ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA
Campus di Cesena

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

Distributing Aggregate Computations on top of Akka Actors

Tesi in
Pervasive Computing

Relatore:
Prof. MIRKO VIROLI

Co-Relatore:
Dott. ROBERTO CASADEI

Presentata da:
MANUEL PERUZZI

Anno Accademico 2017-2018
Sessione II

Contents

Abstract (italiano)

Nel contesto dell'Internet of Things, lo sviluppo di sistemi adattivi, su larga scala, solitamente si concentra sul comportamento del singolo dispositivo. Aggregate programming è un paradigma che fornisce un approccio alternativo, nel quale l'unità di base della computazione è un insieme di dispositivi che cooperano tra loro, invece del singolo dispositivo. `scafi` è un framework basato su Scala, e fornisce una piattaforma basata su Akka per lo sviluppo di applicazioni aggregate, supportando sia reti peer-to-peer che server-based. Inoltre, `scafi` mette a disposizione un modulo simulator per simulare un sistema aggregato.

Il lavoro descritto in questa tesi consiste nell'analisi di `scafi`, nella parziale reingegnerizzazione della sua piattaforma ad attori, e nello sviluppo di nuove feature. L'obiettivo principale è incrementare la flessibilità di `scafi` in un contesto distribuito, favorendo la sua adozione per programmare spatial systems. Per prima cosa, è stata resa possibile la comunicazione tra nodi distribuiti, attraverso la definizione di una strategia di serializzazione basata su JSON, che favorisce interoperabilità. Inoltre, vi è l'introduzione di una piattaforma, che mette in atto una comunicazione peer-to-peer tra i dispositivi, con un'unità centrale che gestisce tutte le informazioni rilevanti relative alla distribuzione spaziale. Questa piattaforma colma la principale mancanza dell'approccio peer-to-peer in un ambiente distribuito: il tracciamento di dispositivi remoti. Inoltre, è implementato un approccio di code mobility, in modo da permettere l'assegnamento di nuovi programmi ai dispositivi, a tempo di esecuzione. Infine, è emerso il concetto di monitoraggio di un sistema aggregato distribuito, portando allo sviluppo di un'interfaccia grafica, che permette di osservare i dispositivi di un sistema in esecuzione.

In questa tesi, presento la nuova architettura e API della piattaforma ad attori di `scafi`, progettata con lo scopo di garantire un approccio più flessibile per lo sviluppo di applicazioni distribuite con aggregate computing.

Parole chiave – *aggregate programming, sistemi distribuiti, attori, Akka, Scala, scafi*

Abstract

In the context of the Internet of Things, development of large-scale, adaptive systems usually focuses on the behavior of the single device. Aggregate programming is a paradigm that provides an alternative approach, in which the basic unit of computing is a cooperating collection of devices, instead of a single device. `scafi` is a Scala framework for aggregate programming, and provides an Akka-based platform for aggregate applications, supporting both peer-to-peer and server-based networks. Moreover, `scafi` offers a simulator module for the simulation of an aggregate system.

The work described in this thesis consists in the analysis of `scafi`, in the partial re-engineering of its internal actor platform, and in the development of new features. The main goal is to enhance the flexibility of `scafi` in a distributed context, promoting its adoption for programming spatial systems. First of all, communication between distributed nodes is enabled, by defining a JSON-based serialization strategy, which promotes interoperability. A hybrid platform is also introduced, exploiting a peer-to-peer communication between devices, with a central unit that manages all the relevant space-related information. This platform fills the main gap of the peer-to-peer approach in a distributed environment: tracking of remote devices. Moreover, a code mobility approach is implemented, allowing the assignment of new programs to devices, at runtime. Lastly, the concept of monitoring a distributed aggregate system emerged, leading to the development of a graphical user interface, observing the devices in a running system.

In this thesis, I present the new architecture and API of the actor platform of `scafi`, designed with the aim of ensure a more flexible approach for the development of distributed applications with aggregate computing.

Keywords – *aggregate programming, distributed systems, actors, Akka, Scala, scafi*

Introduction

Pervasive computing is the growing trend of embedding computational capability into everyday objects to make them effectively communicate and perform tasks of various kind, minimizing the user's need to interact with computers as computers. Programming these large-scale situated systems is complex, since devices are heterogeneous, and they often demand for proximity-based interactions with neighboring devices. Traditional approaches address development of these applications focusing on the behavior of the single device, deriving the global behavior of the system from the interaction of devices. Programming interaction of complex systems involves management of robust coordination, efficient and reliable communication, fault-tolerance techniques, and so on.

Aggregate programming is a paradigm that addresses the development of large-scale adaptive systems focusing on the global behavior of the system. With this approach, the basic unit of computation is a cooperating collection of devices. A layered and compositional approach, based on field calculus, allows to provide developers with a simple programming API, that provides robust and adaptive coordination, guaranteeing resilience and safety.

In [1], an aggregate programming framework, `scafi`, has been developed in Scala, providing an implementation of the field calculus semantics and an Akka-based distributed platform for simulating and executing aggregate applications. In order to support development of systems based on different architectural models, `scafi` offers the possibility of relying on a peer-to-peer platform, or on a server-based platform.

Distribution and interaction are key concepts in `scafi`; both of them must be handled at platform-level, in order to provide the developer with high-level API.

This thesis addresses these topics and their implementation in the framework, and it aims to propose interventions that allow to enhance the flexibility of `scafi` in a distributed context, promoting its adoption for programming spatial systems. In particular, the main interventions concern:

- implementation of a JSON-based serialization strategy, to enable communication between different nodes with a standard format;
- introduction of an actor-based hybrid platform as a combination of the pre-existing platforms (p2p and server-based), to fill the main gap of the p2p approach in a distributed environment: tracking of remote devices;
- support for a form of code mobility, allowing the assignment of new programs to devices at runtime;
- monitoring of the actor-based platform, in order to make a distributed aggregate system observable during its execution.

This thesis is organized into six chapters. In chapter 1, an overview of the key elements of the aggregate programming approach is provided. The focus is placed on the aggregate programming stack based on field calculus and resilient API, in order to examine the multi-layer construction that allows implementation of aggregated applications in the `scafi` framework. In chapter 2, the requirements of this thesis are listed and analyzed, in order to present a detailed description of the needs that the design aims to satisfy. In chapter 3, the architecture of the actor-based platform of `scafi` is illustrated, along with the design of its key elements, highlighting the contribution introduced with this thesis. In chapter 4, the implementation of the most relevant aspects is exposed. In chapter 5, exploiting the presentation of some demonstrative programs, an evaluation process is carried out to verify the achievement of the initial requirements. In chapter 6, final considerations are expressed, along with a presentation of future perspectives.

Chapter 1

Background

This chapter provides a review of the studies carried out in the field of aggregate programming, along with the presentation of `scafi`, an aggregate computing framework on top of the Scala programming language.

Outline:

- Introduction to the aggregate programming paradigm
- Description of the aggregate programming stack
- Key elements of `scafi`

1.1 Aggregate programming

Aggregate programming is an emergent paradigm, where the programmable entity is the aggregate collection of devices that cooperatively carry out a computational process. This paradigm presents a top-down approach, based on the specification of the global system behavior, instead of focusing on the design of the single device (bottom-up approach). In this way, it represents a shift in programming of distributed systems, from the traditional device-centric viewpoint to an aggregate viewpoint.

- Device-centric viewpoint: the developer focuses on the design of structure and local behavior of the single device. The global behavior of the system is derived from the interaction between the device and the other elements of the system.

- Aggregate viewpoint: the developer focuses on the global behavior and patterns of the overall system. The programmable entity is the entire set of devices that make up the system. Local behaviors of individual devices are deduced from the high-level global specifications.

Adopting an aggregate viewpoint, a global-to-local mapping is necessary to automatically translate aggregate application specifications into micro-level computations of single devices.

In [2], the main principles of aggregate programming are presented:

- programming a region of the computational environment, not a single device; abstracting away from the underlying specific details;
- the program is specified as manipulation of data constructs with spatial and temporal extent across the region;
- programs are executed by individual devices in the region, exploiting resilient coordination mechanisms and proximity-based interaction.

The application of these principles allows programmers to focus on the specification of the actual program, since mechanisms for robust coordination are hidden "under the hood". The aggregate computing approach is also compositional, so that coarse-grained services can be built as simple and safe combination of smaller functional blocks.

Raising the abstraction level for the development of distributed systems is a key element in aggregate programming. Developers require user-friendly APIs and guarantees of resilience and safety, in order to address programming of complex distributed application in the IoT field. To address this needs, the aggregate computing stack is built as a multi-layer structure, as shown in Figure 1.1.

In [2], the aggregate programming abstraction layers are defined and exploited for engineering large-scale, opportunistic crowd safety services, demonstrating their composability, resilience, and adaptivity.

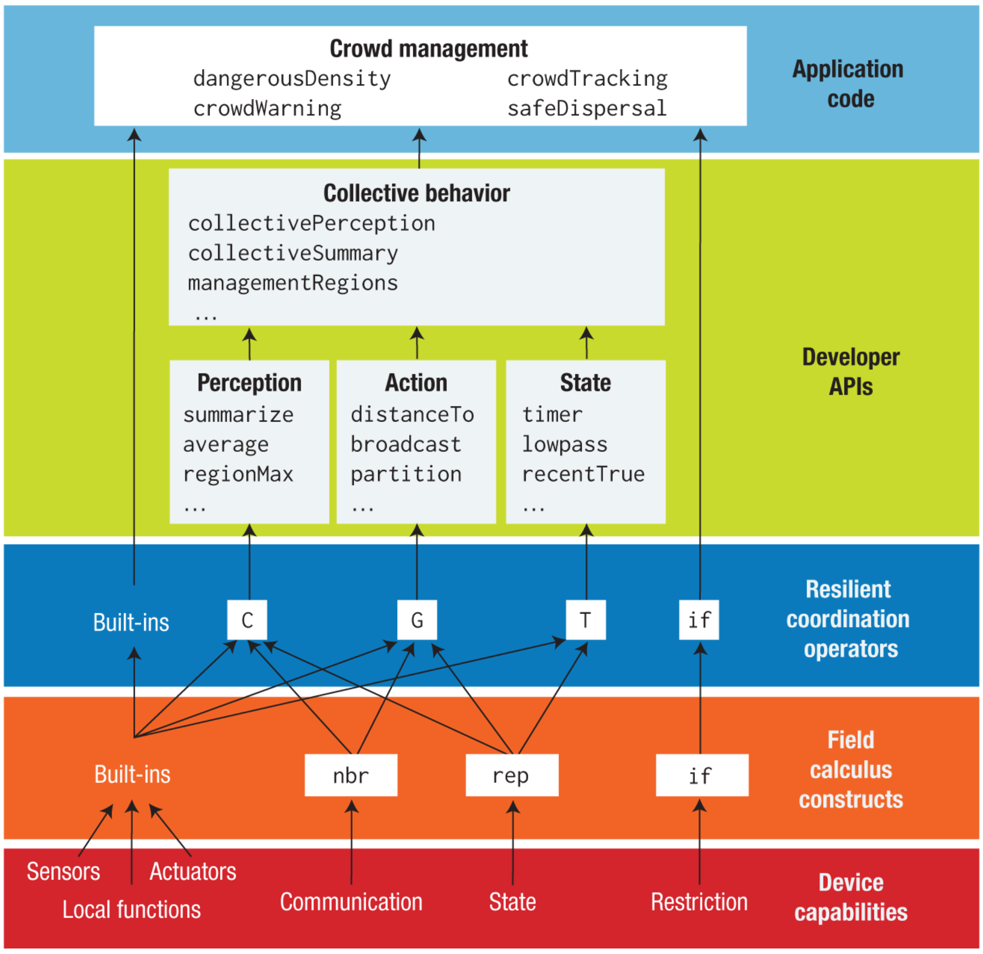

Figure 1.1: Aggregate programming stack (*from [2]*).

In the following subsections, the middle layers of the aggregate programming stack (field calculus constructs, resilient coordination operators, developer APIs), are briefly described, in order to illustrate the way in which device capabilities are modeled to provide an appropriate abstraction level to the programmer in writing the application code.

1.1.1 Field calculus

Field calculus is a minimal universal language for expressing computations based on fields.

Computational fields

The field is the unifying abstraction of the field calculus, inspired by the concept of force field in physics; it's a dynamically evolving function that maps every computational device to a structured value. In field calculus everything is a field (e.g. a collection of temperature sensors produces a field of ambient temperatures).

The behavior of aggregate systems can be expressed as a functional composition of operators that manipulate (evolve, combine, restrict) computational fields [3]. Operators are functions that take input fields and returns output fields.

Field calculus constructs

The field calculus constructs for building and manipulating fields are:

- *Built-in operators* – $b(e_1, \dots e_n)$
  The built-in function b is applied to the input fields $e_1 \dots e_n$. The output field is obtained by the point-wise evaluation of the operator to the input fields (each device applies the operator to its local values of the input fields). The built-in functions are stateless mathematical, logical or algorithmic functions, sensors or actuators, or user-defined or imported library methods.
- *Function definition* – $\text{def}\ \ f(x_1, \dots x_n)\ \{e_B\}$
  The function f is declared with the arguments $x_1 \dots x_n$ and expression e_B as body.
  Function call – $f(e_1, \dots e_n)$
  The function f is applied to inputs $e_1 \dots e_n$. The provided inputs are substituted to the function parameters $x_1 \dots x_n$ in the body expression e_B.
- *Time evolution* – $rep(x \leftarrow v)\ \{s_1; \dots ; s_n\}$
  The rep construct defines a local variable x, initialized with value v, and periodically updated with the result obtained by computing its statements $s_1;\ \dots ; s_n$ against the prior value of x. The output is a dynamically evolving field.
- *Interaction* – $nbr(s)$
  The nbr construct maps each device to a field that consists in the most recent

values of its neighbors, obtained from the evaluation of s. The output is a field of fields.

- *Domain restriction* – $if(e)\,\{s_1;\,...\,;s_n\}\ \ else\ \ \{s'_1;\,...\,;s'_n\}$
  The if construct partitions the network into two subsets: $s_1;\,...\,;s_n$ is computed only in the devices where the condition e is true, while $s'_1;\,...\,;s'_n$ is computed where e evaluates to false.

Higher-order field calculus

In [4], the higher-order field calculus (HFC) is introduced, as an extension of the field calculus with embedded first-class functions. HFC provides the following benefits:

- support for higher-order functions, which are functions that take one or more functions as arguments, and/or return a function as result;
- support for anonymous functions (also known as lambdas), as functions created "on the fly";
- using the `nbr` construct, functions can be moved between devices;
- the executed functions can change over time, via the `rep` construct.

In summary, in HFC, built-in and user-defined functions are naturally handled like any other value, allowing for both code mobility and self-organization.

1.1.2 Building block operators

Field calculus provides basic constructs for manipulation of computational fields. In [5], a set of building blocks is presented, in the form of a library of functions defined on top of the field calculus operators. Building blocks can be combined to create advanced applications involving aggregate programming of collective systems. Follows a brief description of each one of the introduced building blocks.

- *Gradient-cast* – `G(source, init, metric, accumulate)`
  The `G` operator spreads information across space, covering two of the most common distributed algorithms: distance estimation and broadcast.

- *Converge-cast* – `C(potential, accumulate, local, null)`
  The `C` operator collects information from across space. It's complementary to the `G` operator; and it accumulates values up to a potential field.
- *Time-decay* – `T(initial, decay)`
  The `T` operator summarizes information across time. Basically, an initial field is strictly decreased toward zero, according to a decay function.
- *Sparse-choice* – `S(grain, metric)`
  The `S` operator is useful for creating partitions and for selecting subsets of devices.

These operators are extremely general, so each one of them can provide a wide range of different useful services. Building blocks can also be combined, in order to cover many of the most common coordination patterns used in distributed systems.

Moreover, all the listed building blocks are self-stabilizing (it's necessary to point out that self-stabilization of `S` is a particular case); consequently all distributed systems constructed of building blocks will be guaranteed to be self-stabilizing (further information on this topic can be retrieved in [5]). Self-stabilization is a property of an algorithm or a system such that, beginning from any arbitrary state, it will end up to a correct state within finite time. If the inputs of an algorithm stop changing, then its output will self-stabilize and stop changing. An algorithm that takes the output of a self-stabilizing algorithm as input, will have an input that stops changing, and its output, in turn, will self-stabilize.

1.1.3 Developer APIs

As shown in Figure 1.1, the second layer from the top provides a user-friendly API, built using building-block operators, which serves as a support in writing the application code. In this layer, some functions are represented, each one based on one or more building-block operators.

For example, information diffusion functions are typically based on `G`. One common pattern is broadcast of a value from a source; using `nbrRange` as a metric of estimated device-to-device distance, it can be implemented:

```
def broadcast(source, value) {
  G(source, value, () -> {nbrRange}, (v) -> {v})
}
```

These developer APIs are resilient and safely composable, since they rely on resilient operators and field calculus constructs. They can, in turn, be combined, in order to raise the abstraction level and ease the development of applications for IoT scenarios.

1.2 scafi

`scafi`[1] (Scala with computational fields) is an aggregate computing framework on top of the Scala[2] programming language. The purpose of `scafi` is to provide an integrated environment for programming aggregate systems, exploiting the advantages of the Scala programming language, from its powerful static type system to its high-level support for library development.

`scafi` implements a language for field calculus, embedded within Scala, as an internal domain-specific language (DSL), and provides a platform and API for simulating and executing aggregate applications. Moreover, `scafi` comes with a simulator that allows for aggregate computations to be executed and controlled locally.

Follows a brief description of the key elements of `scafi`, based on the information retrieved from [1, 6, 7].

1.2.1 Project organization

The project is divided into separate modules.

- `core`: includes the specifications of the field calculus language, along with a virtual machine (VM) for its execution.
- `tests`: unit and acceptance tests for the `scafi-core` module.

[1]https://github.com/scafi/scafi
[2]https://www.scala-lang.org

- `spala`: definition of the distributed platform and implementation of an actor-based platform.
- `distributed`: realization of predefined platform embodiments.
- `simulator`: implementation of a simulator for simulating aggregate systems.
- `simulator-gui`: definition of a view to show and control a simulation in progress.
- `demos`: provides examples and demonstration programs for both simulation and execution.
- `commons`: provides some useful definitions, including abstractions for modeling space and time.
- `stdlib`: a standard library that provides useful operators (like building blocks).

1.2.2 Field calculus DSL and VM

`scafi` provides an internal domain-specific language (embedded in Scala), with the following characteristics:

- The language implements a variant of the higher-order field calculus, presented in section 1.1.1, supporting distributed first-class functions.
- The language is typed and rely on the Scala typing. Each operation resolves to generic method invocation.
- The language supports all the field calculus constructs listed in section 1.1.1.
- The language is enriched with builtins, as derivate operators built on top of the primitive field calculus constructs.
- The language is concise, easy-to-use, and modular.

Programs written in the `scafi` DSL are executed by a virtual machine (VM), implementing the field calculus semantics. This component maps program elements to the underlying platform (that is Scala).

1.2.3 Aggregate programming platform

`scafi` provides a platform which supports both the definition and the execution of distributed aggregate applications.

Devices

An aggregate system consists of a network of interacting devices, immersed in some kind of environment. Each device may be equipped with sensors, to detect or measure a property of its surrounding environment, and/or actuators, to perform an action in its surrounding environment. Computation is organized in rounds: a device runs its local aggregate computation with a certain frequency. Each device broadcasts the result of its computation (export) to its neighbors, and receives, in turn, messages containing the exports produced by its neighbors. This is a conceptual model, since the actual behavior of a device depends on the kind of platform used.

Actor-based platform

In `scafi`, a general distributed platform is defined for supporting execution of aggregate programming systems. An actor-based platform specifies the base platform providing an implementation based on Akka[3] actors. In turn, the actor-based platform splits into two different platforms:

- peer-to-peer platform (decentralized),
- server-based platform (with a centralized unit for coordination).

In [8], these platforms are illustrated, and the following consideration is pointed out: aggregate programming is a computational model that can transparently fit a variety of execution platforms, due to its ability of declaratively designing systems by global-level abstractions. Therefore, since aggregate programming abstracts from deployment and execution details, other platforms could be introduced to handle these aspects differently than the currently supported platforms (P2P and server-based).

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

Simulation

`scafi` supports the simulation of aggregate programming systems. A simulation platform is introduced, modeling the whole computational system. Spatial simulation are also supported.

Simulations can be performed in different modalities:

- `scafi` provides an internal simulator (`simulator` module) and an appropriate view (`simulator-gui module`), respectively to run the simulation and allow the user to interact with the simulation in progress.
- Simulation can also be performed externally. In [9], an integration between `scafi` and the Alchemist simulator is carried out. Alchemist[10] is an event-driven simulator, mostly written in Java, tailored to the simulation of pervasive systems with a focus on performance. With a mapping between the Alchemist meta-model and the concrete `scafi` entities, complex simulations can be designed in `scafi` and executed in Alchemist.

Platform configuration

The system (platform) configuration in `scafi` works on two levels:

- the developer can build a system architecture by selecting a predefined incarnation or by creating its own. An incarnation is a platform embodiment, in which a platform is specified, defining the set of types to work with.
- setup of the system settings.

Chapter 2

Analysis

This chapter provides a description of the analysis phase in the development process. The goal of the thesis is delineated, along with the functional requirements and overall properties of the system, providing the input of the design phase.

Outline:

- Project requirements
- Requirements analysis, focusing on the concept of flexibility in a framework for distributed systems, `scafi`
- Problem analysis, detection of critical elements and proposal for potential solutions

2.1 Requirements

The requirements share the common goal of ensuring a flexible implementation of distributed systems with aggregate computing. The work must be carried out on the aggregate programming framework `scafi`, using the Scala programming language and the Akka toolkit.

2.1.1 Serialization

- Implementation of a serialization strategy in `scafi` for message passing between different nodes of the network.

- Each message that crosses the network must be serialized (and deserialized once it has reached its destination).
- The serialization format should promote openness and interoperability.

2.1.2 Hybrid platform

In [11, 12], a classification of peer-to-peer networks is outlined, between pure P2P systems and hybrid P2P systems. Hybrid models are a combination of peer-to-peer and client-server models; a common hybrid model is to have a central server that helps peers find each other. In this categorization, the peer-to-peer platform implemented in `scafi` belongs to the class of pure P2P systems (purely decentralized).

The design and implementation of a hybrid platform, fitting in the class of hybrid P2P system, is required.

- It should be possible to use the hybrid platform instead of one of the preexistent platforms (peer-to-peer and server-based).
- The hybrid platform must exploit peer-to-peer communication for the propagation of the exports between the devices.
- The hybrid platform should use a server, to track the positions of the devices and the relations between every device and its neighborhood.
- A device, at pre-start, must register with the server, and it should be able, at any time, to acquire from the server the information needed to contact its neighbors directly.

2.1.3 Spatial platform

- The platform must support a form of spatial computing (spatial network). Each device (or node) of the network must be situated in a space, provided by the spatial abstraction of the platform.
  - A device is located at a position in the space.

  - The neighboring relation is a function from a position to an arbitrary set of positions.
  - Each device has a set of neighbors, consisting of the devices located within a predefined range of distance from its position.

- A device should merge the exports of its neighbors before computing its program.
- The network is dynamic: the position of the devices may change over time.
- Spatial computing should be platform-independent.
  - Server-based platform: the server manages the neighboring relations between the devices of the network.
  - Peer-to-peer platform: each device must be aware of the state of its neighborhood.
  - Hybrid platform: the server keeps track of the neighborhood of every device.

2.1.4 Code mobility

- Integration of a code mobility approach in `scafi`.
- It must be possible to change the aggregate program executed by the devices at runtime. It should be necessary to send the new program to only one device of the network.
  - After receiving a new program, the device immediately starts executing it, replacing the current program (if present).
  - The device propagates the new program in the network, sending it to its neighbors. The propagation of the program should be coherent with the underlying platform (peer-to-peer, hybrid or server-based).

2.1.5 Platform/view integration

`scafi` provides an actor platform for execution of distributed systems and a simulation module that allows for aggregate simulations to be carried out locally. While the simulator is tied to a graphical user-interfaces (GUI), the platform execution has no graphical output.

- An integration between the actor platform and the simulator GUI must be realized.
- The GUI should reflect the state of a running distributed system.
  - It must be possible to observe the execution of a distributed system on the GUI.
  - It must be possible to interact with devices (moving the devices, trigger devices' sensors, . . .).

2.2 Requirements analysis

All the requirements impact on the actor platform of `scafi`; some of them require a direct intervention on the existing code, while others need a more careful analysis and design to identify the best implementation.

2.2.1 Serialization

In the current version of `scafi`, the serialization is not implemented. Since the platform is distributed, this aspect is crucial to support remote message passing. Every message sent by a device could be directed to another device located on a different node; this communication will fail if the platform does not provide a process for converting the state information of a message into a binary or textual form.

The platform messages, defined in module `spala` (mostly inside the trait `it.unibo.scafi.distrib.actor.PlatformMessages`), are object-oriented; objects needs to be serialized as well. For example:

```
case class MsgExport(from: UID, export: ComputationExport)
```

The serialization of `MsgExport`, implies also the serialization of objects `from` and `export`, respectively of type `UID` and `ComputationExport` (both are abstract types[1] defined in `it.unibo.scafi.distrib.BasePlatform`).

A serialization strategy could be implemented in the `scafi` platform, exploiting the Akka framework[13]:

- **Default serialization**: Akka provides a default serializer (customizable by configuration) that relies on Java serialization.
- **Custom serialization**: Akka allows the developer to build its own serializer and apply it for the serialization and deserialization of predefined objects.

Since the requirements demands for openness and interoperability, the definition of a custom serializer is preferable. In this way, messages could be converted into a standard format, like XML or JSON, and subsequently turned into bytes.

JSON[2] (JavaScript Object Notation) is a lightweight data-interchange format. Using JSON, data objects are transmitted in a human-readable text, as attribute-value pairs and array data types. For example, in the serialization process in `scafi`, the message `MsgExport` could be converted into the following JSON object:

```
{
    "from": <ID>,
    "export": <EXPORT>
}
```

A JSON serialization of `ID` and `EXPORT` must be provided.

2.2.2 Hybrid platform

A more accurate study of the hybrid platform concept is necessary. In [14], comparison and analysis of well-known P2P hybrid systems can be found. Most

[1] https://docs.scala-lang.org/tour/abstract-types.html
[2] https://www.json.org

of them are file-sharing services, but the concepts expressed are general enough to be used in different domains as well. In [12], a further classification of P2P hybrid networks is defined:

- Centralized indexing: a central server stores all the information regarding location of nodes. Each peer maintains a connection to the central server, through which the queries are sent.
- Decentralized indexing: a central server registers the users into the system and facilitates the peer discovery process. In these systems, queries are not handled by the server, but by supernodes, nodes that assume a more important role than the rest of the nodes. Supernodes are dynamically elected, usually on the basis of their processing power and bandwidth.

The approach with centralized indexing is the more appropriate solution to match the requirements. Though, the development of a platform that exploits decentralized indexing could be considered as an interesting future work: in these systems, the absence of a single point of failure and the presence of self-organization mechanisms, ensure scalability, robustness, and flexibility.

2.2.3 Spatial platform

The spatial platform has already been partially implemented. In [1], Casadei provides the design (and implementation) of a spatial abstraction in `scafi`; a metric spatial abstraction is also introduced, expressing how distances between positions are calculated. This previous work could be reused and extended to produce a complete implementation of the spatial platform.

2.2.4 Code mobility

Since distribution is a key concept in this work, an implementation of the feature of code mobility must allow the movement of programs between remote nodes, across the network. In `scafi`, a code mobility approach should follow the subsequent steps:

1. A node (sender) initiates a code mobility process by sending some code to another node (receiver).
2. The code is subjected to a serialization process.
3. The receive must be able to retrieve the actual code and to load it locally.
4. The receiver executes the code.

In this context, two different types of code mobility could be analyzed and implemented:

- Weak code mobility: the receiver already has the code. In this case, the receiver, after receiving some code, has to do a local check for the code presence and then, if found, can simply execute it.
- Strong code mobility: more complex scenario, the receiver does not have the code yet. In this case, the sender should provide some mechanisms to allow the receiver to load all the classes used in the code.

The second approach should be the ultimate goal, because it ensures more flexibility, allowing to assign a new program to the devices even at runtime. Though, it can be a not easy target and, initially, could be more appropriate to focus on the first one.

As stated in subsection 1.2.2, `scafi` implements higher-order field calculus, which supports distributed first-class functions, that are handled like any other data. Therefore, functions can be moved between devices (with `nbr` construct). The code mobility problem can be considered a similar problem. In fact, if you can move a user-defined or anonymous function from a node to another, the receiver can apply it, achieving the goal of code mobility. Moving a program is equivalent to moving a parameterless function (`scala.Function0[+R]` in Scala) that returns the program when it's executed.

However, not all functions can be trivially transferred from a device to another; this is the case of impure functions. An impure function is a function that mutates variables/state/data outside of it's lexical scope; its return value does not solely depend on its arguments. Adopting the weak code mobility approach, an impure function can produce different results when transferred to another device. In this work, code mobility has to support only pure functions.

2.2.5 Platform/view integration

This activity could lead to the achievement of some interesting goals. Since the requirements do not force constraints on implications of this work, all possible directions should be analyzed.

Simulation on the actor platform

In this scenario, an actual actor platform is internally built for the purpose of executing the simulation. This approach ensures a more accurate and realistic simulation. A possible workflow could consist of the following steps.

1. Configuration of the simulation, with a flag that specifies the execution environment (classic simulator or internal platform).
2. Creation of the actor platform (the peer-to-peer platform is preferable in terms of performances).
3. Scheduling of the devices, to execute the rounds of the simulation directly on the platform.
4. Observation of the devices to retrieve the exports produced.
5. Periodic update of the view.

This approach to simulation could also lead to a prototype of parallel and distributed simulation with minimal effort, considering that the platform is designed to be distributed on different node. The execution of parallel and distributed simulation that is both consistent and efficient, though, is not an easy target. As stated in [15], a synchronization algorithm is required to ensure that the parallel execution of the simulation produces exactly the same results as a sequential execution on a single processor; moreover, one of the main issues consists in distributing information among the nodes in an efficient and timely manner.

Monitoring of the actor platform

The view could be used as a monitor/controller of a distributed system. The view is already equipped with all the graphical capabilities necessary to display the state of an aggregate programming system. The current source for the visualization process is the set of exports produced by the simulated devices. An alternative source could be defined, observing an actual running system, and using the exports produced by devices that are part of it. A possible workflow could consist of the following steps.

1. Implementation and deployment of a distributed system, developed using the `scafi` framework (and the actor platform).
2. Configuration of the view, to establish a connection to the devices in the deployed system.
3. Observation of the devices, to retrieve the exports produced.
4. Periodic update of the view.

An interesting consideration in this scenario is that the view is an external component to the running system; in this way, a running system could be monitored and controlled at any time.

In this thesis, we refer to the just described approach as monitoring, although it is also characterized by elements typical of a control architecture.

2.3 Problem analysis

2.3.1 Neighborhood control in the spatial platform

In the spatial context, the critical issue relies on the management of the neighborhood of each device. The spatial platform must be implemented for all the supported platforms (server-based, hybrid, and peer-to-peer).

- In the server-based platform, devices don't communicate directly with each other, but their communication is mediated by the server. The server, which has a complete knowledge of the network (including the position of every device), can forward a message received from a device to all its neighbors.

- The peer-to-peer platform is fully decentralized, there is no central unit; therefore each device needs a technique to detect its neighbors and a mechanism to directly contact them. A distinction has to be made, between fixed networks and mobile networks.
  - Fixed network: the devices are located in a specific predefined position, that does not change at runtime. In this case, it's possible to assign to each device all the neighborhood related information before deploying the system.
  - Mobile network: the position of the devices may change at runtime. If a device moves, it must update the status of its neighborhood. The most relevant alternatives for solving this problem are listed below.
    - The following assumption could be made: a device, at any time, is able to sense its neighborhood and interact with the devices contained in it. Ideally, a device should be equipped with a sensor that can detect all the devices nearby, or programmed for a discovery phase.
    - If a device receives the position of all the other devices, and has access to the neighboring logic, it can deduce the identity of its neighbors and interact with them.
    - In the event that previous assumptions can not be guaranteed, use of the hybrid platform is recommended.
- The hybrid platform, analyzed in subsection 2.2.2, solves this problem by using a central unit (server) to track the devices. The server knows the position of the devices, and therefore it is able to provide information about the neighborhood of each device. Devices communicate directly with each other; the server is exploited only to inform a device about its neighbors.

2.3.2 Communication

In `scafi`, the communication technique used in the platform is message passing. The platform is actor-based (and Akka-based), and every node of the network (devices, server) is implemented as an actor. Since one of the main goals of `scafi` is to build an aggregate distributed platform, each actor could be located in a different node of the network, raising the communication problem. Currently, both peer-to-peer platform and server-based platform rely on Akka Remoting.

Akka Remoting, as described in [13], is a communication module for connecting actor systems in a peer-to-peer fashion. It's based on the principle of location transparency; there is nearly no API for the remoting layer of Akka (it is purely driven by configuration). The connection between systems is symmetric: each system is equally connected to the other, and there is no system that only initiates or accepts connection. The client/server model with predefined roles cannot be safely created.

To properly address the communication problem, an analysis of applications that can be potentially developed with `scafi` has to be made. In the context of the Internet of Things, the possibilities are countless; from programming a wireless sensor network with low-power devices and short-range radio communication, to the realization of a system composed of smartphones communicating across the Internet. Therefore, it should be possible to build the most suitable communication for each specific system on top of the platform.

For example, a support for communication across the Internet could be implemented as an extension of the existing platform, exploiting Akka networking techniques, described in [13].

- Akka I/O: is an API designed to match the underlying transport mechanism (TCP, UDP), and to be fully event-driven, non-blocking and asynchronous. Akka I/O is completely actor-based, and is meant to be used for the implementation of network protocols and building higher abstractions.
- Akka HTTP: is a module, on top of the actor module, made for building integration layers based on HTTP. It offers a full client/server stack, for both providing and consuming HTTP-based services.

The latter could be an interesting choice for interoperability and openness, providing support for aggregate heterogeneous systems. In fact, with a formalization of the HTTP API supported by the platform, even devices with different implementation (in terms of architectures or languages), could join the aggregate computation.

In this case study, the analysis of communication techniques for `scafi` has been limited to the only relevant implementations provided by Akka. Nevertheless, some Akka-independent approaches could also be taken into account.

2.3.3 Simulation versus control

In subsection 2.2.5, simulation and monitoring are depicted as two distinct activity. Basically this assumption is true, but considering the execution on the actor-based platform of `scafi`, these two concepts overlaps.

Following the steps enunciated in subsection 2.2.5, a simulation relying on the actor-based platform can be executed. Each device of the simulated system is mapped on a platform actor, that has the task of carrying on the computation. Sensors, actuators and position of every device are simulated, allowing the user of the simulation to set these value as preferred. The simulation view displays the state of each device at any time.

Keeping in mind this simulation context, let's imagine to spread the device actors to different nodes, reachable between them (this operation is absolutely legit, since distribution is a key element of the platform). We have now obtained a form of distributed simulation, since each actor contributes to the result of the simulation. We assume that the node on which an actor is running can be considered as a device, characterized by a position in a space; the node can also be equipped with sensors and actuators. In this case, we have an aggregate system composed of devices distributed in the network, each performing a local computation based on its sensors and on the state of its neighbors; this is the description of a deployed platform in execution. Though, the user of the simulation can still impose values of sensors and position of devices.

Considering this particular situation, questions arise: is it still a simulation?

If not, how can this scenario be defined?

- Simulation, because the simulator orchestrates the computation by controlling the scheduling of devices, and the user can still simulate events and changes in the status of the aggregate system.
- Monitoring, because a view displays the status of a running aggregate system.
- Control, because the user can interact with a running aggregate system, changing its status.

Apart from the formal definition, the analysis of this scenario is particularly worthy of attention, as it allows a transition from a simulation of an aggregate system to its actual monitored execution in a few basic steps.

Chapter 3

Design

The design phase focuses on the inclusion of the concepts produced in the analysis phase into the preexistent architecture of `scafi`. In this chapter, the design of the actor platform of `scafi` is presented, with a particular reference to the work accomplished in [1], and to the contribution introduced with this thesis.

Outline:

- Architecture of the actor platform
- Design that focuses on the key elements for platform distribution
- Design for the realization of platform monitoring

3.1 Platform architecture

The platform of `scafi` has been designed to provide flexibility for design, deployment, and execution of distributed systems. The distributed platform, through its façade API, is responsible for:

- parsing the incoming settings of the aggregate system, provided by the system developer;
- configuration and setup of the platform;
- definition of the aggregate application, along with the creation of devices;
- start of the system.

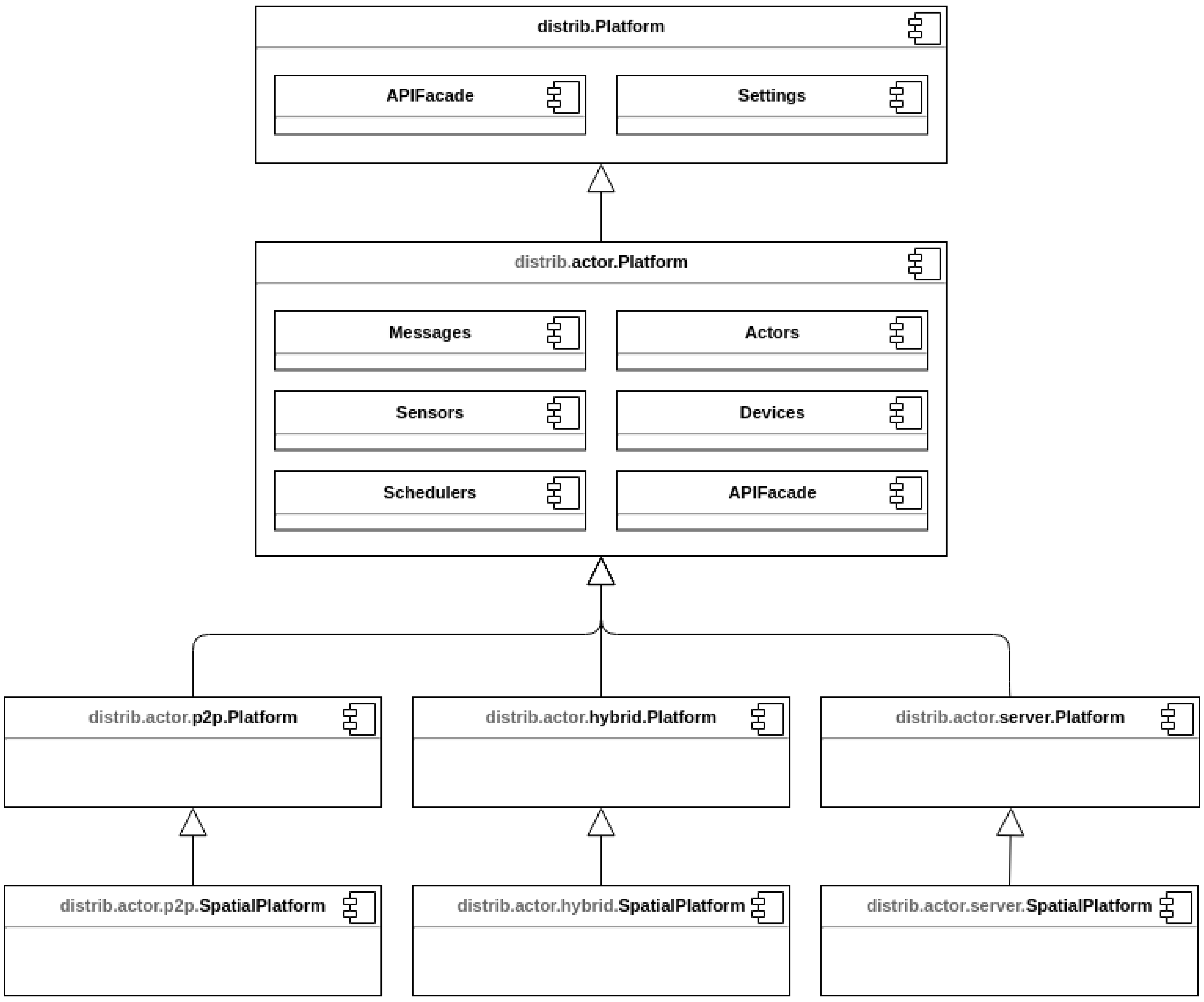
distrib.Platform
APIFacade
Settings
distrib.actor.Platform
Messages
Actors
Sensors
Devices
Schedulers
APIFacade
distrib.actor.p2p.Platform
distrib.actor.hybrid.Platform
distrib.actor.server.Platform
distrib.actor.p2p.SpatialPlatform
distrib.actor.hybrid.SpatialPlatform
distrib.actor.server.SpatialPlatform

Figure 3.1: Architecture of the actor platform (*reworked from [1]*).

The actor platform, as shown in Figure 3.1, is a specialization of the distributed platform, exploiting the actor model. Given the significant complexity of the overall design, the platform is split into several components, each one managing a specific concern.

- `Messages`: definition of the messages exchanged between actors and devices inside the platform.

- `Actors`: definition of actors to manage the given aggregate application (by creating the devices and supervising them).
- `Sensors`: definition of sensor actors, providing input values.
- `Devices`: definition of actors representing system's devices, able to perform computation in the aggregate computing fashion.
- `Schedulers`: definition of scheduling strategies, to organize aggregate computation on devices.
- `APIFacade`: extension of the façade of the distributed platform, to allow for the adaptation to an actor-dependent context (actor platform configuration, creation of device actors, ...).

The actor platform splits into:

- actor-based, peer-to-peer platform (purely decentralized),
- actor-based, hybrid platform (hybrid architecture),
- actor-based, server-based platform (centralized).

For each platform, a different `Spatial Platform` has been defined, as a specialization for providing a form of space-aware computing.

3.2 Detail design

3.2.1 Devices

In the actor-based platform, each device is modeled as an actor. In particular, a `ComputationDeviceActor` is a device capable of executing a local computation, integrating information from sensors and nearby devices. This actor is characterized by a working behavior, in which the device performs a round of the computation, by executing the program locally. The computation phase is performed at regular intervals, with a certain frequency; receiving of a tick message (`GoOn`) triggers the computation. The tick message can be sent to the device both by external components and by device itself. This working behavior allows for the scheduling of the devices by an external entity, as well as the autonomous execution of each device.

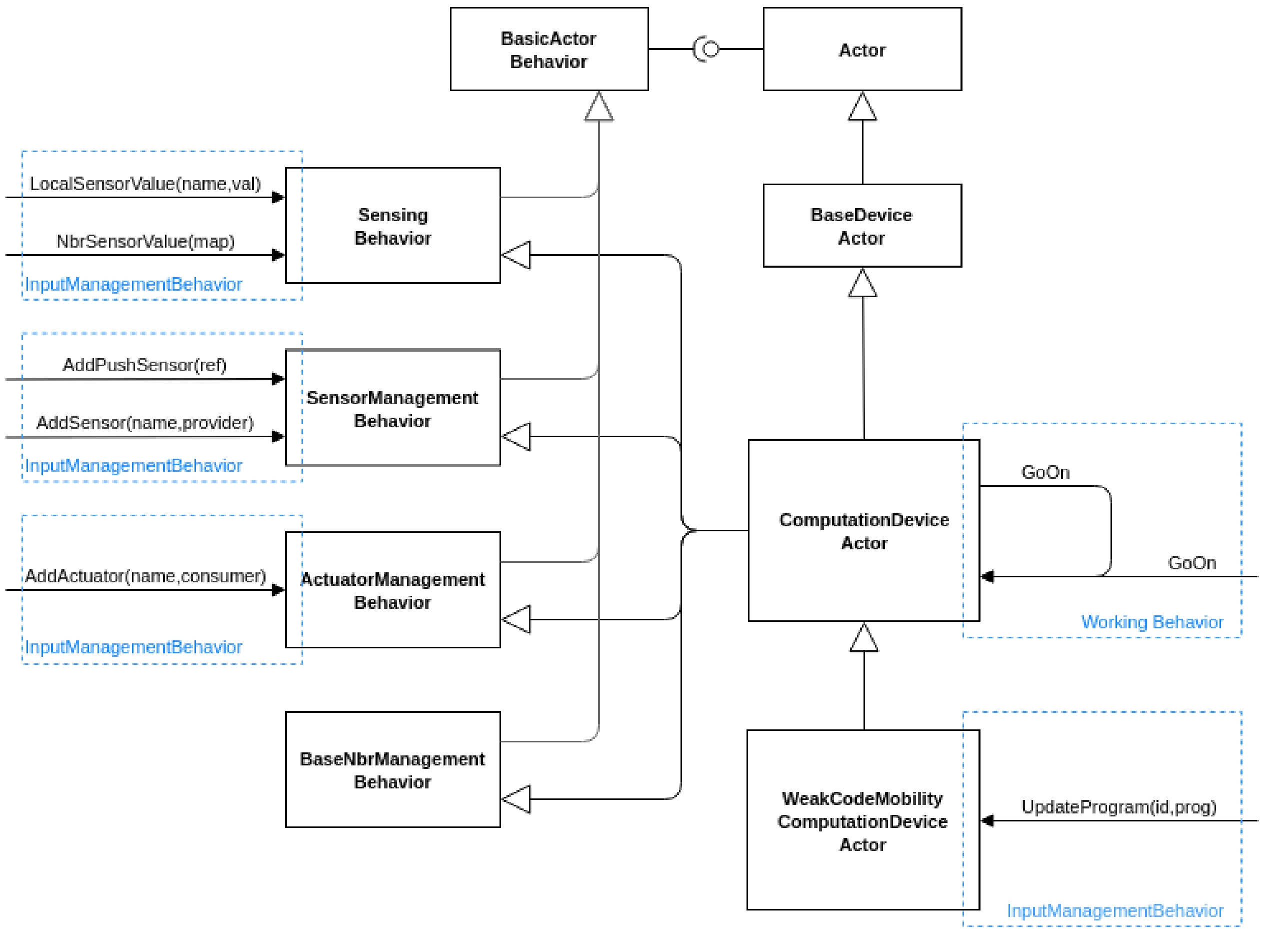

Figure 3.2: Structure and interface of device actors (*reworked from [1]*).

As shown in Figure 3.2, the complex behavior of a `ComputationDeviceActor` is subdivided into different small behaviors, to encourage code reuse and composability. Each behavior is a specialization of `BasicActorBehavior`, and manages a specific kind of interaction.

- `SensingBehavior`: manages all the values retrieved from the sensors of the device. This behavior is crucial for executing a situated computation.
- `SensorManagementBehavior`: manages the connections between the device and its sensors.
- `ActuatorManagementBehavior`: manages the connections between the de-

vice and its actuators.

- `BaseNbrManagementBehavior`: manages the interaction between the device and its neighbors. This behavior is closely related to the type of actor platform used (interaction with the server in server-based platform, direct communication with other devices in both p2p and hybrid platform).

`WeakCodeMobilityComputationDeviceActor` supports the approach of weak code mobility, described in subsection 2.2.4. This actor is defined as an extension of `ComputationDeviceActor`, and manages all the actions needed for the program switch at runtime, along with the propagation of the new program to its neighbors. The implementation of this last task, just like the implementation of the `BaseNbrManagementBehavior`, depends on the type of actor platform in use.

3.2.2 Hybrid platform

The design of the hybrid platform follows the indications expressed in subsection 2.2.2. The devices are clients of a central server that owns the information about the topology of the network.

In Figure 3.3 the static architecture of the hybrid platform is shown, along with significant interactions between components.

- `Registration`: each device, at startup, registers with the server.
- `Export`: after the execution of its local computation, each device propagates its computational state directly to all its neighbors (same behavior as that of the peer-to-peer platform).
- `GetNeighborhoodLocations`: periodically, each device queries the server, receiving in response a message with the identity and reference of its neighbors (`NeighborhoodLocations`).
- `Neighbor`, `Neighborhood`: the server is informed about neighborhood relations of a device.
- `Position`: in the spatial platform, each device communicates its position to the server; in this way the server can provide accurate information about the neighborhood of every device.

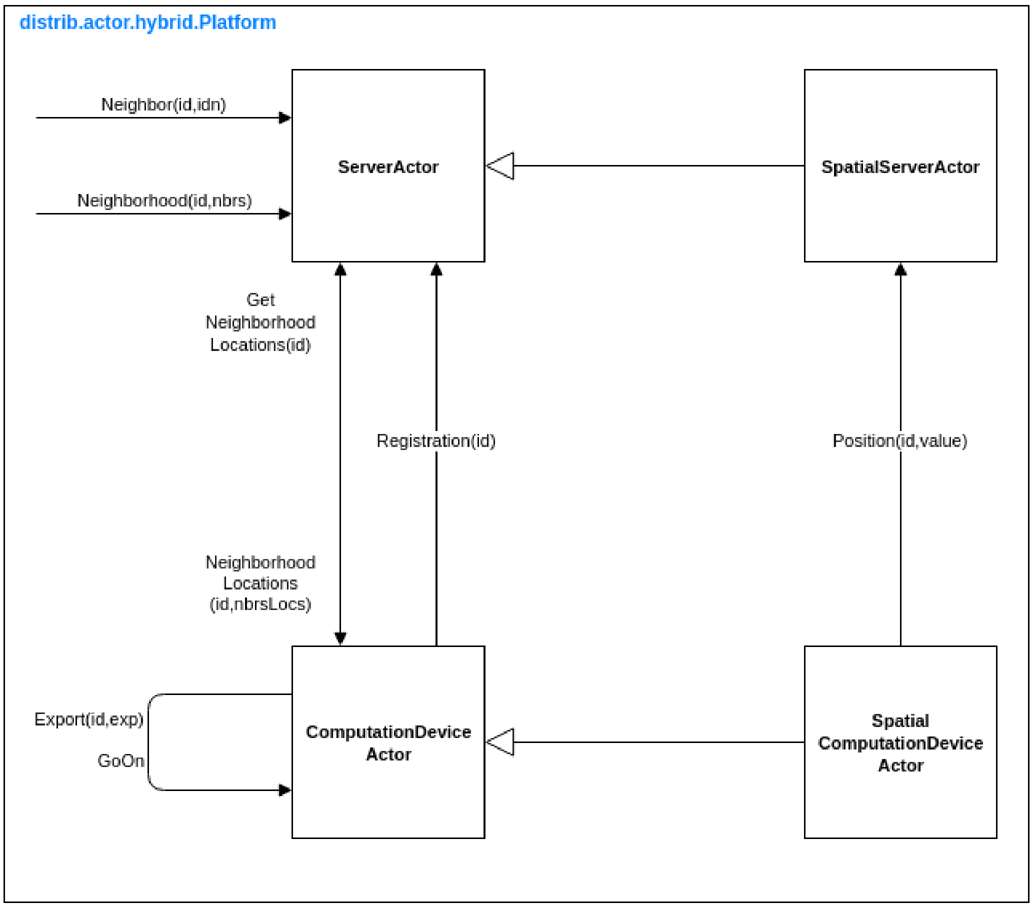

Figure 3.3: Design of the hybrid actor-based platform.

The hybrid platform, like any other actor-based platform, can be enhanced with spatial extension, or used in its base form; though, it's conceived for space-aware devices in a mobile network. If the network is fixed and neighborhood relations are predefined, a peer-to-peer approach should be widely preferable.

3.2.3 Serialization

The serialization strategy consists in the definition of a custom JSON serializer, which ensures interoperability, as stated in subsection 2.2.1.

As shown in Figure 3.4, the design of the serialization adopts a modular approach. The overall task is divided between different components, each responsible for the serialization and deserialization of a specific set of objects.

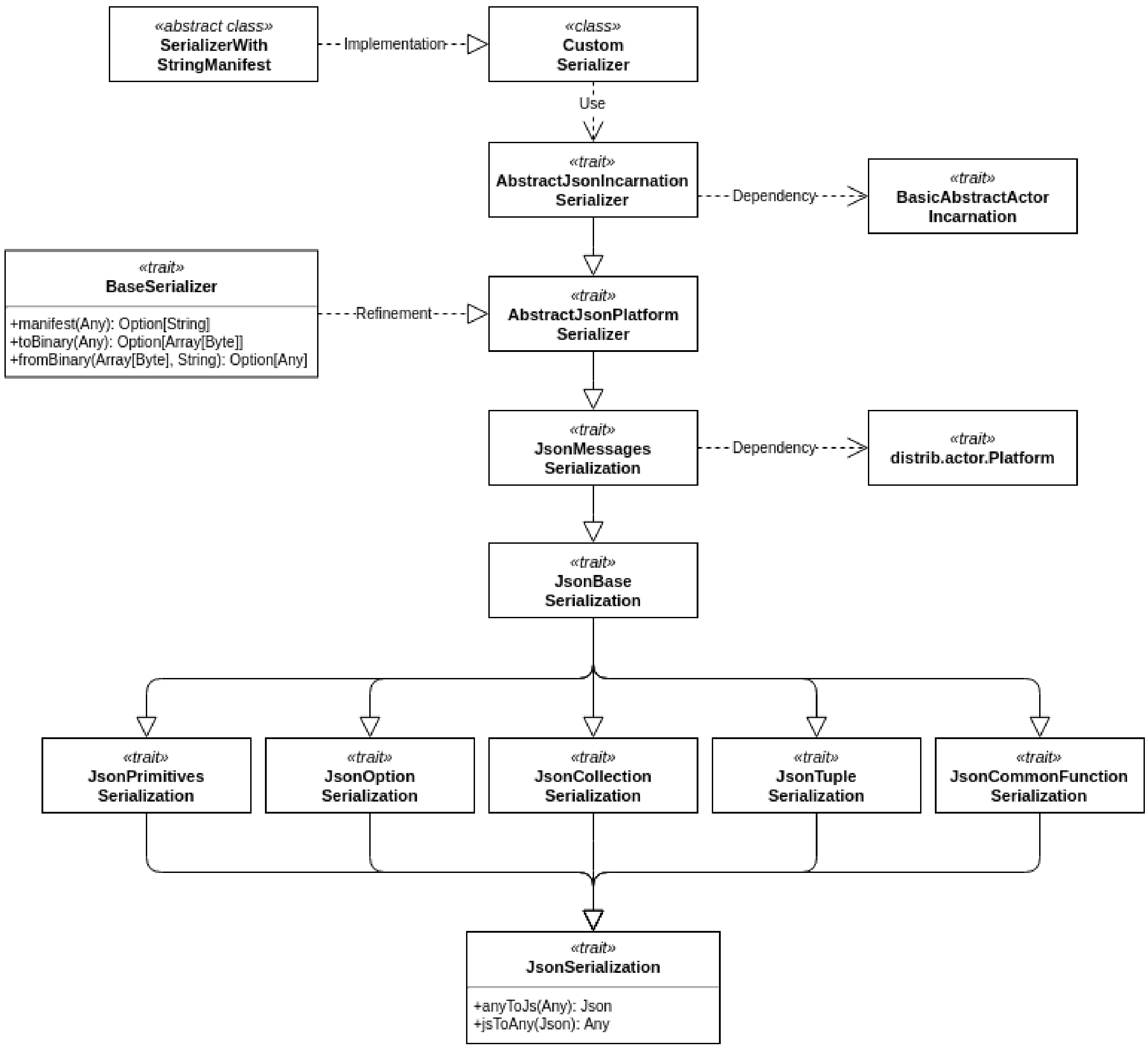
«abstract class» SerializerWith StringManifest
Implementation
«class» Custom Serializer
Use
«trait» AbstractJsonIncarnation Serializer
Dependency
«trait» BasicAbstractActor Incarnation
«trait» BaseSerializer
+manifest(Any): Option[String]
+toBinary(Any): Option[Array[Byte]]
+fromBinary(Array[Byte], String): Option[Any]
Refinement
«trait» AbstractJsonPlatform Serializer
«trait» JsonMessages Serialization
Dependency
«trait» distrib.actor.Platform
«trait» JsonBase Serialization
«trait» JsonPrimitives Serialization
«trait» JsonOption Serialization
«trait» JsonCollection Serialization
«trait» JsonTuple Serialization
«trait» JsonCommonFunction Serialization
«trait» JsonSerialization
+anyToJs(Any): Json
+jsToAny(Json): Any

Figure 3.4: Architecture of the serialization module.

- `JsonSerialization` is an interface that represents a component able to provide both serialization and deserialization of a generic object.
- A bunch of specific components are defined for the serialization of some Scala

built-in classes: predefined value types[1], a subset of immutable collections, the `scala.Option` class, the tuple classes[2], and higher-order functions.

- `JsonBaseSerialization` extends and combines the aforementioned components, to provide serialization of relevant Scala built-in types.
- `JsonMessagesSerialization` is the trait dedicated to the serialization of messages exchanged in the actor platform. This component extends `JsonBaseSerialization`; and it depends on the actor platform, since it has to manage messages and key elements of the aggregate computation.
- `AbstractJsonPlatformSerializer` is the designated serializer for the actor platform. This component implements the contract defined in the interface `Base Serializer`, and exploits the serialization capabilities of `JsonMessagesSerialization`.
- The platform makes extensive use of Scala abstract types, while the actual types are defined inside the incarnation of the actor platform, `BasicAbstractActorIncarnation`. Information about the actual type is mandatory to properly serialize objects typical of aggregate computing (e.g. `ComputationExport`). A further serializer, `AbstractJsonIncarnationSerializer`, is required to manage the serialization of objects defined in the platform incarnation.

As described in [13], a custom serializer, in Akka, has to inherit from a predefined abstract class (in this case `SerializerWithStringManifest`). Therefore, the component `CustomSerializer` is defined as an implementation of the Akka serializer, and uses the `AbstractJsonIncarnationSerializer` for the actual process of serialization.

3.3 Platform monitoring

In subsection 2.2.5, two options are presented and proposed for the integration between the actor platform and the view: simulation on the actor platform and

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

monitoring of the actor platform. These proposals are strictly related to each other, and the realization of one does not exclude the other. However, in this work, the focus is placed on the realization of the second: platform monitoring.

3.3.1 Simulation front-end

The development process that led to the design and implementation of the graphical front-end in `scafi` is described in [16] (considered as a general reference for this section).

As shown in Figure 3.5, the architecture design of the front-end followed the main principles of the architectural pattern model-view-controller (MVC), along with the frequent application of the design pattern observer.

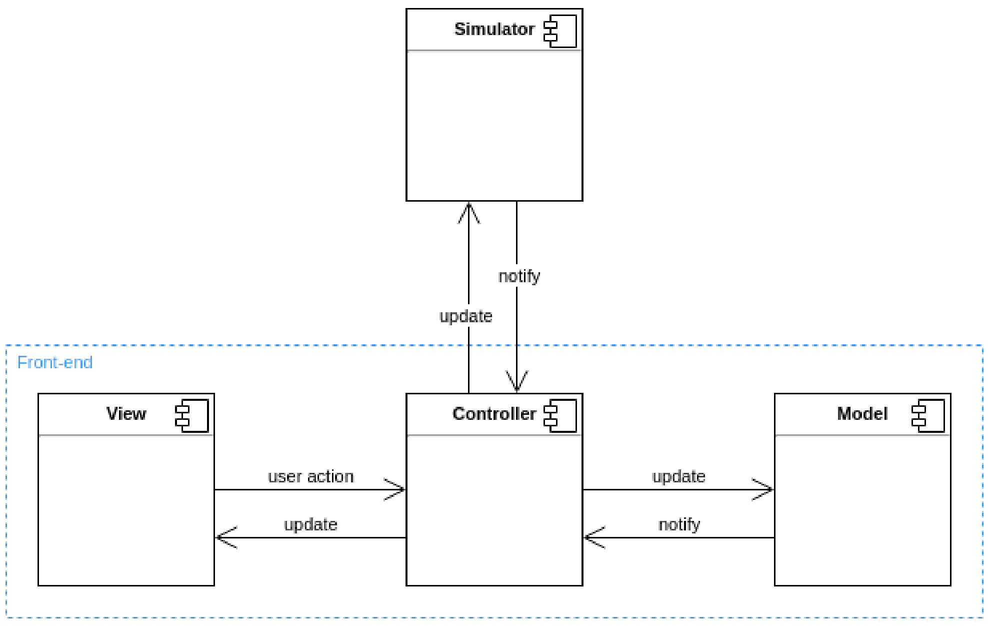

Figure 3.5: Structure of the front-end with macro components.

Unlike classical MVC designs, the front-end is characterized by the definition of two distinct and independent models.

- Logical model (`Simulator` component): maintains the logical representation of the aggregate system. It's updated by the simulator after the execution of each round of computation. The computation frequency of a round is set by the `Controller`.

- Graphical model (`Model` component): represents the world state displayed by the view. It's updated by the `Controller` considering the state of the logical model. The update frequency is set by the `Controller`, and it can be different from the update frequency of the logical model (e.g. it's reasonable to think that, in the same period of time, the update of the logical model is performed more often than the update of the graphical model).

The independence between the two models allows to decouple the two worlds: the visualization is not strictly related to the logical model. In this way, it's possible to change the existing logical system, ensuring the same display results.

3.3.2 Monitoring of the actor platform

Thanks to the separation of concerns principle, applied in the front-end design, the interventions necessary to the implementation of the monitoring task are localized and relate solely to the update mode of the logical model. In the simulation front-end, briefly described in subsection 3.3.1, the logical model is updated by the simulator, after simulating a round of computation. In the monitoring context, there's no simulator, but it's possible to rely on an actual platform in execution.

The `PlatformMonitor` component is introduced (instead of the `Simulation` component), as a bridge from the `Controller` to a running platform, exposing the same external interface provided by the `Simulation` component. A platform is composed of devices performing a local computation; the results of the computation, along with the last values retrieved from local sensors, resides in the single device. In the actor-based platform, a device is represented as a `ComputationDeviceActor` (further details in subsection 3.2.1), which can be extended with an observable behavior (`ObservableActorBehavior`), obtaining an `ObservableDeviceActor`. Inside `PlatformMonitor`, a `PlatformObserverActor` is defined with the purpose of observing all the devices, retrieving all the information necessary to provide a proper visualization of the system on the graphical front-end. Figure 3.6 shows the main structure of the monitoring system of the actor platform, along with the most significant messages exchanged between `PlatformObserverActor` and `ObservableDeviceActor`.

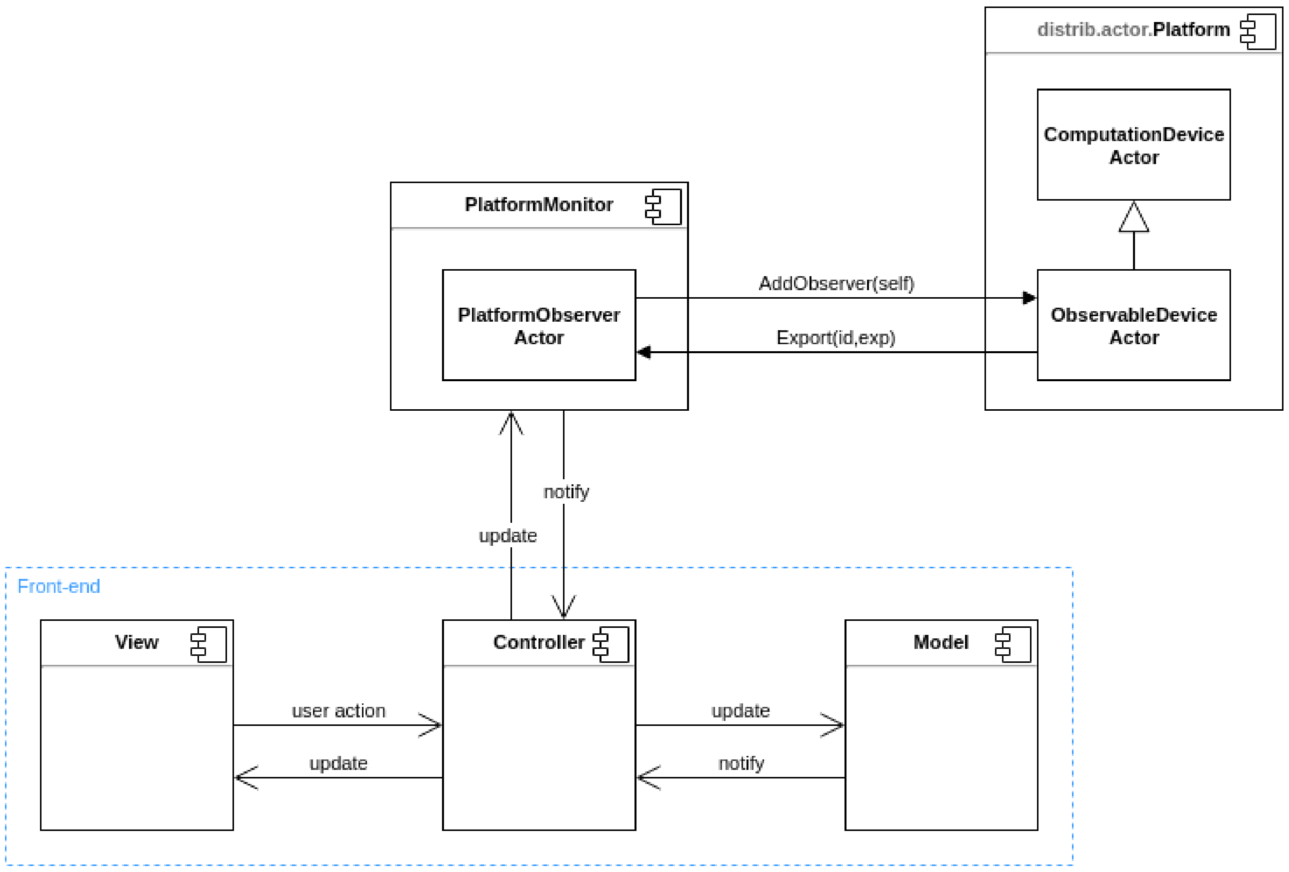

Figure 3.6: Structure of the platform monitoring system.

In this case, the observer pattern is applied to actors. `PlatformObserverActor` registers itself as an observer of `ObservableDeviceActor`, to stay up to date on the status of each device in the running system. An example is provided in Figure 3.6, in which an `Export` message is sent to `PlatformObserverActor`, in order to notify it about the result of a computation round. Though, other messages are sent as an effect of the observer pattern (e.g. the device notify the observers about values retrieved by its sensors).

Chapter 4

Implementation

The key elements for platform distribution, described in section 3.2, are implemented in the preexisting actor platform of `scafi`. This chapter presents an overview of the implementation phase, providing significant pieces of code in the process.

Outline:

- Hybrid platform implementation
- Serialization implementation
- Code mobility implementation
- Platform monitoring implementation

4.1 Hybrid platform

The key elements and constructs to build the hybrid platform are implemented in the package `it.unibo.scafi.distrib.actor.hybrid` of the `spala` module. In this work, some code has been reused from previous platform implementation (from p2p platform and server-based platform), after a refactoring process. The most significant part of the implementation of the hybrid platform consists in the definition of device actor and server actor; in this section, an overview of the implementation in regards to these components is provided.

4.1.1 Base actor platform

The following fragment of code encloses the implementation of `DeviceActor`.

```
class DeviceActor(override val selfId: UID,
                  override var aggregateExecutor: Option[ProgramContract],
                  override var execScope: ExecScope,
                  val server: ActorRef)
  extends P2pBaseDeviceActor with ObservableDeviceActor {

  // Scheduling of a periodic behavior to ask the server for neighbors' info
  context.system.scheduler.schedule(
    initialDelay = 0 seconds,
    interval = 1 second,
    receiver = server,
    message = MsgGetNeighborhoodLocations(selfId)
  )

  override def preStart(): Unit = {
    super.preStart()
    // Registration with the server at startup
    server ! MsgRegistration(selfId)
  }

  override def inputManagementBehavior: Receive =
    super.inputManagementBehavior orElse {
      case MsgNeighborhoodLocations(id, nbs) =>
        // ... Replacing the current neighbrhood info with nbs ...
    }
}
```

`P2pBaseDeviceActor` is a trait that provides the main definitions of a device actor in the peer-to-peer actor-based platform; in addition to general rules (inherited from `ComputationDeviceActor`), this component adopts a behavior to manage all aspects related to communication with neighbors (support for `nbr` construct) in the peer-to-peer fashion of direct communication. The device actor (`distrib.actor.hybrid.DeviceActor`) has a similar behavior, extended in order to perform an interaction with the server: registration at startup, scheduling of a behavior to ask the server for its neighbors, and management of the server response to update its internal knowledge of its neighborhood.

The implementation of the `ServerActor` is shown in the next frame.

```scala
class ServerActor() extends ServerBaseServerActor with ObservableServerActor {
  // Map containing the neighbors of each device
  val neighborhoods: MMap[UID, Set[UID]] = MMap()

  override def neighborhood(id: UID): Set[UID] =
    neighborhoods.getOrElse(id, Set())

  override def queryManagementBehavior: Receive =
    super.queryManagementBehavior.orElse {
      case MsgGetNeighborhoodLocations(id) =>
        // Retriving the reference of every neighbor of the requestor
        val locs = neighborhood(id)
                     .filter(idn => lookupActor(idn).isDefined)
                     .map(idn => idn -> lookupActor(idn).get.path.toString)
                     .toMap
        sender ! MsgNeighborhoodLocations(id, locs)
  }

  override def inputManagementBehavior: Receive =
    super.inputManagementBehavior orElse {
      case MsgNeighbor(id, idn) => addNbrsTo(id, Set(idn))
      case MsgNeighborhood(id, nbrs) => addNbrsTo(id, nbrs)
  }

  def addNbrsTo(id: UID, nbrs: Set[UID]): Unit = {
    // Updating neighborhood map and notification to observers
    neighborhoods += id -> (neighborhood(id) ++ nbrs)
    notifyObservers(MsgNeighborhood(id,nbrs))
  }
}
```

`ServerBaseServerActor` is a trait that provides the most general functionality of a server in the platform: tracking of devices; this component manages registration of devices and offers the implementation of the `lookupActor` definition, usable to retrieve the reference of a previously registered device. The server actor (`distrib.actor.hybrid.ServerActor`) has a similar behavior, extended in order to handle the `MsgGetNeighborhoodLocations` message.

4.1.2 Spatial actor platform

The spatial platform is implemented on top of the base platform, and uses a spatial abstraction, `MetricSpatialAbstraction`, to manage the representation of the space. The implementation requires the redefinition of the actors delineated in subsection 4.1.1. These actors must be extended to supply the capabilities necessary to carry out a form of spatial computation.

The base `DeviceActor` is extended by `SpatialDeviceActor`, as follows.

```
class SpatialDeviceActor(override val selfId: UID,
                         _aggregateExecutor: Option[ProgramContract],
                         _execScope: ExecScope,
                         override val server: ActorRef)
  extends DeviceActor(selfId, _aggregateExecutor, _execScope, server) {

  override def setLocalSensorValue(name: LSensorName, value: Any): Unit = {
    super.setLocalSensorValue(name, value)
    if (name == LocationSensorName) {
      // Notification of location to the server
      server ! MsgPosition(selfId, value)
    }
  }
}
```

The spatial device retrieve its position in the space by means of an appropriate location sensor, and forwards that information to the server.

The base `ServerActor` is extended by `SpatialServerActor`, as follows.

```
class SpatialServerActor(val space: MutableMetricSpace[UID])
  extends ServerActor {

  override def neighborhood(id: UID): Set[UID] = {
    // The neighborhood consists of the devices nearby in the space
    if(space.contains(id)) space.getNeighbors(id).toSet else Set()
  }

  override def inputManagementBehavior: Receive =
    super.inputManagementBehavior orElse {
      case MsgPosition(id, pos) =>
        // Set the device's position in the space
        space.setLocation(id, pos.asInstanceOf[P])
        // Notification of neighborhood of each device to observers
        this.space.getAll().foreach(id => {
          val nbs = this.space.getNeighbors(id)
          notifyObservers(MsgNeighborhood(id,nbs.toSet))
        })
  }
}
```

The spatial server has a representation of the space (MutableMetricSpace), provided by the spatial abstraction. The neighborhood function has to be redefined, since the neighborhood of a device depends on its position in the space. Receiving a MsgPosition message from a device causes a corresponding update in the space.

4.2 Serialization

The actualization of the serialization strategy is mainly implemented in the package it.unibo.scafi.distrib.actor.serialization of the spala module; the only exception is the trait AbstractJsonIncarnationSerializer, which belongs to the package it.unibo.scafi.incarnations of the distributed module. In the serialization process, the Play JSON library[1] has been used as a support for conversions from objects to JSON values, and vice versa.

The following frame shows the basic implementation of JsonSerialization.

```scala
trait JsonSerialization {
  def anyToJs: PartialFunction[Any, JsValue]
  def jsToAny: PartialFunction[JsValue, Any]
}
```

As already mentioned in subsection 3.2.3, serialization and deserialization are carried out respectively by anyToJs and jsToAny. The PartialFunction[-A, +B] return type allows for class composition with mixins[2]. This mechanism is applied to objects serialization in JsonBaseSerialization, as follows.

```scala
trait JsonBaseSerialization extends JsonSerialization
  with JsonPrimitiveSerialization
  with JsonOptionSerialization
  with JsonCollectionSerialization
  with JsonTupleSerialization
  with JsonCommonFunctionSerialization {

  override def anyToJs: PartialFunction[Any, JsValue] =
    // Composition of partial functions
    super[JsonPrimitiveSerialization].anyToJs orElse
    super[JsonOptionSerialization].anyToJs orElse
```

[1] https://www.playframework.com/documentation/2.6.x/ScalaJson
[2] https://docs.scala-lang.org/tour/mixin-class-composition.html

```
    super[JsonCollectionSerialization].anyToJs orElse
    super[JsonTupleSerialization].anyToJs orElse
    super[JsonCommonFunctionSerialization].anyToJs

  override def jsToAny: PartialFunction[JsValue, Any] =
    // ... Composition of partial functions, as above ...
}
```

The Play JSON library supports the definition in scope of implicit conversions[3], to specify how to perform the transformation from objects to JSON values and vice versa. Following this directive, implicits are defined for all the platform messages. The problem related to dependency on the actor platform is addressed through the introduction of a self-type[4], declaring that the `JsonMessagesSerialization` trait must be mixed into an actor-based platform.

```
trait JsonMessagesSerialization extends JsonBaseSerialization {
  self: distrib.actor.Platform =>

  implicit val msgExportWrites: Writes[MsgExport] = msg =>
    Json.obj("from" -> anyToJs(msg.from), "export" -> anyToJs(msg.export))
  implicit val msgExportReads: Reads[MsgExport] = js => JsSuccess {
    MsgExport(
      jsToAny((js \ "from").get).asInstanceOf[UID],
      jsToAny((js \ "export").get).asInstanceOf[ComputationExport]
    )
  }

  // ... implicit conversions for other messages
}
```

`AbstractJsonIncarnationSerializer` extends the serialization functions with incarnation-related types. The implementation of serialization and deserialization of `ComputationExport` is shown below.

```
trait AbstractJsonIncarnationSerializer
  extends AbstractJsonPlatformSerializer {
  self: BasicAbstractActorIncarnation =>

  override def anyToJs: PartialFunction[Any, JsValue] = super.anyToJs orElse {
    case e: ComputationExport =>
```

[3]https://docs.scala-lang.org/tour/implicit-conversions.html
[4]https://docs.scala-lang.org/tour/self-types.html

```
      Json.obj("type" -> "ComputationExport", "val" -> anyToJs(e.getMap))
    case // ... serialization of other objects ...
  }

  override def jsToAny: PartialFunction[JsValue, Any] = super.jsToAny orElse {
    case e if (e \ "type").as[String] == "ComputationExport" =>
      // Creation of an empty export
      val export = new EngineFactory().emptyExport()
      // Use of reflection to set the private map
      val field = classOf[ExportImpl].getDeclaredField("map")
      field.setAccessible(true)
      field.set(export,
        Map(jsToAny((e \ "val").get).asInstanceOf[Map[Any,Any]].toSeq:_*))
      field.setAccessible(false)
      adaptExport(export)
    case // ... deserialization of other objects ...
  }
}
```

4.3 Code mobility

The implementation of code mobility concerns only the preexisting trait `PlatformDevices` in the package `it.unibo.scafi.distrib.actor` of the `spala` module. Adopting the weak code mobility approach, the strategy consists in the definition of different pure functions of type `() => AggregateProgram` (which is syntactic sugar for `Function0[AggregateProgram]`) inside the platform. In fact, in this scenario, each device contains a reference to all programs and can switch between them, after receiving a corresponding message from another device. In particular, when a device receives a `MsgUpdateProgram` message, switches to the specified program (applying the function) and propagates the message to all its neighbors. This behavior produces a transition phase in the platform from the current program to a new one, without needing to stop the system. The `MsgUpdateProgram` message could be sent to a running device from a component external to the platform.

To allow code mobility in a distributed system, functions need to be serialized. Trait `JsonCommonFunctionSerialization`, mentioned in section 4.2, executes a sort of functions serialization based on reflection: the code is not actually transferred from a node to another, only the function reference is moved. In this way, the receiver can look for the function (locally) and apply it.

The key elements of the implementation of `WeakCodeMobilityDeviceActor` are shown in the following frame.

```
trait WeakCodeMobilityDeviceActor extends ComputationDeviceActor {
  var lastProgram: Option[() => Any] = None
  var unreliableNbrs: Set[UID] = Set()

  override def inputManagementBehavior: Receive =
    super.inputManagementBehavior.orElse {
      case MsgUpdateProgram(nid, program) => handleProgram(nid, program)
    }

  def handleProgram(nid: UID, program: () => Any): Unit = {
    // If the incoming program is different from the current one
    if (lastProgram.isEmpty || lastProgram.get != program) {
      lastProgram = Some(program)
      // All neighbors becomes potentially unreliable
      unreliableNbrs = nbrs.keySet
      // Reset of current computational context
      resetComputationState()
      // Replacing the current program with the incoming one
      updateProgram(program)
      // Propagation of program to all neighbors
      propagateProgramToNeighbors(program)
    }
    // The nbr that sent this is reliable, it has alredy embraced the program
    unreliableNbrs = unreliableNbrs - nid
  }

  // ...
}
```

A particular issue regarding program switch consists in successfully exceeding the transition phase. In this phase, there will be a moment in which a device will have already embraced the new program, while a subset of its neighbors have not yet done so. In this case, the devices computes one or more rounds with different programs, producing exports impossible to merge with each other. A further step is required, the selected strategy is the following:

- each device has a `unreliableNbrs` list (initially empty);

- a device adds all its neighbors to the list after receiving a `MsgUpdateProgram` containing a program different from its own;
- when a device receives a `MsgUpdateProgram`, removes the sender from the list (assuming that the sender has already embraced the program);
- only the exports of neighbors that are not in the list are used for the computation phase. This task is achieved by overriding the `beforeJob` method, originally defined in `ComputationDeviceActor`, and executed before each computation. Inside this method, all the exports coming from unreliable neighbors are removed, in order to prevent them being considered in the computation phase.

4.4 Platform monitoring

The implementation of actor platform monitoring is localized in the package `it.unibo.scafi.simulation.gui.incarnation.scafi.bridge.monitoring` of the `simulator-gui` module. The development process related to the monitoring of the actor-based platform focuses on the implementation of the `PlatformMonitor` class. This component has a particular role, since it has to integrate classical object-oriented programming (to provide an interface known by the `Controller`) with the asynchrony and reactivity typical of the actor model (to interact with actors representing the devices of the platform).

Interaction with device actors of the running platform is delegated to `PlatformObserverActor`, an actor created within the `PlatformMonitor` class. This actor immediately registers itself as an observer of every device actor, in order to be notified of every change in status; when this happens, it updates the state of `PlatformMonitor`, triggering the `Controller` (which is registered as an observer of the `PlatformMonitor` component). The implementation of `PlatformObserverActor`, as an inner class of `PlatformMonitor`, is reported in the following frame.

```scala
class PlatformObserverActor(platform: Platform) extends Actor {

  override def preStart(): Unit = {
    super.preStart()

    platformNodes.foreach(dev => {
      // Configuration of Akka path to remote device actor
      val path =
        self.path.address.copy(
          protocol = "akka.tcp",
          system = "scafi",
          host = Some(dev._2._1),
          port = Some(dev._2._2)
        ).toString +
        "/user/" +
        platformName +
        "/dev-" +
        dev._1

      context.system.actorSelection(path).resolveOne(2 seconds).onComplete {
        case Success(ref) => // Remote actor successfully retrieved
          devicesLocation += dev._1 -> Some(ref)
          ref ! MsgAddObserver(self)
        case Failure(e) => // Connection to remote actor failed
      }
    })
  }

  override def receive: Receive = {
    case platform.MsgExport(id, export) =>
      setExport(id, export.asInstanceOf[EXPORT])
    case // ... managing other messages coming from the observed devices
  }
}
```

Some terms, present in the above code, require a brief explanation:

- `platform`: class parameter that represents the platform in execution; its type (`Platform`) is an alias for `BasicAbstractActorIncarnation`, a component that provides some useful definitions of aggregate computing on top of the actor platform.
- `platformNodes`: object of type `Map[ID, (String, Int)]`, which, for each device of the system, indicates:
  - a value that uniquely identifies the device within the platform;
  - a tuple that represents the host in which the device is executing, identified by IP address (`String`) and port (`Int`).

- `path`: an Akka-based mechanism to identify and locate a specific actor in a distributed actor system. If the actor exists, its `ActorRef` can be retrieved from the path, by the `resolveOne` method of the `ActorSelection` class (further information can be found in [13]).
- `platformName`: the name of the remote actor system, in which the actors will be searched.
- `devicesLocation`: object of type `Map[ID, Option[ActorRef]]`, which correlate the `ID` of a device and its reference (if this information is available).

The objects `platformNodes`, `platformName`, and `devicesLocation`, along with the `setExport` function, are defined inside the `PlatformMonitor` class.

Chapter 5

Evaluation

In this chapter, the results of the development phase are analyzed, to determine the degree of success in achieving the initial goals.

Outline:

- Basic demos of future applications in the field of distributed systems
- Verification of the requirements

5.1 Demos

Demonstration programs have been developed as proof of concept, with a dual-purpose: put into practice the key elements introduced with this thesis and test the flexibility of the framework in developing distributed applications. Before this work, five demos were already available in `scafi` (from `Demo0` to `Demo4`), showing how the framework can be used to develop aggregate applications. As a result of the efforts produced in this thesis, three new demos have been added to the preexisting ones.

5.1.1 Spatial nets

The demos that exploit architectures classifiable as spatial nets are `Demo5` and `Demo6`. Each one of them supports:

- server-based, hybrid, peer-to-peer architectures;

- spatial platform;
- mobile network;
- view of the devices;
- command-line configuration.

Each demo is split in three different versions:

(A) server-based platform;

(B) peer-to-peer platform;

(C) hybrid platform.

The view of the devices, briefly described subsequently, has been developed only for demonstration and testing purpose, and it should not be considered as a part of the actual platform provided by the `scafi` framework.

View of the devices

The view supported the whole development process, establishing a way to both observe the state of the running platform and interact with the devices. The main characteristics of the view are summarized, as follows.

- The view is unique for all the devices of the network, even though they may be located on different nodes.
- The view simulates the space in which the devices are located. Each device is represented as a dot in the view, and it's placed at a specific spot in the view (based on its spatial position). The view shows the current export of every device.
- The view is constantly updated, to show an accurate status of the devices.
- A change in the position of a device can be simulated by moving the corresponding node in the view. Each move is notified to the platform.

The key component is `DevViewActor` (package `examples.gui` of the `demos` module). This actor acts as a bridge from the platform to the view and vice versa. Its task is to interact with a single `DeviceActor` to make the visualization of the view consistent with respect to the actual device's state. Although the view is unique, several `DevViewActor` are created, one for each device of the platform.

Spatial platform

The goal of this demo is to demonstrate the proper functioning of the spatial platform and to document programming of a spatial system in `scafi`. The demo is available in three different versions, each one based on a specific kind of actor platform. In this case, an example is shown, exploiting the server-based platform, and all the steps necessary to setup the system are described. Command-line configuration is exploited to complete the definition of the relevant settings.

Firstly, the server-based spatial platform is selected, along with the actor that will act as the view of every device.

```
import it.unibo.scafi.distrib.actor.server.SpatialPlatform
import examples.gui.server.DevViewActor
```

Then, the spatial platform is refined, obtaining the ultimate platform of this demo. This operation provides the embodiment of abstract definitions related to sensors and space.

```
object Demo5A_Platform extends SpatialPlatform
  with BasicAbstractActorIncarnation {

  override val LocationSensorName: String = "LOCATION_SENSOR"
  val SourceSensorName: String = "source"
  override type P = Point2D

  override def buildNewSpace[E](elems: Iterable[(E, P)]): SPACE[E] =
    new Basic3DSpace(elems.toMap) {
      override val proximityThreshold = 1.1
    }
}

import demos.{Demo5A_Platform => Platform}
```

The next step consists in the definition of the aggregate program, as a class extending the `AggregateProgram` trait. In particular, the `main` function contains the actual program, that will be executed by devices. In this case, a simple hop gradient program is implemented. A device is a source for the gradient program if it retrieves true from its `SourceSensor`.

```
class Demo5A_AggregateProgram extends Platform.AggregateProgram {
  def hopGradient(source: Boolean): Double = {
    rep(Double.PositiveInfinity){
      hops => { mux(source) { 0.0 } { 1 + minHood(nbr { hops }) } }
    }
  }
  def main(): Double = hopGradient(sense(Platform.SourceSensorName))
}
```

In order to start the platform from command-line, the `CmdLineMain` class is customized. Settings (that will be set by command-line configuration) are extended, to assign a `DevViewActor` to each device. Moreover, the `onDeviceStarted` function is overridden, to specify the initial sensor values for every device.

```
object Demo5A_MainProgram extends Platform.CmdLineMain {
  // Indication of DevViewActor in Settings as a view of a device
  override def refineSettings(s: Platform.Settings): Platform.Settings = {
    s.copy(profile = s.profile.copy(
      devGuiActorProps = ref => Some(DevViewActor.props(Platform, ref))
    ))
  }
  // Assigning initial value to sensors of every device
  override def onDeviceStarted(dm: Platform.DeviceManager,
                               sys: Platform.SystemFacade): Unit = {
    val devInRow = DevViewActor.DevicesInRow
    val pos = Point2D(dm.selfId % devInRow, (dm.selfId / devInRow).floor)
    dm.addSensorValue(Platform.LocationSensorName, pos)
    dm.addSensorValue(Platform.SourceSensorName, dm.selfId==44)
    dm.start
  }
}
```

Lastly, the server-side entry point is expressed.

```
object Demo5A_ServerMain extends Platform.ServerCmdLineMain
```

An example of command-line configuration for this system is shown in the following frame.

```
Demo5A_ServerMain -h 127.0.0.1 -p 9000 --sched-global rr

Demo5A_MainProgram --program "demos.Demo5A_AggregateProgram" -P 9000 -p 9001
--sched-global rr -r 0to99 --gui
```

This configuration will create:

- a server on localhost, listening on port 9000, that performs a round-robin scheduling of the devices;
- 100 devices on localhost (port 9001), with a reference to the server (on port 9000), which executes the specified program (`Demo5A_AggregateProgram`); the platform will provide a view, to observe the devices and interact with them.

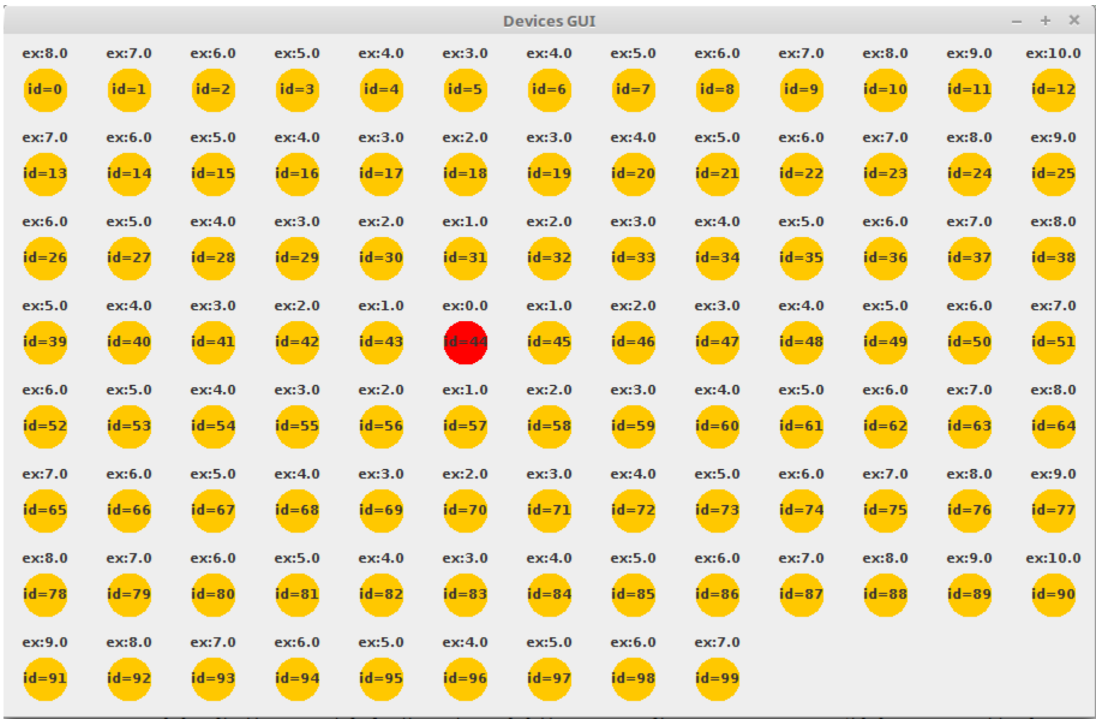

Figure 5.1: Graphical output of the spatial platform demo. Source nodes are highlighted in red. A simple hop gradient program is executed. The export of each node corresponds to the value of its distance from the source (device 44).

Figure 5.1 shows the output of the described demo on the view of the devices.

Code mobility

This demo enhance the previous one (spatial platform demo) with the introduction of the code mobility support. Through interaction with the view, a program switch can be triggered, causing an automatic update of the system. The demo is available in three different versions, each one based on a specific kind of actor platform. In this case, realization of a simple application, relying on the hybrid platform, is described. Command-line configuration is exploited to complete the definition of the relevant settings.

Common elements between all versions of this demo are collected in the definition of `Demo6_Platform`, a first refinement of the actor-based platform. This trait does not specify the actual kind of platform, allowing each version of the demo to combine it with the appropriate incarnation. A `Demo6DeviceActor` is defined, exploiting a code mobility approach based on the behavior expressed in `WeakCodeMobilityDeviceActor` (described in section 4.3). Functions of type `() => AggregateProgram` are declared, to provide different programs to every device, allowing a switch to one of them at runtime.

```
import it.unibo.scafi.distrib.actor.Platform

trait Demo6_Platform extends Platform with BasicAbstractActorIncarnation {
  val SourceSensorName: String = "source"

  trait Demo6DeviceActor extends WeakCodeMobilityDeviceActor {
    override def updateProgram(program: () => Any): Unit = program() match {
      case ap: AggregateProgram =>
        super.updateProgram(() => ap: ProgramContract)
    }
  }

  val NeighborsCountAggregateProgram = () => new AggregateProgram {
    override def main(): Int = foldhoodPlus(0)(_ + _)(1)
  }
  val BooleanGossipAggregateProgram = () => new AggregateProgram {
    override def main(): Boolean = rep(false)(x =>
      sense[Boolean](SourceSensorName) | foldhoodPlus(false)(_|_)(nbr(x)))
  }
  // ... Definition of other aggregate programs ...
}
```

The first step of programming consists in the selection of the hybrid platform.

```
import it.unibo.scafi.distrib.actor.hybrid.SpatialPlatform
import examples.gui.hybrid.DevViewActor
```

The spatial platform is enforced with the implementation of its abstract members. Moreover, the `actor.hybrid.SpatialDeviceActor` is enhanced with support for code mobility.

```
object Demo6C_Platform extends Demo6_Platform with SpatialPlatform {
  override val LocationSensorName: String = "LOCATION_SENSOR"
  override type P = Point2D
  override def buildNewSpace[E](elems: Iterable[(E,P)]): SPACE[E] =
    new Basic3DSpace(elems.toMap) {
      override val proximityThreshold = 1.1
  }

  class HybridDemo6DeviceActor(override val id: UID,
                               _aggEx: Option[ProgramContract],
                               _scope: ExecScope,
                               override val server: ActorRef)
  extends SpatialDeviceActor(id, _aggEx, _scope, server) with Demo6DeviceActor

  object HybridDemo6DeviceActor {
    def props(id: UID, prog: Option[ProgramContract],
              exScope: ExecScope, server: ActorRef): Props =
          Props(classOf[HybridDemo6DeviceActor], id, prog, exScope, server)
  }
}

import demos.{Demo6C_Platform => Platform}
```

A basic aggregate program is defined as the initial program executed by devices.

```
class Demo6C_AggregateProgram extends Platform.AggregateProgram {
  override def main(): String = "ready"
}
```

The definition of the main program is similar to that of the spatial platform demo, with a substantial difference: a custom device actor (`HybridDemo6DeviceActor`) will be used, instead of the default device actor, to represent a device inside the platform.

```
object Demo6C_MainProgram extends Platform.CmdLineMain {
  override def refineSettings(s: Platform.Settings): Platform.Settings = {
    s.copy(profile = s.profile.copy(
      // Indication of the actor to be used to represent a device
      devActorProps = (id, prog, scope, server) =>
        Some(Platform.HybridDemo6DeviceActor.props(id, progr, scope, server)),
      devGuiActorProps = ref => Some(DevViewActor.props(Platform, ref))
    ))
  }
  override def onDeviceStarted(dm: Platform.DeviceManager,
                               sys: Platform.SystemFacade): Unit = {
    val devInRow = DevViewActor.DevicesInRow
    val pos = Point2D(dm.selfId % devInRow, (dm.selfId / devInRow).floor)
    dm.addSensorValue(Platform.LocationSensorName, pos)
    dm.addSensorValue(Platform.SourceSensorName, dm.selfId==44)
    dm.start
  }
}
```

Lastly, the server-side entry point is expressed.

```
object Demo6C_ServerMain extends Platform.ServerCmdLineMain
```

The command-line configuration shown in the frame below produces a system analogous to the one of the spatial platform demo.

```
Demo6C_ServerMain -h 127.0.0.1 -p 9000

Demo6C_MainProgram --program "demos.Demo6C_AggregateProgram" -P 9000 -p 9001
-r 0to99 --gui
```

The output of this demo, in terms of representation of the network, is similar to the one shown in Figure 5.1. Initially, the export produced by each device will be a constant field, the string "ready". But, since it's possible to change the executed program, if the hop gradient program was assigned to the devices, you would get an output equivalent to the one of the spatial platform demo.

5.1.2 Platform monitoring

This demo creates a distributed system, relying on the p2p platform, and start monitoring it. Inside the `MonitoringDemo_Inputs` a programmatic configuration of the platform is in action. The concept of node is introduced, as a physical entity (identified by an address) in which one or more devices are located. The following code shows the process of defining the settings to allow execution on each node.

```
case class Node(host: String, port: Int, devices: Map[ID, (Point2D, Set[ID])])

import it.unibo.scafi.distrib.actor.p2p.SpatialPlatform
object MonitoringDemoPlatform extends SpatialPlatform
  with BasicAbstractActorIncarnation {
  override val LocationSensorName: LSensorName = "LocationSensor"
}
import sims.monitoring.{ MonitoringDemoPlatform => Platform }

object MonitoringDemo_Inputs {
  val nodes: List[Node] = List(
    Node("127.0.0.1", 9000, Map(1 -> (Point2D(50, 54), Set(2)))),
    Node("127.0.0.1", 9100, Map(2 -> (Point2D(50, 104), Set(3)))),
    // ... Definition of other nodes ...
  )

  val platformName: String = "MonitoringDemo"

  class MonitoringDemo_AggregateProgram extends AggregateProgram
    with SensorDefinitions with BlockG {
    def channel(source: Boolean, target: Boolean, width: Double): Boolean =
      distanceTo(source) + distanceTo(target) <=
        distanceBetween(source, target) + width

    override def main() = branch(sense3){false}{channel(sense1, sense2, 1)}
  }

  val aggregateAppSettings = Platform.AggregateApplicationSettings(
    name = platformName,
    program = () => Some(new MonitoringDemo_AggregateProgram())
  )

  val deploySys1 = Platform.DeploymentSettings(nodes(0).host, nodes(0).port)
  // ... Definition  of deployment settings for the other nodes ...

  val settings1: Platform.Settings =
    Platform.settingsFactory.defaultSettings().copy(
      aggregate = aggregateAppSettings,
      platform = Platform.PlatformSettings(
        subsystemDeployment = deploySys1,
        otherSubsystems = Set(Platform.SubsystemSettings(
          subsystemDeployment = deploySys2,
          ids = nodes(1).devices.keySet
        ))
      ),
      deviceConfig = Platform.DeviceConfigurationSettings(
        ids = nodes(0).devices.keySet,
        nbs = nodes(0).devices.map(d => d._1 -> d._2._2))
      )
  // ... Definition of settings for the other nodes ...

  class MonitoringDemoMain(override val settings: Platform.Settings)
    extends Platform.BasicMain(settings) {

    override def onDeviceStarted(dm: Platform.DeviceManager,
                                 sys: Platform.SystemFacade): Unit = {
      val pos = nodes.filter(n => n.devices.contains(dm.selfId)).head
        .devices(dm.selfId)._1
      dm.addSensorValue(Platform.LocationSensorName, pos)
      dm.addSensorValue(SensorName.sensor1, false)
      // ... Set of the other sensors (sensor2, sensor3) to false ...
      dm.start
    }
  }
}
```

The goal of the implemented program, known as `Channel`, is to create the shortest path from a source (devices that retrieves true from `sensor1`) to a destination (devices that retrieves true from `sensor2`), avoiding obstacles (devices that retrieves true from `sensor3`).

After the process of settings definition, an application launcher is set for each node.

```
import MonitoringDemo_Inputs._

object MonitoringDemo_MainProgram_1 extends MonitoringDemoMain(settings1)
object MonitoringDemo_MainProgram_2 extends MonitoringDemoMain(settings2)
// ... Application launcher for the other nodes ...
```

Finally, the monitoring view is created and attached to the platform. The `MonitoringDemo_Monitor` is the entry point of the monitoring application. Further information on how to run the view can be found in [16].

```
import MonitoringDemo_Inputs._

object MonitoringDemo_Monitor extends App {
  val initializer = RadiusSimulation(
    radius = 5,
    platformName = platformName,
    platformNodes = nodes.flatMap(
      n => n.devices.map(_._1 -> (n.host, n.port))
    ).toMap,
    platform = Platform
  )

  ScafiProgramBuilder (
    worldInitializer = Fixed(
      nodes.flatMap(n => n.devices.map(d => d._1 -> d._2._1)).toSet
    ),
    scafiSimulationInfo = SimulationInfo(
      program = classOf[MonitoringDemo_AggregateProgram]
    ),
    simulationInitializer = initializer,
    outputPolicy = GradientFXOutput
  ).launch()
}
```

An execution schema for this demo is:

1. Execution of the actual system, by running all the distinct subsystems through the corresponding launchers (`MonitoringDemo_MainProgram_`*x*).
2. Execution of the monitoring system, by running the appropriate launcher (`MonitoringDemo_Monitor`).
3. Monitoring and control of the system.

Figure 5.2 shows the output of the described demo.

5.2 Requirements verification

Considering the code produced in the implementation phase and the results obtained in the demos, an evaluation of each requirement expressed in section 2.1 has to be carried out.

- **Serialization**: a serialization strategy has been successfully implemented in `scafi`. The adoption of the JSON format promote interoperability in the complex context of the Internet of Things. The modular structure of the serializer, described in section 4.2, ensures extendibility; in fact, the

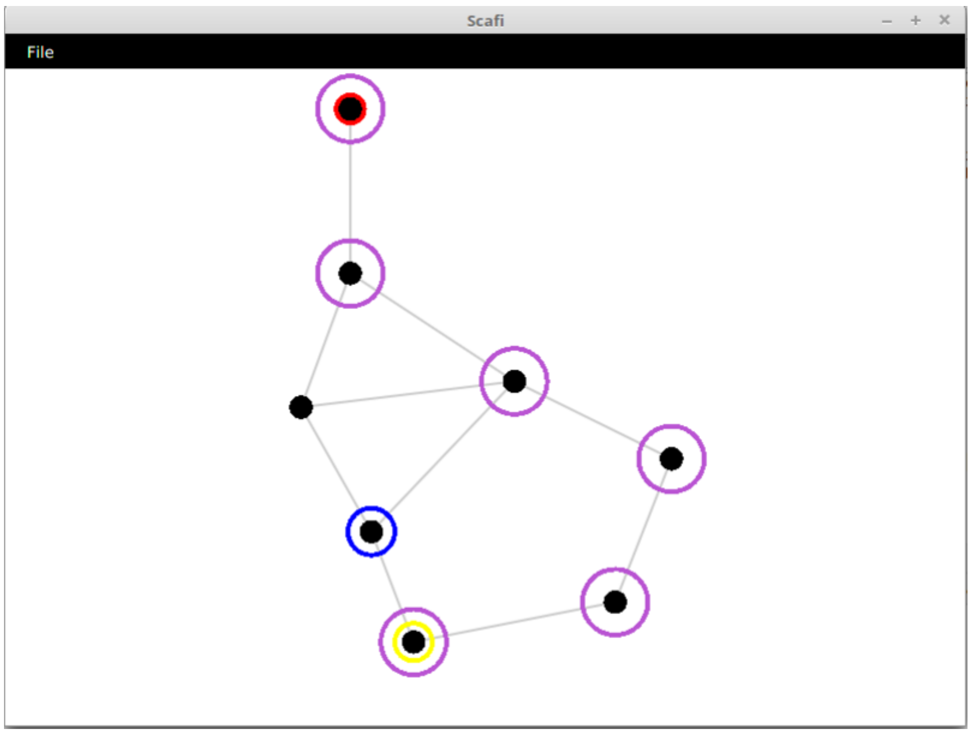

Figure 5.2: Graphical output of the monitoring demo. Graphically, the source is identified by a red circle, the destination by a yellow circle, and obstacles by a blue circle. Each node that is on the shortest path between source and destination is highlighted.

problem of serialization of a new message can only be addressed by defining conversions (from message to JSON value and vice versa) inside the `JsonMessagesSerialization` trait.

- **Hybrid platform**: The implementation of the hybrid platform fulfill its main requirement, since the demos proves that the three platforms are interchangeable. Hence, replacing a server-based or p2p platform with a hybrid platform is a zero-cost operation, that guarantees the same results with a different architecture.
- **Spatial platform**: In the demos related to spatial nets (`Demo5` and `Demo6`), the spatial platform is exploited to create spatial systems, in which devices are situated in an environment. A device can move in the space and retrieve

its position from an appropriate location sensor. In the demos, movement is simulated assigning different values to the mentioned sensor; an execution with actual sensors in the real world could allow a more effective testing of the developed systems.

- **Code mobility**: the ultimate goal of code mobility is to properly exchange programs (code) between devices. The `Demo6` proves the accomplishment of this requirement, allowing the user to change the program executed by the devices at runtime. As described in section 4.3, this result can be easily achieved by using `WeakCodeMobilityDeviceActor` instead of the default `ComputationDeviceActor`. A message (of type `MsgUpdateProgram`) sent to any device triggers the program switch across the platform.
- **Platform/view integration**: the demo related to platform monitoring confirm the integration, since the view displays the state of a running platform (monitoring task). To fulfill the requirements, the monitoring system lets the users directly interact with devices of the platform, and allows them to simulate sensor values at will (control task).

Chapter 6

Conclusion

In this thesis, the problem of distributing aggregate computations has been addressed. Several options have been presented and analyzed to increase flexibility in the development of distributed systems. The most promising ones have been implemented in the preexisting actor-based platform of `scafi`, exploiting the advanced features provided by both the Scala programming language and the Akka framework. This work based on distribution accomplished the following main results:

- support communication between nodes distributed in the network;
- design of systems with a hybrid peer-to-peer architecture model;
- change in the behavior of aggregate systems at runtime, by shipping new code to devices;
- monitoring and control of an aggregate system at execution time.

In the process, different demos have been produced, to verify the correctness of each functionality introduced in `scafi`, and to show the programming of distributed aggregate application with the framework.

6.1 Future work

The path related to consolidation of distributed computing in `scafi`, undertaken with this thesis, can not be considered complete. In particular, some interesting future directions are presented.

- **IoT deployment**: an actual deployment of systems produced with `scafi` would be a great opportunity for testing aggregate applications in action. The platform could be run on mobile devices (like smartphones), equipped with physical sensors and actuators. Moreover, a server-based architecture could be extended, exploiting services offered by cloud computing.
- **Communication on top of the actor platform**: the current Akka-based communication between devices is not suitable for a distributed application in the context of the Internet of Things. As stated in subsection 2.3.2, application-level protocols and communication technologies could be built on top of the existing platform, with moderate efforts. As proof of concept, a demo could be realized, showing the design and implementation of a system distributed on the Internet (maybe exploiting the HTTP protocol).
- **Full support for code mobility**: the approach to code mobility could definitely be improved. In subsection 2.2.4, a distinction is delineated between weak code mobility and strong code mobility. The latter is not currently supported by the platform, but it would bring interesting implications, allowing for unknown code injection at runtime.
- **Simulation on platform**: the key concepts explained in subsection 2.3.3 should be carefully explored, in order to design the most suitable implementation of a simulation based on the actor platform. In particular, defining the connection between the already implemented task of platform monitoring and the actor-based simulation is crucial.

Bibliography

[1] Roberto Casadei. “Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields”. Master’s thesis. University of Bologna, 2016. URL: https://amslaurea.unibo.it/10341/.

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

[3] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. “A type-sound calculus of computational fields”. In: *Science of Computer Programming* 117 (2016), pp. 17–44.

[4] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. “Code mobility meets self-organisation: A higher-order calculus of computational fields”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer. 2015, pp. 113–128.

[5] Jacob Beal and Mirko Viroli. “Building blocks for aggregate programming of self-organising applications”. In: *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2014 IEEE Eighth International Conference on*. IEEE. 2014, pp. 8–13.

[6] Roberto Casadei and Mirko Viroli. “Towards aggregate programming in scala”. In: *First Workshop on Programming Models and Languages for Distributed Computing*. ACM. 2016, p. 5.

[7] Roberto Casadei and Mirko Viroli. “Programming actor-based collective adaptive systems”. In: *Programming with Actors*. Springer, 2018, pp. 94–122.

[8] 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*. ACM. 2016, pp. 1321–1326.

[9] Roberto Casadei, Danilo Pianini, and Mirko Viroli. "Simulating large-scale aggregate MASs with alchemist and scala". In: *Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on*. IEEE. 2016, pp. 1495–1504.

[10] Danilo Pianini, Sara Montagna, and Mirko Viroli. "Chemical-oriented simulation of computational systems with Alchemist". In: *Journal of Simulation* 7.3 (2013), pp. 202–215.

[11] Rüdiger Schollmeier. "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications". In: *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*. IEEE. 2001, pp. 101–102.

[12] B Pourebrahimi, K Bertels, and S Vassiliadis. "A survey of peer-to-peer networks". In: *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc*. Vol. 2005. Citeseer. 2005.

[13] *The Akka documentation*. Version 2.5. URL: https://doc.akka.io/docs/akka/2.5/scala/ (visited on 11/08/2018).

[14] Beverly Yang and Hector Garcia-Molina. "Comparing hybrid peer-to-peer systems". In: *Proceedings of the 27th Intl. Conf. on Very Large Data Bases*. 2001.

[15] Richard Fujimoto. "Parallel and distributed simulation". In: *Winter Simulation Conference (WSC), 2015*. IEEE. 2015, pp. 45–59.

[16] Gianluca Aguzzi. "Sviluppo di un front-end di simulazione per applicazioni aggregate nel framework Scafi". Bachelor's thesis. University of Bologna, 2018. URL: https://amslaurea.unibo.it/16824/.