I/O model. When designing a cross-platform library, it is therefore essential to carefully evaluate which parts of the code can be shared and which need to be platform-specific, ensuring shared code relies only on cross-platform abstractions; for example, no blocking calls are performed.

Scala Native

Following the success of Scala.js, the Scala Native[7] project moved their first steps in 2017 as a research project at EPFL. Its goal is to compile Scala code directly to bare metal machine code without the need for the JVM, making it suitable for resource-constrained environments where the overhead imposed by the JVM in terms of memory footprint and startup time cannot be tolerated. Since the 0.5 release, published in 2024 with the introduction of native concurrency primitives, the project has reached a decent level of maturity and stability, making it viable for production use.

Scala Native is both an *ahead of time* (AOT) compiler and standalone runtime.

The AOT compiler translates Scala code into native machine code leveraging the LLVM toolchain [LA], a widely adopted modular compiler infrastructure for producing optimized machine code for multiple architectures. Compilation proceeds, similarly to Scala.js, through a pipeline of stages, shown in Figure 2.2, performed after the Scala frontend compiler, producing IR files from which the final machine code is generated:

- the first stage of the pipeline is performed by the `nscplugin` that, inspecting the AST, translates the Scala code into a strongly typed Native IR (NIR) consisting of a subset of LLVM IR instructions enriched with additional information to support Scala high level abstractions;

[7] `https://www.scala-native.org/`

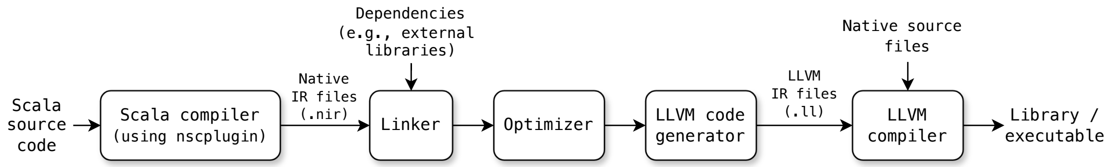


Figure 2.2: Scala Native simplified compiler pipeline.

- NIR files are then linked together with those generated from external libraries and optimized;
- finally, the optimized NIR is translated to low-level LLVM instructions that are then compiled by the LLVM backend to platform-specific machine code.

Other than the compiler itself, Scala Native includes a lightweight runtime layer that supplies essential services for the program execution. Most notable features provided by the runtime are:

- a configurable garbage collector for automatic memory management of heap-allocated Scala objects;
- a threading model, based on native OS threads, which maps Scala's concurrency abstractions to the underlying platform while striving to maintain maximum semantic compatibility with the JVM;
- most of the Scala and Java standard libraries re-implemented to work natively with the same semantics (Scala Native treats any differences in semantics between the two platforms as a bug).
- foreign function and a subset of C interoperability primitives to interface with existing C or C++ libraries and native code through type facades. These can be generated, similarly to Scala.js, via the sbt binding generator plugin[8], which processes C header files and produces Scala Native facades. Scala Native can also expose its libraries to C code via annotations, though currently only static object methods and properties (via accessor methods) can be exported and no automatic generation of C headers is provided.

Concerning the supported architectures and operating systems, Scala Native can target AMD64 (`x86-64`) and ARM64 (`aarch64`) architectures on Linux, macOS and Windows operating systems, also thanks to cross-compilation support—the ability to compile code for a different target architecture and/or OS than the one of the host machine where the compilation is performed. This covers the

[8]`https://sn-bindgen.indoorvivants.com`

vast majority of modern desktop and server environments, including some System-on-Chip (SoC) architectures, like Raspberry Pi devices. Experimental or partial support exists for FreeBSD, OpenBSD, and NetBSD, though sometimes limited to particular architectures (e.g., only AMD64 on OpenBSD). Moreover, with the 0.5 release, Scala Native introduced experimental support for 32-bit architectures, like ARMv7. Mobile platforms (such as Apple devices) and microcontroller architectures (like ESP32 architecture) are not supported: the first due to the lack of support for Objective-C or Swift interoperability, required for accessing Apple's frameworks, and the latter because of the limited resources offered by these devices. While the first limitation could be overcome in the future, the second will hardly be addressed because of the usage of limited or not fully mature LLVM support and the fact that compiled executable include a runtime layer that, alone, requires a few megabytes of memory that typically microcontrollers cannot accommodate.

Cross-platform ecosystem

The Scala ecosystem exhibits fragmentation due to separate projects targeting native and JavaScript platforms, each with independent release cycles and varying maturity levels. Comparing Scala.js and Scala Native, it is evident the former has reached a higher level of maturity and stability, driven by its broader audience and interest in web development. For this reason, the level of library support for Scala.js is higher with respect to Scala Native. Nevertheless, native platform has made significant progress in recent years, and since the introduction of native concurrency primitives, many libraries—such as those in the popular ZIO[9] and Typelevel[10] ecosystems—have been ported or are in the process of being ported. Of course, porting libraries across all platforms requires time and effort, especially when done retrospectively. Therefore, it is expected that in the coming years, the landscape of cross-platform Scala libraries will continue to improve and the number of libraries supporting both Scala.js and Scala Native will increase, driven by the growing adoption and success of both platforms.

Concerning language interoperability, the Scala ecosystem currently lacks production ready libraries exposing a unified and coherent cross-platform, polyglot API. However, despite the challenges and limitations introduced above that will be discussed in the next chapters, both Scala.js and Scala Native provide layers of interoperability that, if properly leveraged, provide a foundation for such solutions, on which this thesis aims to contribute.

[9] `https://zio.dev`
[10] `https://typelevel.org`

2.3.2 Related works

Besides Scala, several other languages and frameworks target multiple platforms and provide language interoperability features. Notable examples include Kotlin Multiplatform, Gleam, Haxe and Flutter/Dart. From a research perspective, a significant contribution to cross-platform polyglot development is the Basilisk architectural pattern, introduced by Bertolotti et al. in [Ber+24]. In the following, these works are briefly reviewed.

Kotlin Multiplatform

Kotlin Multiplatform[11] is Scala's primary competitor in both the JVM and cross-platform ecosystems, as it supports object-oriented and functional paradigms—albeit with less advanced abstractions than Scala—and targets comparable architectures: rooted in the JVM, it supports JavaScript, native platforms via LLVM, and, experimentally, WebAssembly.

Officially supported by JetBrains, Kotlin Multiplatform has gained significant traction in recent years, particularly in mobile development, after Google adopted it as the official language for Android. Its widespread use in the mobile development community, where cross-platform development is crucial, together with JetBrains' unified management, has fostered the growth of a robust, rich and continuously improving ecosystem that is expected to further expand in the coming years. Notable production-ready frameworks demonstrating the ecosystem's breadth are Compose Multiplatform[12], a declarative UI framework for building cross-platform user interfaces, and Ktor[13], a framework for building asynchronous server-side and client-side applications.

Thanks to its mobile-oriented focus, Kotlin Multiplatform also provides robust support for native Apple platforms (iOS, macOS), a capability currently missing in Scala Native. However, similarly to Scala, it cannot target microcontroller architectures because its large runtime requirements and limited LLVM support.

In terms of language interoperability, Kotlin Multiplatform offers strong integration with JavaScript, C, and Swift/Objective-C, other than Java. As in Scala, API exposure is achieved via annotations, instructing the compiler to generate the necessary bindings. Notable toolchain features in this context include the automatic generation of C headers and, experimentally, TypeScript declaration files, thereby minimizing manual integration effort.

[11] `https://www.jetbrains.com/help/kotlin-multiplatform-dev/`
[12] `https://www.jetbrains.com/lp/compose-multiplatform/`
[13] `https://ktor.io`

Gleam

Gleam[14] is an emergent general-purpose, statically-typed functional programming language. Guided by a minimalistic design philosophy, it emphasizes simplicity and ease of use, forgoing object-oriented constructs and advanced functional abstractions, such as type classes, in favor of first-class functions, algebraic data types and pattern matching.

The language provides two distinct compilation targets: the BEAM virtual machine and JavaScript. Targeting the BEAM VM allows Gleam to leverage server-side battle-tested Erlang ecosystem, renowned for its fault-tolerance, massive actor-based concurrency capabilities and distributed computing features. On the other hand, the JavaScript target enables Gleam to be used in web development, both on the client and server-side via Node.js.

Gleam targets these platforms employing a source-to-source transpilation approach, whereby it transpiles its source code into human-readable and pretty-printed Erlang or JavaScript code with TypeScript declaration files. This allows full interoperability with the BEAM and JavaScript ecosystems through annotations and type facades, and allows developers to inspect and debug the generated code directly. However, since the lack of hierarchical modules structure like Scala's or Kotlin's, this architectural choice introduces significant limitations when dealing with platform-specific incompatible abstractions, like I/O and concurrency. When targeting the Erlang's platform, concurrency is managed transparently by the BEAM runtime. In contrast, the JavaScript target requires explicit handling of concurrency via promises or callbacks to accommodate the event-loop model. Consequently, programming styles and constructs differ between targets: code employing Erlang-style concurrency I/O cannot be transpiled to JavaScript and vice versa. Therefore, libraries that make use of concurrent I/O need to decide whether to target one platform or the other, limiting code sharing.

These positions Gleam as a language suitable for full-stack development, allowing developers to write both backend and frontend code in the same language, rather than offering complete cross-platform capabilities.

Haxe

Haxe[15] is general-purpose cross-platform language, primarily designed to target a large variety of platforms and programming languages, including JavaScript, PHP, Python, Lua, C++, Java and C#, on a wide range of architectures and operat-

[14] `https://gleam.run`
[15] `https://haxe.org`

ing systems, including Android and Apple iOS. The language is strictly-typed, object-oriented, featuring an ECMAScript-oriented syntax, similar to JavaScript and Java.

Its cross-platform capabilities are achieved via a hybrid compilation strategy. The Haxe language can both compile into bytecode to be executed on JVM or its own Virtual Machines (HashLink and NekoVM) and it can also transpile into other high-level languages. It features a platform-agnostic standard library and an ecosystem of cross-platform libraries, supporting most of the necessary functionalities for general-purpose programming, including data structures, file I/O, networking and concurrency. Platform-specific code can be written using conditional compilation directives, similarly to C-style preprocessor macros.

Concerning interoperability, Haxe allows to interface with target-specific syntax and APIs creating, like all presented languages, typed facades. However, unlike Scala and Kotlin, these facades are not automatically generated for most of the platforms, requiring manual effort to create them. Moreover, for integrating with native libraries, no unified mechanism exists. Instead, each target platform needs to rely on its own build tool to link with external libraries, which may complicate the build process when targeting many platforms.

Although not as popular as other languages, Haxe has found a niche in video game development and, thanks to its broad platform coverage, is a viable option for projects prioritizing maximum platform coverage over advanced language abstractions or reliance on specific ecosystems.

Flutter/Dart

Dart[16] represents another notable approach to cross-platform development. Developed by Google, Dart is a statically-typed, object-oriented language best known for powering Flutter, a framework that enables cross-platform application development for mobile, desktop, and web from a single codebase. It targets native platforms (including Android and Apple devices) via Ahead-of-Time (AOT) compilation and web platforms through transpilation to JavaScript and WebAssembly.

Interoperability in Dart is achieved through different mechanisms depending on the target platform. On native, Dart both provides an asynchronous message-passing mechanism for communication with platform-specific code, and a Foreign Function Interface (FFI) enabling more direct and efficient invocation of native code from Dart. On web platforms, interoperability is provided through JavaScript interoperability mechanisms allowing Dart code to call JavaScript functions and vice versa.

[16]`https://dart.dev`

These characteristics make Flutter and Dart primarily well-suited for UI-centric and client-side applications, especially in mobile applications, where the framework has reached significant maturity and adoption.

Basilisk

Basilisk [Ber+24] is a high-level architectural pattern specifically designed to facilitate the development of cross-platform and polyglot software systems. It proposes a modular architecture in which core business logic is encapsulated, and platform-specific functionalities are exposed through well-defined interfaces, forming the abstraction layer. The system API is expressed through a Domain Specific Language (DSL) that is automatically transpiled to target languages via a dedicated transpilation infrastructure, which bridges the user-facing API and platform specific logic through Foreign Function Interfaces (FFIs). For example, in Scala output code transpilation, JNA (Java Native Access)[17] is used to interface with C libraries on native platforms, while in Python ctypes[18] is employed for the same purpose. This approach avoids the complexity of transpiling the entire codebase by limiting transpilation to the DSL layer only. While this effectively separates concerns between platform compatibility and language interoperability, it introduces significant complexity derived from the development and maintenance of the transpilation infrastructure and the DSL itself, representing a substantial upfront investment if not factored into an external framework on top of which to build the software product.

2.3.3 Comparative Analysis of Language Ecosystems

Table 2.1 presents a systematic comparison of the analyzed cross-platform languages and their main characteristics. The comparison evaluates each language across several dimensions: supported target platforms, employed approach for cross-platform support (whether cross-compilation, transpilation, or hybrid), language interoperability, and maturity levels in terms of ecosystem and toolchain. Maturity assessments are based on qualitative evaluations: ecosystem maturity considers the availability of libraries and frameworks along with community support, while toolchain maturity evaluates the stability and robustness of development tools. Primary target platforms indicate each language's original design focus, which typically exhibits the most mature tooling and library support. Experimental features represent capabilities under active development that are not

[17]`https://github.com/java-native-access/jna`
[18]`https://docs.python.org/3/library/ctypes.html`

yet complete or production-ready. Finally, indirect interoperability indicates that interaction with certain languages is achieved through a companion or intermediate language rather than direct support. For example, TypeScript interoperability is typically provided through JavaScript via TypeScript's declaration files, rather than direct TypeScript support.

Table 2.1: Cross-platform languages comparison

Language	**Paradigm(s)**	**Supported Environments**	**Primary Approach**	**Language interoperability**	**Ecosystem maturity**	**Toolchain maturity**
Scala	OOP + FP	**JVM**, JS, WASM†, Native, Android$^{\rightsquigarrow}$	Cross-compilation	Java, C and C++$^{\rightsquigarrow}$, JS and TS$^{\rightsquigarrow}$	JS: Mature, Native: Growing	JS: Mature, Native: Developing
Kotlin MP	OOP+FP	**JVM**, JS, WASM†, Native, Mobile (Android, Apple)	Cross-compilation	Java, C and C++$^{\rightsquigarrow}$, Obj-C/Swift, JS and TS$^{\rightsquigarrow}$	Mature	Mature
Gleam	FP	**BEAM**, JS	Transpilation	Erlang and Elixir, JS and TS$^{\rightsquigarrow}$	Emerging	Developing
Haxe	OOP	JS, Native, .NET, JVM, PHP, Python, Lua	Hybrid	C++, C#, Java, PHP, Python, Lua, JS and TS$^{\rightsquigarrow}$	Growing	Developing
Flutter / Dart	OOP	**Mobile (Android, Apple)**, Native, JS, WASM	Cross-compilation	C and C++$^{\rightsquigarrow}$, Obj-C/Swift, JS and TS$^{\rightsquigarrow}$	Established	Production-Grade

Bold: Primary target platform
†: Experimental feature
$^{\rightsquigarrow}$: Indirectly, via a "companion" language

Ecosystem maturity: *Emerging* (limited libraries and adoption), *Growing* (expanding libraries, active development, limited community support), *Mature* (rich ecosystem), *Established* (extensive packages, widely used in industry)
Toolchain maturity: *Experimental* (minimal tooling, unstable), *Developing* (basic tools, some instability), *Mature* (production-ready), *Production-Grade* (robust, widely adopted in production)

3 Context and Motivations

This chapter sets the context and motivations behind the thesis work. First, the fundamentals of Aggregate Computing are introduced. Next, interest in portability and language interoperability in this field is explained, along with the motivations behind this thesis project. Finally, an overview of the Scala 3 implementation of Aggregate Computing is provided, which will be extended in this work.

3.1 Aggregate computing: a bird's eye view

In the realm of complex large-scale systems composed of many interconnected devices, such as Internet of Things (IoT) ecosystems and swarm robotics, traditional approaches to programming individual devices struggle to cope with the scalability, composability, fault-tolerance and declarativeness required to manage such systems effectively.

Aggregate Computing [BPV15] is an emergent research paradigm aiming at programming such systems through the lens of *macro-programming* [Cas23]. Its core philosophy grounds on the idea of programming the collective behavior of the system as a whole rather than on the individual behavior of its components. Following this principle, developers can write a single program representing the desired global behavior, which emerges in a self-organized manner from the interactions of individual devices. The paradigm embraces functional programming principles to ensure composability and modularity of programs, allowing developers to build complex systems in a succinct and declarative manner, abstracting away low-level details about communication and coordination among devices.

Aggregate Computing model

The Aggregate Computing model is based on a collection of interconnected devices capable of exchanging information between themselves according to a neighboring relation. This establishes a dynamic network topology that can evolve over time as a result of mobility, failure and network delays. Each device's dynamics is modelled

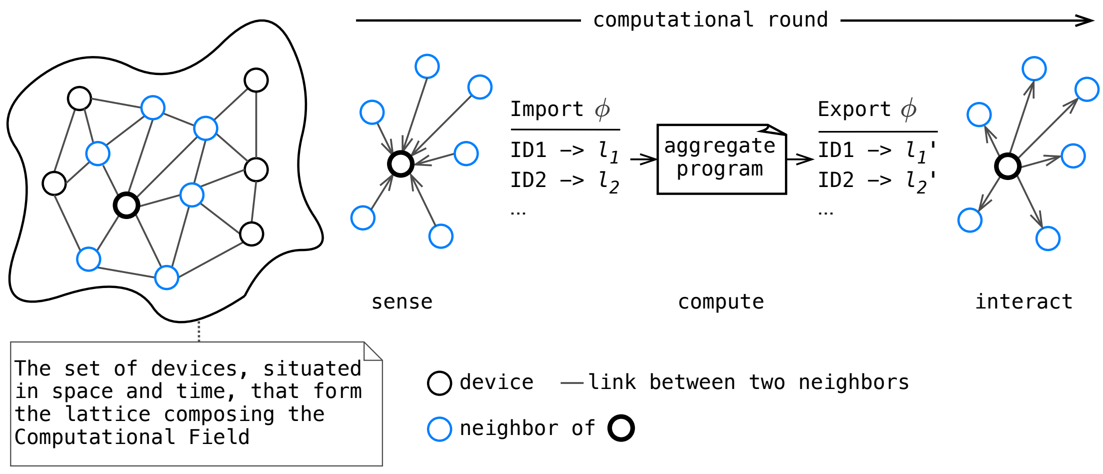


Figure 3.1: Visual representation of a computational *field* and phases of a round.

as a sequence of asynchronous, discrete computational rounds, each consisting of three phases, as illustrated in Figure 3.1:

i. *sense*: the device collects information from the most recent messages received from its neighbors and from its local sensors, through which it can observe its local environment;

ii. *compute*: execute the aggregate program using the updated local context to produce an output (the program's return value) and the messages to be sent to neighbors;

iii. *interact*: the program executes its actuations and sends the messages produced in the compute phase to neighboring devices.

Core Calculus and Operator Semantics

The main abstraction, formalized through the *Field Calculus* [Aud+19], is that of a *computational field*, more briefly referred to as a *field*. A field is a distributed data structure mapping each device to a local value in a specific point in space-time. Aggregate programs are expressed in terms of fields since they represent the first-class citizens of the paradigm, embodying the *"everything is a field"* philosophy: for example, the program controlling the movement of a swarm of robots can be encoded as a field of vectors, where each device has an associated vector indicating the direction and speed it is moving.

Computations, in this context, are performed by manipulating fields through *stateful evolution*, *neighborhood interaction* and *domain partitioning* constructs.

Those can be expressed by the `exchange` primitive and conditional expressions, as formalized in the *Exchange Calculus* (XC) [Aud+24], a tiny core calculus allowing to express the overall behavior of Aggregate Computing systems:

- *stateful evolution* of fields over time (rounds) and *neighborhood interaction* are achieved through the `exchange` primitive:

  $$\mathbf{exchange}\big(e_i,\ (\underline{n})\ \Rightarrow\ \mathbf{return}\ e_r\ \mathbf{send}\ e_s\big)$$

  that is evaluated, for each round, as follows:

  i. the expression e_i is computed to produce the local initial value l_i;
  ii. $\underline{n}$ represents the Neighboring Values (also called *NValues*) received from nearby devices for this exchange. Notationally, Neighboring Values are underlined to distinguish them from local values. Note that, since a device is always considered a neighbor of itself, its own last local value for this exchange from the previous round is also included in $\underline{n}$. For those neighbors for which no value has been received l_i is used as default. The expression e_r is evaluated to the value to be returned by the exchange, allowing to evolve state over rounds based on the neighborhood context;
  iii. the expression e_s is evaluated to produce the values to be sent to neighbors for this exchange in the next round.

  Often, e_r and e_s coincide; in that case the `retsend` shorthand is used:

  $$\mathbf{exchange}(e_i, (\underline{n}) \Rightarrow \mathbf{retsend}\ e)$$

- *domain partitioning* to manage different sub-collectivities within the overall system:

  $$\mathbf{if}\ (condition)\ \{e_{true}\}\ \mathbf{else}\ \{e_{false}\}$$

  that evaluates the `condition` expression on each device and, depending on its truth value, evaluates either the true branch or the false branch, effectively partitioning the computational field into two sub-fields.

Exchange is powerful enough to express also the *Field Calculus* constructs:

- `nbr`, used to access neighbor's values:

  $$\mathbf{nbr}(e : A) : \underline{A} \equiv \mathbf{exchange}\big(e,\ (\underline{n})\ \Rightarrow\ \mathbf{return}\ \underline{n},\ \mathbf{send}\ e\big)$$

- `rep`, used to make evolve the local state over rounds based on the result of the same expression in the previous round:

  $$\mathbf{rep}(e_i : A)\{(x) \Rightarrow e_n\} : A \equiv \mathbf{exchange}\big(e_i,\ (\underline{n}) \Rightarrow \mathbf{retsend}\ e_n[x := self(\underline{n})]\big)$$

- `share`, used to access neighbors' values while computing a new value based on the previous result:

$$\textbf{share}(e_i : A)\{(\underline{x}) \Rightarrow e_n\} : A \equiv \ self(\textbf{exchange}(e_i,\ (\underline{n}) \Rightarrow \textbf{retsend}\ e_n))$$

On top of these constructs, a set of higher-level building blocks and libraries can be built to facilitate the development of aggregate programs in a composable and declarative manner. These include self-healing gradient functions for distance estimation, information spreading mechanisms, state management and time-related constructs to handle stateful and temporal behaviors.

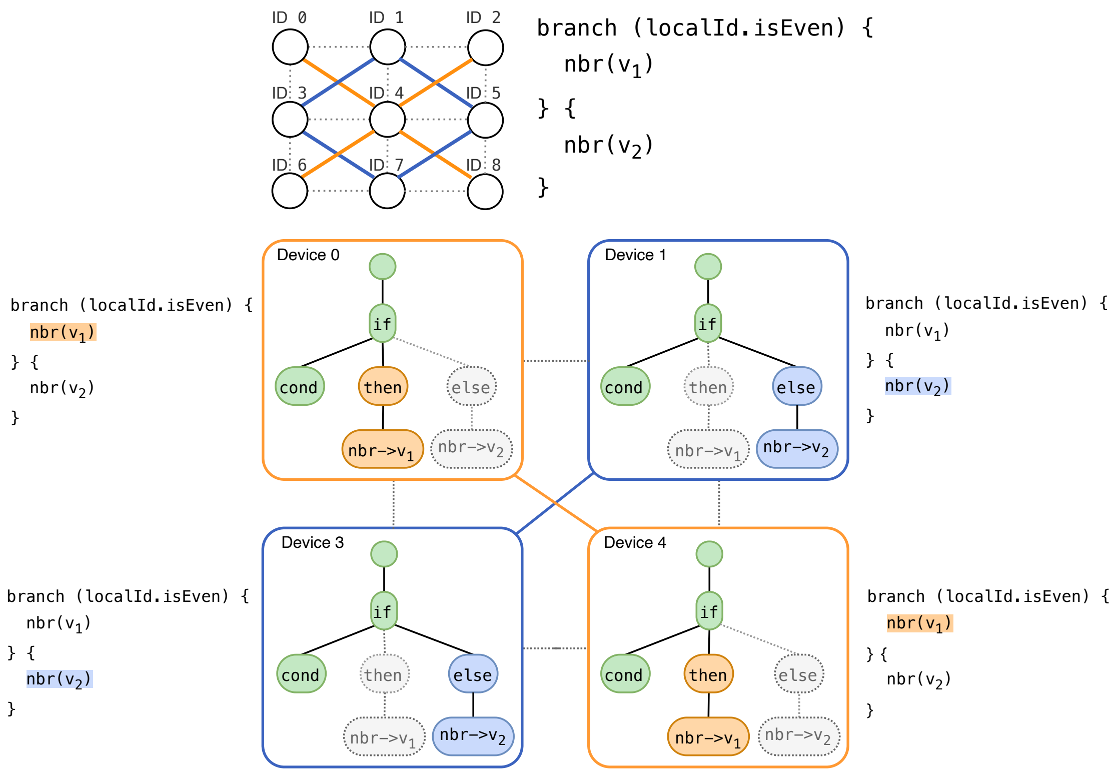
Alignment

Both `exchange` and domain partitioning constructs to work properly rely on *alignment*, the mechanism ensuring the values produced by an exchange are processed only by the corresponding exchanges of neighboring devices, namely those in the same position within the program structure. This prevents different exchanges from interfering with one another and causing inconsistent or erroneous behavior when a program contains multiple exchange expressions, possibly in different branches of a conditional.

As an illustrative example, the program presented in Figure 3.2 contains two `nbr` located in separate branches of a conditional statement. In any round, each device always evaluates the same branch of the conditional depending on the value of its identifier. When evaluating the Neighboring Values, however, only the values produced by devices executing the same `nbr` in the same branch should be considered. Otherwise, the program would mix values coming from different contexts, resulting in incorrect outcomes. Those devices are said to be *aligned* with each other.

As a result, when evaluating `exchange`-based expressions, the *field* of Neighboring Values is restricted to include only those coming from aligned devices. In Figure 3.2, when evaluating the `nbr` expression, each device can only observe Neighboring Values coming from devices with the same identifier parity.

Alignment is achieved by tracking the program's AST, namely the *Value Tree* produced at each evaluation round, and propagating it to neighboring devices so that they can identify aligned devices and values in subsequent rounds.



```
branch (localId.isEven) {
  nbr(v₁)
} {
  nbr(v₂)
}
```

Figure 3.2: Alignment example. The upper part shows a Moore grid of interconnected devices, where each device is connected to its eight neighboring cells and identified by a unique integer. The lower part provides an enlarged view of the four devices in the upper-left corner of the grid, showing their program structure represented as an AST. Grey dashed branches denote the conditional branches that a device does not evaluate. During the evaluation of the `nbr` expression, aligned devices—connected by solid lines—are those whose identifiers share the same parity: devices with even identifiers align only with others having even identifiers, and likewise for odd ones.

3.2 Portability and interoperability in Aggregate Computing

Aggregate Computing targets large-scale distributed systems made up of many heterogeneous devices, from wearables to embedded and mobile ones, each with distinct hardware and software capabilities. Ensuring portability is therefore a primary concern for deploying aggregate programs effectively.

Historically, the first implementation of Aggregate Computing was *Protelis* [PBV17], a JVM-based, Java-interoperable external domain-specific language (DSL) providing a higher-order aggregate programming language. Then, *ScaFi (Scala Fields)* [CV16] was introduced as a Scala 2 strongly-typed internal DSL leveraging Scala's powerful type system and functional programming features.

Over the years, several other implementations have been developed—or are currently being developed—in different languages, including FCPP [Aud20] in C++, Collektive [Tro23][Cor24] in Kotlin and Rufi [Mic24] in Rust. These implementations aim to support the widest range of platforms, enabling real-world deployments on edge devices, while taking advantage of each language's and ecosystem's strengths, such as C++ performance optimizations or Python machine and deep learning libraries. However, they were created from scratch, with neither code reuse nor compatibility in mind. Moreover, the lack of a common framework has led to fragmentation of the aggregate ecosystem, leading to a situation where each implementation has its own set of libraries, ad hoc extensions and maturity level.

To address these issues, there has been growing interest towards a framework capable of targeting multiple platforms while offering interoperability with other languages. Such a framework requires a foundational language that can effectively and idiomatically express the abstractions and computational model of Aggregate Computing. Critically, cross-platform support must not come at the expense of expressiveness. Scala emerges as an ideal "super"-language for this purpose, given its strong type system allowing to build powerful DSLs and its functional programming features enabling higher-order abstractions. Moreover, Scala's growing support for multiplatform development calls for an investigation into whether it can be effectively used to build a portable Aggregate Computing framework.

More generally, abstracting away from AC, the goal of this work is to investigate the feasibility of building a cross-platform and polyglot architecture for building Scala distributed libraries and frameworks. Ultimately, such architecture could be reified into the AC domain. Specifically, this exploration is meant to investigate:

- architectural strategies to design a portable and interoperable layer, while maintaining core abstractions and semantics and enabling complete code reuse;

- interoperability and distribution strategies enabling seamless data exchange and execution across heterogeneous devices and language runtimes;
- performance implications, idiomaticity of the resulting APIs and overall effort required to extend and maintain a cross-platform and polyglot API.

3.3 Scafi3 overview: a Scala 3 library for aggregate programming

ScaFi3[1] [Del24] is a modern Scala 3-grounded implementation of Aggregate Computing. Born as a complete re-engineering of the original ScaFi library to leverage the new features and capabilities of Scala 3, it delivers full reified support for fields alongside full implementation of XC constructs.

The project is organized into two modules: the `core` module, providing the core aggregate language constructs and semantics, and the `alchemist-incarnation` module, providing integration with Alchemist [PMV13], a simulator for pervasive, nature- and chemical-inspired, and aggregate computing systems that enable the simulation of large-scale systems. The `core` module is implemented as a pure Scala 3 module, meaning it has no dependencies on any platform-specific capabilities, making it cross-platform by design. The `alchemist-incarnation` module, instead, depends on the Alchemist simulator and the JVM platform, thus being limited to JVM-based platforms.

ScaFi3 core DSL and libraries entry-points are implemented leveraging Scala 3 contextual abstractions [Ode+17]. More specifically, they are implemented as pure functions in singleton objects, taking as implicit parameters the *language syntaxes*, which provides the necessary capabilities to express the aggregate constructs. These are automatically injected by the ScaFi engine, which is responsible for executing computational rounds: during each round, in the compute phase, the aggregate context is created and implicitly provided when evaluating the aggregate program.

In practice, to write an aggregate program in ScaFi3, the developer defines a type alias representing the required language syntaxes, expressed as intersection type composition, and then defines the aggregate program as a contextual function over that language. For example, in Figure 3.3 a simple aggregate program computing a distance gradient from a source node is shown. The program requires

[1] `https://github.com/scafi/scafi3`

access to field calculus syntax (the `share` construct) and the distance sensor, allowing to sense the distance from neighboring devices. To achieve this, the `Lang` type alias is defined as the intersection of the `AggregateFoundation` root syntax, where the `DeviceId` type is specialized, the `FieldCalculusSyntax`, providing field calculus constructs, and `DistanceSensor`, providing the distance sensor capability.

```scala
object Gradient:
  private type Language =
    AggregateFoundation { type DeviceId = Int }
      & FieldCalculusSyntax
      & DistanceSensor[Double]

  def gradient(using Language): Double =
    share(Double.MaxValue): nvalues =>
      val distances = senseDistance[Double]
      val minDistance = (nvalues, distances)
        .mapN(_ + _)
        .withoutSelf.min
      if isSource then 0.0 else minDistance
```

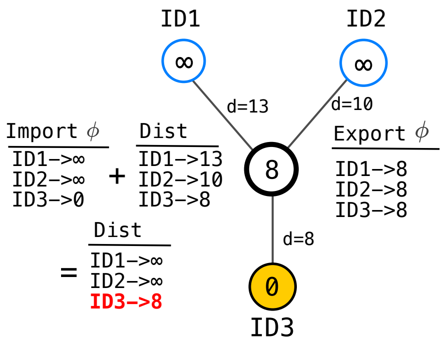


Figure 3.3: On the left, a ScaFi3 aggregate program computing a distance gradient from a source node. At each round, every device shares with its neighbors the minimum distance among them, computed as the pointwise sum of the values received from neighbors and the sensed distance to them (itself excluded). If the device is a source node, it returns 0.0. Eventually, the gradient value converges to the minimum distance from the nearest source node. On the right, a visual representation of the gradient field computed by the program at round 1. The source node is represented in yellow. Links are labeled with the sensed distance values, while nodes are labeled with the computed gradient value.

4 Contribution

This chapter presents the conceptualization of a proposed Scala-based architecture for enabling the development of cross-platform, polyglot and distributed libraries and frameworks.

Its discussion is structured as follows. First, the intents and application scenarios motivating the proposed architecture are discussed. Subsequently, the corresponding requirements and constraints are formalized. Thereafter, the elements composing the architecture are presented. Finally, the implications of adopting this design are analyzed.

4.1 Intents

The intents of the proposed architecture are to enable the development of distributed Scala libraries and frameworks to be both *cross-platform*, that is, able to run on multiple platforms and runtimes; and *polyglot*, that is, designed to be able to interoperate with its public interface from multiple programming languages. Both intents are achieved while maintaining a unified version of the components implementing the application logic of the software product.

4.2 Application scenarios

This architecture is well suited when the following scenarios arise:

- **Application environment heterogeneity.** The library or framework implements functionalities that should be deployed and executed in heterogeneous environments, spanning across multiple platforms and runtimes. This may arise from the heterogeneity of end-user deployment environments or the requirement that platform and runtime selection remain contingent upon the specific application context in which the library is used.
- **Polyglot end-user base.** The library is designed for developers working

across different programming languages, enabling team diversity, facilitating integration with existing codebases in various languages, and leveraging language-specific features and ecosystems.

- **Unified application logic.** The library exposes its functionalities through a unified API, regardless of the target platform, runtime or programming language. This ensures behavioral consistency across all supported environments, enabling seamless interoperability between programs utilizing the library.

4.3 Requirements

Requirements formally describe the boundary of applicability of the proposed architecture. These can be categorized into *user*, *system* and *implementation* requirements and address two types of stakeholders: *library users*, who are developers intending to use the libraries and frameworks designed following the proposed architecture; and *library developers*, who design and implement such libraries and frameworks.

User requirements

U1. Library users must be able to interact with the library from multiple programming languages through a consistent API, benefiting from language-specific ecosystem features. Supported languages include:

  - Scala;
  - Java;
  - JavaScript;
  - TypeScript;
  - C and C++.

U2. The library must support execution across heterogeneous platforms and runtimes to accommodate diverse library user deployment environments. Library users must be able to implement their own solution by building upon the library's core functionalities:

  - using Scala, with deployments possibly spanning across the following platforms and runtimes:
    - JVM;

  - JavaScript environments, both browser-based and server-side (Node.js);
  - Native environments, including desktop, server, and System on Chip (SoC) devices running on 64-bit and ARM architectures.
- using other supported languages, targeting their respective platforms. Except for Scala, Library Users should not need to compile code written for one platform to run on another; e.g., compiling C code to run on Node.js.

U3. Library users can distribute the execution of their application-specific code across multiple machines, with each node potentially running on different platforms, runtimes, and programming languages. The library provides distributed communication mechanisms that enable seamless interoperability and coordination among them.

System requirements

S1. Library developers must be able to implement the library logic once and reuse it across all supported platforms and runtimes. Modifications to the library logic are automatically reflected across all target platforms and runtimes.

S2. Library API must be consistent across all supported languages and behaves uniformly regardless of the target platform, runtime and programming language.

Implementation requirements

I1. Developers implement the library taking advantage of the full Scala language features and capabilities, enabling them to harness functional programming, type-safe abstractions, expressive syntax and compositional design patterns to facilitate the development of robust, scalable and distributed software systems.

I2. The bridge layer between programming languages should not rely on remote procedure calls (RPC) or inter-process communication (IPC) mechanisms to not incur in the performance penalties associated with these techniques.

4.4 Constraints

The proposed architecture imposes a fundamental constraint that significantly influences architectural decisions: all dependencies must support cross-compilation to all target platforms. Where such multi-platform support is unavailable, platform-specific implementations must be provided explicitly to bridge the abstraction gap. If an existing Scala library heavily relies on platform-specific dependency or its design is tightly coupled with a specific platform or runtime, adapting it to the proposed architecture may prove impractical or require substantial reengineering efforts. For example, a distributed Scala library built on Akka[1] and utilizing its remoting and clustering capabilities would require substantial redesign to support multi-platform execution, as Akka targets only the JVM. However, this challenge can be mitigated if these dependencies have been carefully abstracted as replaceable components. Furthermore, when platform-specific dependencies provide critical non-functional properties—such as fault tolerance or scalability—that multi-platform alternatives cannot yet match, adapting to a cross-platform architecture becomes not merely impractical but potentially unfeasible. In these cases, the tight coupling to a specific platform represents an architectural constraint rather than an implementation detail, and the proposed architecture may not constitute a suitable approach for such libraries without revisiting their core design principles.

4.5 Architectural elements

The proposed architecture, presented in Figure 4.1, is composed of three main components:

1. a **pure core module**, implementing the application logic of the library. This module is designed to be platform-agnostic and independent of any specific technology or runtime;
2. a **cross-platform infrastructure module**, responsible for enabling the distribution of the library functionalities across multiple end nodes. This includes a **general cross-platform and polyglot serialization binding**, providing the capability to marshal and unmarshal data structures exchanged between different platforms, runtimes and programming languages;
3. a **polyglot abstraction layer**, exposing a simplified and consistent interface API to library users through different programming languages.

The core module and the cross-platform infrastructure module together constitute the *Software Product*, which implements the library's core functionalities and delivers its primary value. The polyglot abstraction layer, by contrast, forms part of the *User Interface*, providing the programmatic interface through which

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

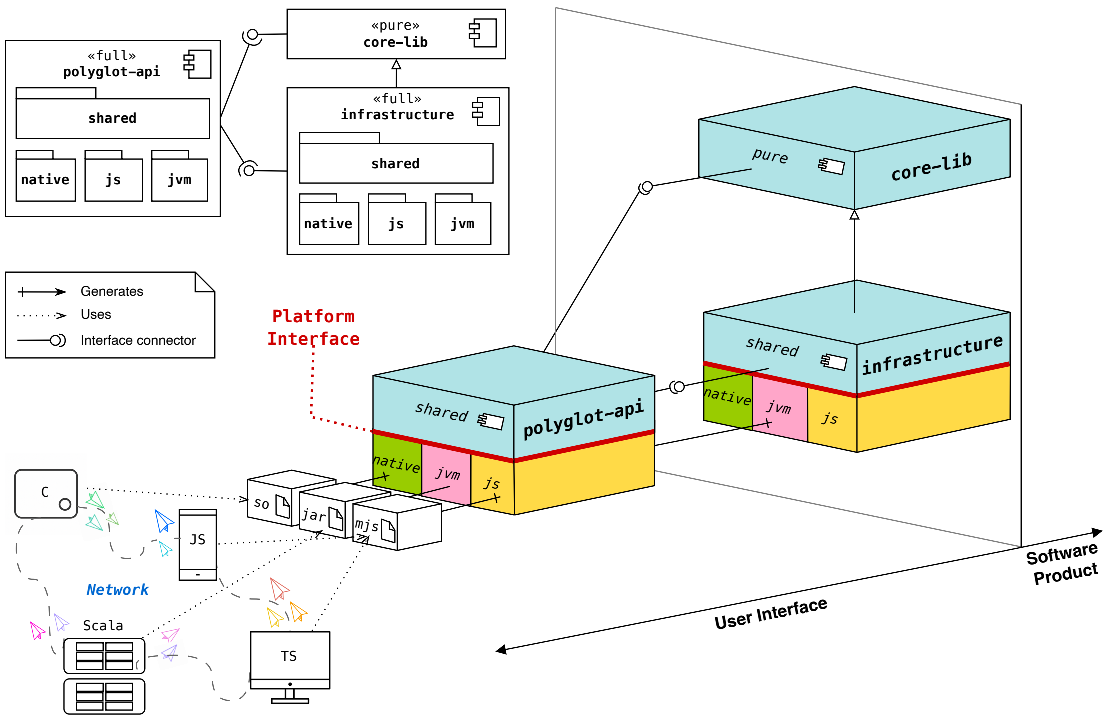


Figure 4.1: Proposed reference architecture for cross-platform, polyglot and distributed Scala libraries and frameworks.

library users interact with the software product using any of the supported programming languages. This separation enables library users to interact seamlessly with the library in their preferred programming language, accessing its functionalities through native packages generated by the polyglot library abstraction layer. Library developers, meanwhile, can focus on implementing the core logic without being concerned about language-specific details.

Once applied, library users can take advantage of the library as depicted in Figure 4.2. They can implement their application logic directly in Scala, leveraging its full language features and capabilities and cross-compiling it to all target platforms and runtimes, by consuming the respective library artifact from a Central repository (e.g., Maven Central). Alternatively, they can program their application logic in any of the supported programming languages, utilizing the corresponding artifacts generated by the polyglot abstraction layer and published to the respective platform-specific package repositories.

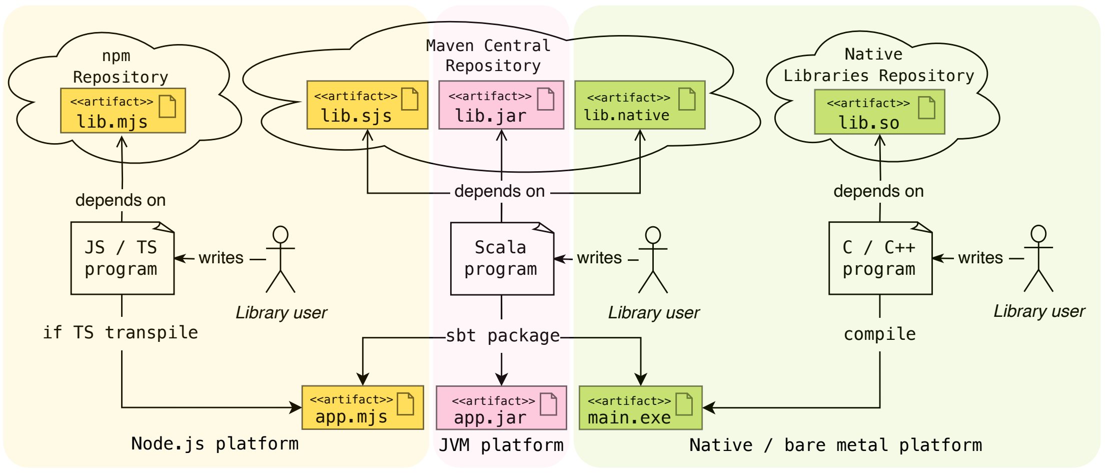


Figure 4.2: Library usage workflow enabled by the proposed architecture.

Pure library core module

The core module is the heart of the library, implementing its core application logic and functionalities in terms of domain modeling, business rules and application use cases. Its architecture is deliberately designed to be both platform- and technology-agnostic, ensuring the core logic to be reusable across multiple execution environments without requiring any modification or adaptation. This design can be achieved by adhering to the Hexagonal (Ports & Adapters) Architectural pattern [Coc05], which promote the decoupling of the core logic from external dependencies by defining clear interfaces (ports) and isolating the core logic from infrastructural concerns (adapters).

Implemented as a Scala pure multi-platform module, it enables the cross-compilation of the core logic to all target platforms and supported runtimes.

Cross-platform infrastructure module

This is the component dealing with infrastructural concerns. Despite the fact it is conceptually represented as a single module, depending on the complexity of the library and its requirements, it can be decomposed into multiple modules, each addressing specific infrastructural aspects. Its core responsibilities include the implementation of a distributed communication mechanism and a general serialization binding.

Cross-platform distribution module

For distributed communication, the library can leverage different underlying technologies and protocols depending on specific use cases and requirements, such as peer-to-peer or publish-subscribe messaging patterns. Regardless of the chosen communication model, the design must maximize code reuse across all supported platforms and runtimes. This is achieved through well-defined *Platform Interfaces* that abstract platform-specific capabilities and services, isolating them from the shared library logic. These interfaces are designed to satisfy both the library's functional requirements and the native paradigms of each target platform. For example, the interfaces use asynchronous API patterns that can be uniformly implemented across all supported environments. Platform-specific adapters then implement these interfaces, encapsulating only the minimal necessary platform-dependent code. Whenever possible, existing multi-platform libraries and frameworks should be leveraged. Conversely, when no suitable multi-platform library exists, custom platform-specific implementations must be developed by interacting directly with the underlying platform capabilities and ecosystem via Scala Native or Scala.js interoperability features. In both cases, all implementation adapters must conform to the defined Platform Interfaces to ensure maintainability and facilitate the integration of alternative platform-specific implementations.

Polyglot serialization binding

Concerning the serialization binding, it must be designed to be format-agnostic and interoperable across different programming languages and platforms. Format agnosticism is desirable to ensure flexibility and interchangeability of serialization formats, while cross-platform interoperability is a strict requirement. Indeed, if the serialization format is not compatible across all supported languages and platforms, the library would not be able to consistently exchange messages between different end nodes, thus undermining its distributed nature.

Selecting a serialization format requires evaluating multiple factors. Performance and efficiency trade-offs between textual and binary formats must be weighed against compatibility with all supported programming languages and platforms. Additionally, the choice between schema-based and schema-less formats affects the flexibility and evolvability of serialized data structures. A critical consideration is the degree of automation in serializer and deserializer generation: while automatic code generation simplifies development, it may limit the ability to define custom data types and structures that leverage language-specific abstractions. This tension is particularly relevant given that different programming languages offer distinct abstractions that cannot always be straightforwardly mapped.

Polyglot library abstraction layer

The polyglot abstraction layer exposes a uniform interface enabling cross-language interaction with the library's core functionalities. Two critical challenges converge to demand the presence of this layer: type system mismatches and limited type construct mapping.

Type system mismatches. Mismatches arise because both Scala Native and Scala.js address language semantics differences by reifying them into new types. For example, Scala.js introduce the `js.Map` type to represent JavaScript maps, distinct from Scala's `Map` type. Similarly, both projects define platform-specific types for interoperability: Scala.js provides `js.UndefOr` for nullable types and `js.FunctionN` for JavaScript functions, while Scala Native uses `null` for nullable types and `CFuncPtrN` for C function pointers. While this approach preserves type-safety, guarantees correct usage, and makes users aware of the underlying platform constraints and peculiarities, it prevents exposing a unified API across all supported languages. Without this layer, library developers would be forced to create different facades for both JavaScript and C, leading to code duplication, increased maintenance effort and potential inconsistencies.

Limited type construct mapping. This is a consequence of the fact that only a subset of Scala type constructs can be mapped and exposed to JavaScript and C. Indeed, while Scala code can be cross-compiled to both JavaScript and native binaries, not all Scala type constructs have a direct counterpart in these target languages. Consequently, the ability to cross-compile a Scala construct does not guarantee it can be exposed to or utilized by other programming languages. For example, Scala's rich type system includes Path Dependent Types [ARO14] and Implicit parameters [Ode+17], which have no direct equivalents in other programming languages.

The proposed polyglot abstraction layer tackles these challenges by introducing *abstract, language-independent types* that are *isomorphic* to Scala types. They are decoupled from any specific programming language or platform, yet maintain a one-to-one mapping to corresponding Scala types. Referred to as **Portable Types**, these are then instantiated within each supported platform, providing language-specific implementations and dedicated bidirectional conversion between them and their corresponding Scala equivalents.

Using this approach the core library API is exposed to library users through portable types, providing simplified interfaces as a thin wrapper around the core library API. Within the polyglot abstraction layer, the conversion between portable types and the corresponding Scala types used by the core library is handled. The exposed interfaces—referred to as **Portable Libraries**—can then be exported, packed and distributed as native packages for each supported programming language, since, during cross-compilation, Portable Types are translated to the corresponding language-specific types. Essentially, Portable Types act as a small standard library for this module, written in a "de-powered" version of Scala that is limited, in its expressiveness, to the constructs that can be mapped and exported to all supported programming languages.

Beyond collection-types, each Portable Library can extend the set of portable types by defining domain-specific data types to model the library domain, mirroring standard type definition practices. These must maintain the same isomorphic mapping principle, ensuring seamless conversion between platform-specific incarnations and Scala types.

Implementation-wise, Portable Libraries are structured as a thin wrapper around the core library API, using only Portable Types in their public interface. Internally, the logic is delegated to the core library after converting Portable Types to their corresponding Scala types using the provided isomorphic mappings, as shown in Figure 4.3.

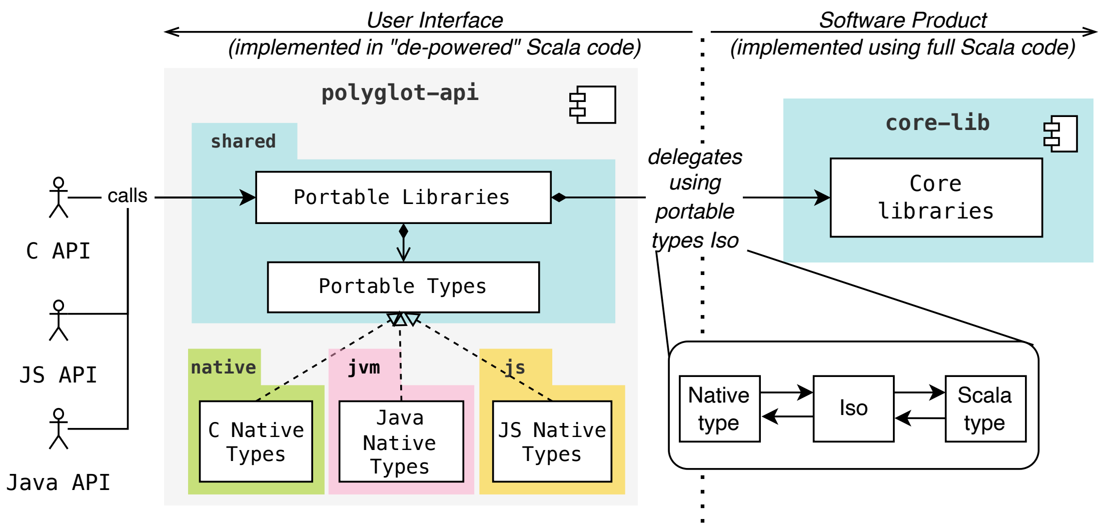


Figure 4.3: Clients interact with the core library API through Portable Libraries, which expose a simplified interface using Portable Types. Thanks to language-specific instantiations of Portable Types and appropriate isomorphisms, Portable Libraries simply delegate to the core library API their implementation.

Notably, the polyglot abstraction layer must account for fundamental differences among all supported programming languages, modeling and abstracting these distinctions to present them through a unified API. These differences encompass:

- error handling mechanisms, such as exceptions in JavaScript and error codes in C;
- API design paradigms, including asynchronous and synchronous styles;
- equality and hashing semantics;
- memory management models.

Chapter 5 presents how Portable Types and Portable Libraries can be implemented to model and abstract these language-level differences.

4.6 Consequences

This architecture enables the development of cross-platform, polyglot and distributed Scala libraries and frameworks by addressing the challenges associated with platform heterogeneity, language interoperability and distributed communication. Its main focus is to maximize code sharing and reuse across all supported platforms and languages, and provide a consistent and unified API to library users.

Some considerations and trade-offs are presented hereafter.

1. **API Maintenance.** While every code change in the core library is automatically reflected across all supported platforms and runtimes, the addition of new API functionalities requires extending the polyglot abstraction layer to expose the new features to library users. Despite not being a significant overhead, since the polyglot abstraction layer is designed to be thin and minimal, it still requires some additional effort to maintain and evolve.
2. **Performance Overhead.** Interoperability across programming languages is achieved through type conversion, which may introduces performance overhead if compared with native implementations, particularly when handling complex data structures or large data volumes. However, this overhead is significantly lower than RPC or IPC approaches, which require additional serialization and boundary crossing costs.
3. **Data Expressiveness.** The expressiveness of exchanged messages between nodes is limited to the capabilities of the adopted cross-platform and polyglot serialization format. This normally does not pose significant limitations since exchanged messages are represented as *records*——immutable data classes composed of fields with no additional behavior. However, complex data structures or language-specific constructs may not be fully representable and may require simplifications or alternative representations.

5 Implementation

This chapter presents the implementation of the abstract architecture proposed in Chapter 4. The discussion is divided in two main sections: the first focuses on implementation details that are library-independent, i.e., applicable to any library aiming to provide cross-platform and polyglot capabilities; the second section delves into specific implementation details within the ScaFi3 (henceforth simply ScaFi) library.

5.1 Library-Independent Module Details

The general implementation strategy to provide cross-platform and polyglot capabilities is centered around the implementation of a general cross-platform serialization bindings and a polyglot API layer.

5.1.1 Serialization binding

Any distributed Scala library, in spite of its specific network protocol or communication model, requires serialization and deserialization capabilities to exchange messages between parties. A general design can be adopted to provide these capabilities while being agnostic to the specific used serialization format.

Technically, this is achieved via a combination of Scala 3 *type classes* [OMO10] and *type lambdas*. The core design is based on `Encodable` and `Decodable` type classes, shown in Listing 5.1, whose purpose is to express the capability of a type to be, respectively, encoded to and decoded from arbitrary serialization formats. Expressing them as type lambdas enables their usage as context bounds on the value parameters of functions dealing with distribution, like presented in Listing 5.2 for the `exchange` construct. Beyond the AC logic presented therein, the `exchange` represents a typical example of a distribution-related primitive requiring encoding and decoding capabilities for its value type in order to fulfill its functionality. Additionally, depending on the specific semantics of each primitive, only one side of the serialization process may be required. For instance, `alignedMessages` requires only encoding capabilities, while `writeValue` requires only decoding. Thanks to *ad-hoc polymorphism*, this ensures that distribution-related primitives can only be used with values having the required encoding and decoding capabilities in scope.

Listing 5.1: Encodable and Decodable type classes and type lambda definitions.

```scala
trait Encodable[-From, +To]:
  def encode(value: From): To

trait Decodable[-From, +To]:
  def decode(data: From): To

trait Codable[Message, Format] extends Encodable[Message, Format]
   with Decodable[Format, Message]

// Encodable and Decodable type lambdas to be used as type bounds
type CodableFromTo[Format] = [Msg] =>> Codable[Msg, Format]

type EncodableTo[Format] = [Msg] =>> Encodable[Msg, Format]

type DecodableFrom[Format] = [Msg] =>> Decodable[Format, Msg]
```

Listing 5.2: Example of encoding and decoding capabilities at work in the `exchange` construct.

```scala
// inside this function, Values can be both encoded and decoded
def exchange[Format, Value: CodableFromTo[Format]](
  initial: SharedData[Value]
)(
  f: SharedData[Value] => ReturnSending[SharedData[Value]]
): SharedData[Value] =
  alignmentScope("exchange"): () =>
    val messages = alignedMessages
    val field = Field(init(localId), messages)
    val (ret, send) = f(field)
    writeValue(send.default, send.alignedValues)
    ret

// Get and decode neighbor messages using Value Decodable instance
def alignedMessages[Format, Value: DecodableFrom[Format]]
    : Map[DeviceId, Value] = ...

// Encode + send neighbor messages using Value Encodable instance
def writeValue[Format, Value: EncodableTo[Format]](
  default: Value,
  overrides: Map[DeviceId, Value]
): Unit = ...
```

User-side, to provide encoding and decoding capabilities for a specific data type and serialization format, it is sufficient to provide implicit instances of the `Codable` type class in scope (which combines `Encodable` and `Decodable` capabilities) for those types. The Scala compiler will automatically summon the correct instance based on the type information available at compile time, ensuring users to call the distribution-related primitives without having to manually pass encoding and decoding functions around or explicitly specify format types. Examples of user-defined instances of the `Encodable` and `Decodable` type classes for different serialization formats are shown in Listing 5.3.

Listing 5.3: User-defined Encodable and Decodable instances for custom data types.

```scala
// A Codable that does not perform any transformation on the
   messages, leaving them as-is - useful for testing or
   simulation
given forInMemoryCommunications[Message]
    : Codable[Message, Message] with
  inline def encode(msg: Message): Message = msg
  inline def decode(msg: Message): Message = msg

// A Codable for encoding and decoding stringified messages in
   binary format
given forStringsInBinaryFormat
    : Codable[String, Array[Byte]] with
  def encode(msg: String): Array[Byte] = msg.getBytes(UTF_8)
  def decode(bytes: Array[Byte]): String =
    new String(bytes, UTF_8)

// A Codable for encoding and decoding Protobuf messages in
   binary format
given forProtobufInBinaryFormat[T <: GeneratedMessage](
  using companion: GeneratedMessageCompanion[T]
): BinaryCodable[T] = new BinaryCodable[T]:
  def encode(value: T): Array[Byte] = value.toByteArray
  def decode(data: Array[Byte]): T = companion.parseFrom(data)
```

The general implementation of the type classes with respect to the serialization format allows:

- to leave its choice to the library user, who can choose the most appropriate format for their specific use case. For example, in Listing 5.3 a production-ready Protobuf-based binary format is provided using ScalaPB[1], a popular

[1]`https://scalapb.github.io`

cross-platform Scala library to work with Protocol Buffers;

- to employ in-memory Codable instances that do not perform any transformation on the messages, leaving them as-is, in non-distributed environments, like simulation and local testing where usually networking operations are mocked or bypassed, making the serialization process unnecessary. Moreover, leveraging Scala 3 inline mechanism, the compiler is able to completely eliminate encoding and decoding operations in such scenarios avoiding any runtime overhead.

While the library remains unopinionated about the serialization format to be used, with format selection impacting exclusively library user code rather than library internals, the only requirement for enabling polyglotism is that the serialization format supports cross-language interoperability. Meeting this requirement, Protocol Buffers has been adopted for all practical implementations and examples in this work.

Protocol Buffers (Protobuf)[2] is a widely adopted, schema-driven binary serialization framework developed by Google for efficient encoding of structured data in distributed systems. Its use is justified by several factors:

- **Multi-language support**. Protobuf is schema-driven, meaning it uses schemas (`.proto` files) to define the structure of the data being serialized, employing a language-agnostic Interface Definition Language (IDL) that serves as contract between parties. Starting from a schema definition, Protobuf compiler (`protoc`) can generate the corresponding classes ready to be used for serialization and deserialization in multiple programming languages, including, but not limited to, C, C++, Scala, Java, Python, JavaScript, TypeScript and C#. An example of Protobuf schema definition is shown in Listing 5.4, showcasing the syntax and major features of Protobuf IDL.
- **Efficiency and performance**. Serializing data into a binary format, Protobuf achieves compact message sizes. This results in smaller payloads, making it more efficient than text-based formats, like JSON, both in terms of network bandwidth and storage requirements. Moreover, Protobuf is designed for high performance, enabling fast serialization and deserialization operations, which is crucial in distributed systems where low latency is often a requirement.
- **Backward and forward compatibility**. Protobuf supports schema evolution, allowing developers to add or remove fields from message definitions without breaking existing implementations: unknown fields are ignored, and

[2]`https://protobuf.dev`

missing fields are treated as having default values. This is particularly important in distributed systems where different components may be updated independently over time.

- **Scala cross-platform libraries support**. The Scala ecosystem provides cross-platform Protobuf support through the ScalaPB library, which targets JVM, JS, and Native platforms, making it a suitable choice for ScaFi's cross-platform goals.

Listing 5.4: Example of Protobuf schema definition for a sensor data message.

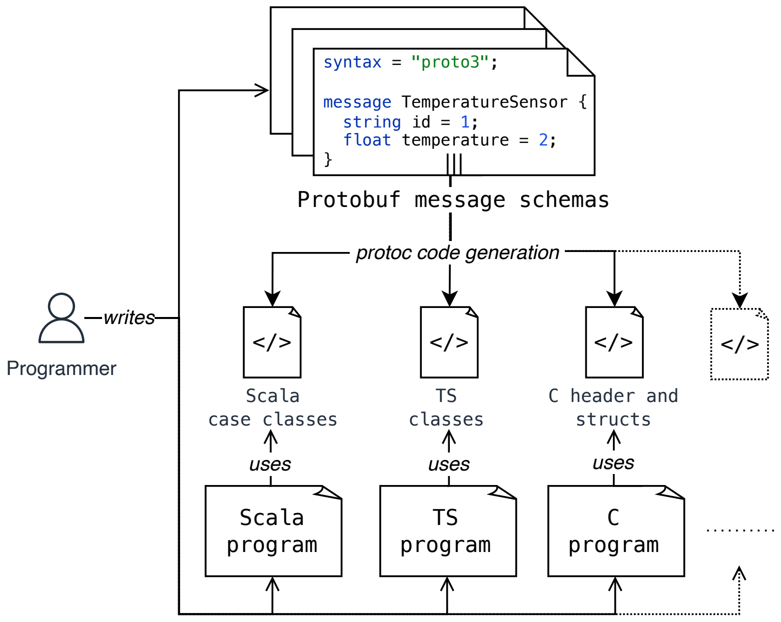
```
syntax = "proto3";

// Imports Google's Timestamp from the Protocol Buffers "Well-
   Known Types"
import "google/protobuf/timestamp.proto";

// Main message representing a temperature sensor reading
message TemperatureMeasurement {
    // Unique identifier for the sensor
    string sensor_id = 1;
    // Temperature value in Celsius using double precision
    double temperature_celsius = 2;
    // Timestamp from Google's library ensuring consistent time
       format across different programming languages
    google.protobuf.Timestamp recorded_at = 3;
    // Optional field: location may be omitted
    optional string location = 4;
    // Repeated field: allows multiple tag values per measurement
    repeated string tags = 5;
    // Optional nested message
    optional Metadata metadata = 6;
}

// Contains technical metadata about the sensor device
message Metadata {
    // Firmware version identifier
    string firmware_version = 1;
    // Optional battery level
    optional double battery_level = 2;
    // Signal strength indicator
    int32 signal_strength = 3;
}
```

The Protobuf IDL is highly expressive, supporting complex data structures with nested messages, enumerations, optional and repeated fields, maps, and well-known types like timestamps. However, as anticipated in Section 4.5, the usage of a schema-driven serialization framework like Protobuf, despite its advantages, reduces the expressiveness of the data types that can be exchanged between devices to the ones supported by the library schema definition language. In the case of Protobuf, this constrains data exchange to record-like structures. In the context of AC, for example, this prevents exchanging higher-order functions, like lambdas or closures, unless only their method references are exchanged, assuming the remote device has the corresponding function implementation available locally—but this is beyond the scope of this work.



```
syntax = "proto3";

message TemperatureSensor {
  string id = 1;
  float temperature = 2;
}
```

Figure 5.1: Protobuf-based serialization/deserialization aggregate programming workflow.

The designed workflow for Protobuf-based serialization and deserialization is illustrated in Figure 5.1. First, the programmer defines the Protobuf schemas for the data types and messages to be exchanged. Then, using language specific plugins, the Protocol Buffers compiler generates language-specific stubs containing ready-to-use serialization and deserialization logic. Depending on the target language, `protoc` generates different artifacts: for example, for Scala it generates case classes, while for C it generates header files with struct definitions and `.c` files with serialization and deserialization functions. Finally, the programmer implements the program in their language of choice using the library's exported API and the generated record-like data types for data exchange.

Implementing codable instances in other languages differs from Scala's approach. While Scala uses context bounds, other supported programming languages (notably, Java, JavaScript and C) lack an equivalent mechanism. Therefore, their APIs express codable capabilities via upper type bounds on values, which are made interoperable with Scala's context bounds using isomorphisms, as discussed in Section 5.1.2. In particular, for JavaScript and TypeScript, exchanged value are expected to conform to a generic contract expressed as an interface with `encode` and `decode` methods, as shown in Listing 5.5.

Listing 5.5: Codable interface definitions for JavaScript and TypeScript.

```
trait JSCodable extends js.Object:
  def encode(message: js.Any): js.Any
  def decode(data: js.Any): js.Any
```

In C, codable requirements on the value are expressed leveraging struct-based polymorphism with function pointers: a `BinaryCodable` struct defines the required operations as function pointers, as shown in Listing 5.6; concrete value types embed this struct as their first member, allowing pointers to be safely cast to `BinaryCodable`, thereby achieving polymorphic behavior through a common interface.

Listing 5.6: Codable struct definition for C.

```
typedef struct BinaryCodable {
    const uint8_t* (*encode)(const void *data, size_t *size);

    const void* (*decode)(const uint8_t *buffer, size_t size);

    void (*free)(void* data);
} BinaryCodable;


// Example: SensorReading type implementing BinaryCodable
typedef struct {
    BinaryCodable codable;
    uint64_t timestamp;
    float temperature;
    float humidity;
} SensorReading;

// Function prototypes for SensorReading creation, encoding,
   decoding, and freeing...
```

In Java, instead, values are expected to implement the Scala `Codable` trait directly, as Scala and Java are interoperable on the JVM platform.

To simplify the creation of codable instances for Protobuf-generated types from other languages, the library can provide automatic JavaScript conversions and C helper functions automatically mapping Protobuf generated types to their corresponding codable representations. As a result, users can use Protobuf-generated types directly in their programs or create codable instances from them via helper functions, avoiding any manual encoding/decoding implementation. Concrete examples will be shown in Chapter 6 when discussing polyglot ScaFi real-use cases.

5.1.2 Polyglot API module

The goal of this module is to implement the polyglot abstraction layer presented in Section 4.5. Its core components are the definition of Portable Types, their isomorphisms with Scala counterparts, and the implementation of Portable Libraries. The general design is illustrated in Figure 5.2 and discussed hereafter.

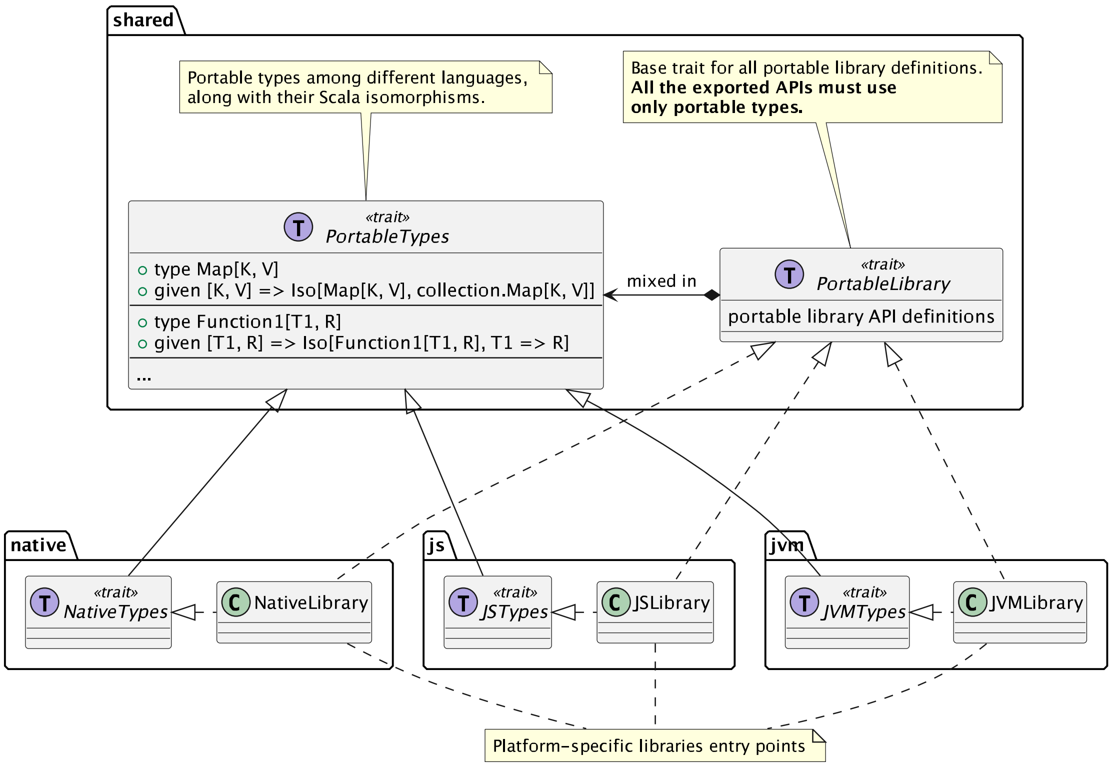

shared
Portable types among different languages, along with their Scala isomorphisms.
Base trait for all portable library definitions. **All the exported APIs must use only portable types.**
«trait» PortableTypes
type Map[K, V]
given [K, V] => Iso[Map[K, V], collection.Map[K, V]]
type Function1[T1, R]
given [T1, R] => Iso[Function1[T1, R], T1 => R]
...
mixed in
«trait» PortableLibrary
portable library API definitions
native
«trait» NativeTypes
NativeLibrary
js
«trait» JSTypes
JSLibrary
jvm
«trait» JVMTypes
JVMLibrary
Platform-specific libraries entry points

Figure 5.2: Portable Types and libraries general design.

Portable Types are implemented as Abstract Type Members [OZ05] in a trait with deferred abstract given isomorphisms between them and their Scala counterparts (see Listing 5.7).

Listing 5.7: Portable Types trait with deferred isomorphisms.

```scala
// Portable types that can be used across different languages
trait PortableTypes:
  export it.unibo.scafi.utils.libraries.Iso
  export it.unibo.scafi.utils.libraries.Iso.given

  // A portable Map, isomorphic to Scala's collection.Map
  type Map[K, V]
  given [K, V] => Iso[Map[K, V], collection.Map[K, V]] = deferred

  // A portable Sequence isomorphic to Scala's collection.Seq
  type Seq[V]
  given [V] => Iso[Seq[V], collection.Seq[V]] = deferred

  // A portable 1-arg function type, isomorphic to Scala's T1 => R
  type Function1[T1, R]
  given toScalaFunction1[T1, R]
      : Conversion[Function1[T1, R], T1 => R]
  // other types...
```

Listing 5.8: Iso type class representing an isomorphism between two types, A and B.

```scala
trait Iso[A, B]:
  def to(a: A): B
  def from(b: B): A

object Iso:

  inline def apply[A, B](
    inline toFn: A => B,
    inline fromFn: B => A
  ): Iso[A, B] = IsoImpl(toFn, fromFn)

  class IsoImpl[A, B](val toFn: A => B, val fromFn: B => A)
     extends Iso[A, B]:
    override def to(a: A): B = toFn(a)
    override def from(b: B): A = fromFn(b)

  // automatic conversions
  given [A, B](using iso: Iso[A, B]): Conversion[A, B] with
    inline def apply(a: A): B = iso.to(a)

  given [A, B](using iso: Iso[A, B]): Conversion[B, A] with
    inline def apply(b: B): A = iso.from(b)
```

An Isomorphism is a bidirectional conversion, represented as a type class, between two types that preserves the structure and semantics of the data being converted (see Listing 5.8). Where only one-way conversions are sufficient, simple Scala Conversions are used instead. Indeed, while isomorphisms provide a more general and powerful abstraction for bidirectional conversions, not always both directions are needed. For example, for function types, they are usually used only to express callbacks. Therefore, converting portable function types to Scala's regular types suffices, since the library only calls user-provided callbacks and never requires the reverse conversion.

Thanks to Scala 3 Abstract Types and deferred givens, the actual implementations of the isomorphisms are deferred to platform-specific modules, allowing to provide different implementations for each target language. For example, in the Scala Native module, the isomorphisms are implemented using C native types and data structures, as shown in Listing 5.9. Equivalently, on JVM and JavaScript, standard Java and Scala.js types and data structures are used, respectively. When no specific language type is available in the respective platform, like for Sequence on Native, custom implementations can be provided.

Listing 5.9: Fragment of the C native implementation of Portable Types.

```scala
trait NativeTypes extends PortableTypes:

  // Portable sequence is mapped to a C generic array struct
  override type Seq[V] = Ptr[CArray]
  override given [V] => Iso[Seq[V], collection.Seq[V]] = Iso(
    cArray =>
      for i <- 0 until (!cArray).size.toInt
      yield (!cArray).items(i),
    scalaSeq =>
      val arr = freshPointer[CArray]
      (!arr).size = scalaSeq.length.toCSize
      (!arr).items = freshPointer[Ptr[Byte]](scalaSeq.length)
      scalaSeq.zipWithIndex.foreach((v, i) => (!arr).items(i) = v)
      arr,
  )

  // Portable 1-arg function is mapped to a C function pointer:
  // R (*f)(T1)
  override type Function1[T1, R] = CFuncPtr1[T1, R]
  given toScalaFunction1[T1, R]
      : Conversion[Function1[T1, R], T1 => R] with
    inline def apply(f: Function1[T1, R]): T1 => R = f.apply

  // ...
```

When implementing Portable Types isomorphisms, it is important to be aware of subtle differences in the behavior of certain data structures across target languages. For example: JVM `Maps` collections rely on `equals` and `hashCode` methods for key comparison, while JavaScript objects use reference equality. This is generally not an issue when dealing with primitive types, but can lead to unexpected behaviors when using complex user-defined types as keys. In this case, a possible solution is to wrap user-defined types in a JVM wrapper class that overrides `equals` and `hashCode` methods to provide the desired behavior.

As shown in Figure 5.2, Portable Libraries result in traits mixing in the Portable Types trait to gain access to them and their isomorphisms. Ultimately, Portable Libraries are implemented in platform-specific modules incarnating the abstract Portable Types with the actual target language implementations. The resulting libraries expose APIs exclusively relying on Portable Types in their signatures, while internally leveraging the provided isomorphisms to fall back to Scala types when needed, fully exploiting the Scala type system. This solution allows cross-language support. Indeed, during cross-compilation, at the entry points of platform-specific library classes, where refinements are present, abstract type members are substituted with the concrete target language types and made visible throughout the scopes in which those refinements apply. Public API methods are then exported using target platform appropriate annotations, respectively `@JSExport` for Scala.js and `@exported` for Scala Native, ensuring the generated artifacts emit the correct symbols for being consumable from the target language. For Scala.js annotations, they can be placed directly into the shared code where Portable Libraries are defined. Indeed, Scala.js provides these annotations as a provided dependency containing only annotation definitions. This allows annotated shared code to compile on all platforms, while the actual annotation processing happens only during Scala.js compilation. Unfortunately, Scala Native does not provide such mechanism (though it can be implemented as a future improvement). Consequently, Native-specific annotations need to be placed in the Native module, where the actual platform-specific library entry points are defined.

In the following, two important implementation aspects of the polyglot API layer are discussed: how Portable Types can deal with synchronous and asynchronous mismatches, and how memory management can be handled to avoid burdening the library user.

Portable types can abstract over synchronous and asynchronous mismatches

It is common for APIs to accept or export functions and callbacks for deferred invocation by the library, which may perform suspending operations such as network access or file I/O. The problem in representing such callbacks as regular functions with their corresponding Portable Types is that, in the JavaScript platform, suspending operations are represented using asynchronous constructs, while on JVM and Native platforms they are typically represented as blocking operations. For example, consider the callback fed to the `Runtime` ScaFi Engine for reacting to each round result, presented in Listing 5.10: after processing the result, the callback suspends for one second before indicating whether to continue or stop the computation. In JavaScript, that callback must be represented as an asynchronous function returning a `Promise`, while in JVM and Native, it can be represented as a regular function performing a blocking sleep operation.

Listing 5.10: Example of round result callback function in TypeScript.

```
await Runtime.engine(deviceId, port, neighbors, (lang) =>
    aggregateProgram(lang), async (result) => {
     console.log(`Round ${currentRound}: ${result}`);
     await sleep(1_000);
     return currentRound++ < rounds;
});
```

To overcome this mismatch, Portable Types can be used to abstract over synchronous and asynchronous function types. This is achieved by defining a new Portable Type, called `Outcome`, that maps to a `Promise`-based type in JavaScript and to a regular type in JVM and Native, and having a corresponding isomorphism with Scala Futures (Listing 5.11). The `Outcome` type enables defining ScaFi `engine`-like function, allowing callbacks to be represented uniformly across all target platforms while respecting their asynchronous or synchronous nature. As a collateral consequence, the implementation of these functions, must be programmed using asynchronous constructs (using Scala Futures) to uniformly handle both synchronous and asynchronous callbacks.

Listing 5.11: Definition of the Outcome portable type.

```
trait PortableTypes:
  type Outcome[T]
  given [T] => Iso[Outcome[T], Future[T]] = compiletime.deferred

trait JSTypes extends PortableTypes:
  override type Outcome[T] = js.Promise[T] | T
  override given [T] => Iso[Outcome[T], Future[T]] = Iso(
    {
      case p: js.Promise[?] => p.toFuture.asInstanceOf[Future[T]]
      case v => Future.successful(v.asInstanceOf[T])
    },
    f =>
      given ExecutionContext = JSExecutionContext.queue
      js.Promise[T]: (resolve, reject) =>
        f.onComplete:
          case Success(value) => resolve(value)
          case Failure(exception) => reject(exception),
  )
trait NativeTypes extends PortableTypes:
  override type Outcome[T] = T
  override given [T] => Iso[Outcome[T], Future[T]] =
    Iso(Future.successful, Await.result(_, Duration.Inf))

trait PortableRuntime:

  def engine[Result](
    localId: DeviceId,
    port: Int,
    neighbors: Map[DeviceId, Endpoint],
    program: Function1[AggregateLibrary, Result],
    onResult: Function1[Result, Outcome[Boolean]],
  ): Outcome[Unit]
```

Memory Management

Another important aspect to carefully consider when designing the polyglot API layer is the memory management. Unlike JVM and JavaScript platforms, which provide automatic memory management via garbage collection, Native platforms offer only partially automatic memory management. Indeed, Scala Native provides two levels of memory management:

- automatic memory management for Scala-allocated objects on the heap, via a configurable Native Garbage Collector ensuring that Scala objects are automatically collected when no longer reachable (see Appendix A);
- semi-automatic memory management mechanism for C-allocated data structures, bound to a temporary delimited scope, via the `Zone` context manager. Programmers can allocate C data structures within a `Zone` context, ensuring that all allocated memory is automatically freed when exiting the `Zone` scope. An example is shown in Listing 5.12.

Listing 5.12: Example of Scala Native Zone usage for C memory management.

```scala
Zone:
  // converts the Scala Byte Array into a C structure using the C
  // decoder which accepts a uint8_t*
  decode(bytes.toUint8Array, bytes.length.toCSize)
// < here memory deallocation happen

extension (bytes: Array[Byte])
  def toUint8Array(using Zone): Ptr[uint8_t] =
    // alloc requests memory sufficient to contain the received
    // bytes; memory is automatically freed when exiting the Zone
    alloc[uint8_t](bytes.length).tap(writeTo)

  def writeTo(ptr: Ptr[uint8_t]): Unit =
    for i <- bytes.indices do !(ptr + i) = bytes(i).toUByte
```

Unfortunately, when interoperating with C code, memory management becomes more complex since both C and Scala Native sides must explicitly allocate unmanaged memory to share data structures. In such scenarios, developers must manually allocate and deallocate memory to avoid memory leaks and dangling pointers. This is tedious and error-prone, often requiring careful tracking of ownership and lifetimes of allocated memory, which can easily become unmanageable in complex interoperability scenarios. For example, in the context of the Field Calculus library, Fields are exchanged and converted several times in a single round of computation and need to be freed only after the entire round completes, at which point the original reference is typically no longer reachable.

Fortunately, in some contexts, it is possible to implement the polyglot API layer such that memory management is handled internally and transparently to the library user. This is achieved by identifying clear memory management boundaries within the library's computation model where deallocation can safely occur. For instance, in ScaFi, the round-based execution makes it possible to define such boundaries at the beginning and end of each computation round. Consequently, the polyglot layer can internally track all C-allocated data structures created within each execution scope and automatically free them at the scope's end (e.g., in ScaFi, at the end of each round). This mechanism relieves the library user from the burden of manually managing memory, simplifying the API usage and reducing the risk of memory-related issues.

The implementation of this mechanism is encapsulated in the `MemorySafeContext` trait presented in Listing 5.13, which provides a scoped memory region for C-allocated data structures. On Native platforms, it is implemented by leveraging the C standard library's memory allocation functions (`malloc`, `free`), while on

JVM and JavaScript platforms it is implemented as a no-op.

Listing 5.13: MemorySafeContext trait for memory-safe operations in a scoped memory region.

```scala
trait Allocator:
  type ManagedObject
  type Disposer = ManagedObject => Unit

  private var trackedObjects
    : Map[ManagedObject, Option[Disposer]] = Map.empty

  // Track an object for auto disposal when the arena is collected
  def track(obj: ManagedObject)(disposer: Disposer): Unit =
    synchronized(trackedObjects += (obj -> Some(disposer)))

  // Disposes all managed objects tracked by this arena
  def collect(): Unit = synchronized:
    trackedObjects.foreach((o, d) => d.fold(dispose)(_(o)))
    trackedObjects = Map.empty

  // Default disposal action for managed objects
  def dispose(obj: ManagedObject): Unit

trait MemorySafeContext:
  type Arena <: Allocator

  // Executes the given block safely within an `Arena` context.
  def safelyRun[T](block: Arena ?=> T): T

  // a shorthand to avoid summononing the arena at call site
  inline def collect()(using arena: Arena): Unit = arena.collect()
```

When the identified boundary is started, a new `MemorySafeContext` is created and passed implicitly to all components involved in the computation, which can use it to safely allocate C data structures. Allocating memory allows the context to track all allocated pointers, ensuring they can be freed later. At the end of the execution scope, all allocated memory within the context is automatically disposed, ensuring no memory leaks occur. Notably, this mechanism, differently from the `Zone` context manager, is able to track and manage memory allocated by library users during the computation scope. This can be achieved by implementing isomorphisms conversion to track all C-allocated data structures when converting from Portable Types to Scala types. In this last case, values are expected to be allocated, C-side, as a struct embedding the freeing logic as a function pointer, like shown in Listing 5.6 for the `BinaryCodable` struct. For example, in ScaFi, Fields created by user-defined functions within the aggregate program are tracked when performing conversions and disposed in its entirety, including all codable values contained therein, at the end of the round.

To show a concrete example of how the `MemorySafeContext` is used in practice, Listing 5.14 presents a fragment of the ScaFi Portable Engine implementation, showcasing how the context is created at the beginning of each round of computation and passed implicitly to all components involved in the round execution. In the end, when the round completes, all C-allocated data structures created during the round are freed.

Listing 5.14: Memory Safe Context usage in the ScaFi Portable Engine.

```scala
trait ScafiEngineBinding extends PortableRuntime:
  self: PortableTypes =>

  inline override def engine[Result](
      deviceId: DeviceId,
      port: Int,
      neighbors: Map[DeviceId, Endpoint],
      program: Function1[AggregateLibrary, Result],
      onResult: Function1[Result, Outcome[Boolean]],
  ): Outcome[Unit] = safelyRun: // creates safe memory region
    val network = socketNetwork(deviceId, port, neighbors)
    network
      .start()
      .flatMap: _ =>
        ScafiEngine(network, exchangeContextFactory)(
          program(library) // program is used in the safe context
        ).cycling(onResult.apply)
      .andThen(_ => Future(network.close()))
      .andThen(reportAnyFailure)

  extension [Result](engine: Engine[Result])
    def cycling(onResult: Result => Future[Boolean])(using Arena):
        Future[Unit] =
      for
        cycleResult <- Future(engine.cycle())
        outcome <- onResult(cycleResult)
        _ = collect() // allocated tracked memory is disposed
        _ <- if outcome then engine.cycling(onResult) else
          successful(())
      yield ()
```

5.2 ScaFi3-specific Implementation Details

Building upon the ScaFi library architecture presented in Section 3.3, this work extends it to support distributed aggregate computing and polyglot capabilities. Figure 5.3 illustrates the module structure of the ScaFi architecture and the dependencies among them, which comprise:

- a **distribution module**, handling distributed communication, messages exchange, and serialization and deserialization;
- a **polyglot API module**, providing polyglot capabilities to the library;
- an **integration module**, where integration tests are performed to ensure the correct functioning of the polyglot and distributed capabilities.

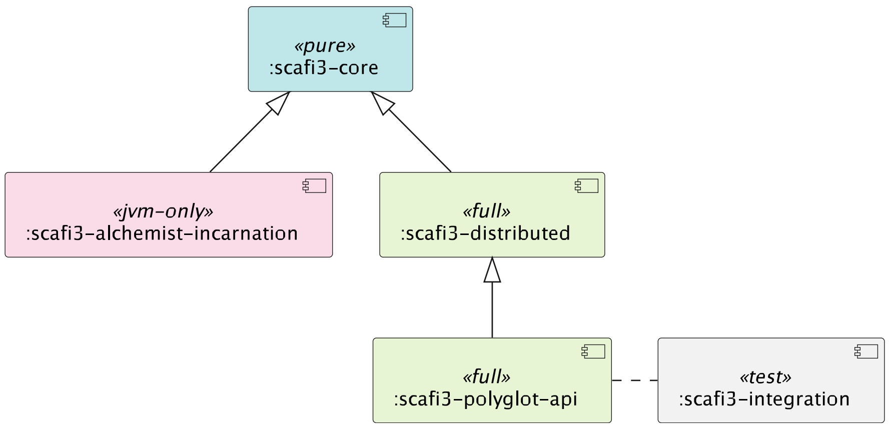


Figure 5.3: UML component diagram of the ScaFi architecture.

5.2.1 Distribution module

Enabling distribution in ScaFi requires implementing an abstract network manager that is able to send and receive Value Trees from and to neighbor devices. The AC framework abstract over the specific protocol used to communicate with neighbors and their discovery mechanism, allowing the implementation of different network managers for different protocols and scenarios.

As an initial step in ScaFi's evolution toward full-featured distributed capabilities, a socket-based network manager has been implemented. This foundation layer employs stream, TCP-based connection-oriented sockets, intentionally prioritizing core communication reliability over advanced distributed features, which remain subjects for future work. Indeed, despite the low-level nature of sockets, they provide a foundational abstraction layer that many higher-level protocols ultimately rely upon (such as HTTP, MQTT, etc.), making them a suitable starting point for building extensible communication mechanisms. Consequently, as shown in Figure 5.4, each device is bound to a specific endpoint (IP address and port) and communicates point-to-point with its neighbors.

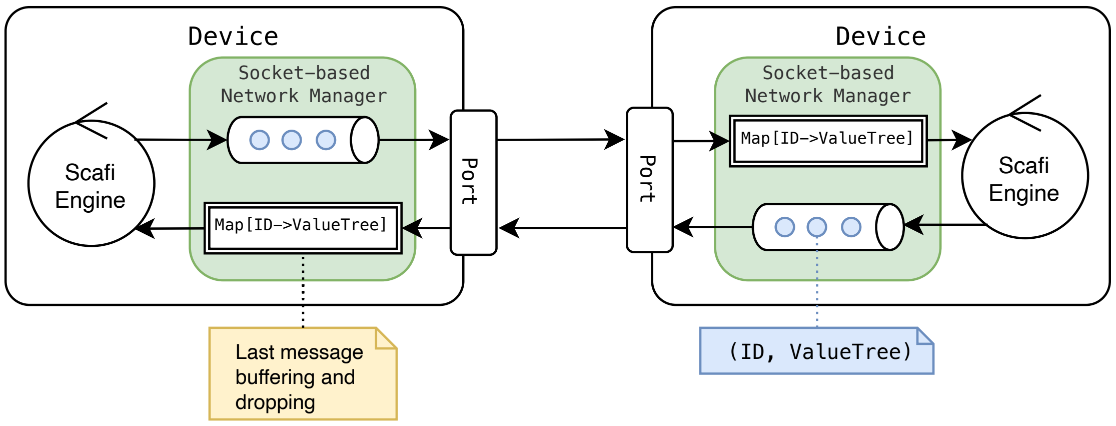


Figure 5.4: Socket-based network manager high level architecture.

The UML class diagram of the socket-based network manager is shown in Figure 5.5. It adopts an asynchronous API design leveraging Scala Futures, with two primary components operating concurrently to handle bidirectional communication:

- the incoming connection `Listener`, continuously listening for incoming messages from neighbor devices. Received Value Trees are stored in a thread-safe `Map` that retains only the most recent one from each neighbor, according to a configurable `Retention Policy` that defines, in absence of new messages, how long a message should be retained before being discarded. The up-to-date view of all the received neighbors' Value Trees is made available to the ScaFi Engine through the `receive` method at the beginning of each round of computation;
- the outgoing message channel, providing a non-blocking interface for message transmission. Upon the end of each round of computation, the ScaFi Engine invokes the `send` method of the network manager to dispatch the device's current Value Tree to all its neighbors. For each neighbor, the Value Tree is pushed through the channel and, asynchronously, dispatched to the corresponding destination. To resolve neighbor addresses the socket-based network manager is mixed in with a `NeighborhoodResolver`, which provides the necessary endpoint resolution capabilities.

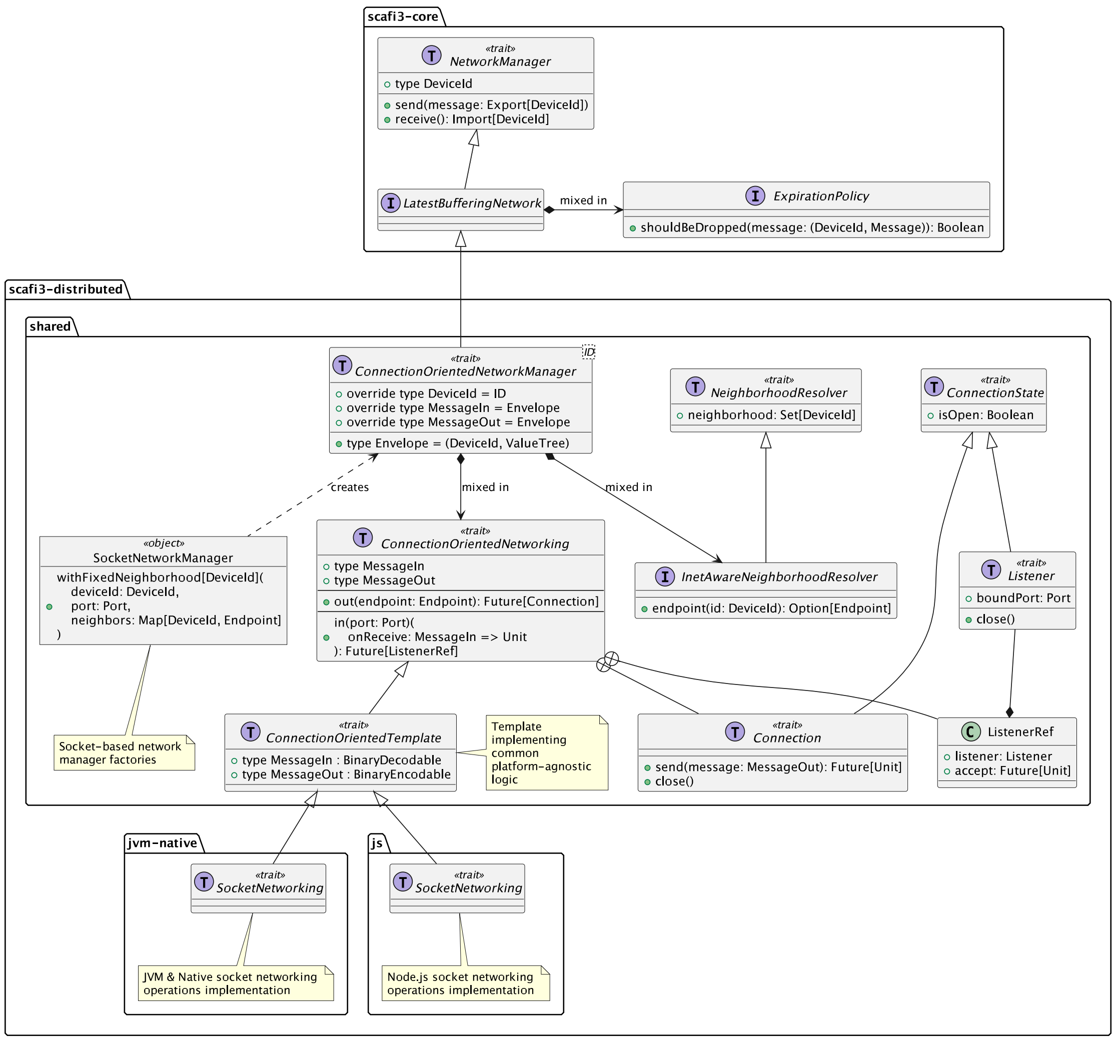

scafi3-core
«trait» NetworkManager
type DeviceId
send(message: Export[DeviceId])
receive(): Import[DeviceId]
LatestBufferingNetwork
mixed in
ExpirationPolicy
shouldBeDropped(message: (DeviceId, Message)): Boolean
scafi3-distributed
shared
«trait» ConnectionOrientedNetworkManager
ID
override type DeviceId = ID
override type MessageIn = Envelope
override type MessageOut = Envelope
type Envelope = (DeviceId, ValueTree)
«trait» NeighborhoodResolver
neighborhood: Set[DeviceId]
«trait» ConnectionState
isOpen: Boolean
creates
mixed in
mixed in
«object» SocketNetworkManager
withFixedNeighborhood[DeviceId](
deviceId: DeviceId,
port: Port,
neighbors: Map[DeviceId, Endpoint]
)
«trait» ConnectionOrientedNetworking
type MessageIn
type MessageOut
out(endpoint: Endpoint): Future[Connection]
in(port: Port)(
onReceive: MessageIn => Unit
): Future[ListenerRef]
InetAwareNeighborhoodResolver
endpoint(id: DeviceId): Option[Endpoint]
«trait» Listener
boundPort: Port
close()
Socket-based network manager factories
«trait» ConnectionOrientedTemplate
type MessageIn : BinaryDecodable
type MessageOut : BinaryEncodable
Template implementing common platform-agnostic logic
«trait» Connection
send(message: MessageOut): Future[Unit]
close()
ListenerRef
listener: Listener
accept: Future[Unit]
jvm-native
«trait» SocketNetworking
JVM & Native socket networking operations implementation
js
«trait» SocketNetworking
Node.js socket networking operations implementation

Figure 5.5: UML class diagram of the socket-based network manager design.

Remote socket-based communication logic are demanded to the `ConnectionOrientedNetworking` trait and their platform-specific implementations. Since the absence of a cross-platform socket library supporting both client and server socket programming, two distinct implementations are provided:

- one for JVM and Native platforms, leveraging the `java.net` package available in the Java standard library. This is possible because Scala Native has reimplemented the `java.net` package to provide socket programming capabilities on native platforms, allowing programmers to use the same API on both JVM and Native platforms without any code divergence, like they would do in pure Scala JVM applications;

- one for JavaScript platforms, leveraging Node.js's `net` module through a Scala.js facade. A portion of this facade is shown in Listing 5.15. It transposes Node.js API definitions into Scala traits and classes using Scala.js types, as well as interoperability annotations such as `@JSImport` to binds Scala.js to a JavaScript module and `js.native` to indicate methods provided by the underlying JavaScript runtime. When linked, these bindings allow Scala.js code to call the underlying Node.js API.

Listing 5.15: Portion of the Scala.js facade for the Node.js `net` module.

```scala
@js.native
@JSImport("net", JSImport.Namespace)
object Net extends js.Object:

  def connect(port: Int, host: String): Socket = js.native

  def createServer(
    connectionListener: js.Function1[Socket, Unit]
  ): Server = js.native
```

Despite targeting all three platforms, the design differs only in platform-specific networking logic, isolated within respective `SocketNetworking` traits. This separation leverages Scala's mixin composition and the template method pattern, defining common logic in abstract trait (`ConnectionOrientedTemplate`) while deferring platform-specific details to specialized implementations.

5.2.2 Serialization binding

Since ScaFi devices exchange pairs of Device Identifiers and their corresponding Value Trees, serialization and deserialization mechanisms must be provided for both data types. The main challenge lies in the encoding and decoding of Value Trees since they can contain arbitrary user-defined values whose types cannot be known beforehand. Consequently, a universal serialization mechanism cannot be provided, as each data type requires its own (de)serialization implementation.

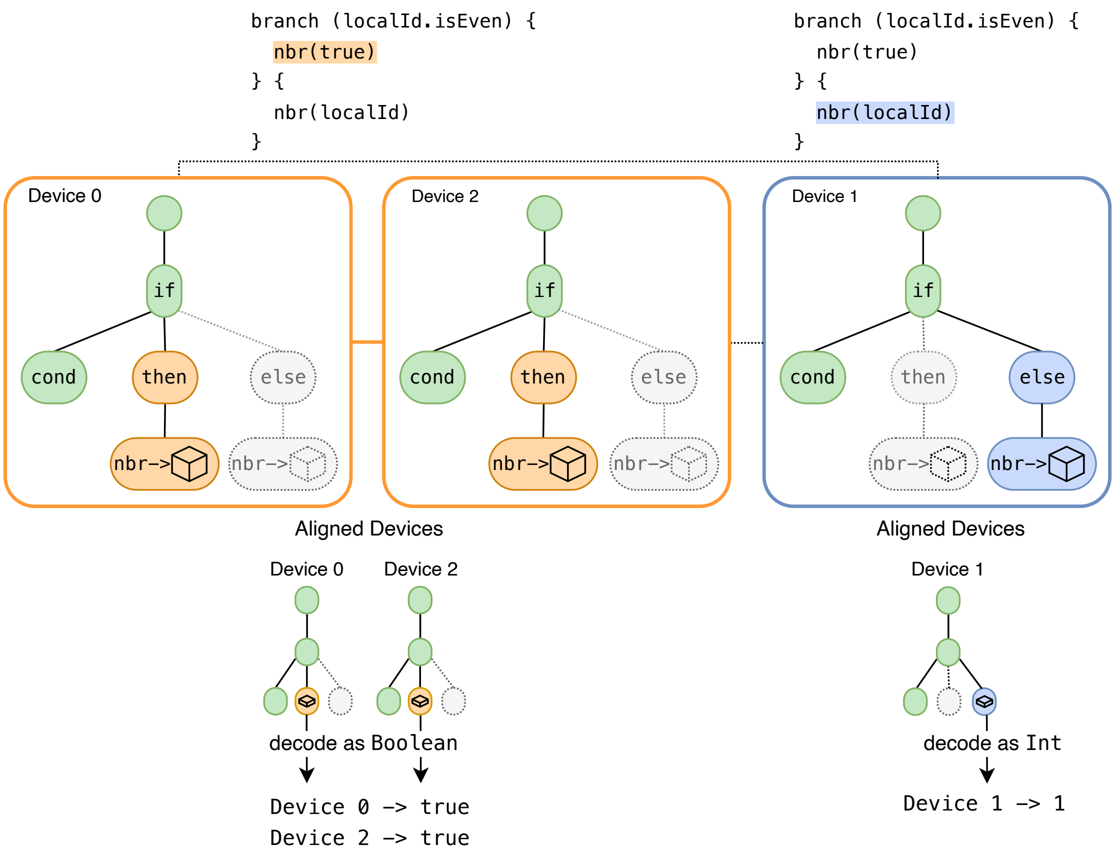
To this end, the following protocol is adopted (exemplified in Figure 5.6):

1. during the execution of the aggregate program, when the Value Tree is built, each value is inserted as encoded using a specific serialization format (e.g., JSON, Byte Array, etc.) provided by the library user. This is possible since, during the evaluation of the aggregate constructs, the type information of the exchanged values is available in that context;

2. at the end of each round of computation, when the Value Tree is sent to

neighbors, it is serialized as a whole structure, given that all values are already encoded;

3. upon receiving a Value Tree from a neighbor device, it is deserialized as a whole structure whose values remain encoded in the specific serialization format;
4. finally, assuming the neighbor device is aligned, when evaluating the aligned aggregate construct, the corresponding neighboring value is extracted by the Value Tree and decoded from the specific serialization format back to the original data type, given that the type information of the expected value is known in that context.



```
branch (localId.isEven) {
  nbr(true)
} {
  nbr(localId)
}
```

```
branch (localId.isEven) {
  nbr(true)
} {
  nbr(localId)
}
```

```
Device 0 -> true
Device 2 -> true
```

```
Device 1 -> 1
```

Figure 5.6: Example of the serialization/deserialization protocol for Value Trees.

This protocol requires library users to provide type-specific encoding and decoding functions for each exchanged data type, with the Scala API ensuring type-safe matching between the provided encoders and decoders and their corresponding types.

This is achieved by leveraging the previously presented codables design, discussed in Section 5.1.1. In the context of ScaFi3 library, the previously discussed design enables another important but peculiar feature: the ability to bypass encoding and decoding operations using in-memory Codable instances. This proves to be particularly useful when the distributed primitives are used only to make the state evolve over rounds without actual neighbor communication, like in the `evolve`, to not force library users to deal with encoding and decoding capabilities for such operations.

5.2.3 ScaFi polyglot API layer

This section focuses on the implementation providing polyglot support for the ScaFi Branching and Field Calculus API, presented in Section 3.1.

As already introduced in Section 3.3, ScaFi libraries API are implemented as functions taking as implicit parameters the language syntaxes to be used. Programmers implement their aggregate program as a context function with required syntaxes as implicit parameters. Library functions can then be invoked without explicitly passing syntax instances, relying on Scala's implicit resolution mechanism. However, this design is not directly portable to other programming languages, as they do not support Scala's equivalent mechanism for implicit parameters and context functions. Two alternatives are possible to overcome this limitation:

- explicitly passing the syntax instances as regular parameters. This is, actually, what Scala internally does when compiling code using implicit parameters:
  ```
  share(lang, initialValue, (f) => { ... })
  ```
- define library functions as methods of classes encapsulating the syntax instances as members. This enables programmers to invoke library methods directly on the syntax instance, like in plain object-oriented programming:
  ```
  lang.share(initialValue, (f) => { ... })
  ```

In pursuit of a more ergonomic API design, the second approach was selected.

Unlike the Scala API, due to the lack of a general mechanism to define a type out of a composition of intersecting interfaces, the `language` instance encapsulates all the required library methods as members. In practice, the aggregate program is turned as a regular function, representable in terms of Portable Types, which takes a `Language` instance as input and returns a value. Like the regular Scala

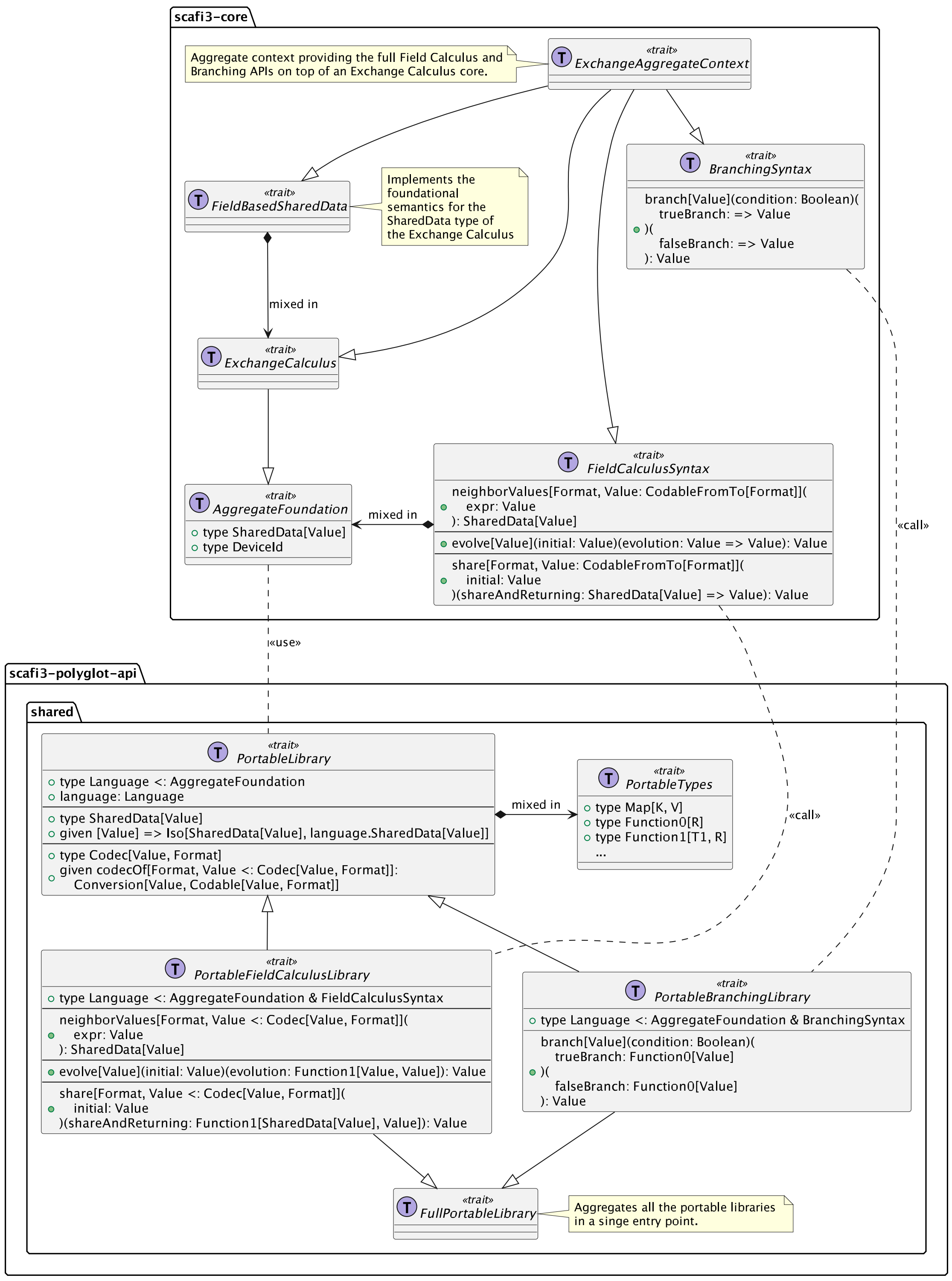

scafi3-core
Aggregate context providing the full Field Calculus and Branching APIs on top of an Exchange Calculus core.
«trait» ExchangeAggregateContext
«trait» FieldBasedSharedData
Implements the foundational semantics for the SharedData type of the Exchange Calculus
«trait» BranchingSyntax
branch[Value](condition: Boolean)(trueBranch: => Value)(falseBranch: => Value): Value
mixed in
«trait» ExchangeCalculus
«trait» FieldCalculusSyntax
neighborValues[Format, Value: CodableFromTo[Format]](expr: Value): SharedData[Value]
evolve[Value](initial: Value)(evolution: Value => Value): Value
share[Format, Value: CodableFromTo[Format]](initial: Value)(shareAndReturning: SharedData[Value] => Value): Value
«trait» AggregateFoundation
type SharedData[Value]
type DeviceId
mixed in
«call»
«use»
scafi3-polyglot-api
shared
«trait» PortableLibrary
type Language <: AggregateFoundation
language: Language
type SharedData[Value]
given [Value] => Iso[SharedData[Value], language.SharedData[Value]]
type Codec[Value, Format]
given codecOf[Format, Value <: Codec[Value, Format]]: Conversion[Value, Codable[Value, Format]]
mixed in
«trait» PortableTypes
type Map[K, V]
type Function0[R]
type Function1[T1, R]
...
«call»
«trait» PortableFieldCalculusLibrary
type Language <: AggregateFoundation & FieldCalculusSyntax
neighborValues[Format, Value <: Codec[Value, Format]](expr: Value): SharedData[Value]
evolve[Value](initial: Value)(evolution: Function1[Value, Value]): Value
share[Format, Value <: Codec[Value, Format]](initial: Value)(shareAndReturning: Function1[SharedData[Value], Value]): Value
«trait» PortableBranchingLibrary
type Language <: AggregateFoundation & BranchingSyntax
branch[Value](condition: Boolean)(trueBranch: Function0[Value])(falseBranch: Function0[Value]): Value
«trait» FullPortableLibrary
Aggregates all the portable libraries in a singe entry point.

Figure 5.7: UML class diagram of the ScaFi polyglot API layer and its relation with the core ScaFi library.

core library, the aggregate program is then fed to the ScaFi Engine, injecting the corresponding `language` instance for the target platform and executing it.

The UML class diagram of the ScaFi polyglot API layer is shown in Figure 5.7. It comprises:

- a `PortableLibrary` trait defining the basic declaration for all portable libraries. It defines an abstract type member `Language` representing the portable aggregate language syntax to which delegate the libraries implementations. This trait also defines a portable definition of `SharedData` and `Codec`, along with their isomorphism with their Scala counterparts;
- a `PortableFieldCalculusLibrary` trait extending `PortableLibrary` and defining the portable Field Calculus API;
- a `PortableBranchingLibrary` trait extending `PortableLibrary` and defining the portable Branching API;
- a `FullPortableLibrary` trait aggregating all the portable libraries in a single entry point. This is the abstract trait corresponding to the `language` instance that programmers will use to implement their aggregate programs, encompassing all the required syntaxes as members.

By having all libraries extend the `PortableLibrary` trait, ensures they all share the same Portable Types, their isomorphisms and are able to access the `Language` instance implementing all the ScaFi core functionalities. Therefore, developing the Portable Libraries simply involves delegating to the corresponding core ScaFi library functions the actual logic, using the provided `language` instance.

Listing 5.16: Fragment of the `FullPortableLibrary` implementation delegating to the core ScaFi library the implementation of API methods.

```scala
trait FullPortableLibrary[Lang <: AggregateFoundation &
    BranchingSyntax & FieldCalculusSyntax](using
     lang: Lang,
) extends PortableCommonLibrary
     with PortableBranchingLibrary
     with PortableFieldCalculusLibrary:
   self: PortableTypes =>

   override type Language = Lang

   override val language: Language = lang

   inline override def evolve[Value](initial: Value)(
         evolution: Function1[Value, Value]
   ): Value = lang.evolve(initial)(evolution)
```

```scala

  inline override def share[
      Format,
      Value <: Codec[Value, Format]
  ](initial: Value)(
    shareAndReturn: Function1[SharedData[Value], Value]
  ): Value =
    lang.share(initial)(shareAndReturn(_))(using codecOf(initial))

  // ...
```

Following the same pattern, a simplified version of the ScaFi Engine API has been implemented to configure and run the distributed aggregate computation.

5.2.4 Limitations

Despite being able to provide polyglot capabilities, three main technical limitations have been identified in the current implementation of this Portable layer that will be discussed hereafter, along with possible future improvements to overcome them.

Scala Native exported annotation gaps. Currently, Scala Native do not allow annotating exported methods in shared source sets. Consequently, they must be re-declared in the Native module—either by hand or by generating them from header files (as discussed in Appendix A)—with appropriate annotation and delegation to the shared implementation. Despite the wrapper logic being unified and not duplicated, this introduces additional boilerplate code for the exported method signatures that could easily be avoided if Scala Native provided a mechanism similar to Scala.js provided annotations.

Delegation also made evident that higher-order exported functions need Portable Type conversions in their declaration context, as the Native compiler generates the type information there required for the conversion to work properly. This can be achieved by inlining the delegation method (see Listing 5.16). However, this prevents applying `@JSExport` annotations, as Scala.js does not support inlining annotated methods. A possible workaround is to separate functions declaration, with annotations, from their inlined implementations.

Nevertheless, this is more a collateral effect of the lack of provided annotations in shared code rather than a fundamental limitation of the polyglot API layer and it would be solved if Scala Native provided such mechanism.

Scala Native missing method exports.

Another limitation of Scala Native is the lack of direct support for exporting classes' methods. This can be worked around by exporting a C struct representing the class and function pointers for each method, as shown in Listing 5.17. This requires to save in a statically reachable location, e.g., a private object, the instance of the class to be able to invoke its methods from the exported functions (see Listing 5.18). Pertaining to ScaFi, this is not a significant limitation since the Aggregate Context, representing the language instance, is instantiated afresh at each round of computation.

Listing 5.17: Aggregate Library API exported as a C struct with function pointers.

```c
typedef struct AggregateLibrary {
    FieldBasedSharedData Field; // Field factory accessor

    DeviceId (*local_id)(void);

    const void* (*branch)(bool condition, const void* (*
       true_branch)(void), const void* (*false_branch)(void));

    const void* (*evolve)(const void* initial, const void* (*
       evolution)(const void*));

    const BinaryCodable* (*share)(const BinaryCodable* initial,
       const BinaryCodable* (*f)(const Field* field));

    const Field* (*neighbor_values)(const BinaryCodable* value);
} AggregateLibrary;
```

Listing 5.18: Adopted workaround for exporting class methods in Scala Native.

```scala
// Invoked at each round start when creating Aggregate Context
def asNative: Ptr[CAggregateLibrary] =
  libraryRef.set(this) // statically reachable library instance
  val lib = allocateTracking[CAggregateLibrary]
  // populate function pointers
  (!lib).evolve =
    (init: Ptr[Byte], evol: Function1[Ptr[Byte], Ptr[Byte]]) =>
      libraryRef.get().evolve(initial)(evolution)
  (!lib).neighbor_values = (value: Ptr[CBinaryCodable]) =>
     libraryRef.get().neighborValues(value)
  // ...
  lib
```

Java Abstract Type Members erasure.

The latter limitation is present on the Java side, rooted in the fact that Abstract Type Members have no Java equivalent——they are erased to `java.lang.Object` in JVM bytecode, appearing to Java library users simply as `Object`. This prevents Java library from correctly resolve Portable Types in Portable Libraries, unless their signatures are overridden in the platform-specific implementations where actual JVM types instantiations are available. Despite this task is largely mechanical and facilitated by IDEs auto-completion features, it introduces additional boilerplate code. A possible future improvement, eliminating this limitation, is the automatic generation of Java override method with `super` delegation thanks to a custom tiny pre-processing plugin integrated into the build process.

6 Validation

This chapter describes how the implemented system has been validated to ensure it meets the specified requirements and functions correctly in various scenarios. The validation process includes unit testing, integration testing, and a demonstration of the system's capabilities.

6.1 Testing

Testing is a crucial aspect of the validation process of a software system, ensuring all components function as intended and interact correctly. In the following, the testing strategies employed to validate the cross-platform and polyglot capabilities of the library are described. Although demonstrated here using the ScaFi library, these apply to any library or application employing the presented cross-platform and polyglot architecture.

6.1.1 Unit testing

All developed components of the library have been subjected to automated unit testing to ensure the correctness of individual code units in all three supported targets (JVM, JavaScript, and Native). ScalaTest[1] has been used as the reference testing framework for writing idiomatic Scala tests that can run seamlessly across all targets.

Beyond cross-target testing, thanks to *Continuous Integration* (CI) pipelines, tests are automatically executed, upon each code push on the repository, on all major operating systems (Linux, Windows, and macOS). This ensures that the library maintains its functionality and reliability across different environments and platforms as development progresses and identify any issues early in the development cycle.

[1]`https://www.scalatest.org`

6.1.2 Integration testing

In addition to unit testing, integration testing has been performed to ensure all exported language APIs work as intended. This involves compiling and running example applications that utilize the ScaFi library in each of the target languages.

Before introducing how integration tests are structured and implemented, it is important to highlight the requirements that these aim to validate:

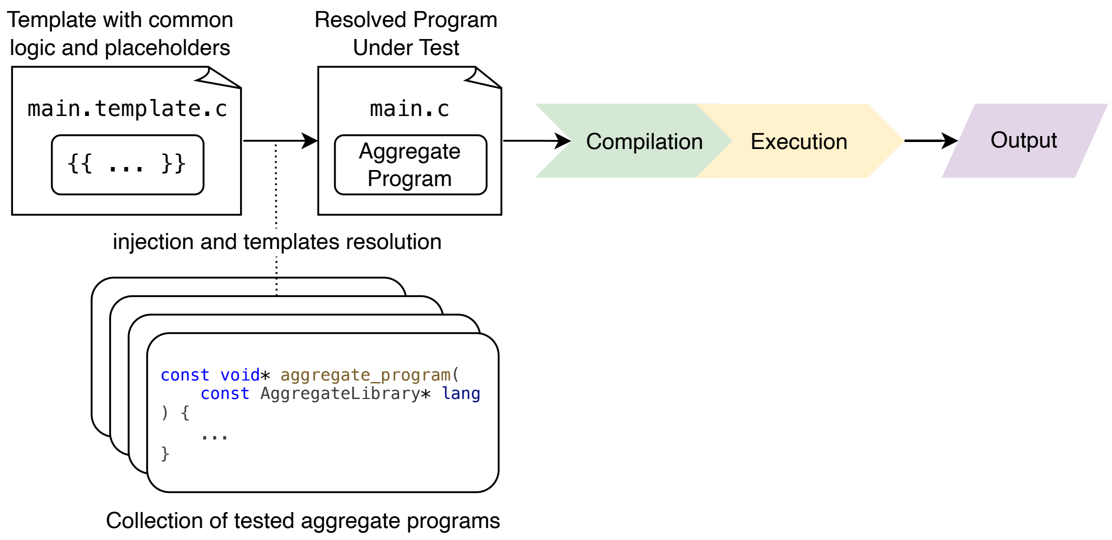
- they must operate against the exported APIs of the Portable libraries and be implemented in each target language rather than in Scala. This ensures tests validate the actual API surface exposed to end-users and not the internal Scala side. This prevents situations where subtle bugs may infiltrate between the boundary of the Scala implementation and the target language API;
- they must run in the corresponding language target environment (e.g., Node.js for JavaScript, bare metal for Native) to accurately reflect real-world usage scenarios. Since the library is designed to be cross-platform, it is expected that developer hosts are equipped with the three target environments toolchains;
- they must cover both core and distributed functionalities of the library to ensure comprehensive validation of its capabilities;
- they should minimize repetition of test setup and configuration, e.g., network and runtime engine initialization, to enhance maintainability and readability of the test code. Developers should focus only on the specific logic being tested.

To meet the above requirements, a small test framework has been developed to facilitate the creation and execution of integration tests. The framework operates as follows:

- each program under test is implemented in one or more sources and saved in a specific folder in the integration test module resources;
- only one source code file for each target language is written, containing the test setup and configuration logic that can be shared across all programs under test. Test-specific parts (e.g., in ScaFi, the configuration of device identifier, port number, and neighborhood) are parameterized and populated at runtime through a simple templating mechanism (`{{...}}`). Essentially, this source file act as a test *template* to be reused across all programs under test for the same target language;
- a Scala Test component is responsible for integrating and coordinating the execution of the tests, by performing the following steps for each program:

1. it creates a temporary folder where the test template source file is copied, along with the necessary build configuration files;
2. it injects into the test template the program under test and its specific parameters in the appropriate template placeholders, generating a complete test source file;
3. it builds and runs the generated test source file in a distributed test environment, capturing its output;
4. it verifies the captured output against the expected results to determine if the test passes or fails.

The overall workflow of the integration testing process is illustrated in Figure 6.1.


Template with common logic and placeholders
main.template.c
{{ ... }}
Resolved Program Under Test
main.c
Aggregate Program
Compilation
Execution
Output
injection and templates resolution
Collection of tested aggregate programs

```c
const void* aggregate_program(
    const AggregateLibrary* lang
) {
    ...
}
```

Figure 6.1: Integration testing workflow.

Listing 6.1 and Listing 6.2 shows, respectively, the test template for JavaScript integration tests and an example of an aggregate program testing the `branch` semantic.

Listing 6.1: Test template for JavaScript integration tests.

```javascript
const { Runtime } = await import("./main.mjs");

const deviceId = {{ deviceId }};
const port = {{ port }};
const neighbors = new Map({{ neighbors }});

let lastResult = null;
let iterations = 10;
await Runtime.engine(deviceId, port, neighbors, lang =>
    aggregateProgram(lang), async result => {
    lastResult = result;
    await sleep(1_000);
    return iterations-- > 0;
});

console.log(lastResult.toString());
```

Listing 6.2: Aggregate program under test for restricted exchange integration test.

```javascript
function aggregateProgram(lang) {
    return lang.branch(
        lang.localId % 2 === 0,
        () => lang.neighborValues(1),
        () => lang.neighborValues(0),
    );
}
```

Scala Test side, the integration component is implemented in the `PlatformTest` trait, shown in Listing 6.3. It leverages *Refinement Types*[2] [JV20]and a small DSL to allow users to define template resolution and test execution in a concise and type-safe manner, as showcased in Listing 6.4 (lines 19-22). This example demonstrates the creation of a Von Neumann grid of devices executing a given aggregate program with template resolution parameters that map template patterns to their corresponding values.

[2]`https://iltotore.github.io/iron/docs/`

Listing 6.3: Root component for integration testing across languages.

```scala
object PlatformTest:

  type ProgramOutput = String

  type Pattern = String :| (StartWith["{{"] & EndWith["}}"])

  type Substitution = (Pattern, String)

  class SubstitutionBuilder:
    private var _substitutions = Set.empty[Substitution]
    private[PlatformTest] def add(key: Pattern, value: String) =
      _substitutions += (key, value)
    def substitutions: Set[Substitution] =
      _substitutions.view.toSet

  extension (pattern: Pattern)
    infix inline def ->(value: String)(using builder: SubstitutionBuilder): Unit =
      builder.add(pattern, value)

trait PlatformTest extends Matchers with OS with FileSystem:

  // Run a test for the specified program
  def testProgram(testName: String)(
      addSubstitutions: SubstitutionBuilder ?=> Unit
  ): Try[ProgramOutput] =
    given builder: SubstitutionBuilder = SubstitutionBuilder()
    addSubstitutions
    for
      workingDir <- createTempDirectory(testName)
      _ <- resolveTemplates(testName, workingDir, builder.substitutions)
      _ <- compile(workingDir)
      out <- run(workingDir)
      _ = delete(workingDir)
    yield out.trim()

  // ...
```

Listing 6.4: Cross-language integration test component.

```scala
trait CrossLanguageTest:

  def neighborsDiscoveryTest(): Unit =
    "Neighbors discovery program" should "spread local values to neighborhood" in:
      sequence:
        aggregateResult("neighbors-discovery", rows = 2, cols = 2)
      .futureValue should contain theSameElementsAs Seq(
        0 -> fieldRepr(default = 0, neighbors = Map(0 -> 0, 1 -> 1, 2 -> 2)),
        1 -> fieldRepr(default = 1, neighbors = Map(0 -> 0, 1 -> 1, 3 -> 3)),
        2 -> fieldRepr(default = 2, neighbors = Map(0 -> 0, 2 -> 2, 3 -> 3)),
        3 -> fieldRepr(default = 3, neighbors = Map(1 -> 1, 2 -> 2, 3 -> 3)),
      )

  def aggregateResult(testName: String, rows: Int, cols: Int)
      : Seq[Future[(Int, ProgramOutput)]] =
    val ports = FreePortFinder.get(rows * cols)
    vonNeumannGrid(rows, cols): (id, neighbors) =>
      Future:
        testProgram(testName):
          "{{ deviceId }}" -> id.toString
          "{{ port }}" -> ports(id).toString
          "{{ neighbors }}" -> neighborsAsCode(id, neighbors, ports)
      .map(res => id -> res.getOrElse(fail(s"Test '$testName' failed on device '
          $id': ${res.failed.get.getMessage}")))

  def neighborsAsCode(id: ID, neighbors: Set[ID], ports: Seq[Port]): String

  def fieldRepr[Value](default: Value, neighbors: Map[ID, Value]): String

trait JSTests extends PlatformTest:

  override def compile(workingDir: Path): Try[String] = Success("JS does not
      require compilation")

  override def run(workingDir: Path): Try[String] =
    execute(workingDir, (if isWindows then "npm.cmd" else "npm") :: "start" :: "--
        silent" :: Nil)
  // ...
```

6.2 Demonstration

The final validation of the cross-platform and polyglot capabilities of the ScaFi library introduced in this work is demonstrated through a simple real-world distributed application implementing a gradient diffusion algorithm by hop count.

6.2.1 Application

At the application level, the program is similar to the one presented in Figure 3.3, where the sensed distance from each neighbor is always considered equal to 1. The result is a gradient field where each device value represents the minimum number of hops required to reach the source device.

The following presents four versions of the same aggregate program implementing the gradient diffusion algorithm, one for each supported target language, showcasing the polyglot capabilities of the ScaFi library. Each of these, have been programmed using only the respective target language API and relying on the Protobuf schema presented in Listing 6.5 to represent and exchange distance values between neighbors.

Listing 6.5: Protobuf schema for the gradient example.

```
syntax = "proto3";

message Distance {
  float value = 1;
}
```

The Scala version is shown in Listing 6.6. It is the most idiomatic and concise and leverages ScalaPB generated case classes for representing the `Distance` message type in conjunction with the `BinaryCodable` presented in Listing 5.3.

Listing 6.6: Scala implementation.

```
given Ordering[Distance] = (x, y) => x.value.compare(y.value)

given UpperBounded[Distance] = new UpperBounded[Distance]:
  override def upperBound: Distance = Distance(Float.MaxValue)

def aggregateProgram(using Lang): Distance =
  share(Distance(Float.MaxValue)): nvalues =>
    val minDistance = nvalues.withoutSelf.min.value + 1
    if isSource then Distance(0) else Distance(minDistance)
```

The JavaScript version, presented in Listing 6.7, uses the generated Protobuf functions generated by *protobuf.js*[3] to create and manipulate `Distance` messages. This relies on the default codable implementation provided by the Polyglot ScaFi layer for Protobuf messages.

[3]https://github.com/protobufjs/protobuf.js/

Listing 6.7: JavaScript implementation.

```
function aggregateProgram(lang) {
    return lang.share(distanceOf(Infinity), (nvalues) => {
        const minDistance = nvalues.withoutSelf
            .map((dist) => dist.value)
            .reduce((min, d) => Math.min(min, d), Infinity) + 1.0;
        return isSource ? distanceOf(0) : distanceOf(minDistance);
    });
}

function distanceOf(distance) {
    return proto.Distance.create({ value: distance });
}
```

In Listing 6.8 is shown the C implementation of the gradient algorithm. Not surprisingly, this is the most verbose implementation due to the low-level nature of the C language, which pushes developers to implement solutions imperatively rather than declaratively. Moreover, due to the lack of generics in C, they are represented as `void*` pointers, which require explicit casting to the appropriate type when accessing their fields. Despite this verbosity, note the automatic memory management handled by the polyglot layer. Concerning message encoding and decoding, the C implementation uses *Protobuf-C*[4] and relies on `proto_of`, a ready-to-use function that simplifies the creation of codable instances from Protobuf-C-generated structures, similarly to what has been presented for JavaScript. Unlike JavaScript, however, C does not support runtime type inspection, hence the need to explicitly use a helper function to create these instances.

[4]`https://github.com/protobuf-c/protobuf-c`

Listing 6.8: C implementation.

```c
const BinaryCodable* distance_of(float value) {
    Distance* d = malloc(sizeof(Distance));
    if (!d) return NULL;
    distance__init(d);
    d->value = value;
    return (const BinaryCodable*) proto_of(d, distance);
}

float min(Array* nvalues) {
    float min_distance = FLT_MAX;
    for (size_t i = 0; i < nvalues->size; i++) {
        const ProtobufValue* d = nvalues->items[i];
        float dist = ((const Distance*)d->message)->value;
        min_distance = dist < min_distance ? dist : min_distance;
    }
    return min_distance;
}

const void* aggregate_program(const AggregateLibrary* lang) {
    return lang->share(
        distance_of(FLT_MAX),
        fn(const BinaryCodable*, (const Field* f), {
            float min_distance = min(without_self(f)) + 1.0;
            return is_source ? distance_of(0.0) : distance_of(
                min_distance);
        })
    );
}
```

Finally, in Listing 6.9 is presented the TypeScript implementation of the gradient algorithm. In this case, differently from JavaScript, a different library for Protobuf handling has been used, *protobuf-es*[5], which generates idiomatic TypeScript classes for each Protobuf message type. This showcase how the polyglot layer can seamlessly interoperate with different serialization libraries and formats. Since using a custom class for representing the `Distance` message type, it is necessary to implement the `Codec` interface to provide encoding and decoding capabilities to it and allow the polyglot layer to handle message serialization and deserialization during network communication.

[5]https://github.com/bufbuild/protobuf-es

Listing 6.9: TypeScript implementation.

```typescript
function aggregateProgram(lang: Language) {
    return lang.share(new Distance(Infinity), (nvalues) => {
        const minDistance = nvalues.withoutSelf
            .map((dist) => dist.value)
            .reduce((min, d) => Math.min(min, d), Infinity) + 1.0;
        return isSource ? new Distance(0) : new Distance(
            minDistance);
    });
}

class Distance implements Codec<Distance, Uint8Array> {
    private instance: ProtoDistance;

    constructor(value: number) {
        this.instance = create(DistanceSchema, { value });
    }

    get codable(): Codable<Distance, Uint8Array> {
        return Distance.codable;
    }

    get value(): number {
        return this.instance.value;
    }

    static codable: Codable<Distance, Uint8Array> = {
        typeName: "Distance",
        encode: (sensor) => toBinary(DistanceSchema, sensor.
            instance),
        decode: (bytes) => {
            const decodedDistance = fromBinary(DistanceSchema,
                bytes);
            return new Distance(decodedDistance.value);
        },
    };
}
```

6.2.2 Deployment

Deployment wise, the application is executed on a heterogeneous grid of devices, running on different platforms and hardware architectures. Specifically, a network composed of six Raspberry Pi single-board computers has been created with a Von

Neumann topology, where each device is connected via sockets to its four orthogonal neighbors (up, down, left, right) as shown in Figure 6.2. Each device executes an instance of the gradient aggregate program using a distinct platform-language combination, as summarized in Figure 6.2. Other than demonstrating the polyglot capabilities of the ScaFi library, this deployment showcases its cross-platform capabilities by cross-compiling and running the Scala version of the application both on bare metal (Scala Native) and on JavaScript (Scala.js).

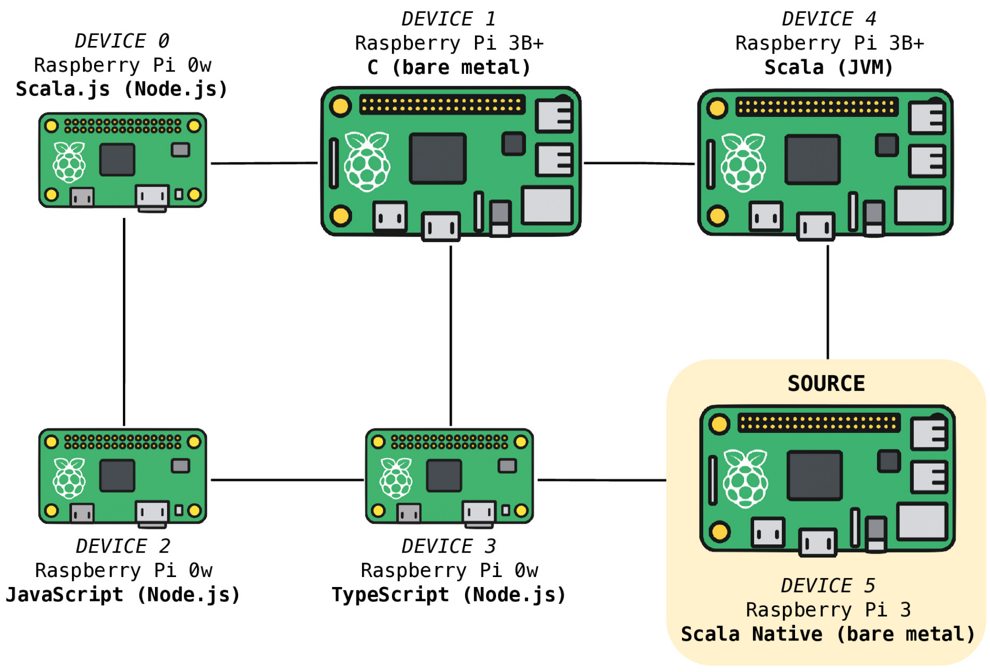


Figure 6.2: Deployment topology for the gradient diffusion demonstration. The device in the bottom-right corner (Device 5) is the one acting as the source from which all other devices compute their distance.

The Raspberry Pi models and specifications used in the deployment are detailed in Table 6.1.

Model	Cores	RAM	Architecture	Year
Raspberry Pi Zero W	1	512 MB	ARMv6 (32-bit)	2017
Raspberry Pi 3	4	1 GB	ARMv8 (64-bit)	2016
Raspberry Pi 3B+	4	1 GB	ARMv8 (64-bit)	2018

Table 6.1: Technical specifications of Raspberry Pi models employed in the deployment.

6.2.3 Qualitative analysis

The node acting as the source device is the Device 5 (Scala Native on Raspberry Pi 3), while all other devices compute their hop-count distance from it. To make evident the correct functioning of the program each Raspberry Pi is equipped with a display showing the current computed distance value from the source device, updated in real-time as the aggregate computation proceeds. Initially, all devices are up and running except for the Device 3 and the source device (Figure 6.3a). Once the source device connects, distance values propagate from the source to all other devices until they stabilize at the correct minimal hop-count distance values, as shown in Figure 6.3b. If Device 2 goes offline for any reason, the network partitions into two subnetworks after a brief transient period: one containing the source device and one without, where distance values become infinite (see Figure 6.3c). Upon reconnection of all devices, the distance values update to reflect the new network topology, demonstrating the adaptability of aggregate computation to dynamic network changes (see Figure 6.3d).

6.2.4 Performance Evaluation

While the primary focus of this work was on achieving cross-platform and polyglot capabilities for a prototypal version of the ScaFi library without focusing on optimization and performances, a preliminary evaluation has been conducted to understand the different performances obtained by the various target platforms and the overheads introduced by the polyglot layer.

Performance was evaluated in terms of memory usage across 2,500 rounds—1 executed every 250ms, totaling over 10 minutes—of the gradient diffusion application presented in the previous section, running on a Raspberry Pi 3B+ device with three neighbors. The performance measurements were conducted for each target platform and language combination—C and Scala Native for bare metal environments, Scala JVM for the JVM environment, and Scala.js and JavaScript for the JavaScript environment—with 10 runs executed per configuration to obtain statistically significant results. Concerning the Native target, the Boehm[6] garbage collector with default configuration is used. Results are presented in Figure 6.4.

As expected, the application running on bare metal (C and cross-compiled Scala Native) exhibits the lowest memory usage, nearly 4x lower than the JVM version. The Scala Native version shows better performance compared to the C version. The difference can be attributed to the overhead imposed by the polyglot layer for the necessary conversions between C and Scala data structures, which

[6] `https://www.hboehm.info/gc/`

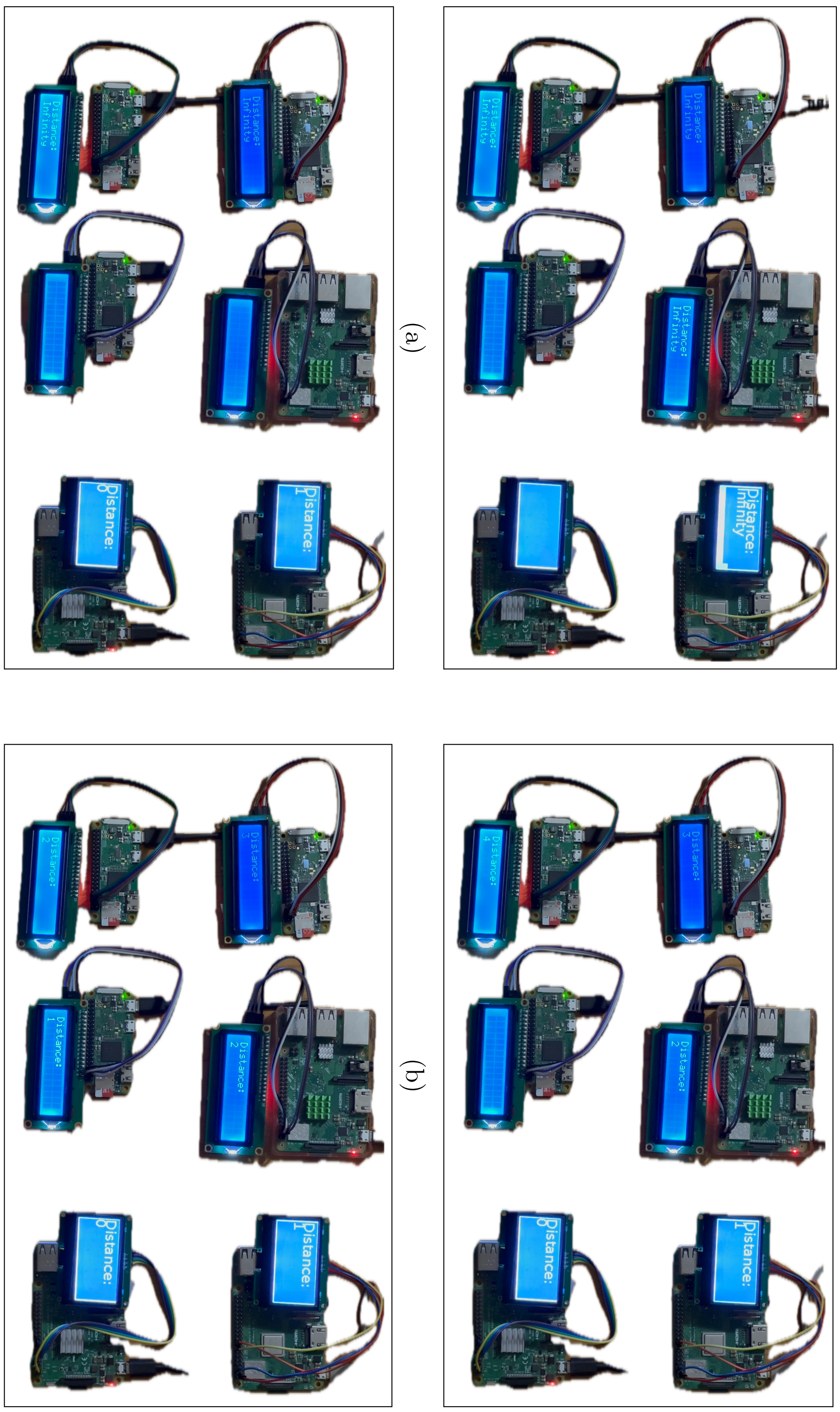
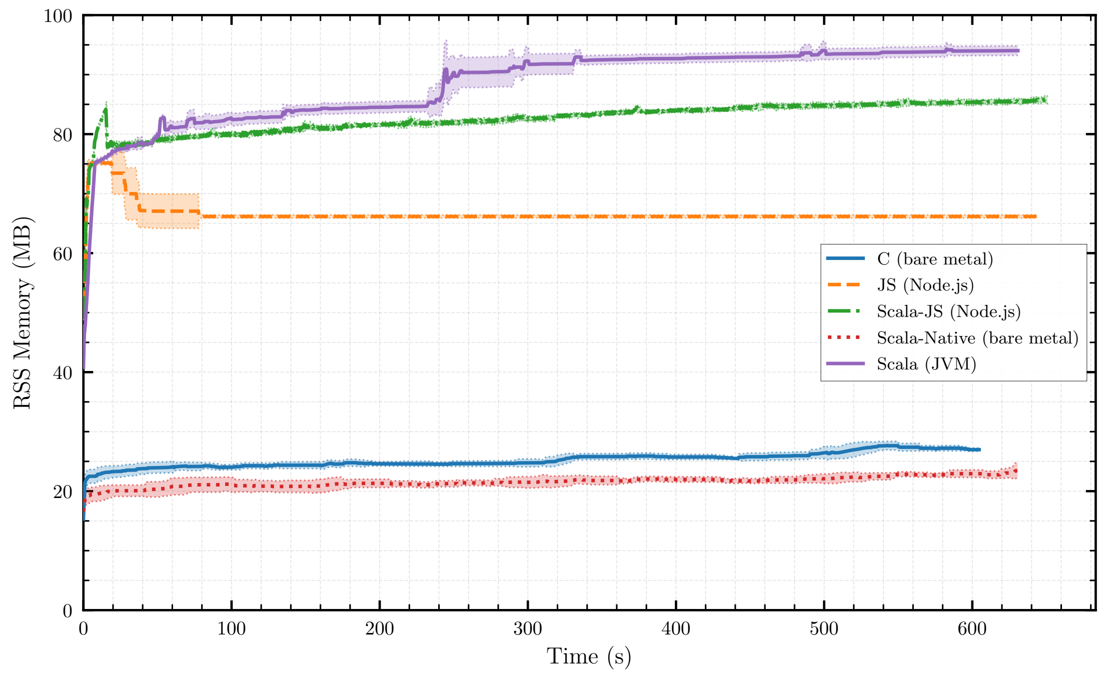
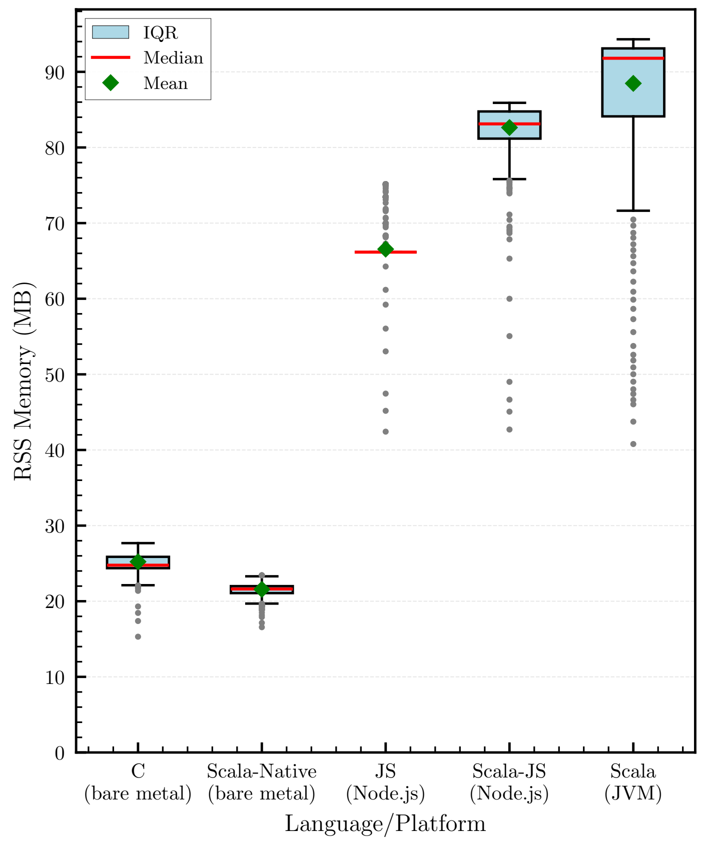


Figure 6.3: Gradient diffusion demonstration showing network topology changes and distance value updates.



(a) Mean memory consumption with standard deviation bands over execution time.



(b) Mean memory consumption per language/platform.

Figure 6.4: Memory consumption analysis for the gradient diffusion application across different platform and language combinations over 2,500 execution rounds (≈ 10 minutes). Measurements are conducted across 10 runs per configuration.

are not necessary in the Scala Native version, where data structures are written as plain Scala and compiled to machine code directly. The JS-based implementations show higher memory usage compared to Native but lower than the JVM version, with the plain JS version surprisingly outperforming the cross-compiled Scala.js version despite the expected overhead introduced by the polyglot layer.

These performance evaluations revealed that one aspect with potential for improvement, expected to moderately impact JVM and Native performance, is the network layer. Initially, the network was implemented with a one-thread-per-client approach on both JVM and Native platforms, which limits scalability and introduces performance overhead. This implementation approach was chosen initially because the asynchronous socket API from Java's standard library is not available in Native. Implementing it would require using native-level primitives and bindings using C Posix library, which would have demanded significantly more development time and effort. However, this is surely needed for future work to enhance the library's performance and scalability. Moreover, on JVM platform, the use of Virtual Threads-based executor[7] could enhance scalability and performance of the network layer by allowing many concurrent connections with minimal resource consumption.

[7]`https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html`

7 Conclusions and Future Works

This work explored the possibility of creating a general framework for developing Scala cross-platform and polyglot libraries. The main goal was to conceive an architecture to run Scala-based software products on heterogeneous computational platforms and enabling programmers to leverage it from various programming languages—chosen based on business goals, preferences, or ecosystem features that provide competitive advantages for their specific use cases. All of this while programming a unified version of the software product in Scala, one of the main leading, powerful and expressive programming language to build reliable, scalable and distributed libraries and applications.

First, current frameworks and languages for multi-platform and polyglot development were analyzed. Then, the Scala ecosystem and capabilities were explored in terms of cross-compilation and languages interoperability. Driven by the desire to provide a solution completely grounded on Scala language features, and in light of present limitations of the Scala cross-platform capabilities, a general architecture has been conceived. To test its feasibility, this architecture was instantiated within the ScaFi3 implementation in the Aggregate Computing research field, revealing main challenges, solution spaces, and improvement areas that must be addressed for the approach to be fully effective. Finally, to validate the work, a real-world demonstration on top of heterogeneous devices and platforms has been conducted to fully understand its capabilities and effectiveness.

Preliminary results show this architecture provides cross-platform and polyglot capabilities to Scala software libraries, allowing developers to write the software product once, using a cross-platform approach, and export it to different programming languages through a minimal polyglot abstraction layer. This allows to maximize code reuse and minimize maintenance effort.

However, this work is only a first step towards a fully-fledged framework for building cross-platform and polyglot Scala libraries. Following the limitations identified in Section 5.2.4 and the demonstration and validation results, several areas for future work have been identified. These can be categorized into improvements to the current ScaFi3 library (L), and long-term enhancements to the overall architecture (A) and tooling (T), which would benefit not only the ScaFi3 library but also other Scala-based projects aiming for cross-platform and polyglot capabilities.

- A+T. Allow Native `@exported` methods to be used both in shared and platform-specific code. This would avoid redefining signatures in platform-specific source sets just to be able to annotate and export them.
- T. Develop a plugin to automatically generate header files from Scala Native `@exported` methods. This would streamline the process of creating native bindings without manually writing C header files.
- T. Enhance the polyglot layer to automatically generate, using an appropriate automation tool, Java override method declarations, overcoming the Abstract Type Members erasure issue.
- A+T+L. Create a tool to automatically generate Python bindings from Scala Native `@exported` methods and validate it by generating the Python bindings for the ScaFi3 Native API. If successful, this would boost the adoption of the presented architecture.
- A+T+L. Improve the type-safety of the Native API by providing a C++ or Rust facade over the current C implementation, leveraging templates for compile-time type checking and eliminating unsafe casts. This limitation affects all libraries using Scala Native, which targets C-based interoperability. Addressing it would require investigating a broader solution, possibly necessitating the creation of a dedicated automated tool for generating such facades.
- L. Support an asynchronous networking module for the Scala Native platform allowing to avoid the current one-thread-per-connection approach, which limits scalability when many neighbors are present. Moreover, more advanced network protocols could be supported, such as MQTT.

Finally, the whole architecture would undoubtedly benefit from the Scala Native 2.0 interoperability proposal[1], including first-class export annotations, even though since the long-standing nature of the proposal appears unlikely to materialize in the foreseeable future.

[1]`https://github.com/scala-native/scala-native/issues/897`

Bibliography

[ARO14] Nada Amin, Tiark Rompf, and Martin Odersky. "Foundations of path-dependent types". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. SPLASH '14. ACM, Oct. 2014, pp. 233–249. DOI: 10.1145/2660193.2660216. URL: http://dx.doi.org/10.1145/2660193.2660216.

[Aud+19] Giorgio Audrito et al. "A Higher-Order Calculus of Computational Fields". In: *ACM Transactions on Computational Logic* 20.1 (Jan. 2019), pp. 1–55. ISSN: 1557-945X. DOI: 10.1145/3285956. URL: http://dx.doi.org/10.1145/3285956.

[Aud+24] Giorgio Audrito et al. "The eXchange Calculus (XC): A functional programming language design for distributed collective systems". In: *Journal of Systems and Software* 210 (Apr. 2024), p. 111976. ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.111976. URL: http://dx.doi.org/10.1016/j.jss.2024.111976.

[Aud20] Giorgio Audrito. "FCPP: an efficient and extensible Field Calculus framework". In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, Aug. 2020, pp. 153–159. DOI: 10.1109/acsos49614.2020.00037. URL: http://dx.doi.org/10.1109/ACSOS49614.2020.00037.

[Bea03] D.M. Beazley. "Automated scientific software scripting with SWIG". In: *Future Generation Computer Systems* 19.5 (July 2003), pp. 599–609. ISSN: 0167-739X. DOI: 10.1016/s0167-739x(02)00171-1. URL: http://dx.doi.org/10.1016/S0167-739X(02)00171-1.

[Ber+24] Francesco Bertolotti et al. "When the dragons defeat the knight: Basilisk an architectural pattern for platform and language independent development". In: *Journal of Systems and Software* 215 (Sept. 2024), p. 112088. ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.112088. URL: http://dx.doi.org/10.1016/j.jss.2024.112088.

[BL21] J.Z. Blanco and D. Lucrédio. “A holistic approach for cross-platform software development”. In: *Journal of Systems and Software* 179 (Sept. 2021), p. 110985. ISSN: 0164-1212. DOI: `10.1016/j.jss.2021.110985`. URL: `http://dx.doi.org/10.1016/j.jss.2021.110985`.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. “Aggregate Programming for the Internet of Things”. In: *Computer* 48.9 (Sept. 2015), pp. 22–30. ISSN: 0018-9162. DOI: `10.1109/mc.2015.261`. URL: `http://dx.doi.org/10.1109/MC.2015.261`.

[Cas23] Roberto Casadei. “Macroprogramming: Concepts, State of the Art, and Opportunities of Macroscopic Behaviour Modelling”. In: *ACM Computing Surveys* 55.13s (July 2023), pp. 1–37. ISSN: 1557-7341. DOI: `10.1145/3579353`. URL: `http://dx.doi.org/10.1145/3579353`.

[Coc05] Alistair Cockburn. *The Hexagonal (Ports & Adapters) Architecture.* Technical Report 2005.02. Alistair Cockburn, 2005. URL: `https://alistair.cockburn.us/hexagonal-architecture` (visited on 11/10/2025).

[Cor24] Angela Cortecchia. “A Kotlin multi-platform implementation of aggregate computing based on XC”. Master’s thesis. Università di Bologna, 2024. URL: `https://amslaurea.unibo.it/id/eprint/31080`.

[CV16] Roberto Casadei and Mirko Viroli. “Towards Aggregate Programming in Scala”. In: *First Workshop on Programming Models and Languages for Distributed Computing.* PMLDC ’16. ACM, July 2016, pp. 1–7. DOI: `10.1145/2957319.2957372`. URL: `http://dx.doi.org/10.1145/2957319.2957372`.

[Del24] Luca Deluigi. “Design and implementation of a scalable domain specific language foundation for ScaFi with Scala 3”. Master’s thesis. Università di Bologna, 2024. URL: `https://amslaurea.unibo.it/id/eprint/31164`.

[Doe18] Sébastien Doeraene. “Cross-Platform Language Design”. PhD thesis. École Polytechnique Fédérale de Lausanne, 2018. URL: `https://lampwww.epfl.ch/~doeraene/thesis/doeraene-thesis-2018-cross-platform-language-design.pdf`.

[El-+17] Wafaa S. El-Kassas et al. “Taxonomy of Cross-Platform Mobile Applications Development Approaches”. In: *Ain Shams Engineering Journal* 8.2 (June 2017), pp. 163–190. ISSN: 2090-4479. DOI: `10.1016/j.asej.2015.08.004`. URL: `http://dx.doi.org/10.1016/j.asej.2015.08.004`.

[Gri+18] Matthias Grimmer et al. "Cross-language interoperability in a multi-language runtime". In: *ACM Transactions on Programming Languages and Systems* 40.2 (May 2018), pp. 1–43. DOI: `10.1145/3201898`.

[JV20] Ranjit Jhala and Niki Vazou. *Refinement Types: A Tutorial.* 2020. DOI: `10.48550/ARXIV.2010.07763`. URL: `https://arxiv.org/abs/2010.07763`.

[LA] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, pp. 75–86. DOI: `10.1109/cgo.2004.1281665`. URL: `http://dx.doi.org/10.1109/CGO.2004.1281665`.

[Mic24] Leonardo Micelli. "Design and development of a Rust-based execution platform for Aggregate Computing". Master's thesis. Università di Bologna, 2024. URL: `https://amslaurea.unibo.it/id/eprint/31133`.

[Moo] James D. Mooney. "Developing Portable Software". In: *Information Technology.* Kluwer Academic Publishers, pp. 55–84. ISBN: 1402081588. DOI: `10.1007/1-4020-8159-6_3`. URL: `http://dx.doi.org/10.1007/1-4020-8159-6_3`.

[Ode+17] Martin Odersky et al. "Simplicitly: foundations and applications of implicit function types". In: *Proceedings of the ACM on Programming Languages* 2.POPL (Dec. 2017), pp. 1–29. ISSN: 2475-1421. DOI: `10.1145/3158130`. URL: `http://dx.doi.org/10.1145/3158130`.

[OMO10] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. "Type classes as objects and implicits". In: *ACM SIGPLAN Notices* 45.10 (Oct. 2010), pp. 341–360. ISSN: 1558-1160. DOI: `10.1145/1932682.1869489`. URL: `http://dx.doi.org/10.1145/1932682.1869489`.

[OZ05] Martin Odersky and Matthias Zenger. "Scalable component abstractions". In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* OOPSLA05. ACM, Oct. 2005, pp. 41–57. DOI: `10.1145/1094811.1094815`. URL: `http://dx.doi.org/10.1145/1094811.1094815`.

[PBV17] Danilo Pianini, Jacob Beal, and Mirko Viroli. "Practical Aggregate Programming with Protelis". In: *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W).*

IEEE, Sept. 2017, pp. 391–392. DOI: 10.1109/fas-w.2017.186. URL: http://dx.doi.org/10.1109/FAS-W.2017.186.

[PMV13] D Pianini, S Montagna, and M Viroli. "Chemical-oriented simulation of computational systems with ALCHEMIST". In: *Journal of Simulation* 7.3 (Aug. 2013), pp. 202–215. ISSN: 1747-7786. DOI: 10.1057/jos.2012.27. URL: http://dx.doi.org/10.1057/jos.2012.27.

[Ran20] Allison Randal. "The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers". In: *ACM Computing Surveys* 53.1 (Feb. 2020), pp. 1–31. ISSN: 1557-7341. DOI: 10.1145/3365199. URL: http://dx.doi.org/10.1145/3365199.

[Tro23] Elisa Tronetti. "Towards Aggregate Programming in pure Kotlin through compiler-level metaprogramming." Master's thesis. Università di Bologna, 2023. URL: https://amslaurea.unibo.it/id/eprint/28077/.

[Zah+16] Matei Zaharia et al. "Apache Spark: a unified engine for big data processing". In: *Communications of the ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 1557-7317. DOI: 10.1145/2934664. URL: http://dx.doi.org/10.1145/2934664.

A Cross-platform Development in Scala

This appendix provides a brief guide on the main concepts and tools available for building cross-platform applications and libraries in Scala.

Build Configuration

Cross-platform development is enabled by SBT Cross Project Plugin[1] which allows defining cross-platform Scala projects targeting JVM, JavaScript and Native platforms. An example of SBT cross-build configuration is provided in Listing A.1. Notable configuration options to pay attention to include:

- **Flavor.** Programmers can choose between two different flavors of cross-projects: *Pure* and *Full.* The first is intended for code that does not depend directly on any platform-specific libraries or functionality, though it may rely on third-party cross-platform libraries. The latter allows defining platform-specific code and dependencies, while still sharing common code across all platforms
- **Native GC.** On the Native platform, the Garbage Collector (GC) is responsible for automatically managing heap-allocated Scala objects.
- **Link Time Optimization (LTO).** On the Native platform, LTO can be enabled to optimize the final binary by performing optimizations across module boundaries during the linking phase. This can lead to smaller and faster executables, at the cost of increased compilation time. Beware on some architectures (like Raspberry Pi ARMv7) LTO may not be fully supported.
- **Native Build Target.** On the Native platform, Build Target option allows choosing between building an executable application or a shared/static

[1]`https://github.com/portable-scala/sbt-crossproject`

library. In the first case, the output is a standalone binary that can be executed directly on the target platform. With static or shared library, the output is a `.a` or `.so` file that can be linked and used by other applications (it requires exported definitions);

- **JS Module Initializer.** On Scala.js platform, the module initializer option instructs the Scala.js compiler to generate a JavaScript module that automatically invokes the application main method when the resulting JavaScript file is loaded.

Listing A.1: Example of SBT cross-build configuration for Scala cross-platform development.

```scala
lazy val crossModule = crossProject(JSPlatform, JVMPlatform, NativePlatform)
  .crossType(CrossType.Full)
  .in(file("crossModule"))
  .settings(
    // common settings here, including dependencies (use %%%)
  )
  .jsSettings(
    // Scala.js specific settings
  )
  .jvmSettings(
    // JVM specific settings
  )
  .nativeSettings( // Native specific settings
    nativeConfig ~= {
      _.withGC(GC.immix) // or boehm | commix | none
        .withLTO(LTO.full) // link time optimization: thin | none
        .withMode(Mode.releaseSize) // or debug | releaseFull
        .withBuildTarget(BuildTarget.libraryDynamic) // application |
            libraryStatic
    }
  )
```

Project Structure and code organization

Once applied, depending on the chosen flavor, the plugin organizes the project structure accordingly:

- **Pure Cross-Project**: the project structure contains a single shared source set, under the canonical `src/` directory, containing all the application or library code. No platform-specific source code is allowed, and the programmer can only rely on cross-platform libraries and functionalities;
- **Full Cross-Project**:
  - `shared/`: contains source code that can be shared across all platforms.

If some functionality is not supported on a specific platform the programmer needs to provide platform-specific implementations in appropriate directories;

- `jvm/`, `js/`, `native/`: contain platform-specific source code and dependencies for JVM, JavaScript and Scala Native platforms respectively. Inside these directories the programmer can directly access platform-specific libraries and functionalities, like Node.js APIs in Scala.js or C type system abstractions in Scala Native;
- `jvm-native/`, `js-native/` and all other combinations: contain source code and dependencies shared between a subset of platforms.

Despite other cross-platform programming languages, Scala does not provide an explicit mechanism for building platform-specific contracts and implementations, like Kotlin's Multiplatform `expect`/`actual` mechanism. However, a similar mechanism can be achieved using standard trait definitions in the shared code and providing platform-specific implementations in the appropriate source sets. Indeed, as long as each supported platform provides an implementation of a given concept, the shared code can reference and use it without concern for platform-specific details, provided that no platform-dependent functionality leaks through the abstract interface. For example, Listing A.2 shows how can be defined an abstract contract for asynchronous sleep operation in the shared code, and how to provide platform-specific implementations for JVM and JavaScript platforms. Despite `Async` object being defined in platform-specific source sets, the shared code can reference and use it transparently as long as it is implemented for all target platforms.

Listing A.2: Example of how to implement platform-specific contracts and implementations in Scala cross-platform development.

```scala
trait AsyncOperations:
  def sleep(duration: FiniteDuration): Future[Unit]

// JVM & Native implementation under ‘jvm-native‘ source set
object Async:

  def operations: AsyncOperations = new AsyncOperations:
    given ExecutionContext = ExecutionContext.fromExecutor(
        Executors.newCachedThreadPool())

    override def sleep(duration: FiniteDuration): Future[Unit] =
      Future(blocking(Thread.sleep(duration.toMillis)))

```

```
// JS implementation under 'js' source set
object Async:

  def operations: AsyncOperations = duration =>
    val p = Promise[Unit]()
    setTimeout(duration)(p.success(())): Unit
    p.future
// In 'shared' source set you can access Async object
for
  _ <- Async.operations.sleep(2.seconds)
  ...
yield ...
```

Scala Native Interoperability

Scala Native interoperability with C code is provided by using Scala Native's annotations and type system abstractions for C language constructs. The interaction is bidirectional: Scala Native code can call C functions and libraries, while C code can call Scala Native exported functions. The key mechanisms include:

- `@extern` annotations, applied to object definitions, define C function bindings in Scala. Like C header files, they declare function signatures that the compiler maps to actual C implementations in linked libraries, allowing Scala code to call C functions directly;
- `@link` annotation specifies which native libraries should be linked when compiling, enabling Scala Native to resolve external C function calls at link time;
- `@exported` annotations mark Scala Native functions that should be visible and callable from C code, making them available when linking Scala Native code as a library. Currently, it only supports static object methods and properties. A proposal exists to add a `@struct` annotation for exporting classes as C structs[2], but this has been blocked due to issues in passing structs by value.

C interoperability relies on Scala Native's type abstractions in the `scala-native.unsafe` package, which provides Scala representations for C constructs, including, but not limited to, `CStructN` for structures and `Ptr[T]` for pointers. Primitive type aliases, like `CString`, enable direct manipulation of C data while maintaining Scala's type safety.

[2]`https://github.com/scala-native/scala-native/issues/897`

While both Scala Native type facades and exporting mechanisms can be written manually, the Scala Native Bindgen plugin[3] can be used to automatically generate Scala Native bindings from C header files. Its usage significantly reduces the effort required to interface with existing C libraries by automating the generation of necessary Scala Native code that otherwise would need to be written by hand. Listing A.3 illustrates an SBT configuration employing the Bindgen plugin to generate Scala Native bindings from the ScaFi C header file located in the `resources/` directory.

Listing A.3: SBT configuration for the Polyglot Abstraction Layer using the Scala Native Bindgen plugin to generate C bindings.

```scala
lazy val `scafi3-polyglot-api` = crossProject(JSPlatform, JVMPlatform,
    NativePlatform)
  .crossType(CrossType.Full)
  .in(file("scafi3-polyglot-api"))
  .dependsOn(
    `scafi3-core` % "compile->compile;test->test",
    `scafi3-distributed` % "compile->compile;test->test"
  )
  .nativeEnablePlugins(BindgenPlugin)
  .nativeSettings(
    commonNativeSettings,
    bindgenBindings += Binding(
      header = (Compile / resourceDirectory).value / "include" / "scafi3.h",
      packageName = "it.unibo.scafi.nativebindings",
    )
  )
```

Unfortunately, the automation of header file generation from Scala Native exported functions is not currently supported. Indeed, developers are expected to manually write the header files declaring the exported functions to be used in C code and then generate the corresponding Scala Native code with appropriate annotations using the Bindgen plugin.

Several key aspects of interoperability should be noted:

- While theoretically possible to expose Scala objects and classes directly to C code, via an opaque pointer mechanism, representable as `Ptr[Byte]` in Scala Native, this approach is not suggested due to the complexity of managing object lifetimes and memory safety across language boundaries. Indeed, Scala objects are managed by Scala Native's Garbage Collector, which may lead to issues if C code holds references to Scala objects that have been collected. If necessary, it is necessary to maintain references to Scala objects in Scala code to prevent their collection while they are still needed by C code.

[3] `https://sn-bindgen.indoorvivants.com`

- Generic types can be represented, in Scala Native exported functions, as `Ptr[Byte]` pointers (`void*` in C). The direct usage of type parameters, while not prohibited, is discouraged since they are erased to `java.lang.Object` in the generated code and therefore not correctly represented as Scala Native `Ptr[T]` abstractions. As a result, Scala Native expects JVM object references to be passed, which may lead to runtime errors if the actual type is a native type, like a pointer.

Scala JS Interoperability

Similarly to Scala Native, Scala.js provides interoperability mechanisms to interact with JavaScript code:

- `@js.native` annotation define type-safe Scala interfaces for JavaScript APIs. Like `@extern` objects in Scala Native, they map Scala types to JavaScript objects and functions, enabling statically-typed access to JavaScript code;
- `@JSImport` and `@JSGlobal` specify module imports and global object access, serving a similar role to `@link` by declaring which JavaScript code to bind against;
- `@JSExport` and `@JSExportTopLevel` expose Scala.js members and top-level definitions to JavaScript for making Scala code callable from the host language.

Automated binding generation is also available through ScalablyTyped[4], which generates Scala.js facades from TypeScript definition files.

[4] `https://scalablytyped.org`