ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Design and Implementation of a Portable Framework for Application Decomposition and Deployment in Edge-Cloud Systems

Tesi di laurea in
Pervasive Computing

Relatore
Prof. Mirko Viroli

Correlatore
Prof. Danilo Pianini

Candidato
Nicolas Farabegoli

Anno Accademico 2021-2022

Abstract

The emergence of cyber-physical systems has brought about a significant increase in complexity and heterogeneity in the infrastructure on which these systems are deployed. One particular example of this complexity is the interplay between cloud, fog, and edge computing. However, the complexity of these systems can pose challenges when it comes to implementing self-organizing mechanisms, which are often designed to work on flat networks. Therefore, it is essential to separate the application logic from the specific deployment aspects to promote reusability and flexibility in infrastructure exploitation.

Starting from the existing "pulverization" approach, which involves breaking down a system into smaller computational units that can be deployed on the available infrastructure, this thesis presents the design and implementation of a portable framework that enables the "pulverization" of cyber-physical systems. The main objective of the framework is to pave the way for the deployment of cyber-physical systems in the edge-cloud continuum by reducing the complexity of the infrastructure and exploit opportunistically the heterogeneous resources available on it. Different scenarios are presented to highlight the effectiveness of the framework in different heterogeneous infrastructures and devices. This thesis work makes a crucial step toward dynamic deployment in cyber-physical systems exploiting the edge-cloud continuum.

Alla mia famiglia.

Acknowledgements

First of all, I would like to express my gratitude towards Professors Viroli and Pianini for their invaluable assistance and insightful guidance, which was essential for the realization of this thesis work. I thank my family infinitely for their continuous support and motivation during these years of study and for never letting me lack anything. A special thanks go to Giulia who has always been there in these two years supporting and putting up with me in both the most difficult and the most beautiful moments of this path. I would also like to extend my thanks to the members of the Atedeg group with whom I have shared countless unforgettable moments and experiences. Additionally, I am grateful to all my university colleagues for the enjoyable times we have spent together. Lastly, but certainly not least, I am grateful to all my friends who have made these two important years so joyful and carefree. To everyone who believed in me and supported me during this time, I express my heartfelt thanks.

Contents

List of Figures

Listings

Chapter 1

Introduction

With the Internet of Things (IoT), more and more devices are connected to the network producing a very large amount of data. Cloud computing has been established as a technology for acquiring computational power and storage in support of various applications; however, it is not always suitable for handling all kinds of systems' requirements: latency, security, and privacy are some of the main concerns. This comes true especially with IoT systems since they produce lots of data and in some scenarios, they must respect real-time constraints. For these reasons, fog computing tries to overcome the cloud's limitation by defining a computing model that sits between IoT devices and the cloud. It allows for the collection, aggregation, and processing of data from IoT devices (or more in general edge devices) using a hierarchy of computing power. The combination of fog computing with the cloud can reduce data transfers and communication bottlenecks to the cloud, and can also contribute to reduced latencies since fog computing resources are closer to the edge.

Nevertheless, realizing systems that operate in the edge-cloud continuum is an open challenge [1]: the heterogeneity of the devices combined with the dynamic nature of the requirements that modern systems must have, leveraging the flexibility of the edge-cloud continuum is found to be as strategic as it is complex.

Different approaches have been proposed to address the challenges of realizing systems that well interoperate in heterogeneous infrastructures. Some notable methodologies and frameworks are represented by the osmotic computing paradigm [2], DR-BIP and its extensions [3, 4, 5]. While the first approach is mainly oriented to distributed microservices, the latter is more focused on the orchestration of distributed applications by dynamically adapting the system to the changing requirements basing the systems on the *motif* concept.

In the CPS context, engineering systems featuring distributed intelligence in a *self-organization* fashion is one of the main relevant approaches. In this way, the global behaviour of the system is obtained by the interaction of the individual

components giving robustness to the system. The current trend of large-scale, dynamic and heterogeneous Cyber-Physical Systems requires increasingly complex and diverse infrastructures. Remote clouds offer a seemingly supply of computing, storage, and services on demand, but this comes with the caveat of high costs and potential latency issues, as well as data protection concerns that must align with the specific requirements of each application. Edge computing, on the other hand, brings resources closer to users, resulting in reduced latency and increased reactivity, while simultaneously addressing data dissemination concerns.

As stated before, such infrastructures are not easy to manage and orchestrate, complicating the engineering phase where the logic of the system tends to be coupled with infrastructure aspects. Generally, this prevents the reusing of design elements across different scenarios by exploiting the underlying infrastructure opportunistically.

To tackle this problem in [6] the *pulverization approach* is proposed: this framework brakes the system behaviour into small computational pieces logically linked to sensors and actuators that are continuously executed and scheduled in the available infrastructure. In this way, the system can be seamlessly mapped onto a variety of multi-layered deployment infrastructures. It is based on a flexible logical model which can be decomposed into a set of sub-components with well-defined relationships that can be deployed and wired separately. The pulverization facilitates the deployment independence of a system, namely the ability to run the application with no change on various deployments retaining its original functional semantics. In this way, the application logic will obtain the functional goals independently of the actual deployment since the choice of the deployment strategy is affected typically by non-functional requirements such as latency, security, performance and cost. This approach is formalized to provide an unambiguous specification of what constitutes pulverization by clarifying subtle aspects of the model and state the deployment-independence property rigorously.

1.1 Motivation of the thesis work

At the time of writing, the pulverization approach is tested and validated in a simulated environment. The simulation represents a very important step to figure out the validity of the approach by testing it in a controlled environment. In a large view, the simulation represents a powerful tool to validate the correctness of the systems and to identify the potential problems that may arise in a real deployment. However, closing the gap between a simulated system and its deployment in a real infrastructure is not trivial and it is more challenging to do it seamlessly.

A desired development process might be to port the simulated system to the real one with as few changes as possible. In this sense, it is desired to make use of

tools that can appropriately handle specific aspects of the infrastructure on which the system will be deployed without the user having to worry about them, or even worse having to tie platform-specific aspects in the application logic.

As said before, the pulverization approach has been theorized and verified in simulation and for this reason, it is not corroborated by any tools or frameworks that enable the deployment of systems in a real infrastructure leveraging this approach.

To solve these issues, this thesis aims to pave the way for closing the gap between the simulation of systems and their deployment by leveraging the pulverization approach via a dedicated framework.

The developed framework leverages the pulverization approach to orchestrate distributed applications in the edge-cloud continuum. The framework aims to be versatile enough to allow pulverization in non-aggregate systems, thus expanding its scope of applications beyond aggregate computing.

The framework is hosted in Kotlin multiplatform, which allows the framework to be used on several platforms, including Android, iOS, JVM and native target. This variety of platforms is fundamental to enable the framework to be used in a variety of scenarios, spanning from cloud servers to embedded devices. Another relevant technology adopted in the framework is RabbitMQ, which is a message broker that allows communication between the different components of the system. The decision to use RabbitMQ is justified by its compatibility with multiple protocols, including AMQP, MQTT, and WebSocket, which enables a broad spectrum of supported devices and platforms. Furthermore, RabbitMQ facilitates several communication patterns, such as publish/subscribe, request/reply, and routing, thereby offering significant flexibility in how communications should occur.

To showcase the effectiveness of the framework, some scenarios in the context of CPS have been identified by using the framework to deploy such systems, highlighting the potential that pulverization has as a methodology. In particular, the framework is used in conjunction with *embedded systems* to recreate a heterogeneous infrastructure where the framework runs on.

For what concerns the testing of the framework and performance evaluation, an exhaustive test suite (composed of unit and integration tests) has been developed to validate the operational semantics of the framework. Moreover, as stated above, the framework is tested in real scenarios by deploying a set of demos that showcase the potential of the framework and its resilience to failures. In future work, evaluating the performance of various deployment strategies of a given system and their impact on communication in terms of latency and throughput represents a relevant topic to be investigated.

Thesis Structure. Accordingly, the remainder of this thesis is structured as follows. Chapter 2 discusses the background and related works. Chapter 3 summarize the requirements of the framework and give an overview of relevant deployment scenarios that are worth to be considered during the validation of the framework. Chapter 4 presents the framework and its architectural design. Chapter 5 describes the implementation details of the framework. Chapter 6 shows the validation process of the framework, including the experimental setup and the results obtained by the demos. Finally, Chapter 7 concludes this thesis by summarizing its main contribution, with a focus on future works.

Chapter 2

Background

This chapter is organized into three sections. The first section provides an overview of the current layered and heterogeneous infrastructure defined by the could-edge interplay. The second section describes the problem of the deployment independence of the applications by giving an overview of the actual frameworks and methodologies. Finally, the third section describes the pulverization methodology in aggregate computing and cyber-physical systems.

2.1 Layered and heterogeneous infrastructure

Nowadays, electronic devices are capable of generating vast amounts of data, from measuring natural phenomena to human behavior. The growth of the Internet of Things (IoT) is expected to connect virtually all objects, leading to a need to transfer, store, and process unprecedented amounts of data.

Cloud computing has become an accessible platform for storing and processing data for a variety of applications, including IoT devices. It offers flexibility and low initial costs, but its adoption has exposed limitations in fulfilling requirements for real-time, low-latency, and mobile applications. Centralized cloud data centers are often physically and logically distant from the client, requiring multiple hops and causing delays and consuming network bandwidth.

The adoption of cloud computing and the increasing ability of edge devices to generate and consume heterogeneous data requires new distributed computing infrastructures that can handle diverse application requirements. Recent computing infrastructures that enact applications at edge devices have improved response time and reduced bandwidth use. Fog computing has emerged as a paradigm that combines the ability to run localized applications at the edge with the high capacity of the cloud, supporting the heterogeneous requirements of both small and large applications through multiple layers of the computational infrastructure.

Taking advantage of the characteristics of the edge-cloud continuum enables many opportunities in the Internet of Things field, like having systems that comply with requirements such as security, data locality, real-time computation, etc. Nevertheless, the deployment of applications in this infrastructure is still a challenge and several approaches have been proposed to address this issue [1].

In the following section will be provided an overview of the main aspects and challenges that demonstrate the suitability of combining edge, fog, and cloud computing for various applications used by the Internet of Things.

2.1.1 Cloud Fog and Edge interplay

This section introduces the concepts and terminology of cloud, fog, and edge computing, discussing their main characteristics and finally, how those paradigms can be combined to provide a more flexible and efficient solution for a wide range of applications.

Cloud computing

Over the last decade, cloud computing has become a widely adopted computing paradigm for many applications due to its dynamic characteristics such as elasticity and pay-per-use (achievable via virtualization and containerization), reaching a mature state.

Cloud providers offer on-demand computing through three main models, which are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [7]. IaaS provides users with remote access to computing power as a service, while PaaS offers a platform for software development with necessary libraries and databases to deploy and run applications, and SaaS provides software that relies on cloud providers' infrastructure to offload computing and/or data. The concept of Everything as a Service (XaaS) has emerged, which includes a wide variety of cloud service levels.

Cloud services operate under a Service Level Agreement (SLA) that determines the services offered and the costs for using them. Common pricing models include charging by time unit, amount of data transfer, and the number of requests. Cloud computing's features of elasticity, ubiquitous access, and on-demand provisioning make it an attractive option, allowing for lower upfront investments and faster time to market, with reduced operational costs.

Fog computing

The evolution of hardware in personal devices has increased computing capacity at the edge and the size of mobile devices has shrunk, allowing them to run

applications with reasonable complexity and quality of service (QoS). As a result, distributed computing paradigms are being utilized, where edge devices are used to run applications and store data.

Fog computing creates a bridge between edge devices and the cloud, and introduces a hierarchy of computing capacity, with fog nodes, cloudlets, or micro data centers located between the edge and the cloud. This hierarchy can be spread throughout the network, with nodes higher in the hierarchy having larger computing capacity and serving more users, while nodes lower in the hierarchy are closer to the edge and have lower communication delays.

The computing hierarchy in the fog infrastructure can offer a wider range of service levels, supporting applications that cannot be supported by cloud computing alone. A fog infrastructure can handle applications with a variety of QoS requirements, as applications can run at a hierarchy level that provides adequate processing capacity and meets latency requirements. Another consequence of the use of processing closer to the edge is to reduce (aggregate) bandwidth use in the network along the path between the edge and the cloud.

Edge computing

Edge computing is a distributed computing paradigm that brings computation and data storage closer to where it is needed, reducing the distance that data must travel and minimizing the latency for applications that require quick responses. Edge computing evolved from the growth of mobile devices and the hardware evolution of personal devices.

The combination of higher computing capacity and edge networks enabled distributed computing paradigms that propose the utilization of edge devices to run applications and store data. Edge computing is characterized by the use of devices such as smartphones, tablets, and IoT sensors and actuators as sources of computational and storage resources. These devices have limited computational capacity and battery life but can support application execution and storage capabilities.

Edge computing is expected to enable new applications in areas such as augmented reality, autonomous vehicles, and smart cities, among others, by allowing data processing and analysis to occur closer to the data source, reducing response time and network congestion. Edge computing also enables the collection of data from a variety of sources and the aggregation of data at the edge for further processing, analysis, and decision-making.

Fog computing and edge computing are two related but distinct concepts in the field of distributed computing. However, they are often used interchangeably or confused with one another, which can lead to misunderstandings and miscommunication.

One reason for the confusion is that both fog and edge computing refer to distributed computing infrastructures that process data closer to where it is generated, such as on the edge of the network. They both aim to reduce network latency and bandwidth consumption by processing data locally rather than sending it to a centralized cloud server.

However, the main difference between fog and edge computing lies in the level of hierarchy at which they operate. Edge computing typically refers to processing that occurs at the outermost layer of the network, closer to end-user devices and sensors. It involves lightweight computing devices and microservices that are often embedded in sensors, smartphones, or IoT devices.

Fog computing, on the other hand, involves a hierarchy of computing nodes that are distributed between the edge and the cloud. These nodes, also known as fog nodes, cloudlets, or micro data centers, can be located at access points, routing devices in the network, or even at the core of the network. The idea is to provide a distributed infrastructure that can handle data processing and analysis at different levels of the network hierarchy while minimizing latency and bandwidth usage.

In conclusion, while fog computing and edge computing share similar goals and concepts, they are distinct in their approach and level of hierarchy.

Edge-cloud continuum problems

The edge-cloud continuum presents many challenges that must be addressed to optimize its performance. One of the primary issues is managing the resources distributed across the continuum in a way that ensures efficient resource utilization and maintains QoS levels. This is particularly challenging due to the heterogeneity of devices and applications that comprise the continuum, as well as the dynamic nature of the network topology caused by device mobility and varying application requirements.

The movement of services in the Edge-Cloud infrastructure is an important consideration due to the inherent heterogeneity of the devices and applications in the system. As edge and fog computing become more prevalent, there is a greater need for services to move between devices in the hierarchy to optimize the use of resources and provide the required Quality of Service (QoS). However, managing the automatic adaptation of services to different deployment locations while considering resource constraints at each level of the infrastructure is a major challenge. Additionally, the heterogeneous network topology and frequent changes in device mobility and application requirements make it even more complex.

2.2 Deployment independence

The advantages of integrating different computing paradigms, such as cloud and edge computing, have already been acknowledged by numerous industry and academia-based initiatives, one example is the OpenFog Consortium [1].

The cloud-edge computing integration is an open research topic since each of the two paradigms has its use cases and advantages. The cloud computing paradigm is well suited for large-scale applications that require high computational power and storage capacity. On the other hand, edge computing is well suited for applications that require low latency and high reliability, such as autonomous vehicles, smart cities, and industrial automation. The integration of the two paradigms can provide a more flexible and efficient solution for a wide range of applications. Nevertheless, the integration of the two paradigms is not trivial, and different approaches can be used to tackle this problem.

In this context, we refer to "deployment independence" as the ability of an application to be deployed on any computing infrastructure by separating the business logic from deployment and infrastructure aspects. In this way, the system logic can be developed without considering the underlying infrastructure, since they are orthogonal aspects. This approach, on the one hand, allows for better-engineered systems where aspects of development and deployment are separated; while on the other hand, one can make the best use of the available infrastructure according to the dynamics of the system.

The following section will review the main methodologies that are in the literature and aspire to develop systems that integrate cloud-edge infrastructure.

2.2.1 Actual frameworks and methodologies

Many frameworks and methodologies have been proposed in the literature to handle the edge-cloud continuum problem. The different approaches proposed vary in complexity and use cases, each trying to solve a specific problem.

Among the methodologies and frameworks worth mentioning is osmotic computing which operates in the IoT environment focusing on a three-tier architecture by leveraging microservices that can be moved around the infrastructure and frameworks such as DR-BIP and DReAM that are based on the concept of "motif" and interaction rules and reconfiguration rules to manage system deployment.

The following is an overview of how these two methodologies work, highlighting their main features and how they try to solve the integration problem.

[1] `https://opcfoundation.org/markets-collaboration/openfog/`

Osmotic computing

Osmotic computing [2] utilizes a concept known as a MEL (microelement) to encompass resources, services, and data. In the realm of IoT, MELs can be structured as a graph and relocated across various infrastructures based on factors such as cost, security, privacy, and performance. MELs encapsulate four distinct elements: microservices that provide specific functionality, microdata representing the flow of information to and from sensors or actuators, microcomputing that performs various computational tasks using real-time and historic data, and microactuators that control the state of physical resources using actuators at the network edge.

Each application can be decomposed into (cooperative) subprograms to improve deployability and scalability. This decomposition, in osmotic computing, defines several interacting MELs, which are atomic entities providing simple functionalities. A graph of MELs can include several microservices (MS) and microdata (MD) combined to provide a specific behavior. In osmosis, containers (or virtual components) are used to deploy dynamically and support the migration of MELs across heterogeneous systems.

In osmotic computing, the computing environment is divided into three layers: cloud data centers (L1), edge systems and micro data centers (L2), and IoT devices (L3). At L3, the IoT devices capture raw data from the environment at a fixed frequency or by events. The L2 layer is composed of network devices such as routers, switches and gateways, supported by protocols like *OpenFlow* or hardware that enables network components to be accessed remotely. Finally, L1 is composed of data centers, which are large-scale computing facilities that provide a large number of computing resources and storage capacity. The L2 layer can collect the data coming from devices at L3 enabling the collection of raw data and performing some computations before transferring these data to L1.

Osmotic computing is an extension of elastic resource management, in which the deployment and migration strategies of microelements (MELs) can change over time based on changing infrastructure and application requirements. Osmotic computing automates the configuration and reconfiguration of MELs based on factors such as quality of service, security, and runtime perturbations.

The purpose of an osmotic platform is to balance the needs of both the infrastructure and the applications by automatically relocating microservices to appropriate deployment locations. This approach focuses mainly on systems that are centrally managed and coordinated.

DR-BIP and DReAM framework

The *Dynamic Reconfigurable BIP* framework (DR-BIP) [3] includes three main aspects of dynamism: (I) the ability to describe parametric system coordination for

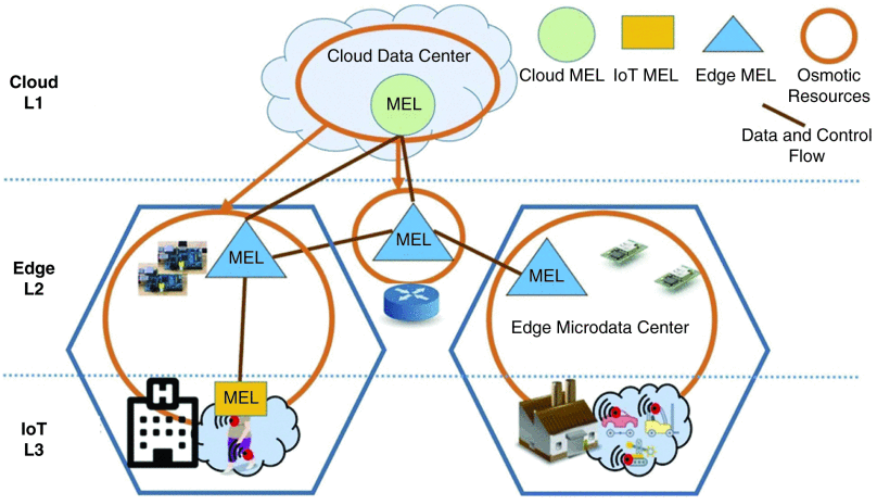


Figure 2.1: Osmotic computing architecture reference. Picture taken from [2].

an arbitrary number of instances of component types, (II) the ability to add/delete components and manage their interaction rules depending on dynamically changing conditions, and (III) allow services to seamlessly continue their activity on any available device or computer (fluid architectures [8]). The *DR-BIP* framework is an extension of the *Behavioral Interconnection Protocol* (BIP) [9] and *Dy-BIP* (a former extension that support dynamic interactions) [4].

The DR-DIP framework provides support for runtime changes in the system, including component creation and removal, migration between motifs, and both programmed and triggered reconfiguration. The use of motifs allows components to interact with others based on their behavior and interaction rules within their new motif, providing a flexible framework for coordination. The platform shares similarities with DReAM [5], but the use of constraints allows for more expressive coordination.

DR-BIP uses motifs as the basic unit for describing dynamic architectures (see Figure 2.2). Each motif includes the behavior of components, the rules for interaction between components, and the rules for reconfiguring the motif, including adding, removing, or moving components. Motifs are structurally organized as the deployment of component instances on a logical map. Maps are arbitrary graph-like structures consisting of interconnected positions. Deployments relate components to positions on the map. The definition of the motif is completed by two sets of rules: (I) the interaction rules, which define the behavior of the components and the interaction between them, and (II) the reconfiguration rules, which define the conditions under which the motif can be reconfigured.

Systems are defined as collections of motif instances, each of which can evolve

independently or in coordination with other motifs through shared components or inter-motif reconfiguration rules. Inter-motif reconfiguration rules also allow for the creation and deletion of motif instances and the exchange of components between motifs.

DR-BIP's behavior in a motif-based system is defined in a compositional way, where every motif has its own set of interactions determined by its local structure. These interactions remain constant until the motif executes a reconfiguration action. In the absence of reconfigurations, the system maintains a fixed architecture and operates like a normal BIP system. Interactions do not affect the architecture, while system and/or motif reconfigurations change the architecture, but do not impact components, meaning running components retain their state, even though new components may be added or removed.

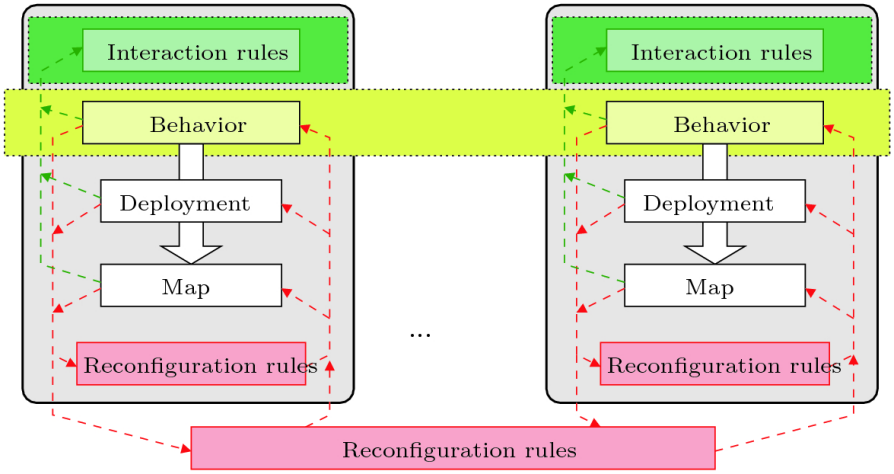


Figure 2.2: Motif-based System Concept. Picture taken from [3].

2.3 Pulverization in Aggregate Computing and CPS

This section presents a brief overview of the state of the art in the field of aggregate computing and cyber-physical with a main focus on the pulverization and which problems try to solve.

Self-organizing systems are a way of engineering distributed intelligence in which the system's global behavior and structure are achieved through the continuous interaction of simple individual components. This approach allows for inherent adaptation to unexpected or unforeseeable situations and has been applied in various contexts, such as human social behavior, and swarm robotics.

Artificial self-organizing systems are software-based systems that regulate their internal structures and behavior without external control, often by mimicking the self-organization mechanisms observed in nature. Self-organizing approaches are applied to distributed cyber-physical systems (CPS), where individual system components interact with each other based on physical proximity to collect and process information generated by distributed sensors and use it to control the system behavior.

However, recent advances in technology have made modern CPS increasingly large-scale, heterogeneous, and dynamic, which makes it challenging to engineer distributed intelligent systems that can be deployed in different contexts and exploit available resources opportunistically. To address this problem, a framework based on the pulverization approach is proposed [6], which breaks the overall system behavior into tiny pieces of computation linked to sensors, actuators, and neighboring components. These sub-components can be deployed and wired separately, allowing for a separation of concerns between the self-organization logic and the deployment context.

The pulverization approach can be implemented in the framework of Aggregate Computing [10], where global self-organizing behavior can be specified declaratively by composing pure functions expressing increasingly complex distributed algorithms. This approach allows for the design of distributed adaptive behavior for large-scale CPS that can be deployed in a deployment-independent way, meaning that the behavioral description of the self-organization logic remains unchanged regardless of the specifics of the deployment context.

The pulverization approach was exercised simulating a CPS whose aim is to reduce the contribution of household winter heating to air pollution by imposing a custom maximum temperature relative to the level of particulate matter (PM) in the area surrounding the household [6]. The system implements the functionality in a self-organization fashion, where there isn't a central coordinator and the system autonomously organizes its behaviour even in the face of disturbance. The goal of the experiment is to show that via the pulverization approach, the system's business logic, defined once, can be reused in different deployment schemes, preserving its functional behavior.

Another initial research effort [11] was made by combining the pulverization approach with the multi-tier programming paradigm [12]. The multi-tier programming paradigm defines a distributed architecture in a single compilation unit with a single language. Once the program is specified, the compiler (or the runtime) is responsible for splitting the computation among different peers. A language that supports multi-tier programming is *ScalaLoci* [13, 14], a type-safe multi-tier language hosted in Scala. A ScalaLoci application is structured through *peers* and *ties,*where peers abstract over the locations representing the components of an

application, while the ties define the connection between peers. Only tied peers can communicate with each other. The example provided in [11] shows how the pulverization approach can be fitted into the multi-tier programming paradigm, by defining a logical node as a peer which in turn is composed of a set of peers representing the pulverized device. Moreover, the example shows the conjunction of ScaFi, a Scala internal DSL that can run on the JVM or in the browser and ScalaLoci showing how this could be the foundation stone of a unified framework living in the Scala ecosystem. Finally, the example shows how different deployment schemes can be defined by changing the ties between peers by preserving the functional behaviour of the system.

Chapter 3

Requirements

This chapter introduces the pulverization approach by explaining the terminologies and concepts that will be used in the rest of the thesis. The main principles of pulverization are presented, followed by a description of the pulverization domain model supported by some relevant examples. Next, the framework's requirements are reported and, finally, the chapter concludes with relevant scenarios that are worth using to test the use and effectiveness of the framework.

3.1 Pulverization domain model

Pulverization is an approach to conceive self-organization in distributed systems that facilitate deployment independence, i.e., the ability of an application to run with no change on various deployments while retaining its original functional semantics. The main idea is to organize the structure and the behaviour of a system in a way that the developer can focus on the logical model, abstracting from the deployment details, scheduling and communication. The logical model can be partitioned into a set of software components that can be deployed on the available infrastructure, while the application logic will preserve the functional goals independently from the actual deployment.

To better formalize the terminology coming from the pulverization, the following *ubiquitous language* [15] is proposed in Table 3.1. The main reasons for the use of this ubiquitous language are to avoid ambiguity and to ensure that the concepts are understood in the same way by anyone who approaches the pulverization.

Henceforth, concepts characterizing pulverization will be used with the meaning given in Table 3.1.

A *Logical Device* is the representation of a device in the system abstracting from the specific deployment details. It is composed of *five* components: *Sensors*, *Actuators*, *Behaviour*, *Communication* and *State* while the interaction between them

Concept	Definition
Sensors	Component that represents a set of logical *sensors*
Actuators	Component that represents a set of logical *actuators*
Behaviour	Component that models the device behaviour
Communication	Component that handles the interaction with neighbours (other devices)
State	Component the holds the representation of the device's knowledge
Thin host	A device that has limited computational power and memory
Thick host	A device that has a powerful computational power and memory
Logical device	A logical representation of a device composed of several components which they can be deployed on the available infrastructure
Logical neighbouring link	Defines a logical connection between two logical devices defining the network topology. The aforementioned structure can change over time.

Table 3.1: Pulverization Ubiquitous language.

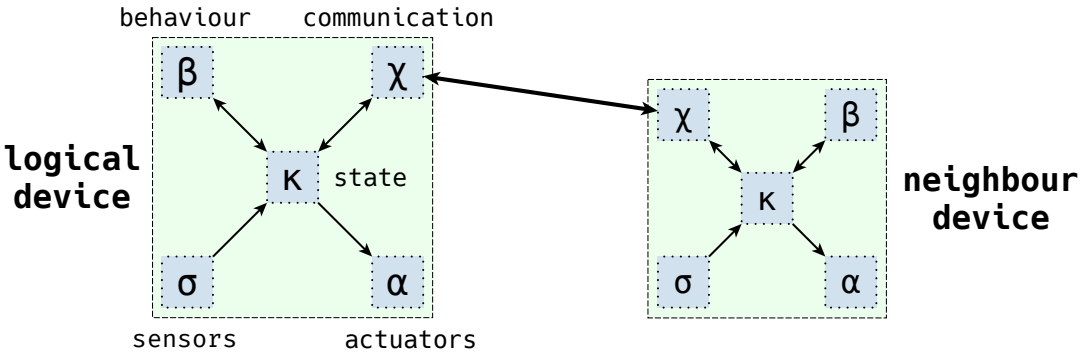


Figure 3.1: Representation of a logical device split into its components and a connection with one of its neighbours

determines the device's logic in the system. The Figure 3.1 shows a representation of a logical device split into its components defining also a link with another device.

The *Sensors* σ and *Actuators* α components represent the way the device interacts with the environment: the former is used to acquire information from the environment, while the latter is used to perform actions on the environment. The *State* κ component represents the device's knowledge and abstract from the actual storage mechanism or representation. The *Communication* χ component handles the interaction with neighbours by holding the information about the identity of the neighbours and how to reach them. The send and receive operations occur through respectively the *input channels* and the *output channels*, where the output channel of a device is connected to the input channel of its neighbours. Finally, the *Behaviour* β component models the device behaviour via a *function* which maps the state of the device to a new state, defines a set of prescriptive actions to be performed and a set of coordination data to be propagated to the neighbours.

Each device performs a MAPE-like cycle that includes the following steps and that defines the interactions between the device's subcomponents as depicted in Figure 3.1, where the arrows denote the message flow:

1. **Context acquisition:** the device acquires information from its sensors and the communication component, storing them in the device state
2. **Computation:** the device behaviour function is computed against the device state
3. **Coordination data propagation:** coordination data is sent to all the neighbour's device
4. **Actuation:** the actuators are activated to execute a set of prescriptive actions

A platform is a collection of physical *hosts* connected by a dynamic graph of physical network links, representing the communication channel between two hosts. A host is an entity with a unique identifier (e.g. an IP address, URI resource, etc.) and can be a computer system, an embedded device holding sensors and actuators, a virtual machine or a software container. The type of communication channel (the link between hosts) may vary depending on the underlying network infrastructure and protocols. The *hosts* types are divided into two categories: *thin hosts* and *thick hosts*. Thin hosts are devices with limited computational power and memory, while thick hosts can compute and may even do so on behalf of multiple logical devices.

The deployment of the CPS can be defined as an *allocation map* placing each component of each device to specific hosts in the platform. An example

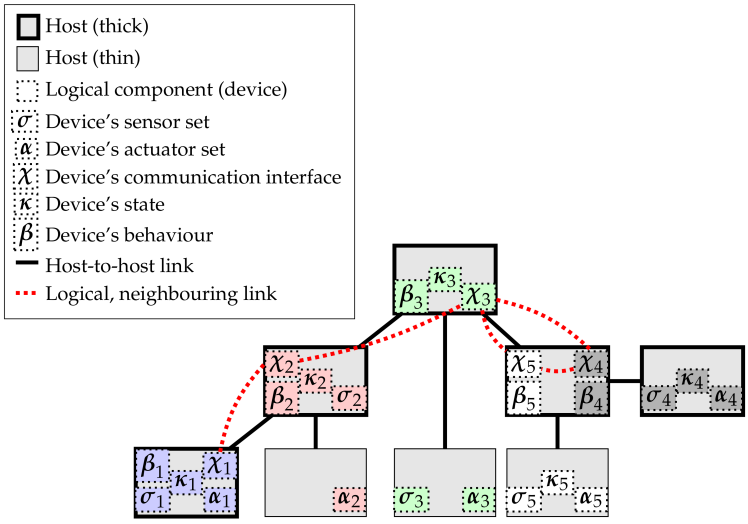


Figure 3.2: Example of instantiation of the CPS model. Picture taken from [6].

of a deployment can be seen in Figure 3.2. In the example is assumed that all the sensors and actuators are deployed on the same host, though this is not a requirement, sensors and actuators may be deployed on different hosts.

Examples of deployment are provided in Figure 3.3 showing an increasing number of responsibilities are centralized. The Figure 3.3a shows a peer-to-peer style deployment where, for each device, all the components are deployed on the same host. This style is not suitable in a sensor network where a device is designed to operate for a long time using solely battery power and is not equipped with enough power to host a β-component. The Figure 3.3b shows a broker-based style deployment where all the χ-components are deployed on a separate host (broker). This is a common scenario in IoT systems where a broker is used to handle the communication between devices. The Figure 3.3c shows "big data in the cloud" style deployment where all the κ-components are deployed in the cloud enabling big-data analysis. The Figure 3.3d shows an embedded device with sensors/actuators style deployment where all the α and σ-components are deployed on the same thin host while all the remaining components are deployed on a thick host. This scenario covers the case of a device with limited computational power and memory by offloading the β-component to a thick host.

3.2 Framework requirements

The main objective of this thesis work is to develop a framework that can "fill the gap" between the modelling of a CPS (and its simulation) and the physical

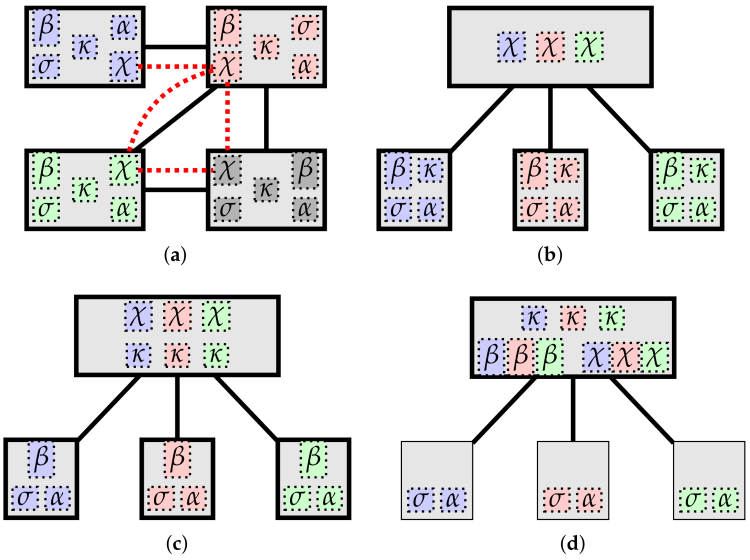


Figure 3.3: **(a)** Peer-to-peer style; **(b)** broker-based style (e.g. MQTT); **(c)** Big data in the cloud style; **(d)** embedded device with sensors/actuators. Picture taken from [6].

deployment of the system on an infrastructure. The framework should model the pulverization concept and provide a clear separation between the behaviour of the overall system and low-level details of the deployment via a simple and concise API. Also, the framework must capture the concepts defined by the pulverizing approach to provide a good starting point on which the framework can be evolved.

Business requirements

As previously mentioned, the main objective of the framework is to provide a way to deploy CPSs via the pulverization approach. The business requirements identified are reported as follows:

- The pulverization approach can be used to simply and effectively deploy CPS systems
- The framework should be flexible to support different deployment strategies
- The framework should be extensible to support different communication protocols

User requirements

The user requirements are identified from the perspective of the developer who will use the framework. The user requirements identified are reported as follows:

- It should be possible to configure the system to deploy by defining the structure of each device *logical device*
- It should be possible to configure the *deployment unit* for each *logical device*
- It should be possible to configure the *logical devices* and *deployment unit* via a DSL
- It should be possible to create a *sensors* component
- It should be possible to create a *actuators* component
- It should be possible to create a *communication* component
- It should be possible to create a *behaviour* component
- It should be possible to create a *state* component

Functional requirements

The functional requirements, obtained from the user requirements, are reported below:

- Multiple *logical devices* can be defined
- For each *logical device*, the *components* that compose it can be defined
- Each *component* defined for a *logical device* must be configured to be deployed on a specific tier of the infrastructure
- A check must be performed to ensure that the configuration of the *logical devices* is valid and consistent
- Links between *logical devices* can be defined
- Multiple deployment units can be defined for each *logical device* using the system configuration
- The user-defined *components* can be added to the deployment unit
- The *deployment unit* can be started

- The *deployment unit* can be stopped
- Prevent the run of the *deployment unit* if the configuration is not honored
- Different protocols can be added to the *deployment unit* to enable intra-component communication
- For each component, a custom implementation of the logic that implements how the communication with other components should occur can be provided

Non-functional requirements

- The framework should be easy to use by providing a simple and clean API simplifying the development of the system and adoption of the framework
- The framework should be extensible in the sense that the user can customize some aspects of the framework like the communication protocols, and the logic of each component
- The framework should be flexible to support different deployment strategies coping with different scenarios and infrastructures
- The framework should support a wide range of architectures to support a heterogeneous set of devices enabling wide adoption of the framework

3.3 Reference scenarios

This section gives examples of scenarios in which the framework can be used to implement a Cyber-Physical System. The proposed scenarios are intended on the one hand to show in what contexts the framework can operate and on the other hand to provide guiding examples of using the framework for users interested in using it.

The following are three examples that can be used as a reference for applying the pulverization approach.

The first example is the simplest scenario where the pulverization can fit in and can be considered the "hello world" of the pulverization. The scenario is a simple system composed of a single *device* whose objective is to control the moisture level of the soil. The *device* is composed of a sensor that measures the moisture level of the soil and an actuator that controls the irrigation system. The objective is to keep the moisture level of the soil at a predefined level. From this description emerge a device composed of the following components: *sensor, actuator, behaviour,* and *state*. The system is deployed on three hosts: two *thin host* which host the *sensor*

and *actuator* components and a *thick host* which hosts the *behaviour* and *state* components. In this scenario, the device is "decomposed" into sub-component that are deployed on different hosts and the communication between the components is handled by the framework obtaining the global behaviour of the device. This example shows the basic building blocks of pulverization by realizing a system with only one device focusing on the use of either *thin hosts* and *thick hosts.*

The second example is a more complex scenario where multiple devices come into play and where communication between them is the main focus. In this example, two types of devices are defined: one that needs to find another device and the device that needs to be found. The first device described may be a smartphone, while the second may be an embedded device with low computational and memory characteristics. The goal of this example is to use smartphones to find the embedded device: the closer the smartphones get to the embedded device, the more intense light the embedded device will produce; conversely, as the smartphones move away, the light will decrease in intensity. Meanwhile, smartphones communicate with each other by exchanging information about the distance to the embedded device that needs to be found. On smartphones are executed the *sensor* and *actuator* components, while *behavior* and *state* components are offloaded to the cloud. Similarly, the embedded device hosts only the *actuator* component while the *behavior* is offloaded to the cloud. As for communication between the devices, all *communication* components are instantiated in the cloud. This scenario emphasizes the communication that takes place between devices in the system: devices are decomposed into their components that are then instantiated in the infrastructure where communication does not reside in the devices themselves but rather on the cloud, nevertheless the correctness of system operation will be preserved.

The last example proposed is quite similar to the previous one but the focus is on having some devices perform the behavior locally while others are offloaded to the cloud. This example is intended to show that it is possible to support different ways of deploying the system seamlessly. The goal of the example is to implement a system that measures the level of aggregation of people (leveraging a mobile device) through, for example, a crowd estimation algorithm. The system consists of many devices that interact with each other and exchange information about the distance between them. This information is then shared with an additional device in the network that is responsible for making an estimate of the crowd and performing an actuation proportional to the computed value. The peculiarity of this deployment lies in the fact that some devices perform the behavior computation on the device itself, while other devices perform the computation offloaded in the cloud. Thus, by varying the system deployment strategy for the same device, we want to observe that the system maintains its functional correctness.

Chapter 4

Design

This chapter discusses the design elements characterizing the framework by illustrating how the pulverization concepts were modeled and what adaptations were made to make the pulverization approach better fit in the framework. For each module of the framework, the relevant design choices are discussed and finally, a detailed description of the interactions between the framework's modules is also provided to better understand the framework's architecture.

4.1 Framework architecture

The framework is articulated in modules: each module takes into account a specific aspect of the pulverization. The modularity of the framework enables from one side, the possibility to use only the needed modules, preventing the bloating of the project; on the other side, modularity allows the customization of some implementations of the framework.

The two fundamentals modules of the pulverization framework are: *core* and *platform* which respectively define the core concepts of pulverization like the type of components and all the logic needed to run the pulverized system like defining the components reference, loading the user-defined components and setup the communications between all of them.

The third module is *rabbitmq-platform* which is highly dependent on the two modules described above and its purpose is to rely on **RabbitMQ**[1] to enable the communications between all the components. This component manages all the low-level aspects related to communication like the connection to the broker, declaring queues and so on.

[1]**RabbitMQ** is an open-source message-broker (or message-oriented-middleware) that originally implement the *AMQP* protocol and has since been extended with a plug-in architecture to support other protocols like *MQTT*.

In Figure 4.1 are represented all the framework's modules and the relationship between them.

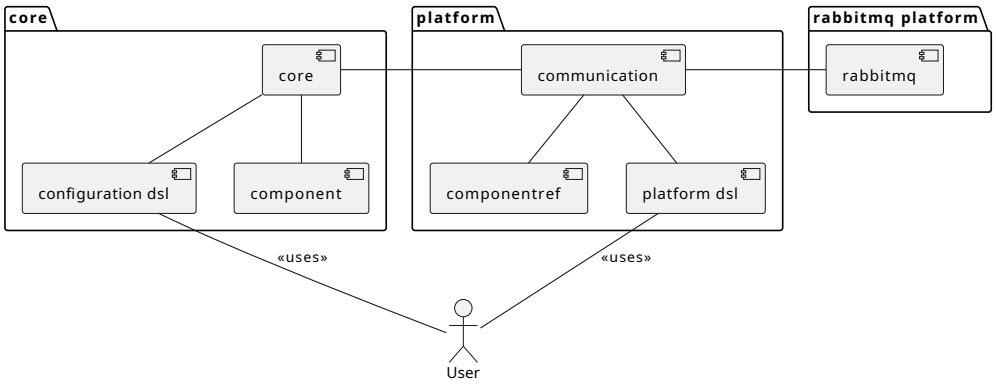


Figure 4.1: Package diagram showing the modules that constitute the framework and their relationship.

In the Table 4.1 are reported a synthetic representation of the modules that constitute the framework with a corresponding description.

Module	**Description**
`Core`	Defines all the pulverization concepts, exposing them as interfaces. Provides a DSL to configure each logical device in the system and a way to define the relationship between them.
`Platform`	Is responsible for executing all the device's components on the available infrastructure. Provides a DSL to configure the platform specifying which components should be used.
`RabbitMQ Platform`	Represents a possible implementation for enabling intra-component communication leveraging RabbitMQ as protocol.

Table 4.1: A tabular representation of the modules that constitute the framework.

The pulverization framework relies on a three-level architecture. Each level of the framework's architecture is designed to use the functionalities of the layer above and makes accessible their functionalities to the layer below.

The described architecture takes with it the implicit "one-way dependency" where the layer below depends on the layer above and not vice versa. The Figure 4.2 depicts the architecture's choice made to design the framework.

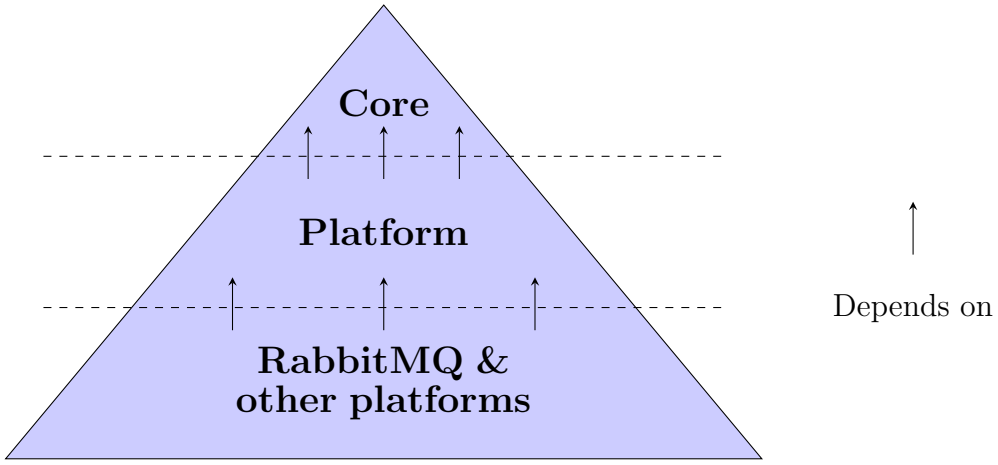


Figure 4.2: Architectural diagram showing how the pulverization framework is designed.

Given that the framework is intended to be used by multiple users, it is essential to reduce the cognitive effort required by users to use it effectively. By prioritizing the minimization of cognitive effort, the framework can be made more accessible, user-friendly, and ultimately, successful in achieving its intended purpose.

To achieve this goal, the framework is designed following a "pyramid-like" architecture (see Figure 4.3) where the tip of the pyramid represents the core pulverization concepts that the user use and implements to build up the system. Those abstraction needs to be as clear as possible from a software engineering perspective to avoid the user to be overwhelmed by the complexity and also because these interfaces depend on the whole framework. As you move down the "pyramid", the complexity of the modules increases but the user's knowledge needed to run the framework decreases.

By designing the framework in this way, we open up different usage scenarios such as a basic use that requires only an understanding of the basic concepts, to advanced uses that require a deep understanding of the framework enabling its complete usage and extension.

The sections below will describe the architectural choices made for each framework's module.

4.1.1 Core module

Architecturally, the *core* module is rather simple. Its simplicity is a consequence of the fact that this module is the main entry point for the user, and the lower the complexity of this module, the faster the user can become familiar with the framework. Moreover, a correct design of the interfaces defined in the *core* module is

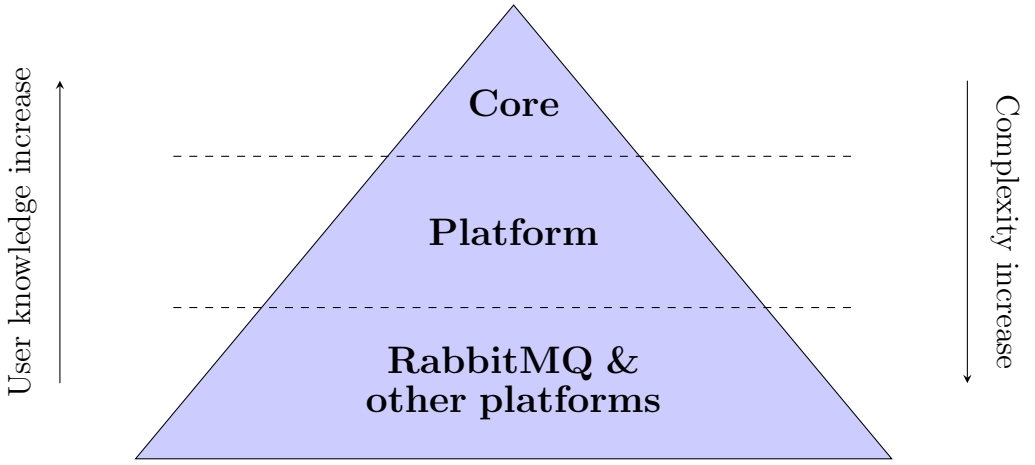


Figure 4.3: A correlation between the framework's complexity and the required user's knowledge to use the framework.

a crucial aspect to consider to be aligned with the pulverization concepts illustrated in the article [6].

The pulverization represents a device as the combination of five components: *state*, *behaviour*, *communication*, *sensors* and *actuators* [6]. All of those concepts are modeled by the framework through interfaces that the user will implement based on the specific scenario.

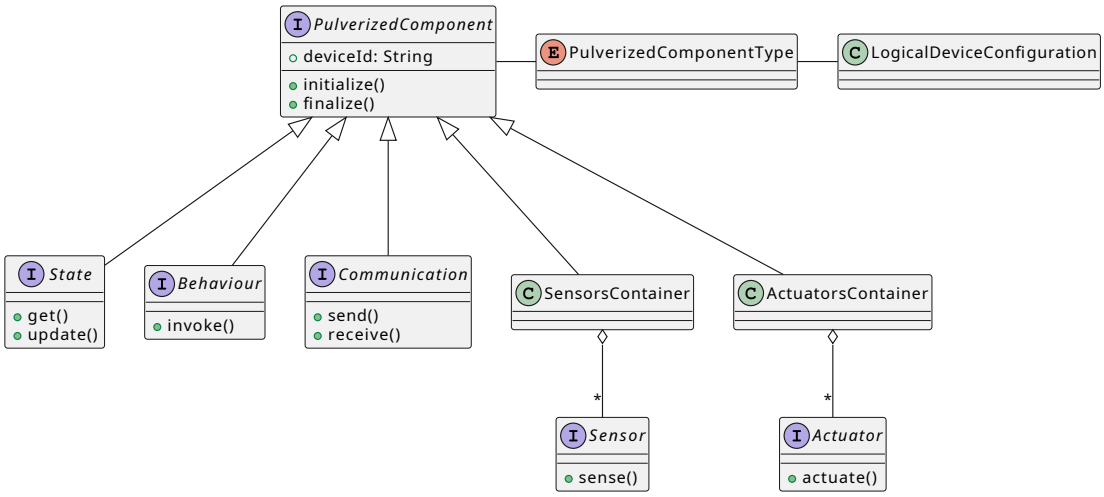


Figure 4.4: Class diagram showing the interfaces defined by the core module.

The Figure 4.4 shows the concepts defined by the core module and their relationship.

This module provides a DSL to generate the configuration needed for the platform to run. In particular, the DSL provides a simple, clean and handy way to create in a declarative fashion how many logical devices should the platform manage, and how those devices are made.

4.1.2 Platform module

The platform module defines the enabling concepts for system execution like intra-component communication and provides an abstraction for representing a remote component and how to reach it; finally, it manages all the machinery needed to run the system.

The highly distributed nature of a "pulverized system" has forced the design of the framework to abstract from the actual place where components are actually deployed; in this way, we avoid the need for the user to specify and manage specific aspects of deployment but can focus solely on application logic while remaining adherent to the objectives of the pulverization, which among many want to separate aspects of deployment from aspects of application logic [6].

Although the abstractions defined in this module are fundamental to the execution of the system, their understanding by the user is not essential. Nevertheless, their understanding becomes crucial when the user wants to extend the framework with new features, like implementing a new protocol to enable intra-communication components.

As said before, communication between components is a fundamental aspect to consider; for this reason, the *communicator* concept comes in. The communicator abstracts the way how the communication between two (pulverized) components occurs. The design of this component abstracts from the message format and the type of the involved components, effectively making the communicator highly generic and delegating all those complexities to the specific communicator implementation used by the platform (see Section 4.1.3).

Finally, the platform module provides a DSL to allow the user to instantiate the platform and then actually run the system. The DSL allows, declaratively, to specify the components intended to be executed in that specific deployment unit, as well as indicate which specific communicator implementation to use. In Figure 4.5 is depicted the overall architecture of the platform module.

4.1.3 Rabbitmq-platform module

This module implements a possible communicator that bases its operation on RabbitMQ. Although this module, at the time of writing, represents the only implementation of a communicator, this does not mean that it should be the

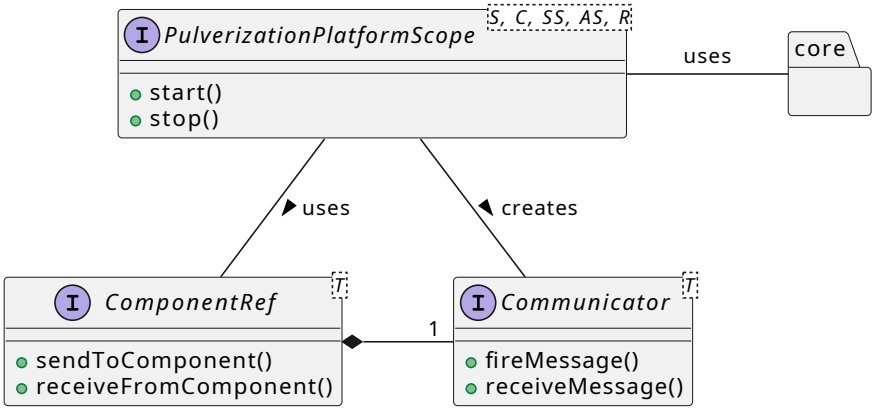


Figure 4.5: Class diagram showing the interfaces defined by the platform module.

only possible solution. Other communicators based on different technologies and infrastructures will likely be implemented in the future.

In this module, all the communication aspects that will be used for communication between components of the pulverized system are defined. The design of the framework delegates to these types of implementations to handle low-level aspects like connections, retry on failure and so on.

While this module (or more generally this kind of module) requires a very good understanding of the concepts defined in section 4.1.2 to be implemented, it requires no cognitive effort on the user side to be used.

4.2 Data flow in the framework

A device, in pulverization, obtains its logical behavior through the interaction of its constituent components. The communications between the components and the related exchange of messages represent fundamental aspects of understanding how the behavior of the device is obtained. The "fragmentation" of the devices into components allows the system to work independently from the specific deployment, focusing entirely on the business logic of the application. Is the responsibility of the framework to take care of the communication between components, and define how those communications should be handled.

The following sections will describe the data flow in the framework, from the perspective of the components and the device.

4.2.1 Components interaction

The original formulation of the pulverization defines the component's interaction as follows (see Figure 4.6):

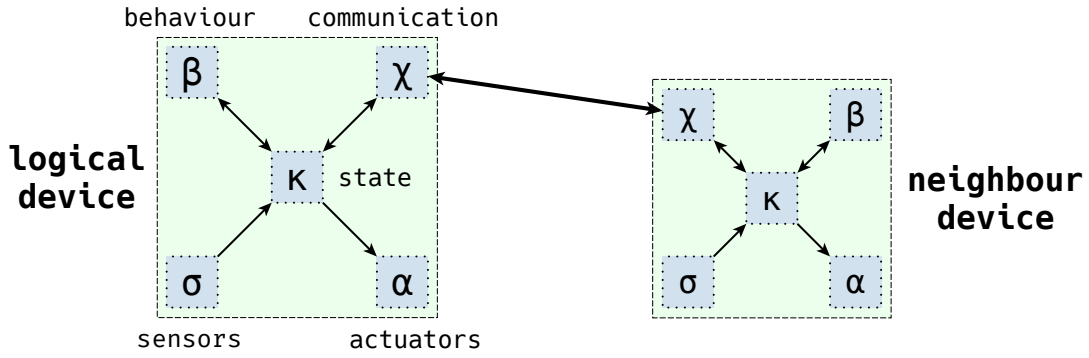


Figure 4.6: Design of the interactions between components proposed in the original article [6].

Four interactions are involved in pulverization:

- **Behaviour** to **State**: the **Behaviour** read from the **State** the *sensed values*, the *communications* and the current *state*, then update the **State** with information like *new communication* to send to all the neighbours, a set of *prescriptive actions* to perform and the *new state*.
- **Sensors** to **State**: the **Sensors** send to the **State** a set of *sensed values*.
- **Actuators** to **State**: the **Actuators** receives from the **State** a set of *prescriptive actions* to perform on the environment.
- **State** to **Communication**: the **State** sends to the **Communication** a *new message* to send to all the neighbours and correspondingly the **Communication** send to the state all the *messages* coming from the neighbours.

Despite this formulation being very clear and reasonable, it requires some extra communication to achieve the result. For example, when the **Behaviour** component computes the *new communication* to send to all the neighbours, it needs to send it to the **State** component, which will then send it to the **Communicator** component. This not represents a problem per se but forces an extra step to complete the communication, resulting in possible inefficient communication and complexity of the framework.

This kind of "extra communication" can be observed also in other component interactions, like the one between **Sensors**/ **Behaviour** and **Actuators**/**Behaviour**.

In all of those cases, the **State** component is involved in the communication creating an extra step.

The framework uses a different formulation of the component's interaction to reduce the extra communication simplifying the overall communication pattern. This formulation is depicted in Figure 4.7.

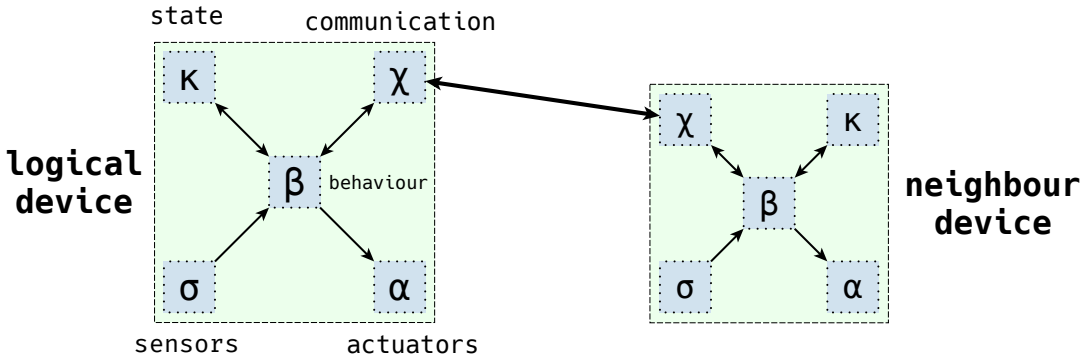


Figure 4.7: Interaction between components changed to best fit into the framework.

This new formulation is based on the fact that the **behaviour** component has a direct dependency on all the other four components, in this way, the **behaviour** component can directly interact with the other components without the need to be intermediated by the **State**.

The component's interaction is now defined as follows:

- **Sensors** to **Behaviour:** the **Sensors** send to the **Behaviour** a set of *sensed values*.
- **Actuators** to **Behaviour:** the **Actuators** receives from the **Behaviour** a set of *prescriptive actions* to perform.
- **Communication** to **Behaviour:** the **Communication** sends to the **Behaviour** a set of *communications* from the neighbours and receives from the **Behaviour** the *new communication* for the neighbours.
- **State** to **Behaviour:** the **Behaviour** read from the **State** the *current state* and write to it the *new state*.

Now, the **Behaviour** component is central in the communication between components, and it is the only component that needs to interact with all the other components.

Below, is presented a detailed description of each component's interaction with the **Behaviour** component.

For what concern the **Sensors** and **Actuators** components, the interaction with the **Behaviour** component is specular: the sensors send to the behaviour the sensed values and the actuators receive from the behaviour the prescriptive actions to perform. The sequence diagrams in Figure 4.8 show the interaction of the sensors and actuator with the behaviour.

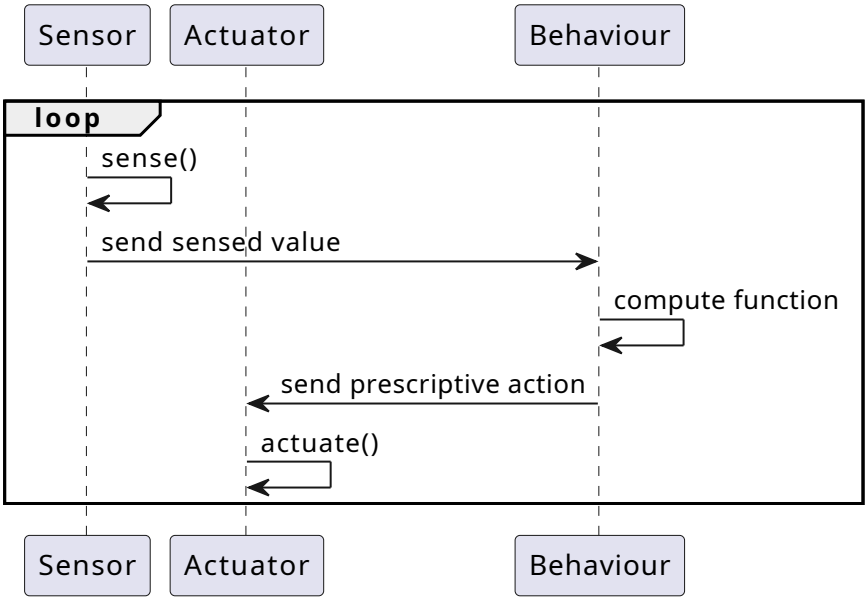


Figure 4.8: Interaction between sensors and actuators with the behaviour.

The **Communication** and **Behaviour** interaction is bidirectional, which means that the communication sends to the behaviour all the messages coming from the neighbours and the behaviour sends to the communication component the new messages that should be propagated to the neighbours. Reasoning on the way this interaction occurs could lead to modeling it using a specific pattern (e.g. synchronous or asynchronous) but is fundamental to abstract over the specific pattern giving the freedom to use the one that better fits the current scenario. The sequence diagrams in Figure 4.9 shows the interaction between the communication and behaviour components using a communication pattern which not represents the only possible one.

Even the **State/Behaviour** interaction is bidirectional. The behaviour queries the state component to get the current state, then, the behaviour computes the new state and writes back to it. Even in this case, the way this interaction is modeled is not relevant and should be abstracted over the specific pattern. The sequence diagrams in Figure 4.10 shows the interaction between the state and behaviour.

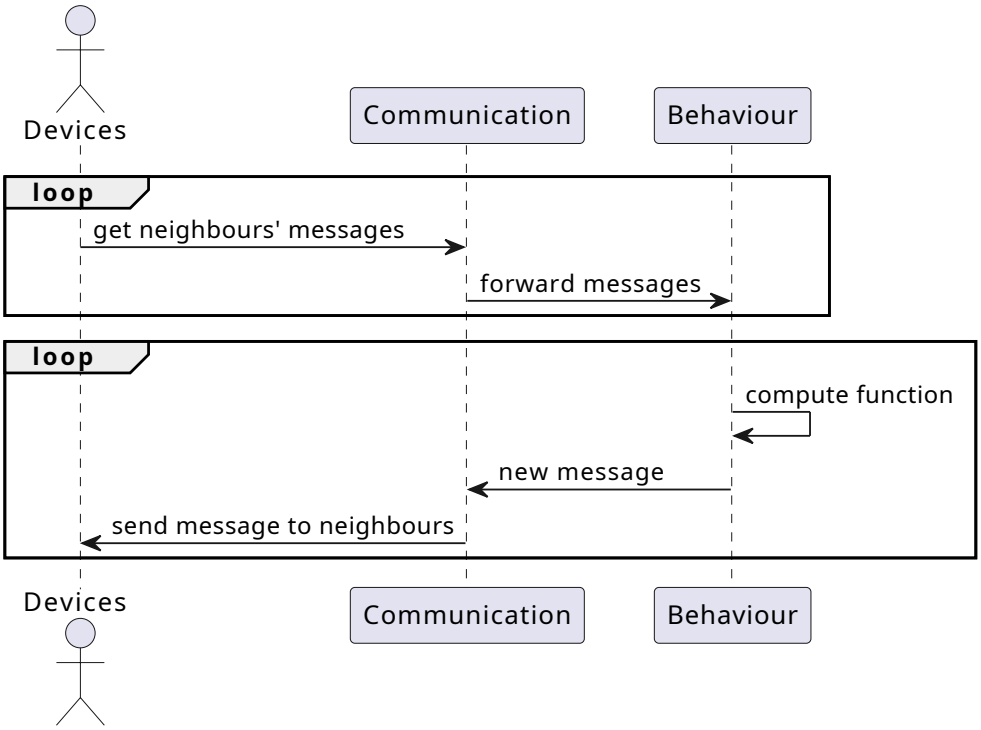


Figure 4.9: Interaction between communication and behaviour.

4.2.2 Device cycle

In the previous section, we looked at what interactions occur between the various components. In this section, we analyze how these interactions are synchronized to ensure the overall operation of the device.

By device cycle, we mean the sequence of operations that the device must perform to execute its functional logic. In the context of pulverization, this cycle is pre-determined and well-structured.

The cycle consists of the following steps, as previously described i Chapter 3:

- *Context acquisition:* the device retrieves information from sensors and communications.
- *Computation:* the behaviour function is applied using the state, sensors and communications, producing an output.
- *Coordination:* the coordination data is sent to all the neighbours.
- *Actuation:* the actuators are activated to execute the prescriptive actions produced by the behaviour.

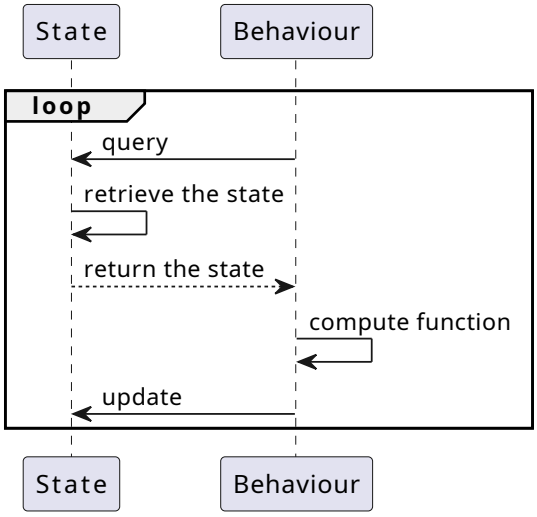


Figure 4.10: Interaction between state and behaviour.

Given the highly distributed nature of pulverization, it is quite complex to manage this cycle properly. For this reason, it is left to the platform to manage any synchronization to ensure that the cycle runs smoothly.

To deal with this problem, a model was created that abstracts from where the various components are deployed, thus creating a uniform level to access the components while delegating to the platform the logic on how to reach the component "physically". In this way it is also possible to make optimizations on communications, e.g., if two components belong to the same deployment unit, then they communicate directly in memory, otherwise, they take advantage of one of the provided implementations to communicate over the network.

The modeling provided for this problem involves the use of two concepts: the *ComponentRef* and the *Communicator*. The former embodies the concept of "reference to a component" abstracting from where the component is physically deployed. In this way, the communication with another component can be done seamlessly. The latter is used by the *ComponentRef* to communicate with the component it refers to. The *Communicator* manages all the low level aspects of the communication, e.g., the protocol used to communicate, the serialization of the data, etc.

Is the responsibility of the platform to create the right *Communicator* for each *ComponentRef* based on the initial configuration given by the user. Separating the reference to a component from how the communication occurs, allows the platform to optimize all the communication and change the communicator based, for example, on the new deployment.

Chapter 5

Implementation

This chapter discusses in detail all the implementation aspects of the framework, showing all its characterizing elements. First, the reasons for choosing Kotlin as the language to implement the framework are explained. Then, for each module, the logic governing it, the main classes, and usage scenarios will be analyzed. Finally, the chapter will conclude with a discussion of the configuration DSL and the platform DSL and how they support the user in the system configuration.

5.1 Languages with multiplatform targets

Pulverization is born in the context of CPS where device heterogeneity is a real scenario. For those reasons, we can deal with networks of embedded devices (which have very limited computational resources) up to networks of computers with high computational power and memory. Nowadays, architectures that combine these two scenarios are increasingly common, thus having to manage architecturally heterogeneous networks of devices.

For these reasons, it is necessary to build a framework that can support a wide range of architectures and platforms to maximize the number of devices on which the framework can run.

In this context, the choice of the language to implement the framework is crucial. A cross-platform language (or even known as multiplatform language) is a language that allows the same code to be compiled for different platforms. The trend over the last years is to use the same language to span over several runtime and VMs (e.g. JVM, JavaScript, native platforms, etc.) in order to reduce the effort of maintaining the codebase and to increase the portability of the application. The other main advantage of using a multiplatform language is that all the shared concepts and logic can be implemented in a single codebase that can be reused in all the specific platforms.

In this way, we can use one programming language and manage the targeting of multiple platforms effectively (see Figure 5.1).

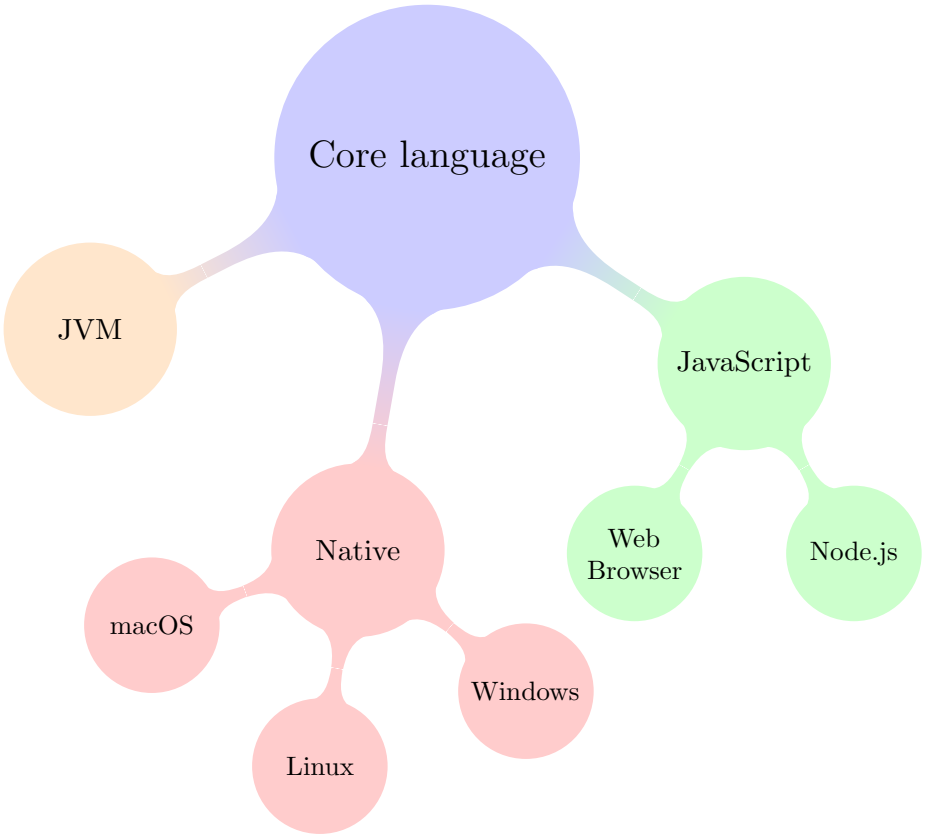


Figure 5.1: Diagram showing the rationale behind the multiplatform languages.

The two following sections will examine two of the main relevant language based on the JVM ecosystem that supports multiplatform targets.

5.1.1 Scala Language

The Scala programming language is a general-purpose programming language that is designed to combine object-oriented and functional programming in one concise, high-level language.

Although Scala was born under the JVM, it has been extended to support other platforms such as JavaScript, and native platforms.

Support for cross-platform is enabled through external plugins and not directly by the Scala compiler itself. The plugin enables compiler extensions allowing the generation of *intermediate representations* (IRs) containing platform-specific as-

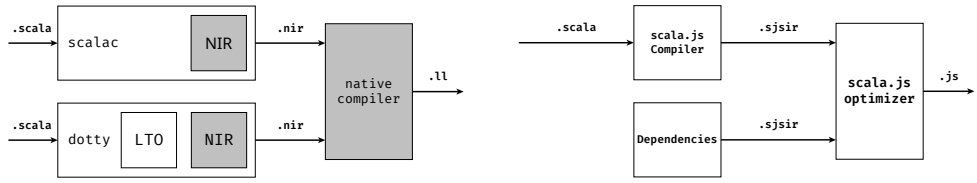
pects; with the IR, the compiler makes optimization, linking and other dependencies management.

The Listing 5.1 shows the minimal configuration required to enable the cross-platform support for the Scala language, in particular are enabled JVM, JavaScript and native platforms.

```
import sbtcrossproject.CrossPlugin.autoImport.
  crossProject

lazy val root = project(file("."))
    .crossProject(
      JSPlatform,
      JVMPlatform,
      NativePlatform,
    )
    .settings(
      name := "project-name",
      scalaVersion := "3.2.2",
    )
```

Listing 5.1: Minimal configuration to enable cross-platform support for Scala.



(a) Scala native compilation pipeline. (b) Scala.js compilation pipeline.

Figure 5.2: Representation of the Scala Native and Scala.js IR generation and compilation pipelines.

The Figure 5.2 depicts the two main targets and for each one shows the compilation pipeline in particular, the Figure 5.2a shows the compilation pipeline for the native target, while the Figure 5.2b shows the compilation pipeline for the JavaScript target.

For what concern the native target, the compilation pipeline is composed of the following steps [1]:

[1] https://scala-native.org/en/stable/contrib/compiler.html

- **Scala code compiled into Native Intermediate Representation:** the `nscplugin` takes the Scala source code and inspecting the AST, it generates the `.nir` files.

- **LLVM final compilation:** all the `.nir` files are compiled into `.ll` files and passed to the LLVM compiler that produces the native binary file.

The pipeline for the JavaScript target is quite more articulated but for the sake of simplicity, are reported only the main phases of the compilation steps [16]:

- **Generation of the Scala.js IR:** the `scalajs-compiler` takes the Scala source code and generates the `.sjsir` files.

- **Optimization (linking):** in this phase the `scalajs-optimizer` takes the `.sjsir` files and performs optimizations taking also other `.sjsir` files coming from dependencies.

- **Output file:** the `scalajs-optimizer` generates the `.js` file that is the output of the compilation.

During the first step, the `.scala` source files are compiled with scalac, augmented with the Scala.js compiler plugin. The compiler plugin takes the internal compiler ASTs that have been lowered to contain JVM-style classes, interfaces, methods and instructions, and turns them into Scala.js IR (`.sjsir` files).

The `.sjsir` files are similar to `.class` files, although they are AST-based (instead of stack-machine-based) and contain features dedicated to JavaScript interoperability. The `.sjsir` format and specification are independent of Scala: meaning that the linker is independent of the language version.

Scala multiplatform ecosystem

This section will examine the current ecosystem concerning the multiplatform support for the Scala language to have more awareness about the usability of this technology. In particular, will be addressed two main factors: the number of libraries that supports multiplatform targeting and the maturity of each platform.

At first glance, there is a strong sense of fragmentation of projects and communities. Some communities pervasively support all three platforms (JVM, JS, and native) while others do not, and this is reflected in the amount of multi-targeting compatible libraries. For example, the *typelevel* ecosystem supports all three platforms for all their main libraries: **Cats Effects** [2] and **FS2** [3]. From the other

[2] `https://typelevel.org/cats-effect/`
[3] `https://fs2.io/`

side, the *zio* [4] ecosystem supports only JVM and JavaScript while the native platform is an experimental stage. Even if the actual number of libraries is not so high, the Scala ecosystem is quite mature to be used in its multiplatform version. Moreover, there is a lot of work being done by the Scala community to improve the multiplatform support for the language, so it's very likely that in the future the number of libraries targeting multiplatform will increase.

To complete the analysis and to have a more complete picture of the Scala multiplatform ecosystem, will be examined the maturity of each platform. Starting from *Scala.js*, the platform is quite mature and stable: the project was born several years ago and it's used in production by many companies. Year to year several improvements were made to reach a high level of performance and stability [17].

Finally, the *Scala Native* platform works quite well but has some limitations that make it not suitable for all production use. One of the biggest and most important limitations is the lack of support for multithreading [5]. If for some projects this limitation is not a problem, for others it can be a big issue; nevertheless, the *typelevel* community dealt with this restriction by implementing an event-loop-based concurrency model to support native projects. Another limitation is represented by the supported architectures: at the time of writing, only a subset of the platforms supported by LLVM can be targeted, which means that not all the embedded devices can be supported.

5.1.2 Kotlin Language

Kotlin is a cross-platform, statically typed, general-purpose high-level programming language. It's designed to interoperate fully with Java but also compile to JavaScript or native code via LLVM.

Kotlin multiplatform is designed to simplify the development of cross-platform projects by reducing the time spent writing and maintaining the same code for different platforms.

The Kotlin multiplatform use cases can be synthesized in the following points:

- **Android and iOS applications** sharing the code between mobile platforms enable the building of cross-platform mobile applications sharing the common code between Android and iOS.

- **Full-stack web applications** when building web applications, it's possible to share the code between the client and the server reusing the same logic on both sides.

[4] `https://zio.dev/`

[5] `https://typelevel.org/blog/2022/09/19/typelevel-native.html`

- **Multiplatform libraries** a multiplatform library with common code and its platform-specific implementations for JVM, JS, and Native platforms can be created. Once published, a multiplatform library can be used in other cross-platform projects as a dependency.

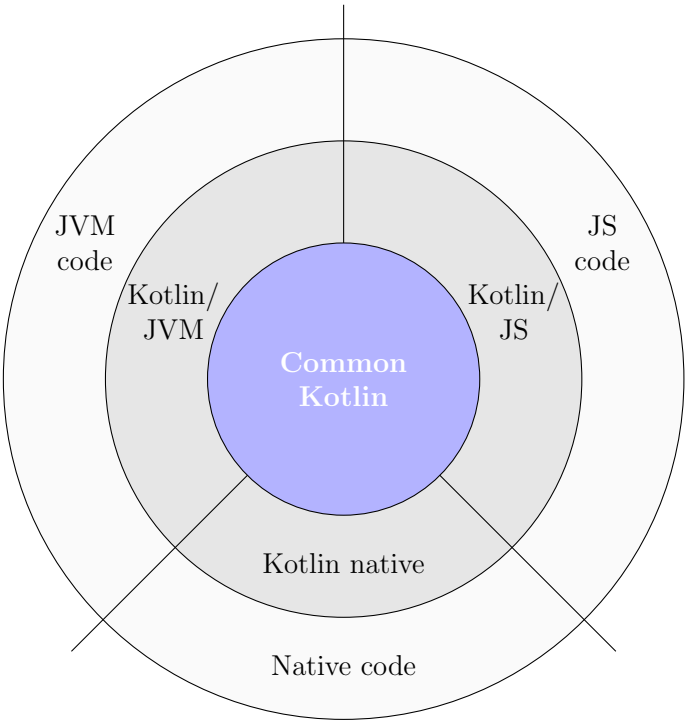


Figure 5.3: Kotlin multiplatform structure.

The Kotlin multiplatform works using a structure (see Figure 5.3) where the common code is at the center and works everywhere on all platforms, and to interoperate with platforms, a specific version of Kotlin is used that includes platform-specific libraries and tools. Through these platforms, you can access the platform's native code and leverage all native capabilities.

Similarly to Scala, the multiplatform support for Kotlin is enabled via a Gradle [6] plugin. As for Scala, the plugin enables a series of tools and compiler extensions to support multiplatform development.

The Listing 5.2 shows a basic setup of Kotlin multiplatform using the Gradle plugin.

[6] **Gradle** is a build automation tool for multi-language software development. It's based on *Apache Ant* and *Apache Maven* introducing a Groovy and Kotlin DSL. The main supported languages are *Java, Kotlin, Groovy* and *Scala*.

To share code between all the platforms, Kotlin provides a specific mechanism using a hierarchical structure of modules. The common code is placed in the `commonMain` module and it's used to share the common business logic that applies to all the platforms. Often there is the need to create several native targets that could potentially reuse a lot of the common logic and third-party APIs. Kotlin allows to create a specific and flexible structure to reuse as much code as possible between the different targets. The Figure 5.4 shows a possible representation of a Kotlin multiplatform project hierarchical structure.

```
plugins {
    kotlin("multiplatform") version "1.8.10"
}

kotlin {
    jvm()
    android()

    ios()
    watchos()
    tvos()

    linuxX64(); linuxArm64()
    mingwX64();
    macosX64(); macosArm64()
}
```

Listing 5.2: Minimal Example of Kotlin multiplatform setup using Gradle.

To access the platform-specific APIs from the shared code, Kotlin provides a specific mechanism called *expect/actual* declarations [7]. With this mechanism, a common source set defines an expected declaration, and platform source sets must provide the actual declaration that corresponds to the expected declaration.

The *expect/actual* mechanism is shown in Figure 5.5 where a class in the common source set is marked as `expect` and the platform-specific source set provides the actual implementation of the class leveraging platform-specific API.

The compiler ensures that every declaration marked as `expect` has a corresponding declaration marked as `actual` in the corresponding platform modules. In this way, is guaranteed that every platform has an implementation for that class or function.

[7] `https://kotlinlang.org/docs/multiplatform-connect-to-apis.html`

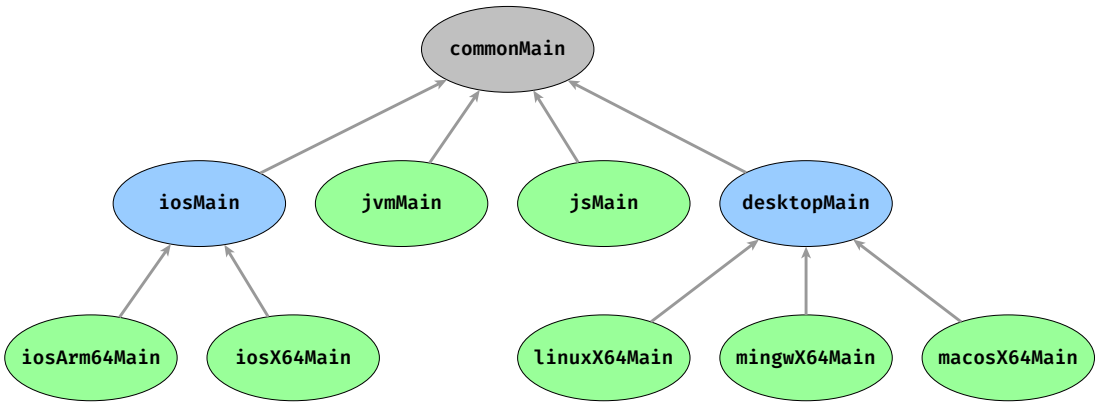


Figure 5.4: Kotlin multiplatform hierarchical structure.

The following will be a brief introduction to how Kotlin can generate native code and js code from the same code base.

The *Kotlin/JS IR* compiler is responsible for compiling Kotlin code into JavaScript code. The compiler backend rather than generating directly JavaScript code generates an intermediate representation (IR) of the code which is subsequently compiled into JavaScript code. This strategy enables aggressive optimizations, improving, for example, the generated code size.

Similarly, the *Kotlin/Native* compiler is responsible for compiling Kotlin code into native code. The compiler is available for all the main operating systems (macOS, Linux, and Windows) and supports different targets like **iOS, Windows, macOS, Linux, Raspberry PI,** and **WebAssembly**. Unfortunately, there aren't details about how the compiler pipeline works, but it's possible to see that the compiler generates an IR representation of the code and then compiles it into native code via LLVM. A relevant feature of *Kotlin/Native* is the interoperability with *C* code. This feature allows using existing C libraries in Kotlin using the *cinterop* tools by generating Kotlin bindings for the C library. The *cinterop* tool requires a `.def` file that describes what `.a/.so` libraries to include in the build and the corresponding `.h` files to parse. Finally, the *cinterop* tool generates a Kotlin library that can be used in the Kotlin code.

When a *Kotlin/Native* library is distributed, a special file with extension `.klib` is generated. This file contains all the information and file specifics for each platform. It's a `.zip` file containing a predefined directory structure: given the `foo.klib` file, when unpacked as `foo/`, contains the following files and directories:

- in the folder with the *component name* is contained the serialized Kotlin IR
- in the folder `targets` are placed the platform-specific files, in particular in the folder `kotlin` there is Kotlin compiled into LLVM bitcode; in the `native`

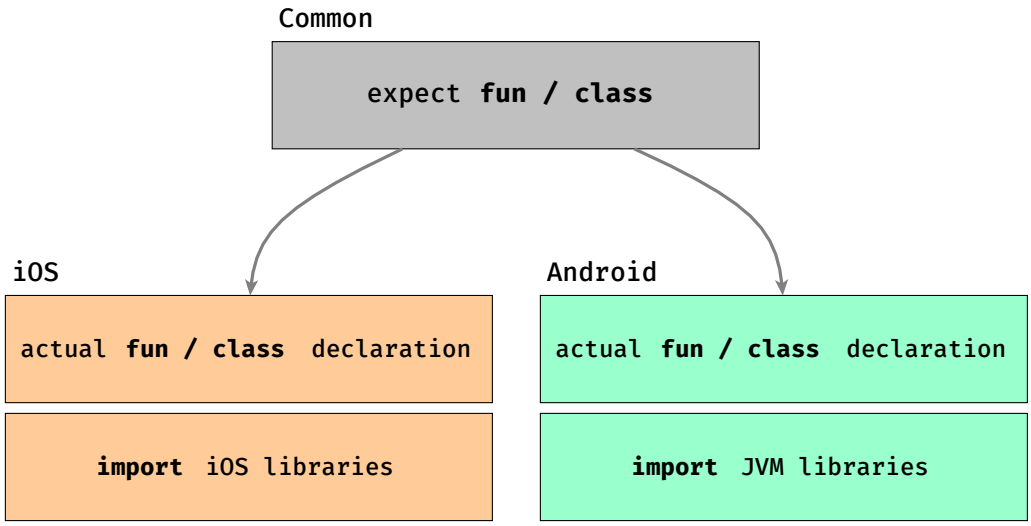


Figure 5.5: Kotlin multiplatform `expect/actual` mechanism.

folder there are the bitcode files of additional native objects

- the `linkdata` folder contains a set of *ProtoBuf* [8] files with serialized linkage metadata
- the `resources` folder contains resources such as images, fonts, and other files
- the `manifest` file in the java property format describing the library.

This structure allows having a single `.klib` file that can be used on different platforms without the need to recompile the library for each one of them. In a sense, the `.klib` file is a portable binary format that can be used on different platforms.

Kotlin multiplatform ecosystem

As already done for Scala, the ecosystem will be examined for Kotlin to get a better awareness of the usability of this technology. Again, the number of supported libraries and the maturity of the framework will be considered to evaluate the adoption of this technology.

Differently from the Scala multiplatform ecosystem, the Kotlin one seems to be more coherent and structured: lots of libraries like **kotlinx.serialization**, **kotlinx.coroutines**, and **ktor** are available for all the target platforms supported

[8] Protocol Buffers (Protobuf) is a free and open-source cross-platform data format used to serialize structured data.

by Kotlin. The other difference is that most of the libraries are developed directly (or with the support of) the Kotlin team, meaning that support and the development is more aligned and coherent with the language itself. The identification of the magnitude of libraries targeting Kotlin multiplatform is quite simple since a site [9] collects all the available Kotlin multiplatform libraries. From this site can be seen that more than 140 libraries are available for Kotlin multiplatform, spacing from different categories and applications. Of course, not all the available libraries are collected on this site, but it's a good starting point to get an idea of the ecosystem.

For concern the maturity of the framework, the Kotlin multiplatform is still in beta, nevertheless, its stability and usability make it a good candidate for a production-ready product. The Kotlin team is working hard to improve the framework and make it more stable and usable. If in Scala the specific module for each supported platform (JS and native) is developed by an external community, reducing the guarantee of stability and coherence with the language, in Kotlin the multiplatform module is developed by the JetBrains team itself, increasing the guarantee of stability of the entire framework.

5.1.3 Why Kotlin multiplatform as a choice

This section will present the reasons that led to the use of Kotlin as a language for the framework implementation instead of Scala. The decision was based on three main factors: the *support* of the overall multiplatform ecosystem, *libraries availability* with multiplatform support and the number of *supported platforms* by each language.

As for *support* related to Scala multiplatform, the issue is controversial. On the one hand, there's Scala.js, which has always been supported and maintained and has reached a very high level of maturity over time. On the other hand, scala native is managed by another community that has contributed to the project in a seesaw manner over time, where in some cases the project had even been abandoned. On the other side, Kotlin multiplatform is entirely supported by JetBrains and is constantly evolving and improving. JetBrains has proven over time to carefully curate its products ensuring very high-quality standards, a symptom that Kotlin multiplatform may also fall into this case.

The *availability* of libraries that support multiplatform is a key aspect to consider: having a large number of supported available libraries could be strategic to develop complex applications. In this regard, the Kotlin multiplatform has a clear advantage over Scala multiplatform: the Kotlin ecosystem is much more structured and coherent than the Scala one. The Kotlin team has developed a large number of libraries that support multiplatform, and the community has also

[9] `https://libs.kmp.icerock.dev/`

contributed to the development of many other libraries.

Finally, the *number of supported platforms* is the most important factor to consider: without extensive support of the most common architectures and platforms, the adoption of the pulverization framework could be limited. In this regard, Scala native is not clear about which specific targets supports or it will support in the future, but for sure currently, it does not support all the Apple mobile ecosystem like iOS, watchOS and tvOS. On the other hand, Kotlin native has wide support for the Apple mobile ecosystem and also for the Linux platform targeting ARM32, ARM64 and x86_64 architectures as well as for the Windows platform targeting x86 and x86_64 architectures.

For all the reasons mentioned above, Kotlin multiplatform was chosen as the framework implementation language.

5.2 Technologies used in the framework

This section offers a comprehensive summary of the various technologies employed across the framework's different modules. Specifically, the key libraries and mechanisms used in the framework will be highlighted, with a focus on those that are shared across all modules. It should be noted that technologies and dependencies unique to each module will be discussed in their respective sections.

5.2.1 Kotlin coroutines

Asynchronous programming returned to the mainstream in recent years. Created in the '80s, but with the advent of multi-core processors, it has been superseded by multi-threaded programming, which was the de facto standard for concurrent computations. Since the 2000s, lots of programming languages have introduced asynchronous programming features from the start while other languages have added support later.

Multithreading is a traditional method of performing multiple computations in parallel by executing each computation in a separate thread. However, it has drawbacks such as increased programming complexity and reduced performance for tasks that are IO-bound. Different flavours of asynchronous programming are an alternative to multithreading. Unlike multithreading, which is based on coarse-grained threads, asynchronous programming is implemented via fine-grained suspendable computations, which can more effectively interleave with each other. Different attempts have been made over the years to provide programming languages with facilities to support asynchronous programming: *callbacks, futures* and *promise* are only some of the most common approaches. A callback is a function defined by the user, passed to a callback-aware API as a function value, lambda, function pointer,

etc., with the purpose of being called at a later time when a certain condition is met. Although this is a rather simple mechanism, callback-based frameworks are known to suffer from a complication of code structure disproportionate to the complexity of the logic that code expresses, commonly referred to as "callback hell" [18]. Futures and promises are a step above callback-based computations, working as special proxies for not-yet-completed results of asynchronous computations. The two main operations that can be performed on a promise are: checking for completion and getting the result of the computation. Promises are a good solution to the callback hell problem, but they are not without their problems: they are not composable, and they lack a way to express dependencies between asynchronous computations. *async/await* is an approach that brings asynchronous programming as a first-class language citizen: *async/await* is based around expressing asynchronous computations as two interconnected parts.

The first language to introduce *async/await* was C#. A lot of programming languages used the C# approach as an inspiration and follow its async/await implementation. JavaScript, TypeScript, Dart, Python, Rust — all these languages use async functions containing await operators.

As follow is introduced the concept of the coroutine and how it is implemented in Kotlin, discussing the basic building block for asynchronous programming.

Even though the concepts of *coroutine* being used for over 50 years, there is no standard definition of what a coroutine is. One valid definition could be "function which can suspend and resume its execution, preserving the state between suspensions" [19]. Coroutines have been classified using the following axes, where implementations may differ: **Symmetric/Asymmetric Control Transfer** and **Stackfull/Stackless Implementation**.

Symmetric coroutines can suspend themselves and resume the execution to an arbitrary coroutine, in this context the control transfer between coroutines is symmetrical. On the other hand, asymmetric coroutines can only suspend themselves and resume the execution to the coroutine that suspended them, in this context the control transfer between coroutines is asymmetric. While symmetric coroutines are more expressive than asymmetric coroutines, they are also more complex to understand, for this reason, most of the implementation of coroutines are asymmetric [19].

A stackful coroutine implementation allows for suspension at any point within nested functions and, when resumed, continues execution from the exact point of suspension, restoring the original function call stack. Conversely, a stackless implementation can only be suspended within itself, requiring that nested functions also be coroutines for asynchronous execution to occur. The majority of modern languages use stackless coroutines. While stackful coroutines are more powerful than stackless ones, stackless coroutines can match most (if not the same) capabilities

via careful handling of nested coroutine calls; moreover, stackful coroutines are noticeably harder to implement efficiently.

One important aspect when implementing coroutines is the *error handling* mechanism. There are two approaches currently used in asynchronous programming, the first one is based on supervision trees, which was pioneered in Erlang, and involves arranging asynchronous tasks in parent-child trees for desired error propagation. The second approach, structured concurrency [10], aims to transfer the idea of structured programming to asynchronous programming. It connects tasks to their origins and ensures that the lifetime of a task cannot exceed the lifetime of its origin, thereby establishing a *launcher-launchee* relation and describing error and cancellation propagation.

Structured concurrency, in comparison to supervision trees, has a fixed error-handling strategy and is therefore less flexible. However, it is more concise and aligns well with the typical way asynchronous code is written. This is why structured concurrency has gained significant attention in recent years, either as structured concurrency libraries or as a built-in language feature [11]. Kotlin uses the structured concurrency approach to manage error handling in coroutines.

Kotlin coroutines are built upon the following goals [19]:

- **Independence from low-level platforms implementation:** Kotlin being a multi-platform language, building its asynchronous support on existing implementations like futures in JVM would lead to interoperability issues between platforms. Therefore, a unique approach is necessary to ensure seamless asynchronous support across all platforms.

- **Adaptability to existing implementation:** a strong emphasis is placed on interoperability with existing code, particularly with Java code on the JVM platform. To ensure a smooth experience for developers, Kotlin should support seamless integration of established asynchronous APIs, such as promises in JavaScript or non-blocking input/output in JVM.

- **Support for pragmatic asynchronous programming:** the popularity of the async/await approach in asynchronous programming highlights the significance of code readability. Although less expressive than full coroutines, async/await offers adequate coverage for practical use cases and improved performance.

Kotlin asynchronous programming is built around the "suspending function" concept, similar to the "async function" concept in JavaScript. A suspending

[10] `https://250bpm.com/blog:71/`

[11] `https://wiki.openjdk.org/display/loom/Structured+Concurrency`

```
suspend fun guessLocaleFromText(text: String): Locale {
    // locale detection implementation
}

suspend fun guessWebPageLocale (url: URL): Locale {
    val text = HttpClient().get<String>(url)
    val localeGuess = guessLocaleFromText(text)
    return localeGuess
}
```

Listing 5.3: Example of a suspending function.

function is marked with the `suspend` modifier and can be called from other suspending functions. However, their call sites are *not marked* as `await`, i.e., calls to suspending functions are implicitly awaited. In this way, the problem of "forgotten await", characterizing the *async/await* approaches, is avoided.

The Listing 5.3 shows an example of a suspending function, which is called from another suspending function showing how in Kotlin suspensive functions are called implicitly. In the example can be seen that the suspending function is implicitly awaited, and the result is returned to the caller.

As follow, a brief introduction to how Kotlin coroutines are implemented is presented. Each suspendable function goes through a transformation from normally invoked function to a continuation-passing style (CPS). For a suspendable function with $p_1, p_2, \ldots, p_n$ parameters and result type T, a new function is generated with an additional parameter of type `Continuation<T>` and the return type change to `Any?`. Also, the calling convention changes: the function may either *suspend* or *return*. When the function returns some result, this result is directly returned from the function (as usual); if the function suspends, it returns the special value `COROUTINE_SUSPENDED`. The compiler takes care of the transformation, so the developer does not need to worry about it. When the user wants to suspend a coroutine's execution, they access the coroutine's continuation by calling an intrinsic function `suspendCoroutineUninterceptedOrReturn`. Then, stores the continuation object to resume it later. Finally, pass the `COROUTINE_SUSPENDED` the intrinsic which is then returned from the function.

Kotlin implements suspendable functions as state machines since such implementation does not require runtime support. While being able to wrap different existing frameworks for asynchronous computations, Kotlin coroutines also allow writing asynchronous code in different styles: *async/await* style, using *Channels* or *Generator*.

Coroutines were widely used in the framework to implement asynchronous

behaviors idiomatically in Kotlin. In addition, since in the framework, the exchange of messages between components can occur asynchronously, this was captured through a coroutine-based construct called Flow [12].

Kotlin Flow is a new addition to the Kotlin coroutines library, and it provides a way to perform asynchronous, sequential computations that emit values and can be transformed into streams. A Flow is a sequence of values that are emitted asynchronously and can be processed in a non-blocking manner. Unlike traditional streams, Flows are also cancellable, meaning that they can be stopped at any point in their execution. Flows are designed to be highly composable, allowing the building of complex, multi-step computations by combining and transforming smaller Flows. They are also designed to work well with coroutines, making them an excellent choice for asynchronous and reactive programming in Kotlin.

For those reasons, *coroutine* and *Flow* are the main tools used in the framework providing a simple and efficient way to implement asynchronous behaviors.

5.2.2 Dependency Injection: Koin

Dependency Injection is a software design pattern that allows for the separation of concerns in a software application. It enables the creation of loosely-coupled code, which is more flexible and easier to maintain. Dependency Injection is achieved through the injection of dependencies, or required objects, into the objects that need them. This can be done either manually or through the use of a Dependency Injection framework.

One such framework is the *Koin* library, which is a popular, lightweight, and pragmatic Dependency Injection solution for Android and Kotlin. The library offers a simple and efficient API, with no reflection or code generation required, making it a fast and efficient choice.

Koin uses a modular approach to Dependency Injection, allowing the definition of dependencies in modules, assembling and managing them easily. This makes it easy to maintain and update dependencies as needed. In addition, Koin offers a range of features, including scope management, property injection, and support for multi-threading, making it a comprehensive solution for Dependency Injection in Android and Kotlin projects.

The Listing 5.4 shows how to use Koin to inject a dependency into a class.

In this example, a data class `MySimpleClass` is defined, and a module `myModule` is also defined to provide an instance of the class as a singleton. The `startKoin` function is then used to start Koin and use the defined module. Finally, the dependency is injected into `MyClass` using the by `inject()` property delegate. The main function creates an instance of MyClass and prints the message from the

[12] `https://kotlinlang.org/docs/flow.html`

```
data class MySimpleClass (val message: String)
startKoin {
    module {
        single { MySimpleClass("Hello from Koin") }
    }
}
class MyClass {
    val mySimpleClass: MySimpleClass by inject()
}

fun main() {
    val myClass = MyClass()
    println(myClass.mySimpleClass.message)
}
```

Listing 5.4: Example of Koin usage.

injected `MySimpleClass`.

The use of Koin in the framework allows for the easy injection of dependencies into the different components of the framework, making it easy to maintain and update the framework as needed. Moreover, the use of Koin allows for the easy creation of new components, without an excessive amount of boilerplate code.

5.3 Core module

The `core` module models the core concepts of the framework. The Figure 5.6 shows the package structure of the module, which is divided into two main packages: `core` and `dsl`.

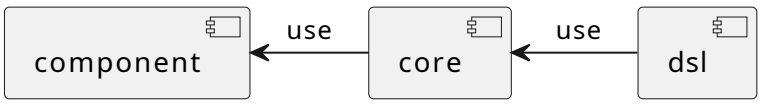


Figure 5.6: Core module package diagram.

The `core` package contains all the interfaces that model the pulverization components. In particular, in Table 5.1 are listed the fundamental interface defined in the `core` package.

All the interfaces illustrated in Table 5.1 are based on another concept expressed by the `Initializable` interface. The `Initializable` interface is used to model the

Interface	Description
`Behaviour`	The interface that models the behaviour of a component
`SensorsContainer`	The interface that models all the sensors belonging to a device
`ActuatorsContainer`	The interface that models all the actuators belonging to a device
`State`	The interface that represents the state of a device
`Communication`	The interface that models the capability of a device to communicate with other devices

Table 5.1: Core interfaces

initialization of a component. Every component, before being used, should allocate resources or perform some operations before becoming operative, as well as release those resources when the component should be destroyed. The `Initializable` interface is used to model this concept and is implemented by defining two methods: `initialize` and `finalize`. The former must be invoked before the use of the component, while the latter must be invoked when the component is no longer needed, during the finalization step of the system.

Analyzing the common aspects of the components defined in Table 5.1, it is possible to identify another common concept that can be isolated in a single common interface: the context in which the component is executed. In this specific scenario, the `Context` interface holds information about the specific device that components belong to, representing this information with the field `deviceID`.

All those two concepts are implemented in a third interface that models the concept of "generic pulverization component" called `PulverizedComponent`, which is the base interface for all the components defined in Table 5.1. This interface implements also an external interface named `KoinComponent`; this interface enables field injection for all the class that implements it. This feature is used to dynamically inject the `Context` inside of each component.

The Figure 5.7 shows the class diagram that models the relationship between the core interfaces of the framework illustrated previously.

Sensors and Actuators

The modeling of the concepts of "sensors" and "actuators" within the framework and the implementation choices made to model these concepts in the framework are discussed in more detail below.

All the considerations and choices made to model the concept of *sensor* hold also for the concept of *actuator*, so for the sake of brevity, only the discussion about

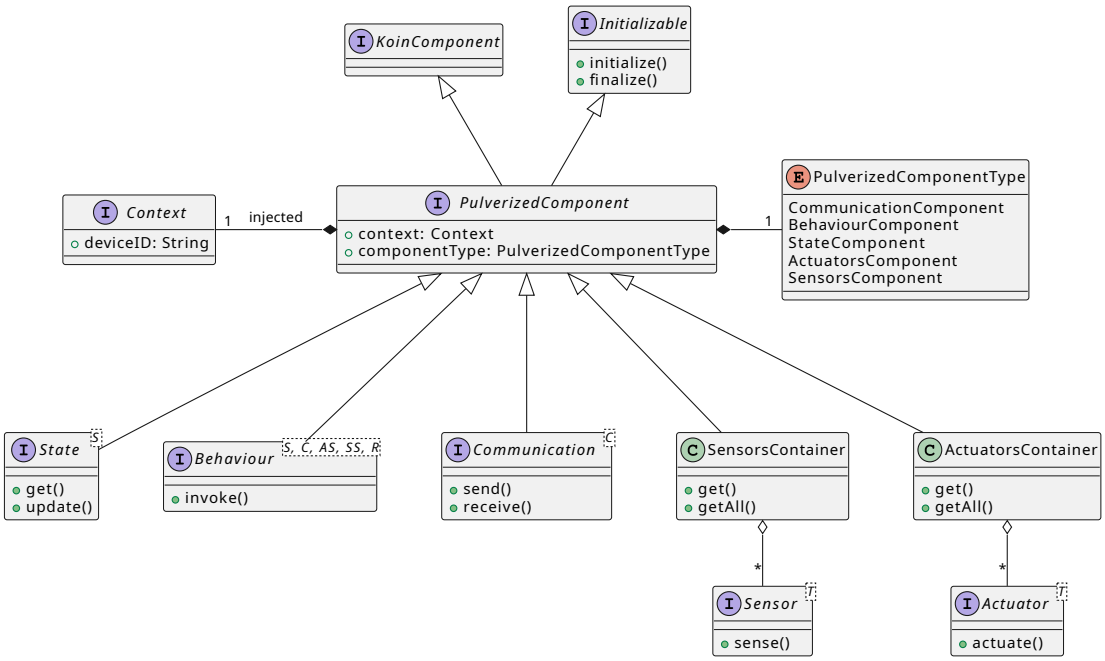


Figure 5.7: Class diagram showing the relation between the `core` interfaces of the framework.

the *sensor* concept is reported.

The formulation presented in the original paper [6] defines the sensors module as *"a set σ of logical sensors"*; starting from this assertion the **sensor** component is broken down into two separate concepts: the *sensor* which models the physical sensor from which the data is collected, and the *sensors container* which represents the collector of the sensors belonging to a specific device.

The `Sensor` interface, represented in Listing 5.5, models a single sensor defining the method `sense` which is use to perform the operation of *sensing the environment.*

```
interface Sensor<out T> : Initializable {
    suspend fun sense(): T
}
```

Listing 5.5: Sensor interface defined in the framework.

The interface is generic in a type variable `T` that represents the type of data collected by the sensor and implements the `Initializable` interface to model the ability of initialization of the sensor.

More complex and with relevant design choices is the `SensorContainer` class.

```
abstract class SensorsContainer : PulverizedComponent {
    private var sensors: Set<Sensor<*>> = emptySet()

    operator fun <P, S : Sensor<P>>
    plusAssign(sensor: S) {
        sensors = sensors + sensor
    }

    fun <P, S : Sensor<P>>
    addAll(vararg allSensor: S) {
        sensors = sensors + allSensor.toSet()
    }

    operator fun <T, S : Sensor<T>>
    get(type: KClass<S>): S? =
        sensors.firstOrNull(type::isInstance) as? S

    fun <T, S : Sensor<T>>
    getAll(type: KClass<S>): Set<S> =
        sensors.mapNotNull { e ->
            e.takeIf { type.isInstance(it) } as? S
        }.toSet()

    inline fun <reified S : Sensor<*>>
    get(): S? = this[S::class]

    inline fun <reified S : Sensor<*>>
    get(run: S.() -> Unit) = this[S::class]?.run()
}
```

Listing 5.6: Implementation of the `SensorsContainer` using inline methods and `KClass`.

The sensors' container is modeled via an abstract class which implement the `PulverizationComponent` interface making the container itself initializable. The storing of the sensors is done via a `Set` of `Sensor` objects; since the `Sensor` interface is generic, it is not possible to determine which type of sensor will be used, for this reason, the Kotlin type projection over the type variable `T` of the `Sensor` is used. In this way, the `SensorContainer` can store sensors of different types, and it will be the responsibility of the container to provide methods that allow retrieving the sensors

of a specific type in a type-safe way. In this regard, the `SensorContainer` provides three-way of retrieving the sensors: `get<T>()`, `getAll<T>()` and `get<T>(run)`. The first method returns a single sensor of type `T` if it is present in the container, otherwise, it returns `null`. This method should be used with care considering that if more sensors with type `T` are present in the container, only one of them will be returned with no guarantee on which one will be returned. This method is useful when in the container there is only one sensor of the specified type. The second method returns a `Set` of sensors of type `T` if at least one sensor of that type is present in the container, otherwise, an empty `Set` is returned. The third method finds the sensor of type `T` and invokes the `run` function over the retrieved sensor if it is not `null`, otherwise, it does nothing.

One of the challenges faced in implementing these methods stemmed from the presence of type erasure. The type erasure is a feature of the Java virtual machine that removes the type information from the compiled bytecode, making it impossible to retrieve the type of a generic type variable at runtime. To overcome this limitation, the Kotlin mechanism of *reified type parameters* is used. This mechanism leverages another mechanism of the Kotlin language called *inline functions* that allows inlining the body of a function inside the caller function. In this way, a sort of "local monomorphization" is performed, allowing the retrieval of the type of the generic type cleanly, without the need of using reflection via `KClass<T>`.

Above was discussed the problem of retrieving a sensor in a type-safe way, this problem is solved by inspecting all the objects stored in the container and returning only those that are an instance of the type `T`. The complete implementation of the `SensorContainer` class is reported in Listing 5.6.

As stated above, all the considerations and choices made for the `Sensor` and `SensorContainer` are valid also for the `Actuator` and `ActuatorContainer` classes.

Communication

The other relevant component in the *core* module is the **communication** component, which is responsible for the communication between the devices. The communication component is described as *"A communication component χ handling interaction with neighbours, holding information on the identity of neighbors and how to reach them, managing input channels used to receive external messages into the device's state, and output channels for emitting messages to all its neighbours"*. From this description emerges the bidirectional nature of communication where sending and receiving channels are neatly separated. Another aspect that emerges from the description is the need to hold a reference to the neighbors of the device and their identity.

In light of the considerations made above, it was decided to demand the user

```
interface Communication<P : Any> : PulverizedComponent {
    override val componentType: PulverizedComponentType
        get() = CommunicationComponent

    suspend fun send(payload: P)

    fun receive(): Flow<P>
}
```

Listing 5.7: Communication interface defined in the framework.

the responsibility of holding the references to the neighbors and their identity, and to provide a simple interface to send and receive messages. The rationale behind this choice is that is quite difficult to provide a generic representation of the network topology and determine how it can change over time, and for this reason, it is better to leave the responsibility of managing the neighbors to the specific implementation of the communication component.

The `Communication` interface, presented in Listing 5.7, models the communication component and defines two methods: `send` and `receive`.

The interface is generic in a type variable `P` that represents the type of messages that can be sent and received by the component. Moreover, the type variable is bounded to the `Any` type: this captures the fact that the message could not be nullable. While at first analysis it might seem redundant to specify this type bound, it turns out to be fundamental since in the type hierarchy in Kotlin, the topmost type is `Any?` which represents any nullable data type, enabling the possibility of sending and receiving `null` messages, a scenario that is not desirable.

While the `send` method is straightforward, the `receive` method returns a specific type: `Flow<P>`. This type represents an asynchronous stream of values that in this case represents the messages received by all the neighbours.

Behaviour

The **behaviour** component is the heart of the device, it is responsible for the execution of the device's logic. The behaviour component is described as *"A computation function β modeling the device behavior, which maps the state of the device to a new state, a prescriptive set of actuations to be performed, and coordination messages to be emitted"*.

The `Behaviour` interface, represented in Listing 5.8, models the behaviour component and defines a single method: `invoke`. Moreover, in the listing are also presented the data classes used to represent the output of the behaviour function.

```
data class BehaviourOutput<S, E, A, O>(
    val newState: S,
    val newExport: E,
    val actuations: A,
    val outcome: O,
)

interface Behaviour<S, E, W, A, O> :
    PulverizedComponent {
    override val componentType = BehaviourComponent
    operator fun invoke(
        state: S, export: List<E>, sensedValues: W
    ): BehaviourOutput<S, E, A, O>
}
```

Listing 5.8: Behaviour interface defined by the framework.

How the behaviour function should be obtained can be easily inferred from the description above: it should take as input the current state of the device, the sensed values and the received messages, and it should return the new state of the device, the actuations to be performed and the messages to be sent. For this reason, the interface is generic in five type variables: `S` for the state, `E` for the communication, `W` for the sensed values, `A` for the actuations and `O` for the outcome of the function. The output produced by the behaviour function is represented by the `BehaviourOutput` data class, which is defined in Listing 5.8. This class holds information about the new state, the new communication, the actuations to be performed, and the function's outcome.

The peculiarity of the `Behaviour` class is that the method `invoke` is marked as `operator` which allows the invocation of the function using the `()` operator over the class instance.

State

Finally, the **state** component is responsible for the representation of the state of the device. The state component is described as *"A state κ, representing the device's local knowledge"*.

The framework should abstract the representation of the state of the device, and for this reason, the `State` interface, represented in Listing 5.9, models the state component and defines two methods: `get` and `update`. The former is used to retrieve the current value of the state, while the latter is used to update the

```
interface State<S : Any> : PulverizedComponent {
    override val componentType: PulverizedComponentType
        get() = StateComponent

    fun get(): S

    fun update(newState: S): S
}
```

Listing 5.9: State interface defined by the framework.

state with a new value. Is the responsibility of the user to implement the concrete representation of the state, and for this reason, the interface is generic in a type variable `S` that represents the type of the state. Is also the responsibility of the user to persist the state somehow, the reason why the interface does not provide any details about persistence.

The other relevant construct of this module is the configuration DSL, which will be discussed more in detail in the Section 5.6.

5.4 Platform module

This module represents the most important part of the framework since it is the one that provides the implementation of the platform. It is organized in a *communication* package that contains the main abstraction for intra-components communication, a *componentsref* package which contains all the interfaces needed to uniform the representation of a component reference, a *context* package that contains the logics for the context creation and, finally, a *dsl* package that contains the DSL used to configure the pulverization platform. The Figure 5.8 shows the package diagram of the `platform` module.

Communication package

In this package are given the main abstractions for intra-components communication. In particular, the main relevant interface is `Communicator`: it is the interface that represents the communication between two components of a logical device. This interface is developed keeping in mind the fact that the concrete implementation could be based on any kind of communication medium, and for this reason, specific protocol aspects are abstracted over. This specific abstraction is captured by

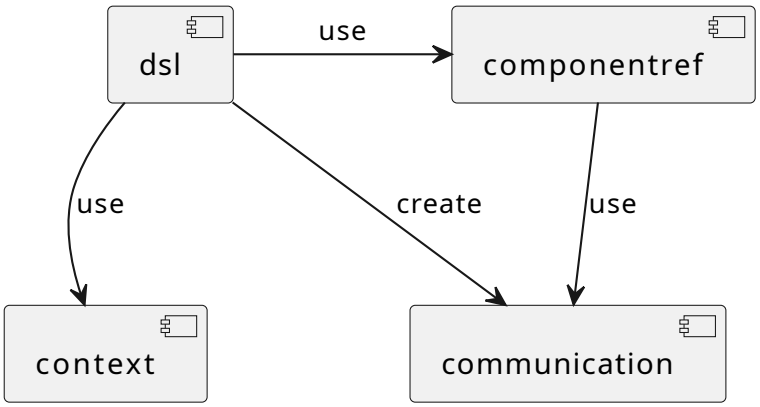


Figure 5.8: Package diagram of the `platform` module showing the relationship between the packages.

the `RemotePlace` data class that represents the remote place that the receiver's component is deployed on. This class is composed of two fields named `who` and `where:` the former identifies the receiver's component answering the question of "who is the other component", while the latter represents where the component is located, answering the question of "where can I reach the other components". In this way, each communicator will specify how to represent that two fields using protocol-specific aspects: for example, a communicator based on TCP socket could represent the `who` field as the IP address of the receiver's component, while the `where` field could be the port. In conjunction with the `RemotePlace` class, the `RemotePlaceProvider` interface is used to provide the specific remote place based on the specific protocol and context. The interface provides a method `get` that takes as input a `PulverizedComponentType` and return a `RemotePlace` if any. The implementation of this interface is demanded by the specific communicator implementation.

The `Communicator` interface, reported in Listing 5.10, has a `setup` method that accept as arguments respectively the `Binding` and a `RemotePlace`; those two arguments are used to set up the communication between the two components (defined by the `Binding`) and how to establish the connection (using the `RemotePlace`). The `finalize` method is used to close the connection between the two components and release resources if any.

The framework, by default, provides an implementation of the `Communicator` interface called `LocalCommunicator` that is used to implement the communication between two components that coexist in the same deployment unit. This implementation is based on the use of `Flow` which represents an asynchronous flow, in particular, two `Flow` are used to implement the communication: one for receiving

```
interface Communicator {

    suspend fun setup(
        binding: Binding, remotePlace: RemotePlace?
    )

    suspend fun finalize()

    suspend fun fireMessage(message: ByteArray)

    fun receiveMessage(): Flow<ByteArray>
}
```

Listing 5.10: Communicator interface defined by the framework.

messages (`inbox` flow) and one for sending messages (`outbox` flow). The `outbox` flow is used by the `fireMessage` method to send messages to the other component, while the `inbox` flow is used by the `receiveMessage` method to receive messages from the other component. This specific implementation is used by the platform to enable communication between components in memory in the same deployment unit.

An important aspect to consider is the fact that the `Flow`s used by the `LocalCommunicator` should be shared across all the local instances otherwise each instance will have its own `Flow` and the communication will not occur. To overcome this issue the `LocalCommunicator` uses another class called `CommManager` which is responsible for managing the `Flow`s. The `CommManager` is a singleton that is used to create the `Flow`s and to share them across all the local instances. The `CommManager` relies on the `lazy` property of Kotlin to create the `Flow`s only when they are needed and give the same flow instance on subsequent calls. The `lazy` property can guarantee a different level of thread safety, in particular, the `lazy` property accepts a parameter of type `LazyThreadSafetyMode` which specifies the behaviour of the lazy property. In this case, is specified `LazyThreadSafetyMode.PUBLICATION` which means that the `Flow` will be created only once and the same instance will be returned on subsequent calls, obtaining the desired behaviour.

Componentsref package

In this package are modeled the interfaces that are used to represent a component reference. In the framework, the concept of "component reference" is introduced to abstract over specific aspects like the protocol used to communicate with the

component and the way how a component can be reached.

```
internal class ComponentRefImpl<S : Any>(
    private val serializer: KSerializer<S>,
    private val binding: Binding,
    private val communicator: Communicator,
) : ComponentRef<S>, KoinComponent {
    private val remotePlaceProvider by inject()
    private var last: S? = null

    suspend fun setup() {
        communicator.setup(
            binding, remotePlaceProvider[binding.second]
        )
    }
    suspend fun sendToComponent(message: S) {
        communicator.fireMessage(message.serialize())
    }
    suspend fun receiveFromComponent(): Flow<S> {
        return communicator.receiveMessage()
            .map { it.deserialize() }
            .onEach { last = it }
    }
    suspend fun receiveLastFromComponent(): S? = last
}
```

Listing 5.11: Implementation of the `ComponentRef` interface.

The `ComponentRef` interface is the main interface that represents a component reference; is generic in a type parameter `S` that represents the type of message that the component can send and receive. Even in this case, the type parameter is bounded to `Any` to prevent the use of nullable types. The interface provides three main methods: `sendToComponent`, `receiveFromComponent` and `receiveLastFromComponent`.

This interface is entirely managed by the platform which is responsible for creating the component reference and for providing the right implementation based on the context in which the component is deployed. For this reason, the end user should not care about the implementation of this interface and should not extend or implement it. As a design choice, all the implementations of this interface are marked as `internal` so that they are not visible outside the platform module.

The `ComponentRef` interface can be used in two main scenarios: the first one is when the component to which we refer exists in the same deployment unit or remotely, and the second one is when the component to which we refer not exists at all (e.g when a device does not have one of the five components).

For the first scenario, the `ComponentRefImpl` provides an implementation of the `ComponentRef` interface, while the second scenario is handled by the `NoOpComponentRef` that is an implementation of the `ComponentRef` interface that does nothing (see Listing 5.11).

The `ComponentRefImpl` defines a three-argument constructor that accepts respectively a *serializer*, a *binding* and a *communicator*. The *serializer* is used to serialize and deserialize the messages that the component can send and receive, while the *binding* is used to identify the component to which we refer (and consequently the source component) and the *communicator* is used to establish the communication between the two components. The implementation of this class is quite straightforward, in particular, the `sendToComponent` method uses the `fireMessage` method of the *communicator* to send the message to the other component, while the `receiveFromComponent` and `receiveLastFromComponent` methods use the `receiveMessage` method of the *communicator* to receive messages from the other component. The serializer is used to serialize and deserialize the messages that the component can send and receive, while the binding is used to setup the communicator. A notable aspect is how the serializer is managed: the serialization is managed via a Kotlin compiler plugin which enriches any class annotated with `@Serializable` with the corresponding serializer; in this way on any serializable class can be used the method `serializer<C>()` to retrieve the serializer of the class, but this method is inline so can not be used in the constructor of the `ComponentRefImpl` class because of type erasure. To overcome this issue, inside the *companion object* of the class is defined the operator `invoke` as an inlined function that accepts only the *binding* and *communicator* as parameters, while the *serializer* is obtained via the reified type, using the aforementioned `serializer` method. Since the operator `invoke` enable the use of the `()` operator, the class constructor is emulated reducing the boilerplate needed to create the class.

The implementation of the `NoOpComponentRef`, shown on the Listing 5.12, provides a no-ops implementation of the `ComponentRef` interface. The implementation provides that the send operation does nothing, while the receive operations always return `null` for the `receiveLastFromComponent` and an empty flow for `receiveFromComponent` method.

Context package

The context package defines only a function called `createContext` which is responsible for creating the context that will be made available to all the components.

```
internal class NoOpComponentRef<S : Any> :
    ComponentRef<S> {
    override suspend fun sendToComponent(message: S) {}

    override suspend fun receiveFromComponent(): Flow<S>
        = emptyFlow()

    override suspend fun receiveLastFromComponent(): S?
        = null
}
```

Listing 5.12: Implementation of the `ComponentRef` interface that does nothing.

Since this function could have a specific implementation base on the target platform in which the framework is deployed, the function is marked as `expect` so that the specific target platform can provide the implementation. At the time of writing, only the JVM target platform is implemented with the following behaviour: first of all, the `.pulverization.env` file is searched in the given path, if the file is not found, the `DEVICE_ID` environment variable is searched, if the variable is not found, the function raises an exception.

Dsl package

The last package is *dsl* which contains the DSL used to configure and execute the pulverization platform. The discussion of the details of the DSL is deferred to the Section 5.7. Below, will be reported all the specific aspects of the platform created via the DSL, including the algorithm used to set up the deployment unit, create the components references and configure the communicators.

The platform configuration via the DSL produces a platform object which is used to start and stop the platform. The creation process of the platform is made via the `start` method and is made of several activities executed in a specific order: first of all, the dependency injection framework is initialized registering the *context*, the *communicator manager* and the *remote place provider*. Then, from the given configuration is determined which components belong to the same deployment unit and which components are remote. To do this, all the components defined in the logical device are retrieved, then from the configuration is checked if the components registered by the user matches the configuration otherwise an exception is raised. After that, for each component, the corresponding *component reference* is created and the *communicator* is configured. The creation of a *component reference* takes as argument the *serializer*, a set containing all the *components*

belonging to the logical device, a set containing the *deployment unit* components, and the *communicator*. The two sets (the one containing all the components and the one containing the deployment unit components) are used to determine if the component is remote or not, and consequently leveraging the local communicator or the remote one (provided as argument). Once all the components are created, for each one, the corresponding logic is executed saving into a set the reference of the spawned job. In Figure 5.9 is reported the activity diagram showing the platform creation process.

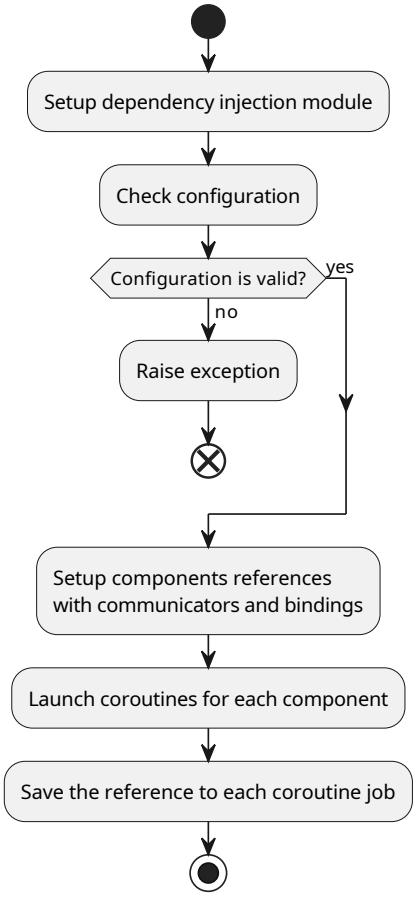


Figure 5.9: Activity diagram showing the platform creation process.

The creation of component references for the behavior component, requires further analysis: in fact, the behaviour holds a reference for each component, namely the *state, actuators, sensors,* and *communication*. In this regard, not all four components might be defined, so the behavior will have to create a dummy component reference for the missing components. This is managed automatically

when creating component references by looking at the configuration, and if a component is neither local nor remote, then the dummy implementation is used.

5.5 RabbitMQ module

The RabbitMQ module, at the time of writing, is the only module that provides an implementation of the `Communicator` interface. In particular, this module enables the communication between components via **RabbitMQ**, a message broker that implements the AMQP protocol.

RabbitMQ overview

RabbiMQ is an open source **message broker** written in Erlang that implements the **AMQP** protocol [20]. RabbitMQ is lightweight and easy to deploy on-premises and in the cloud. It supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements [13].

RabbitMQ is based on the notion of **broker**, which receives messages from publishers (applications that publish them, also known as producers) and route them to consumers (applications that process them). The publishers, consumers and the broker can all reside on different machines, reasons why RabbitMQ could be a good choice for a distributed system, in particular in the context of the pulverization.

The AMQP protocol has the following view of the world: messages are published to exchanges, which are often compared to post offices or mailboxes. *Exchanges* then distribute message copies to queues using rules called bindings. Then the broker either delivers messages to consumers subscribed to queues or consumers fetch/pull messages from queues on demand.

Networks are unreliable and applications may fail to process messages therefore the AMQP model has a notion of message acknowledgments: when a message is delivered to a consumer the consumer notifies the broker, either automatically or as soon as the application developer chooses to do so. When message acknowledgements are in use, a broker will only completely remove a message from a queue when it receives a notification for that message (or group of messages).

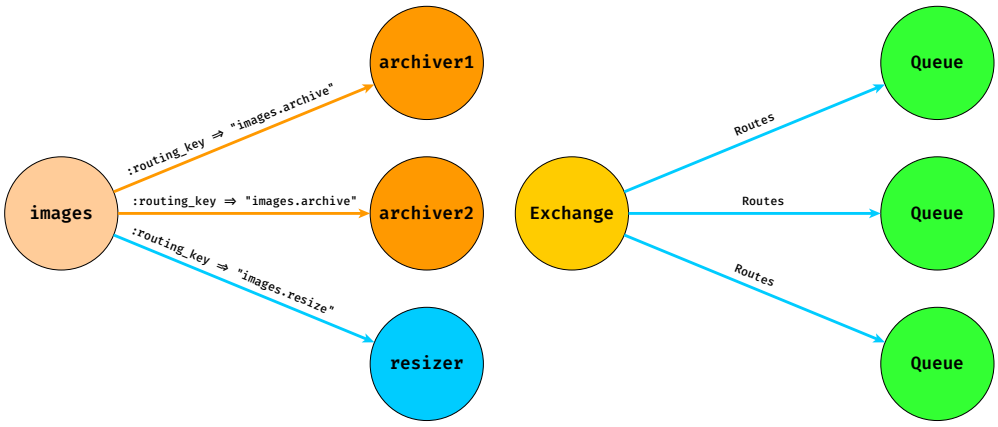
Exchanges are entities that receive messages from producers and route them to zero or more queues. The routing algorithm depends on the exchange type and rules called bindings. The AMQP protocol defines four exchange types that are reported in Table 5.2.

[13] `https://www.rabbitmq.com/#features`

Exchange type	Default names
Direct exchange	(Empty string) and `amq.direct`
Fanout exchange	`amq.fanout`
Topic exchange	`amq.topic`
Headers exchange	`amq.match` (and `amq.headers` in RabbitMQ)

Table 5.2: Exchange types.

The *default exchange* is a direct exchange with no name (empty string) pre-declared by the broker. It has one special property that makes it very useful for simple applications: every queue that is created is automatically bound to it with a routing key which is the same as the queue name.



(a) Direct exchange example. (b) Fanout exchange example.

Figure 5.10: Representation of the two main exchanges defined in the AMQP protocol.

A direct exchange delivers messages to queues based on the message routing key. A direct exchange is ideal for the unicast routing of messages (although they can be used for multicast routing as well). Below is reported how a direct exchange works:

- A queue binds to the exchange with a routing key K
- When a new message with routing key R arrives at the direct exchange, the exchange routes it to the queue if $K = R$

A direct exchange can be represented graphically as in Figure 5.10a.

A fanout exchange routes messages to all of the queues that are bound to it and the routing key is ignored. If N queues are bound to a fanout exchange, when a new message is published to that exchange a copy of the message is delivered to all N queues. Fanout exchanges are ideal for the broadcast routing of messages. A fanout exchange can be represented graphically as in Figure 5.10b.

Topic exchanges route messages to one or many queues based on matching between a message routing key and the pattern that was used to bind a queue to an exchange. The topic exchange type is often used to implement various publish/-subscribe pattern variations. Topic exchanges are commonly used for the multicast routing of messages. Whenever a problem involves multiple consumers/applications that selectively choose which type of messages they want to receive, the use of topic exchanges should be considered.

Rabbimq communicator

When implementing a new `Communicator`, two main steps are required: the first one is the implementation of the `Communicator` interface and the second one is the implementation of the `RemotePlaceProvider`. The `Communicator` interface is implemented by the `RabbitMQCommunicator` class, while the default implementation of the `RemotePlaceProvider` interface is provided by the `defaultRabbitMQ-RemotePlace` method.

The remote place provider provides the implementation of the concepts "who" and "where" (defined by the `RemotePlace` class) in the following way: the `who` is the *device id*, while the `where` is the *name of the component*. This representation will be used by the `RabbitmqCommunicator` to create the queues that will be used to communicate between the components.

The `RabbitmqCommunicator` should rely on a library that implements the AMQP protocol to communicate with the RabbitMQ broker, but since there isn't a Kotlin multiplatform library that implements the AMQP protocol, the class is marked as `expect` so that the specific target platform can provide the implementation. At the time of writing, only the JVM target platform is implemented using the `reactor-rabbitmq` library. The choice of this library is because it is based on project *Reactor* [14] and can be seamlessly used in combination with the Kotlin coroutines using the `kotlinx-coroutine-reactor` [15] library.

The class constructor accepts all the RabbitMQ-specific parameters like: *host*, *port*, *username*, *password*, and *virtual host*. Noticeable is the `setup` method that is used to set up the connection with the RabbitMQ broker, declare the exchange and the queues, and finally bind the queues to the exchange. In this method, the

[14] `https://projectreactor.io/`
[15] `https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-reactor/`

`binding` and the `remote place` are used to create the queues that will be used to communicate between the components.

Below will be described how the queues are created and how they are used to communicate between the components. The `RabbitmqCommunicator` class defines two queues: a *sendQueue* and a *receiveQueue*, the first one is used to send messages to the remote component, while the second one is used to receive messages from the remote component. In this communicator, the send queue name format follows the pattern `<local component name>/<remote component name>/<remote component id`, while the receive queue name format follows the pattern `<remote component name>/<local component name>/<remote component id>`. The *local component name* is retrieved from the `binding` parameter, while the *remote component name* and the *remote component id* are retrieved from the `remote place` parameter. For example, if the communicator should manage the communication between the *Behaviour* and the *State* (remote) component for a device with ID "1", the send queue name will be `Behaviour/State/1`, while the receive queue name will be `State/Behaviour/1`. Vice versa, the counterpart communicator that manages the communication between the *State* and the *Behaviour* (remote) component defines the send queue name as `State/Behaviour/1` and the receive queue name as `Behaviour/State/1`. As can be seen, the send queue name for the first communicator is the same as the receive queue name for the counterpart communicator, and vice versa. This means that the message sent by the first communicator will be received by the counterpart communicator. The Figure 5.11 depicts the interaction pattern described above using two communicators.

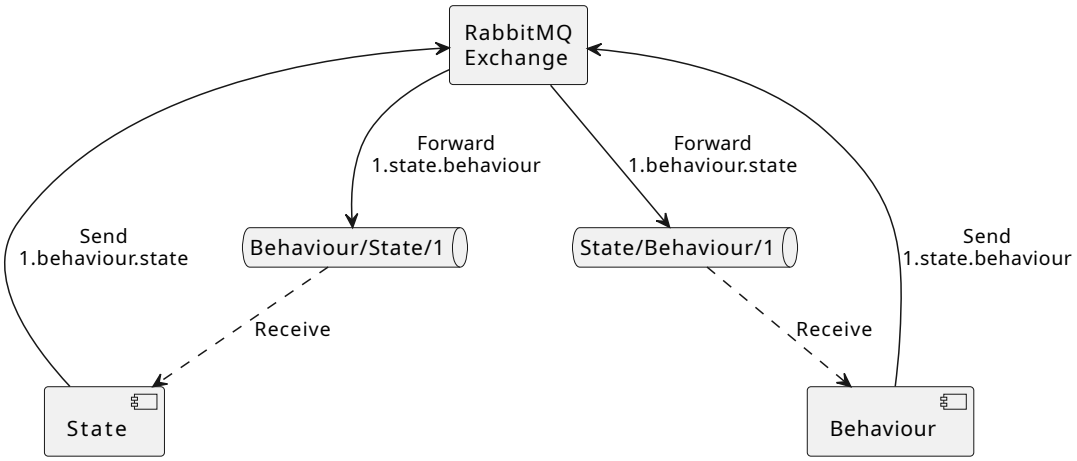


Figure 5.11: Example of queue declaration in the framework for the communication between the *Behaviour* and the *State* components.

The `fireMessage` and `receiveMessage` methods implementation are straight-

forward: the `fireMessage` method sends the message to the remote component via the send queue using the `Sender` object, while the `receiveMessage` method receives the message from the remote component via the receive queue using the `Receiver` object.

5.6 Configuration DSL

The configuration DSL is used to define the structure of each logical device in terms of defining components and where they are deployed. Before proceeding to discuss the details of the DSL, it is important to define which information the configuration should have.

The recurring terms and concepts in the pulverization are *logical device*, *deployment unit* and *place* where the components are deployed. The first concept is modeled by the `LogicalDeviceConfiguration` class which defines the name of the device, the set of components type belonging to the logical device, and a set of deployment units. The *deployment unit* is modeled by the `DeploymentUnit` class which defines the set of components type that should be deployed and the `Tier` which represents the place where the components should be deployed. The *tier* is modeled via a sealed interface which has three possible values: `Cloud`, `Edge` and `Device`.

The configuration captures also the concept of "relationship" between logical devices. In particular, the configuration DSL defines the `DeviceRelationsConfiguration` and the `DeviceLink` classes which are used to hold information about the link between the logical devices.

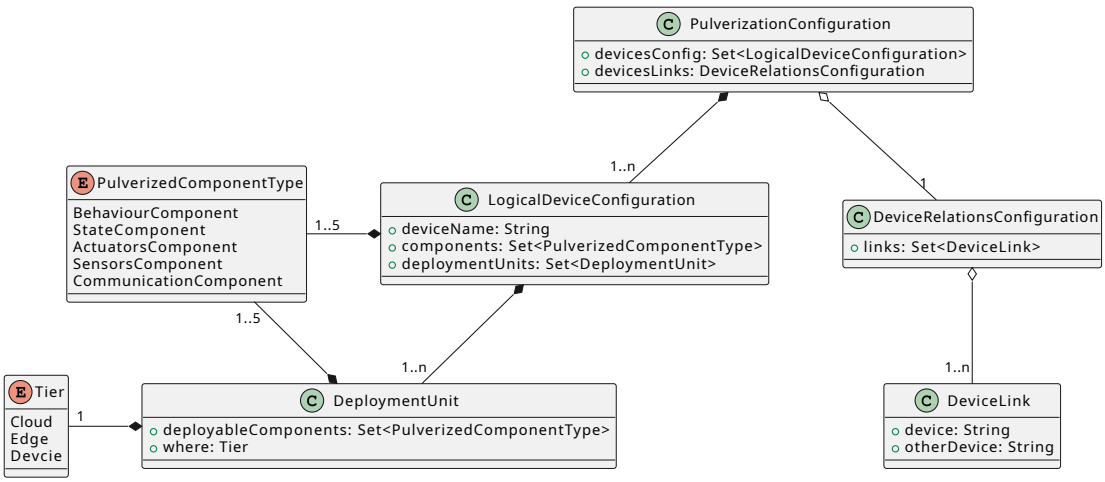


Figure 5.12: Configuration DSL class diagram.

Finally, the `PulverizationConfiguration` class has a set of `LogicalDeviceConfiguration` and a `DeviceRelationsConfiguration`; this class represents the root of the configuration and is used to configure the pulverization platform. The Figure 5.12 depicts the relationship between the classes that define the configuration that can be created via the configuration DSL.

The configuration is enriched with some extension methods that allow getting the logical device configuration from a given device name and from a logical device configuration, can be retrieved the deployment unit from the given components.

An example of configuration is shown on the Listing 5.13. The configuration defines three logical devices where the *device-1* can communicate with the *device-2* and the *device-3*. At the time of writing, the information about the links between the logical devices is not used by the framework, but in a future version, this information can be used, for example, by the communication components to establish the right communication within the network.

```
val config = pulverizationConfig {
    logicalDevice("device-1") {
        BehaviourComponent and StateComponent
            deployableOn Cloud
        StateComponent deployableOn Cloud
        CommunicationComponent deployableOn Edge
        SensorsComponent deployableOn Device
    }
    logicalDevice("device-2") {
        BehaviourComponent and StateComponent
            and CommunicationComponent
            deployableOn Device
    }
    logicalDevice("device-3") {
        BehaviourComponent deployableOn Cloud
        CommunicationComponent deployableOn Edge
        SensorsComponent deployableOn Device
    }
    deviceLinks {
        "device-1" linkedWith "device-2"
        "device-1" linkedWith "device-3"
    }
}
```

Listing 5.13: Configuration DSL

5.7 Platform DSL

The platform DSL represents one fundamental building block of the entire framework: with this DSL the user can configure the platform in a simple declarative way, without the need to manage specific aspects like how the components should be instantiated or how the communication between them should occur. The DSL is implemented in the *dsl* package of the *platform* module and should provide the following features:

- The ability to register the user-defined components with their corresponding logic
- The ability to specify which communicator should be used to communicate between the components
- The ability to specify a custom context that should be used.
- Produce a platform instance that can be used to start the pulverized system.

All of those four features are implemented in the DSL using the syntax and the structure defined in the Listing 5.14 which provides an example of the use of the DSL to configure the platform.

First of all, the DSL takes as arguments the configuration of the device that should be executed; the configuration is used to know which components the logical device has and how they are distributed across the infrastructure. With that information, the platform can determine which components are local (in the same deployment unit) and which are remote, instantiating the right components and communicators.

Then, the DSL provides a way to register the user-defined components with their corresponding logic. If the given components do not match the configuration, an exception is raised.

Finally, the type of communicator and the remote place provider can be specified respectively via the `withPlatform` and `withRemotePlace` methods. The context can be overridden via the `withContext` method.

The development of this DSL has involved the resolution of several issues, in particular, the management of serialization aspects and a limitation of the Kotlin type inference. Below are described those issues and the solutions adopted to overcome them.

Serialization

Serialization represents an important aspect of the working of the framework but at the same time, you don't want to force the user to manage serialization aspects. Is

```
val platform = pulverizationPlatform(
    config.getDeviceConfiguration("device-1")
) {
    behaviourLogic(
        BehaviourComponent(), ::behaviourLogic
    )
    stateLogic(StateComponent(), ::stateLogic)
    communicationLogic(
        CommunicationComponent(), ::communicationLogic
    )
    actuatorsLogic(
        ActuatorsComponent(), ::actuatorsLogic
    )
    sensorsLogic(SensorsComponent(), ::sensorsLogic)

    withPlatform { RabbitmqCommunicator() }
    withRemotePlace { defaultRabbitMQRemotePlace() }

    withContxt {
        configFilePath("config/.pulverization.env")
    }
}
```

Listing 5.14: Example of the use of the DSL to configure the platform.

the responsibility of the framework to retrieve the serializer from the user-defined components and to use it in conjunction with other framework elements like the communicators. Thus, the low complexity of using and configuring the framework is guaranteed.

The first issue is how to retrieve the serializer from the user-defined components without forcing the user to provide it. The solution adopted defines the DSL entry point as an inline function that accepts a reified type parameter, in this way, the serializer can be retrieved via the `serializer` method. In particular, the function defines five type parameters: `S, C, SS, AS,` and `R` which respectively represents the type of the state, the type of communication, the type of the sensors, the type of the actuators, and the type of behaviour result.

The problem occurs when a logical device does not define all of the five components but only a subset of them. For example, if a logical device is made of the *behaviour, communication, sensors,* and *actuators* components when using the DSL, a type for the *state* should be provided. A first, elegant approach would

be to default all the unspecified types to the `Nothing` bottom type; however, this approach is not feasible, since the `Nothing` type is not serializable and thus, it cannot be inlined as a type. The solution adopted is to use `Any` as the default type for the unspecified components. This solution enables the inlining of the type, so the function can be used (as inline) but the drawback is that there is no serializer for the `Any` type. Since the absence of the type means that the component is not used, the serializer for the `Any` type is a dummy serializer that does nothing but makes the function sound. In this way, a check can be made to verify if the given type is `Any` (and rely on the dummy serializer) or if it is a specified type retrieving the serializer from it.

The Listing 5.15 shows the implementation of the dummy serializer for the `Any` type and how the serializer is retrieved from the user-defined types.

```
internal class AnySerializer<S> : KSerializer<S> {
    override val descriptor: SerialDescriptor =
        PrimitiveSerialDescriptor(
            "kotlin.Any", PrimitiveKind.STRING
        )
    override fun deserialize(decoder: Decoder): S =
        TODO("Not yet implemented")
    override fun serialize(encoder: Encoder, value: S) =
        TODO("Not yet implemented")
}

internal inline fun <reified T> getSerializer():
    KSerializer<T> = when (T::class) {
        Any::class -> AnySerializer()
        else -> serializer()
    }
```

Listing 5.15: Dummy serializer for the Any type.

Type inference problem

The second issue is related to the type inference of Kotlin that does not infer a default type for an unspecified generic type. As said in the previous sections, the simplicity of the use of the DSL is one of the main goals of the framework. In particular, when the user configures the platform using the DSL, the user does not need to specify the type of components, since the framework can infer them from the configuration. However, the Kotlin type inference does not work for the generic

types that are not specified raising a compilation error, forcing to specify all the types required by the entry point function of the DSL. The Listing 5.16 shows an example of the use of the DSL when all the types are specified. The proposed example although valid and works, is not very elegant because forces the user to remember the positional order of the types and specify all of them.

To minimize the amount of boilerplate code and maintain the simplicity of the DSL, the proposed solution utilizes extension methods to facilitate type inference for all five types included in the user-defined configuration base. With the use of the extension methods, the Kotlin type inference algorithm can infer the right types without specifying them explicitly. This is a sort of workaround to deal with the limitation of the Kotlin type inference algorithm that does not infer a default type for an unspecified generic type.

```
val platform = pulverizationPlatform<StateOps, Comm,
    SensorsPayload, ActuatorsPayload, Unit>(
    config.getDeviceConfiguration("device-1")
) {
    behaviourLogic(BehaviourComponent(), ::bhvLogi)
    stateLogic(StateComponent(), ::stateLogic)
    communicationLogic(CommComponent(), ::commLogic)
    actuatorsLogic(ActuatorsComponent(), ::actLogic)
    sensorsLogic(SensorsComponent(), ::sensLogic)
}
```

Listing 5.16: Example of the verbosity of the DSL when all the types are specified.

The Listing 5.17 shows the implementation of the extension methods that help the type inference to infer the right types without specifying them explicitly.

```
companion object {
    fun <S, C, SS, AS, R>
    PulverizationPlatformScope<S, C, SS, AS, R>
    .behaviourLogic(
        behaviour: Behaviour<S, C, SS, AS, R>,
        logic: BehaviourLogicType<S, C, SS, AS, R>,
    ) where S, C : Any, SS : Any, AS : Any, R : Any {
        configuredComponents += BehaviourComponent
        behaviourComponent = behaviour
        behaviourLogic = logic
    } // Other methods omitted for brevity
}
```

Listing 5.17: Extension methods to help the Kotlin type inference algorithm.

Chapter 6

Validation

This chapter describes the validation process of the framework. The validation process is divided into two parts: the validation of the framework itself via unit and integration testing, and the validation of the framework's use cases via the demos. Aspects of CI/CD used for the development and maintenance of the project will be explained, as well as the methodologies used to deploy the framework. Finally, the Section 6.5 describes the current limitations of the framework and future work geared toward extending and improving the framework.

6.1 Testing

Software testing is a critical process for verifying whether the actual software product conforms to the specified requirements and meets the required quality standards, thereby ensuring its integrity and "defect-free" performance. This process involves the systematic execution of software or system components, using either manual or automated tools, to evaluate one or more properties of interest. The overall objective of software testing is to identify any discrepancies or deficiencies, such as errors, gaps, or missing requirements, that may exist between the actual system and its intended design.

6.1.1 Unit testing

Unit testing is a type of software testing where the focus is on individual units or components of a software system. Its purpose is to validate that each unit of the software works as intended meeting the requirements. Unit testing is usually performed by the developer and is the first level of testing performed on the software.

Generally, unit tests are automated and executed whenever a change is made to

the source code to ensure that the new code does not break the existing functionality. Unit tests are designed to validate the smallest possible unit of code, such as a single function or method, testing them in isolation from the rest of the system.

Usually, a lot of unit tests are written to try to cover as much code area as possible by going to test corner cases and wrong uses of the code. One metric that indicates the amount of testing that is present in the code base is called code coverage. This metric, often expressed as a percentage, defines how many lines of code were covered by unit tests indicating the pervasiveness of the tests but does not provide any guarantee that the tests are correct or that they cover all the possible cases. Therefore, an alternate interpretation of the notion of code coverage is the number of lines of code that are untested.

The importance of testing has been recognized as a fundamental tool in the development and maintenance of a code base, and therefore several test suites have been developed. The most relevant in the JVM ecosystem are JUnit [1] and TestNG [2], which are both unit-testing frameworks for the Java programming language.

In recent times, the concept of "test as specification" has emerged, and for that reason, they must follow good programming practices and be as clear and expressive as possible to be easily read and interpreted, al well as the main codebase. In this regard, testing frameworks have emerged that provide DSLs enabling the writing of clear, well-organized and contextualized tests. The most relevant in the JVM ecosystem are Spock [3] and Kotest [4].

The following will outline the unit testing aspects involved in the framework, explaining the rationale for choosing Kotest as the testing framework and which key aspects of the framework were subject to unit testing.

The requirements that a testing framework must have for this project are as follows:

- It must provide a DSL for writing tests
- It must provide several testing and assertion styles
- It must support out-of-the-box support for Kotlin coroutines

[1] **JUnit** goal is to create an up-to-date foundation for developer-side testing on the JVM. `https://junit.org/junit5/`

[2] **TestNG** is inspired by JUnit and NUnit but introduces some new functionalities that make it more powerful and easier to use. `https://testng.org/doc/`

[3] **Spock** is a testing and specification framework for Java and Groovy applications. `http://spockframework.org/spock/docs/1.3/all_in_one.html`

[4] **Kotest** is a testing framework for Kotlin that provides a rich set of tools for testing. `https://kotest.io/docs/`

The first two requirements are needed to write clear and expressive tests, while the third is needed to test the framework in a seamless way since is entirely based on coroutines. The two main candidates for this project are Spock and Kotest. Although Spock provides a DSL for writing tests and different styles, it does not provide any support for coroutine, since it was born for Java and groovy. In contrast, Kotest is entirely written in Kotlin and provides a full DSL with various styles for testing, as well as native support for coroutines. For these reasons, Kotest was chosen as the testing framework for this project. Moreover, Kotest supports Kotlin multiplatform, a feature that perfectly fits the framework's goal of being cross-platform; in this way, all the tests are executed on the JVM, JS and Native platforms providing an effective way of testing the framework for all of them.

For most of the tests in the pulverization framework, the *FreeSpec* style was used, which is a style that allows writing tests in a specification-like way, where the test cases are written as a sentence, and the test body is written as a code block. This style is particularly suitable for testing the framework since it allows writing tests clearly and expressively, the Listing 6.1 shows an example of a test written in this style.

As can be seen, the test is written as a sentence in a specification-like fashion; in this way, even people who are not familiar with the framework can understand the test and its purpose, as well as understand the framework API and its usage.

```
class BasePulverizationConfigTest : FreeSpec({
    "The configuration DSL" - {
    "should configure a logical device" {
        val config = pulverizationConfig {
            logicalDevice("device-1") {
                StateComponent deployableOn Cloud
                BehaviourComponent deployableOn Edge
            }
            logicalDevice("device-2") { }
        }
        config.devicesConfig.size shouldBe 2
        config.getDeviceConfiguration("device-1")?.let {
            it.deviceName shouldBe "device-1"
            it.deploymentUnits.size shouldBe 2
        }
    }}
})
```

Listing 6.1: Example of a test written in the *FreeSpec* style.

As follow will be described the relevant testing aspect for each module of the framework.

Core module

The core module is mainly composed of interfaces that the user must implement to use the framework. For this reason, the tests for this module are limited and mainly focus on testing the *configuration DSL*, the *sensors container* and the *actuators container*.

The testing of the sensors and actuators container presents some interesting details that are worth mentioning. The first aspect to consider is how the *dependency injection* is performed during the tests. Is recalled that the `SensorsContainer` and `ActuatorsContainer` implement the `PulverizedComponent` interface and therefore they must give an instance of the `Context` interface. Although in this test scenario, the context will not be used, it is necessary to provide an instance of the context to the container. The fast and easy way to do this is to use a mocked version of the context, which is provided to the container via dependency injection. The Koin framework makes available a `KoinTest` class and overriding the `getKoin` method, it is possible to provide a mocked version of the context to the container. The Listing 6.2 shows an example of how to provide a mocked version of the context to the container.

```
class ActuatorsContainerTest : FreeSpec(), KoinTest {
    override fun getKoin(): Koin = koinApplication {
        module {
            single {
                object : Context {
                    override val deviceID = "test"
                }
            }
        }
    }.koin

    // Tests
}
```

Listing 6.2: Example of how to provide a mocked version of the context to the container during the tests.

The effective test of the container is performed with the use of fixtures: the container to be tested requires some components like sensors, actuators and so

on to be added to it. For this reason, a fixture containing dummy components is created. In this way, the test is focused on testing the container itself and not managing also the creation of the components, simplifying the overall test class.

The test conducted on the container are trivial, but they are useful to ensure that the container is working properly. The container is a delicate component because it is queried via the type of sensor or actuator that needs to be retrieved, which is why it is necessary to exhaustively test all possible scenarios that may occur.

The testing of the DSL is also trivial: the DSL is a simple class that provides a set of functions to configure the framework. The tests are focused on ensuring that the configuration produced is consistent with the use of the DSL. In addition to testing in normal DSL use, special attention was paid to recreating possible uses of DSL that would lead to inconsistencies in configuration and verify whether all such cases were handled correctly. For example, was tested the case in which the user define the same component in two different deployment units, which is not allowed and should produce an error. Finally, all the utility functions provided to work with the configuration were tested.

Platform module

The testing of the platform module is divided into four main parts: the testing of the *communicator*, the testing of the *components reference*, and finally, the testing of the *dsl*.

For what concern the *communicator* testing, only the local communicator was tested, since the testing of the remote ones is delegated to the specific module that implements them. The local communicator relies on the `CommManager` class to use the right flow for communication with the other local communicator. The test consists in registering the `CommManager` to the dependency injection module and then testing that the class returns the same instance of the flow for the same communication type.

The testing of the local communicator, instead, is more complex. First of all, is verified that the local communicator can not be initialized with a self-reference; this means that initialization of the local communicator with the same component as source and destination is not allowed. Then, is tested the communication between two local communicators: the test consists in creating two local communicators, spawning each one in a different coroutine and then sending a message from one to the other. The test is successful if the message is received by the other local communicator. If a problem occurs during the receiving of the message, the test can hang indefinitely, so a timeout is set to avoid this problem. So the test fails if the timeout is reached or if the payload differs from sender to receiver, in all the

other case the test succeed. Finally, is tested the condition in which the sender sends more messages than the receiver can receive, in this case, the receiver should receive only the last message sent by the sender. To emulate this condition, the sender and the receiver are spawned in different coroutines and the sender starts sending messages to the receiver. The receiver, instead, when spawned is blocked for a certain amount of time, to simulate a high workload. After the delay, the receiver starts collecting the messages sent by the sender but only the last one should be collected.

The testing of *components reference* is straightforward: the test consists in creating a `ComponentsRefImpl` and then set up it with the pair of the components to be referenced. After that, is tested that the reference is correctly set up and the class relies on the `LocalCommunicator` to send and receive the messages.

The *dsl* testing is focused to figure out possible illegal configurations of the platform. In this regard, several test cases were created to test, for example, the case in which more or fewer components are registered than the one specified in the configuration. During the development of a demo of the framework, it was discovered a bug in the DSL relative to the type inference; the bug was fixed and a test was added to ensure that the bug will not be reintroduced in the future.

Code coverage

The metric of code coverage holds significant importance in the development of a software project. Its management is facilitated by two distinct tools that operate at separate levels of the project. The first tool, Kover, is a plugin for the Gradle build system that is designed to provide code coverage for Kotlin projects. The second tool, Codecov, is a cloud-based service that provides code coverage for GitHub projects.

Kover operates by instrumenting the test suite to collect the code coverage data. Subsequently, it publishes code coverage reports in different formats, mainly `.html` and `.xml`. On the other hand, Codecov collects code coverage data from various sources and provides a web interface to visualize the code coverage of the project. All coverage reports are uploaded to Codecov using a GitHub Action that is automatically triggered every time a new commit is pushed to the repository.

6.1.2 Integration testing

Integration testing is a type of testing that aims to test the interaction between different components of the system. A typical software project is composed of several modules, and each one of them is responsible for a specific task; the integration testing is focused on testing the interaction between them to ensure that they work together as expected when integrated.

There are four main approaches or strategies to perform integration testing: the *bottom-up*, the *top-down*, the *big bang* and the *sandwich* approach. Each strategy has its own advantages and disadvantages, and the choice of the strategy to use depends on the specific project and the type of testing that is required.

The *big bang* approach involves integrating all modules at once and testing them all as one unit. This approach has the following advantages:

- It is easy to implement and it is suitable for small projects
- It is easy to identify errors, saving time and speeding up the deployment

However, the *big bang* approach has the following disadvantages:

- It is difficult to locate the source of the error since different modules are integrated as one unit
- It is time-consuming for large projects with lots of modules
- It must wait until all modules are developed before starting the testing phase

The *top down* approach is an incremental approach that involves testing from the topmost module and then proceeding to the lower modules. Each module is tested one by one and then integrated with the other. This approach has the following advantages:

- It is easier to identify defects and isolate their sources
- Testers check important units first, so they are more likely to find critical design flaws.
- It is possible to create an early prototype of the system

The main disadvantage of the *top down* approach is that when too many testing stubs are involved, the testing process can become complicated.

The *bottom up* approach is the opposite of the *top down* approach: it involves testing the lower modules first and then integrating them with the upper modules. Once the lower-level modules are tested and integrated, then the next level of modules is formed. The main advantages of using this approach are: easier to find and localize faults and no time is wasted waiting for all modules to be developed, unlike the big bang approach. However, the main disadvantage of this approach is that critical modules which control the flow of the application are tested last and may be prone to defects.

Finally, the *sandwich* approach is a combination of the *top down* and *bottom up* approaches. In this approach, top down and bottom up testing approaches

are combined. The top-level modules are tested with low-level modules and the low-level modules are tested with high-level modules simultaneously. Each module interface is tested, so there are fewer chances of a defect. The main advantages derived from this approach are represented by the combination of the benefits of both top down and bottom up strategies. Moreover, this approach reduces the amount of time spent on the process and all the modules are tested comprehensively. The main disadvantage of this approach is that it is more complex than the other approaches.

In the framework development, the *big bang* approach was used to perform the integration testing. The reason for this choice is that the framework is composed of a few modules, in this way a simpler integration testing process can be achieved.

The most important integration test is defined in the `AsyncScenario` test class. This test class is responsible for testing the framework as a whole. In this test, are used the two DSLs to configure the devices and the platform, and then the platform is started testing that each deployed component is correctly started and that the messages are correctly sent and received. The importance of the test comes from the fact that it involves the entire stack of the framework bringing all its components into play by verifying that they are operating correctly.

6.2 Continuous Integration and Delivery

The continuous integration and continuous delivery (CI/CD) pipeline is a systematic approach to software development that involves building, testing, and deploying code. Automating the pipeline minimizes the risk of human error and ensures a consistent release process. It involves various tools, such as code compilation, unit testing, code analysis, security, and binary creation. CI/CD forms the foundation of a DevOps methodology and unifies developers and IT operations teams in software deployment. The Figure 6.1 shows a typical CI/CD pipeline.

To enable a pipeline, adequate tools and infrastructure are required. Over the last ten years, the CI/CD ecosystem has grown significantly, and there are many tools available to support the pipeline. As follow, a brief landscape of the most popular CI/CD tools is presented, showing the main features, the main advantages and disadvantages.

Jenkins is a free, open-source automation server that helps streamline the software development process by automating tasks such as building, testing, and deploying code. It offers hundreds of plugins to support various technologies and tools, making it a highly versatile platform. Jenkins is often used for continuous integration and continuous delivery (CI/CD) workflows, allowing developers to quickly and easily validate and release code changes. The Listing 6.3 shows an

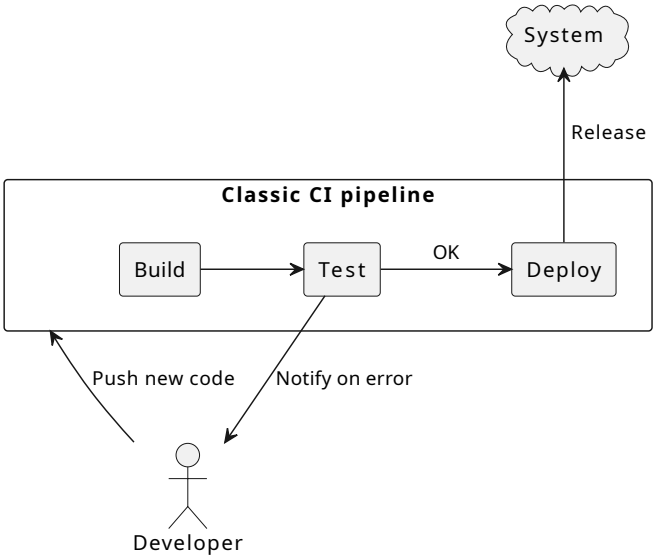


Figure 6.1: A classic example of a CI/CD pipeline.

example of a *Jenkinsfile* written in Groovy syntax that builds a java project and subsequently runs the tests.

In the example, the Jenkinsfile defines a pipeline with two stages: one for building the application using Maven, and one for testing it. This is just a simple example of what can be achieved with Jenkins, but it demonstrates how it can help automate and streamline the software development process.

GitHub Actions is a powerful CI/CD platform integrated into GitHub that allows developers to automate their software development workflows. With GitHub Actions custom workflows that run on specific events can be defined, such as code pushes, pull requests, and releases. Workflows are defined using *YAML* files and can include multiple steps, such as building and testing code, deploying to various environments, and more.

Here is a simple example of a GitHub Actions workflow that builds and tests a Java application:

In the Listing 6.4, the workflow is triggered on pushes to the main branch and runs on an Ubuntu-based runner. The workflow includes steps to check out the code, set up a Java environment, build the application, and run tests. With GitHub Actions, you can easily automate and streamline your software development workflows, helping to improve efficiency and reduce errors.

An *action* in the context of GitHub Actions is a pre-written, reusable piece of code that performs a specific task within a software development workflow. These actions can be run as part of a workflow to automate tasks such as building code, testing it, deploying it, and more.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh './gradlew build'
            }
        }
        stage('Test') {
            steps {
                sh './gradlew test'
            }
        }
    }
}
```

Listing 6.3: A Jenkinsfile example that builds and tests a Java project.

The choice to use GitHub Actions to implement the CI/CD pipeline was made primarily for reasons of practicality and prior knowledge, but also because it boasts a large community that supports and maintains the actions. The pipeline is structured into three main jobs:

- **Build**: the build job is responsible for running all the quality assurance tools, compiling the source code and generating the artifacts
- **Test**: the test job runs all the unit tests and integration tests, and it is responsible to upload the artifacts on the Maven Central repository and close the repository
- **Release**: the release job is responsible for releasing the artifacts on the Maven Central repository

All the aforementioned jobs are executed in parallel over three different operating systems: Windows, Linux and macOS. The use of three different operating systems allows for verification that the framework is compatible with all the major operating systems and the relative platforms, reducing the probability of errors on a specific platform.

When working with Kotlin multiplatform, the pipeline slightly changes, since the artifacts must be generated for each supported platform. To achieve this, the pipeline is enriched with some additional jobs: the very first job is responsible for

```yaml
name: Java CI
on:
  push:
    branches:
      - main
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Setup Java
      uses: actions/setup-java@v2
      with:
        java-version: 19
    - name: Build
      run: ./gradlew build
    - name: Test
      run: ./gradlew test
```

Listing 6.4: A GitHub Actions workflow example.

creating the staging repository on the Maven Central repository, setting up the environment for a possible release. Then, are executed the build, test and release jobs for each platform. A specific job is also added after the build job to close the staging repository on the Maven Central repository, and consequently, trigger the checks over the repository. Finally, the release job is executed to release the artifacts on the Maven Central repository if needed (see Figure 6.2).

The rationale behind the choice of using a dedicated job for creating the staging repository is due the fact that the subsequent jobs are executed in parallel over different OS, which means that they would create a different staging repository for each OS, leading to an unintended condition. Using the aforementioned approach, the staging repository is created only once, and then, the subsequent jobs are executed over the same repository by sharing the *repository id*. In this way, the artifacts coming from different OS are uploaded to the same repository (using the repository id generated before), and consequently, the release job releases all the artifacts on the Maven Central repository consistently.

Code maintenance is a key aspect of good project lifecycle management. For this reason, tools such as *renovate, mergify* and *semantic-release* have been used to automate these aspects.

Renovate is an automated tool that is used to manage and automate the process

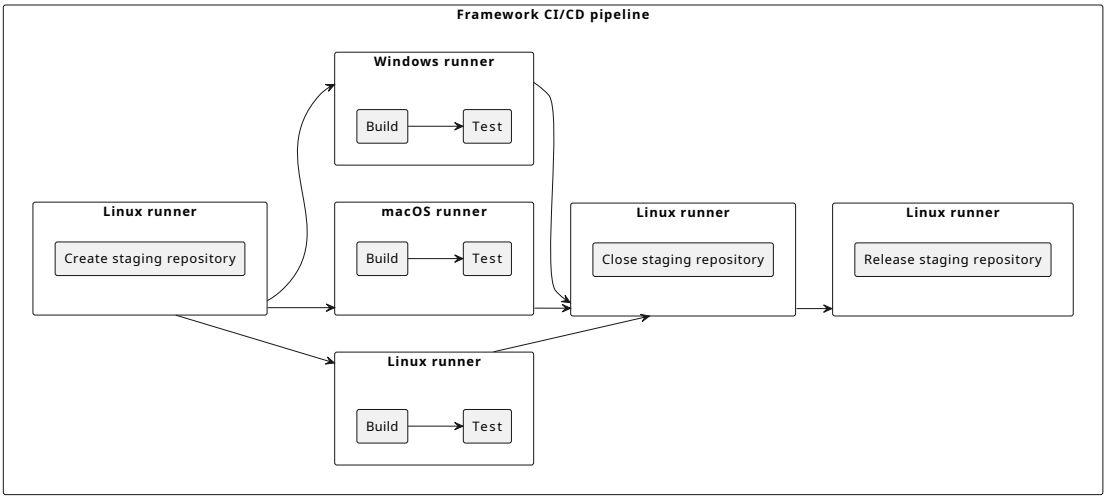


Figure 6.2: The CI/CD pipeline of the framework.

of updating dependencies and keeping projects up-to-date. The Renovate Bot can be integrated into a project's code repository and configured to automatically detect when updates are available and then create pull requests to implement the updates. This can help to streamline the process of keeping dependencies up-to-date and reduce the risk of security vulnerabilities or compatibility issues.

Mergify is an automated tool that helps to manage and automate the process of merging pull requests in a code repository. The Mergify bot can be integrated into a project's code repository and configured to automatically merge pull requests that meet specific criteria, such as passing all required tests or receiving approval from a specified number of reviewers. This can help to speed up the code review process and reduce the workload of maintainers, freeing them up to focus on more strategic tasks.

The semantic-release is a tool that automates the process of versioning and releasing software projects. It uses a specific commit message syntax to determine the type of changes made in each commit message [5] and automatically generates a new version number and make a release based on those changes. This helps to ensure that version numbers are consistently and accurately updated, reducing the risk of errors and allowing teams to focus on more important tasks. Additionally, by automating the release process, semantic-release can help to speed up the development cycle and allow teams to deliver new features and bug fixes to users more quickly and efficiently.

The joint use of these three tools on the one hand greatly speeds up the release and maintenance process of the code but on the other hand, should be paid attention

[5] `https://www.conventionalcommits.org/en/v1.0.0/`

to the fact that since are automatic tools, they could cause unintended updates causing repercussions on the proper functioning of the project, such as updating a dependency that modifies a public API thus making it incompatible with its use in the code. To minimize these scenarios, an extensive and complete test suite should be set up, but also a CI/CD pipeline that intercepts similar a problem should be used; in this way, maintainers are notified and they proceed with manual intervention to resolve the issue.

Therefore, all these details have been carefully attended to so that no situations arise that lead to inconsistencies or incompatibilities in the code base: renovate is configured to open a pull request as soon as a new version of any dependency is available; mergify is configured to merge automatically the PRs coming from Renovate which the status check is passed, and semantic-release is configured to make a release only from the *master* branch and if the previous jobs are completed successfully.

6.3 Demo 1: Single Device Multiple Components

With the first demo, a simple system was aimed at highlighting the main aspects of pulverization instantiated in a real physical scenario. As a second goal, the demo aims to provide a reference on how it is possible to "pulverize" a device through the framework by also testing it in a context closer to real use cases.

This demo models the following scenario: you want to monitor the moisture status of soil through a device by sensing the moisture level of the soil and through a valve, set the water flow to properly adjust the desired moisture level. This simple scenario involves several aspects of the pulverization: first, we find the concepts of sensor and actuator, which respectively serve to acquire the soil moisture level and regulate water flow. Finally, the behavior specifies how the flow regulation should occur based on the detected moisture. The Figure 6.3 shows the components defining the logical device.

The physical system is composed of three devices: two embedded systems used respectively for moisture sensing and water flow regulation and a server which represents the infrastructure where the pulverized system runs. Two **ESP32** [6] boards are used to implement the sensor component and the actuator component. The primary rationale behind selecting these boards is their cost-effectiveness and ease of use. Additionally, they come equipped with a Wi-Fi module, facilitating communication with the pulverization platform.

In Figure 6.4 is reported the physical system architecture where are reported the communication between the components and the server.

[6] `https://www.espressif.com/en/products/socs/esp32`

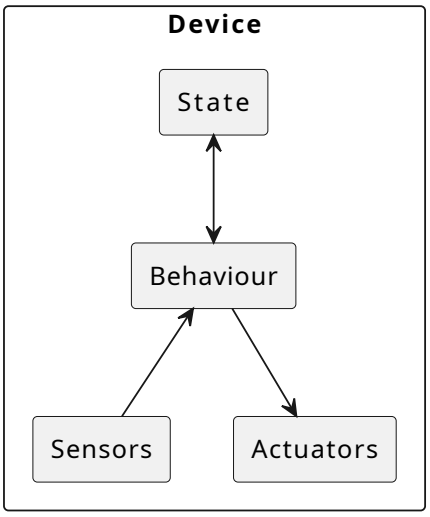


Figure 6.3: Decomposition of the moisture device into the pulverization components showing the interaction between them.

The multiplatform nature of the framework allows the execution of each "pulverized component" on different architectures. But at the time of writing, even if the number of supported architectures in Kotlin multiplatform is quite high, the framework does not support the **Xtensa** [7] architecture used by the ESP32 boards.

This limitation is solved as follows: we can think of the *sensor component* as acting as a kind of proxy by collecting data from the embedded device. In this way, the firmware that will control the board will be written in a language that supports the Xtensa architecture and will provide a communication mechanism with the *sensor component* that will simply forward the received message to other components. As can be seen, the framework is extremely flexible, allowing it to accommodate limitations such as the one described above.

In this specific case, the limitation described above is solved by writing the **ESP32** firmware in *Rust* (a language that supports the Xtensa architecture) and implement the communication with the *sensor component* using a TCP socket. The *sensor component* is implemented using the framework interface where a TCP socket is opened to listen for incoming messages from the embedded device. On each received message, the sensor's value is extracted and saved in a local state; in this way, on each `sense` method call, the last received value from the embedded device is returned. This simple workaround allows the use of the pulverization

[7] **Xtensa** is a configurable and extensible processor architecture developed by Tensilica, now owned by Cadence Design Systems. It is designed to meet the unique requirements of a wide range of applications and systems, from low-power IoT devices to high-performance computing systems.

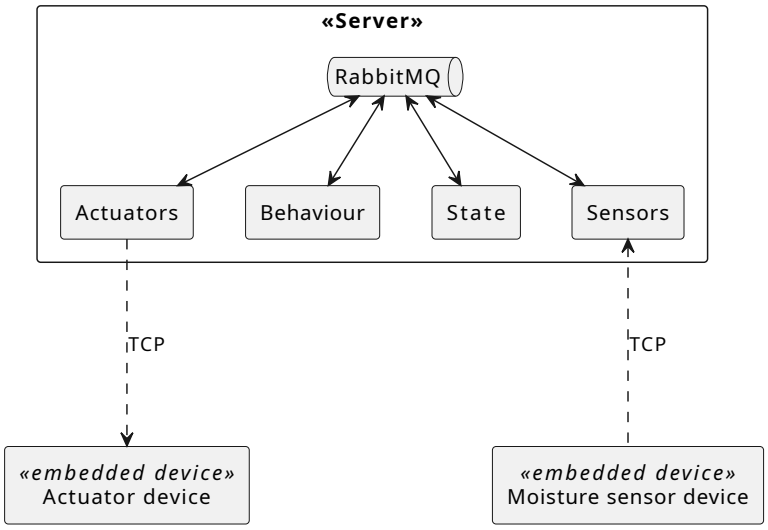


Figure 6.4: Physical system architecture where are reported the communication between the components and the server.

framework also in devices that do not support the Kotlin multiplatform architecture. The explanation of the limitation was made using the *sensor component* as an example, but the same principle was applied to the *actuator component*, so again via TCP socket, a message is sent to the embedded device for opening or closing the valve.

As follow, are reported the main relevant details of the implementation of this demo. As discussed above, the device has four components: *sensor*, *actuator*, *behavior* and *state*; the Listing 6.5 shows the configuration of the logical device where the *sensor* and *actuator* components are deployed into two devices, and the *behaviour* and *state* components are deployed into the server.

```
val configuration = pulverizationConfig {
    logicalDevice("moisture-device") {
        SensorsComponent deployableOn Device
        ActuatorsComponent deployableOn Device
        StateComponent
            and BehaviourComponent
            deployableOn Edge
    }
}
```

Listing 6.5: Configuration of the logical device named “moisture-device”.

The *behaviour* component defines the logic of the device: retrieve the moisture level from the *sensor* component and, if the moisture level is below a threshold, open the valve using the *actuator* component. The *state* component is used to store the moisture level. The Listing 6.6 shows the implementation of the *behaviour* component.

Finally, the three deployment units are defined and containerized using Docker. In particular, a *RabbitMQ* container is used to provide the communication between the components, while the other three containers are used to run the *sensor*, *actuator* and *behaviour* components. Since the three components are deployed into three different containers, these may be deployed on different machines without affecting the execution of the logical device.

The test of the demo was conducted on a local Linux machine running the four containers and the two ESP32 boards. All the containers are deployed using *Docker Compose*. The two ESP32 boards are connected to the same Wi-Fi network and the *sensor* and *actuator* components are configured to connect to the respective container using the IP address of the machine where the container is running. Finally, the increase and decrease of soil moisture were simulated by verifying that the valve opened and closed properly to maintain the desired level of moisture in the soil.

```
class SoilMoistureBehaviour :
    Behaviour<StateOps, Unit, Double, Boolean, Unit> {
    companion object {
        private const val TARGET_MOISTURE = 30.0
    }

    override fun invoke(
        state: StateOps,
        export: List<Unit>,
        sValues: Double,
    ): BehaviourOutput<StateOps, Unit, Boolean, Unit> {
        val action = sValues < TARGET_MOISTURE
        return BehaviourOutput(
            MoistureState(sValues), Unit, action, Unit
        )
    }
}
```

Listing 6.6: Implementation of the *behaviour* component for the demo 1.

6.4 Demo 2: Multi Devices Multi Components

Demo 2 aims to represent a more complex scenario than the previous one, involving multiple devices and enabling communication between them. Also, we want to introduce more types of devices and show how they are supported by the framework. This demo gets even closer to real scenarios involving multiple types of devices and where communication between them is a prerequisite.

This demo tries to replicate the hot-warm-cold game using two types of devices: an embedded device that needs to be found and many smartphones that need to find it. The smartphones connect to the embedded device via Bluetooth, through which they determine its distance and communicate this information with other smartphones. The embedded device receives the information on the distances of the smartphones and sets a light intensity of an LED proportional to the proximity of the smartphones to it. Thus, the closer the smartphones are to the embedded device, the brighter the led will emit; while the farther away the smartphones are, the less light will be emitted. Each smartphone sends its distance to all other smartphones and simultaneously receives the distance of all other smartphones from the embedded device. Sharing distance information is intended to test inter-device communication while simultaneously providing clues as to where the device that needs to be found is located so that it can be found more quickly.

A Raspberry PI was used as the embedded device since it has both Wi-Fi and Bluetooth. The ESP32 was not chosen, as in the previous demo, because it has hardware limitations that prevent Bluetooth and Wi-Fi from being used simultaneously. As for smartphones, Android smartphones were used, thus allowing the framework to be tested on this platform as well.

When designing a pulverized system, it is good to represent the system from two different viewpoints: a viewpoint that captures the logical level of the interactions between logical devices (see Figure 6.5) and a physical viewpoint that shows how the system is deployed in the infrastructure (see Figure 6.6).

Figure 6.5 shows the network topology and how devices are tied together. This level of abstraction is what should be used by the user who intends to implement the system by exploiting the pulverization framework: he or she does not have to worry about infrastructure or deployment aspects; how communications take place is handled directly by the framework. This greatly simplifies the development of the system. How then the system is deployed in the infrastructure is depicted in Figure 6.6, which shows at the physical level where components are executed and how intra-component communications of each device take place. Moreover, are depicted all the physical devices involved in the system.

Design choices for the implementation of the demo are explained below. The demo is divided into four main modules:

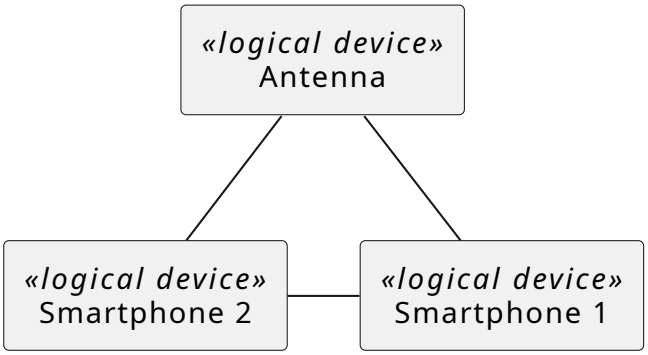


Figure 6.5: Logical diagram of the connection between the logical devices.

- **Android application:** in this module is implemented the mobile application that runs on smartphones
- **Raspberry PI firmware:** this module implements the firmware that runs on the Raspberry PI
- **Common module:** this module contains the common code shared between the *Platform* and *Android application* modules
- **Platform module:** this module contains the code that runs all the devices' components logics.

The *common module* contains the code shared between the *platform* and *android application* modules. In particular, it contains the implementation of each component like the *behaviour, sensors* and *actuators* components, either for the embedded device and the smartphones. The reason why all the components are implemented in the same module is because they can be reused whenever a new device is added to the system. In this way, the specific device should not implement its specific version of the component but instead reuse the one already implemented.

The android application is structured as follows: during the *initialization* stage the pulverization platform with its components is initialized and the *Bluetooth LE* module configured. Then, a user interface is shown to the user, which allows them to start the system by providing the IP of the machine where the platform is running and the device id. Once the user starts the system, the *Bluetooth LE* module starts scanning for nearby devices and, when the embedded device is found (the Raspberry PI), the *Bluetooth LE* module connects to it and starts sending the distance information to all its neighbours. At the same time, the application listen for incoming messages from the neighbours and shows their distance on the screen.

The Raspberry PI firmware is structured as follows: first of all, the *Bluetooth LE* module is configured as a server and starts the advertising process, so that all

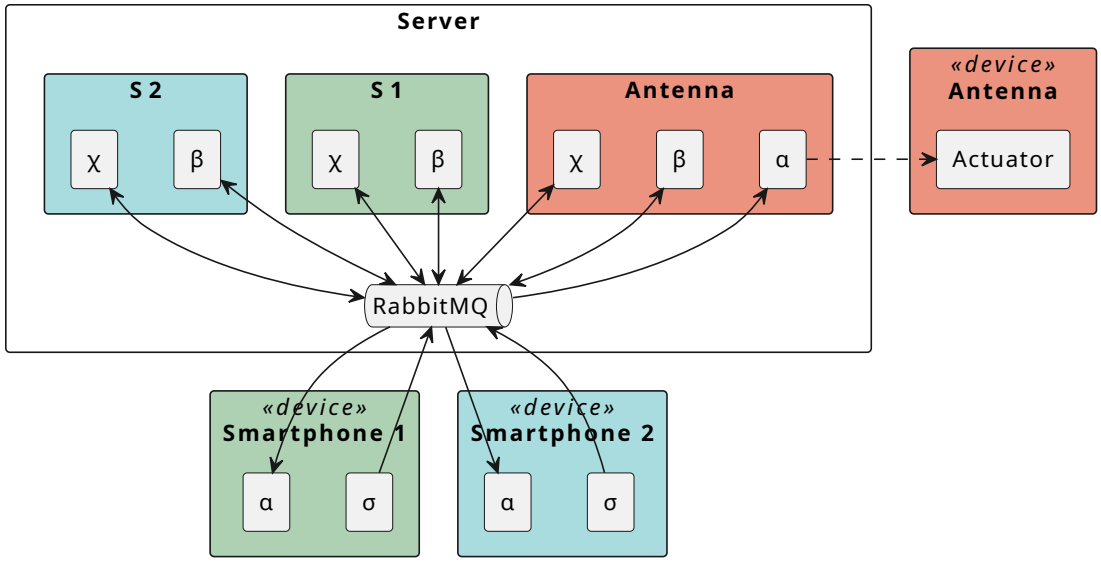


Figure 6.6: Physical diagram of the connection between the logical devices.

the nearby devices can connect to it. Then, a TCP socket is opened toward the *actuator component* deployed on the server. Once the connection is established, all the incoming messages are collected. The message contains a decimal value between 0 and 1 that respectively means LED turned off and LED turned on; all the intermediate values are used to set the LED intensity.

Finally, the *Platform* module defines for each logical device in the system all its deployment units. In particular, runs the *behaviour, sensors* and *actuators* components for each smartphone and the *behaviour* and *actuators* components for the embedded device.

The system was dockerized and deployed using docker compose. The system, once started, was tested by using two smartphones that were continuously moved around the room to observe how it varied the LED intensity accordingly. The conducted test did not reveal any anomalies or malfunctions, except for some inaccuracies in the calculation of the distance of the devices from the Bluetooth antenna due to signal fluctuations.

One notable test that has been conducted involves simulating a device failure and observing how the system reacts to this condition. The failure of a device was simulated by disconnecting it from the network. As soon as the device was disconnected from the network, the system continued to operate as expected. Devices that were still active, store the last message received from that device and thus did not alter their behavior. When the device returns operational in the network, then it will start sending the newly updated data to its neighbors again, which will then update the information about that device's distance from

the antenna.

In conclusion, this demo brought out the effectiveness of the framework in clearly separating business logic development from infrastructural and deployment aspects, highlighting how more complicated systems are achievable through the pulverization framework. Again, it highlighted how a failure of one component does not preclude the system's operation.

This demo can be extended by going to improve the calculation of the distance of the devices from the antenna, for example, by implementing a filter that reduces the noise in the signal acquisition producing more stable and truthful values. Another interesting aspect to analyze may be to have some smartphones with all components running on them, while other smartphones have the behavior running in the cloud, thus observing that the overall behavior does not change as the deployment structure changes.

6.5 Current framework limitations

At present, the framework has some limitations: dynamics and support for different protocols are some examples of shortcomings. The goal of this section is therefore to provide an overview of the main shortcomings of the framework by going on to examine in what contexts these, if implemented, could solve certain problems.

6.5.1 Dynamics

By dynamism, in this context, we mean the ability of the framework to be able to dynamically relocate pulverized components in the infrastructure.

At present, the framework does not handle this aspect: the deployment structure is defined a priori and remains so throughout the life of the executed system. Although dynamism is a key aspect in pulverization, it was decided to focus more on good domain modeling to build a solid foundation on which the framework can be extended, rather than implementing as many aspects as possible running the risk of creating a rigid framework that is not very extensible and difficult to use.

6.5.2 Multi-protocols

The management of multiple protocols to enable intra-component communication is an important aspect since pulverization can involve a large number of heterogeneous devices that have different computational and communication capabilities.

For this reason, the choice of one protocol over another is not mutually exclusive, but it may be appropriate, for example, to provide multiple protocols that are used in the pulverized system. Again, you might be dealing with devices that do

not support certain protocols, or you are in a situation where you want to use a very lightweight protocol (e.g. MQTT) for devices with very low computational resources and instead use a higher-performance protocol for those parts of the system with high computational power.

At the time of writing, the only protocol implemented to enable intra-component communication is RabbitMQ as it represents a good compromise between ease of use, performance and adoption.

Adoption of additional protocols would increase the framework's potential to be used in many contexts with strong device heterogeneity.

6.5.3 Performance evaluation

Currently, no performance evaluation has been conducted on the framework. This is a very important aspect that must be addressed in the future to understand how the framework behaves in terms of performance and scalability. In particular, it is important to understand, in terms of latency and throughput, how these aspects are affected using different deployment strategies. For example, could be interesting to understand how the latency and throughput are affected when all the components are deployed on the same deployment unit (leveraging an in-memory communication) and when the components are deployed on different deployment units leveraging the *communicator* implementation to communicate.

Moreover, another aspect to evaluate could be how the latency and throughput are affected when different protocols are used at the same time: for example, when different *communicator* implementations are used to opportunistically exploit the underlying infrastructure.

The main reason why this aspect has not been addressed yet is that all the effort has been focused on testing the operational semantics of the framework checking that all the functional requirements are met. However, the performance aspect is very important and must be addressed in the future.

Chapter 7

Conclusions

The increasing use of Internet of Things (IoT) devices and the resulting large amounts of data being produced present significant challenges for cloud computing. While cloud computing has proven effective for many applications, it may not always be suitable for real-time constraints and handling data from IoT devices.

Fog computing offers a promising solution by providing a computing model that sits between IoT devices and the cloud, allowing for the collection, aggregation, and processing of data using a hierarchy of computing power. Combining fog computing with the cloud can reduce data transfers and communication bottlenecks, as well as contribute to reduced latencies. However, realizing systems that operate in the edge-cloud continuum is a complex challenge due to the heterogeneity of devices and dynamic requirements of today's systems.

Various approaches have been proposed to address these challenges, including self-organizing systems and methodologies such as osmotic computing. The orchestration of distributed applications requires careful management of the underlying infrastructure to enable the reuse of design elements across different scenarios.

The thesis focuses more specifically on the pulverization approach: a framework that breaks the system behaviour into small computational pieces that are continuously executed and scheduled in the available infrastructure. In this way, the business logic of a system is neatly separated from infrastructure or deployment concerns enabling the concept of *deployment independent* systems. Reuse and independency from the deployment are the two main pillars of the pulverization approach, by which the framework aims to enable the deployment of systems in the edge-cloud continuum.

The main contribution of the thesis is the development of a framework that leverages the pulverization approach to deploy Cyber-Physical Systems. The framework aims to lay the groundwork for closing the gap between the simulation of these systems and their deployment by exploiting the pulverization methodology.

The framework is built trying to maintain ease of use, modularity, and ex-

tensibility by modeling the foundational concepts of pulverization over which the framework can be extended and improved.

Several technology solutions were examined that could support cross-platform targets allowing the framework to be used seamlessly across different platforms and architectures. Kotlin multiplatform was identified as a suitable technology for the development of the framework since a wide range of architectures is supported.

The most significant implementation details and technology challenges that were faced during the development of the framework and what solutions were employed to achieve it were reported.

Finally, topics such as testing and validation were covered by showing what strategies were used to validate the framework, as well as relevant demos were developed to show how the framework works in different contexts, each of which brings with it peculiar characteristics that go to corroborate the proper operation and effectiveness of the framework.

From using the framework, has emerged that deployment or infrastructural aspects never appear during system implementation, leading to the advantage of most efforts on development, delegating infrastructure and communication aspects to the framework. In addition, it is apparent how the reusability of the developed components is easily achieved by design: this leads to the same component being able to be reused in different deployment strategies, making the application flexible to changes in how it is deployed.

7.1 Future Work

The framework is still in its early stages of development and there is still a lot of work to be done to make it more robust and complete. The following are some of the topics that could be explored in future work.

Due to the heterogeneity of the devices that can be used in the edge-cloud continuum, the framework should be able to support different communication protocols. This would allow the framework to be used in a wider range of scenarios allowing the use of different communication protocols based on device capabilities. Moreover, this work can be extended to support different communication protocols at the same time to opportunistically exploit the best protocol for the actual system requirements or quality of services.

Dynamics is a key aspect of the edge-cloud continuum, and the framework should be able to support dynamic changes in the system. This would allow the framework to be used in scenarios where the system is subject to changes in the number of devices, the number of resources, or opportunistically exploit the best deployment strategy for the actual system.

At present, to deploy the system, certain manual procedures, such as con-

tainerization of the deployment units, which are then executed on the existing infrastructure, need to be carried out. By automating this process, the deployment time can be reduced, and the risk of errors in container deployment can be minimized. Consequently, it is wise to investigate how containers can be automatically deployed into the infrastructure via DevOps (CI/CD) methodologies

With this thesis, I brought in research topics and problems such as pulverization. It was motivating to delve into and understand the concepts of pulverization and carry them into the implementation of a framework. Was interesting to see the framework work in different contexts and understand how it could be evolved in the future. This experience allowed me to study the literature and understand what related work has already drawn useful insights from it to implement the framework. In addition, this thesis allowed me to delve into the Kotlin ecosystem in its multiplatform version by understanding how this technology may be suitable to support the implementation of the framework.

The completion of this thesis has facilitated my technical growth in two distinct ways. Firstly, it provided me with the opportunity to explore various aspects and tools within the field, thereby enhancing my technical capabilities. Secondly, it enabled me to tackle novel issues by motivating me to seek innovative solutions to support the project.

Bibliography

[1] Luiz F. Bittencourt et al. "The Internet of Things, Fog and Cloud continuum: Integration and challenges". In: *Internet Things* 3-4 (2018), pp. 134–155. DOI: 10.1016/j.iot.2018.09.005. URL: https://doi.org/10.1016/j.iot.2018.09.005.

[2] Massimo Villari et al. "Osmosis: The Osmotic Computing Platform for Microelements in the Cloud, Edge, and Internet of Things". In: *Computer* 52.8 (2019), pp. 14–26. DOI: 10.1109/MC.2018.2888767. URL: https://doi.org/10.1109/MC.2018.2888767.

[3] Rim El Ballouli et al. "Four Exercises in Programming Dynamic Reconfigurable Systems: Methodology and Solution in DR-BIP". In: *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 11246. Lecture Notes in Computer Science. Springer, 2018, pp. 304–320. DOI: 10.1007/978-3-030-03424-5_20. URL: https://doi.org/10.1007/978-3-030-03424-5%5C_20.

[4] Marius Bozga et al. "Modeling Dynamic Architectures Using Dy-BIP". In: *Software Composition - 11th International Conference, SC@TOOLS 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*. Ed. by Thomas Gschwind et al. Vol. 7306. Lecture Notes in Computer Science. Springer, 2012, pp. 1–16. DOI: 10.1007/978-3-642-30564-1_1. URL: https://doi.org/10.1007/978-3-642-30564-1%5C_1.

[5] Rocco De Nicola, Alessandro Maggi, and Joseph Sifakis. "The DReAM framework for dynamic reconfigurable architecture modelling: theory and applications". In: *Int. J. Softw. Tools Technol. Transf.* 22.4 (2020), pp. 437–455. DOI: 10.1007/s10009-020-00555-2. URL: https://doi.org/10.1007/s10009-020-00555-2.

[6] Roberto Casadei et al. "Pulverization in Cyber-Physical Systems: Engineering the Self-Organizing Logic Separated from Deployment". In: *Future Internet*

12.11 (2020), p. 203. DOI: 10.3390/fi12110203. URL: https://doi.org/10.3390/fi12110203.

[7] Michael Armbrust et al. "A view of cloud computing". In: *Commun. ACM* 53.4 (2010), pp. 50–58. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672.

[8] Antero Taivalsaari, Tommi Mikkonen, and Kari Systä. "Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture". In: *IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*. IEEE Computer Society, 2014, pp. 338–343. DOI: 10.1109/COMPSAC.2014.56. URL: https://doi.org/10.1109/COMPSAC.2014.56.

[9] Ananda Basu et al. "Rigorous Component-Based System Design Using the BIP Framework". In: *IEEE Softw.* 28.3 (2011), pp. 41–48. DOI: 10.1109/MS.2011.27. URL: https://doi.org/10.1109/MS.2011.27.

[10] Jacob Beal, Danilo Pianini, and Mirko Viroli. "Aggregate Programming for the Internet of Things". In: *Computer* 48.9 (2015), pp. 22–30. DOI: 10.1109/MC.2015.261. URL: https://doi.org/10.1109/MC.2015.261.

[11] Gianluca Aguzzi et al. "Towards Pulverised Architectures for Collective Adaptive Systems through Multi-Tier Programming". In: *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021, Companion Volume, Washington, DC, USA, September 27 - Oct. 1, 2021*. Ed. by Esam El-Araby et al. IEEE, 2021, pp. 99–104. DOI: 10.1109/ACSOS-C52956.2021.00033. URL: https://doi.org/10.1109/ACSOS-C52956.2021.00033.

[12] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. "A Survey of Multitier Programming". In: *ACM Comput. Surv.* 53.4 (2021), 81:1–81:35. DOI: 10.1145/3397495. URL: https://doi.org/10.1145/3397495.

[13] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. "Distributed system development with ScalaLoci". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 129:1–129:30. DOI: 10.1145/3276499. URL: https://doi.org/10.1145/3276499.

[14] Pascal Weisenburger and Guido Salvaneschi. "Implementing a Language for Distributed Systems: Choices and Experiences with Type Level and Macro Programming in Scala". In: *Art Sci. Eng. Program.* 4.3 (2020), p. 17. DOI: 10.22152/programming-journal.org/2020/4/17. URL: https://doi.org/10.22152/programming-journal.org/2020/4/17.

[15] *Domain-driven design - tackling complexity in the heart of software*. Addison-Wesley, 2004. ISBN: 978-0-321-12521-7.

[16] Sébastien Doeraene. "Cross-Platform Language Design". PhD thesis. EPFL, Switzerland, 2018. DOI: 10.5075/epfl-thesis-8733. URL: https://doi.org/10.5075/epfl-thesis-8733.

[17] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. "Cross-language compiler benchmarking: are we fast yet?" In: *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*. Ed. by Roberto Ierusalimschy. ACM, 2016, pp. 120–131. DOI: 10.1145/2989225.2989232. URL: https://doi.org/10.1145/2989225.2989232.

[18] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. "Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript". In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*. IEEE Computer Society, 2015, pp. 247–256. DOI: 10.1109/ESEM.2015.7321196. URL: https://doi.org/10.1109/ESEM.2015.7321196.

[19] Roman Elizarov et al. "Kotlin coroutines: design and implementation". In: *Onward! 2021: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Virtual Event / Chicago, IL, USA, October 20-22, 2021*. Ed. by Wolfgang De Meuter and Elisa L. A. Baniassad. ACM, 2021, pp. 68–84. DOI: 10.1145/3486607.3486751. URL: https://doi.org/10.1145/3486607.3486751.

[20] Steve Vinoski. "Advanced Message Queuing Protocol". In: *IEEE Internet Comput.* 10.6 (2006), pp. 87–89. DOI: 10.1109/MIC.2006.116. URL: https://doi.org/10.1109/MIC.2006.116.