Grado en Ingeniería Informática

Proyecto de final de carrera

Sobre el diseño y la implementación de un modelo de programación puramente funcional para sistemas distribuidos y procesamiento de datos.

Autor:
Jorge Vicente Cantero

Supervisor (EPFL):
Dr. Heather Miller
Supervisor (UJI):
Prof. Juan Miguel Vilar
Responsable (EPFL):
Prof. Martin Odersky

Fecha de lectura: 13 de Junio de 2016
Curso académico 2015/2016

Resumen

El modelo function passing proporciona un soporte más directo sobre el cual construir sistemas distribuidos orientados a datos. En resumen, construye una estructura de datos funcional y persistente que representa transformaciones en datos immutables y distribuidos pasando funciones serializables y bien tipadas entre distintos nodos y aplicandolas sobre datos immutables. De esta manera, el diseño del modelo simplifica la tolerancia a fallos—los datos se pueden recuperar aplicando otra vez las funciones sobre los datos originarios guardados en memoria. Técnicas como la evaluación diferida son centrales en el modelo, haciéndolo eficiente y fácil de entender, y evaluándolo sólo en el punto en el que una comunicación en la red comienza.

Este proyecto provee un resumen de la implementación de tal modelo en Scala, discutiendo importantes mejores requeridas en dos complejas extensiones del compilador de Scala: Scala Pickling y Spores, así como lo requerido para aunar ambos proyectos de una forma óptima. El presente trabajo permite una míriada de nuevas oportunidades para construir sistemas distribuidos orientados a datos; ya no solo sistemas como Apache Spark, aunque puede ser visto como el caso de uso típico para este modelo. Mientras el modelo estó diseñado para ser agnóstico de la plataforma (está implementado en Scala y se ejecuta sobre la JVM), puede interoperar con otros lenguajes de programación como Javascript mediante Scala.js, un plugin del compilador de Scala que permite a cualquier framework basado en Javascript beneficiarse de esta contribución.

Palabras clave

computación distribuida, function passing, serializable, comunicación tipada, apache, spark, tipos, modelo de programación, big data, procesamiento de datos, diseño de lenguajes de programación, programación funcional

Keywords

distributed computing, function passing, serializable, typed communication, functional programming, data processing, programming languages, apache, spark, types, programing model, big data, programming languages design

Resumen extendido

Los frameworks con más éxito para el procesamiento de big data han adoptado APIs funcionales. Sin embargo, sus implementaciones están construidas sobre tecnologías poco tipadas, basadas en un paradigma imperativo, que complican el diseño e implementación de propiedades clave en los sistemas distribuidos como la tolerancia a fallos. Consecuentemente, estas plataformas se enfrentan a los siguientes problemas:

1. **Dificultad de uso.** Los sistemas no pueden prevenir estáticamente el mal uso de ciertos host language features (del lenguaje de programación utilizado) que no han sido pensadas para un paradigma distribuido. Por consiguiente, esto produce errores en tiempo de ejecución muy difíciles de depurar y solucionar. Un claro ejemplo es la serialización de unsafe closures (funciones que no están tipadas y son distribuidas a otros nodos).

2. **Complejo mantenimiento.** Contrariamente a las APIs, las capas de implementación de menor nivel no están estáticamente tipadas y las tareas de mantenimiento y refactoring, claves para la calidad y futuro del producto, se hacen mucho más difíciles.

3. **Pérdida de oportunidades de optimización.** La ausencia de información estática sobre los tipos impide al compilador realizar optimizaciones clave sobre el código emitido. Por ejemplo, la garantía de crear código concreto para serializar objetos de cualquier tipo en tiempo de compilación es crucial, pues evita el uso de serialización en tiempo de ejecución, la cual es ineficiente y produce una sobrecarga del sistema, especialmente aquellos en tiempo real. Éste hecho ha obligado a muchas plataformas que se ejecutan sobre la JVM a optar por otras alternativas como Avro, Protocol Buffers or Kryo, puesto que la serialización estándar implementada en Java es notablemente ineficiente.

El modelo *function passing* proporciona un soporte más directo sobre el cual construir sistemas distribuidos orientados a datos. En resumen, construye una estructura de datos funcional y persistente que representa transformaciones en datos immutables y distribuidos pasando funciones serializables y bien tipadas entre distintos nodos y aplicandolas sobre datos immutables. De esta manera, el diseño del modelo simplifica la tolerancia a fallos—los datos se pueden recuperar aplicando otra vez las funciones sobre los datos originarios guardados en memoria. Técnicas como la evaluación diferida son centrales en el modelo, haciéndolo eficiente y fácil de entender, y evaluándolo sólo en el punto en el que una comunicación en la red comienza. Su diseño se centra en evitar los problemas descritos anteriormente y proveer al programador de un soporte más intuitivo para la construcción de sistemas distribuidos.

El trabajo aquí expuesto se basa en dos líneas de investigación anteriores: generación de código para serializar de una forma eficiente y esporas tipadas, *closures* que pueden ser serializables sin perder información estática sobre sus tipos.

El modelo extiende la programación monádica al envío y recepción de datos entre redes de ordenadores. Es, de hecho, un modelo dual a la programación basada en actores (actor model), con la salvedad de que en esta ocasión mantenemos los datos estacionados y los transformamos enviando la funcionalidad a los hosts que los almacenan, en lugar de enviar los datos e implementar la lógica de negocio en los actores.

En general, el model *function-passing* trae conjuntamente inmutabilidad, estructuras de datos persistentes, funciones monádicas de alto orden, fuerte estático tipado y lazy evaluation, siendo éste último el más importante de todos, puesto que sin él el modelo presentado sería ineficiente en ambos tiempo y memoria. Las garantías de tolerancia a fallos están basadas en el concepto de lineage, que utiliza un DAG (grafo acíclico dirigido) para representar la secuencia de transformaciones a partir de unos datos iniciales (concepto inspirado por Resilient Distributed Datasets, también conocidos como RDDs). Así, en caso de algún fallo, la información puede volverse a computar recorriendo el grafo y aplicando secuencialmente las transformaciones. Este concepto está siendo utilizado en producción por sistemas como Apache Spark, Twitter's Scalding, Scoobi, Dryad and Hadoop MapReduce. Conjuntamente, para asegurar la recepción de mensajes entre los nodos, se utiliza el algoritmo Reply-ACK y otros algoritmos orientados a proporcionar un sistema tolerante a fallos embebido en el mismo diseño.

Esta proyecto provee un resumen de la implementación de tal modelo en Scala, discutiendo importantes mejores requeridas en dos complejas extensiones del compilador de Scala: *Scala Pickling* y *Spores*, así como lo requerido para aunar ambos proyectos de una forma óptima. El presente trabajo permite una míriada de nuevas oportunidades para construir sistemas distribuidos orientados a datos; ya no solo sistemas como Apache Spark, aunque puede ser visto como el caso de uso típico para este modelo. Mientras el modelo está diseñado para ser agnóstico de la plataforma (está implementado en Scala y se ejecuta sobre la JVM), puede interoperar con otros lenguajes de programación como Javascript mediante *Scala.js*, un plugin del compilador de Scala que permite a cualquier framework basado en Javascript beneficiarse de esta contribución.

El proyecto ha sido llevado a cabo en el LAMP (Laboratorio de Métodos de Programación), situado en la EPFL (Escuela Politécnica Federal de Lausanne, Suiza) y supervisado por Heather Miller (doctora e investigadora) y Martin Odersky (catedrático y creador del lenguaje de programación Scala). El modelo presentado está diseñado para ser adoptado por lenguajes y sistemas del mundo real. Así pues, este proyecto consiste en la implementación de tal sistema por y para Scala, que ya es el lenguaje host de alguna de las plataformas mencionadas, así como aportes y mejoras teóricas al modelo. A continuación, se presenta el contenido de la memoria, que está escrito en inglés.

Declaration of Authorship

UNIVERSITAT JAUME I

Abstract

Computer Science School

Departament de Llenguatges i Sistemes Informàtics

Computer Engineering

On the design and implementation of a purely functional programming model for distributed systems and big data processing

by Jorge VICENTE CANTERO

The *function passing* model provides a more principled substrate on which to build data-centric distributed systems. In a nutshell, it builds up a persistent functional data structure representing transformations on distributed immutable data by passing well-typed serializable functions over the wire and applying them to this distributed data. Thus, the model simplifies failure recovery by design—data is recovered by replaying function applications atop immutable data loaded from stable storage. Deferred evaluation is central to the model and makes it efficient and easy to reason about by incorporating it only at the point of initiating network communication.

This thesis provides an overview of its implementation in Scala, discussing important improvements required in two non-trivial Scala compiler extensions as open source frameworks: *Scala Pickling* and *Spores*, as well as what's required to provide an optimal binding between the two. The present work enables a myriad of new opportunities to build fault-tolerant data-centric distributed systems, not just systems like Apache Spark alone, although it can be viewed as the canonical use case for the function passing model. While the model is designed to be platform-agnostic, it is implemented in Scala and runs on the JVM, interoperating with other programming languages like JavaScript through *Scala.js*, a Scala compiler plugin that allows full-stack JavaScript applications to also benefit from this contribution.

Acknowledgements

The present Bachelor's thesis couldn't have been done without the inconditional support of Heather Miller, who introduced me to the project, helped me to arrange my stay at LAMP (*EPFL*, Lausanne, Switzerland) for the Spring of 2016, and gave me the chance to work in one of the best functional programming research laboratories in Europe. I kindly appreciate the confidence that she has placed on me. Her trust has made me realize what I'm capable of and, most of all, has inspired me to pursue my academic studies further in the field.

I would also like to thank Juan Miguel Vilar, whose feedback has been invaluable to improve the quality of this thesis, and who has also encouraged me to continue exploring the world of Computer Science and, particularly, Functional Programming.

Contents

List of Figures

List of Tables

Dedicated to those who have always been by my side.

Chapter 1

Introduction

1.1 Context and Motivation

1.1.1 Data-centric Programming

State-of-the-art software for distributed data processing is essential in modern societies. While these frameworks take care of the storage, partition and analysis of this data, developers interact with them to perform concrete business-related tasks, dealing only with transformations over data and forgetting about the nitty-gritty details of these complex systems. This way of thinking gave birth to data-centric programming, a recent movement that is rapidly growing in importance.

With it, it can be observed that most successful systems for programming with "big data" have adopted ideas from functional programming; *i.e.* programming with first-class functions and higher-order functions. These functional ideas are often touted to be the key to the success of these frameworks. Authors and users alike claim that a functional, declarative interface to data distributed over tens to thousands of nodes provides a more natural way for users as diverse as distributed systems engineers to data scientists to reason about data.

Popular implementations of the MapReduce (Dean and Ghemawat, 2008) model, such as Hadoop MapReduce (Apache, 2015b) for Java, have been developed without making use of functional language features such as closures. In recent years, a new generation of distributed systems for large-scale data processing have suddenly cropped up, using emerging functional languages like Scala; such systems include Apache Spark (Zaharia et al., 2012), Twitter's Scalding (Twitter, 2015), and Scoobi (NICTA, 2015) and make use of functional language features in Scala in order to provide high-level, declarative APIs to end-users. Also, the benefits provided by functional programming have also won over framework designers as well—some have noticed that immutability, and data transformation via higher-order functions makes it much easier, by design, to tackle concerns central to distributed systems.

1.1.2 Disadvantages of the State of the Art

These frameworks greatly benefit from these ideas, since they foster a frictionless interaction with end-users.

However, the frameworks' internals still leverage imperative programming and build atop of tall stack of untyped code, losing the benefits enjoyed by the users of their high-level APIs. These design decisions come at great expense. Suddenly, key concerns like correctness, fault tolerance and concurrency become blurred, difficult to reason about and realize in practice. Further, as distributed systems are an intrinsic source of runtime errors,[1] the untyped low-level layers cannot statically prevent common usage errors and provide rich error reports. As a result, users are confronted with hard-to-debug errors whose cause is unknown beforehand. Consequently, the original strategy of these frameworks backfires and greatly impacts ease-of-use.

Yet, end-users are not the only ones affected. Other problems manifest themselves in areas like maintenance and performance due to the absence of types. Developers, who have to deal with these issues, experience slowdowns in day-to-day tasks like code reviews and refactorings, that could potentially introduce new, hidden misbehaviours.[2] Also, performance is difficult to improve because the compiler cannot apply type-specific optimizations, like the elimination of the boxing and unboxing techniques performed by the JVM,[3] and serialization techniques cannot use type-specific serialization code, thus exclusively relying on reflection-based tools, which are slow and problematic in the long term.

In conclusion, the aforementioned issues affect both end-users and developers. Their effects skyrocket in large codebases and hurt the desired user experience. Sooner or later, developers not only struggle to introduce new features but suffer to maintain them.

As these large-scale data processing and distributed applications continue to grow in importance, what can we as language designers and software developers do to make it easier for more of these frameworks to rise? Are these pitfalls inherent to the nature of such systems? Or, conversely, can we do better?

1.2 A New Solution

Looking at the root of the problem, one may find that untyped, imperative code is an aftermath of proper support for distributed use cases; missing language features that force the users to roll their own ways to develop these distributed frameworks, and end up in large unmantainable codebases.

There is no doubt that, whereas modern programming languages are trying to keep up, very few of them have been designed with distributed programming in mind. In addition, such support is particularly hard to get right—not only basic features need to be provided, users expect more advanced features

[1]Existing distributed systems range accross different use cases and scenarios. However, their distributed nature rely on error-prone tasks like connection accross networks and data serialization.

[2]Note that conventional test suites are not enough to test the correct functioning of distributed systems. The space of cases to test increase abruptly and the overall behaviour is nondeterministic and, therefore, difficult to predict.

[3]For a full of explanation of these optimizations, see (Dragos, 2010), which thoroughly describes these techniques in and for the Scala programming language.

to work as well. Striking a balance in such a task is difficult and requires an in-depth consideration.

This sudden proliferation of new frameworks for distributed data-centric programming concurrent with the sudden growth in popularity of an emerging programming language begs the question: has it been our programming languages that have limited us? Could it be that the primitives we build our systems upon are too low-level, causing us to struggle to reinvent the same tricky wheel over and over again?

To answer that question, (Miller et al., 2016) previously proposed a new programming model called the *function passing* model which has been designed to be a more principled substrate (or middleware) upon which to build data-centric distributed systems. It can be viewed as a generalization of the MapReduce/Spark programming model—though it is not limited to the MapReduce/Spark programming model alone.

The function-passing model would allow any distributed system to become a reality, regardless of its type, size and location. The model is generic and transparent; it does not have any assumption or constraint on the myriad of possibilities that enables, from systems that need to scale out[4] to those that scale in[5]. These decisions boil down to the users' needs, whereas the rest is taken care of by the network backend.

This interesting idea came along with an initial proof of concept, built atop of Akka (Typesafe, 2015). Unfortunately, its primary goal was not to work correctly, only to prove that such model was indeed realizable. To name a few of the disadvantages: it lacked implementation for the most important primitives, it had rudimentary support for fault-tolerance techniques, serialization relied on flaky versions of *Scala Pickling*, efficiency was seriously damaged by the overhead of *Akka* and only a very constrained subset of spores (safe closures) could be sent over the wire. As a result, real-world applications weren't able to benefit from it, and its existence was ignored.

In an attempt to turn around this situation, the goal of this thesis is to make the function-passing programming model happen, therefore improving the language support for distributed applications in Scala. Hence, along this document, I expose the different approaches to achieve a fully-working efficient implementation, ready to be adopted by industry or, at least, set a starting point towards better distributed frameworks built atop of *Scala*.

1.3 Goals and Contributions

The function-passing implementation circumscribes important improvements in two non-trivial Scala compiler extensions as open source frameworks:

[4]Horizontal scalability is the process of adding or removing nodes to a distributed software application. These nodes can be both in local or remote networks.

[5]Vertical scalability is the process of adding or removing computing resources to a single node in a distributed system

TABLE 1.1: The contributed lines of code to *Spores*, *Scala Pickling* and the new *function-passing* implementation.

Lines	**Spores**	**Scala Pickling**	**Function-passing**	**Total**
New	1,167 ++	13,166 ++	7,423 ++	21,756 ++
Removed	1,045 −−	12,270 −−	-7,146 −−	20,461 −−

Scala Pickling and *Spores*, as well as what's required to provide an optimal binding between the two. While the model is designed to be platform-agnostic, it is implemented in Scala,[6] runs on the JVM and it interoperates with other programming languages like JavaScript through *Scala.js*, a Scala compiler plugin that allows full-stack JavaScript applications to also benefit from this contribution. In short, this thesis contributes:

- ***A distributed implementation of the programming model*** in Scala, available for the JVM and with support for JavaScript platforms, that is simple, modular, efficient and backend independent. This production-ready implementation has allowed us to build popular frameworks like Spark and MBrace (Dzik et al., 2013), and end-user applications we have built using each of these prototype frameworks. The design and use of these are, however, not included in this thesis.

- ***A concrete analysis of the requirements for an efficient implementation of our programming model***, as well as a discussion of our improvements in Scala Pickling (Miller et al., 2013) and Spores (Miller, Haller, and Odersky, 2014) for enabling an efficient implementation. Efficiently serializing safe closures is a fundamental problem and the cornerstone of uniting functional programming and distributed computing. Despite using Scala, the solutions that we propose are language-independent and can be replicated in other functional programming languages.

- ***Theoretical contributions to the original paper*** that explain several ways to share *SiloRef*s among different nodes and approach the problem of memory reclamation and fault tolerance. Such contributions have been published in the official function-passing paper (Miller et al., 2016).[7]

The above contributions are quantified in Table 1.1, according to Github, and shows the overall amount of contributed lines of code. It is of particular interest the relation between the new and removed lines of code. While a lot of functionality has been incorporated in the three projects, significant parts have also been refactored and generalised, encouraging code reusability and improving the overall quality.

[6]Scala is the host language of some of the important distributed big-data frameworks, such as Apache Spark (Zaharia et al., 2012)

[7]Note that the contents of these thesis have been published by Jorge Vicente Cantero in (Cantero, Miller, and Haller, 2016), a paper that is under submission to *ECOOP 2016*. Arguably, one of the main objectives of this thesis was to introduce the student to real-world research and, as such, this is the result.

1.4 Structure and Organization of the Thesis

The rest of this thesis is organized as follows. We begin in Chapter 2 with a general overview of the function-passing model and how it works in theory. Further, in Chapter 3 we analyse its requirements and take a close look to the required organisation to realize its implementation, along with a detailed project schedule. Later, Chapter 4 delves into the function-passing implementation, bringing everything together and describing its design. Since this thesis's contributions relate directly to intricacies derived from the interaction of other non-trivial frameworks, it first starts with a few background sections.

In Section 4.2.1, we give an introduction to Scala Pickling, an automatic and performant serialization framework which the function passing framework makes heavy use of. In Section 4.2.2 we give an introduction to Scala Spores, a framework for ensuring that closures are guaranteed to be serializable. We get to the meat of our contributions in Section 4.3 by detailing the myriad ways we improved the Scala Pickling framework. Conversely, Section 4.4 details the ways in which we improved the Scala Spores framework. We tie everything together and show how to pickle spores in Section 4.5. And finally, we conclude in Chapter 5.

For readers unfamiliar with Scala and their key language features, we refer them to the appendices. Appendix A introduces *implicits* in Scala, a language feature that is central to each of these frameworks, and Appendix B introduces *macros*, a compile-time metaprogramming technique in Scala that is the main underlying component of the implementation of Scala Pickling and Spores.

Chapter 2

The Function-passing Model

2.1 The Essence

The key idea behind the function passing model is to keep distributed (immutable) data stationary, and to instead send functionality as function closures over the network. This enables two important benefits for distributed system builders; (a) since all computations are functional transformations on immutable data, fault-tolerance is made simple by design, and (b) communication is made well-typed by design, a common pain point for builders of distributed systems in Scala. Said another way, the function passing model attempts to more naturally model the paradigm of data-centric programming by extending monadic programming to the network.

On this note, one might observe that the function passing model can actually be interpreted as somewhat of a dual to the actor model; rather than keeping functionality stationary and sending data, the model keeps data stationary and send functionality to the data. This idea is exemplified in figures 2.1 and 2.2.

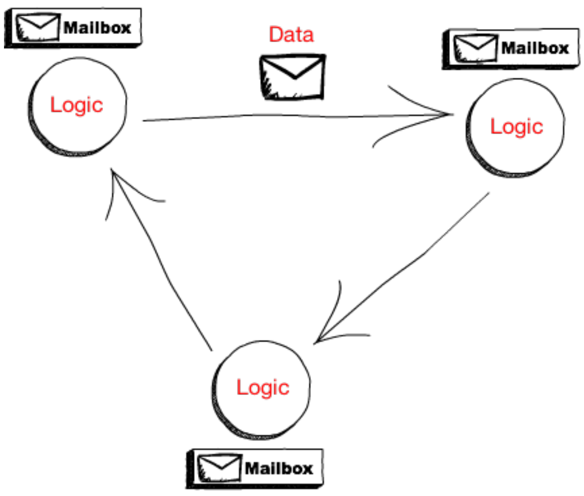


FIGURE 2.1: The Actor model.

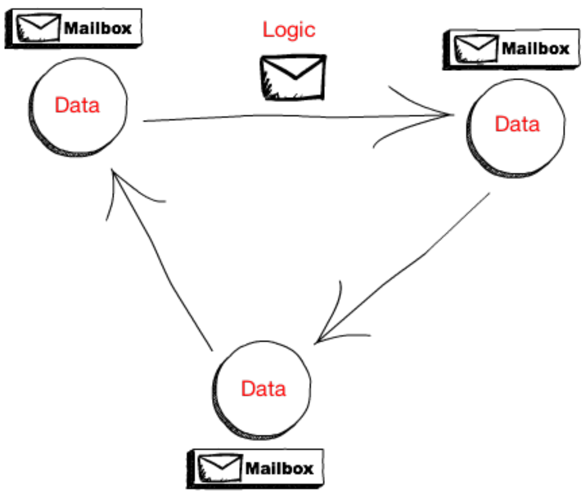


FIGURE 2.2: The Function-Passing model.

The function passing model brings together immutable, persistent data structures, monadic higher-order functions, strong static typing, and deferred evaluation—pillars of functional programming—to provide a more type-safe, and easy to reason about foundation upon which to build data-centric distributed systems. Further, it provides a precise specification of the semantics of functional fault recovery.

In the broadest sense, the function passing model can be thought of as a sort of persistent functional data structure with structural sharing. However, rather than containing pure data, instead the data structure represents a directed acyclic graph (DAG) of functional transformations on distributed data. The root node of is immutable data read from stable storage (*e.g.* Amazon S3); edges represent functional transformations on immutable data represented as nodes of the DAG.

Importantly, since this DAG of computations is a persistent data structure itself, it is safe to exchange (copies of) subgraphs of a DAG between remote nodes. This enables a robust and easy-to-reason-about model of fault tolerance. Subgraphs of the DAG are called *lineages*; lineages enable restoring the data of failed nodes through re-applying the transformations represented by their DAG. This sequence of applications must begin with data available from stable storage.

Central to the function passing model is the careful use of deferred evaluation. Computations on distributed data are typically not executed eagerly; instead, applying a function to distributed data just creates an immutable lineage. To make a network call and thus obtain the result of a computation, it is necessary to first "kick off" computation, or to force its lineage. Within the programming model, this force operation (called `send()`) makes network communication (and thus possibilities for latency) explicit, which is considered to be a strength when designing distributed systems (Waldo et al., 1996). Deferred evaluation also enables optimizing distributed computations through operation fusion, which avoids the creation of unnecessary intermediate data structures—this is efficient in time as well as space. This kind of optimization is particularly important and effective in distributed systems (Chambers et al., 2010).

2.1.1 Basic introduction to functional programming

Functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data (Wikipedia, 2016b).

The fundamental concept of functional programming is that of higher-order functions. Given the background of this discipline, that comes from the lambda calculus (Church, 1941), a mathematical abstraction for the description and evaluation of functions, functions are placed at its very core. By convention, functional programming languages treat them as first-class members and programs are written thanks to function composition. As we will discuss later, they play an invaluable role in the function-passing model. In addition, it also benefits from the following peculiarities of functional programming.

Immutability and state. While mainstream software builds upon the idea of mutable state, functional programmers dodge it. Instead, they rely on immutability: once an element has been created, it cannot be changed. As a result, a new one is created and propagated through the whole program. Intuitively, this approach that may sound inefficient but thanks to structural sharing it becomes possible to use it in practice with almost no overhead.

Non-strict evaluation. Also known as lazy evaluation, it delays the evaluation of an expression until its value is required in the program. In practice, this means that no initialization will occur at runtime but only at the first call site. As an alternative to strict evaluation, it's not meant to be a replacement but a powerful evaluation technique that enables critical performance improvements in concrete scenarios. In Scala, a variable can be initialized lazily by adding the modifier `lazy`, whereas in other languages like Haskell every expression is by default evaluated in a non-strict way by the compiler.

Fusion operation. A unique advantage of dealing with pure functions as the most elemental logical part of a program is fusion operation. This technique allows us to merge any number of functions into one and improves the overall performance of the program—a compiler is able to merge functions and avoid object initialization overhead, a significant issue for languages like Scala that represent functions as classes.

Monads. Scala, as a programming language that combines object-oriented and functional programming, encourages developers to write pure functions. Pure functions do not perform any side effect, they map values of different types without altering a state or having an observable interaction with the outside world. A typical instance of an effectful function is:

```
def sum(x: Int, y: Int) = {
  val res = x + y
  println(s"x + y = {res}")
```

```
    res
  }
```

In the above example, `println` is a side-effect, since it triggers an IO effect within a function that was supposed to map `Int => Int`. Although this is a not recommended practice when programming in Scala, side effects are at the core of any program and we cannot get rid of them. Yet, functional programming provides a better way to deal with them, through the so-called *monad*.

Formally, a monad is a mathematical structure that represents computations as sequences of steps, a pipeline of computations from an input to an output. In short, it can model computations and their combinations, even if they contain side effects. This introductory explanation does not aim at giving a full explanation of this abstraction—there are books devoted to the sole monad abstraction—, but a general overview of what a monad is.

Mathematically, a monad is defined by two functions:

1. **return**, also known as **unit**.
2. **join**, also known as **flatMap**.

Monads abide by the so-called monad laws, which are not described in this introduction. For readers interested in monads and their properties, we refer them to (Wadler, 2010) that explains thoroughly their essence and importance in functional programming.

Closures. Closures lexical closures or function closures) are a technique for implementing lexically scoped name binding in languages with first-class functions (Wikipedia, 2016a). A closure is composed of a function together with an environment. An environment is defined as a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or storage location to which the name was bound when the closure was created. Closures allow the body of the function to access those captured variables through the closure's reference to them, even when the function is invoked outside their scope.

2.1.2 The Function-Passing Model

The function passing model consists of three main components:

- **Silos:** stationary, typed, immutable data containers.
- **SiloRefs:** references to local or remote Silos.
- **Spores:** safe, serializable functions.

Silos A silo is a typed and immutable data container. It is stationary in the sense that it does not move between machines—it remains on the machine where it was created. Data stored in a silo is typically loaded from stable storage, such as a distributed file system. A program operating on data stored in a silo can only do so using a reference to the silo, a `SiloRef`.

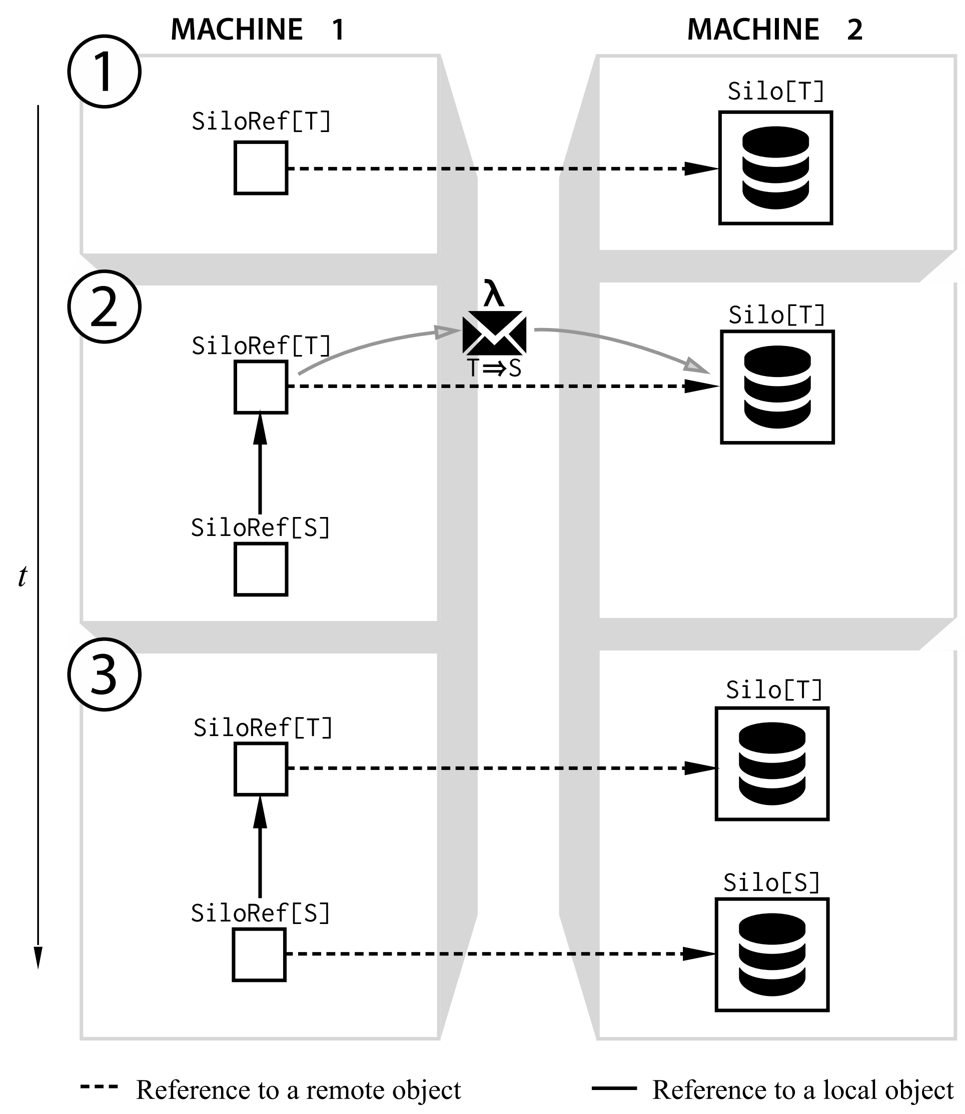


FIGURE 2.3: Basic function passing model.

SiloRefs Similar to a proxy object, a SiloRef represents, and allows interacting with, both local and remote silos. SiloRefs are immutable, storing identifiers to locate possibly remote silos. SiloRefs are also typed (`SiloRef[T]`) corresponding to the type of their silo's data, leading to well-typed network communication. The SiloRef provides three primitive operations/combinators: `map`, `flatMap`, and `send`. The `map` method makes use of deferred evaluation; it applies a user-defined function to data pointed to by the SiloRef, creating in a new silo containing the result of this application, though this application is *deferred*. That is, this computation is only kicked off when the `send` method is invoked. This makes it possible to queue up or stage transformations in order to optimize network communication. Like `map`, the application of `flatMap` is deferred. `flatMap` applies a user-defined function to data pointed to by the SiloRef. Unlike `map`, however, the user-defined function passed to `flatMap` returns a SiloRef whose contents are transferred to the new silo returned by `flatMap`. Essentially, `flatMap` enables accessing the contents of (local or remote) silos from within remote computations. I illustrate these primitives in more detail in Section 2.1.4.

2.1.3 Basic Usage

Let's begin with a simple visual example to illustrate the basics of the function passing model.

The main handle users have to the framework is via SiloRefs. A SiloRef can be thought of as an immutable handle to distributed data contained within a corresponding silo. Users interact with this distributed data by applying functions (as spores) to SiloRefs, which are transmitted over the wire and later applied to the data within the corresponding silo. As it is the case for persistent data structures, when a function is applied to a piece of distributed data via a SiloRef, a SiloRef representing a new silo containing the transformed data is returned.

The simplest illustration of the model is shown in Figure 2.3 (time flows vertically from top to bottom). Here, we start with a `SiloRef[T]` which points to a piece of remote data contained within a `Silo[T]`. When the function shown as λ of type $T \Rightarrow S$ is applied to `SiloRef[T]` and "forced" (sent over the wire), a new SiloRef of type `SiloRef[S]` is immediately returned. Note that `SiloRef[S]` contains a reference to its parent SiloRef, `SiloRef[T]`. (This is how *lineages* are constructed.) Meanwhile, the function is asynchronously sent over the wire and is applied to `Silo[T]`, eventually producing a new `Silo[S]` containing the data transformed by function λ. This new `SiloRef[S]` can be used even before its corresponding silo is materialized (*i.e.* before the data in `Silo[S]` is computed) – the function passing framework queues up operations applied to `SiloRef[S]` and applies them when `Silo[S]` is fully materialized.

Different sorts of complex DAGs can be asynchronously built up in this way. Though first, to see how this is possible, let's develop a clearer idea of the primitive operations available on SiloRefs and their semantics in the following section.

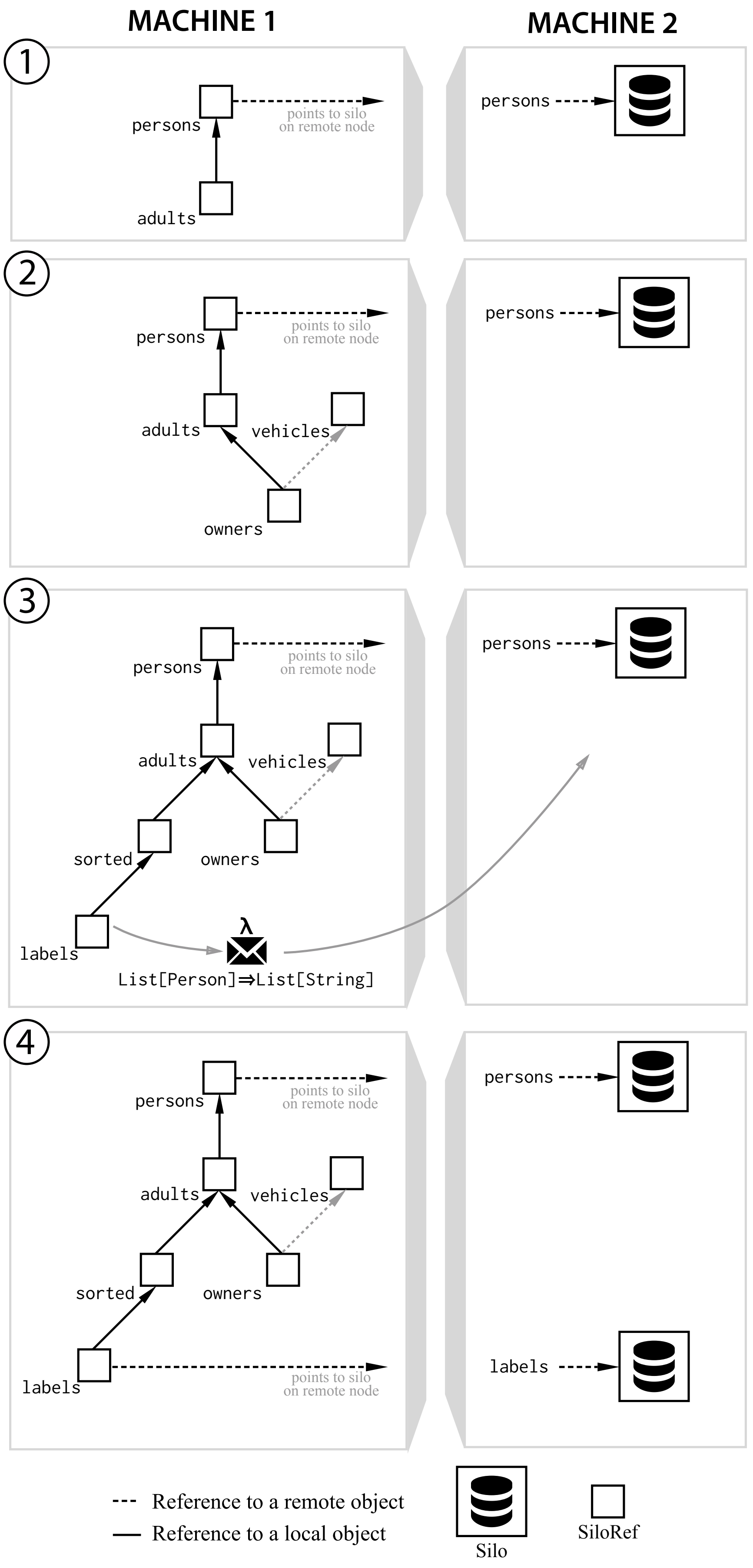

MACHINE 1
MACHINE 2
1
persons
adults
points to silo on remote node
persons
2
persons
adults
vehicles
owners
points to silo on remote node
persons
3
persons
adults
vehicles
sorted
owners
labels
λ
List[Person]⇒List[String]
points to silo on remote node
persons
4
persons
adults
vehicles
sorted
owners
labels
points to silo on remote node
persons
labels
--- Reference to a remote object
— Reference to a local object
Silo
SiloRef

FIGURE 2.4: A simple lineage or *DAG* in the function passing model.

2.1.4 Primitives

There are four basic primitive operations on SiloRefs that together can be used to build the higher-order operations common to popular data-centric distributed systems. In this section, I introduce these primitives in the context of a running example. These primitives include:

- `map`
- `flatMap`
- `send`
- `cache`

map `def map[S](s: Spore[T, S]): SiloRef[S]`
The `map` method takes a spore that is to be applied to the data in the silo associated with the given SiloRef. Rather than immediately sending the spore across the network, and waiting for the operation to finish, the `map` method's evaluation is *deferred*. Without involving any network communication, it immediately returns a SiloRef referring to a new, soon-to-be-created silo. This new SiloRef only contains lineage information, namely, a reference to the original SiloRef, a reference to the argument spore, and the information that it is the result of a `map` invocation. As explained below, another method, `send` or `cache`, must be called explicitly to force the materialization of the result silo.

To better understand how DAGs are created and how remote silos are materialized, I will develop a running example throughout this section. Given a silo containing a list of `Person` records, the following application of `map` defines a (not-yet-materialized) silo containing only the records of adults (graphically shown in Figure 2.4, part 1):

```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.map(spore { ps => ps.filter(p => p.age >= 18) })
```

flatMap `def flatMap[S](s: Spore[T, SiloRef[S]]): SiloRef[S]`

Like `map`, the `flatMap` method takes a spore that is to be applied to the data in the silo of the given SiloRef. However, the crucial difference is in the type of the spore argument whose result type is a SiloRef in this case. Semantically, the new silo created by `flatMap` is defined to contain the data of the silo that the user-defined spore returns. The `flatMap` combinator adds expressiveness to the model that is essential to express more interesting computation DAGs. For example, consider the problem of combining the information contained in two different silos (potentially located on different hosts). Suppose the information of a silo containing `Vehicle` records should be enriched with other details only found in the `adults` silo. In the following, `flatMap` is used to create a silo of `(Person, Vehicle)` pairs where the names of person and vehicle owner match (graphically shown in Figure 2.4, part 2):

```
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
```

```
val owners = adults.flatMap(spore {
  val localVehicles = vehicles // spore header
  ps =>
    localVehicles.map(spore {
      val localps = ps // spore header
      vs =>
        localps.flatMap(p =>
          // list of (p, v) for a single person p
          vs.flatMap {
            v => if (v.owner.name == p.name) List((p, v)) else Nil
          }
        )
    })
})
```

Note that the spore passed to `flatMap` declares the capturing of the `vehicles` SiloRef in its so-called "spore header." The spore header spans all variable definitions between the spore marker and the parameter list of the spore's closure. The spore header defines the variables that the spore's closure is allowed to access. Essentially, spores limit the free variables of their closure's body to the closure's parameters and the variables declared in the spore's header. Within the spore's closure, it is necessary to read the data of the `vehicles` silo in addition to the `ps` list of `Person` records. This requires calling `map` on `localVehicles`. However, `map` returns a SiloRef; thus, invoking `map` on `adults` instead of `flatMap` would be impossible, since there would be no way to get the data out of the silo returned by `localVehicles.map(..)`. With the use of `flatMap`, however, the call to `localVehicles.map(..)` creates the final result silo, whose data is then also contained in the silo returned by `flatMap`.

Although the expressiveness of the `flatMap` combinator subsumes that of the `map` combinator, keeping `map` as a (lightweight) primitive enables more opportunities for optimizing computation DAGs (*e.g.* operation fusion (Chambers et al., 2010)).

send `def send(): Future[T]`
As mentioned earlier, the execution of computations built using SiloRefs is deferred. The `send` operation *forces* the deferred computation defined by the given SiloRef. Forcing is explicit in the model, because it requires sending the lineage to the remote node on which the result silo should be created. Given that network communication has a latency several orders of magnitude greater than accessing a word in main memory, providing an explicit send operation is a judicious choice (Waldo et al., 1996).

To enable materialization of remote silos to proceed concurrently, the `send` operation immediately returns a future (Haller et al., 2012). This future is then asynchronously completed with the data of the given silo. Since calling `send` will materialize a silo and send its data to the current node, `send` should only be called on silos with reasonably small data (for example, in the implementation of an aggregate operation such as `reduce` on a distributed collection).

cache `def cache(): Future[Unit]`
The performance of typical data analytics jobs can be increased dramatically by caching large data sets in memory (Zaharia et al., 2012). To do this, the silo containing the computed data set needs to be materialized. So far, the only way to materialize a silo shown is using the `send` primitive. However, `send` additionally transfers the contents of a silo to the requesting node—too much if a large remote data set should merely be cached in memory remotely. Therefore, an additional primitive called `cache` is provided, which forces the materialization of the given SiloRef, returning `Future[Unit]`.

Given the running example so far, one can add another subgraph branching off of `adults`, which sorts each `Person` by age, produces a `String` gretting, and then "kicks-off" remote computation by calling `cache` and caching the result in remote memory (graphically shown in Figure 2.4, part 3 and 4):

```
val sorted =
  adults.map(spore { ps => ps.sortWith(p => p.age) })
val labels =
  sorted.map(spore { ps => ps.map(p => "Welcome, " + p.name) })
labels.cache()
```

Assuming one would also cache the `owners` SiloRef from the previous example, the resulting lineage graph would look as illustrated in Figure 2.4. Note that `vehicles` is not a regular parent in the lineage of `owners`; it is an indirect input used to compute `owners` by virtue of being *captured* by the spore used to compute `owners`.

Creating Silos

Besides a type definition for SiloRef, the framework also provides a companion singleton object (Scala's form of modules). The singleton object provides factory methods for obtaining SiloRefs referring to silos populated with some initial data:[1]

```
object SiloRef {
  def fromTextFile(h: Host)(f: File): SiloRef[List[String]]
  def fromFun[T](h: Host)(s: Spore[Unit, T]): SiloRef[T]
  def fromLineage[T](h: Host)(s: SiloRef[T]): SiloRef[T]
}
```

Each of the factory methods has a `h` parameter that specifies the target host (address/port) on which to create the silo. Note that the `fromFun` method takes a spore closure as an argument to make sure it can be serialized and sent to `h`. In each case, the returned SiloRef contains its `host` as well as a host-unique identifier. The `fromLineage` method is particularly interesting as it creates a copy of a previously existing silo based on the lineage of a SiloRef `s`. Note that only the SiloRef is necessary for this operation to successfully complete; the silo originally hosting `s` might already have failed.

[1]For clarity, only method signatures are shown. Please, also note that the below definition is slightly simplified, and therefore does not match completely the current implementation.

2.2 Fault Handling

The function passing model includes overloaded variants of the primitives discussed so far which enable the definition of flexible fault handling semantics. The main idea is to specify fault handlers for *subgraphs of computation DAGs*. The guiding principle is to make the definition of the failure-free path through a computation DAG as simple as possible, while still enabling the handling of faults at the fine-granular level of individual SiloRefs.

Defining fault handlers Fault handlers may be specified whenever the lineage of a SiloRef is extended. For this purpose, the introduced `map` and `flatMap` primitives are overloaded. For example, consider the previous example, but extended with a fault handler:

```
val persons: SiloRef[List[Person]] = ...
val vehicles: SiloRef[List[Vehicle]] = ...
// copy of `vehicles` on different host `h`
val vehicles2 = SiloRef.fromFun(h)(spore {
  val localVehicles = vehicles
  () => localVehicles
})

val adults =
  persons.map(spore { ps => ps.filter(p => p.age >= 18) })

// adults that own a vehicle
def computeOwners(v: SiloRef[List[Vehicle]]) =
  spore {
    val localVehicles = v
    (ps: List[Person]) => localVehicles.map(...)
  }

val owners: SiloRef[List[(Person, Vehicle)]] =
  adults.flatMap(computeOwners(vehicles),
                 computeOwners(vehicles2))
```

Importantly, in the `flatMap` call on the last line, in addition to `computeOwners(vehicles)`, the regular spore argument of `flatMap`, `computeOwners(vehicles2)` is passed as an additional argument. The second argument registers a *failure handler* for the subgraph of the computation DAG starting at `adults`. This means that if during the execution of `computeOwners(vehicles)` it is detected that the `vehicles` SiloRef has failed, it is checked whether the SiloRef that the higher-order combinator was invoked on (in this case, `adults`) has a failure handler registered. In that case, the failure handler is used as an alternative spore to compute the result of `adults.flatMap(..)`. In this example, we specified `computeOwners(vehicles2)` as the failure handler; thus, in case `vehicles` has failed, the computation is retried using `vehicles2` instead.

2.3 Peer-to-peer Patterns

So far, our examples have focused on master-worker topologies that underly models like Spark—*i.e.* a master node specifies identical DAGs of computation for all worker nodes to follow.

The function passing model, however, is not limited to these sorts of topologies. It is indeed possible to develop decentralized, peer-to-peer topologies on top of the function passing model. For example, a single compute node may host silos that are remotely referenced by remote SiloRefs, as well as SiloRefs remotely referencing silos on other compute nodes.

Further, as we show in the following example, it's also possible for multiple clients to build completely different DAGs of computation off of some source silo. In effect, this enables datasets to be shared—they exist once in memory on some node, but can be used and transformed in different ways by different clients.

Consider the following example. We start by populating an initial silo representing a dataset of `Vehicle` objects on `Host("lmpsrv1.scala-lang.org", 9999)`.

```
val lmpsrv1 = Host("lmpsrv1.scala-lang.org", 9999)

// client #1
// populate initial silo
val vehicles: SiloRef[List[Vehicle]] =
  Silo.fromTextFile(lmpsrv1)("hdfs://...")

val silo2 = vehicles.map(spore {
  (vs: List[Vehicle]) =>
    // extract US state from license plate string, e.g, "FL329098"
    vs.map(v => (v.licensePlate.take(2), v)).toMap
})
val vehiclesPerState = silo2.send()

// client #2
// get siloref for silo that is being materialized due to client #1
val vehicles: SiloRef[List[Vehicle]] =
  Silo.fromTextFile(lmpsrv1)("hdfs://...")

val silo2 = vehicles.map(spore {
  // list all vehicles manufactured since 2013
  (vs: List[Vehicle]) => vs.filter(v => v.yearManufactured >= 2013)
})
val vehiclesSince2013 = silo2.send()
```

Here, client #1 would like to perform some sort of computation based on the states that vehicles are registered in. Another client, client #2 would also like to access this dataset. To do so, one must simply once again invoke `fromTextFile` on the same host, `Host("lmpsrv1.scala-lang.org",9999)` to obtain a SiloRef that points to a corresponding silo that is already or soon to be materialized. From here, client #2 is able to build an entirely different DAG of computations, for instance in this example, filtering the original `vehicle` dataset to obtain only vehicles manufactured since 2013.

2.3.1 Decentralized Fault-Handling

Another peer-to-peer pattern possible in the function passing model is decentralized fault handling. One may specify strategies to transfer computation to other nodes in the event of failure.

Consider the following example: an aggregation should be performed as soon as two silos `vehicles` and `persons` have been materialized. The aggregation result is then combined with a silo `info` on some host different from the local host. The final result is written to a distributed file system:

```
object Utils {
  def aggregate(vs: SiloRef[List[Vehicle]],
                ps: SiloRef[List[Person]]): SiloRef[String] = ...
  def write(result: String, fileName: String): Unit = ...
}
val vehicles: SiloRef[List[Vehicle]] = ...
val persons:  SiloRef[List[Person]]  = ...
val info:     SiloRef[Info]          = ...
val fileName: String                 = "hdfs://..."
val done    = info.flatMap(spore {
  val localVehicles = vehicles
  val localPersons  = persons
  (localInfo: Info) =>
    aggregate(localVehicles, localPersons).map(spore {
      val in = localInfo
      res => combine(res, in)
    })
}).map(spore {
  val captured = fileName
  combined => Utils.write(combined, captured)
})
done.cache() // force computation
```

This program does not tolerate failures of the host of `info`: if it fails before the computation is complete, the result is never written to the file.

We can overcome this using fault handlers. It is possible to introduce another backup host which takes over in case the host of `info` (which is the same as the host of `done`) fails at any point. Let's try the above computation again, this time using fault handlers to transfer the computation to a backup node in the event of a failure:

```
val doCombine = spore {
  val localVehicles = vehicles
  val localPersons  = persons
  (localInfo: Info) =>
    aggregate(localVehicles, localPersons).map(spore {
      val in = localInfo
      res => combine(res, in)
    })
}
val doWrite = spore {
  val captured = fileName
  combined => Utils.write(combined, captured)
}
val done      = info.flatMap(doCombine).map(doWrite)
val backup    = SiloRef.fromFun(hostb)(spore { () => true })
val recovered = backup.flatMap(
  spore {
    val localDone = done
    x => localDone
  },
  spore { // fault handler
    val localInfo      = info
    val localDoCombine = doCombine
    val localDoWrite   = doWrite
    val localHostb     = hostb
    x =>
      // fromLineage makes sure, we re-run on hostb, rather than
      // the host of info. That is, we just duplicate the lineage.
      val restoredInfo = SiloRef.fromLineage(localHostb)(localInfo)
      restoredInfo.flatMap(localDoCombine).map(localDoWrite)
  }
)
done.cache()      // force computation on host of local
recovered.cache() // force computation on backup host
```

First, the local variables `doCombine` and `doWrite` refer to the verbatim spores passed to `flatMap` and `map` above. Second, `backup` is a dummy silo on a backup host `hostb`. It is used to send a spore to the backup host in a way that allows it to detect whether the host of `done`/`info` has failed. The fault handling is done by calling `flatMap` on `backup`, passing (a) a spore for the non-failure case and (b) a spore for the failure case. The spore for the non-failure case simply returns the `done` SiloRef. Importantly, this enables `hostb` to detect failures of the host of `done`. Upon detecting such a failure, `backup.flatMap` applies the spore for the failure case. In this case, the lineage of the captured `info` SiloRef is used to restore its original contents in a new silo created on the backup host `hostb`. Its SiloRef is then used to retry the original computation.

Chapter 3

Analysis and Project Schedule

When first looking at the realization of any project, one must take crucial decisions that certainly determine its success. This chapter presents a brief overview of the requirement analysis and delves into the process of figuring out the needs and properties of the project. Further, it explains the software development methodology and takes a close look to the project schedule. As a whole, it elucidates the decision-making process that guided the following implementation.

3.1 Methodology

The function-passing implementation has followed a standard software development process based on agile methodologies. Given the nature of the project, Scrum (Sutherland and Schwaber, 1995) was the employed methodology. This decision has been proven to be key to the final success of the project.

In hindsight, traditional alternatives such as the waterfall model would have headed this project to failure. Their major disadvantages are their strictness and inability to lazily adapt to possible real-world scenarios. While putting them in practice, they make strong assumptions on the stability of the technology stacks on which the software projects are based, and don't take into account unpredicted circumstances. As an example, several unexpected situations did indeed occur throughout the development process. These situations, explained later in detail, made reconsider aspects of the project, as well as changed the prioritisation of the tasks.

In particular, these properties don't suit the needs of a research project that sports the following features:

- **Volatility of requirements**. Although requirements were set from the beginning, goals like memory reclamation were initially included in the implementation schedule. After the start of the project, however, investigation concluded that it was a non-trivial feature and required further theoretical considerations. The theory details were figured out, but its implementation was deliberately set aside.
- **Instability of software dependencies**. The function-passing model depends on two previous veins of work: Scala Pickling (Miller et al.,

2013) and (Miller, Haller, and Odersky, 2014). These research projects were relatively young and considered to be stable. However, the complexity of the addressed problems introduced unexpected bugs that were discovered and fixed during the execution of the implementation, as described in Section 4.3.

- **Discovery of gaps in the theory that make the implementation unpractical**. For instance, as explained in Section 4.5, the static serialization of messages went through different implementation attempts that eventually showed its unfeasibility. Consequently, the model took another different approach.

Unlike most of the traditional industry software, research needs an even higher level of flexibility: additional requirements are discovered during the ongoing development, and original ideas to tackle fundamental problems turn out to be unfeasible in practice. This is not a consequence of an ill definition of requirements, but an intrinsic risk of any research project. Independent of the quality of their definition, the degree of predictability is lower. As a result, both theory and practice have to happen together, confirming each other through experimentation. This is known as empiricism, an empirical process control theory that is the basis for the Scrum theory (Sutherland and Schwaber, 2013).

In short, a customary Scrum technique worked flawlessly for such kind of project. It is an effective method for projects with tight timelines, changing requirements, and business criticality (Pressman and Maxim, 2015). Other agile alternatives were ruled out based on the work environment, as in the case of XP (Extreme Programming). The next section provides a thorough explanation of the used methodology.

3.1.1 Scrum

In a nutshell, Scrum is a framework for developing and sustaining complex products. It helps address complex adaptive problems, while productively and creatively delivering products of the highest value possible (Sutherland and Schwaber, 2013).

With it, the function-passing model makes use of a slight variation of Scrum that combines it with Kanban, a technique that promotes a better task management. Also, the roles and meetings section differs from a conventional Scrum approach.

Scrum has three major components: teams, sprints and scrum meetings. These three combined result in an easy iterative workflow, which is the essence of the model. The workflow is depicted in figure 3.1.

The Scrum Team

The Scrum team consists of the *Product Owner*, the *Development Team*, and the *Scrum Master*. By definition, teams are self-organizing and cross-functional, that is, they are experts that know how to accomplish their work

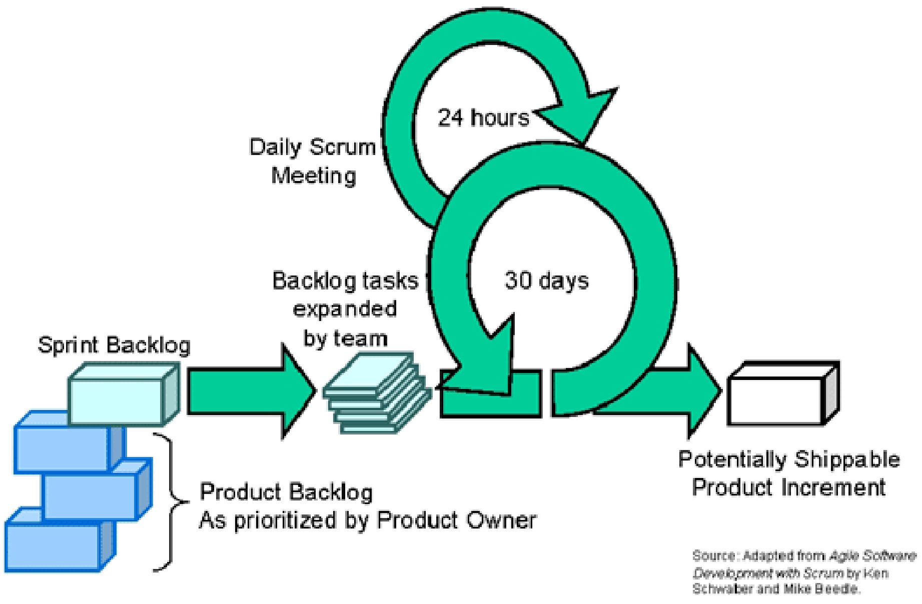

24 hours
Daily Scrum Meeting
30 days
Backlog tasks expanded by team
Sprint Backlog
Product Backlog
As prioritized by Product Owner
Potentially Shippable Product Increment
Source: Adapted from *Agile Software Development with Scrum* by Ken Schwaber and Mike Beedle.

FIGURE 3.1: Example of a normal Scrum workflow, from (Schwaber, 2002).

without the intervention of third parties. The general idea is to deliver products by iterations, maximizing the feedback.

The roles of the Scrum team are the following:

- **Product Owner**: Heather Miller, the original creator of the function-passing model and first author of the projects *Scala Pickling* and *Spores* (safe closures). She was in charge of ensuring a positive outcome out of the project by supervising it directly.
- **Scrum Master**: Heather Miller. As Scrum master, she was also the project manager of the function-passing implementation, and helped the development team to organise the project in manageable chunks of work.
- **Development Team**: Jorge Vicente Cantero. As the only component of the development team, he was in change of developing the implementation, as well as contributing theoretical improvements to the theory. My concrete contributions are explained in Chapter 4.

Heather Miller, as the *Product Owner* and the *Scrum Master*, was able to follow closely the evolution of the implementation, make meet the deadlines and ensure the correct execution of the methodology. Her vision of the final result and previous knowledge of all the technology stack placed her in a favorable position to orchestrate the entire project.

Sprints

Sprints are independent, self-contained periods of time in which the product is developed. As the basic time unit of any project, it comprises several tasks

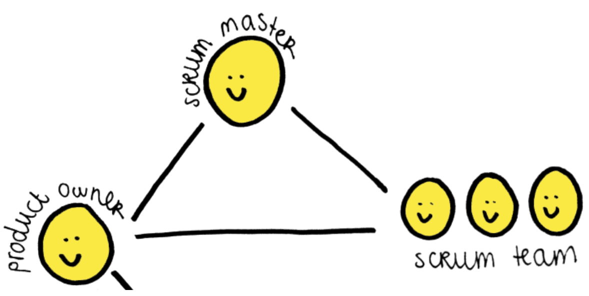


FIGURE 3.2: Roles in the Scrum Team, from (Vashishtha, 2012).

that achieve a concrete goal towards the desired software implementation. It is meant to iterate over the functionality and produce concrete, working prototypes that can be turned in to the product owner and receive appropiate feedback.

Sprints usually last one month and run sequentially. Their length can vary, although it is customary to keep the duration as is. The time requirements for this thesis set the duration of the project to a total amount of 300 working hours. Therefore, to comply with the timeline, adjustments were applied to both the length and the quantity of sprints. These time constraints eventually resulted in one sprint every 12 days with a workload of 5 hours. That added up to a total of 4 sprints.

Scrum Meetings

While a correct scheduling of the sprints is indispensable, Scrum meetings are an important element of the methodology and improve the control over the project evolution.

Traditionally, there are two types of meetings: *daily* meetings that give day-to-day feedback on the status of the tasks under development, and *sprint review* meetings, also known as *montly* meetings, that report on the goals achieved in every sprint (Sutherland and Schwaber, 2013).

As for the meetings, they were similarly held after each sprint. The daily meetings, however, underwent some changes. A weekly meeting was carried out during one or two hours, and there was a summary of the achieved goals and difficulties almost every day. Unlike the *sprint review* meetings, these were more fast and informal. When no progress was done, in cases where the complexity of the tasks was high and required time, they were skipped.

3.1.2 Kanban

Kanban (Monden, 1983) is a scheduling system for lean manufacturing and just-in-time manufacturing (*Kanban*). In short, it is widely used in the software industry to accurately organise the product backlog, coordinate the team members and show the overall status of the project in one board.

3.2 Requirement Analysis

A careful requirement analysis sets the scope, studies the dependencies between features, and helps prioritise the critical functionalities. This process usually works as follows: (1) elicits requirements, (2) analyzes them, and (3) documents them, translating them into concrete, measurable goals.

The requirements elicitation is the collection of all the requirements of a system from the product owner and the users,[1] *i.e.* distributed systems builders. Whereas most of them were specified by a previous draft of the function-passing model paper,[2] others were discovered during the course of this analysis and extracted from interviews with the supervisor.

Once requirements are set and analyzed, the next step is to document them. These specifications provide actionable items to later phases of the project. They are self-contained and explicit, usually stated in a constrained structure and with a finite set of technical words. Requirements can be documented in several ways, from use case diagrams to process specifications. In our case, user stories are the employed format. Following the agile mindset (Beck et al., 2001), user stories are, we believe, a perfect way to convey the essence of the requirements. Widely used in industry, they are a perfect fit for those who prefer simplicity over exhaustivity.

User Stories User stories are particularly useful for testing purposes. They don't only document requirements, but validate them. Their structure is explained in Figure 3.3. By convention, they are worded in the following fixed structure *'As a <role>, I want <a goal/desire> so that <benefit>'*. They are identified by unique numbers and have a more detailed description, time and priority estimates[3].

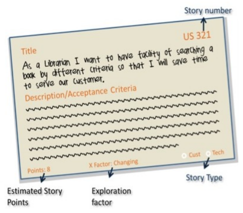

Story number
US 321
Title
As a Librarian I want to have facility of searching a book by different criteria so that I will save time to serve our customer.
Description/Acceptance Criteria
Points: 8
X Factor: Changing
Cust
Tech
Estimated Story Points
Exploration factor
Story Type

FIGURE 3.3: Structure of a user story, from (Yodiz, 2012).

[1]Any reference to stakeholders is consciously ommitted. As a research project, the stakeholders are not clear and well defined. Intuitively, they would be the whole Scala community, as well as distributed systems' builders, but they haven't officially participated in this analysis. Previous contacts with popular companies like Databricks showed positive feedback on the proposed model.

[2]The current version of the paper is (Miller et al., 2016) and it will be formally presented to the public in the Onward 2016 conference.

[3]Time estimates are often represented in terms of story points.

3.2.1 Functional Requirements

The user stories are specified as follows:

#1 Story points 6

As a user of the API, I want to create remote silos so that I can store my data.

Priority: 100

#2 Story points 6

As a user of the API, I want to map over any silo so that I can transform the stored data.

Priority: 90

#3 Story points 6

As a user of the API, I want to flat map over any silo so that I can transform the stored data.

Priority: 90

#4 Story points 2

As a user of the API, I want to extend the fault-tolerance mechanism so that I can express my own busines logic.

Priority: 80

#5 Story points 4

As a user of the API, I want to plug in any network backend so that I can choose how nodes are connected across networks.

Priority: 60

#6 Story points 2

As a user of the API, I want to implement higher-order functions based on map and flatmap so that I can build other interfaces atop of it.

Priority: 80

#7 Story points 4

As a user of the API, I want to cache silos so that they are reused by subsequent transformations.

Priority: 70

#8 Story points 6

As a user of the API, I want to asynchronously send transformations so that I can build my own event-based framework.

Priority: 90

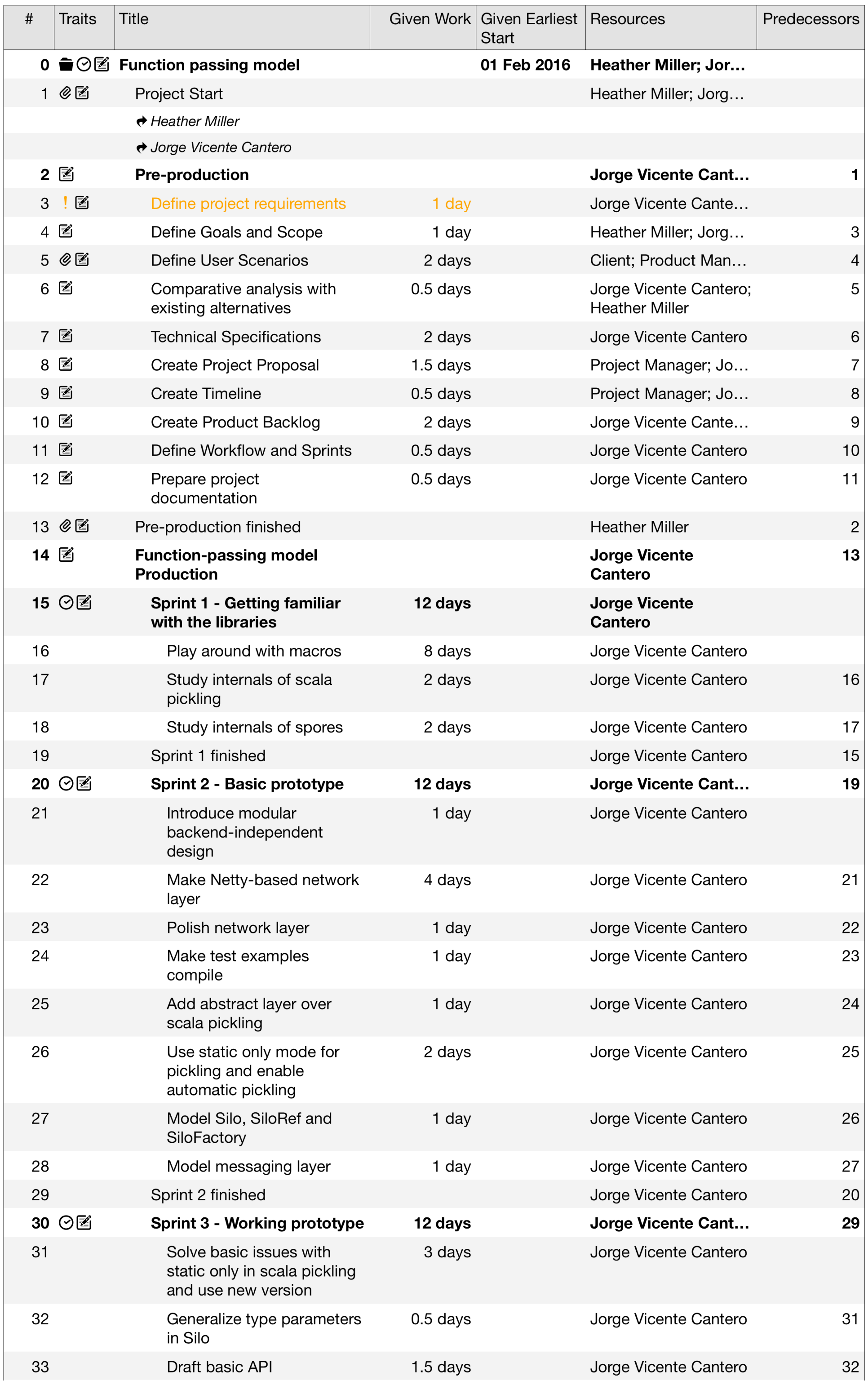
The function-passing model is a middleware. As such, it provides a high-level API to other software developers that want to build distributed data-oriented systems. User stories are good at depicting the interaction of the users and the product—such interaction is accurate and abstracted. But, when it comes to the design of frameworks, the level of precision vanishes. For instance, let's take a simple user story like #2. It states one concrete behaviour of the system, but for this feature to work several layers of the framework need to be carefully built, going from the network backend to the final user interface, and passing through other several layers like the fault-tolerance layer and stubs[4]. To solve this issue, a new level of indirection is required.

[4]The stub layers provide modularity and allow to plug in different software projects to satisfy a particular need.

In order to make the development process easier, these tasks were subdivided into implementation-oriented tasks. This twofold process encompassed:

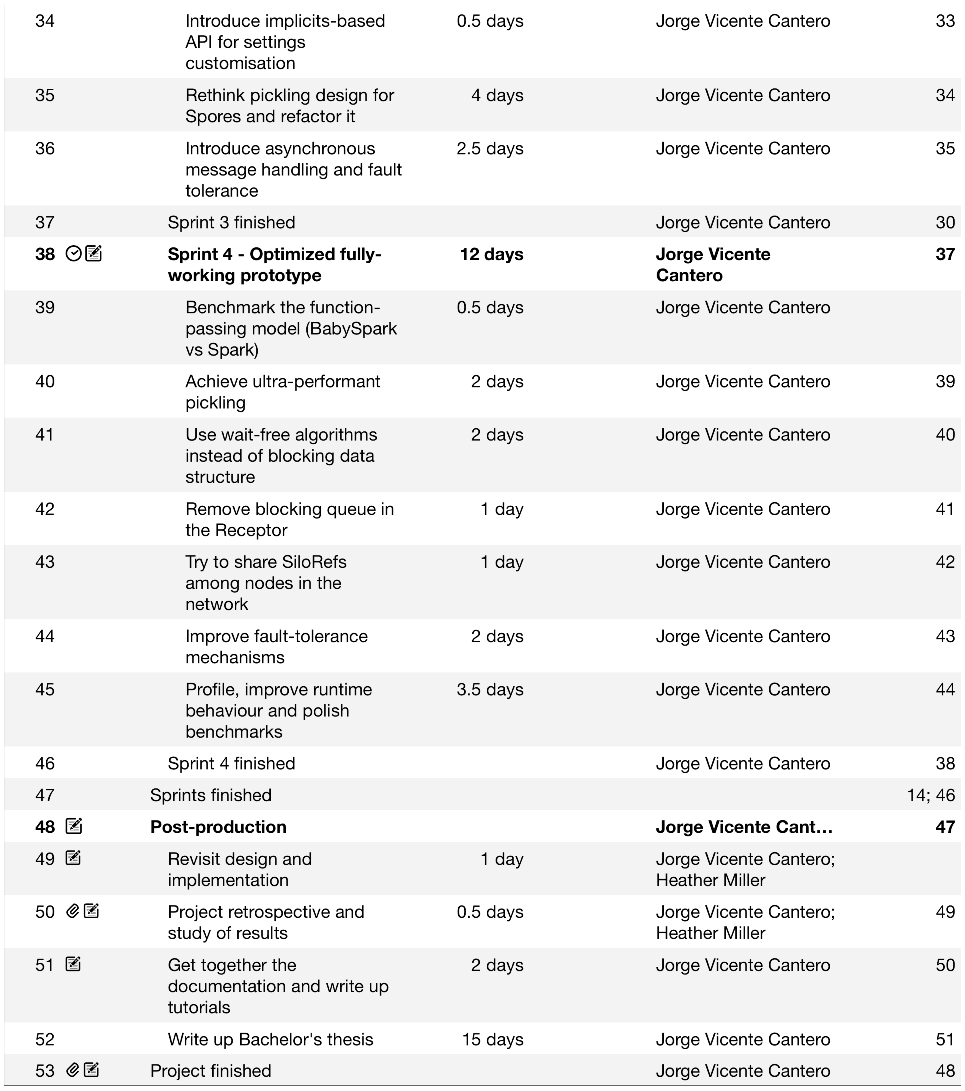
- A direct mapping from abstract user stories to the requirements of each layer of the framework.
- A prioritisation of the features of each layer based on the degree of dependency that the user stories required.

Figure 3.4 and 3.5 show the final outcome of the process. Tasks have predecessors, on which they depend, as well as time estimates. Tasks were executed sequentially since there was only one developer. The dependency graph, which was rather intricate, ruled out any possibility for asynchronous tasks.



#	Traits	Title	Given Work	Given Earliest Start	Resources	Predecessors
0	💼 ⏲ ✎	**Function passing model**		**01 Feb 2016**	**Heather Miller; Jor…**	
1	📎 ✎	Project Start			Heather Miller; Jorg…	
		➦ *Heather Miller*				
		➦ *Jorge Vicente Cantero*				
2	✎	**Pre-production**			**Jorge Vicente Cant…**	**1**
3	! ✎	Define project requirements	1 day		Jorge Vicente Cante…	
4	✎	Define Goals and Scope	1 day		Heather Miller; Jorg…	3
5	📎 ✎	Define User Scenarios	2 days		Client; Product Man…	4
6	✎	Comparative analysis with existing alternatives	0.5 days		Jorge Vicente Cantero; Heather Miller	5
7	✎	Technical Specifications	2 days		Jorge Vicente Cantero	6
8	✎	Create Project Proposal	1.5 days		Project Manager; Jo…	7
9	✎	Create Timeline	0.5 days		Project Manager; Jo…	8
10	✎	Create Product Backlog	2 days		Jorge Vicente Cante…	9
11	✎	Define Workflow and Sprints	0.5 days		Jorge Vicente Cantero	10
12	✎	Prepare project documentation	0.5 days		Jorge Vicente Cantero	11
13	📎 ✎	Pre-production finished			Heather Miller	2
14	✎	**Function-passing model Production**			**Jorge Vicente Cantero**	**13**
15	⏲ ✎	**Sprint 1 - Getting familiar with the libraries**	**12 days**		**Jorge Vicente Cantero**	
16		Play around with macros	8 days		Jorge Vicente Cantero	
17		Study internals of scala pickling	2 days		Jorge Vicente Cantero	16
18		Study internals of spores	2 days		Jorge Vicente Cantero	17
19		Sprint 1 finished			Jorge Vicente Cantero	15
20	⏲ ✎	**Sprint 2 - Basic prototype**	**12 days**		**Jorge Vicente Cant…**	**19**
21		Introduce modular backend-independent design	1 day		Jorge Vicente Cantero	
22		Make Netty-based network layer	4 days		Jorge Vicente Cantero	21
23		Polish network layer	1 day		Jorge Vicente Cantero	22
24		Make test examples compile	1 day		Jorge Vicente Cantero	23
25		Add abstract layer over scala pickling	1 day		Jorge Vicente Cantero	24
26		Use static only mode for pickling and enable automatic pickling	2 days		Jorge Vicente Cantero	25
27		Model Silo, SiloRef and SiloFactory	1 day		Jorge Vicente Cantero	26
28		Model messaging layer	1 day		Jorge Vicente Cantero	27
29		Sprint 2 finished			Jorge Vicente Cantero	20
30	⏲ ✎	**Sprint 3 - Working prototype**	**12 days**		**Jorge Vicente Cant…**	**29**
31		Solve basic issues with static only in scala pickling and use new version	3 days		Jorge Vicente Cantero	
32		Generalize type parameters in Silo	0.5 days		Jorge Vicente Cantero	31
33		Draft basic API	1.5 days		Jorge Vicente Cantero	32

FIGURE 3.4: View of the implementation tasks 1-33.



34		Introduce implicits-based API for settings customisation	0.5 days	Jorge Vicente Cantero	33
35		Rethink pickling design for Spores and refactor it	4 days	Jorge Vicente Cantero	34
36		Introduce asynchronous message handling and fault tolerance	2.5 days	Jorge Vicente Cantero	35
37		Sprint 3 finished		Jorge Vicente Cantero	30
38	🕒 ✎	**Sprint 4 - Optimized fully-working prototype**	**12 days**	**Jorge Vicente Cantero**	**37**
39		Benchmark the function-passing model (BabySpark vs Spark)	0.5 days	Jorge Vicente Cantero	
40		Achieve ultra-performant pickling	2 days	Jorge Vicente Cantero	39
41		Use wait-free algorithms instead of blocking data structure	2 days	Jorge Vicente Cantero	40
42		Remove blocking queue in the Receptor	1 day	Jorge Vicente Cantero	41
43		Try to share SiloRefs among nodes in the network	1 day	Jorge Vicente Cantero	42
44		Improve fault-tolerance mechanisms	2 days	Jorge Vicente Cantero	43
45		Profile, improve runtime behaviour and polish benchmarks	3.5 days	Jorge Vicente Cantero	44
46		Sprint 4 finished		Jorge Vicente Cantero	38
47		Sprints finished			14; 46
48	✎	**Post-production**		**Jorge Vicente Cant…**	**47**
49	✎	Revisit design and implementation	1 day	Jorge Vicente Cantero; Heather Miller	
50	📎 ✎	Project retrospective and study of results	0.5 days	Jorge Vicente Cantero; Heather Miller	49
51	✎	Get together the documentation and write up tutorials	2 days	Jorge Vicente Cantero	50
52		Write up Bachelor's thesis	15 days	Jorge Vicente Cantero	51
53	📎 ✎	Project finished		Jorge Vicente Cantero	48

Figure 3.5: View of the implementation tasks 34-53.

3.2.2 Non-functional Requirements

Non-functional requirements are the quality attributes of the function-passing model, and describe non-behavioural requirements. They summarize the strong points of the final implementation and focus on their properties.

At the core of the system, it lies the idea that frameworks built atop of the function-passing model have more benefits than their counterparts.[5] Therefore, it aims at achieving the following properties:

1. **Performance**. The difference between rolling-your-own-framework strategy and leveraging the function-passing model has to be noticeable, and significantly faster.

[5]The counterparts are the current frameworks working in production that roll their own implementations to support only one use case: passing closures between distributed networks in a fault-tolerant, reliable way.

2. **Fault tolerance**. Any distributed system builder knows that fault tolerance is difficult. The function-passing model addresses this issue by design—it provides extensible ways to enrich runtime behaviours when failures occur.
3. **Modularity**. End-users have specific requirements when building their products. These needs change depending on the technology stack that is used. In order to satisfy any potential users, they must be able to make use of the proposed model over any network backend.
4. **Mantainability**. Non-maintainable products are not able to keep up with the time. By building atop the model, frameworks don't need to implement their own solutions to solve a general problem. Consequently, codebases shrink and are simplified.

The aforementioned properties are explained in detail in Chapter 4, that discusses the implementation goals and how they are achieved.

3.3 Project Schedule

Research projects need special attention from a software engineering perspective, as pointed out in 3.1. The same argument applies for the project schedule—it is impossible to make objective, predictable time estimates over the whole duration of the project. A new procedure to schedule the project is necessary.

Based on the Scrum methodology, it's possible to measure the time of the project by adding the story points of every user story, and having a time estimate for every story point. Such information, which usually is backed up by real data of previous projects, gives a hint on the overall amount of time required to build the framework. The lack of data results in a pessimistic estimation: 5 hours per story point.

Thereby, a sensible time estimate is 180 hours ($5\ \textit{hours/story point} \cdot 36\ \textit{story points} = 180\ \textit{hours}$), without considering the time for getting familiar with the ecosystem and libraries. As a sprint is 12 days long, *i.e.* 48 hours, there is still time for an additional sprint. Below, we synthesize the general tasks and ideas behind each sprint. A complete description of the carried out tasks are is shown in Figures 3.6, 3.7 and 3.8.

Sprint 1 – Getting ready As the first sprint, the goal is to introduce the technology stack and, in particular, compile-time metaprogramming in *Scala*, which lay the foundations of *Scala Pickling* and *Spores*. Not only restricted to an in-depth study of both frameworks, this sprint involves experimentation with the *Scala and Java Reflection APIs*, the internals of the JVM and the GC (Lindholm et al., 2013; Bloom, 2013; Microsystems, 2006; Systems, 2012).[6]

[6]Previous knowledge about the JVM is critical with regard to building a performant framework, and even more important for frameworks whose internals make use of reflection. Understanding how the JVM facilitates such operations is key to avoid endless debugging sessions. As a side note, reflection in the function-passing model would be disabled by default, but scala-pickling still needs to deal with it.

Sprint 2 – Basic prototype The second sprint aims at building the network primitives, modularising the network layer and performing a quick modeling of `Silos` and `SiloRefs`. The expected output is a proof of concept that illustrates the feasibility of the function-passing model atop of Netty (Netty, 2011).

Sprint 3 – Working prototype At this point, the frameworks is in a rough state. Yet, some key features are missing, fault-tolerance is flaky and performance is bad. The working prototype turns the proof of concept into a more robust and rich framework, generalizes the use cases and starts to draft the final API.

Sprint 4 – Optimizing the model While the working prototype suits the needs of most users, advanced developers that need to build high-performance scalable frameworks wouldn't be happy with the current implementation of the programming model. Hence, this sprint polishes the model, the interfaces, the fault-tolerance primitives and dramatically improves serialization.

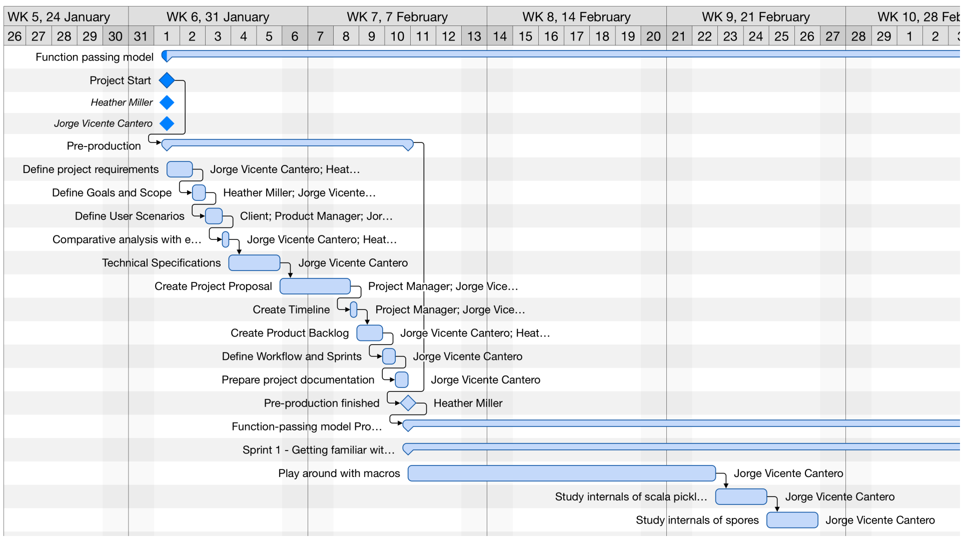


Figure 3.6: Gantt chart that shows the start of the project and the schedule for sprint 1.

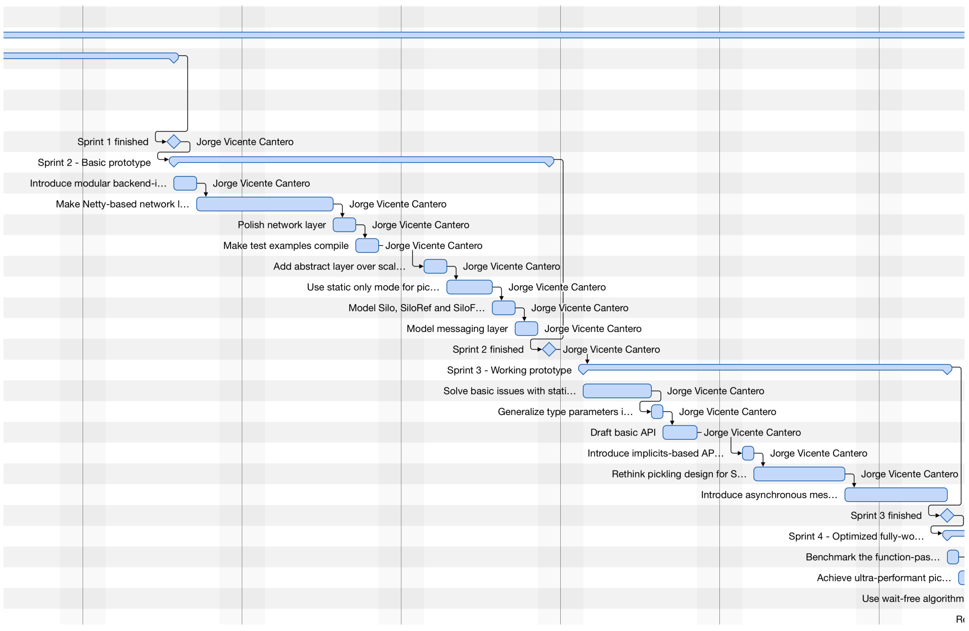


Figure 3.7: Gantt chart that shows the schedule for sprints 2 and 3.

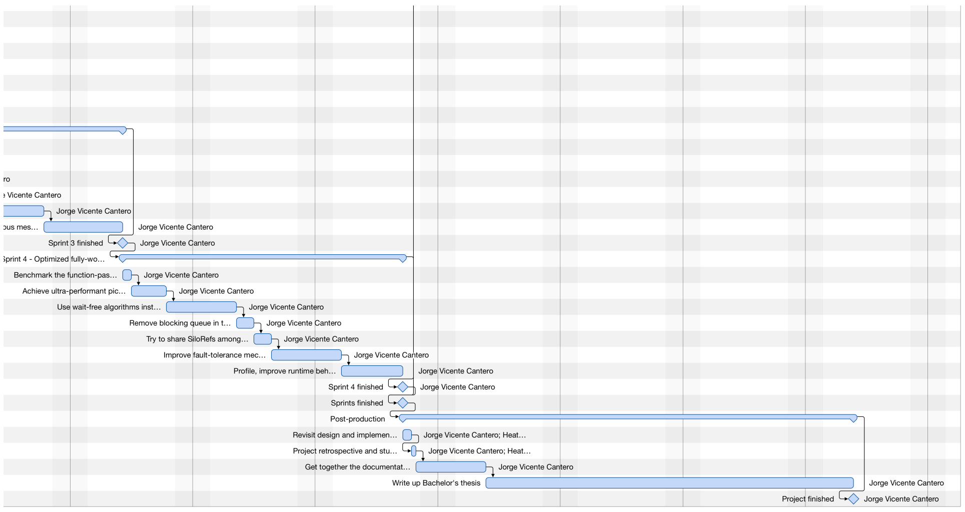


Figure 3.8: Gantt chart that shows the schedule for the last sprint.

Chapter 4

The Function-Passing Implementation

The presented programming model has been fully implemented in Scala, a functional programming language that runs on both JVMs and JavaScript runtimes. The function passing model is compiled and run using Scala 2.11.8. The implementation, which has been published as an open-source project,[1] runs currently only on the JVM[2] and builds on two main Scala extensions: Scala Pickling (Miller et al., 2013)[3] and Spores (Miller, Haller, and Odersky, 2014).[4] Along this chapter, we focus on high-level concepts and the improvements performed in the aforementioned dependencies, not on hidden and specific difficulties of the implementation.

These two projects are the main pillars of the implementation. It turns out that binding them efficiently involves more intricacies than one would expect, due to the addititional difficulty of reusing two **macro-based compiler extensions**.[5] Such binding accounts for a large amount of the work to get the function-passing model working correctly. The effort to bind both projects is explained in detail below.

Additionally, the implementation required external changes and improvements on Spores and Scala Pickling. In the next subsections, we'll discuss the main contributions and detail our solutions.

4.1 Goals of the Implementation

The design of the framework has been guided by the following principles:

[1]https://github.com/jvican/function-passing.

[2]The implementation is theoretically compatible with Scala.js as it avoids the use of all the missing features in JavaScript environments. However, we don't provide a working implementation for JavaScript since Netty, the network backend, makes a wide use of reflection and, thus, it's not Scala.js compatible. Third parties are invited to contribute a network backend for JavaScript.

[3]`https://github.com/scala/pickling`

[4]`https://github.com/heathermiller/spores`

[5]To fully understand how they work, we recommend having a look at Appendix A and Appendix B.

Simplicity and ease of use To encourage potential users to employ the programming model, public interfaces are kept simple troughout the implementation. Concretely, we opt for providing sane defaults settings that ease the basic users' workflow, while we allow advanced users to override them using implicits. This results in a flexible, powerful framework that can carry out high-level tasks, while still allowing advanced users to benefit from a more low-level approach that enables them to roll out their own frameworks.

Backend decoupling The function-passing model uses an abstraction over network connections. Ultimately, users take the crucial decisions about the internals of their systems based on their particular needs. Whether they build upon the model in existing codebases or from scratch, a clear separation between frontend and backend facilitate potential users to switch backends and use a concrete technology stack. Teams already settled on concrete technologies would likely want to reuse them. Modularity comes at a price: it entails limiting the number of fixed dependencies of the project, to make it lightweight and easy to plug in.

With this in mind, the implementation decouples the network layer from the higher-level layers. In the default implementation, the Netty network framework (Netty, 2011) is the official backend. Third parties are given the possibility of building their own network layer for other backends (*e.g.* be it Akka or their own).

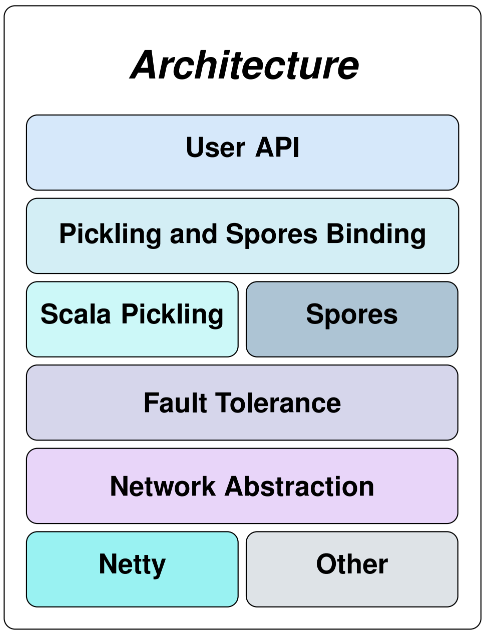


FIGURE 4.1: Architecture that show the layers of the function-passing implementation.

Fault tolerance The function-passing model is fault-tolerant by design. In case anything goes wrong, lost data is recovered by recomputation. Yet, it is possible for the implementation to provide more fault-tolerant guarantees by taking care of network and power errors. To face them, the implementation provides its own network protocol, that protects the system from spurious failed deliveries of messages, and a persistence layer, that keeps track of the state at any given moment and allows to recover from it.

Performance The programming model is aimed at laying the groundwork for building high-performance applications. Importants aspects that deeply affect performance are the speed of the network layer and serialization/deserialization of messages. Thus, we achieve performance by two means: (1) we build on top of Netty, one of the most popular asynchronous event-driven network application framework on the JVM, and (2) we use Scala Pickling, a performant serialization framework that outperforms the Java Serialization library (Carpenter et al., 1999) and other popular projects like Avro (Apache, 2015a).

4.2 Background

4.2.1 Scala Pickling

Scala Pickling is a type-safe and performant serialization framework based on object-oriented pickler combinators which (a) enables retrofitting existing types with pickling support, (b) supports automatically generating picklers at compile time and at runtime, (c) supports pluggable pickle formats, and (d) does not require changes to the host language or the underlying virtual machine. The function passing implementation benefits from the maturity of the project, which supports pickling/unpickling a wide range of Scala type constructors. Pickling has evolved from a research prototype to a production-ready serialization framework that is now in widespread commercial use.

Furthermore, it is extensible in several important ways. First, building on an object-oriented type-class-like mechanism (S. Oliveira, Moors, and Odersky, 2010), this approach enables retroactively adding pickling support to existing, unmodified types. Second, it provides pluggable pickle formats which guarantee that type-specialized picklers are portable and carry over to different pickle formats.

Among other, it sports the following properties:

- **Ease of use**. Simplified programming interface that aims to minimise boilerplate, alike other more mainstream libraries like Java's serialization framework (Carpenter et al., 1999).
- **Performance**. The generated picklers are efficient, enabling their use in high-performance distributed systems, and allow both static and runtime pickling.
- **Extensibility**. The design of the framework allows type-class-like extensibility through implicits, enabling pickler definition in third-party libraries and easing the process of supporting types retroactively.

- **Pluggable Pickle Formats**. Scala Pickling allows to swap target pickle formats, or for users to provide their own customized format (via typeclasses in Scala). By default, it allows binary and json formats.
- **Type safety**. Picklers are type safe through (a) type specialization and (b) dynamic type checks when unpickling to transition unpickled objects into the statically-typed "world" at a well-defined program point.
- **Robust support for object-orientation**. Concepts such as subtyping and mix-in composition, widely used in Scala, are also supported.

The Basics

Scala Pickling was designed so as to require as little boilerplate from the programmer as possible. For that reason, pickling or unpickling an object `obj` of type `Obj` requires simply:

```
import scala.pickling._
val pickled = obj.pickle
val obj2 = pickle.unpickle[Obj]
```

Here, the `import` statement imports scala/pickling, the method `pickle` triggers static pickler generation, and the method `unpickle` triggers static unpickler generation, where `unpickle` is parameterized on `obj`'s precise type `Obj`. Note that not every type has a `pickle` method; it is implemented as an extension method using an *implicit conversion*. This implicit conversion is imported into scope as a member of the `scala.pickling` package.

Optionally, a user can import a `PickleFormat`. By default, the framework provides a Scala Binary Format, an efficient representation based on arrays of bytes, though the framework provides other formats which can easily be imported, including a JSON format. Furthermore, users can easily extend the framework by providing their own `PickleFormats`. For more information, refer to the original paper (Miller et al., 2013).

Appendix B explains macros and provides an example that illustrates the basic principles of the Scala Pickling implementation.

4.2.2 Spores

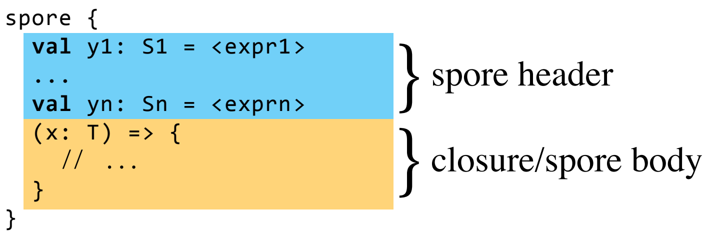
The Basics

Spores (Miller, Haller, and Odersky, 2014) are safe closures that are guaranteed to be serializable and thus distributable. They are a closure-like abstraction and type system which gives authors of distributed frameworks a principled way of controlling the environment which a closure (provided by client code) can capture. This is achieved by (a) enforcing a specific syntactic shape which dictates how the environment of a spore is declared, and (b) providing additional type-checking to ensure that types being captured have certain properties.

A spore consists of two parts:

- **the spore header**, composed of a list of value definitions.
- **the spore body** (sometimes referred to as the "spore closure"), a regular closure.

This shape is illustrated below.



```
spore {
  val y1: S1 = <expr1>
  ...
  val yn: Sn = <exprn>
  (x: T) => {
    // ...
  }
}
```

The characteristic property of a spore is that the spore body is only allowed to access its parameter, the values in the spore header, as well as top-level singleton objects (Scala's form of modules). The spore closure is not allowed to capture variables other than those declared in the spore header (*i.e.* a spore may not capture variables in the environment). By enforcing this shape, the environment of a spore is always declared explicitly in the spore header, which avoids accidentally capturing problematic references. Moreover, importantly for object-oriented languages like Scala, it's no longer possible to accidentally capture the `this` reference.

Spores also come with additional type-checking. Type information corresponding to captured variables are included in the type of a spore. This enables authors of distributed frameworks to customize type-checking of spores to, for example, *exclude* a certain type from being captured by user-provided spores. Authors of distributed frameworks may kick on this type-checking by simply including information about excluded types (or other type-based properties) in the signature of a method. A concrete example would be to ensure that the `map` method on `RDD`s in Spark (a distributed collection) accepts only spores which do not capture `SparkContext` (a non-serializable internal framework class).

The Benefits of Pickling Spores

In order to understand the pickling process, let's look at the arity-1 spore type signature:

```
trait Spore[-A, +B] extends Function1[A, B] {
    type Captured
    type Excluded
}
```

Spores keep the same semantics as regular functions. Therefore, spores are also defined contravariant in their argument type `A` and covariant in their result type `B`. Also, the `Spore` trait doesn't define any extra method; a concrete spore would override the abstract `apply()` method defined in the `Function1` trait. Nevertheless, spores enrich the function type signature by adding two new type members: `Captured` and `Excluded`. These are known as type refinements and enforce a certain shape of the spore. Concretely, the `Captured` type member will contain the types of each captured variable. These types allow users to match concrete spore types. For instance, any function from `T` to `S` that uses two environment variables of type `String` and `Int`, matches `Spore[T, S] { type Captured = (String, Int) }`. Note that captured types are stored in left-biased tuples and that there's not size limitation; capturing one more environment variable, a `String`, would result in ((String, Int), String).

To ensure safe and efficient distribution of closures, the model leverages both syntactic and type-based restrictions. For instance, closures sent to remote machines are required to conform to the restrictions imposed by the so-called "spore" abstraction. Among others, the syntax and static semantics of spores guarantees the absence of runtime serialization errors due to closure environments that are not serializable. Since spores are guaranteed to capture only the defined variables in the spore header, checking the proper functioning of the serialization code is performed at compile-time based on the type of the captured variables, unlike other distributed systems whose serialization errors occur at runtime and functions can capture any environment variable. In this case, the compiler looks for implicit instances of `Picklers` and `Unpicklers` of a spore, that in turn looks for the same instances for every captured type, failing if there's none. Note that the implicit search is triggered by the full spore type, which is enriched with the captured types and excluded types.

Besides, spores provide another advantage over conventional distributed functions—they improve error reporting. In case the serialization of a captured variable is not possible, the user receives rich feedback at compile-time, which pinpoints the concrete issues and the types involved in them. This feature eliminates the hassle of figuring out runtime serialization errors, which are typical of state-of-the-art distributed systems (Miller, Haller, and Odersky, 2014), and whose error reports lack concrete type information and hinders problem diagnosis and debugging.

Spores and Stable Paths

The body of spores can only reference to stable paths. These are expressions that contain selections and identifiers and for which each selected entity is stable. For instance, spores can be defined by object definitions or by value definition of non-volatile types. A spore that references to a term declared inside a class is not in a stable path—its accessibility depends on the class instance.

```scala
// Spore references to a stable path, compiles
val sp: Spore2[Int, Int, Int] = spore {
  (x: Int, y: Int) => Math.abs(x + y)
}

// Spore references to a non-stable path, doesn't compile
val sp: Spore2[Int, Int, Int] = spore {
  (x: Int, y: Int) => mathUtils.abs(x + y)
}
```

For a deeper understanding of spores, see the corresponding publication (Miller, Haller, and Odersky, 2014).

4.3 Extensions to Scala Pickling

The function-passing model implementation makes extensive and advanced use of the Scala Pickling framework. Throughout the intensive development of the model, I discovered and worked around several shortcomings in Scala Pickling, such as:

4.3.1 Unnecessary Allocation Overhead of Picklers

Problem The absence of implicit picklers in the scope of the call-site impedes the sharing of picklers. Therefore, since no implicit is present, Scala Pickling generates picklers and unpicklers per call-site. At runtime, the program triggers the initialization of every independent pickler and unpickler (which in turns initialises inner picklers and unpicklers), leaving a non-negligible footprint in performance.

Solution The root of this problem lays in the end-users since they interact directly with the function-passing API. Every operation of such interface requires implicit instances that, if not in scope, will be generated at the call sites. Suppose the following program:

```
case class User(id: String, name: String, age: Int, sex: Boolean)

// Defines a reference to a silo of users
val users: SiloRef[Vector[User]] = ...

val adults = users.map(spore {
  (us: Vector[User]) =>
    us.filter(u => u.age >= 18)
})

val women = adults.map(spore {
  (us: Vector[User]) =>
    us.filter(u => u.sex == 1)
})
```

Every method of `SiloRef` demands an implicit instance of pickler and unpickler for the function passed as an argument. The spore type signature is the same for the two map operations: `Spore[Vector[User], Vector[User]]`. At the macro expansion, both pickler and unpickler for that spore are generated, but **twice**. By nature, macro expansions happen locally in the place where they're run and cannot share access to a top-level element and, consequently, are unable to share implicit instances. Implementing this feature in macros involves dealing with a lot of compiler issues, specifically related to incremental compilation.[6] This limitation, which is exemplified in Appendix B provokes independent pickler and unpickler generation, thus an impact in performance: longer compilation times (the macro generator is executed per call site) and a hidden cost of object initialization.

Shrewd users can solve this problem by caching picklers themselves:

```
// Define an object containing the implicits
object CachedPicklers {
  type CachedSpore = Spore[Vector[User], Vector[user]]
  implicit val ps = implicitly[Pickler[CachedSpore]]
  implicit val us = implicitly[Unpickler[CachedSpore]]
}

// Import them in the scope of your call sites
import CachedPicklers._
```

Nevertheless, the implementation cannot rely on its users' skills. As we cannot put an end to this problem once and for all, the solution aims at reducing its impact. We address the issue of object initialization by caching and checking whether a pickler/unpickler for a concrete type has already been used. If so, that one is used. Otherwise, it gets initialised.

4.3.2 Static Pickler Generation

Problem In certain cases, Scala Pickling reports the correct generation of static picklers, when instead it silently fallbacks to reflection. That is, the generated picklers will introspect the runtime classes to find out their internal structure. This poses a major problem: uncertainty, which prevent users from reasoning about deterministic behaviors in their programs. Further, runtime generated picklers hurt performance and interoperability with JavaScript. Reflection poses a compatibility problem with JavaScript because most of the Java Reflection API is not supported in Scala.js (Doeraene, 2015). Although this doesn't mean that every use of reflection is swept away, the reality is that only a constrained subset of it is possible.

Solution To achieve these guarantees, some parts of the code generator had to be rewritten from scratch. These refactorings involved: (a) clearly separating the logic of the code generator into independent, easy-to-mantain modules (improving the macro structure), and (b) disabling runtime generation via the use of the `staticOnly` implicit. This value, when imported in the scope,[7] changes the generator backend and disables any kind of runtime behavior.

With regard to the reflection and Scala.js support, `scalajs-reflect` is used, a linker plugin that offers key functionality like lookup by class name. Reducing the use of the Reflection API to a bare minimum has made Scala Pickling a Scala.js-friendly project.

[6]It is unclear if upcoming macro support for the next versions of Scala will include this feature

[7]The static flag needs to be imported in all the scopes that make use of Scala Pickling.

4.3.3 Long Compilation Times

Problem Compilation times are an important and, often overlooked, aspect of any programming language. In the proposed programming model, compilation times increased significantly because implicits recursively trigger macro expansions that, in turn, execute an implicit search for every pickler/unpickler generation. As implicit resolution is expensive, a significant amount of time is spent on it. Further inspection of the issue revealed that there was room for improvement on the way Scala Pickling chained the picklers through implicits.

In order to introduce these improvements, we use a running example that goes through all the steps of the implicit search based on the type of a spore. The following snippet of code

```
implicitly[Pickler[Spore[Vector[Int], Int]]]
```

will sequentially look up the implicits as described in Figure 4.2.

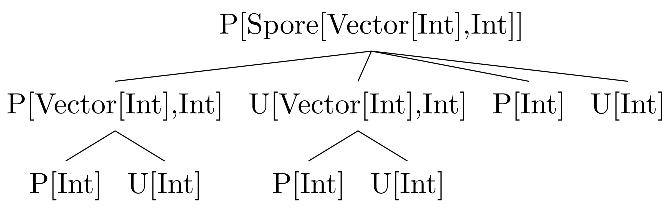


FIGURE 4.2: Tree describing how the implicit search is performed and illustrating dependencies among picklers and unpicklers for a concrete case. Note that `Pickler` has been shortened to `P` and `Unpickler` to `U`.

As we observe, some implicit searches are repeated several times, although the compiler has already found them before (as in the case of `Pickler[Int]` and `Unpickler[Int]`). Note that, despite looking up the implicit for a `Pickler`, `Unpicklers` are also looked up. This is due to the fact that picklers and unpicklers are generated together[8] and realized into a new class of type `PU`, defined as: `abstract class PU[T] extends Pickler[T] with Unpickler[T]`. To generate an unpickler for the spore, a proof that the type parameters are unpicklable is required.[9] The problem gets even worse when we realise that the above implicit search **will be duplicated** for the implicit search of `Unpickler[Spore[Vector[Int], Int]]`, that will repeat the same search in Figure 4.2. As a consequence, `P[Int]` will be looked up 8 times. It follows that the hidden cost of the implicit search is exponential in the depth of the type parameters and type members.

Solution The two proposed changes to address these issues are:

[8]Picklers and unpicklers are generated together because this helps performance and cuts off half of the initialization of pickling objects throughout the program lifetime.

[9]It is useful to imagine implicits as a way to provide proofs to the compiler about what a type can do. In this case, it's reasonable to fail compilation if we ask for the generation of an unpickler of a type `T[S]` where `T` has an instance of `Unpickler` but `S` doesn't, which means that `S` cannot be unpickled.

- Carefully tweaking and reordering the default implicit values to make the implicit search shorter. Following the Scala specification language (Odersky, 2014), the implicit search has a notion of priorities, assigned depending on the location of each implicit.[10] Based on these priorities, the compiler chooses the highest-priority available implicit instance in scope. Ideally, the sooner the compiler finds this instance, the shorter the compilation times.

- Performing a concrete, tailor-made implicit search for picklers and unpicklers. Essentially, this algorithm reproduces the expensive nested implicit resolution, prefetching all the picklers and unpickler that every path of the implicit tree shown above needs.

  The fact that several implicit searches occur hurt compile-time performance dramatically. Working around this problem is, then, essential to speed up compilation times. The Scala compiler starts the implicit resolution in the same scope as the location where they are requested. The solution takes advantage of this fact, and places all the required picklers and unpicklers in the same scope so that the compiler immediately finds the requested implicits. The hit rate is 100%. The macro generates equivalent code to:

  ```
  // Used implicits instances
  implicitly[Pickler[Int]]
  implicitly[Unpickler[Int]]
  implicitly[Pickler[Vector[Int]]]
  implicitly[Unpickler[Vector[Int]]]

  // Pickler and Unpickler code for the spore type
  object SporePU extends PU[Spore[Vector[Int], Int]] { ... }
  ```

The results were promising for both approaches combined. We noticed an improvement in compilation times between 2x and 3x (worst-case scenario). As an example, the compilation of the Scala Pickling test suite (consisting of more than 350 tests) dropped from 500 to less than 200 seconds. Note that users can increase these speedups if they cache picklers as explained in section 4.3.1, which is a recommended practice.

4.3.4 Inability to Customize Runtime Behavior

Problem Lookup of picklers at runtime is a powerful feature that enables the re-use of already used picklers in distributed applications.[11] However, when no cached pickler is found, the default mechanism leverages reflection to invoke runtime generation.

Solution To still benefit from the caches but avoid reflection, we added the `HybridMode`, a pickling strategy that allows runtime pickler lookup but cuts down any type of runtime generation. We thoroughly explain the solution in section 4.3.8.

[10]The Scala linearization order plays a key role in the implicit search since it determines how the implicit resolution is run.

[11]The same happens for unpicklers.

4.3.5 Pickling and Unpickling Unknown Types

Problem In the function-passing topologies, sender and receiver exchange messages to produce a meaningful result. The sender, who drives the logic of the programs, has total information about the exchanged communication and knows what sends and what expects back as a response. Conversely, the receiver lacks this information and yet it must be able to communicate seamlessly with the client, that is, generic message types in the server required the use of `Any`, the top type in the Scala type system, whose correct serialization and deserialization turns out to be essential.

Solution Due to generic message types, `Any` has to be able to support most of the possible types.[12] Also, support for pickling `Any` was not provided. Before, there was not a principled approach to deal with all the possible cases. Nevertheless, improvements in its functioning were succesful in achieving so, at the cost of a minimum use of reflection. `AnyPicklerUnpickler` is only used for pickling, when the discovery of the class type is obtained via `getClass`. At that point, such type information is used for the discovery of cached picklers/unpicklers. In extreme cases where no cached picklers are available, it's theoretically possible to cache runtime generated picklers[13] for later use explained in section 4.3.4. However, enabling this option requires the end-user not to use the `HybridMode`. For unpickling generic messages, no reflection is used at all, instead using types extracted from the received messages. A full explanation of this situation and its solution is discussed in section 4.3.8.

4.3.6 Miscellaneous

During the development process, several misbehaviors of the framework have been reported, especifically in the domain of pickler generation, revealing some corner cases that were not handled by default. For instance, compiling

```
// Define B and declare a spore
case class B[S](value: S)
val s: Spore[T, B[S]] = ??? // undefined

// Generate picklers and unpickler
implicitly[Pickler[Spore[T, B[S]]]
implicitly[Unpickler[Spore[T, B[S]]]
```

fails despite Scala Pickling is capable of generating both pickler and unpickler for `B`. This error happens when runtime generation is enabled. If there is no pickler for a class `B`, wrapped by another class `W` (in this case, `Spore`), then a brand new pickler for `W[A]` is created from scratch, without reusing the static pickler available for `W`. Therefore, the generated picklers make use of reflection. Other minor issues are related to: inexistent support for value

[12] Structural types are not supported by default. By nature, they require runtime checks.

[13] As we explain in section 4.3.8 Scala.js supports a restricted subset of the Java Reflection API, making this possible.

classes, failed generation for case classes that receive implicit parameters, etcetera.

The encountered bugs have been filed in the Scala Pickling repository as the following issues: #426, #423, #422, #417, #399, #398, #397, #395, #391.

4.3.7 Serialization in the Presence of Existential Quantification

Initially, to serialize most message types exchanged by the network communication layer, runtime-based unpicklers had to be used (meaning unpickling code discovering the structure of a type through introspection at runtime). A major disadvantage of runtime-based unpickling is its significant impact on performance. The reason for its initial necessity was that message types are typically generic, but the generic type arguments are *existentially-quantified type variables* on the receiver's side. For example, the lineage of a SiloRef may contain instances of a type Mapped. This generic type has four type parameters. The receiver of a freshly unpickled Mapped instance typically uses a pattern match:

```
case mapped: Mapped[t, s] =>
```

The type arguments `u`, and `s` are *type variables*. While unknown, the static type of `mapped` is still useful for type-safety:

```
val newSilo = new LocalSilo[v, s](mapped.fun(value))
```

However, it is impossible to generate type-specific code to unpickle a type like `Mapped[t, s]` since they are not concrete types that scala pickling can generate picklers for. As a solution to this problem, we explore another alternative called "self-describing" pickles.

Self-describing picklers The idea is to augment the serialized representation with additional information about how to unpickle. The key is to capture the type-specific pickler and unpickler when the fully-concrete type of a `Mapped` instance is known (at the call-site):

```
def doPickle[T](msg: T)
  (implicit pickler: Pickler[T],
            unpickler: Unpickler[T]): Array[Byte] = ???
```

Essentially, this means when `doPickle` is called with a concrete type `T`, say:[14]

```
doPickle[Mapped[List[Int], List[String]]](mapped)
```

not only a type-specific implicit pickler (a type class instance) is looked up, but also a type-specific implicit unpickler. These implicit values are carried along in every function signature of the framework. In other words, implicit values have to propagate from end-user code to the network API, responsible

[14]Note that the type arguments are inferred by the Scala compiler; they are only shown for clarity.

of the communication with other nodes.[15] The `doPickle` method could then build a self-describing pickle as follows. First, the actual message is pickled using the pickler, yielding a byte array. Then, an instance of the following simple record-like class is created:

```
case class SelfDescribing(
  blob: Array[Byte],
  unpicklerClassName: String
)
```

Besides the just produced byte array, it contains the class name of the type-specific unpickler.[16] This enables, using this fully type-specific unpickler, even when the message type to be unpickled is only partially known. All that is required is an unpickler for type `SelfDescribing`. First, it reads the byte array and class name from the pickle. Second, it instantiates the type-specific unpickler reflectively using the class name (Note that this is possible on both the JVM as well as on JavaScript runtimes using Scala's current JavaScript backend). Finally, the unpickler is used to unpickle the byte array.

However, this sounds too good to be true. The reality is that the instantiation of the unpickler via reflection is possible in theory, but unfeasible in practice, due to the way implicit resolution works in Scala. Let's explain this in detail with an example. The class `Mapped` is defined as follows:

```
final case class Mapped[T, S](
  target: Node,     // points to the parent node
  sp: Spore[T, S], // spore that we're sending over
  nodeId: NodeId   // identify the node
)(implicit p: Pickler[Spore[T, S]],
           u: Unpickler[Spore[T, S]]) extends Node
```

Following the previous example, we want to send a node of the lineage representing a `map` transformation from `List[String]` to `List[Int]`. To serialize (and deserialize) this `Mapped` instance, we ask for the picklers and unpicklers of such concrete type:

```
val p = implicitly[Pickler[Mapped[List[String], List[Int]]]]
val u = implicitly[Unpickler[Mapped[List[String], List[Int]]]]
```

By definition, the serialization code of `Mapped` depends on the pickler of type `Spore[T, S]`, which in turn depends on the picklers of `T` and `S`.[17] Similarly, when Scala Pickling generates picklers for `Mapped`, the resulting code delegates to the picklers of the type parameters of the constructor, if any. When the implicit resolution takes places, it reuses picklers in the scope and binds to them (Odersky, 2014).

This natural dependency among picklers, that the compiler resolves successfully, also manifests itself in the bytecode-level. Implicit parameters in class

[15]Note that `doPickle` is only an example and doesn't represent the interface that would be invoked by the end-user

[16]Note that this class name is the fully qualified name of a `JVM Class`, not a type that can be known at compile-time.

[17]The same reasoning applies for the deserialization logic.

definition are turned into parameters of the constructors, meaning that such dependency is explicit for the virtual machine that executes the program. This turns the pickler reflective instantiation impossible: the pickler dependencies are unknown when we unpickle `SelfDescribing` and, thus, we lose the ability to invoke the constructor of a class.[18]

A possible workaround would be to keep track of the pickler dependency tree in the serialized message, and reconstruct it in the receiver node. Yet, such solution assumes that the dependencies would always be of type `Pickler` or `Unpickler`, a claim that relies on how the Scala bytecode generation works in theory. In fact, the Scala bytecode generator does indeed break this assumption; instead, when a pickler definition references to more than one `Pickler` and `Unpickler` defined in the outer scope, it uses a reference to the enclosing scope (be it an `object`[19] or a `class`). To the best of our knowledge, there's not a solution to this problem or, at least, a simple way to address it that doesn't depend on the internals of the compiler, prone to change over time. Due to the lack of a generalized solution, we have decided not to pursue this path any further. Below, we present a new solution to this problem.

4.3.8 Caching Picklers at Runtime

So far, we have shown that full static serialization and deserialization is not possible for generic message types on the receiver's side. Conversely, the urge for efficiency prevents us from using runtime-based picklers. In this section, we present a general solution that reconciles both approaches in an efficient simple technique.

The key idea is to provide pickler and unpickler templates for every exchanged message of the protocol. Templates work for all the Scala types, being able to handle from simple types to higher-kind types and type constructors. They are independent and don't delegate their logic to other picklers, stripping out the dependency tree from the executed code. Instead, they rely on the runtime type information[20] (also known as `tags`) extracted from the serialized messages.[21]

The runtime general form of a type consists in replacing all type parameters by `Any` (*e.g.* `List[Int]` becomes the more universal `List[Any]`). This transformation is key—it enables the lookups of pickler and unpickler in a cache that is populated lazily at boot-time. Unlike previously, the cache contains templates and it is checked before falling back to pickler runtime generation.

The proposed solution is to cache default pickler and unpickler templates for every possible used type. Essentially, we're replacing the way we resolve the dependencies of the picklers. Instead of hardcoding these dependencies in the

[18] Note that the use of `sun.misc.Unsafe.instance.allocateInstance` does not solve the problem. Picklers on which the code delegates will be `null`.

[19] By objects we mean Scala `object` definitions, not instances of classes.

[20] Scala Pickling allows to elide types in the serialized objects when the receiver knows what it's expecting to unpickle. This feature must not be enabled for the success of the solution.

[21] `FastTypeTags` are an important component in scala pickling that represent either concrete types at compile-time or general types at runtime.

generated code, we reconstruct them from the augmented type information in the serialized message and lookups in a cache. For the previous example, this involves providing picklers for `Spore[List[Int], List[String]]`, `List[Int]`, `List[String]`, `String` and `Int`.

In conclusion, this approach ensures (a) that a type that is pickleable using a type-specific pickler is guaranteed to be unpickleable by the receiver, and (b) that the efficiency of pickling and unpickling in both sides is roughly the same, with a small penalty for looking up the cached picklers.

4.4 Extensions to Spores

The programming model makes extensive use of spores, closure-like objects with explicit, typed environments. While (Miller, Haller, and Odersky, 2014) has reported an empirical evaluation of spores, the presented programming model and implementation turned out to be an extensive validation of spores in the context of distributed programming. In addition, the implementation required a thorough refinement of the way spores are pickled.

Compile-time metaprogramming in any programming language is hard, since its fragile interaction with the compiler makes it difficult to achieve full correctness. The implementation of spores left some room for improvement. Some of the issues we found were minor (type inference problems, correctness when dealing with path-dependent types) and, as such, are not discussed in this paper. However, we discuss in detail the most relevant improvement in the spores project.

4.4.1 Missing support for Function0

Problem `Function0` is a trait in the Scala standard library. Any parameterless function extends it.[22] Spores provide implicit conversions from functions to spores, that essentially extend function objects with spore objects. As the use of functions is prominent in Scala, one would expect direct conversion to a spore for any kind of function. As the following example shows, this is not the case for functions with no parameters.

```
// Works! There's an implicit conversion!
val f1 = (name: String) => println(s"Hello, name")
val sp1: Spore[String, String] = f1

// Doesn't work, no spore0 class and implicit conversion!
val f0 = () => println(s"Hello, Joe")
val sp0: ??? = f0
```

Solution The solution to the issue was rather straightforward, performed in two steps. First, we added an analogous spore class to `Function0`, called `NullarySpore`. Second, we provided default implicit conversions between the two. And finally we exposed pickler and unpickler instances for it.

[22]Note that Scala functions are represented as objects.

4.4.2 Structural Changes in Spores

Previously, at the time of the spore class generation, spores not capturing anything from the environment extended `Spore[T, S]` for some concrete types `T` and `S`.

At the type level, it's necessary to distinguish between a spore that captures variables from the environment and one that doesn't. The latter type of spore is more specific and, therefore, it's represented as a subclass of `Spore`.

```
trait SporeWithEnv[-T, +R] extends Spore[T, R] { ... }
```

It is conceivable, though, that developers want to have more control when choosing their spore types, since the super spore type is not explicit enough.[23] A compelling situation, for instance, is that of a library designer that wants their users not to capture anything from the environment. For those cases, she would use `Spore[T,S] {type Captured = Nothing}` instead of the old one.

4.5 Tying Things Together: Pickling Spores

Pickling and unpickling spores is a topic that has been presented in previous research. The Spores paper (Miller, Haller, and Odersky, 2014) introduces an extensible type-based mechanism to define custom properties via implicits. The capability to serialize and deserialize can be expressed as one of these properties, as we have already shown in previous sections. Whilst this thesis explains specific-type picklers and unpicklers, it doesn't look into neither runtime pickling strategies nor templates. Once the programming model implementation settled on using pickler and unpickler templates, such support was essential. As a result, we incorporated them into the default existing implementation.

4.5.1 Representing Spores that Capture Variables

The natural distinction between different kinds of types also changes the pickling and unpickling logic—unlike `Spore`, `SporeWithEnv` needs to serialize and deserialize its environment (the captured variables).

The previous implementation roughly provided this feature. However, most concrete spore types failed to pick the correct `Pickler` and `Unpickler`. Suppose the following situation:

```
// Define a spore with environment
val captured = 1
val sp: Spore[Int, String] = spore {
  val c: Int = captured
  (i: Int) =>
    (i + c).toString
}
```

[23]Note that, from an object-oriented perspective, super types also represent concrete subclasses.

```
// Pickle a spore
val p = sp.pickle
```

The right-hand side of the spore definition assigns a spore of type `SporeWithEnv [Int, String]` to a variable `sp`. However, `sp` has the type `Spore[Int, String]`. As the implicit search only takes into account the type signature that triggers it, it picks the pickler for the super class when, intuitively, should pick the pickler for the concrete class `SporeWithEnv`. The fact that implicits do not fully support type inheritance is not a limitation, but the way it's supposed to work.

The aforementioned case occurs frequently when designing interfaces, as in the case of the function-passing implementation. For instance, the `map` method doesn't know the kind of spore that will receive as an argument. Thereby, it picks the most general one, `Spore`.

The proposed solution adds a new level of indirection and has been reworked from scratch. Special `Picklers` and `Unpicklers` are provided for the general spore type. Nonetheless, those picklers as an intermediate dispatcher that checks whether the spore to be pickled is indeed an instance of `SporeWithEnv`. In that case, a tailor-made pickler and unpickler for that instance is used. Otherwise, conventional pickling and unpickling takes place. The tests and experimentation confirm the success of such improvement to pickle and unpickle spores, without additional corner cases.

4.5.2 Pickling and Unpickling Spores

The function-passing model implementation assumes shared compilation, meaning that all the different programs interacting via the programming model need to be compiled in the same jar. Although this could be regarded as a limitation, it enables pickling and unpickling spores, a fundamental feature of the implementation. Below, we explain the relation between these two seemingly unconnected topics.

The spore macro phase works as follows: whenever it sees a `spore`, the macro is expanded and creates a new spore class. That class extends the inferred spore type and contains the captured variables of the environment. The identifier of the brand new class is unique; it is created through with `freshName`, a method exposed in the *Context* that takes care of avoiding name collision.[24]

As a result, pickling and unpickling spores is straightforward: send the unique spore class name and instantiate it on the receiver side through basic reflection.[26] In the case of spores with environments, the captured variables

[24]Name collision is a typical problem of metaprogramming; macros could define terms with names that are used in the transformed program, resulting into naming issues at runtime. To avoid them, the internal implementation increments a counter and merges its value with the current FQN (fully qualified name)[25] every time a new fresh name is created and adds it to the end of the current fully qualified name at the place of the macro expansion.

[26]By basic reflection, we mean usual optimized methods of the reflection API like `Class.forName`, that it happens to work in both Scala.js and the JVM. For that, it's required to have access to the concrete class name.

are also transmitted and set in the instantiated spore.

Why, then, is shared compilation required? Let's suppose for a moment that client and server code are compiled separately. The client will successfully send any message to the server. Those messages contain, among other things, spores representing the function to be evaluated in the server. When the server receives one of these spores, it will try to deserialize the spore and fail. The reason of this failure is that the spore unpickler will get the spore class name, and that class name does not exist for the server *JVM*; unfortunately, it only exists for the client. The only valid approach to this issue is to bundle up any part of the system altogether and share that jar in order to run it.

Yet, users concerned and constrained by shared compilation are not left at their fate. Possible alternatives to work around this problem include, but are not restricted to, several strategies that leverage dynamic class loading. These techniques have been disregarded because of security and performance reasons. On the one hand, potential attackers would be able to execute malicious code by carefully creating and loading classes of the expected type. On the other hand, the use of a secure network protocol and the cost in message size and class loading [27] would be too much of an overhead for these systems, that usually run in private inaccesible clusters.

However, when it comes to pickle and unpickle templates, there's a problem. On the receiver's side, the `FastTypeTag` that contains the necessary type information for looking up and generating picklers/unpicklers is created from the runtime class name (the concrete type is unknown and, therefore, `AnyPicklerUnpickler` is used). Thereby, looking up a cached pickler does not work: the pickler/unpickler template is registered for the type `Spore[Any, Any]` and the actual type is the unique compile-time class name. This problem requires a two-fold solution:

- Include spore type information in the unique class name
- Modify `FastTypeTag` to specially handle class names that are recognized to be from a spore.

When not pickling and unpickling raw spores, but classes whose class arguments are bound to the general type `Spore[Any, Any]`, this problem disappears.

Realizing these changes in both frameworks has finally enabled a flawless mechanism that allows proper serialization and deserialization of spores even if the received message types are unknown.

Dealing with all the spore types

Spore types are complex and sophisticated. Class inheritance is one of the main responsibles, although type members play an even bigger part in it. Formally, spore types are either *parameterized types* (and more concretely,

[27]The JVM internally optimizes basic kinds of reflection and class loading is considered a hot path when the server mode is enabled. Still, some other JVMs do not necessarily behave in the same way and its support could change in future releases. In addition, class loading is not for free and could significantly affect performance for a large amount of received messages.

function types) or *compound types* (Odersky, 2014). On the one hand, parameterized types are type constructors that take other types as parameters and yield a concrete type. Conversely, compount types contain declaration and type definitions and they are referred informally as structural types. At the same time, yype declarations and definitions are also known as [type refinements].

```
// Examples of some spore types
Spore[Int, String]
Spore[Int, String] {type Captured = Nothing}
Spore[Int, String] {type Captured = ((String, String), Double)}
Spore[Int, String] {type Excluded = Nothing}
Spore[Int, String] {type Excluded = ActorRef}
Spore[Int, String] {type Captured = Int; type Excluded = String}
SporeWithEnv[Int, String]
SporeWithEnv[Int, String] {type Captured = Nothing}
SporeWithEnv[Int, String] {type Captured = ((String, String), Double)}
SporeWithEnv[Int, String] {type Excluded = Nothing}
SporeWithEnv[Int, String] {type Excluded = ActorRef}
SporeWithEnv[Int, String] {type Captured = Int; type Excluded = String}
```

As you see, the complexity doesn't come only from the type parameters of a function type but type members. This explosion in different combinations of valid types is remarkable; not only all of them need to be picklable and unpicklable, but also keep the same semantics in this process, respecting the natural object inheritance.

Fortunately, the `Excluded` type member can be safely ignored when generating pickling code—its main goal is to make the compiler fail under certain circumstances. Therefore, only the `Captured` type is necessary. This situation, this time considerably more manageable, has still required a challenging implementation.

4.5.3 Shortcomings in the Pickling Design

Previously, we discussed how the server is able to pickle and unpickle generic messages. Our solution depended on pickler and unpickler templates, that are self-registered in a dictionary at run time. These templates are `object`s, the equivalent of a singleton class.

Singleton classes have special initialization procedures. Other programming languages like Java make use of the *synchronized* keyword to ensure that such classes are thread-safe. Java software developers are required to use the same "recipe" every time they need singleton classes. On the contrary, Scala provides this feature by default, allowing the user to define any singleton class as an *object* instead of a *class*. Thereby, Scala deals with the thread-safe and correct implementation of the object on behalf of the user. Nonetheless, this implementation imposes some conditions, that may be regarded as beneficial or perjudicial depending on the use case. These conditions are introduced because of performance reasons. The most relevant one is the lazy initialization of objects, meaning that their constructor won't be executed until the program explicitly invokes it.

This is, in the context of Scala Pickling, a drawback. Even though we define picklers and unpicklers via macros, these are not initialized by default. As a workaround, Scala Pickling explicitly registers templates which are defined in the library, such as those for primitive types and Java/Scala collections. But, in the case of custom user-defined classes, the templates won't be registered in the server—the server never gets to initialize the picklers and unpicklers because it doesn't use them—, thus its serialization and deserialization would turn impossible without reflection.

As a consequence, here it comes the only manual task that users of the function-passing model need to do: every time they want to pickle and unpickle their own classes, they need to take care of registering them in the server. In order to do so, they are required to either use the internal API of Scala Pickling, or invoke methods of the templates. Such disadvantage is not, however, a unique limitation of Scala Pickling but it is also present other popular serialization libraries like Protocol Buffers (Google, 2015). We believe it to be an intrinsic problem of static-based serialization, unlike reflection-based serialization libraries which are somewhat more general but far more inefficient.

4.6 Fault Tolerance

Whereas fault tolerance is simplified by design, real-world issues like power or network failures are not automatically solved by the proposed model. Their consequences are severe, ranging from data loss to unavailability of systems. It's hard to deal with these problems, though; frameworks like *Spark* and *Akka* devote considerable large pieces of code to minimize them.

The function-passing implementation copes with failure in a similar way as mainstream frameworks. The implementation is heavily inspired by the fault-tolerant best practices of Akka (Roestenburg, Bakker, and Williams, 2012). This section discusses some of these approaches for the function-passing model.

Network Failures

Failures in network communication happen when sending commands to remote machines or when receiving their confirmation (or *ACK* messages). Spurious errors may affect both directions and are unpredictable.

In order to solve this problem, systems have delivery guarantees. The defacto strategy is exactly-once delivery (Huang and Garcia-Molina, 2001), meaning that a message is only processed once but may have been sent or received as many times as necessary. A diagram of its behaviour is provided in Figure 4.3.

In short, nodes send commands among them. In case a command is lost, the client waits for confirmation during a concrete period of time (customizable by end-users). After that, the command is sent again until an acknowledgement is received.

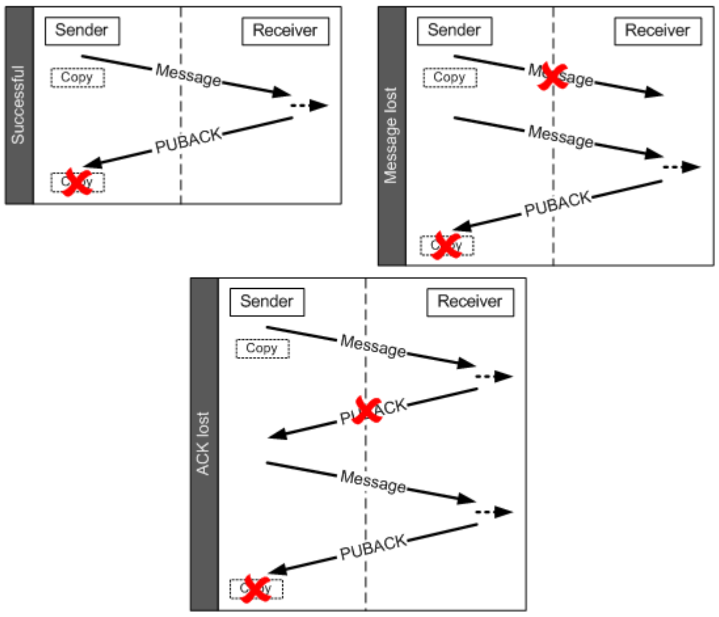


FIGURE 4.3: Diagram of a exactly-once delivery, from (Moyer, 2012).

Nevertheless, is this a valid approach? What if a temporary network latency problem happens and the server eventually receives two messages, the first being the duplicate of the second one? It is clear that one cannot ignore this possibility and leave it to its fate.

To solve it, we introduce the concept of expected id. The expected id is a counter that is incremented once a message is received. Duplicated messages with the same id will be rejected because their id is lower than the expected one. For the previous example, once the first message has been processed, the counter is incremented and, thus, different to the duplicated message's id. Further, messages with higher ids will be stored in memory and its processing is delayed. This technique is, at the best knowledge of the writer, known as the *ACK-Retry* protocol and keeps FIFO semantics (Fraser, 1982), which are particularly relevant because, then, the original order of the client code will be executed sequentially.

4.6.1 Power Failures

Power failures are the most severe kind of failures in a distributed system. They get rid of the machine state and make difficult its later inclusion into the system. The absence of state unables failed nodes to keep up with their peers and continue previous communications.

Persistence combats power failures—nodes store any message upon reception. Messages represent the state of the system and allow to replay the history of previous transformations over data. When crashed nodes boot, they check if there's any previous state. Whether it's available in a distributed or local storage, they use it to remember their latest state and be up for receiving more communications from other nodes. This conforms to the idea of *single source of truth*. It is a common practice for structuring information models that has been successfully used in real-world distributed systems (Hammant, 2012; Typesafe, 2014), as well as in distributed log frameworks (Kreps, 2013).

4.6.2 Dealing with Unexpected Errors

There exists a subset of failures that, yet very very unlikely to happen, can affect the correctness of the system. Such failure may appear depending upon transient bugs or the platform on which the system is running. Whereas there could be ignored, the function-passing model proposes a way to address them.

It is well-regarded to implement extra high-level reliability layers atop of distributed systems (Saltzer, Reed, and Clark, 1984), even though low-level network protocols implement their own. The rationale behind this reasoning is that when application make an extended use of the IO layers, they cannot blindly rely on them, expecting them to work correctly, because one day they won't. While it's unlikely that low-level failures occur, they are not negligible and account for a significant amount of real-world errors, *e.g.* buffer-copy errors that may switch over bytes of messages. Consequently, the function-passing implementation adds a checksum to every exchanged message, and checks upon reception that they match with the checksum of the message contents.

The handling of power failures 4.6.1 and unexpected errors 4.6.2 has been implemented out of the project schedule because of timing issues, while working around network failures 4.6 has been successfully implemented on time.

4.7 Memory reclamation

The *Java Virtual Machine* is known for encouraging software developers to avoid memory management by providing a default garbage collector—developers just write their programs without directly dealing with heap allocations and deallocations.[28]. The success of this approach has revolutionized the industry. Garbage collectors are a must in the toolbox of any modern programming language that wants to succeed. They make easier software development at a small cost in performance, and developers are willing to pay this price.

The function-passing model stands under the same motto. In a similar way, it wants to provide safe guarantees, performance and ease of use for data analysts and developers, even though sometimes there's a price. Immutable `Silos` pose a problem for memory management. As data transformations are persisted in a dictionary and indexed by their silo identifiers, their references are never nulled out.[29] Thus, the JVM Garbage Collector does not

[28]In fact, only a few limited APIs allow direct access to low-level memory operations, and some rumours confirm their removal in future releases of the *Oracle JVM* (Beckwith, 2016)

[29]There always exist a reference to these objects in the running programs.

recognize unused silos as garbage and they may pile up in memory until an `OutOfMemoryException` is thrown.

As a consequence, one has to explicitly deal with all these intermediate silos that float around memory. While this is still a field of active research at the time of this writing, we have found out two sensible user-friendly approaches to this problem:

- The first approach uses Java's *WeakReference*s (SE, 2012b) and *Soft-Reference*s (SE, 2012a) to detect when a SiloRef is no longer reachable from local GC roots. Upon detection the host of the corresponding silo is notified to decrease the silo's reference count; the host's reference(s) to the silo are nulled out when the reference count reaches zero. It is important to note that this strategy requires notifying a silo's host whenever a `SiloRef` to the silo reaches a new machine, to increase the silo's reference count. This notification could be automated via macros; since the judgment of when a silo is useless can be inferred from the client's code at compile-time, macros could extract and process this information, removing the burden of providing explicit silo information to hosts. The proposed approach enables the JVM to automatically collect objects pointed by *WeakReference*s and *SoftReference*s when memory resources are scarce, even if they are still "used"/referenced in the program. Note that such solution doesn't pose a problem to end-users—silos could be recomputed from their lineage.
- The second approach leverages uniqueness types in Scala (Haller and Odersky, 2010). Here, SiloRefs are locally unique, and the programmer can explicitly declare a SiloRef as unused; the type system ensures that such an "unused" SiloRef is not used again subsequently. As in the first approach, upon marking a SiloRef as unused, the corresponding silo's host is notified to decrease the silo's reference count.

As a future extension, the function-passing model will also be able to behave as *Spark* does: dumping and loading data from disk when there's not enough free memory available.

4.8 Extensibility of the function-passing model

The described implementation may sound limited for some use-case scenarios. The reality is that the proposed programming model does not aim at covering all of them. However, it does provide a minimum set of primitives that allow further extensibility of the model.

The Scala programming language is known for providing primitives for asynchronous operations by default in the standard library, whose main elements are `Futures` and `Promises` (Haller et al., 2012). Thus, the function-passing model, which uses them for any operation througout the framework, is by definition extensible; potential users may build atop of it and benefit from its seamless integration with other future-based libraries, because the key idea behind `Futures` is composability.

As a conclusion, there is no natural limitation in the function-passing model. For instance, advanced users would be able to share `SiloRefs` among different nodes just by storing them to distributed databases or message brokers like Redis, ZeroMQ, or RabbitMQ.

Chapter 5

Conclusion

We have presented the function passing model, a new programming model and new substrate or middleware upon which to build data-centric distributed systems. This enables two important benefits for distributed system builders; since (a) all computations are functional transformations on immutable data, fault-tolerance is made simple by design, and (b) communication is made well-typed by design, the function passing model attempts to more naturally model the paradigm of data-centric programming by extending monadic programming to the network.

In addition, we have gone in detail through the two previous veins of work that have enabled the presented programming model: Scala Pickling and Spores, and explained how they interact with each other. We then discussed fundamental issues of these projects and our proposed solutions, whose result has made the cross-platform implementation of the function-passing model possible. Therefore, uniting functional programming and distributed computing in a pure setting. The novelty of this implementation is its interoperability with both JVM-based programming languages and JavaScript.

Finally, we have shown our approach to efficiently pickle and unpickle functions in Scala, even if the function type is only known at runtime. It combines compile-time and runtime generated picklers, along with runtime pickler and unpickler caches. The success in this task has allowed us to implement different popular patterns of distributed processing atop our programming model, such as batch processing with Spark's RDDs and MBrace's cloud-based asynchronous tasks.

Appendix A

Implicits in Scala

Implicits are a powerful mechanism in Scala for scraping boilerplate. The general idea is to allow the compiler to help software developers write less code. The compiler allows them to define "special" values that will be placed in certain places of the program on behalf of the user. The compiler is guaranteed to choose the most specific type for that value based on the guarantees of the place that requests it. In order to understand how they work, let's start by the `implicitly` method.

The `implicitly` method, part of Scala's standard library, is defined as follows:

```
def implicitly[T](implicit e: T) = e
```

That is, request an implicit value of type `T` as a parameter and return it.

Annotating the parameter (actually, the parameter list) using the `implicit` keyword means that in an invocation of `implicitly`, the implicit argument list may be omitted if, for each parameter of that list, there is exactly one value of the right type in the *implicit scope*. The implicit scope is an adaptation of the regular variable scope; imported implicits, or implicits declared in an enclosing scope are contained in the implicit scope of a method invocation.

As a result, `implicitly[T]` returns the uniquely-defined implicit value of type `T` which is in scope at the invocation site. In the context of picklers, there might not be an implicit value of a certain type in scope. In that case, a suitable pickler instance is generated using a macro def.

Implicit conversions. Implicit conversions can be thought of as methods which can be implicitly invoked based upon their type, and whether or not they are present in implicit scope. Implicit conversions carry the `implicit` keyword before their declaration. The `greeting` method is provided using the following implicit conversion:

```
implicit def normalToRichConversion(i: Int): RichInt =
  new RichInt(1)

class RichInt(i: Int) {
  override def greeting(): String =
    "Hello, I am a rich Int"
}
```

```
// What developers usually do
println(RichInt(1).greeting)
```

In a nutshell, the above implicit conversion is meant to implicitly invoke `RichInt`, wrapping any `Int` whenever the `greeting` method is invoked. The above example shows a synthesized and less verbose code that simulates the previous behaviour:

```
// The compiler wraps 1 with `new RichInt()`
println(1.greeting)
```

Implicit classes. The success of implicit conversions and its widespread adoption in the Scala projects inspired new features that aimed at simplifying the use of implicits. Among them, implicit classes marry the worlds of classes and implicits once and for all.

Instead of separate implicit conversions and class definitions, one could express the same example as:

```
implicit class RichInt(i: Int) {
  override def greeting(): String =
    "Hello, I am a rich Int"
}
```

where the implicit keyword has been moved to the beginning of the class definition. Due to its usefulness, it's become a popular Scala idiom that reduces boilerplate.

Appendix B

Macros: Compile-Time Metaprogramming in Scala

Compile-time metaprogramming is a technique that allows algorithmic construction of programs at compile-time. It's often used with the intent of allowing programmers to generate parts of their programs rather than having to write these program portions themselves. Thus, metaprograms are programs who have a knowledge of other programs, and which can manipulate them (Burmako, 2013b).

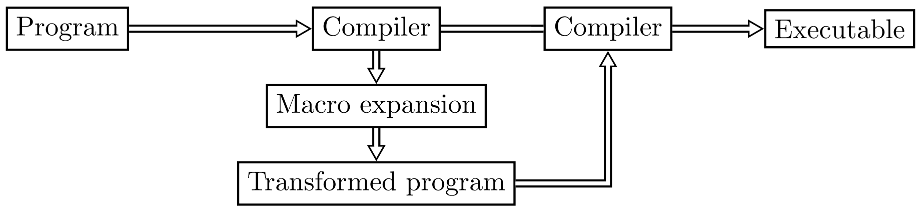


FIGURE B.1: Describes the usual compiler pipeline when macros are enabled.

When a macro is invoked, it exposes internals of the Scala compiler API, thereby allowing us to manipulate and transform expression trees (ASTs). An illustration of the usual compiler pipeline is described in Figure B.1. There are many different flavors of Scala macros, which enable manipulation of types as well as terms. In this appendix, only the most basic form of macros are explained, *def macros*.

Def macros are methods whose calls are expanded at compile-time. An expansion is a transformation into a code snippet (usually in the form of a method) and its arguments. When macros are expanded, they operate with a *Context* that exposes the code to be expanded and an interface to interoperate with *Scala ASTs*. Imagine that one want to create a smart `assert` function that is able to fail compilation when some objects are not of the same type.[1] This is a traditional problem that can be solved by def macros:

```
def assert(cond: Boolean, msg: String) = macro assertImpl

def assertImpl(c: Context)(cond: c.Tree, msg: c.Tree) = {
```

[1]Note that `equals` is extended by the class `Object` which, in turn, is extended by any *Java* and *Scala* class.

```
  // Import the compiler universe
  import c.universe._
  // Continue the implementation
  ???
}
```

In the above example, there's a macro definition and macro implementation. The `assert` method expands the macro, that executes the body of `assertImpl` at compile-time, manipulates its arguments and produces a tree. Such tree will replace the initial invokation of `assert` and will be executed at runtime.

Macros have enabled countless opportunities of automating tasks, removing boilerplate and creating *DSL*s. Without them, projects like *Scala Pickling* and *Spores* would have been unconceivable. In order to illustrate how Scala Pickling makes use of macros, let's have a look at the following example:

```
/* Framework code */

// Define a simple pickler interface
trait Pickler[T] { def pickle(x: T): String }
def pickle[T](x: T)(implicit ev: Pickler[T]) = ev.pickle(x)

// Declare a macro that will generate the pickling code for a type T
implicit def generatePickler[T]: Pickler[T] = macro ...

/* User code */

// Define a person and pickle an instance
case class Person(name: String)
pickle(Person("Jorge"))
```

First, we generate a `Pickler` and a `pickle` method that asks for an implicit of `Pickler[T]`. Then, we define an implicit conversion that provides any pickler for any given type `T`.[2] As there is an implicit conversion in scope that provides such type, the Scala compiler selects it and transforms the last line into:

```
pickle(Person("Jorge"))(generatePickler[Person])
```

However, `generatePickler` is a macro definition that will be expanded by the compiler. After macro expansion, the resulting code will be similar to:

```
pickle(Person("Jorge"))(new Pickler[Person] {
  def pickle(x: Person): String = ???
})
```

The above explanation describes superficially how the framework actually works. For more information about macros, see (Burmako, 2013b), (Burmako, Shabalin, and Odersky, 2013) and (Burmako, 2013a).

[2]Actually, this is an oversimplification of what *Scala Pickling* does. It's impossible to generate pickling code for any type of object. When Scala Pickling detects this unfeasibility, it will fail with a compiler error at the call-site of the macro expansion.

Bibliography

Apache (2015a). *Avro*. http://avro.apache.org/.
– (2015b). *Hadoop*. http://hadoop.apache.org/.
Beck, Kent et al. (2001). *Manifesto for Agile Software Development*. http://agilemanifesto.org/.
Beckwith, Monica (2016). *Oracle's OpenJDK Cleanup of Unsafe Implementation*. https://en.wikipedia.org/wiki/Functional_programming.
Bloom, James D. (2013). *JVM Internals*. http://blog.jamesdbloom.com/JVMInternals.html.
Burmako, Eugene (2013a). *Macro Paradise*. http://docs.scala-lang.org/overviews/macros/paradise.html.
– (2013b). "Scala Macros: Let our powers combine!" In: *Montepellier Scala Symposium*.
Burmako, Eugene, Denys Shabalin, and Martin Odersky (2013). *Scala Macros Poster*. http://scalamacros.org/paperstalks/2013-11-25-ScalaMacrosPoster.pdf.
Cantero, Jorge Vicente, Heather Miller, and Philipp Haller (2016). "Scala Pickling and Spores: Conquering the Awkward Squad in the Function-Passing Model". In:
Carpenter, Bryan et al. (1999). "Object Serialization for Marshalling Data in a Java Interface to MPI". In: *Java Grande*, pp. 66–71.
Chambers, Craig et al. (2010). "FlumeJava: easy, efficient data-parallel pipelines". In: *PLDI*, pp. 363–375.
Church, Alonzo (1941). "The calculi of lambda conversion". In: Princeton University Press.
Dean, Jeffrey and Sanjay Ghemawat (2008). "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1, pp. 107–113.
Doeraene, Sébastien (2015). *Scala.js*. https://www.scala-js.org/.
Dragos, Iulian (2010). "Compiling Scala for Performance". PhD thesis. École Polytechnique Federal de Lausanne.
Dzik, Jan et al. (2013). "MBrace: cloud computing with monads". In: *PLOS 2013, Farmington, Pennsylvania, USA, November 3-6, 2013*, 7:1–7:6.
Fraser, Alexander G. (1982). "First-in, first-out (FIFO) memory configuration for queue storage". In: US Patents.
Google (2015). *Protocol Buffers*. https://github.com/google/protobuf/.
Haller, Philipp and Martin Odersky (2010). "Capabilities for Uniqueness and Borrowing". In: *ECOOP 2010, Maribor, Slovenia, June 21-25, 2010.* Pp. 354–378.
Haller, Philipp et al. (2012). "Futures and promises". In: http://docs.scala-lang.org/overviews/core/futures.html.
Hammant, Paul (2012). *The Document is the Single Source of Truth*. https://dzone.com/articles/my-opinion-document-single.

Huang, Yongqiang and H. Garcia-Molina (2001). “Exactly-once semantics in a replicated messaging system”. In: Data Engineering, 2001. Proceedings. 17th International Conference.

Kreps, Jay (2013). *The Log: What every software engineer should know about real-time data’s unifying abstraction.* `https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying`.

Lindholm, Tim et al. (2013). *The Java Virtual Machine Specification.* `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html`.

Microsystems, Sun (2006). *Memory Management in the Java HotSpot™ Virtual Machine.* `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`.

Miller, Heather, Philipp Haller, and Martin Odersky (2014). “Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution”. In: *ECOOP*, pp. 308–333.

Miller, Heather et al. (2013). “Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization”. In: *OOPSLA*, pp. 183–202.

Miller, Heather et al. (2016). “Capabilities for Uniqueness and Borrowing”. In: *Onward! 2016, Amsterdam, Netherlands, November 2-4, 2016.*

Monden, Yasushiro (1983). *Toyota production system: practical approach to production management.* `https://books.google.ch/books/about/Toyota_production_system.html?id=_9RTAAAAMAAJ&redir_esc=y`.

Moyer, Bryon (2012). *Is exactly once-delivery possible?* `http://www.eejournal.com/blog/is-exactly-once-delivery-possible-with-mqtt/`.

Netty (2011). *Netty: asynchronous, event-driven network application framework.* `http://netty.io/index.html`.

NICTA (2015). *Scoobi.* `https://github.com/nicta/scoobi`.

Odersky, Martin (2014). *The Scala Language Specification.* `http://www.scala-lang.org/files/archive/spec/2.11/`.

Pressman, Roger S and Bruce R. Maxim (2015). *Software Engineering: A Practitioner’s Approach.* McGraw Hill. ISBN: 0078022126.

Roestenburg, Raymond, Rob Bakker, and Rob Williams (2012). *Akka in Action.* Manning Publications. ISBN: 9781617291012.

S. Oliveira, Bruno C. d., Adriaan Moors, and Martin Odersky (2010). “Type classes as objects and implicits”. In: *OOPSLA*, pp. 341–360.

Saltzer, J.H., D.P. Reed, and D.D. Clark (1984). *End-to-end arguments in System Design.* `http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf`.

Schwaber, Ken (2002). *Development With Scrum.* `https://www.amazon.com/Agile-Software-Development-Scrum/dp/0130676349`.

SE, Java Platform (2012a). *SoftReference Official Documentation.* `https://docs.oracle.com/javase/6/docs/api/java/lang/ref/SoftReference.html`.

– (2012b). *WeakReference Official Documentation.* `https://docs.oracle.com/javase/6/docs/api/java/lang/ref/WeakReference.html`.

Sutherland, Jeff and Ken Schwaber (1995). *SCRUM Development Process.* `http://www.jeffsutherland.org/oopsla/schwapub.pdf`.

– (2013). *The Scrum Guide.* `http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-us.pdf`.

Systems, Azul (2012). *Understanding Java Garbage Collection*. http://www.azulsystems.com/sites/default/files/images/Understanding_Java_Garbage_Collection_v3.pdf.

Twitter (2015). *Scalding*. https://github.com/twitter/scalding.

Typesafe (2014). *Akka Persistence*. http://doc.akka.io/docs/akka/snapshot/scala/persistence.html.

– (2015). *Akka*. http://akka.io/.

Vashishtha, Shrikant (2012). *Agile Scrum Team*. http://www.agilebuddha.com/agile/agile-offshore-business-analyst-complementing-the-role-of-product-owner/.

Wadler, Philip (2010). "The essence of functional programming". In:

Waldo, Jim et al. (1996). "A Note on Distributed Computing". In: *MOS*, pp. 49–64.

Wikipedia (2016a). *Closures*. https://en.wikipedia.org/wiki/Closure_(computer_programming).

– (2016b). *Functional Programming*. https://en.wikipedia.org/wiki/Functional_programming.

Wikipedia, the free encyclopedia. *Kanban*. https://en.wikipedia.org/wiki/Kanban.

Yodiz (2012). *User stories*. http://www.yodiz.com/help/agile-user-stories-and-groomed-product-backlog/.

Zaharia, Matei et al. (2012). "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *NSDI*, pp. 15–28.