ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS OF CESENA

Computer Science and Engineering Department
Second Cycle Degree in Computer Science and Engineering

Developing Distributed Programs For The Cloud-Edge Computing Combining Multitier And Aggregate Programming

Thesis in
Pervasive Computing

Supervisor
Prof. Mirko Viroli

Co-supervisors
Prof. Guido Salvaneschi
Dr. Pascal Weisenburger

Presented by
Linda Vitali

Academic Year 2022-2023

To my amazing dad.

To my sister. You can do it.

Abstract

In the digital era, the increasing number of interconnected devices caused by the advent of Internet of Things and cyber-physical systems, has made distributed systems more and more pervasive as well as difficult to develop. This has brought about new challenges in terms of complexity and heterogeneity in the infrastructure on which these systems are deployed. For instance, in a scenario where the devices are distributed across the cloud and the edge. To tackle this complexity, two programming paradigms emerged in the last years: aggregate and multitier programming. The former is a paradigm that addresses the development of large-scale distributed systems by considering the system as a collection of devices (i.e. aggregates). Those devices can be programmed by using functions composition. The latter is a paradigm that allows the development of distributed systems by abstracting from the communication layer and developing every component of a distributed system in a single code base. The integration of these two paradigms can be beneficial for the development of the so-called Cloud-Edge continuum since aggregate computing can be used to program the logic of the devices at the edge, and the multitier paradigm can be used to address specific nodes in the cloud as well as easily deploy the system's components. The work described in this thesis aims to investigate the integration of the two programming models and to provide a middleware that uses the aggregate programming language ScaFi and the multitier programming language ScalaLoci to allow the development of distributed systems that can be deployed on the cloud, the edge, or a combination of both.

Acknowledgements

I would like to thank Prof. Mirko Viroli and Prof. Guido Salvaneschi, for giving me the opportunity of this experience abroad and for their guidance and valuable advice during the development of this thesis. I would also like to thank Dr. Pascal Weisenburger, Dr. Roberto Casadei, and Gianluca Aguzzi for their patience and essential help. Thanks to the people I encountered in St. Gallen, particularly the Programming Group and the others in Torstrasse 25, for the breaks, the food, the stories, the hikes, and the laughs. A profound thanks to my family for their support and love. I want to thank Pierre for believing in me in every single thing I do. A special thanks to my Atedeg mates: you taught me so much and helped me during the master's years without even realizing it. Finally, a thanks to all my friends, you are the people who have been there for me since the very beginning (I love you even though some of you think I'm graduating in "fixing printers").

Contents

Abstract **i**

1 Introduction **1**
1.1 Motivations and goals . . . 2
1.2 Thesis outline . . . 4

2 Background **5**
2.1 Aggregate computing . . . 5
2.1.1 Assumptions made by aggregate programming . . . 6
2.1.2 Execution model . . . 7
2.2 ScaFi . . . 7
2.3 Multitier programming . . . 9
2.4 ScalaLoci . . . 10
2.4.1 Multitier modules . . . 11

3 State of the art **15**
3.1 Multitier pulverized aggregate computing . . . 15

4 Towards a seamless integration of ScaFi and ScalaLoci **19**
4.1 Dynamic connections . . . 19
4.2 Aggregate base station . . . 21
4.3 Considerations . . . 22

5 Middleware requirements **23**
5.1 Business requirements . . . 23
5.2 User requirements . . . 23
5.3 Functional requirements . . . 24

6 Design **25**
6.1 Middleware architecture . . . 25
6.2 Implementation . . . 29
6.2.1 Node's communication strategy . . . 29

6.2.2 Topology definition with multitier modules 29

7 Evaluation 31

7.1 Case studies 32
7.2 Preservation of fault tolerance 34
7.3 Scafi-Loci middleware on Android devices 34
7.4 Middleware limitations 36
7.4.1 Neighboring logic 36
7.5 Different nodes interaction 36
7.6 Middleware and pulverization 36
7.6.1 Scalability concerns 36
7.6.2 Limited Error Handling 37
7.6.3 Performance evaluation 37

8 Conclusion 39

8.1 Future work 40

Bibliography 43

List of Figures

2.1 Compiling process in multitier programming [7] 10

3.1 Pulverized device in ScalaLoci [10] 16
3.2 ScaFi and ScalaLoci integration [10] 17

6.1 Middleware architecture's modules 26
6.2 Middleware architecture with topology configuration 26
6.3 Runtime module 27
6.4 Configuration module 28
6.5 Default topologies provided 30
6.6 General topologies 30

Listings

2.1 Field constructs in ScaFi . . . 8
2.2 An aggregate program in ScaFi . . . 9
2.3 Multi-client-server architecture in ScalaLoci . . . 10
2.4 Int placed on Node . . . 11
2.5 Encapsulation with multitier modules . . . 12
2.6 Abstract peer types . . . 12
4.1 First integration experiment's main method . . . 20
4.2 Connections management . . . 20
4.3 Monitoring multitier module . . . 21
4.4 Monitoring multitier module . . . 21
6.1 Architecture trait . . . 29
7.1 General hybrid multi-client-server topology . . . 33
7.2 General pure multi-client-server topology . . . 33
7.3 General hybrid peer-to-peer topology . . . 33
7.4 Case study example code . . . 34
7.5 ScalaLoci code to enable the compilation of Scala both to JVM bytecode and to Javascript . . . 35

Chapter 1

Introduction

The vast expanse of digital transformation has made distributed systems an indispensable asset of modern computing. Their importance is underscored by the evolution of technology, from individual isolated systems to intricate networks of interconnected devices. While these systems present myriad opportunities, especially in scaling processes and enabling ubiquitous access to data and services, they are also abounding with challenges. These complexities span from the technical nuances of concurrency and asynchronous execution to the operational challenges of message losses and unpredictable device malfunctions.

The transformative impact of pervasive computing, Internet of Things (IoT), and cyber-physical systems (CPS), leading to increasing interconnection of devices, further amplifies the role and challenges of distributed systems. As potentially thousands of devices come online, generating and processing vast amounts of data, traditional Cloud computing finds its limitations. Cloud's characteristic strength in centralizing computing often becomes a problem when real-time processing and responsiveness are critical. It is against this backdrop that Edge Computing emerges bridging the gap between data generation and data processing. By situating computational tasks closer to where data originates, at the "edge" of the network, Edge Computing offers not just reduced latency, but a promise of more astute, immediate, and context-aware data handling. The integration of cloud and edge — often termed "cloud-edge computing" — proposes a balanced approach, synergizing the robustness of centralized data processing with the agility of decentralized computation, providing a more holistic approach to distributed system design.

An important aspect to consider in the context of Cloud-Edge computing is the intricate coordination between a multitude of devices functioning in a distributed environment. This coordination presents adaptive challenges, necessitating systems to automatically adjust in response to fluctuating environmental factors or changing needs. The interconnected web of devices in this environment must possess not only singular intelligence for independent operation but also the ability to act

collectively. Beyond this collective intelligence, the system must be endowed with self-adaptive and self-organizing capacities to gracefully handle variations, be they in device distribution, energy supply, computational demands, or unexpected faults. Also, as these devices span various locations, platforms, and functions, ensuring security, interoperability, and resilience becomes paramount.

Researchers explore new approaches to make the development of distributed systems easier, and every approach focuses on different aspects of the problem: some on the communication aspect, others on the data representation, and others on the programming model. The aspects considered in this thesis is related to combining programming models that allow programmers to write code for homogeneous and collective devices that is easy to maintain by adding abstraction levels over low-level implementation details such as data representation, serialization, and networking.

Aggregate programming [1] and multitier programming [2] are two programming models considered in this thesis. The first one emphasizes a collective perspective where the basic unit of computing is no longer a single device but instead a cooperating collection of devices. The second one deals with developing the components of different *tiers* in a distributed system (e.g., client and server), mixing them in the same compilation unit allowing to target the programming of specific devices.

1.1 Motivations and goals

This thesis work was carried out while being hosted by the University of St. Gallen at the Programming Group, a research group of the Institute of Computer Science (ICS) which, among other things, works on topics like programming languages and software engineering including languages and architectures for distributed systems. The group developed the ScalaLoci [3] programming language, which allows for writing type-safe multitier distributed applications.

The collaboration with this group lies in the necessity of combining the programming of collective and homogeneous devices on the Edge, with the programming of specific and well-defined devices on the Cloud. To fulfill this need, the idea is to combine the power of aggregate programming with multitier programming using ScalaLoci and ScaFi [4].

The two main motivations and goals of this thesis are summarized as follows:

- Aggregate programming is a good fit for the programming of “weak” nodes (e.g. sensors on the edge) and it makes very few assumptions on the underlying physical system, while the latter is a good fit for the programming of “strong” nodes (e.g. servers on the cloud) and it does a lot of assumptions about the underlying physical system. The combination of the two programming models

will allow the programmer to target the programming of either collective or specific devices, depending on the needs of the system

- Even without making assumptions about the underlying hardware, the use of aggregate programming and its benefits for programming collective devices can be simplified by leveraging the benefits of multitier programming. This allows the developers to focus only on the logic of the system, designing the aggregate program to execute and the topology of the network, abstracting away the low-level details of the underlying system.

This thesis proposes the realization of a middleware that leverages the power of the Aggregate and Multitier programming through the use of ScalaLoci and ScaFi which will be described in detail in the following chapter. The middleware will be able to execute aggregate programs and deploy them on different architectures, in the Cloud, the Edge, or a combination of both. Moreover, it will allow the integration and communication between aggregate and non-aggregate programs.

To showcase the effectiveness of the middleware, some scenarios and example have been identified and implemented. Every example uses a different architecture with the same aggregate program, showing the flexibility of the middleware and the possibility to target different devices. The case studies include both the execution of an aggregate program in every device of the network, and the execution of a specific program in a single device that communicates with the aggregate nodes, showing the possibility to target both collective and specific devices at the same time.

For what concerns the testing of the middleware, a test suite has been developed in order to validate the correct execution of the Aggregate Programming execution model and its integration with ScalaLoci.

Moreover, some test has been done to run the middleware on Android devices through the use of Scala.js[1] supported by ScalaLoci.

[1] https://www.scala-js.org/

1.2 Thesis outline

The thesis's structure is organized as follows: Chapter 2 provides the necessary background information to better understand the previously illustrated motivations of this thesis and how the frameworks on which the project is based work. Chapter 3 describes the related work and the state of the art of integration between aggregate and multitier programming. Chapter 4 describes the first steps of this thesis work, describing the implementation of a prototype of the integration. Chapter 5 describes the requirements of the project, Chapter 6 the design and implementation, and Chapter 7 describes the case studies as well as considerations about the middleware's current limitations. Finally, Chapter 8 concludes the thesis and provides an overview of future work.

Chapter 2

Background

This chapter provides the necessary background information to better understand the previously illustrated motivations of this thesis and how ScaFi and ScalaLoci — on which the project is based — work. The first section provides an overview of Aggregate Programming the second section describes the ScaFi framework, the third section consider Multitier programming aspects and the fourth section defines ScalaLoci.

2.1 Aggregate computing

Aggregate computing simplifies the design, creation, and maintenance of complex distributed software systems guaranteeing the reusability and composability of components for collective adaptive behavior. It supports the construction of layered APIs with formal behavior guarantees, sufficient to readily enable the creation of complex applications [5]. This paradigm is based on the concept of *computational field*, which offers a compelling solution to bridge the divide between the macro-level (defining the collective behavior of the system) and the micro-level (actions and interactions performed by individual devices to realize the collective behavior). By acting as an intermediary, computational fields enhance the development of collective APIs and complex systems, enabling work at more elevated levels of abstraction.

With this technique, the basic unit of computing is no longer a single device but instead a cooperating collection of devices [6]. Aggregate Computing provides a functional programming model that enables three main elements:

- collective behavior is abstract and composable
- the focus is on the desired outcomes at a high level (aggregate level) without having to micro-manage each component Individual components collabo-

ratively adapt and organize themselves to realize a collective objective or behavior.

- flexibility of mapping aggregate computation onto the considered infrastructure

2.1.1 Assumptions made by aggregate programming

In this model, it is presumed that a potentially distributed platform exists, supplying each device in the aggregate with local computational and interactive functions. Specifically:

1. Neighborhood: At any given time, a device is surrounded by a set of devices it can communicate with — its "neighborhood". This set can be fixed or can evolve, reflecting changes like movement or system failures.

2. Sensors: A device can fetch data from its local sensors when needed. These sensors grant access to observable environmental factors. Generally, the sensor setup is consistent across devices. For instance, sensors might detect a device's unique identifier or the ambient temperature.

3. Actuators: A device can undertake actions impacting either itself or its surroundings.

4. Message Reception: As devices interact, they swap messages. At any time, a device can access a record of the most recent communications from its neighbors, with previous messages being omitted.

5. Message Broadcast: A device can, on demand, transmit a message to all surrounding devices. This communication process is asynchronous and retains order. Message losses are compensated by adjustments in neighbor connections.

6. State: Every device has local storage to retain data over time.

7. Computation: Each device divides computations into segments referred to as "computational rounds". These rounds are concise and conclusive, relying on the local functionalities listed above and yielding a potentially organized output. Such output could, for example, activate actuators.

8. Scheduling: Local schedulers initiate these computational rounds. In general, the process is asynchronous yet equitable, with devices possibly operating at diverse rates. It is essential to note that a new round is only scheduled after the completion of the preceding one.

Furthermore, it is crucial to understand that an aggregate primarily signifies a logical network of devices. This network can be variously overlaid onto the tangible, physical network of computing nodes, and in simple setups, there might be a direct correspondence between logical and physical devices. For instance, in a robot cluster, each robot might independently run the aggregate program in its specific context.

2.1.2 Execution model

Each device processes the comprehensive aggregate program by structuring computational rounds in the following manner:

1. It discerns its local context by gathering environmental status samples using sensors and receiving messages from neighbors.

2. The device then reassesses the aggregate program based on this local context, yielding an output along with a "coordination message" termed as "export".

3. The device then interacts with the environment utilizing actuators, as directed by the program.

4. Subsequently, the device conveys the export to neighboring devices. This step essentially serves to notify the nearby area about alterations in the local context, facilitating the transition from local to global state progression.

Given that the computational rounds across different devices often occur asynchronously, aggregate programs typically depict the way an entire aggregate gradually adjusts to environmental shifts and internal system changes (e.g., failures, movement) and produces eventually consistent responses.

2.2 ScaFi

ScaFi, short for Scala Computational Fields, is a toolkit designed for building aggregate systems using the Scala programming language, from which inherits the syntax and semantics. It offers a domain-specific language (DSL), an API that effectively acts as an "embedded language". Alongside this, ScaFi provides a comprehensive library of functions tailored for field programming and other essential development utilities, including simulation tools. The field constructs are captured by the `Constructs` trait (Listing 16). Higher-level functions can be defined by combining the constructs to capture increasingly complex collective behavior.

```scala
trait Constructs {
	def rep[A](init: => A)(fun: A => A): A
	def nbr[A](expr: => A): A
	def foldhood[A](init: => A)(acc: (A, A) => A)(expr: => A): A
	def aggregate[A](f: => A): A

	// the following (aggregate IF construct) can be defined upon AGGREGATE()
	def branch[A](cond: => Boolean)(th: => A)(el: => A)
	// the following is a variant of REP()
	def share[A](init: => A)(fun: (A, () => A) => A): A

	def mid: ID
	def sense[A](sensorName: String): A
	def nbrvar[A](name: CNAME): A
}
```

Listing 2.1: Field constructs in ScaFi

This interface is implemented by an abstract class called `AggregateProgram` which provides its subclasses with access to the filed constructs. A brief description of what these elements do is the following:

- `rep` captures state evolution, starting from an init value that is updated each round through fun
- `nbr` captures communication, of its expr value, with neighbors; it is used only inside the argument expr of foldhood, which supports data aggregation of neighborhood-dependent data to single values, through the input accumulator function acc;
- `branch` captures domain partitioning, or space-time branching;
- `mid` is a built-in sensor providing the identifier of devices;
- `sense` abstracts access to local sensors; and
- `nbrvar` abstracts access to neighboring sensors that behave similarly to nbr but are provided by the platform.

As mentioned before, an aggregate program specifies collective behavior in terms of both computation and neighbor-to-neighbor interaction. To write an aggregate program it is sufficient to extend the `AggregateProgram` class and implement the `main` method and compose the constructs to obtain the desired behavior. Listing 6 shows an example of an aggregate program that simply counts how many rounds each device has executed.

```
1     class MyAggregateProgram extends AggregateProgram {
2         override def main(): Any = {
3             rep(0){ x => x + 1 }
4         }
5     }
```

Listing 2.2: An aggregate program in ScaFi

2.3 Multitier programming

A typical distributed system's structure is multi-tiered, meaning it consists of several layers, with each layer addressing a specific functional aspect, such as data handling or application processes. Traditionally, these unique tiers and functionalities that span across them have been developed as separate compilation units, frequently utilizing different programming languages. This approach tends to increase both the initial development and ongoing maintenance expenses. In multitier programming, components specific to various system tiers, e.g. client and server, are integrated into a single compilation unit with a single programming language. Depending on the language of choice, the code of different tiers is then either generated at run time or split by the compiler following user annotations and static analysis, types, or a combination of these. The developer doesn't have to worry about low-level concerns (e.g. network communication, serialization, and data conversions) since is the compiler that breaks the computation into deployment units (Figure 2.1). It also generates the communication code that is required for such modules to interact during program execution. In type-level multi-tier programming, the specification leverages the type system of the language to ensure the correctness and coherence of the architecture.

This paradigm allows the description of a distributed system in a declarative way, where the programmer specifies the system as a whole and not every component separately enforcing the coherence of the system as well as formal reasoning and software design [7].

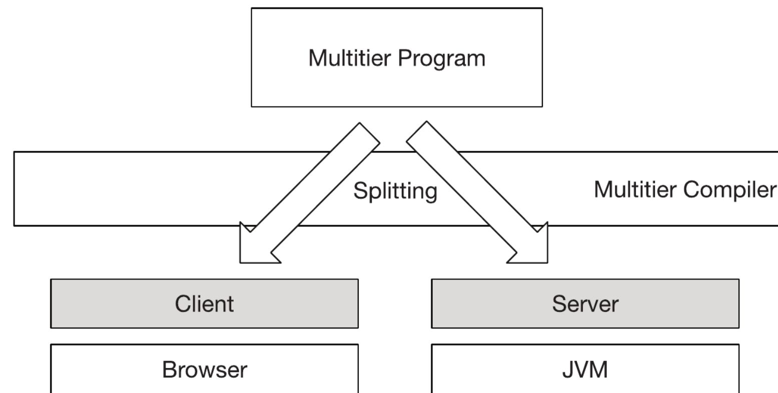

Figure 2.1: Compiling process in multitier programming [7]

The multitier language considered in this thesis is ScalaLoci, which will be described in detail in the next section.

2.4 ScalaLoci

Multitier languages are often integrated into general-purpose languages by extending them with support for distribution. This is the case of ScalaLoci, a type-safe language hosted in Scala language that extends it with multitier abstractions. ScalaLoci allows the programmer to abstract over low-level communication details and data conversions and brings to the world of multitier languages the possibility to specify an architecture based on peer types, thus, supporting generic distributed systems whose architecture can be defined by the developer. Also, with ScalaLoci is possible to associate locations both with data and computations.

The main language constructs of ScalaLoci are *peers* and *ties*. A peer represents the different kinds of components of the system, like a server, a client, a database, etc. A tie specifies the kind of relation among peers, like Single, Multiple, or Optional. Only tied peers can communicate with each other. Listing 2.3 depicts an example of a multi-client-server architecture in ScalaLoci, where a server is tied to multiple clients and a client is tied to a single server. Placement types are the core concept of ScalaLoci and the peers defined in the system architecture to allow specifying the placement of data and computations. Placement is part of the type system and the type checker can reason about resource location in the application.

```
1  @multitier object SimpleSystem {
2      @peer type Server <: { type Tie <: Multiple[Client] }
3      @peer type Client <: { type Tie <: Single[Server] }
```

```
4
5           on[Client] { println("Hello, I'm the client!") }
6           on[Server] { println("Hello, I'm the server!") }
7       }
```

Listing 2.3: Multi-client-server architecture in ScalaLoci

The sample compiles two executables representing the client and the server, whose instances can be deployed and executed on different physical nodes.

Placed data of type `T on P` represent a value of type T that is placed on peer P. For instance, is possible to define a value of type `Int` (e.g. the node's id) that is placed on a peer of type `Node` as shown in Listing 2.4.

```
1 @peer type Node <: { type Tie <: Multiple[Node] }
2 val id: Int on Node = UUID.randomUUID().hashCode()
```

Listing 2.4: Int placed on Node

ScalaLoci uses asynchronous multitier reactives like signals and events to allow the composition of non-blocking data flows that span across multiple tiers. Events are used to represent discrete changes, while signals are used to represent continuous time-changing values.

To access the value of a placed data there are different ways depending on the type of tie. To access the value of a placed data on a peer with a single tie, it is used the expression `.asLocal` while to access the value of a placed data on a peer with multiple ties, the expression `.asLocalFromAll` is provided. Accessing remote values creates a local representation of the remote value by transmitting it over the network or by establishing a remote dependency.

2.4.1 Multitier modules

One of the main features of ScalaLoci is the possibility to define multitier modules [8] to support strong interfaces and achieve encapsulation and information hiding, such that implementations can be easily exchanged. This approach makes peer types abstract enabling the definition of abstract modules, which capture a specific component of a distributed system. Every module can be further composed with other abstract modules and can eventually be instantiated for a concrete software architecture. This language mechanism allows developers to design applications based on logical functionalities rather than network boundaries.

The example in Listing 2.5 show an example of encapsulation using a module for a simple Chat that defines a client peer type and a server peer type. The `ChatApp` also requires an instance of the MessageService multitier module to handle messaging functionalities.

```
@multitier trait ChatApp {
	@peer type Client <: { type Tie <: Single[Server] }
	@peer type Server <: { type Tie <: Single[Client] }
	val messageService: MessageService
}

@multitier trait MessageService {
	@peer type UserInterface <: { type Tie <: Single[Database] }
	@peer type Database <: { type Tie <: Single[UserInterface] }

	def sendMessage(message: String): Unit on UserInterface =
	placed { remote call storeMsg(message) }

	def fetchMessages(): Future[List[String]] on UserInterface =
	placed { (remote call retrieveMsgs()).asLocal }

	private def storeMsg(message: String): Unit on Database =
		// storing logics

	private def retrieveMsgs(): List[String] on Database =
		// retrieving logics
}
```

Listing 2.5: Encapsulation with multitier modules

The `MessageService` module specifies two peer types: `UserInterface` for handling user interactions and `Database` for storing and retrieving messages. The `sendMessage` and `fetchMessages` methods can be invoked on the `UserInterface` peer and make remote calls to `storeMessage` and `retrieveMessages` methods on the `Database` peer. This module encapsulates all the messaging functionalities, including the communication between the `UserInterface` and the `Database`.

Other than encapsulation and the definition of module interfaces, LociMod modules enable abstracting over placement using abstract peer types. Peer types are abstract type members of traits, i.e. they can be overridden in sub-traits specializing their type.

```
@multitier trait ChatApp {
	@peer type Client <: messageHandler.UserInterface {  type
    Tie <: Single[Server] }
	@peer type Server <: messageHandler.Database { type Tie <:
    Single[Client] }
	val messageHandler: MessageService
}
```

Listing 2.6: Abstract peer types

As shown in the example, the peers defined in a module can be specialized with

the role of other modules' peers. This is useful if `UserInterface` or the `Database` are not physical peers in the system, but just a logical place.

Multiter modules can be mixed-in and this enables including the implementations of different subsystems in a single module. However, it is necessary to use mechanisms like subtyping or overriding to specify that a peer also implements the placed values of the overridden or subtyped peers.

One last important feature is the possibility to specify constrained modules, meaning that it is possible to express that a functionality is required by a module to make it work. This is realized by Scala's self-type[1] annotations.

[1] `https://docs.scala-lang.org/tour/self-types.html`

Chapter 3

State of the art

This chapter describes an existing integration approach between aggregate programming and multitier programming with ScalaLoci in the context of *pulverization* [9]. This was the starting point for the work presented in this thesis.

3.1 Multitier pulverized aggregate computing

Pulverization is a technique proposed for aggregate computing in which behavioral and deployment concerns are separated. A logical device is decomposed into micro-components that can be deployed independently and it is possible to abstract away from the underlying communication protocol. Those components are Sensors, Actuators, State, Behavior, and Communication. Since this approach does not provide a way to specify deployment concerns, in the paper [10] the authors propose a combination with multitier programming, in particular using ScalaLoci, in order to provide such a specification in a declarative and statically checked way. This unification also allows the deployment and execution on multiple different network structures of pulverized systems. The result introduces a novel architecture designed for multitiered deployment strategies in pulverized systems that allows to specify functional behavior independently from deployment.

The logical system of a pulverized aggregate program is decomposed into multitier modules, and it is possible to define every function associated with each pulverized component. Moreover, it is possible to decide the network structure of the system in terms of connections among nodes of different kinds (e.g. cloud, edge, etc.). Finally, each pulverized component can be assigned to a certain node kind. Figure 3.1 represents a possible implementation of a pulverized device in ScalaLoci.

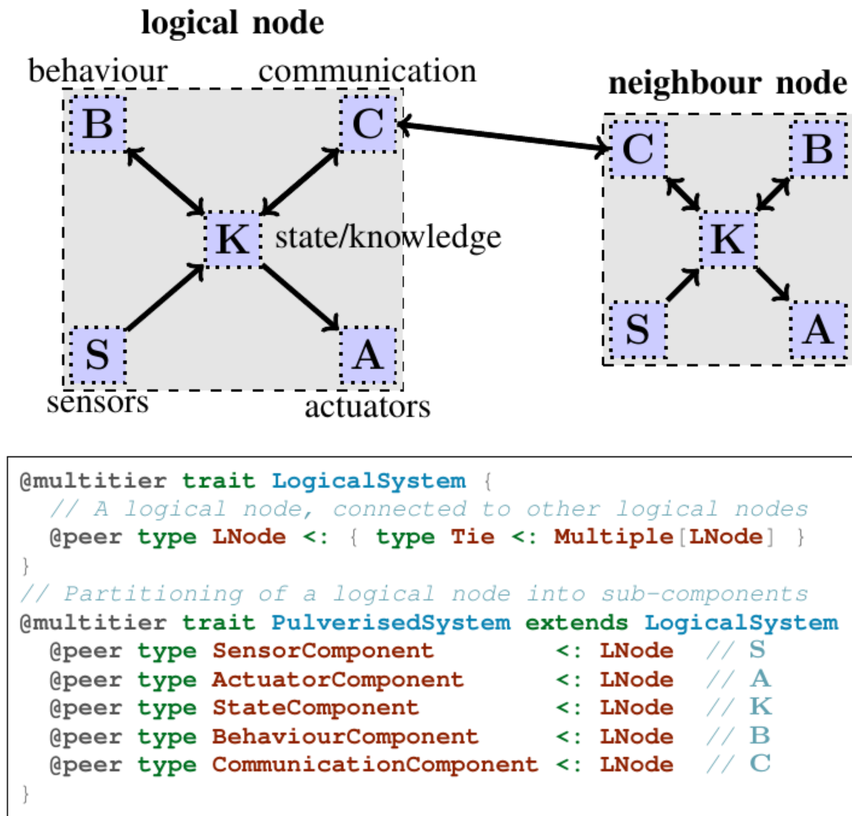

Figure 3.1: Pulverized device in ScalaLoci [10]

A prototype has also been developed to show the possibility of combining aggregate programming and ScalaLoci leveraging the ScaFi framework. Such integration is possible since both languages are written on top of the Scala programming language. As previously illustrated, ScaFi requires the definition of an object that contains the aggregate program (i.e. the aggregate application logic) and a *context* that contains all the information to execute the program like the previous state, the sensors' data and the messages from the neighbors. Since in the pulverized approach this elements are embedded in the State component, it is possible to run the aggregate program leveraging the architecture described in Figure 3.1. The resulting code presented in the paper can be seen in Figure 3.2.

```
def compute(
  deviceIdentifier: Id,
  state: State
): State on BehaviourComponent = {
  val context = new ContextImpl(
    deviceIdentifier,
    export = state.exports,
    localSensor = state.sensors,
    neighbourSensor = state.neighbourSensor
  )
  val program : AggregateProgram = ... // business logic
  // actual execution; returns the new State
  program.round(context)
}
```

Figure 3.2: ScaFi and ScalaLoci integration [10]

This chapter establishes the foundational knowledge upon which this thesis is built. It delves into the pre-existing integration techniques between aggregate programming and multitier programming. Serving as more than just a review, it sets the stage for the novel research and experiments presented later in the thesis. The emphasis here is on the technique of pulverization and its application in aggregate computing and how ScalaLoci can be used as a support for this use case.

The thesis uses to a certain extent the work in this paper regarding the approach of running aggregate programs in ScaFi along with the multitier modules approach to provide a way to specify the deployment of the aggregate program. However, does not consider the pulverization aspects yet, but rather a more general approach of integration between aggregate programming using ScalaLoci and ScaFi. The next chapter will introduce the first steps toward this integration.

Chapter 4

Towards a seamless integration of ScaFi and ScalaLoci

This chapter seeks to offer a comprehensive depiction of the initial phase of the integration between ScaFi and ScalaLoci. The approach uses both libraries side-by-side, in order to evaluate the feasibility of the integration and is based on the work described in the previous chapter.

4.1 Dynamic connections

The goal of the experiment is to show a system that runs several examples of aggregate programs using ScaFi's constructs on a set of nodes connected via ScalaLoci. The system can adapt to changes e.g. a node leaves the system and the computed values change accordingly.

The behavior of the experiment is the following:

- the system is composed of a set of nodes, organized in a peer-to-peer architecture using ScalaLoci's peers and ties;
- every node's neighbors are the nodes that are directly connected to it;
- it is possible to run an aggregate program written in ScaFi on top of ScalaLoci leveraging the Multitier Modules;
- aggregate computing execution model is supported and every phase of the execution is executed on every node of the system by leveraging ScalaLoci's placement types;
- the system captures the dynamicity of the network allowing nodes to join and leave the network;

- messages containing neighbors' exports are exchanged between nodes using ScalaLoci's reactives abstractions.

In Listing 4.4, the code of the integration between ScaFi (lines 5, 6) and ScalaLoci (lines 1, 2, 4, 7) is shown by reporting the main method of the program that runs on every node of the system. The code is a simplified version and omits the creation of the context and the execution of the aggregate program. The latter is defined in `LogicalSystem`.

```
1 @multitier trait Node extends LogicalSystem {
2 @peer type AGNode <: { type Tie <: Multiple[AGNode] }
3
4       def main(): Unit on AGNode = {
5             val ctx = // build context
6             val export = computeLocal(ctx)// execute round
7             remote.call(export)
8       }
9 }
```

Listing 4.1: First integration experiment's main method

Neighbors that receive the node's export are the nodes directly connected with it. It is possible to get the list of connected nodes by using ScalaLoci's `remote[P].connected`, a reactive abstraction that provides a time-changing list of currently connected peer instances of type `P`.

```
1 def updateConnections(nodes: Seq[Remote[AGNode]]): Local[Unit] on
    AGNode = {
2       val remoteNodes = // get remote nodes list
3       val nodesToAdd = nodes diff remoteNodes
4       val nodesToRemove = remoteNodes diff nodes
5
6       nodesToAdd foreach addRemoteNode // nodes that will receive
    node's export
7       nodesToRemove foreach removeExport // remove export of nodes
     that left
8 }
9
10 def main(): Unit on AGNode = {
11       remote[AGNode].connected observe updateConnections
12       // aggregate logics
13 }
```

Listing 4.2: Connections management

4.2 Aggregate base station

This experiment introduces a node that doesn't run the aggregate program (e.g. a base station) that monitors the other nodes and collects the output calculated by each node connected to it.

This time the architecture is multi-client-server, where the base station is the server and the nodes that run the aggregate program are the clients.

The behavior of the base station is implemented using multitier modules, specifically using the concepts of "Monitor" and "Monitored" where the base station acts as a monitor and observes the results computed by the aggregate nodes that act as monitored.

```
1 @multitier trait Monitoring {
2   @peer type Node
3   @peer type Monitor <: Node { type Tie <: Multiple[Monitored] }
4   @peer type Monitored <: Node { type Tie <: Optional[Monitor] }
5
6   def monitorNode(remote: Remote[Monitored], output: EXPORT,
     exports: Map[ID, EXPORT]): Local[Unit] on Monitor = {
7       // collect output
8   }
9 }
```

Listing 4.3: Monitoring multitier module

```
1 @multitier trait AggregateBaseStation extends LogicalSystem with
     Monitoring {
2   @peer type BaseStation <: Monitor { type Tie <: Multiple[AGNode]
      with Multiple[Monitored] }
3   @peer type AGNode <: Monitored {
4     type Tie <: Multiple[AGNode] with Optional[BaseStation] with
     Optional[Monitor]}
5
6     val currentNodeState: Evt[(EXPORT, Map[ID, EXPORT])] on AGNode
      = // ...
7
8     def gatherValues(): Unit on BaseStation = {
9       currentNodeState.asLocalFromAllSeq observe { /* ... */ }
10    }
11
12    def main(): Unit on AGNode = {
13      // connections management
14      // aggregate logics
15    }
16 }
```

Listing 4.4: Monitoring multitier module

4.3 Considerations

The experiments show that the integration between ScaFi and ScalaLoci is feasible and that the two libraries can be used side-by-side. However, the integration is not seamless and it requires a lot of boilerplate code to be written by the programmer.

The next step to be done is to modularize all the aggregate and ScalaLoci constructs and provide a middleware that allows the developer to only write the logic of the aggregate program and chose the architecture of the system (e.g. peer-to-peer or client-server) specifying among which nodes the export exchange should happen.

Chapter 5

Middleware requirements

The previous chapter presented the initial experiments of possible integration between aggregate and multitier programming via ScaFi and ScalaLoci. However, as previously mentioned, the integration can be improved by providing a middleware that allows easily run aggregate programs on top of ScalaLoci. This chapter will present the requirements for such middleware.

5.1 Business requirements

Since the main objective of the middleware is to provide a way to run aggregate programs on top of ScalaLoci using ScaFi, the main business requirements are:

- The middleware can be used to deploy aggregate systems;
- The middleware should be flexible enough to allow different topologies;
- The middleware should be modular and extensible to support new topologies.

5.2 User requirements

The user requirements are identified from the perspective of the developer who will use the middleware. The middleware should allow the programmer to:

- define the aggregate program to be executed;
- configure the sensors;
- configure the actuators;
- use some sort of topology configuration;

- specify their own topology;
- specify the frequency on which the aggregate program should be executed;
- allow to specify which nodes run the aggregate program and which not.

5.3 Functional requirements

The functional requirements, obtained from the user requirements, define what the middleware should be able to do:

- add sensors and actuators to make them available to the aggregate program;
- remove sensors and actuators to not make them available to the aggregate program;
- access data from local sensors and neighbor sensors;
- send data to actuators;
- collect the exports from the neighbors;
- create the context for the aggregate program;
- run the aggregate program on specific nodes of the topology;
- send the exports to the neighbors;
- take track of the neighbors according to a specific neighboring logic;
- given an aggregate program written in ScaFi and a topology written in ScalaLoci, should generate the executable code for every node;
- allow execution on different topologies (e.g. ring, peer-to-peer, client-server);
- have a modular and extensible architecture.

Chapter 6

Design

This chapter discusses the design elements of the middleware by illustrating its components and their interactions. For each module of the middleware, the relevant design choices are discussed. Starting with an overview of the middleware's architectural structure, its components will be progressively broken down, providing a granular understanding of each module's function. The following section, some relevant aspects of the implementation choices will be presented, mostly focusing on the topology module and the communication between nodes aspects.

6.1 Middleware architecture

The middleware is structured into three primary modules: the *configuration*, the *topology*, and the *runtime*. From a developer's perspective, the requirements are minimal. The user is tasked with outlining the aggregate program they wish to execute and specifying the necessary sensors and actuators required for its seamless operation. This abstraction is deliberate, allowing users to focus on core logic without being burdened by underlying complexities.

The middleware comes equipped with predefined topologies. This implies that users don't have to manually design or implement network topologies. Instead, they can rely on these ready-made structures. However, the middleware also defines a generic topology that can be used to define custom topologies if needed.

Once the user's input is processed and integrated with the chosen topology, the middleware takes over the heavy task. Utilizing the ScalaLoci compiler, it translates this amalgamated data into an executable that can be systematically deployed across the nodes of the network.

The basic version of the architecture is depicted in Figure 6.1.

The middleware also allows the user to define a custom topology. In this case, the user needs to specify the nodes involved and their type (aggregate or individual).

The middleware will then deploy the components on the nodes according to the topology. This architecture is depicted in Figure 6.2.

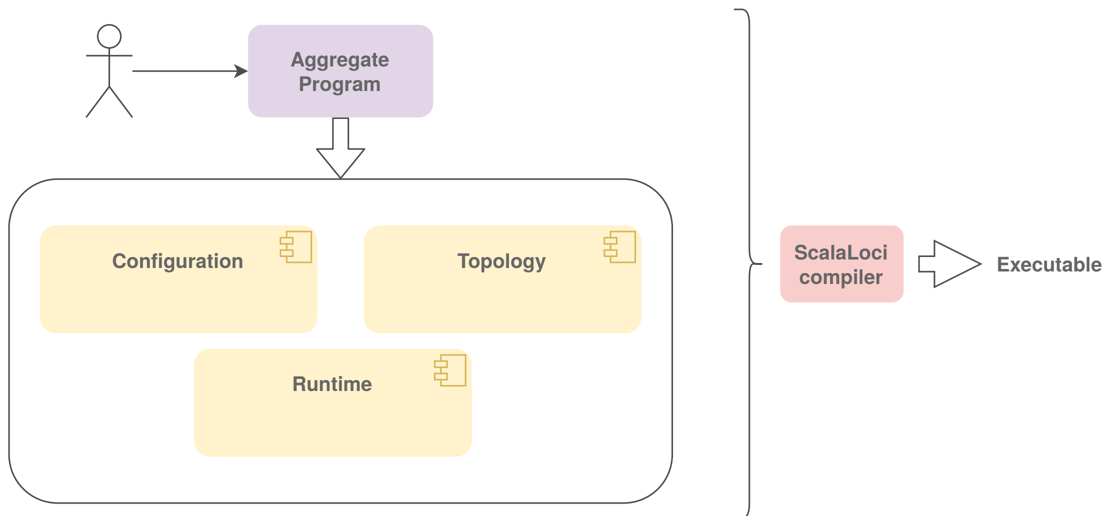

Figure 6.1: Middleware architecture's modules

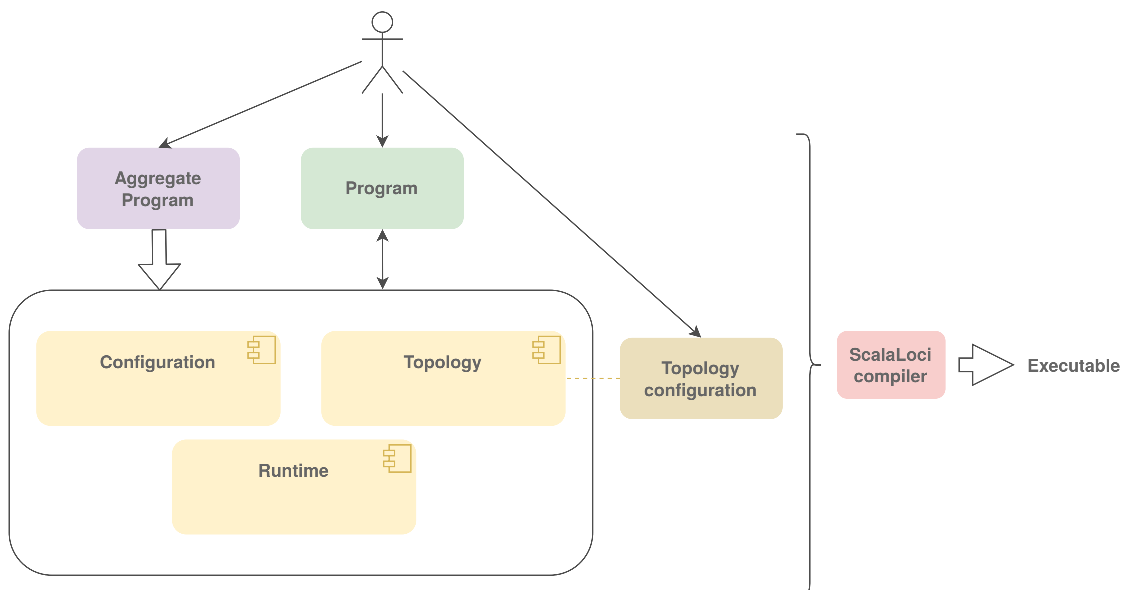

Figure 6.2: Middleware architecture with topology configuration

Runtime module

The runtime module is the core of the middleware. It is responsible for the execution of the aggregate program and contains the communication strategy among the nodes. The runtime module's main component is the `ScafiLociExecutor`, which is a ScalaLoci module. This class has all the logic and stores the data *placed* on an `AGNode` (aggregate node) which is provided, through the Scala mechanism self-type, by the `Architecture` module. The logic manages the execution of the aggregate program using ScaFi constructs. A more detailed architecture of the component can be seen in Figure 6.3.

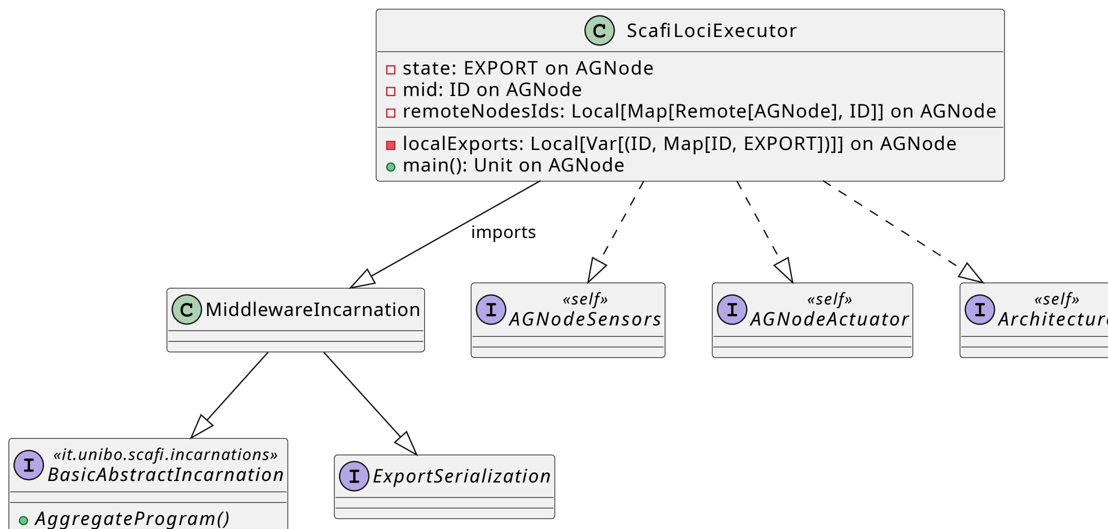

Figure 6.3: Runtime module

`ScafiLociExecutor` imports functionalities from `MiddlewareIncarnation` allowing the use of the aggregate constructs provided by ScaFi, such as `AggregateProgram`, `mid`, `EXPORT` and `ID`.

One thing to note is that the `MiddlewareIncarnation` extends `ExportSerialization`. This is needed because the exports are automatically sent to the neighbors of the node by ScalaLoci over the network.

Finally, the class adapts its behavior based on sensors data and actuation, elements provided by `AGNodeSensors` and `AGNodeActuator` respectively.

Configuration module

`AGNodeSensors` is responsible for the management of local and neighboring sensors' data. `Sensor` is the main interface that represents a generic sensor. It has a method name of type `CNAME` that is a ScaFi element imported from `MiddlewareIncarnation`.

`NbrSensor` and `LocalSensor` are two implementations of the `Sensor` interface. The first one is used to represent a neighboring sensor (e.g. NBR Range) and the `query` method returns the distance of a node from another according to the concrete implementation logic; the second one is used to represent a local sensor (e.g. GPS, temperature, etc.) and the `query` method returns the value of the sensor. It is important to point out that `AGNodeSensors` uses the self-type mechanism to require the `Architecture` module. This is needed to access the `AGNode` type and to allow ScalaLoci's modules composition.

In Figure 6.4 the architecture of the module is depicted.

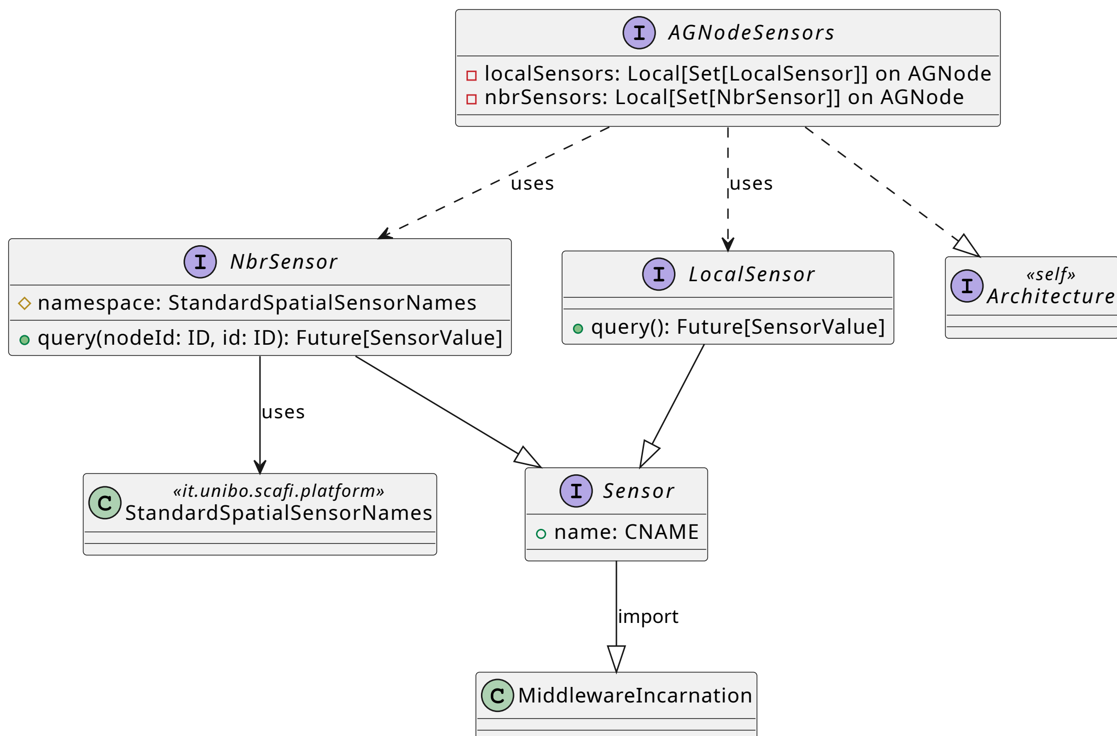

Figure 6.4: Configuration module

6.2 Implementation

This section discusses some implementation details of the middleware regarding the communication between nodes and the topology definition.

6.2.1 Node's communication strategy

The communication strategy between nodes is encapsulated in the runtime module and particularly in `ScafiLociExecutor`. The interaction between nodes leverages the reactive constructs provided by ScalaLoci. In particular, the "export" exchange between neighbors is implemented using a reactive variable `Var` that is placed on the aggregate node. The variable is defined as follows:

```
1     var localExports: Local[Var[(ID, Map[ID, EXPORT])]] on AGNode
```

Leveraging the ScalaLoci's method `remote.call`, after every node's round it is possible to update the variable of every connected node (in this case the neighbors) with the new computed exports as shown in Listing 6.2.1.

```
1     def process(id: ID, export: EXPORT): Unit on AGNode =
2         localExports.transform {
3             case (myId, exports) =>
4                 (myId, exports + (id -> export))
5         }
```

6.2.2 Topology definition with multitier modules

One interesting element of the middleware to discuss is the implementation of the `Topology` component.It is used in ScafiLociExecutor to define the aggregate node on which the aggregate logics is executed. This component (Listing 6.1) is realized with multitier modules that are parametric on peer types, and can be used leveraging Scala's mixin composition. The `Architecture` trait contains the base topology and can be extended by specializing `AGNode` and defining custom types.

```
1 @multitier trait Architecture {
2     @peer type AGNode <: { type Tie <: Multiple[AGNode] }
3 }
```

Listing 6.1: Architecture trait

As stated before, the middleware allows the developer to leverage a predefined topology without having to worry about the deployment of the components. The topology is defined in the *topology* module of the middleware and the possible architectures are depicted in Figure 6.5.

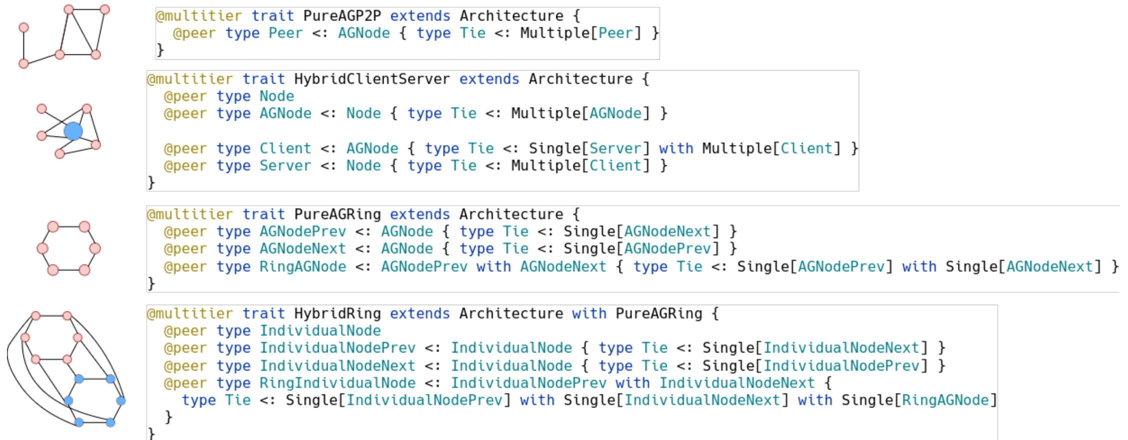

Figure 6.5: Default topologies provided

The user can also define a custom topology. In this case, the middleware provides a generic architecture with `AGNode` (the nodes that will run an aggregate program) and `IndividualNode` (the nodes that will run a different program) and the user has to specify which nodes are of which type. The generic topologies are depicted in Figure 6.6 along with a schematized representation of the network.

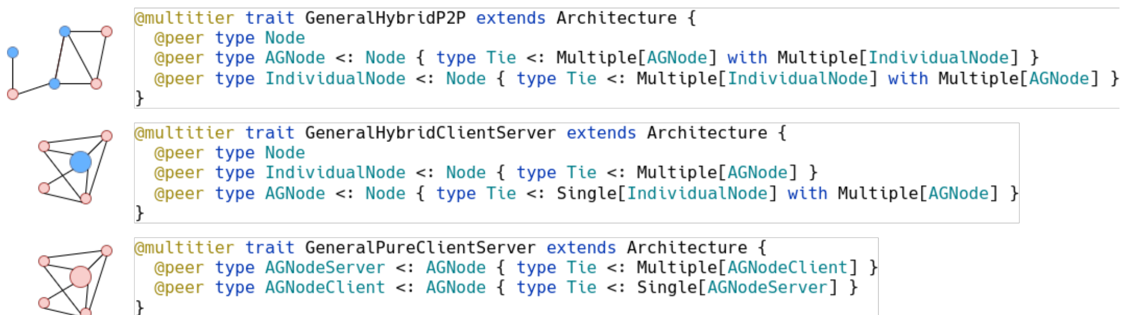

Figure 6.6: General topologies

Chapter 7

Evaluation

Upon the successful development of a middleware that merges the powers of ScalaLoci and ScaFi, it becomes imperative to evaluate its efficacy, correctness, and reliability. This chapter unfolds a series of carefully crafted case studies, shedding light on the system's capabilities, the advantages it brings to the table, and ensuring that aggregate computing inherent properties, such as fault tolerance, remain intact. Additionally, the case studies provide a reference on how it is possible to use the middleware to develop aggregate systems.

The middleware satisfies the requirements presented in Chapter 5:

- Various test scenarios were created to deploy aggregate systems, demonstrating the middleware's capability to run them on top of ScalaLoci
- Multiple options were provided for topology configurations, ensuring flexibility. Developers could also specify custom topologies, allowing for unique deployment scenarios
- The middleware's modular design allows the user to define custom topologies and aggregate programs
- The middleware provides a way to specify the frequency of the aggregate program's execution
- The aggregate program can be executed on specific nodes as specified by the developer
- The middleware has mechanisms to add, remove, and access data from both local and neighbor sensors
- The system collects correctly exports from neighbors and creates the context accordingly

Based on the user and functional requirements, a series of test cases were designed. Each test case aims at a specific requirement or a group of related requirements, ensuring comprehensive coverage.

7.1 Case studies

The criteria for choosing the case studies are based on showing both the abilities of the middleware to manage the neighbors of a node and the capabilities of the aggregate programs to recover from failures. The tested aggregate programs are the following:

- **NeighboursId**: a program that returns the set of the IDs of the node's neighbors
- **AverageTemperature**: the aggregate program that computes the average temperature of the network, considering the value of a local temperature sensor on each node
- **Gradient**: the aggregate program that computes the minimum distances from source nodes. This example uses two kinds of sensors: a local sensor "Source", that returns true on the source nodes, and a neighboring sensor (nbr range) "NbrHopCount", that counts the distance from a source node by calculating the number of hops from it.

The topologies used are the following:

- **General Hybrid Multi-Client-Server**: a hybrid topology where the nodes are divided into two groups: clients and servers. The clients are the AGNodes (i.e. execute an aggregate program) and are connected with other AGNodes as well as to a server. The server is an IndividualNode and executes a different program. The topology is defined by extending the `GeneralHybridClientServer` trait.
- **General Pure Multi-Client-Server**: a pure topology similar to the previous one where each node is an AGNode (the server as well). The topology is defined by extending the `GeneralPureClientServer` trait
- **General Hybrid Peer to Peer**: a hybrid peer-to-peer topology where each node can be either an AGNode or an IndividualNode. The aggregate nodes execute an aggregate Program. The topology is defined by extending the `GeneralHybridP2P` trait

The topologies to extend are the ones presented in Section 6.2. The following code snippets show how the topologies are extended to define the nodes involved in the case studies.

```
1  @multitier trait MyHybridClientServer extends
   GeneralHybridClientServer {
2      @peer type Client <: AGNode
3      @peer type Server <: IndividualNode
4    }
```

Listing 7.1: General hybrid multi-client-server topology

```
1  @multitier trait MyPureClientServer extends
   GeneralPureClientServer {
2    @peer type Client <: AGNodeClient
3    @peer type Server <: AGNodeServer
4  }
```

Listing 7.2: General pure multi-client-server topology

```
1  @multitier trait MyHybridP2P extends GeneralHybridP2P {
2      @peer type Type1Peer <: AGNode
3      @peer type Type2Peer <: IndividualNode
4    }
```

Listing 7.3: General hybrid peer-to-peer topology

Table 7.2 presents an overview of the different case studies that were developed to evaluate the middleware. Each case study is crafted to simulate real-world scenarios. The table delineates each case study emphasizing the specific topologies employed and the aggregate program executed.

Aggregate Program	**Topology**
AverageTemperature	Hybrid Multi-Client-Server
AverageTemperature	Pure Peer to Peer
Gradient	Pure Peer to Peer
NeighboursId	Pure Ring
NeighboursId	General Hybrid Multi-Client-Server
NeighboursId	General Pure Multi-Client-Server
AverageTemperature	General Hybrid Peer to Peer

Table 7.2: Case studies configurations

An example of the code needed to run one of the case studies is shown in Listing 7.4. The code describes a multitier application named MySystem. This ap-

plication leverages our previously introduced ScafiLociExecutor class, that uses the AverageTemperature aggregate program. Additionally, the application is provided with sensors (AGNodeSensors) and actuator (MyActuator) modules, alongside the dedicated topology defined in MyClientServerHybrid.

```
1      @multitier object MySystem
2          extends ScafiLociExecutor(new AverageTemperature())
3          with AGNodeSensors
4          with MyActuator
5          with MyP2P {
6              def main(): Unit on Node = {
7                  on[Type1Peer] {
8                      println("Hello AGNode")
9                      runAggreteProgram()
10                 } and on[Type2Peer] {
11                     println("Hello Individual Node")
12                     runIndividualProgram()
13                 }
14             }
15 }
```

Listing 7.4: Case study example code

7.2 Preservation of fault tolerance

One of the cornerstones of aggregate computing is fault tolerance. It is imperative that despite the new integrations and functionalities introduced by the middleware, this essential property remains unaffected. A random set of nodes in the network was made unresponsive to test the middleware's ability to correctly rearranges nodes' neighborhood. The middleware ensures that aggregate programs continue to run correctly on the remaining active nodes that eventually compute the correct updated values.

7.3 Scafi-Loci middleware on Android devices

ScalaLoci, with its support for Scala.js, offers a unique advantage when it comes to creating cross-platform applications. Recognizing this potential, an exploration was initiated to delve deeper into the feasibility of deploying the middleware on Android devices.

For the experiments, it was used Scala.js, a compiler that translates Scala code to JavaScript, to facilitate running of the middleware on Android. The advantage here is twofold. Firstly, the ubiquity of JavaScript ensures that the code is portable across various platforms, including mobile browsers and hybrid mobile application

frameworks. Secondly, with Scala.js, developers can maintain a single codebase in Scala, which can then be targeted to both JVM-based backends and JavaScript frontends, ensuring code consistency and reduced development effort. As mentioned, ScalaLoci is already compatible with Scala.js and to compile the middleware to Javascript it was sufficient to add the lines of code in Listing 7.5 and to configure the build tool (sbt) accordingly, as well as choose the communication protocol to use. In this example, the communication protocol used is WebSocket.

```
1      object BaseStation extends App {
2          platform(platform.jvm) {
3            val port = 8080
4            val server = new Server()
5            // server configuration
6            multitier start new Instance[HybridSystem.Server](
7              listen[HybridSystem.Client] {
8                jetty.WS(context, "/ws/*")
9              }
10           )
11         // server start
12         }
13       }
14
15     object AGNodeBS1 {
16         @JSExportTopLevel("main")
17         def main(args: Array[String]): Unit = {
18             platform(platform.js) {
19             multitier start new Instance[HybridSystem.Client](
20             connect[HybridSystem.Server] {
21                 webnative.WS("ws://10.0.2.2:8080/ws/")
22             })
23           }
24   }
25 }
```

Listing 7.5: ScalaLoci code to enable the compilation of Scala both to JVM bytecode and to Javascript

This way, the code written on `HybridSystem.Client` (i.e. the aggregate node in this example) can be directly run on Android and the communication between the server and the client is made through WebSocket.

However, it is important to note that a proper serialization of the exports is required since all the libraries used must be compatible with Scala.js.

7.4 Middleware limitations

At the time of writing, the middleware has some limitations that prevent it from being used in a production environment. The goal of this section is therefore to provide an overview of the main shortcomings of the middleware.

7.4.1 Neighboring logic

In the current design of the middleware, the logic for determining a node's neighbors is primarily based on connectivity — nodes that are directly connected are considered neighbors. This simplistic approach, while effective for many use cases, might not cater to the diverse needs of certain applications or scenarios. For instance, in a dense network, just because two devices are connected doesn't necessarily mean they should be interacting as neighbors. The real-world context might require a different logic. Devices could be deemed neighbors based on their geographical distance, as given by GPS coordinates. Similarly, Bluetooth strength could be a determinant of how close two devices are, with only those within a certain strength threshold treated as neighbors. Such an approach would be beneficial in applications like location-based services, disaster response systems, etc. where the physical location and proximity of nodes matter more than just network connectivity.

7.5 Different nodes interaction

The middleware currently supports the interaction between aggregate and individual nodes by accessing the reactive variable exposed by the middleware. However, there could be more efficient ways to enable this interaction.

7.6 Middleware and pulverization

As mentioned in the state-of-the-art chapter, some works have been done to pair the pulverization concepts with ScalaLoci. However, this thesis focus doesn't address this particular aspect of aggregate computing. Nevertheless, the middleware could be extendend to support pulverization concepts by allowing to execute ScafiLociExecutor components on different nodes.

7.6.1 Scalability concerns

The middleware has been tested for a defined set of scenarios. However, as the number of nodes or the complexity of interactions increases, there might be more

challenges arising. Ensuring scalability as the network grows remains an area of concern.

7.6.2 Limited Error Handling

The middleware currently only relies on the aggregate and ScalaLoci constructs to handle errors. This could be not enough to ensure a robust error-handling mechanism.

7.6.3 Performance evaluation

The performance of a middleware, especially one designed for distributed systems, is an essential factor in determining its efficacy and reliability. While the initial development and deployment phases have focused on functionality and compatibility, the absence of a comprehensive performance evaluation leaves potential bottlenecks and inefficiencies unaddressed.

Chapter 8

Conclusion

Digital evolution has made distributed systems vital in today's computing landscape. The transition from standalone systems to complex networks of devices presents both opportunities and challenges. The rise of pervasive computing, Internet of Things (IoT), and cyber-physical systems intensifies the role of distributed systems. Traditional Cloud computing faces limits and Edge Computing can help bridge the gap between data generation and data processing. By computing closer to data sources, it offers reduced latency and more context-sensitive operations. Merging cloud and edge, harmonizes centralized and decentralized processing, crafting a comprehensive strategy for distributed systems.

Aggregate programming and multitier programming represent two advanced paradigms in the field of software development. Aggregate programming emphasizes a collective approach where the focus isn't on a single device but on a coordinated ensemble of devices. Instead of treating each device as an individual unit, this model views the collective as the primary computing entity, leading to solutions that are inherently decentralized and robust against individual device failures. On the other hand, multitier programming is concerned with programming different components or "tiers", such as the client and server layers, in a single compilation unit. Multitier programming facilitates programming distributed systems as well as their deployment. This way allows targeting specific devices for the creation of complex distributed applications. Together, these programming paradigms offer a comprehensive toolkit for addressing the multifaceted challenges of modern distributed system design.

This thesis embarked on a journey, aiming to explore and bridge the unique attributes of aggregate and multitier programming. Because of the growing need for distributed systems that can handle the intricacies of coordination among a multitude of devices, the middleware elaborated in this thesis showcases a solution that leverages the synergy between ScalaLoci and ScaFi. The middleware proves the possibility of executing aggregate programs across varied architectures—whether

Cloud, Edge, or a hybrid combination.

The executed case studies underscore the middleware's potential. They not only demonstrate its adaptability across varying architectures but also validate its utility in scenarios with simultaneous requirements of both aggregate and "individual" programming.

However, there is most certainly room for improvements and feature expansion. The middleware, in its current form, encapsulates vast potential, but there are clear avenues for further optimization and expansion, particularly concerning enhanced node interactions, extended topology configurations, and comprehensive support for platforms like Android. Additionally, performance assessment remains a critical facet warranting further exploration.

8.1 Future work

The middleware, in its current form, encapsulates vast potential, but there are clear avenues for further expansion to make it more robust and complete. The following are some of the possible aspects that can be addressed in future work.

- A prominent limitation identified in the middleware is the simplistic approach toward determining a node's neighbors, primarily based on direct connectivity. As outlined in the Middleware Limitations (Section 7.4), this method may not be optimal for all scenarios. Future research can focus on incorporating geographical data, like GPS coordinates, utilizing Bluetooth signal strength as a metric, etc.

- The middleware currently supports a limited set of topologies. Future work can focus on expanding the topologies to include more complex structures. Also, the provided hybrid architectures assume that every node is connected to the base station. This assumption can be relaxed allowing for different configurations.

- The absence of pulverization concepts in the integration with ScalaLoci, could be addressed in the future as there are potential areas of exploration. One of them could be, for example, facilitating the ScafiLociExecutor's components to operate on diverse nodes.

- Finally, it would be useful to provide the developer with a DSL, to easily configure every aspect of the middleware in a declarative way.

Bibliography

[1] Jacob Beal and Mirko Viroli. “Aggregate Programming: From Foundations to Applications”. In: *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems: 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016, Advanced Lectures*. Ed. by Marco Bernardo, Rocco De Nicola, and Jane Hillston. Cham: Springer International Publishing, 2016, pp. 233–260. ISBN: 978-3-319-34096-8. DOI: 10.1007/978-3-319-34096-8_8. URL: https://doi.org/10.1007/978-3-319-34096-8_8.

[2] 2023. URL: https://en.wikipedia.org/wiki/Multitier_programming.

[3] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. “Distributed System Development with ScalaLoci”. In: 2.OOPSLA (2018). DOI: 10.1145/3276499. URL: https://doi.org/10.1145/3276499.

[4] Roberto Casadei et al. “ScaFi: A Scala DSL and Toolkit for Aggregate Programming”. In: *SoftwareX* 20 (2022), p. 101248. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2022.101248. URL: https://www.sciencedirect.com/science/article/pii/S2352711022001662.

[5] Giorgio Audrito et al. “A Higher-Order Calculus of Computational Fields”. In: *ACM Trans. Comput. Logic* 20.1 (2019). ISSN: 1529-3785. DOI: 10.1145/3285956. URL: https://doi.org/10.1145/3285956.

[6] 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.

[7] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. “A Survey of Multitier Programming”. In: *ACM Comput. Surv.* 53.4 (2020). ISSN: 0360-0300. DOI: 10.1145/3397495. URL: https://doi.org/10.1145/3397495.

[8] Pascal Weisenburger and Guido Salvaneschi. “Multitier Modules”. In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer

Informatik, 2019, 3:1–3:29. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.3. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10795.

[9] Roberto Casadei et al. "Pulverization in Cyber-Physical Systems: Engineering the Self-Organizing Logic Separated from Deployment". In: *Future Internet* 12.11 (2020). ISSN: 1999-5903. DOI: 10.3390/fi12110203. URL: https://www.mdpi.com/1999-5903/12/11/203.

[10] Gianluca Aguzzi et al. "Towards Pulverised Architectures for Collective Adaptive Systems through Multi-Tier Programming". In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. 2021, pp. 99–104. DOI: 10.1109/ACSOS-C52956.2021.00033.

[11] Pascal Weisenburger and Guido Salvaneschi. "Developing Distributed Systems with Multitier Programming". In: *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*. DEBS '19. Darmstadt, Germany: Association for Computing Machinery, 2019, 203–204. ISBN: 9781450367943. DOI: 10.1145/3328905.3332465. URL: https://doi.org/10.1145/3328905.3332465.

[12] Saverio Giallorenzo et al. "Multiparty Languages: The Choreographic and Multitier Cases". In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Ed. by Anders Møller and Manu Sridharan. Vol. 194. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 22:1–22:27. ISBN: 978-3-95977-190-0. DOI: 10.4230/LIPIcs.ECOOP.2021.22. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14065.

[13] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. "Type-Safe Dynamic Placement with First-Class Placed Values". In: vol. 32. 2023. DOI: https://doi.org/10.1145/3622873.

[14] Pascal Weisenburger and Guido Salvaneschi. "Implementing a Language for Distributed Systems: Choices and Experiences with Type Level and Macro Programming in Scala". In: *CoRR* abs/2002.06184 (2020). arXiv: 2002.06184. URL: https://arxiv.org/abs/2002.06184.

[15] Giorgio Audrito et al. "Functional Programming for Distributed Systems with XC". In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 20:1–20:28. ISBN: 978-3-95977-225-9. DOI: 10.4230/LIPIcs.ECOOP.2022.20. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16248.

[16] Roberto Casadei et al. "Engineering collective intelligence at the edge with aggregate processes". In: *Engineering Applications of Artificial Intelligence* 97 (2021), p. 104081. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2020.104081. URL: https://www.sciencedirect.com/science/article/pii/S0952197620303389.

[17] Roberto Casadei. "Macroprogramming: Concepts, State of the Art, and Opportunities of Macroscopic Behaviour Modelling". In: *CoRR* abs/2201.03473 (2022). arXiv: 2201.03473. URL: https://arxiv.org/abs/2201.03473.

[18] Mirko Viroli, Roberto Casadei, and Danilo Pianini. "On Execution Platforms for Large-Scale Aggregate Computing". In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. UbiComp '16. Heidelberg, Germany: Association for Computing Machinery, 2016, 1321–1326. ISBN: 9781450344623. DOI: 10.1145/2968219.2979129. URL: https://doi.org/10.1145/2968219.2979129.

[19] Peri Tarr et al. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE '99. Los Angeles, California, USA: Association for Computing Machinery, 1999, 107–119. ISBN: 1581130740. DOI: 10.1145/302405.302457. URL: https://doi.org/10.1145/302405.302457.

[20] Roberto Casadei, Gianluca Aguzzi, and Mirko Viroli. "A Programming Approach to Collective Autonomy". In: *Journal of Sensor and Actuator Networks* 10.2 (2021). ISSN: 2224-2708. DOI: 10.3390/jsan10020027. URL: https://www.mdpi.com/2224-2708/10/2/27.

[21] Roberto Casadei and Mirko Viroli. "Coordinating Computation at the Edge: a Decentralized, Self-Organizing, Spatial Approach". In: *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*. 2019, pp. 60–67. DOI: 10.1109/FMEC.2019.8795355.